V$_R$4000 μPD30400/401/402

V$_R$3600 μPD30360

User's Manual

Contents User's Manual VR4000

Page

Chapter 1 Introduction 1-1- 1

1.1 Data Formats and Addressing 1-1- 1
1.2 General Registers 1-1- 4
1.3 Special Registers 1-1- 5
1.4 Coprocessor Registers 1-1- 5
1.5 Restrictions 1-1- 5
1.6 Changes in this Edition 1-1- 5

Chapter 2 Instruction Set 1-2- 1

2.1 Instruction Formats and Notational Conventions 1-2- 1
2.2 Memory Access Specification 1-2- 4
2.3 Instruction Fetch and Decode 1-2- 6
2.4 Instruction Classes 1-2- 6
2.5 Load and Store Instructions 1-2- 8
2.6 Computational Instructions 1-2- 44
2.6.1 ALU Immediate Instructions 1-2- 44
2.6.2 ALU 3-Operand Register-Type Instructions 1-2- 55
2.6.3 Shift Instructions 1-2- 70
2.6.4 Multiply/Divide Instructions 1-2- 86
2.7 Jump and Branch Instructions 1-2- 99
2.7.1 Jump Instructions 1-2- 99
2.7.2 Branch on Condition 1-2-105
2.8 Exception Instructions 1-2-123
2.9 Coprocessor Instructions 1-2-140
2.9.1 Coprocessor Unit and Implementation code Assignments 1-2-140
2.9.2 Coprocessor Load and Store 1-2-141
2.9.3 Move To/From Coprocessor 1-2-146
2.9.4 Coprocessor Operations 1-2-151
2.9.5 Branch on Coprocessor Condition 1-2-153

Chapter 3 Floating Point Coprocessor 1-3- 1

3.1 Floating-point coprocessor architecture 1-3- 1
3.1.1 Instruction format 1-3- 1
3.1.1.1 Floating-point loads, stores, and moves 1-3- 1
3.1.1.2 Floating-point operations 1-3- 2
3.1.1.3 Floating-point comparisons 1-3- 2
3.1.2 Coprocessor registers 1-3- 3
3.1.2.1 MIPS I and II Register layout — processor viewpoint 1-3- 5
3.1.2.2 MIPS I and II Register format — coprocessor viewpoint 1-3- 6
3.1.2.3 MIPS III Register format — coprocessor viewpoint 1-3- 6
3.1.2.3.1 Floating-point formats 1-3- 7
3.1.2.3.2 Binary fixed-point format 1-3- 10
3.1.2.3.3 Implementation and revision register 1-3- 10
3.1.2.3.4 Exception instruction register (optional) 1-3- 11
3.1.2.3.5 Control and status register 1-3- 12
3.1.3 Save and restore state 1-3- 14

Contents

		Page
3.1.4	Exception trap processing	1-3-15
3.1.4.1	Invalid operation exception	1-3-15
3.1.4.2	Division-by-zero exception	1-3-16
3.1.4.3	Overflow exception	1-3-16
3.1.4.4	Underflow exception	1-3-16
3.1.4.5	Inexact exception	1-3-17
3.1.4.6	Unimplemented operation exception	1-3-17
3.1.4.7	Trap handlers for the five IEEE Standard 754 exceptions	1-3-17
3.2	Floating-point instruction set	1-3-19
3.2.1	Instruction Formats and Notational Conventions	1-3-19
3.2.2	Load and Store to/from Floating-point coprocessor	1-3-24
3.2.3	Move To/From Floating-point Coprocessor	1-3-32
3.2.4	Move To/From Floating-point Coprocessor Control Registers	1-3-37
3.2.5	Computational Instructions	1-3-40
3.2.6	Branch on Floating-point Coprocessor Condition	1-3-67
3.3	Compatibility with the IEEE 754 floating-point standard	1-3-73
3.3.1	Interpretation of the standard	1-3-73
3.3.2	Software assistance for IEEE 754 standard compatibility	1-3-73
3.3.3	IEEE 754 exception trapping	1-3-74
3.3.4	IEEE 754 format compatibility	1-3-74
3.3.5	Compatibility of commercially available devices	1-3-74
3.4	Floating-point coprocessor implementation	1-3-77
3.4.1	Software implementation	1-3-77
3.4.2	Hardware implementations	1-3-77
3.4.2.1	Optional implementation of operations, formats and exceptions	1-3-77
3.4.2.2	Pipelining and concurrent execution	1-3-77
3.4.2.2.1	Precise exceptions traps	1-3-78
3.4.2.2.2	Imprecise Exception Traps	1-3-78
3.4.2.3	Simplifying exception handling	1-3-79
3.4.3	Standard coprocessor exception trap handler	1-3-79
3.4.4	Operation timing	1-3-79
3.4.4.1	Pipelining and overlapping	1-3-81
3.4.5	Software implementation of IEEE 754 standard operations	1-3-81
3.4.5.1	Remainder	1-3-81
3.4.5.2	Round to integer	1-3-82
3.4.5.3	Convert between binary and decimal	1-3-82
3.4.5.4	Copy sign	1-3-82
3.4.5.5	Scale binary	1-3-83
3.4.5.6	Log binary	1-3-83
3.4.5.7	Next after	1-3-83
3.4.5.8	Finite	1-3-83
3.4.5.9	Is NaN	1-3-83
3.4.5.10	Arithmetic inequality	1-3-83
3.4.5.11	Class	1-3-83
3.4.6	Software implementation of IEEE 754 standard trap handlers	1-3-83
3.4.7	Implementation in single-chip VLSI	1-3-84
3.4.7.1	Unimplemented instruction	1-3-84
3.4.7.1.1	Extended or quad-precision	1-3-84
3.4.7.1.2	Square root	1-3-84
3.4.7.1.3	Denormalized operand	1-3-84
3.4.7.1.4	Quiet Not a Number operand	1-3-84

Contents

Page

3.4.7.2 Invalid operation exception 1-3-85
3.4.7.3 Division-by-zero exception 1-3-85
3.4.7.4 Overflow 1-3-85
3.4.7.5 Underflow exception 1-3-85
3.4.7.5.1 Denormalized result 1-3-85
3.4.7.6 Inexact exception 1-3-85
3.4.8 Implementation using commercially available floating-point devices 1-3-85
3.4.8.1 Invalid operation exception 1-3-86
3.4.8.2 Division-by-zero exception 1-3-86
3.4.8.3 Overflow exception 1-3-86
3.4.8.4 Underflow exception 1-3-87
3.4.8.5 Inexact exception 1-3-87
3.4.8.6 Unimplemented operation exception 1-3-87
3.4.8.6.1 extended- and quad-precision 1-3-87
3.4.8.6.2 square root 1-3-87
3.4.8.6.3 denormalized operand 1-3-87
3.4.8.6.4 Not a Number operand 1-3-88
3.4.8.6.5 Denormalized result 1-3-88

Chapter 4 System Control Coprocessor 1-4- 1
4.1 System Control Coprocessor Compatibility 1-4- 1
4.2 MIPS III 1-4- 1
4.3 Memory System Architecture 1-4- 1
4.3.1 MIPS I and II User-mode Virtual Addressing 1-4- 3
4.3.2 R2000, R3000, R6000 Kernel-mode Virtual Addressing 1-4- 4
4.3.3 MIPS II R4000 Supervisor-mode Virtual Addressing 1-4- 5
4.3.4 MIPS II R4000 Kernel-mode Virtual Addressing 1-4- 6
4.3.5 MIPS III User-mode Virtual Addressing 1-4- 8
4.3.6 MIPS III Supervisor-mode Virtual Addressing 1-4- 9
4.3.7 MIPS III R4000 Kernel-mode Virtual Addressing 1-4-10
4.3.8 Translation Lookaside Buffer 1-4-14
4.3.8.1 On-chip TLB 1-4-14
4.3.8.2 In-cache TLB 1-4-17
4.3.9 Cache Memory 1-4-18
4.3.9.1 R2000 caches 1-4-19
4.3.9.2 R3000 caches 1-4-19
4.3.9.3 R4000 primary instruction cache 1-4-20
4.3.9.4 R4000 primary data cache 1-4-21
4.3.9.5 R4000 secondary cache 1-4-23
4.3.9.6 R6000 cache 1-4-24
4.3.10 Functional Operation of Memory System 1-4-26
4.3.10.1 R2000 processor 1-4-26
4.3.10.2 R3000 processor 1-4-28
4.3.10.3 R4000 processor 1-4-29
4.3.10.4 R6000 processor 1-4-29
4.3.11 Transparent implementation features of the cache and virtual memory system 1-4-29
4.3.11.1 R2000 1-4-29
4.3.11.2 R3000 1-4-30
4.3.11.3 R4000 1-4-30
4.3.11.4 R6000 1-4-30

Contents

Page

4.4 System Control Coprocessor Registers 1-4-31
4.4.1 EntryHi 1-4-33
4.4.2 PageMask 1-4-34
4.4.3 EntryLo 1-4-34
4.4.4 TLB Index 1-4-36
4.4.5 TLB Wired 1-4-36
4.4.6 TLB Random 1-4-37
4.4.7 Bad Virtual Address 1-4-38
4.4.8 Context 1-4-38
4.4.9 XContext 1-4-39
4.4.10 Status 1-4-40
4.4.10.1 Diagnostic Status 1-4-43
4.4.10.1.1 R2000, R3000 1-4-43
4.4.10.1.2 R4000 1-4-44
4.4.10.1.3 R6000 1-4-45
4.4.11 Cause 1-4-46
4.4.12 Error 1-4-48
4.4.13 EPC 1-4-49
4.4.14 ErrorEPC 1-4-50
4.4.15 CacheErr 1-4-51
4.4.16 TagLo/TagHi 1-4-52
4.4.17 ECC 1-4-54
4.4.18 Processor Revision Identifier 1-4-55
4.4.19 Count 1-4-55
4.4.20 Compare 1-4-55
4.4.21 LLAddr 1-4-56
4.4.22 WatchLo and WatchHi 1-4-56
4.4.23 Config 1-4-57
4.5 System Control Coprocessor Instructions 1-4-59
4.5.1 Move To/From System Control Coprocessor 1-4-59
4.5.2 System Control Coprocessor Operations 1-4-64

Chapter 5 Exception Handling 1-5- 1

5.1 Exception operation 1-5- 1
5.2 Precision of Exceptions 1-5- 3
5.3 Exception Types 1-5- 4
5.4 In-cache TLB Exceptions 1-5- 5
5.5 Exception vectors 1-5- 5
5.6 Priority of Exceptions 1-5- 6
5.6.1 Reset 1-5- 6
5.6.2 Soft Reset 1-5- 7
5.6.3 Non-maskable Interrupt 1-5- 7
5.6.4 TLB Refill and Extended addressing TLB Refill 1-5- 8
5.6.5 TLB Invalid 1-5- 9
5.6.6 TLB Modified 1-5- 9
5.6.7 Bus Error 1-5-10
5.6.8 Address Error 1-5-10
5.6.9 Integer overflow 1-5-11
5.6.10 Trap (MIPS II and III only) 1-5-11
5.6.11 System Call 1-5-11

Contents

		Page
5.6.12	Breakpoint	1-5-12
5.6.13	Reserved Instruction	1-5-12
5.6.14	Coprocessor Unusable	1-5-13
5.6.15	Interrupt	1-5-13
5.6.16	Machine Check (R6000 only)	1-5-14
5.6.17	Uncached LDC_Z/SDC_Z (R6000 only)	1-5-14
5.6.18	Virtual Coherency (R4000 only)	1-5-14
5.6.19	Cache Error (R4000 only)	1-5-15
5.6.20	Watch (R4000 only)	1-5-15
5.6.21	Floating Point (R4000 only)	1-5-15
Chapter 6	**Hazards and Interlocks**	1-6- 1
6.1	Introduction to pipeline hazards	1-6- 1
6.2	Introduction to restartability hazards	1-6- 2
6.3	Hazards allowed by the R-series architecture	1-6- 3
6.3.1	Load delay slot	1-6- 3
6.3.2	Branch delay slot	1-6- 3
6.3.3	Setting up a coprocessor condition	1-6- 4
6.3.4	No bypassing for HI and LO registers	1-6- 4
6.3.5	Combinations of scheduling hazards	1-6- 4
6.3.6	Additional requirements for R2360 floating-point board	1-6- 4
6.4	R2000 and R3000 Pipeline	1-6- 4
6.5	R6000 Pipeline	1-6- 6
6.6	R4000 Pipeline	1-6- 8
6.7	Coprocessor 0 Hazards	1-6-10
6.7.1	R3000 coprocessor 0 Hazards	1-6-11
6.7.2	R6000 memory management hazards	1-6-11
6.7.3	R4000 coprocessor 0 hazards	1-6-12
Chapter 7	**Instruction Encoding**	1-7- 1
7.1	MIPS I and MIPS II opcode assignments	1-7- 2
7.2	MIPS III opcode assignments	1-7- 4
Chapter 8	**Floating Point Instruction Encoding**	1-8- 1
8.1	MIPS I and MIPS II opcode assignments	1-8- 2
8.2	MIPS III opcode assignments	1-8- 3
Index		1-Index-1

Contents User's Manual V_R3600

Page

Part 1 **Overview**

Chapter 1 **Overview** 2-1-1- 1

1.1 V_R3600 Features 2-1-1- 1
1.2 Functional Description 2-1-1- 3
1.3 Mode Selection and Pin Condition 2-1-1- 9
1.4 Internal Block Diagram 2-1-1-10
1.5 System Configuration 2-1-1-12

Part 2 **CPU Architekture**

Chapter 1 **Overview CPU Architecture** 2-2-1- 1

1.1 What is RISC? 2-2-1- 1
1.2 Defining Performance 2-2-1- 3
1.2.1 Time per instruction 2-2-1- 3
1.2.2 Cycles per instruction (C) 2-2-1- 5
1.2.3 Time per cycle (T) 2-2-1-14
1.2.4 Instruction per task (I) 2-2-1-18
1.3 Hidden Benefits of RISC Design 2-2-1-25
1.4 Rules of Description 2-2-1-27
1.4.1 Representation of numeric values 2-2-1-27
1.4.2 Representation of data 2-2-1-27
1.4.3 Representation of memory 2-2-1-30
1.4.4 Terms 2-2-1-30

Chapter 2 **V_R3600 CPU Architecture** 2-2-2- 1

2.1 V_R3600 Processor Features 2-2-2- 2
2.2 V_R3600 CPU Registers 2-2-2- 3
2.3 Instruction Set Overview 2-2-2- 4
2.4 V_R3600 Processor Programming Model 2-2-2- 9
2.4.1 Data formats and addressing 2-2-2- 9
2.4.2 V_R3600 CPU general register 2-2-2-11
2.5 V_R3600 System Control Coprocessor (CPO) 2-2-2-13
2.6 V_R3600 Pipeline Architecture 2-2-2-14
2.7 Memory Management System 2-2-2-16
2.7.1 The TLB (Translation Lookaside Buffer) 2-2-2-16
2.7.2 V_R3600 operating modes 2-2-2-17
2.8 Memory System Hierarchy 2-2-2-19

Chapter 3 **V_R600 Instruction Set Summary** 2-2-3- 1

3.1 Instruction Formats 2-2-3- 1
3.2 Instruction Notational Conventions 2-2-3- 3
3.3 Load and Store Instructions 2-2-3- 4
3.4 Computational Instructions 2-2-3- 8
3.5 Jump and Branch Instructions 2-2-3-14
3.6 Special Instructions 2-2-3-18
3.7 Coprocessor Instructions 2-2-3-19
3.8 System Control Coprocessor (CPO) Instruction 2-2-3-21

Contents

Page

Chapter 4 V_R3600 Instruction Pipeline 2-2-4- 1
4.1 Pipeline Processing 2-2-4- 1
4.2 The Delayed Instruction Slot 2-2-4- 3

Chapter 5 Memory Management System 2-2-5- 1
5.1 Memory System Architecture 2-2-5- 1
5.1.1 Privilege states 2-2-5- 2
5.1.2 Unser-mode virtual addressing 2-2-5- 4
5.1.3 Kernel-mode virtual addressing 2-2-5- 4
5.2 Virtual Memory and the TLB 2-2-5- 6
5.2.1 TLB entries 2-2-5- 7
5.2.2 EntryHI & EntryLO registers 2-2-5- 7
5.2.3 Virtual address translation 2-2-5- 8
5.2.4 The index register 2-2-5-11
5.2.5 The random register 2-2-5-11
5.2.6 TLB instructions 2-2-5-12

Chapter 6 Exception Processing 2-2-6- 1
6.1 The Exception Handling List 2-2-6- 2
6.2 The Exception Handling Register 2-2-6- 3
6.2.1 The cause register 2-2-6- 4
6.2.2 The EPC (Exception Program Counter) register 2-2-6- 6
6.2.3 The status register 2-2-6- 6
6.2.4 Status register mode bits and exception processing 2-2-6-11
6.2.5 BadVAddr register 2-2-6-12
6.2.6 Context register 2-2-6-12
6.2.7 Processor revision identifier register 2-2-6-13
6.3 Exception Description Details 2-2-6-15
6.3.1 Address error exception 2-2-6-16
6.3.2 Breakpoint exception 2-2-6-17
6.3.3 Bus error exception 2-2-6-17
6.3.4 Coprocessor unusable exception 2-2-6-19
6.3.5 Interrupt exception 2-2-6-20
6.3.6 Overflow exception 2-2-6-21
6.3.7 Reserved instruction exception 2-2-6-22
6.3.8 Reset exception 2-2-6-23
6.3.9 System call exception 2-2-6-24
6.3.10 TLB miss exceptions 2-2-6-25

Chapter 7 Instruction Set Details 2-2-7- 1
7.1 Instruction Classes 2-2-7- 2
7.2 Instruction Formats 2-2-7- 3
7.3 Instruction Notation Conventions 2-2-7- 4
7.4 Instruction Class Summary 2-2-7- 7
7.4.1 Load and store instructions 2-2-7- 7
7.4.2 Jump and branch instructions 2-2-7- 9
7.4.3 Coprocessor instructions 2-2-7-10
7.4.4 System control coprocessor (CPO) instructions 2-2-7-11
7.5 Instructions 2-2-7-12

Contents

Page

Part 3 FPU Architecture

Chapter 1 V_RFPU Overview 2-3-1- 1

1.1 V_R3600 FPU Features 2-3-1- 2
1.2 V_R3600 FPU Programming Model 2-3-1- 3
1.3 Floating-Point Formats 2-3-1-12
1.4 Number Definitions 2-3-1-14
1.5 Coprocessor Operation 2-3-1-16
1.6 Instruction Set overview 2-3-1-18
1.7 Pipeline Architecture 2-3-1-20

Chapter 2 Instruction Set Summary & Instruction Pipeline .. 2-3-2- 1

2.1 Instruction Set Summary 2-3-2- 1
2.2 The Instruction Pipeline 2-3-2-10
2.2.1 The pipeline processing 2-3-2-10
2.2.2 Instruction execution times 2-3-2-13
2.2.3 Overlapping FPU instructions 2-3-2-15

Chapter 3 Floating Point Exceptions 2-3-3- 1

3.1 Exception Trap Processing 2-3-3- 3
3.2 Saving and Restoring State 2-3-3-10

Chapter 4 Instruction Set Details 2-3-4- 1

4.1 Instruction Set Summary 2-3-4- 1
4.1.1 Instruction formats 2-3-4- 1
4.1.2 Instruction nottional conventions 2-3-4- 1
4.1.3 Load and store instructions 2-3-4- 4
4.1.4 Computational instructions 2-3-4- 6
4.1.6 Move instructions 2-3-4- 9
4.1.6 Branch instructions 2-3-4-10
4.2 Instruction Details 2-3-4-11

Appendix A V_R3600 CPU Instruction Opcode Bit Encoding 2-A-1

Appendix B V_R3600 FPU Instruction Opcode Bit Encoding 2-B-1

MIPS

NEC V_R4000

User's Manual

1. Introduction

The MIPS R-series processor architecture is designed to be an efficient interface between three levels of technology: it is a good target for MIPS optimizing compilers; it enables a clean implementation of general-purpose operating systems, such as UNIX; and it supports RISC machine organizations that are well-balanced at the current state of the art in semiconductor technologies.

As all three of these technologies advance, however, the architecture evolves in response to a shifting compromise between software and hardware resources in the system. This evolution maintains object-code compatibility for programs that execute in user-mode. The first MIPS processors, the R2000 and R3000, implement the MIPS I Instruction Set Architecture (ISA) for user-mode programs. User-mode programs that conform to the MIPS I ISA will execute on any MIPS architecture implementation. Subsequent MIPS processors, such as the R6000 and R4000, implement the MIPS II ISA. The MIPS II ISA is a super-set of MIPS I, and so these implementations execute MIPS I programs directly.

The MIPS I and MIPS II architectures specify only user-mode operation; the operation of the system coprocessor is implementation specific. For example, R2000 kernel-mode operation is defined by the R2000 architecture, not by MIPS I. Nonetheless, the system coprocessors for all present R-series processors are sufficiently similar that they are documented together in Chapter 4.

The architecture of systems based on R-series processors is not only implementation specific, but outside the scope of this specification. This includes the operation of I/O, DMA, peripherals, and operating systems. The operation of memory and caches not directly controlled by R-series processors is not subject to this specification. MIPS I also does not define the operation of processors in a multiprocessor environment. MIPS II specifies some limited semantics for multiprocessors.

All MIPS I and MIPS II instructions and registers described in this document, are implemented in at least one MIPS R-series processor, except for features explicitly described as future extensions.

One such future extension of the instruction set is planned (MIPS III). The purpose of MIPS III is to provide a 64-bit instruction set architecture. It is a super-set of the MIPS II ISA, so MIPS I and II user-mode programs can be directly executed on MIPS III implementations. In addition, MIPS III processors will be able to operate on 64-bit data and addresses. The additional instructions and the larger virtual address space can be enabled or not for user mode.

Caveat: Descriptions of R4000 processors in this manual are not necessarily up to date with the design in progress.

The basic instruction set is described in chapter 2 of this manual. Included in the basic instruction set is a general schema for extending the architecture by the addition of "coprocessors." The architecture currently defines two of the four coprocessor slots, one for floating-point arithmetic (described in chapter 3), and another for memory management (the "system coprocessor", described in in chapter 4). The remaining coprocessor slots may be defined in future revisions of the MIPS architecture. The term "coprocessor" does not necessarily imply a separate physical implementation (the system coprocessor is implemented on the cpu chip in all current implementations, for example), but rather that the degree of interaction is limited. Processor implementations are may support external coprocessors, but this is not required.

The MIPS III instruction set no longer supports coprocessor 3. Coprocessor 3 opcodes have been used to implement 64-bit operations in the processor and floating-point coprocessor.

1.1. Data Formats and Addressing

The MIPS machine defines an 8-bit **byte,** a 16-bit **halfword,** a 32-bit **word,** and a 64-bit **doubleword.**

The byte ordering is configurable into either big-endian or little-endian byte ordering. When configured as a big-endian system, byte 0 is always the most significant (leftmost) byte; thereby effecting compatibility with

MC 68000† conventions. When configured as a little-endian system, byte 0 is always the least significant (rightmost) byte, which is compatible with iAPX x86, NS 32000, and DEC VAX conventions.

Within this specification, bit 0 is always the least significant (rightmost) bit; thus bit designations are always little-endian. It should be noted, however, that no instructions explicitly designate bit positions within words. Some high-level languages may define different bit designations than are described here; in particular, many languages may define bit 0 to be the most significant (leftmost) bit of a data item, on a big-endian system configuration.

The diagrams below show the ordering of bytes within words and the ordering of words (in decreasing order of significance) within multiple-word structures for each of the two conventions:

Big-endian byte ordering

31 — 24	23 — 16	15 — 8	7 — 0
0	1	2	3
4	5	6	7
8	9	10	11
8	8	8	8

Little-endian byte ordering

31 — 24	23 — 16	15 — 8	7 — 0
11	10	9	8
7	6	5	4
3	2	1	0
8	8	8	8

The machine uses byte addressing, with alignment constraints, for halfword and word accesses; halfword accesses must be aligned on an even byte boundary; word accesses must be aligned on a byte boundary divisible by four; and doubleword accesses must be aligned on a byte boundary divisible by eight.

As shown in the diagrams below, the address of a multiple-byte data item is the address of the most-significant byte on a big-endian configuration, and is the address of the least-significant byte on a little-endian configuration.

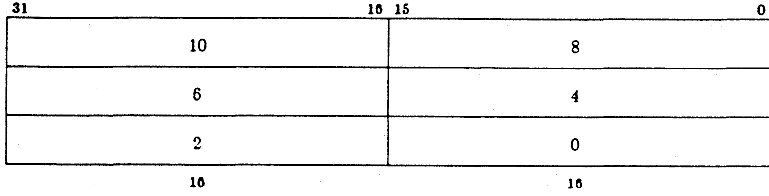
† MC 68000, iAPX x86, NS 32000, DEC and VAX are trademarks of Motorola, Intel, National, Digital and Digital respectively.

The diagrams below show the ordering and addresses of halfwords for each of the two conventions:

Halfwords: Big-endian byte ordering

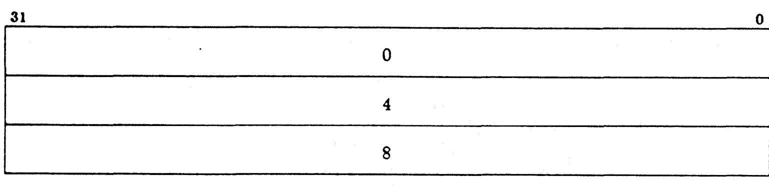
31 — 16	15 — 0
0	2
4	6
8	10
16	16

Halfwords: Little-endian byte ordering

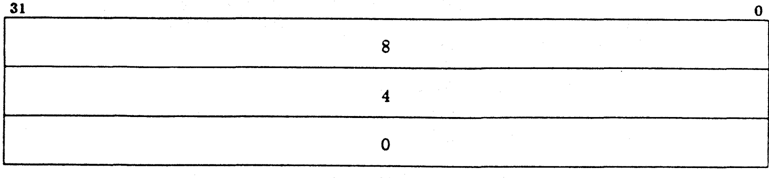


The diagrams below show the ordering and addresses of words (in decreasing order of significance) for each of the two conventions:

Words: Big-endian byte ordering



Words: Little-endian byte ordering



Special instructions are provided for addressing words which are not aligned on 4-byte boundaries (Load/Store-Word-Left/Right; LWL, LWR, SWL, SWR). These instructions are used in pairs to provide addressing of misaligned words with one additional instruction cycle over that required for aligned words.

In the diagrams below, the bytes accessed when addressing a misaligned word with a byte address of 3 are shown for each of the two conventions.

Misaligned word: Big-endian byte ordering

31 ... 24	23 ... 16	15 ... 8	7 ... 0
-	-	-	3
4	5	6	-
8	8	8	8

Misaligned word: Little-endian byte ordering

31 ... 24	23 ... 16	15 ... 8	7 ... 0
-	6	5	4
3	-	-	-
8	8	8	8

1.2. General Registers

There are 32 general registers. On R2000, R3000, and R6000 processors, each register is a single word (32 bits). On R4000 processors, each register is a doubleword (64 bits). The 32 general registers are all equivalent, with two exceptions: r0 is hardwired to a zero value, and r31 is the link register for jump and link instructions.

Register r0 may be specified as a target register for any instruction for which the result of the operation need not be placed in any general register. The register maintains a value of zero when used as a source register under all conditions.

Registers are assigned uses and special names by convention in the assembler and compiler, as shown in the table below:

Register number	Symbolic name	use
r0	zero	read-only zero value
r1	at	assembler temporary
r2..r3	v0..v1	integer function value
r4..r7	a0..a3	parameters
r8..r15	t0..t7	general use, not preserved by subroutines
r16..r23	s0..s7	general use, preserved by subroutines
r24..r25	t8..t9	general use, not preserved by subroutines
r26..r27	k0..k1	kernel
r28	gp	global pointer
r29	sp	stack pointer
r30	s8	general use, preserved by subroutines
r31	ra	link register

Of these registers, only r0 (zero) and r31 (return address), are fixed by the hardware. The other uses are by software convention only. For further information on register usage conventions, consult the MIPS Assembly Language Programmer's Guide.

1.3. Special Registers

The MIPS processor defines three special registers, whose use or modification is implicit in certain instructions. The special registers are:

PC	Program Counter
HI	Multiply/Divide register higher word
LO	Multiply/Divide register lower word

1.4. Coprocessor Registers

Each of 4 coprocessor units may define up to 32 general registers and 32 control registers. Registers associated with coprocessor zero, the system control coprocessor, are used to view and manipulate state associated with the exception handling, memory management, and diagnostic features of the machine. Coprocessor unit number one is used for the floating-point coprocessor, while units two and three are reserved for future definition by MIPS.

The system control coprocessor registers are described in chapter 4. Floating-point registers are described in chapter 3.

1.5. Restrictions

Elements of systems using R-series processors, such as caches, translation buffers, and buses, can and do affect both the performance and semantics of programs. Many of these effects are relevant only to the operating system kernel; this environment is both processor and system specific, and not part of the MIPS ISA specification. The semantics of user-mode programs is, at least partially, the domain of the ISA, which must be restricted to accommodate certain aspects of processor architectures. A complete specification of user-mode environments is the subject of Application Programming Interface (API) and Application Binary Interface (ABI) standards.

R-Series processors typically use separate caches for instructions and data without a hardware coherency mechanism. If a program writes data, and then attempts to execute that data, the operation is undefined and unpredictable. Data cache writeback and instruction cache invalidation between write and execute is necessary for correct operation. This service must be provided by the operating system for user-mode programs.

The MIPS I ISA does not address the semantics of a multiple processor environment. The MIPS II ISA adds three instructions (LL, SC, and SYNC) that allow well-defined operation of parallel programs in a multiprocessor environment.

1.6. Changes in this Edition

This edition of the MIPS R-Series Architecture continues the definition of the R4000 coprocessor 0 architecture.

This edition also continues definition of the MIPS III architecture. The MIPS III architecture is speculative at this point in time, and is intended only for use and discussion within MIPS.

The following is a brief summary of the changes from the January 1991 printing. Some of the changes are marked in the text with change bars. However, troff does not handle change bars in tables well, and so many changes are not so marked.

Add R6000 hazard table.

Update R4000 coprocessor 0 hazard information.

Clarify description of the UX, SX, and KX Status bits.

Coprocessor 0 register TagHi is not a R4000 register; it is reserved for future implementation.

Change low bits of TLBP result when no match occurs.

Rewrite LL/SC restrictions.

Move SYNC instruction from load/store section to SPEC section.

Rewrite LWCz, SWCz, LDCz, and SDCz formal specifications to allow the coprocessor to take or deliver data any way it likes.

Remove incorrect "indeterminate result" comment from the floating point architecture.

2. Instruction Set

The MIPS instruction set, as described in this chapter, provides an architecture-level definition of the MIPS user-mode instructions. Matters which are explicitly stated as undefined must be considered to be implementation-dependent and should therefore not be relied upon when, as is usually the case, transportability between implementations is required.

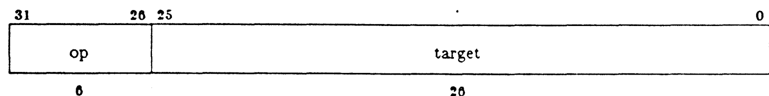
In particular, instruction formats which are marked with constant values must be used only with those constants, as future architecture extensions may use these field for other purposes, or implementations may rely on these fields having the constant values specified.

2.1. Instruction Formats and Notational Conventions

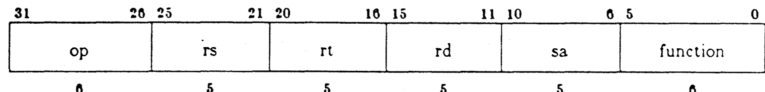
Every instruction consists of a single word (32 bits) aligned on a word boundary.

The major instruction formats are as follows:

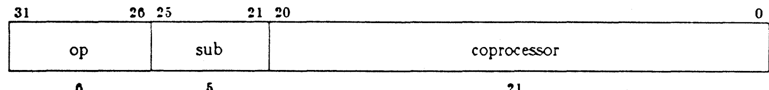
I-type (Immediate):



J-type (Jump):



R-type (Register):



Coprocessor:

31 — 26	25 — 21	20 — 0
op	sub	coprocessor
6	5	21

where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier
rt	is a 5-bit source/destination register specifier or sub-operation code
immediate	is a 16-bit immediate, branch displacement or address displacement
target	is a 26-bit jump target address
rd	is a 5-bit destination register specifier
sa	is a 5-bit shift amount
function	is a 6-bit function field

sub	is a 5-bit sub-operation code
coprocessor	the interpretation of rest of the instruction is coprocessor-specific

This document adopts the notational convention that all variable subfields in an instruction format (such as rs, rt, immediate, etc.) have lower case names, while subfields that are filled with constant values have upper case names.

For clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, rs = base in the format for Load and Store instructions.

The two instruction subfields op and function have constant 6-bit values for specific instructions. These values are given upper case mnemonic names in the main body of this document. For example, op = LB in the Load Byte instruction; op = SPECIAL and function = ADD in the Add instruction. In some cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. For example, LWCz (Load Coprocessor z) represents 4 different 6-bit opcodes, each composed of the fixed 4-bit subfield LWC concatenated with a variable 2-bit subfield z which designates one of the 4 coprocessor classes.

The actual bit encoding for all the mnemonics is specified in chapters 7 and 8.

Under the sub-heading "Operation," the operation performed by each instruction is described, using a high-level language notation. Special symbols used in the notation are described below.

symbol	meaning
←	Assignment
||	Bit string concatenation.
x^y	Replication of bit value x into a y-bit string. Note that x is always a single bit value.
$x_{y..z}$	Selection of bits y through z of bit string x. Little-endian bit notation is used. If y is less than z, this expression is an empty (zero-length) bit string.
+	Two's complement or floating-point addition
-	Two's complement or floating-point subtraction
*	Two's complement or floating-point multiplication
div	Two's complement integer division
mod	Two's complement modulo
/	Floating-point division
<	Two's complement less than comparison
and	Bitwise logical and
or	Bitwise logical or
xor	Bitwise logical xor
nor	Bitwise logical nor
GPR[x]	General register x. The contents of GPR[0] is always zero. Attempts to alter the contents of GPR[0] have no effect.
CPR[z, x]	Coprocessor unit z, general register x.
CCR[z, x]	Coprocessor unit z, control register x.
COC[z]	coprocessor unit z condition signal.
BigEndianMem	Big endian mode as configured at reset (0 → Little, 1 → Big). Specifies the endianess of the memory interface (see LoadMemory and StoreMemory), and the endianess of kernel and supervisor mode execution.
ReverseEndian	Signal to reverse the endianess of load and store instructions. This feature is available in user mode only, and is effected by setting the RE bit of the Status register. Thus ReverseEndian may be computed as (SR_{25} and UserMode).
BigEndianCPU	The endianess for load and store instructions (0 → Little, 1 → Big). In user mode, this endianess may be reversed by setting SR_{25}. Thus BigEndianCPU may be computed as BigEndianMem xor ReverseEndian.
LLbit	Bit of state to specify synchronization instructions. Set by LL, cleared by RFE and invalidate, and read by SC. (MIPS II only)
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 of two operations which execute at the same time should be pessimistic; the order is not defined.

The variables GPR[0..31], PC, HI, LO, CPR[0..3, 0..31], CCR[0..3, 0..31], COC[0..3], LLbit, BigEndianMem, TLB[0..TLBSIZE-1], Cache[INSTRUCTION..DATA][0..CACHESIZE-1], and Memory[0..MEMORYSIZE-1] represent the state of the machine and are therefore static. All other variables used in describing instructions are dynamic and local to the function or instruction involved.

The description of the immediate causes and manner of handling of exceptions is omitted from the instruction descriptions in this chapter. The exceptions that may occur due to the execution of each instruction is explicitly listed, and the causes and handling of these exceptions is detailed in chapter 5.

2.2. Memory Access Specification

In the operation description sections, the following functions are used to encapsulate the handling of virtual addresses, caching, and physical memory. These functions are defined in depth in chapter 4, and may differ between implementations.

function	meaning
AddressTranslation(vAddr, IorD)	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the page containing the virtual address is not present in the TLB.
LoadMemory(uncached, accessType, pAddr, vAddr, IorD)	For MIPS I and MIPS II processors, uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache. For MIPS III processors, uses the cache and main memory to find the contents of the doubleword containing the specified physical address. The low-order three bits of the address and the access type field indicates which of each of the eight bytes within the data doubleword need to be returned. If the cache is enabled for this access, the entire doubleword is returned and loaded into the cache.
StoreMemory(uncached, accessType, memoryWord, pAddr, vAddr, IorD)	For MIPS I and MIPS II processors, uses the cache, write buffer and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word should be stored. For MIPS III processors, uses the cache, write buffer and main memory to store the doubleword or part of doubleword specified as data in the doubleword containing the specified physical address. The low-order three bits of the address and the access type field indicates which of each of the eight bytes within the data doubleword should be stored.

The access type field indicates the size of the data item to be loaded or stored, according to the following table. Regardless of access type or byte-numbering order (endianess), the address specifies the byte which has the smallest byte address of each byte in the addressed field. For big-endian machines, this is the leftmost byte; for little-endian machines, this is the rightmost byte.

Access type		Meaning
Mnemonic	Value	
DOUBLEWORD	7	doubleword (64 bits)
SEPTIBYTE	6	seven bytes (56 bits)
SEXTIBYTE	5	six bytes (48 bits)
QUINTIBYTE	4	five bytes (40 bits)
WORD	3	word (32 bits)
TRIPLEBYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

The bytes within the addressed double-word which are used can be determined directly from the access type and the low-order three bits of the address, as given in the following table. Note that certain combinations of access type and low-order address bits can never occur, and cause address error exceptions (DOUBLEWORD: 001, 010, 011, 100, 101, 110, 111; WORD: 001, 010, 011, 101, 110, 111; TRIPLE-BYTE: 010, 011, 110, 111 HALFWORD: 001, 011, 101, 111).

Access type	Low-order address bits	Big-Endian Bytes accessed							
DOUBLE-WORD	000 (0)	0	1	2	3	4	5	6	7
WORD	000 (0)	0	1	2	3				
	100 (4)					4	5	6	7
TRIPLE-BYTE	000 (0)	0	1	2					
	001 (1)		1	2	3				
	100 (4)					4	5	6	
	101 (5)						5	6	7
HALF-WORD	000 (0)	0	1						
	010 (2)			2	3				
	100 (4)					4	5		
	110 (6)							6	7
BYTE	000 (0)	0							
	001 (1)		1						
	010 (2)			2					
	011 (3)				3				
	100 (4)					4			
	101 (5)						5		
	110 (6)							6	
	111 (7)								7

Access type	Low-order address bits	Little-Endian Bytes accessed							
DOUBLE-WORD	000 (0)	7	6	5	4	3	2	1	0
WORD	000 (0)					3	2	1	0
	100 (4)	7	6	5	4				
TRIPLE-BYTE	000 (0)						2	1	0
	001 (1)					3	2	1	
	100 (4)		6	5	4				
	101 (5)	7	6	5					
HALF-WORD	000 (0)							1	0
	010 (2)					3	2		
	100 (4)			5	4				
	110 (6)	7	6						
BYTE	000 (0)								0
	001 (1)							1	
	010 (2)						2		
	011 (3)					3			
	100 (4)				4				
	101 (5)			5					
	110 (6)		6						
	111 (7)	7							

2.3. Instruction Fetch and Decode

In the operation description sections, the actions required to fetch and decode instructions are not indicated, as they are the same for all instructions. In this section, we present pseudocode for fetching instructions, using the functions defined above.

Operation:

T: (physPC, uncached) ← AddressTranslation (PC, INSTRUCTION)
instruction ← LoadMemory (uncached, WORD, physPC, PC, INSTRUCTION)
PC ← PC + 4

2.4. Instruction Classes

MIPS instructions can be divided into a few general classes as follows:

- Load/Store instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is base register + 16-bit immediate offset.
- Computational instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.
- Jump and Branch instructions change the control flow of a program. Jumps are always to absolute 26-bit word addresses (J-type format, for subroutine calls), or 32-bit register byte addresses (R-type, for returns and dispatches). Branches have 16-bit offsets relative to the program counter (I-type). Jump and Link instructions save a return address in Register 31.
- Coprocessor instructions perform operations in the coprocessors. Coprocessor Loads and Stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see coprocessor

manuals). Coprocessor zero instructions manipulate the memory management and exception handling facilities of the processor.

- Special instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.

2.5. Load and Store Instructions

Load and store instructions transfer a variety of data types between the general registers and the memory system. Signed and unsigned bytes and halfwords may be loaded and sign- or zero-extended into word-wide registers, using the "load byte/halfword" and "load byte/halfword unsigned" instructions. The least-significant byte or halfword of a general register may be stored to memory using the "store byte/halfword" instructions. Halfword memory addresses must be aligned to halfword boundaries to avoid address exceptions. Using the "load/store word" instructions, word-sized values may be moved to and from word-aligned memory locations. An address error results if the memory address is not word-aligned.

Word-sized values that are not word-aligned may be loaded using the "load word left/right" instructions as a pair, and likewise, the "store word left/right" instructions work together to store unaligned, word-sized values. These instructions separately address the words containing the left and right ends of an unaligned word, and appropriately combine or divide the value when transferring to or from memory. Unaligned halfword values must be manipulated using the byte instructions; usually three instructions suffice to transfer a halfword value to or from an unaligned halfword address.

Two special instructions are provided in the MIPS II instruction set, named "Load Linked" and "Store Conditional." These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counts.

In the MIPS I instruction set, all loads are implemented with a latency of one instruction. That is, the instruction immediately following a load cannot use the contents of the register which will be loaded with the data being fetched from storage. An exception is the target register for the "load word left" and "load word right" instructions, which may be specified as the same register used as the destination of a load instruction that immediately precedes it.

In the MIPS II instruction set, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware will interlock, requiring additional real cycles, so scheduling load delay slots is still desirable, though not required for functional code.

Instruction Format:

31 26	25 21	20 16	15 0
op	base	rt	offset
6	5	5	16

where:

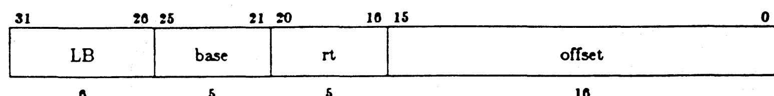
op	is the 6-bit operation code
base	is the 5-bit base register specifier
rt	is the 5-bit source (for stores) or destination (for loads) register specifier
offset	is the 16-bit signed immediate offset

Description	op
Load Byte	LB
Load Byte Unsigned	LBU
Load Halfword	LH
Load Halfword Unsigned	LHU
Load Word	LW
Load Word Left	LWL
Load Word Right	LWR
Load Linked	LL
Load Doubleword	LD
Load Doubleword Left	LDL
Load Doubleword Right	LDR
Load Linked Doubleword	LLD
Store Byte	SB
Store Halfword	SH
Store Word	SW
Store Word Left	SWL
Store Word Right	SWR
Store Conditional	SC
Store Doubleword	SD
Store Doubleword Left	SDL
Store Doubleword Right	SDR
Store Conditional Doubleword	SCD

LOAD BYTE

Format:

LB rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register rt.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: vAddr ← $((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0}) + \text{GPR[base]}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{1..0}$ xor BigEndianCPU^2
GPR[rt] ← undefined

T+1: GPR[rt] ← $(\text{mem}_{7+8*\text{byte}})^{24} \,\|\, \text{mem}_{7+8*\text{byte}..8*\text{byte}}$

MIPS II operation:

T: vAddr ← $((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0}) + \text{GPR[base]}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..2} \,\|\, (\text{pAddr}_{1..0}$ xor $\text{ReverseEndian}^2)$
mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{1..0}$ xor BigEndianCPU^2
GPR[rt] ← $(\text{mem}_{7+8*\text{byte}})^{24} \,\|\, \text{mem}_{7+8*\text{byte}..8*\text{byte}}$

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \,\|\, \text{offset}_{15..0}) + \text{GPR[base]}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,\|\, (\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^3)$
mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{2..0}$ xor BigEndianCPU^3
GPR[rt] ← $(\text{mem}_{7+8*\text{byte}})^{56} \,\|\, \text{mem}_{7+8*\text{byte}..8*\text{byte}}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD BYTE UNSIGNED

Format:

LBU rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LBU	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register rt.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
$GPR[rt] \leftarrow undefined$

T+1: $GPR[rt] \leftarrow 0^{24} \| mem_{7+8*byte..8*byte}$

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ ReverseEndian^2)$
$mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
$GPR[rt] \leftarrow 0^{24} \| mem_{7+8*byte..8*byte}$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$
$mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^3$
$GPR[rt] \leftarrow 0^{56} \| mem_{7+8*byte..8*byte}$

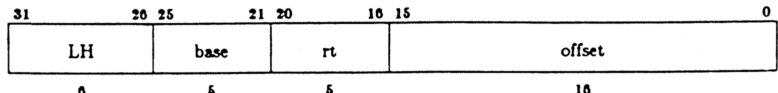
Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD HALFWORD

Format:

LH rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register rt.

If the least significant bit of the effective address is non-zero, an address error exception takes place.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \| 0)$
$GPR[rt] \leftarrow undefined$

T+1: $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte..8*byte}$

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ (ReverseEndian \| 0))$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \| 0)$
$GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte..8*byte}$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ (ReverseEndian^2 \| 0))$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^2 \| 0)$
$GPR[rt] \leftarrow (mem_{15+8*byte})^{48} \| mem_{15+8*byte..8*byte}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD HALFWORD UNSIGNED

Format:

LHU rt,offset(base)

31 26	25 21	20 16	15 0
LHU	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register rt.

If the least significant bit of the effective address is non-zero, an address error exception takes place.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \,||\, 0)$
$GPR[rt] \leftarrow undefined$

T+1: $GPR[rt] \leftarrow 0^{16} \,||\, mem_{15+8*byte..8*byte}$

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ (ReverseEndian \,||\, 0))$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \,||\, 0)$
$GPR[rt] \leftarrow 0^{16} \,||\, mem_{15+8*byte..8*byte}$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian^2 \,||\, 0))$
$mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^2 \,||\, 0)$
$GPR[rt] \leftarrow 0^{48} \,||\, mem_{15+8*byte..8*byte}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD WORD

Format:

LW   rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LW	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
$mem \leftarrow$ LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
$GPR[rt] \leftarrow$ undefined

T+1: $GPR[rt] \leftarrow mem$

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
$mem \leftarrow$ LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
$GPR[rt] \leftarrow mem$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}$ xor $(ReverseEndian \| 0^2))$
$mem \leftarrow$ LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
$byte \leftarrow vAddr_{2..0}$ xor $(BigEndianCPU \| 0^2)$
$GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \| mem_{31+8*byte..8*byte}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD WORD UNSIGNED

Format:

LWU rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LWU	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ (ReverseEndian \| 0^2))$
$mem \leftarrow LoadMemory (uncached, WORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU \| 0^2)$
$GPR[rt] \leftarrow 0^{32} \| mem_{31+8*byte..8*byte}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

LOAD DOUBLEWORD

Format:

LD rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
LD	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form the virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into general register rt.

If any of the three least significant bits of the virtual address is non-zero, an address error exception takes place.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← mem

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

LOAD WORD LEFT

Format:

LWL rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
LWL	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are shifted left so that the addressed byte is in the leftmost byte of a word. The bytes loaded from memory are merged with the contents of general register rt and the result is loaded into general register rt.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction. However, the contents of general register rt are internally bypassed within the processor so that it is permissible to specify register rt the same as a register which has been named as the target register of a load instruction on the previous instruction.

Address error exceptions due to byte alignment are suppressed by this instruction.

MIPS I operation:

T: $\text{vAddr} \leftarrow ((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
$(\text{pAddr}, \text{uncached}) \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA})$
$\text{byte} \leftarrow \text{vAddr}_{1..0} \text{ xor BigEndianCPU}^2$
if BigEndianMem = 0 then
$\quad \text{pAddr} \leftarrow \text{pAddr}_{31..2} \,\|\, 0^2$
endif
$\text{mem} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, \text{pAddr}, \text{vAddr}, \text{DATA})$

T+1: $\text{GPR}[\text{rt}] \leftarrow \text{mem}_{7+8*\text{byte}..0} \,\|\, \text{GPR}[\text{rt}]_{23-8*\text{byte}..0}$

MIPS II operation:

T: $\text{vAddr} \leftarrow ((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
$(\text{pAddr}, \text{uncached}) \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA})$
$\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \,\|\, (\text{pAddr}_{1..0} \text{ xor ReverseEndian}^2)$
if BigEndianMem = 0 then
$\quad \text{pAddr} \leftarrow \text{pAddr}_{31..2} \,\|\, 0^2$
endif
$\text{byte} \leftarrow \text{vAddr}_{1..0} \text{ xor BigEndianCPU}^2$
$\text{mem} \leftarrow \text{LoadMemory}(\text{uncached}, \text{byte}, \text{pAddr}, \text{vAddr}, \text{DATA})$
$\text{GPR}[\text{rt}] \leftarrow \text{mem}_{7+8*\text{byte}..0} \,\|\, \text{GPR}[\text{rt}]_{23-8*\text{byte}..0}$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
if $BigEndianMem = 0$ then
  $pAddr \leftarrow pAddr_{PSIZE-1..3} \| 0^{3}$
endif
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
$word \leftarrow vAddr_{2}\ xor\ BigEndianCPU$
$mem \leftarrow LoadMemory\ (uncached, 0 \| byte, pAddr, vAddr, DATA)$
$temp \leftarrow mem_{31+32*word-8*byte..32*word} \| GPR[rt]_{23-8*byte..0}$
$GPR[rt] \leftarrow (temp_{31})^{32} \| temp$

The operation of LWL given a doubleword in a register and a doubleword in memory is as follows:

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

where:

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
type	AccessType sent to memory
offset	$pAddr_{2..0}$ sent to memory
S	sign-extend of $destination_{31}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD WORD RIGHT

Format:

LWR rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
LWR	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are shifted right so that the addressed byte is in the rightmost byte of a word. The bytes loaded from memory are merged with the contents of general register rt and the result is loaded into general register rt.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction. However, the contents of general register rt are internally bypassed within the processor so that it is permissible to specify register rt the same as a register which has been named as the target register of a load instruction on the previous instruction.

Address error exceptions due to byte alignment are suppressed by this instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$byte \leftarrow vAddr_{1..0}$ xor $BigEndianCPU^2$
if BigEndianMem = 1 then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
mem ← LoadMemory (uncached, WORD-byte, pAddr, vAddr, DATA)

T+1: $GPR[rt] \leftarrow GPR[rt]_{31..32-8*byte} \| mem_{31..8*byte}$

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}$ xor $ReverseEndian^2)$
if BigEndianMem = 1 then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
$byte \leftarrow vAddr_{1..0}$ xor $BigEndianCPU^2$
mem ← LoadMemory (uncached, WORD-byte, pAddr, vAddr, DATA)
$GPR[rt] \leftarrow GPR[rt]_{31..32-8*byte} \| mem_{31..8*byte}$

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$
if $BigEndianMem = 1$ then
  $pAddr \leftarrow pAddr_{31..3} \| 0^3$
endif
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
$word \leftarrow vAddr_2\ xor\ BigEndianCPU$
$mem \leftarrow LoadMemory\ (uncached, WORD\text{-}byte, pAddr, vAddr, DATA)$
$temp \leftarrow GPR[rt]_{31..32-8*byte} \| mem_{31+32*word..32*word+8*byte}$
$GPR[rt] \leftarrow (temp_{31})^{32} \| temp$

The operation of LWR given a doubleword in a register and a doubleword in memory is as follows:

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	S S S S E F G I	0	7	0
1	S S S S E M N O	2	1	4	S S S S E F I J	1	6	0
2	S S S S E F M N	1	2	4	S S S S E I J K	2	5	0
3	S S S S 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	S S S S E F G M	0	3	4
5	S S S S E I J K	2	5	0	S S S S E F M N	1	2	4
6	S S S S E F I J	1	6	0	S S S S E M N O	2	1	4
7	S S S S E F G I	0	7	0	S S S S M N O P	3	0	4

where:

- LEM BigEndianMem = 0
- BEM BigEndianMem = 1
- type AccessType sent to memory
- offset $pAddr_{2..0}$ sent to memory
- S sign-extend of $destination_{31}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD DOUBLEWORD LEFT

Format:

LDL rt,offset(base)

31 26	25 21	20 16	15 0
LDL	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are shifted left so that the addressed byte is in the leftmost byte of a doubleword. The bytes loaded from memory are merged with the contents of general register rt and the result is loaded into general register rt.

Address error exceptions due to byte alignment are suppressed by this instruction.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$

$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$

$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$

if BigEndianMem = 0 then

$\quad pAddr \leftarrow pAddr_{PSIZE-1..3} \| 0^3$

endif

$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^3$

$mem \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$

$GPR[rt] \leftarrow mem_{7+8*byte..0} \| GPR[rt]_{63-8*byte..0}$

The operation of LDL given a doubleword in a register and a doubleword in memory is as follows:

register	A	B	C	D	E	F	G	H
memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G P	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

where:

LEM BigEndianMem = 0
BEM BigEndianMem = 1
type AccessType sent to memory
offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

LOAD DOUBLEWORD RIGHT

Format:

LDR rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
LDR	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are shifted right so that the addressed byte is in the rightmost byte of a doubleword. The bytes loaded from memory are merged with the contents of general register rt and the result is loaded into general register rt.

Address error exceptions due to byte alignment are suppressed by this instruction.

This instruction is defined only in MIPS III implementations.

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^3)$
if BigEndianMem = 1 then
  $pAddr \leftarrow pAddr_{31..3} \,||\, 0^3$
endif
$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^3$
$mem \leftarrow LoadMemory\ (uncached, DOUBLEWORD\text{-}byte, pAddr, vAddr, DATA)$
$GPR[rt] \leftarrow GPR[rt]_{63..64-8*byte} \,||\, mem_{63..8*byte}$

The operation of LDR given a doubleword in a register and a doubleword in memory is as follows:

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	B B C D E F G I	0	7	0	I J K L M N O P	7	0	0

where:

LEM BigEndianMem = 0
BEM BigEndianMem = 1
type AccessType sent to memory
offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

LOAD LINKED

Format:

LL rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
LL	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the LL must access memory before the LL, and loads and stores to shared memory fetched subsequent to the LL must access memory after the LL.

The processor will begin checking the accessed word for modification by other processors and devices.

Load Linked and Store Conditional can be used to atomically update memory locations, for example:

```
L1:
        LL      T1, (T0)
        ADD     T2, T1, 1
        SC      T2, (T0)
        BEQ     T2, 0, L1
         NOP
```

atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set.

The operation of LL is undefined if the addressed location is uncached. For synchronization between multiple processors, the operation of LL is undefined if the addressed location is non-coherent. A cache miss that occurs between LL and SC may cause SC to fail and so no load or store instruction should occur between LL and SC, otherwise the SC may never succeed. Exceptions also cause SC to fail, and so persistent exceptions must be avoided.

This instruction is available in user-mode; it is not necessary for coprocessor 0 to be enabled.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is not valid in MIPS I implementations.

MIPS II operation:

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
LLbit ← 1
SyncOperation()

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } (ReverseEndian \| 0^2))$
$mem \leftarrow LoadMemory (uncached, WORD, pAddr, vAddr, DATA)$
$byte \leftarrow vAddr_{2..0} \text{ xor } (BigEndianCPU \| 0^2)$
$GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \| mem_{31+8*byte..8*byte}$
$LLbit \leftarrow 1$
SyncOperation()

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD LINKED DOUBLE

Format:

LLD rt,offset(base)

31 26	25 21	20 16	15 0
LLD	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded into general register rt.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the LLD must access memory before the LLD, and loads and stores to shared memory fetched subsequent to the LLD must access memory after the LLD.

The processor will begin checking the accessed doubleword for modification by other processors and devices.

Load Linked Double and Store Conditional Double can be used to atomically update memory locations, for example:

```
L1:
        LLD     T1, (T0)
        ADD     T2, T1, 1
        SCD     T2, (T0)
        BEQ     T2, 0, L1
         NOP
```

atomically increments the doubleword addressed by T0. Changing the ADD to an OR changes this to an atomic bit test and set.

The operation of LLD is undefined if the addressed location is uncached. For synchronization between multiple processors, the operation of LLD is undefined if the addressed location is non-coherent. A cache miss that occurs between LLD and SCD may cause SCD to fail and so no load or store instruction should occur between LLD and SCD, otherwise the SCD may never succeed. Exceptions also cause SCD to fail, and so persistent exceptions must be avoided.

This instruction is available in user-mode; it is not necessary for coprocessor 0 to be enabled.

If either of the three least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt] ← mem
LLbit ← 1

Exceptions:

TLB refill exception
TLB invalid exception

Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

STORE BYTE

Format:

SB rt,offset(base)

31 – 26	25 – 21	20 – 16	15 – 0
SB	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The least significant byte of the contents of register rt is stored at the effective address.

MIPS I/II operation:

T:
$vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^{2})$
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
$data \leftarrow GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte}$
StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)

MIPS III operation:

T:
$vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
$data \leftarrow GPR[rt]_{63-8*byte..0} \,||\, 0^{8*byte}$
StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)

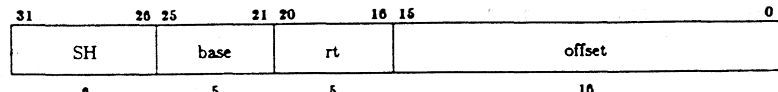
Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE HALFWORD

Format:

SH rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The least significant halfword of the contents of register rt is stored at the effective address.

If the least significant bit of the effective address is non-zero, an address error exception takes place.

MIPS I/II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ (ReverseEndian \,||\, 0))$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \,||\, 0)$
$data \leftarrow GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte}$
StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian^2 \,||\, 0))$
$byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^2 \,||\, 0)$
$data \leftarrow GPR[rt]_{63-8*byte..0} \,||\, 0^{8*byte}$
StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE WORD

Format:

SW rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
SW	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

MIPS I/II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \,\|\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$data \leftarrow GPR[rt]$
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \,\|\, offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \,\|\, (pAddr_{2..0}\ xor\ (ReverseEndian \,\|\, 0^2))$
$byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU \,\|\, 0^2)$
$data \leftarrow GPR[rt]_{63-8*byte..0} \,\|\, 0^{8*byte}$
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

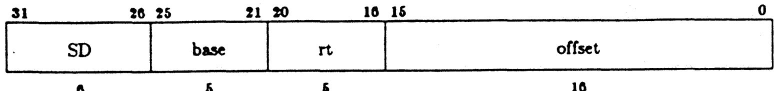
Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE DOUBLEWORD

Format:

SD rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form the virtual address. The contents of general register rt are stored at the memory location specified by the effective address.

If any of the three least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: vAddr ← $((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← GPR[rt]
StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and MIPS II)

STORE WORD LEFT

Format:

SWL rt,offset(base)

31 … 26	25 … 21	20 … 16	15 … 0
SWL	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are shifted right so that the leftmost byte of the word is in the position of the addressed byte. The bytes contained in the word after shifting are stored into the word containing the addressed byte.

Address error exceptions due to byte alignment are suppressed by this instruction.

MIPS I/II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0} \text{ xor } ReverseEndian^2)$
if $BigEndianMem = 0$ then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
$byte \leftarrow vAddr_{1..0} \text{ xor } BigEndianCPU^2$
$data \leftarrow 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte}$
StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } ReverseEndian^3)$
if $BigEndianMem = 0$ then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
$byte \leftarrow vAddr_{1..0} \text{ xor } BigEndianCPU^2$
if $(vAddr_2 \text{ xor } BigEndianCPU) = 0$ 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)

The operation of SWL given a doubleword in a register and a doubleword in memory is as follows:

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

where:

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
type	AccessType sent to memory
offset	$pAddr_{2..0}$ sent to memory
S	sign-extend of $destination_{31}$

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE WORD RIGHT

Format:

SWR rt,offset(base)

31 … 26	25 … 21	20 … 16	15 … 0
SWR	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are shifted left so that the right byte of the word is in the position of the addressed byte. The bytes contained in the word after shifting are stored into the word containing the addressed byte.

Address error exceptions due to byte alignment are suppressed by this instruction.

MIPS I/II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ ReverseEndian^2)$
if BigEndianMem = 1 then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
$data \leftarrow GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$
if BigEndianMem = 1 then
  $pAddr \leftarrow pAddr_{31..2} \| 0^2$
endif
$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
$data \leftarrow GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
if $(vAddr_2\ xor\ BigEndianCPU) = 0$ then
  $data \leftarrow 0^{32} \| GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
else
  $data \leftarrow GPR[rt]_{31-8*byte..0} \| 0^{8*byte} \| 0^{32}$
endif
StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)

The operation of SWR given a doubleword in a register and a doubleword in memory is as follows:

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

where:

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
type	AccessType sent to memory
offset	$pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE DOUBLEWORD LEFT

Format:

SDL rt,offset(base)

31 26	25 21	20 16	15 0
SDL	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are shifted right so that the leftmost byte of the doubleword is in the position of the addressed byte. The bytes contained in the doubleword after shifting are stored into the doubleword containing the addressed byte.

Address error exceptions due to byte alignment are suppressed by this instruction.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \| (\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^3)$
if BigEndianMem = 0 then
  pAddr ← $\text{pAddr}_{31..3} \| 0^3$
endif
byte ← $\text{vAddr}_{2..0}$ xor BigEndianCPU^3
data ← $0^{56-8*\text{byte}} \| \text{GPR}[\text{rt}]_{63..56-8*\text{byte}}$
StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)

The operation of SDL given a doubleword in a register and a doubleword in memory is as follows:

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

where:

LEM BigEndianMem = 0
BEM BigEndianMem = 1
type AccessType sent to memory
offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

STORE DOUBLEWORD RIGHT

Format:

SDR rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
SDR	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are shifted left so that the right byte of the doubleword is in the position of the addressed byte. The bytes contained in the doubleword after shifting are stored into the doubleword containing the addressed byte.

Address error exceptions due to byte alignment are suppressed by this instruction.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$
if BigEndianMem = 1 then
$pAddr \leftarrow pAddr_{31..3} \| 0^3$
endif
$byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^3$
$data \leftarrow GPR[rt]_{63-8*byte..0} \| 0^{8*byte}$
StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr, DATA)

The operation of SDR given a doubleword in a register and a doubleword in memory is as follows:

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

where:

LEM BigEndianMem = 0
BEM BigEndianMem = 1
type AccessType sent to memory
offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

STORE CONDITIONAL

Format:

SC rt,offset(base)

31 — 26	25 — 21	20 — 16	15 — 0
SC	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are conditionally stored at the memory location specified by the effective address.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the SC must access memory before the SC, and loads and stores to shared memory fetched subsequent to the SC must access memory after the SC.

If any other processor or device has modified this physical address since the time of the previous Load Linked instruction, or if a RFE or ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register rt after execution of the instruction. A successful store sets the contents of general register rt to 1, and an unsuccessful store sets it to 0.

The operation of Store Conditional is undefined when the address is different from the address used in the last Load Linked.

This instruction is available in user-mode; it is not necessary for coprocessor 0 to be enabled.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

If this instruction could both fail and take an exception, the exception takes precedence.

This instruction is not valid in MIPS I implementations.

MIPS II operation:

T: vAddr ← $((\text{offset}_{15})^{16} \| \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← GPR[rt]
if LLbit then
  StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0^{31} || LLbit
SyncOperation()

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \| (\text{pAddr}_{2..0}$ xor (ReverseEndian $\| 0^2$))
byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU $\| 0^2$)
data ← $\text{GPR[rt]}_{63-8*\text{byte}..0} \| 0^{8*\text{byte}}$
if LLbit then
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
endif
GPR[rt] ← $0^{63} \|$ LLbit
SyncOperation()

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE CONDITIONAL DOUBLEWORD

Format:

SCD rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
SCD	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are conditionally stored at the memory location specified by the effective address.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the SCD must access memory before the SCD, and loads and stores to shared memory fetched subsequent to the SCD must access memory after the SCD.

If any other processor or device has modified this physical address since the time of the previous Load Linked Doubleword instruction, or if a RFE or ERET instruction occurs between the Load Linked Double instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register rt after execution of the instruction. A successful store sets the contents of general register rt to 1, and an unsuccessful store sets it to 0.

The operation of Store Conditional Double is undefined when the address is different from the address used in the last Load Linked Double.

This instruction is available in user-mode; it is not necessary for coprocessor 0 to be enabled.

If either of the three least significant bits of the effective address is non-zero, an address error exception takes place.

If this instruction could both fail and take an exception, the exception takes precedence.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

```
T:    vAddr ← ((offset_15)^48 || offset_15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      data ← GPR[rt]
      if LLbit then
        StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
      endif
      GPR[rt] ← 0^63 || LLbit
      SyncOperation()
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I and II only)

2.6. Computational Instructions

2.6.1. ALU Immediate Instructions

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 0
op	rs	rt	immediate
6	5	5	16

where:

op	is a 6-bit operation code
rs	is a 5-bit register source
rt	is a 5-bit register destination
immediate	is a 16-bit immediate operand

Description	op
Add Immediate	ADDI
Add Immediate Unsigned	ADDIU
Set on Less Than Immediate	SLTI
Set on Less Than Immediate Unsigned	SLTIU
And Immediate	ANDI
Or Immediate	ORI
Exclusive Or Immediate	XORI
Load Upper Immediate	LUI
Doubleword Add Immediate	DADDI
Doubleword Add Immediate Unsigned	DADDIU

The immediate operand is sign-extended for the "add" and "set on less than" instructions, and zero-extended for the "and," "or," and "exclusive or" instructions.

ADD IMMEDIATE

Format:

ADDI rt,rs,immediate

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

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form the result. The result is placed into general register rt.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ (two's complement overflow). The destination register rt is not modified when an integer overflow exception occurs.

MIPS I/II operation:

T: $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \| immediate_{15..0}$

MIPS III operation:

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

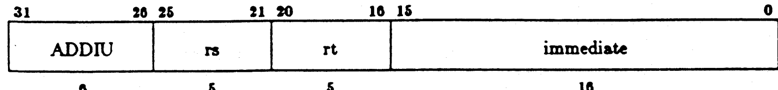
Exceptions:

Integer overflow exception

ADD IMMEDIATE UNSIGNED

Format:

ADDIU rt,rs,immediate



Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form a result. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances.

The "unsigned" in this instruction name is a misnomer; this instruction differs from ADD Immediate only in that it does not trap on overflow.

MIPS I/II operation:

T: $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \| immediate_{15..0}$

MIPS III operation:

T: $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \| immediate_{15..0}$

$GPR[rt] \leftarrow (temp_{31})^{32} \| temp_{31..0}$

Exceptions:

none

DOUBLEWORD ADD IMMEDIATE

Format:

DADDI rt,rs,immediate

31 ... 26	25 ... 21	20 ... 16	15 ... 0
DADDI	rs	rt	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form the result. The result is placed into general register rt.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (two's complement overflow). The destination register rt is not modified when an integer overflow exception occurs.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

$$T: \quad GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{48} \,||\, immediate_{15..0}$$

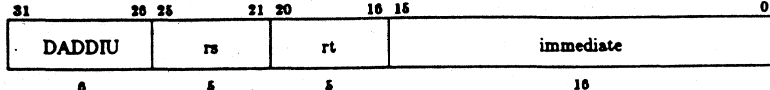
Exceptions:

Integer overflow exception
Reserved instruction exception (MIPS I and II only)

DOUBLEWORD ADD IMMEDIATE UNSIGNED

Format:

DADDIU rt,rs,immediate



Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form a result. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances.

The "unsigned" in this instruction name is a misnomer; this instruction differs from Doubleword ADD Immediate only in that it does not trap on overflow.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{48} \| immediate_{15..0}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

SET ON LESS THAN IMMEDIATE

Format:

SLTI rt,rs,immediate

31–26	25–21	20–16	15–0
SLTI	rs	rt	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as signed integers, if rs is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction overflows.

MIPS I/II operation:

T: if GPR[rs] < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$ then
  GPR[rt] ← 0^{31} || 1
else
  GPR[rt] ← 0^{32}
endif

MIPS III operation:

T: if GPR[rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$ then
  GPR[rt] ← 0^{63} || 1
else
  GPR[rt] ← 0^{64}
endif

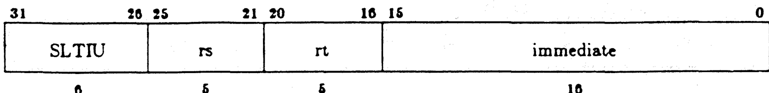
Exceptions:

none

SET ON LESS THAN UNSIGNED IMMEDIATE

Format:

SLTIU rt,rs,immediate



Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if rs is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction overflows.

MIPS I/II operation:

```
T:    if 0 || GPR[rs] < 0 || (immediate15)^16 || immediate15..0 then
         GPR[rt] ← 0^31 || 1
      else
         GPR[rt] ← 0^32
      endif
```

MIPS III operation:

```
T:    if 0 || GPR[rs] < 0 || (immediate15)^48 || immediate15..0 then
         GPR[rt] ← 0^63 || 1
      else
         GPR[rt] ← 0^64
      endif
```

Exceptions:

none

AND IMMEDIATE

Format:

ANDI rt,rs,immediate

31 — 26	25 — 21	20 — 16	15 — 0
ANDI	rs	rt	immediate
6	5	5	16

Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bitwise logical and operation. The result is placed into general register rt.

MIPS I/II operation:

T: $GPR[rt] \leftarrow 0^{16} \,||\, (\text{immediate and } GPR[rs]_{15..0})$

MIPS III operation:

T: $GPR[rt] \leftarrow 0^{48} \,||\, (\text{immediate and } GPR[rs]_{15..0})$

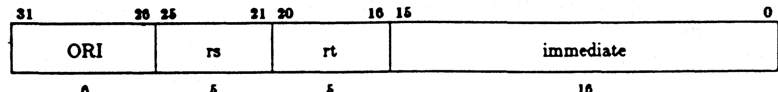
Exceptions:

none

OR IMMEDIATE

Format:

ORI rt,rs,immediate



Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bitwise logical or operation. The result is placed into general register rt.

MIPS I/II operation:

T: $GPR[rt] \leftarrow GPR[rs]_{31..16} \,||\, (immediate\ or\ GPR[rs]_{15..0})$

MIPS III operation:

T: $GPR[rt] \leftarrow GPR[rs]_{63..16} \,||\, (immediate\ or\ GPR[rs]_{15..0})$

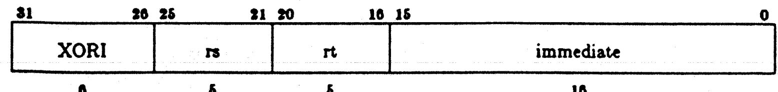
Exceptions:

none

EXCLUSIVE OR IMMEDIATE

Format:

XORI rt,rs,immediate



Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bitwise logical exclusive-or operation. The result is placed into general register rt.

MIPS I/II operation:

T: $GPR[rt] \leftarrow GPR[rs]$ xor $(0^{16} \| immediate)$

MIPS III operation:

T: $GPR[rt] \leftarrow GPR[rs]$ xor $(0^{48} \| immediate)$

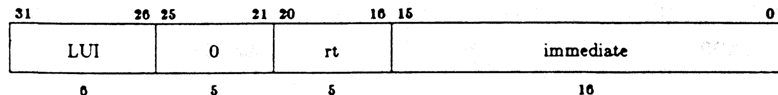
Exceptions:

none

LOAD UPPER IMMEDIATE

Format:

LUI rt,immediate



Description:

The 16-bit immediate is shifted left 16-bits and concatenated to 16 bits of zeroes. The result is placed into general register rt.

MIPS I/II operation:

T: $GPR[rt] \leftarrow immediate \parallel 0^{16}$

MIPS III operation:

T: $GPR[rt] \leftarrow (immediate_{15})^{32} \parallel immediate \parallel 0^{16}$

Exceptions:

none

2.6.2. ALU 3-Operand Register-Type Instructions

Instruction Format:

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

where:

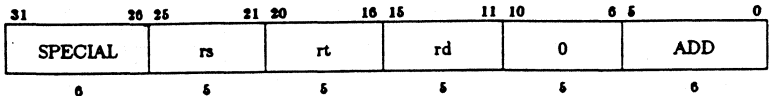
SPECIAL	is a 6-bit major operation code
rs	is a 5-bit source1 register
rt	is a 5-bit source2 register
rd	is a 5-bit destination register
sa	is a 5-bit unused field
function	is a 6-bit function code

Description	function
Add	ADD
Add Unsigned	ADDU
Subtract	SUB
Subtract Unsigned	SUBU
Doubleword Add	DADD
Doubleword Add Unsigned	DADDU
Doubleword Subtract	DSUB
Doubleword Subtract Unsigned	DSUBU
Set on Less Than	SLT
Set on Less Than Unsigned	SLTU
And	AND
Or	OR
Exclusive Or	XOR
Nor	NOR

ADD

Format:

ADD rd,rs,rt



Description:

The contents of general register rs and the contents of general register rt are added to form a result. The result is placed into general register rd.

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

MIPS I/II operation:

T: GPR[rd] ← GPR[rs] + GPR[rt]

MIPS III operation:

T: temp ← GPR[rs] + GPR[rt]
GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$

Exceptions:

Integer overflow exception

ADD UNSIGNED

Format:

ADDU rd,rs,rt

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

Description:

The contents of general register rs and the contents of general register rt are added to form a result. The result is placed into general register rd.

The "unsigned" in this instruction name is a misnomer; this instruction differs from ADD only in that it does not trap on overflow.

No integer overflow exception occurs under any circumstances.

MIPS I/II operation:

T: GPR[rd] ← GPR[rs] + GPR[rt]

MIPS III operation:

T: temp ← GPR[rs] + GPR[rt]
GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$

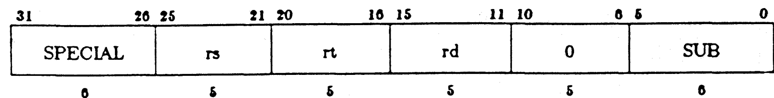
Exceptions:

none

SUBTRACT

Format:

SUB rd,rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs to form a result. The result is placed into general register rd.

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

MIPS I/II operation:

T: GPR[rd] ← GPR[rs] − GPR[rt]

MIPS III operation:

T: temp ← GPR[rs] − GPR[rt]
GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$

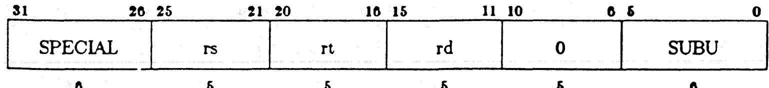
Exceptions:

Integer overflow exception

SUBTRACT UNSIGNED

Format:

SUBU rd,rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs to form a result. The result is placed into general register rd.

The "unsigned" in this instruction name is a misnomer; this instruction differs from SUB only in that it does not trap on overflow.

No integer overflow exception occurs under any circumstances.

MIPS I/II operation:

T: GPR[rd] ← GPR[rs] − GPR[rt]

MIPS III operation:

T: temp ← GPR[rs] − GPR[rt]
GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$

Exceptions:

none

DOUBLEWORD ADD

Format:

DADD rd,rs,rt

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

Description:

The contents of general register rs and the contents of general register rt are added to form a result. The result is placed into general register rd.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (two's complement overflow). The destination register is not modified if the exception is taken.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: GPR[rd] ← GPR[rs] + GPR[rt]

Exceptions:

Integer overflow exception
Reserved instruction exception (MIPS I and II only)

DOUBLEWORD ADD UNSIGNED

Format:

DADDU rd,rs,rt

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

Description:

The contents of general register rs and the contents of general register rt are added to form a result. The result is placed into general register rd.

The "unsigned" in this instruction name is a misnomer; this instruction differs from ADD only in that it does not trap on overflow.

No integer overflow exception occurs under any circumstances.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III Operation:

T: GPR[rd] ← GPR[rs] + GPR[rt]

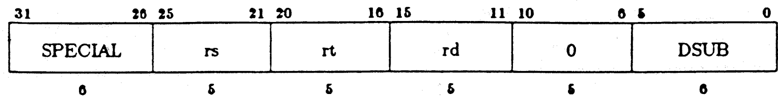
Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SUBTRACT

Format:

DSUB rd,rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs to form a result. The result is placed into general register rd.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (two's complement overflow). The destination register is not modified if the exception is taken.

MIPS III operation:

T: GPR[rd] ← GPR[rs] − GPR[rt]

Exceptions:

Integer overflow exception
Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SUBTRACT UNSIGNED

Format:

DSUBU rd,rs,rt

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

Description:

The contents of general register rt is subtracted from the contents of general register rs to form a result. The result is placed into general register rd.

The "unsigned" in this instruction name is a misnomer; this instruction differs from SUB only in that it does not trap on overflow.

No integer overflow exception occurs under any circumstances.

MIPS III operation:

T: GPR[rd] ← GPR[rs] – GPR[rt]

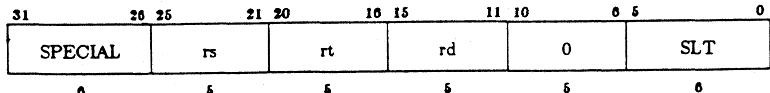
Exceptions:

Reserved instruction exception (MIPS I and II only)

SET ON LESS THAN

Format:

SLT rd,rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are less than the contents of general register rt, the result is set to one, otherwise the result is set to zero. The result is placed into general register rd.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction overflows.

MIPS I/II operation:

T: if GPR[rs] < GPR[rt] then
  $GPR[rd] \leftarrow 0^{31} \,||\, 1$
else
  $GPR[rd] \leftarrow 0^{32}$
endif

MIPS III operation:

T: if GPR[rs] < GPR[rt] then
  $GPR[rd] \leftarrow 0^{63} \,||\, 1$
else
  $GPR[rd] \leftarrow 0^{64}$
endif

Exceptions:

none

SET ON LESS THAN UNSIGNED

Format:

SLTU rd,rs,rt

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

Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are less than the contents of general register rt, the result is set to one, otherwise the result is set to zero. The result is placed into general register rd.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction overflows.

MIPS I/II operation:

T: if $0 \| GPR[rs] < 0 \| GPR[rt]$ then
  $GPR[rd] \leftarrow 0^{31} \| 1$
else
  $GPR[rd] \leftarrow 0^{32}$
endif

MIPS III operation:

T: if $0 \| GPR[rs] < 0 \| GPR[rt]$ then
  $GPR[rd] \leftarrow 0^{63} \| 1$
else
  $GPR[rd] \leftarrow 0^{64}$
endif

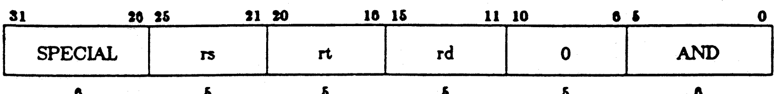
Exceptions:

none

AND

Format:

AND rd,rs,rt



Description:

The contents of general register rs is combined with the contents of general register rt in a bitwise logical and operation. The result is placed into general register rd.

Operation:

T: GPR[rd] ← GPR[rs] and GPR[rt]

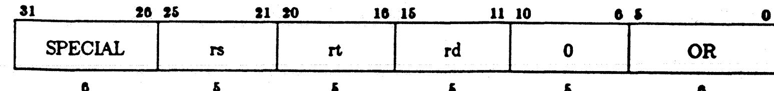
Exceptions:

none

OR

Format:

OR rd,rs,rt



Description:

The contents of general register rs is combined with the contents of general register rt in a bitwise logical or operation. The result is placed into general register rd.

Operation:

T: GPR[rd] ← GPR[rs] or GPR[rt]

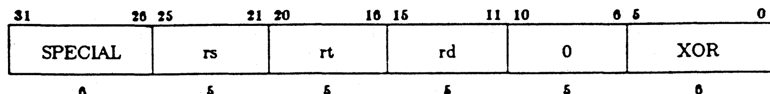
Exceptions:

none

EXCLUSIVE OR

Format:

XOR rd,rs,rt



Description:

The contents of general register rs is combined with the contents of general register rt in a bitwise logical exclusive or operation. The result is placed into general register rd.

Operation:

T: GPR[rd] ← GPR[rs] xor GPR[rt]

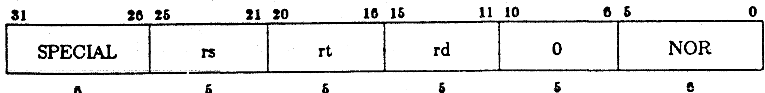
Exceptions:

none

NOR

Format:

NOR rd,rs,rt



Description:

The contents of general register rs is combined with the contents of general register rt in a bitwise logical nor operation. The result is placed into general register rd.

Operation:

T: GPR[rd] ← GPR[rs] nor GPR[rt]

Exceptions:

none

2.6.3. Shift Instructions

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	rs	rt	rd	sa	function
6	5	5	5	5	6

where:

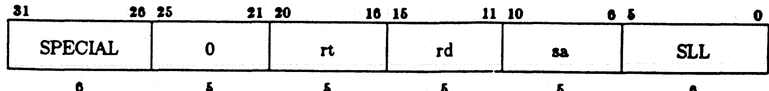
SPECIAL	is the 6-bit Special operation code
rs	is a 5-bit source register specifier or the most significant bit of a 6-bit shift amount
rt	is a 5-bit source register specifier
rd	is a 5-bit destination register specifier
sa	is a 5-bit shift amount field
function	is a 6-bit function field

Description	function
Shift Left Logical	SLL
Shift Right Logical	SRL
Shift Right Arithmetic	SRA
Shift Left Logical Variable	SLLV
Shift Right Logical Variable	SRLV
Shift Right Arithmetic Variable	SRAV
Doubleword Shift Left Logical	DSLL
Doubleword Shift Right Logical	DSRL
Doubleword Shift Right Arithmetic	DSRA
Doubleword Shift Left Logical + 32	DSLL32
Doubleword Shift Right Logical + 32	DSRL32
Doubleword Shift Right Arithmetic + 32	DSRA32
Doubleword Shift Left Logical Variable	DSLLV
Doubleword Shift Right Logical Variable	DSRLV
Doubleword Shift Right Arithmetic Variable	DSRAV

SHIFT LEFT LOGICAL

Format:

SLL rd,rt,sa



Description:

The contents of general register rt are shifted left by sa bits, inserting zeroes into the low order bits. The result is placed in register rd.

MIPS I/II operation:

T: $GPR[rd] \leftarrow GPR[rt]_{31-sa..0} \parallel 0^{sa}$

MIPS III operation:

T: $s \leftarrow 0 \parallel sa$
$temp \leftarrow GPR[rt]_{31-s..0} \parallel 0^{s}$
$GPR[rd] \leftarrow (temp_{31})^{32} \parallel temp$

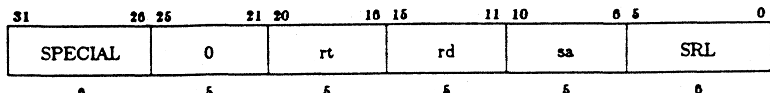
Exceptions:

none

SHIFT RIGHT LOGICAL

Format:

SRL rd,rt,sa



Description:

The contents of general register rt are shifted right by sa bits, inserting zeroes into the high order bits. The result is placed in register rd.

MIPS I/II operation:

T: $GPR[rd] \leftarrow 0^{sa} \| GPR[rt]_{31..sa}$

MIPS III operation:

T: $s \leftarrow 0 \| sa$
$temp \leftarrow 0^{s} \| GPR[rt]_{31..s}$
$GPR[rd] \leftarrow (temp_{31})^{32} \| temp$

Exceptions:

none

SHIFT RIGHT ARITHMETIC

Format:

SRA rd,rt,sa

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
SPECIAL	0	rt	rd	sa	SRA
6	5	5	5	5	6

Description:

The contents of general register rt are shifted right by sa bits, sign-extending the high order bits. The result is placed in register rd.

MIPS I/II operation:

T: $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} \,||\, GPR[rt]_{31..sa}$

MIPS III operation:

T: $s \leftarrow 0 \,||\, sa$
$temp \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$
$GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$

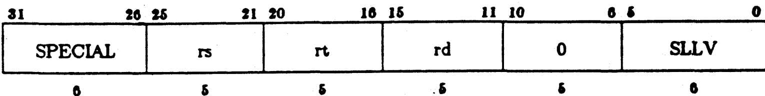
Exceptions:

none

SHIFT LEFT LOGICAL VARIABLE

Format:

SLLV rd,rt,rs



Description:

The contents of general register rt are shifted left by the number of bits specified by the low-order 5 bits of the contents of general register rs, inserting zeroes into the low order bits. The result is placed in register rd.

MIPS I/II operation:

T: $s \leftarrow GPR[rs]_{4..0}$
$GPR[rd] \leftarrow GPR[rt]_{(31-s)..0} \mathbin{||} 0^s$

MIPS III operation:

T: $s \leftarrow 0 \mathbin{||} GPR[rs]_{4..0}$
$temp \leftarrow GPR[rt]_{(31-s)..0} \mathbin{||} 0^s$
$GPR[rd] \leftarrow (temp_{31})^{32} \mathbin{||} temp$

Exceptions:

none

SHIFT RIGHT LOGICAL VARIABLE

Format:

SRLV rd,rt,rs

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

Description:

The contents of general register rt are shifted right by the number of bits specified by the low-order 5 bits of the contents of general register rs, inserting zeroes into the high order bits. The result is placed in register rd.

MIPS I/II operation:

T: $s \leftarrow GPR[rs]_{4..0}$
$GPR[rd] \leftarrow 0^s \,||\, GPR[rt]_{31..s}$

MIPS III operation:

T: $s \leftarrow 0 \,||\, GPR[rs]_{4..0}$
$temp \leftarrow 0^s \,||\, GPR[rt]_{31..s}$
$GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$

Exceptions:

none

SHIFT RIGHT ARITHMETIC VARIABLE

Format:

SRAV rd,rt,rs

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

Description:

The contents of general register rt are shifted right by the number of bits specified by the low-order 5 bits of the contents of general register rs, sign-extending the high order bits. The result is placed in register rd.

MIPS I/II operation:

T: $s \leftarrow GPR[rs]_{4..0}$
$GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \| GPR[rt]_{31..s}$

MIPS III operation:

T: $s \leftarrow 0 \| GPR[rs]_{4..0}$
$temp \leftarrow (GPR[rt]_{31})^{s} \| GPR[rt]_{31..s}$
$GPR[rd] \leftarrow (temp_{31})^{32} \| temp$

Exceptions:

none

DOUBLEWORD SHIFT LEFT LOGICAL

Format:

DSLL rd,rt,sa

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	0	rt	rd	sa	DSLL
6	5	5	5	5	6

Description:

The contents of general register rt are shifted left by sa bits, inserting zeroes into the low order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 0 \| sa$
$GPR[rd] \leftarrow GPR[rt]_{63-s..0} \| 0^s$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT LEFT LOGICAL + 32

Format:

DSLL32 rd,rt,sa

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
SPECIAL	0	rt	rd	sa	DSLL32
6	5	5	5	5	6

Description:

The contents of general register rt are shifted left by 32+sa bits, inserting zeroes into the low order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 1 \,||\, sa$

$GPR[rd] \leftarrow GPR[rt]_{63-s..0} \,||\, 0^s$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT LOGICAL

Format:

DSRL rd,rt,sa

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	0	rt	rd	sa	DSRL
6	5	5	5	5	6

Description:

The contents of general register rt are shifted right by sa bits, inserting zeroes into the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 0 \parallel sa$
$GPR[rd] \leftarrow 0^s \parallel GPR[rt]_{63..s}$

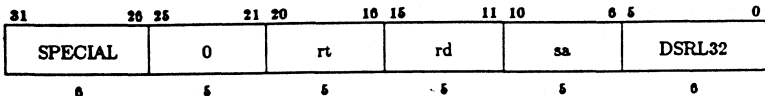
Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT LOGICAL + 32

Format:

DSRL32 rd,rt,sa



Description:

The contents of general register rt are shifted right by 32+sa bits, inserting zeroes into the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 1 \| sa$

$GPR[rd] \leftarrow 0^s \| GPR[rt]_{63..s}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT ARITHMETIC

Format:

DSRA rd,rt,sa

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	0	rt	rd	sa	DSRA
6	5	5	5	5	6

Description:

The contents of general register rt are shifted right by sa bits, sign-extending the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 0 \,\|\, sa$

$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,\|\, GPR[rt]_{63..s}$

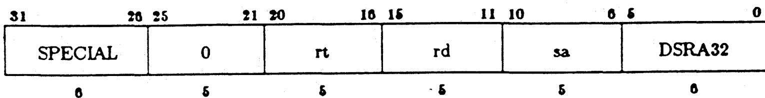
Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT ARITHMETIC + 32

Format:

DSRA32 rd,rt,sa



Description:

The contents of general register rt are shifted right by 32+sa bits, sign-extending the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow 1 \,||\, sa$

$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT LEFT LOGICAL VARIABLE

Format:

DSLLV rd,rt,rs

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

Description:

The contents of general register rt are shifted left by the number of bits specified by the low-order 6 bits of the contents of general register rs, inserting zeroes into the low order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow GPR[rs]_{5..0}$
$GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \| 0^s$

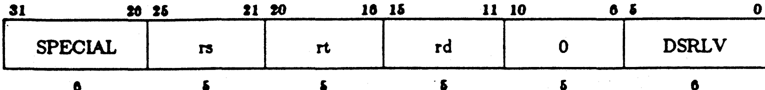
Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT LOGICAL VARIABLE

Format:

DSRLV rd,rt,rs



Description:

The contents of general register rt are shifted right by the number of bits specified by the low-order 6 bits of the contents of general register rs, inserting zeroes into the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow GPR[rs]_{5..0}$
$GPR[rd] \leftarrow 0^s \,||\, GPR[rt]_{63..s}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD SHIFT RIGHT ARITHMETIC VARIABLE

Format:

DSRAV rd,rt,rs

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

Description:

The contents of general register rt are is shifted right by the number of bits specified by the low-order 6 bits of the contents of general register rs, sign-extending the high order bits. The result is placed in register rd.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T: $s \leftarrow GPR[rs]_{5..0}$
$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \| GPR[rt]_{63..s}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

2.6.4. Multiply/Divide Instructions

Instruction Format:

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL	rs	rt	rd	sa	function
6	5	5	5	5	6

where:

SPECIAL	is the 6-bit Special opcode
rs	is a 5-bit source register specifier
rt	is a 5-bit source register specifier
rd	is a 5-bit destination register
sa	is not used
function	is a 6-bit function field

Description	function
Multiply	MULT
Multiply Unsigned	MULTU
Divide	DIV
Divide Unsigned	DIVU
Doubleword Multiply	DMULT
Doubleword Multiply Unsigned	DMULTU
Doubleword Divide	DDIV
Doubleword Divide Unsigned	DDIVU
Move From HI	MFHI
Move To HI	MTHI
Move From LO	MFLO
Move To LO	MTLO

In some processors, multiply and divide operations are performed by a separate, autonomous execution unit. After a multiply or divide operation is started, execution of other instructions may continue in parallel. The multiply/divide unit may continue to operate during cache miss and other delaying cycles in which no instructions are executed.

The number of cycles required for multiply/divide operations is implementation-dependent. The MFHI and MFLO instructions are interlocked so that any attempt to read them before operations have completed will cause execution of instructions to be delayed until the operation finishes.

The table below gives the number of cycles required between a MULT, MULTU, DIV or DIVU operation and a subsequent MFHI or MFLO operation, in order that no interlock or stall occurs for each implementation:

implementation	cycles required							
	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
R2000	12	12	35	35	-	-	-	-
R3000	12	12	35	35	-	-	-	-
R4000	12	12	76	76	20	20	133	133
R6000	17	18	38	37	-	-	-	-

MULTIPLY

Format:

MULT rs,rt

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL	rs	rt	0	MULT
6	5	5	10	6

Description:

The contents of general register rs and the contents of general register rt are multipled, treating both operands as two's complement values.

No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

MIPS I/II operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: $t \leftarrow GPR[rs] * GPR[rt]$
$LO \leftarrow t_{31..0}$
$HI \leftarrow t_{63..32}$

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: $t \leftarrow GPR[rs]_{31..0} * GPR[rt]_{31..0}$
$LO \leftarrow (t_{31})^{32} \| t_{31..0}$
$HI \leftarrow (t_{63})^{32} \| t_{63..32}$

Exceptions:

none

MULTIPLY UNSIGNED

Format:

MULTU rs,rt

31 26	25 21	20 16	15 6	5 0
SPECIAL	rs	rt	0	MULTU
6	5	5	10	6

Description:

The contents of general register rs and the contents of general register rt are multipled, treating both operands as unsigned values.

No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

MIPS I/II operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: $t \leftarrow (0 \parallel GPR[rs]) * (0 \parallel GPR[rt])$
$LO \leftarrow t_{31..0}$
$HI \leftarrow t_{63..32}$

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: $t \leftarrow (0 \parallel GPR[rs]_{31..0}) * (0 \parallel GPR[rt]_{31..0})$
$LO \leftarrow (t_{31})^{32} \parallel t_{31..0}$
$HI \leftarrow (t_{63})^{32} \parallel t_{63..32}$

Exceptions:

none

DIVIDE

Format:

DIV rs,rt

31 ... 26	25 ... 21	20 ... 16	15 ... 6	5 ... 0
SPECIAL	rs	rt	0	DIV
6	5	5	10	6

Description:

The contents of general register rs is divided by the contents of general register rt, treating both operands as two's complement values.

No integer overflow exception occurs under any circumstances.

The results of this operation are undefined when the divisor is zero.

Typically this instruction will be followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

MIPS I/II Operation:

T-2:	LO ← undefined HI ← undefined
T-1:	LO ← undefined HI ← undefined
T:	LO ← GPR[rs] div GPR[rt] HI ← GPR[rs] mod GPR[rt]

MIPS III operation:

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 $GPR[rt]_{31..0}$ $LO \leftarrow (q_{31})^{32} \parallel q_{31..0}$ $HI \leftarrow (r_{31})^{32} \parallel r_{31..0}$

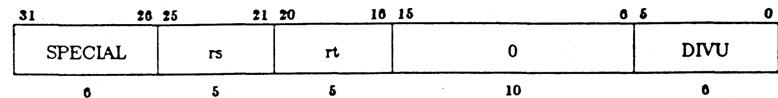
Exceptions:

none

DIVIDE UNSIGNED

Format:

DIVU rs,rt



Description:

The contents of general register rs is divided by the contents of general register rt, treating both operands as unsigned values.

No integer overflow exception occurs under any circumstances.

The results of this operation are undefined when the divisor is zero.

Typically this instruction will be followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

MIPS I/II operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: LO ← (0 || GPR[rs]) div (0 || GPR[rt])
HI ← (0 || GPR[rs]) mod (0 || GPR[rt])

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: $q \leftarrow (0 \,||\, GPR[rs]_{31..0})$ div $(0 \,||\, GPR[rt]_{31..0})$
$r \leftarrow (0 \,||\, GPR[rs]_{31..0})$ mod $(0 \,||\, GPR[rt]_{31..0})$
$LO \leftarrow (q_{31})^{32} \,||\, q_{31..0}$
$HI \leftarrow (r_{31})^{32} \,||\, r_{31..0}$

Exceptions:

none

DOUBLEWORD MULTIPLY

Format:

DMULT rs,rt

31 26	25 21	20 16	15 6	5 0
SPECIAL	rs	rt	0	DMULT
6	5	5	10	6

Description:

The contents of general register rs and the contents of general register rt are multipled, treating both operands as two's complement values.

No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: t ← GPR[rs] * GPR[rt]
LO ← $t_{63..0}$
HI ← $t_{127..64}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD MULTIPLY UNSIGNED

Format:

DMULTU rs,rt

31 26	25 21	20 16	15 6	5 0
SPECIAL	rs	rt	0	DMULTU
6	5	5	10	6

Description:

The contents of general register rs and the contents of general register rt are multipled, treating both operands as unsigned values.

No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: t ← (0 || GPR[rs]) * (0 || GPR[rt])
LO ← $t_{63..0}$
HI ← $t_{127..64}$

Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD DIVIDE

Format:

DDIV rs,rt

31 — 26	25 — 21	20 — 16	15 — 6	5 — 0
SPECIAL	rs	rt	0	DDIV
6	5	5	10	6

Description:

The contents of general register rs is divided by the contents of general register rt, treating both operands as two's complement values.

No integer overflow exception occurs under any circumstances.

The results of this operation are undefined when the divisor is zero.

Typically this instruction will be followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T-2: LO ← undefined
HI ← undefined

T-1: LO ← undefined
HI ← undefined

T: LO ← GPR[rs] div GPR[rt]
HI ← GPR[rs] mod GPR[rt]

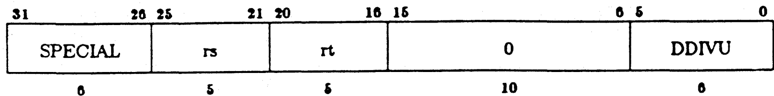
Exceptions:

Reserved instruction exception (MIPS I and II only)

DOUBLEWORD DIVIDE UNSIGNED

Format:

DDIVU rs,rt



Description:

The contents of general register rs is divided by the contents of general register rt, treating both operands as unsigned values.

No integer overflow exception occurs under any circumstances.

The results of this operation are undefined when the divisor is zero.

Typically this instruction will be followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

This instruction is only valid when rd = 0.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more other instructions.

This instruction is defined only in MIPS III implementations and causes a reserved instruction exception on MIPS I and MIPS II processors.

MIPS III operation:

T-2:	LO ← undefined HI ← undefined
T-1:	LO ← undefined HI ← undefined
T:	LO ← (0 || GPR[rs]) div (0 || GPR[rt]) HI ← (0 || GPR[rs]) mod (0 || GPR[rt])

Exceptions:

Reserved instruction exception (MIPS I and II only)

MOVE FROM HI

Format:

MFHI rd

31 26	25 16	15 11	10 6	5 0
SPECIAL	0	rd	0	MFHI
6	10	5	5	6

Description:

The contents of special register HI is loaded into general register rd.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the HI register: MULT, MULTU, DIV, DIVU, MTHI.

Operation:

T: GPR[rd] ← HI

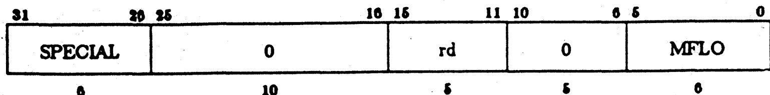
Exceptions:

none

MOVE FROM LO

Format:

MFLO rd



Description:

The contents of special register LO is loaded into general register rd.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the LO register: MULT, MULTU, DIV, DIVU, MTLO.

Operation:

T: GPR[rd] ← LO

Exceptions:

none

MOVE TO HI

Format:

MTHI rs

31 ... 26	25 ... 21	20 ... 6	5 ... 0
SPECIAL	rs	0	MTHI
6	5	15	6

Description:

The contents of general register rs is loaded into special register HI.

This instruction is only valid when rd = 0.

If a MTHI operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register LO are undefined.

Operation:

T-2: HI ← undefined

T-1: HI ← undefined

T: HI ← GPR[rs]

Exceptions:

none

MOVE TO LO

Format:

MTLO rs

31 26	25 21	20 6	5 0
SPECIAL	rs	0	MTLO
6	5	15	6

Description:

The contents of general register rs is loaded into special register LO.

This instruction is only valid when rd = 0.

If a MTLO operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register HI are undefined.

Operation:

T-2: LO ← undefined

T-1: LO ← undefined

T: LO ← GPR[rs]

Exceptions:

none

2.7. Jump and Branch Instructions

All jump and branch instructions are implemented with a delay of exactly one instruction. That is, the instruction immediately following a jump or branch (i.e. occupying the "delay slot") is always executed while the target instruction is being fetched from storage. It is not valid for a delay slot to be occupied itself by a jump, or branch instruction; however, this error is not detected, and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, hardware will set the EPC register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are re-executed.

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

Instructions must be word-aligned, and a "jump register" or "jump and link register" that uses a register whose low-order 2 bits are non-zero will cause an address error exception to occur when the jump target instruction is fetched.

2.7.1. Jump Instructions

Instruction Format:

31 — 26	25 — 0
op	target
6	26

where:

op is a 6-bit operation code
target is a 26-bit jump target address

Description	op
Jump	J
Jump and Link	JAL

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	rs	rt	rd	sa	function
6	5	5	5	5	6

where:

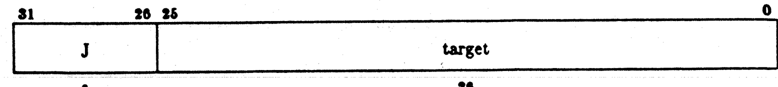
SPECIAL	is the 6-bit Special instruction opcode
rs	is a 5-bit target register field
rt, sa	are not used and should be zero
rd	is a 5-bit destination register field
function	is a 6-bit function field

Description	function
Jump to Register	JR
Jump and Link Register	JALR

JUMP

Format:

J target



Description:

The 26-bit target address is shifted left two bits, combined with the high order 4 bits of the address of the delay slot, and unconditionally jumped to, with a delay of one instruction.

MIPS I/II operation:

T: temp ← target

T+1: PC ← $PC_{31..28}$ || temp || 0^2

MIPS III operation:

T: temp ← target

T+1: PC ← $PC_{63..28}$ || temp || 0^2

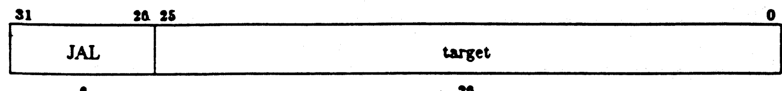
Exceptions:

none

JUMP AND LINK

Format:

JAL target



Description:

The 26-bit target address is shifted left two bits, combined with the high order 4 bits of the address of the delay slot, and unconditionally jumped to, with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, r31.

MIPS I/II operation:

T: temp ← target
GPR[31] ← PC + 8

T+1: PC ← $PC_{31..28}$ || temp || 0^2

MIPS III operation:

T: temp ← target
GPR[31] ← PC + 8

T+1: PC ← $PC_{63..28}$ || temp || 0^2

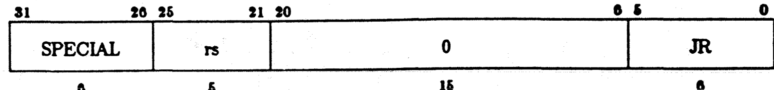
Exceptions:

none

JUMP REGISTER

Format:

JR rs



Description:

The contents of general register rs are unconditionally jumped to, with a delay of one instruction.

This instruction is only valid when rd = 0.

Operation:

T: temp ← GPR[rs]

T+1: PC ← temp

Exceptions:

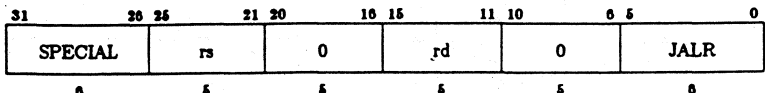
none

JUMP AND LINK REGISTER

Format:

JALR rs

JALR rd, rs



Description:

The contents of general register rs are unconditionally jumped to, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register rd. The default value of rd, if omitted in the assembly language instruction is 31.

Register specifiers rs and rd may not be equal, because such an instruction does not have the same effect when re-executed (i.e. is not idempotent). However, an attempt to execute this instruction is *not* trapped; the result of executing such an instruction is undefined.

Operation:

```
T:      temp ← GPR[rs]
        GPR[rd] ← PC + 8

T+1:    PC ← temp
```

Exceptions:

none

2.7.2. Branch on Condition

Instruction Formats:

31 ... 26	25 ... 21	20 ... 16	15 ... 0
op	rs	rt	offset
6	5	5	16

where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier
rt	is a 5-bit source register specifier
offset	is a 16-bit signed offset

Description	op
Branch on Equal	BEQ
Branch on Not Equal	BNE
Branch on Less Than or Equal to Zero	BLEZ
Branch on Greater Than Zero	BGTZ
Branch on Equal Likely	BEQL
Branch on Not Equal Likely	BNEL
Branch on Less Than or Equal to Zero Likely	BLEZL
Branch on Greater Than Zero Likely	BGTZL

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM	rs	rt	offset
6	5	5	16

where:

REGIMM	is the 6-bit REGIMM opcode
rs	is a 5-bit source register specifier
rt	is a 5-bit sub-operation opcode
offset	is a 16-bit signed offset

Description	rt
Branch on Less Than Zero	BLTZ
Branch on Greater Than or Equal to Zero	BGEZ
Branch on Less Than Zero and Link	BLTZAL
Branch on Greater Than or Equal to Zero and Link	BGEZAL
Branch on Less Than Zero Likely	BLTZL
Branch on Greater Than or Equal to Zero Likely	BGEZL
Branch on Less Than Zero and Link Likely	BLTZALL
Branch on Greater Than or Equal to Zero and Link Likely	BGEZALL

BRANCH ON EQUAL

Format:

BEQ rs,rt,offset

31–26	25–21	20–16	15–0
BEQ	rs	rt	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

T: target ← $(\text{offset}_{15})^{14}$ || offset || 0^2
condition ← (GPR[rs] = GPR[rt])

T+1: if condition then
PC ← PC + target
endif

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← (GPR[rs] = GPR[rt])

T+1: if condition then
PC ← PC + target
endif

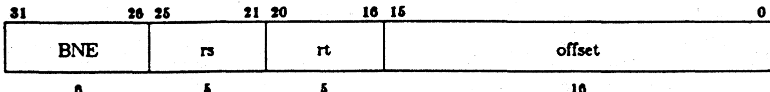
Exceptions:

none

BRANCH ON NOT EQUAL

Format:

BNE rs,rt,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are not equal, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

T: target ← $(offset_{15})^{14}$ || offset || 0^2
condition ← (GPR[rs] ≠ GPR[rt])

T+1: if condition then
PC ← PC + target
endif

MIPS III operation:

T: target ← $(offset_{15})^{38}$ || offset || 0^2
condition ← (GPR[rs] ≠ GPR[rt])

T+1: if condition then
PC ← PC + target
endif

Exceptions:

none

BRANCH ON LESS THAN OR EQUAL TO ZERO

Format:

BLEZ rs,offset

31 … 26	25 … 21	20 … 16	15 … 0
BLEZ	rs	0	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs is compared to zero. If the contents of general register rs has the sign bit set, or is equal to zero, the target address is branched to, with a delay of one instruction.

This instruction is only valid when rt = 0.

MIPS I/II operation:

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
endif

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← $(\text{GPR}[rs]_{63} = 1)$ or $(\text{GPR}[rs] = 0^{64})$

T+1: if condition then
PC ← PC + target
endif

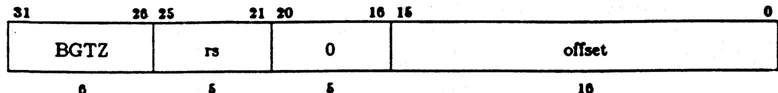
Exceptions:

none

BRANCH ON GREATER THAN ZERO

Format:

BGTZ rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs is compared to zero. If the contents of general register rs has the sign bit cleared and is not equal to zero, the target address is branched to, with a delay of one instruction.

This instruction is only valid when rt = 0.

MIPS I/II operation:

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

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || 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

BRANCH ON LESS THAN ZERO

Format:

BLTZ rs,offset

31 26	25 21	20 16	15 0
REGIMM	rs	BLTZ	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs has the sign bit set, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

T: target ← $(\text{offset}_{15})^{14}$ || offset || 0^2
condition ← $(\text{GPR}[rs]_{31} = 1)$

T+1: if condition then
PC ← PC + target
endif

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← $(\text{GPR}[rs]_{63} = 1)$

T+1: if condition then
PC ← PC + target
endif

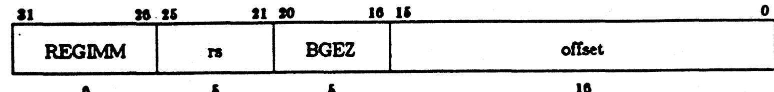
Exceptions:

none

BRANCH ON GREATER THAN OR EQUAL TO ZERO

Format:

BGEZ rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs has the sign bit cleared, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

T: target $\leftarrow$ $(\text{offset}_{15})^{14}$ || offset || 0^2
condition $\leftarrow$ $(\text{GPR}[rs]_{31} = 0)$

T+1: if condition then
PC $\leftarrow$ PC + target
endif

MIPS III operation:

T: target $\leftarrow$ $(\text{offset}_{15})^{38}$ || offset || 0^2
condition $\leftarrow$ $(\text{GPR}[rs]_{63} = 0)$

T+1: if condition then
PC $\leftarrow$ PC + target
endif

Exceptions:

none

BRANCH ON LESS THAN ZERO AND LINK

Format:

BLTZAL rs,offset

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM	rs	BLTZAL	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs has the sign bit set, the target address is branched to, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however.

MIPS I/II operation:

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

MIPS III operation:

T: $target \leftarrow (offset_{15})^{38} \,||\, offset \,||\, 0^2$
$condition \leftarrow (GPR[rs]_{63} = 1)$
$GPR[31] \leftarrow PC + 8$

T+1: if condition then
$PC \leftarrow PC + target$
endif

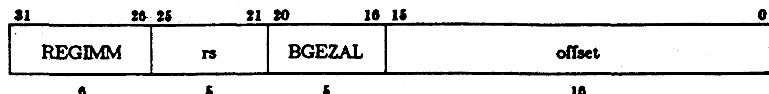
Exceptions:

none

BRANCH ON GREATER THAN OR EQUAL TO ZERO AND LINK

Format:

BGEZAL rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs has the sign bit cleared, the target address is branched to, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however.

MIPS I/II operation:

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
endif

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← $(\text{GPR}[rs]_{63} = 0)$
GPR[31] ← PC + 8

T+1: if condition then
PC ← PC + target
endif

Exceptions:

none

BRANCH ON EQUAL LIKELY

Format:

BEQL rs,rt,offset

31 … 26	25 … 21	20 … 16	15 … 0
BEQL	rs	rt	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors.

MIPS II operation:

```
T:      target ← (offset_15)^14 || offset || 0^2
        condition ← (GPR[rs] = GPR[rt])

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

MIPS III operation:

```
T:      target ← (offset_15)^38 || offset || 0^2
        condition ← (GPR[rs] = GPR[rt])

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

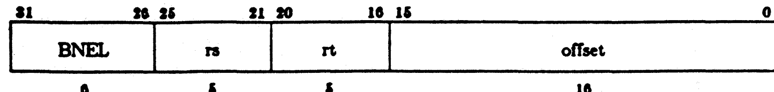
Exceptions:

Reserved Instruction exception (MIPS I only)

BRANCH ON NOT EQUAL LIKELY

Format:

BNEL rs,rt,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are not equal, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors.

MIPS II operation:

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

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← (GPR[rs] ≠ GPR[rt])

T+1: if condition then
PC ← PC + target
else
NullifyCurrentInstruction
endif

Exceptions:

Reserved Instruction exception (MIPS I only)

BRANCH ON LESS THAN OR EQUAL TO ZERO LIKELY

Format:

BLEZL rs,offset

31 — 26	25 — 21	20 — 16	15 — 0
BLEZL	rs	0	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs is compared to zero. If the contents of general register rs has the sign bit set, or is equal to zero, the target address is branched to, with a delay of one instruction.

This instruction is only valid when rt = 0.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors.

MIPS II operation:

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
else
NullifyCurrentInstruction
endif

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← ($GPR[rs]_{63}$ = 1) or (GPR[rs] = 0^{64})

T+1: if condition then
PC ← PC + target
else
NullifyCurrentInstruction
endif

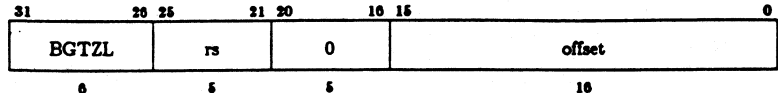
Exceptions:

Reserved Instruction exception (MIPS I only)

BRANCH ON GREATER THAN ZERO LIKELY

Format:

BGTZL rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs is compared to zero. If the contents of general register rs has the sign bit cleared and is not equal to zero, the target address is branched to, with a delay of one instruction.

This instruction is only valid when rt = 0.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors.

MIPS II operation:

```
T:      target ← (offset15)^14 || offset || 0^2
        condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

MIPS III operation:

```
T:      target ← (offset15)^38 || offset || 0^2
        condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 0^64)

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

Exceptions:

Reserved Instruction exception (MIPS I only)

BRANCH ON LESS THAN ZERO LIKELY

Format:

BLTZL rs,offset

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM	rs	BLTZL	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs has the sign bit set, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

```
T:      target ← (offset_15)^14 || offset || 0^2
        condition ← (GPR[rs]_31 = 1)

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

MIPS III operation:

```
T:      target ← (offset_15)^38 || offset || 0^2
        condition ← (GPR[rs]_63 = 1)

T+1:    if condition then
           PC ← PC + target
        else
           NullifyCurrentInstruction
        endif
```

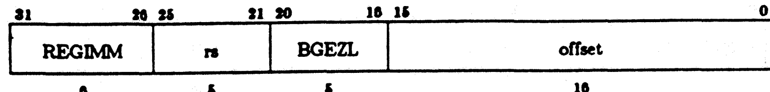
Exceptions:

None

BRANCH ON GREATER THAN OR EQUAL TO ZERO LIKELY

Format:

BGEZL rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs has the sign bit cleared, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

```
T:     target ← (offset15)^14 || offset || 0^2
       condition ← (GPR[rs]31 = 0)

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

MIPS III operation:

```
T:     target ← (offset15)^46 || offset || 0^2
       condition ← (GPR[rs]63 = 0)

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

Exceptions:

None

BRANCH ON LESS THAN ZERO AND LINK LIKELY

Format:

BLTZALL rs,offset

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM	rs	BLTZALL	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs has the sign bit set, the target address is branched to, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

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

MIPS III operation:

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2
condition ← $(\text{GPR}[rs]_{63} = 1)$
GPR[31] ← PC + 8

T+1: if condition then
PC ← PC + target
else
NullifyCurrentInstruction
endif

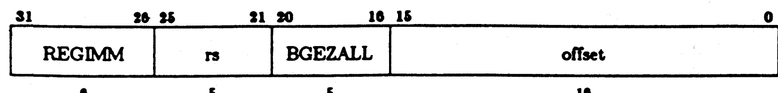
Exceptions:

None

BRANCH ON GREATER THAN OR EQUAL TO ZERO AND LINK LIKELY

Format:

BGEZALL rs,offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs has the sign bit cleared, the target address is branched to, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: target ← $(offset_{15})^{14}$ || offset || 0^2
condition ← $(GPR[rs]_{31} = 0)$
GPR[31] ← PC + 8

T+1: if condition then
PC ← PC + target
else
NullifyCurrentInstruction
endif

MIPS III operation:

T: target ← $(offset_{15})^{38}$ || offset || 0^2
condition ← $(GPR[rs]_{63} = 0)$
GPR[31] ← PC + 8

T+1: if condition then
PC ← PC + target
else
NullifyCurrentInstruction
endif

Exceptions:

None

2.8. Exception Instructions

Exception instructions have as their sole purpose the causing of a branch to the general exception handling vector. In the case of the system call or breakpoint instructions, this branch in unconditional; for the trap instructions, this branch is conditional upon the result of a comparison either of the contents of two general registers or one general register and an immediate value.

Instruction Formats:

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	rs	rt	rd	sa	function
6	5	5	5	5	6

where:

SPECIAL	is the 6-bit Special opcode
rs, rt	are a 5-bit source register specifiers
rd, sa	are not used
function	is a 6-bit function opcode

Description	function
System Call Break Synchronize	SYSCALL BREAK
Trap if Greater Than or Equal Trap if Greater Than or Equal Unsigned Trap if Less Than Trap if Less Than Unsigned Trap if Equal Trap if Not Equal	TGE TGEU TLT TLTU TEQ TNE

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM	rs	rt	immediate
6	5	5	16

where:

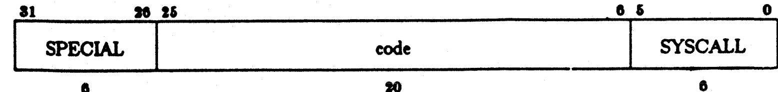
REGIMM is the 6-bit REGIMM operation code
rs is a 5-bit register source
rt is a 5-bit sub-operation code
immediate is a 16-bit immediate operand

Description	rt
Trap if Greater Than or Equal Immediate	TGEI
Trap if Greater Than or Equal Unsigned Immediate	TGEIU
Trap if Less Than Immediate	TLTI
Trap if Less Than Unsigned Immediate	TLTIU
Trap if Equal Immediate	TEQI
Trap if Not Equal Immediate	TNEI

SYSTEM CALL

Format:

SYSCALL



Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

T: SystemCallException

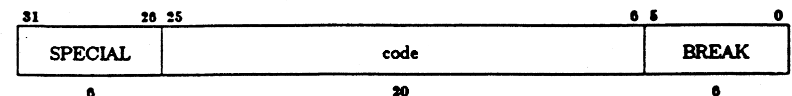
Exceptions:

System Call exception

BREAKPOINT

Format:

BREAK



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:

T: BreakpointException

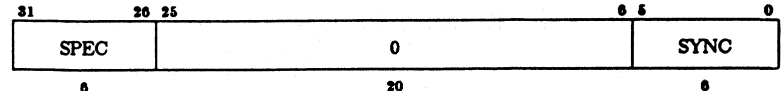
Exceptions:

Breakpoint exception

SYNC

Format:

SYNC



Description:

The SYNC instruction ensures that loads and stores fetched before the instruction complete before any loads or stores after this instruction start. Use of the SYNC instruction to serialize certain memory references may be required in multiprocessor environment for proper synchronization.

For example:

processor A			processor B	
SW	R1, DATA	1:	LW	R2, FLAG
LI	R2, 1		BEQ	R2, R0, 1B
SYNC			NOP	
SW	R2, FLAG		SYNC	
			LW	R1, DATA

The SYNC in processor A prevents DATA being written after FLAG, which could cause processor B to read stale data. The SYNC in processor B prevents DATA from being read before FLAG, which could likewise result in reading stale data.

For processors which only execute loads and stores in order with respect to shared memory, this instruction is a NOP.

The LL, LLD, SC, and SCD instructions implicitly perform a SYNC.

This instruction is not valid on MIPS I processors, and causes a reserved instruction exception. This instruction is allowed in user-mode.

Operation:

T: SyncOperation()

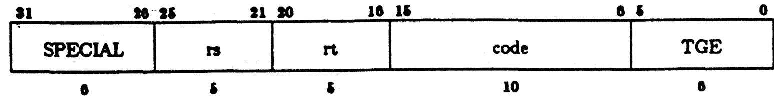
Exceptions:

Reserved instruction exception (MIPS I only)

TRAP IF GREATER THAN OR EQUAL

Format:

TGE rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are greater than or equal to the contents of general register rt, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

This instruction causes a reserved instruction exception on MIPS I processors.

Operation:

```
T:    if GPR[rs] ≥ GPR[rt]
         TrapException
      end
```

Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF GREATER THAN OR EQUAL UNSIGNED

Format:

TGEU rs,rt

31 26	25 21	20 16	15 6	5 0
SPECIAL	rs	rt	code	TGEU
6	5	5	10	6

Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are greater than or equal to the contents of general register rt, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

This instruction causes a reserved instruction exception on MIPS I processors.

Operation:

```
T:    if (0 || GPR[rs]) ≥ (0 || GPR[rt])
         TrapException
      end
```

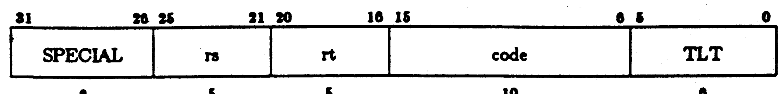
Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF LESS THAN

Format:

TLT rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are less than the contents of general register rt, 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.

This instruction causes a reserved instruction exception for the MIPS I processors.

Operation:

```
T:    if GPR[rs] < GPR[rt]
          TrapException
      end
```

Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF LESS THAN UNSIGNED

Format:

TLTU rs,rt

31 — 26	25 — 21	20 — 16	15 — 6	5 — 0
SPECIAL	rs	rt	code	TLTU
6	5	5	10	6

Description:

The contents of general register rt is subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are less than the contents of general register rt, 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.

This instruction causes a reserved instruction exception on MIPS I processors.

Operation:

```
T:    if (0 || GPR[rs]) < (0 || GPR[rt])
          TrapException
      end
```

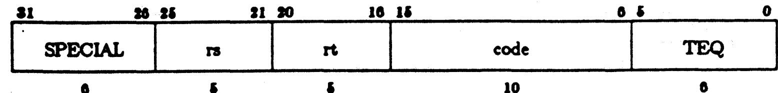
Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF EQUAL

Format:

TEQ rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs. If the contents of general register rs are equal to the contents of general register rt, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

This instruction causes a reserved instruction exception on MIPS I processors.

Operation:

```
T:    if GPR[rs] = GPR[rt]
          TrapException
      end
```

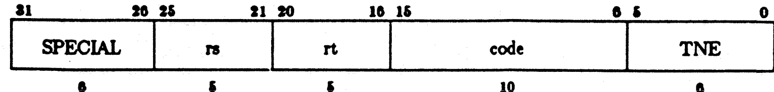
Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF NOT EQUAL

Format:

TNE rs,rt



Description:

The contents of general register rt is subtracted from the contents of general register rs. If the contents of general register rs are not equal to the contents of general register rt, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

This instruction causes a reserved instruction exception on MIPS I processors.

Operation:

```
T:      if GPR[rs] ≠ GPR[rt]
            TrapException
        end
```

Exceptions:

Trap exception
Reserved Instruction exception (MIPS I only)

TRAP IF GREATER THAN OR EQUAL IMMEDIATE

Format:

TGEI rs,immediate

31 ... 26	25 ... 21	20 ... 16	15 ... 0
REGIMM	rs	TGEI	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are greater than or equal to the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: if GPR[rs] $\geq$ $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$
    TrapException
end

MIPS III operation:

T: if GPR[rs] $\geq$ $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$
    TrapException
end

Exceptions:

Trap exception

TRAP IF GREATER THAN OR EQUAL UNSIGNED IMMEDIATE

Format:

TGEIU rs,immediate

31 ... 26	25 ... 21	20 ... 16	15 ... 0
REGIMM	rs	TGEIU	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are greater than or equal to the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: if $(0 \,||\, GPR[rs]) \geq (0 \,||\, (immediate_{15})^{16} \,||\, immediate_{15..0})$
TrapException
end

MIPS III operation:

T: if $(0 \,||\, GPR[rs]) \geq (0 \,||\, (immediate_{15})^{48} \,||\, immediate_{15..0})$
TrapException
end

Exceptions:

Trap exception

TRAP IF LESS THAN IMMEDIATE

Format:

TLTI rs,immediate

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM	rs	TLTI	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are less than the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

```
T:    if GPR[rs] < (immediate_15)^16 || immediate_15..0
         TrapException
      end
```

MIPS III operation:

```
T:    if GPR[rs] < (immediate_15)^48 || immediate_15..0
         TrapException
      end
```

Exceptions:

Trap exception

TRAP IF LESS THAN UNSIGNED IMMEDIATE

Format:

TLTIU rs,immediate

31–26	25–21	20–16	15–0
REGIMM	rs	TLTIU	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are less than the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: if $(0 \| GPR[rs]) < (0 \| (immediate_{15})^{16} \| immediate_{15..0})$
TrapException
end

MIPS III operation:

T: if $(0 \| GPR[rs]) < (0 \| (immediate_{15})^{48} \| immediate_{15..0})$
TrapException
end

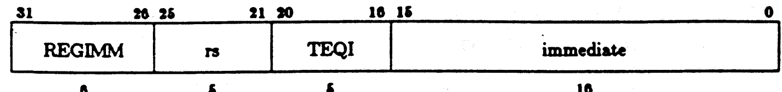
Exceptions:

Trap exception

TRAP IF EQUAL IMMEDIATE

Format:

TEQI rs,immediate



Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. If the contents of general register rs are equal to the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: if GPR[rs] = $(immediate_{15})^{16}$ || $immediate_{15..0}$
TrapException
end

MIPS III operation:

T: if GPR[rs] = $(immediate_{15})^{48}$ || $immediate_{15..0}$
TrapException
end

Exceptions:

Trap exception

TRAP IF NOT EQUAL IMMEDIATE

Format:

TNEI rs,immediate

31 26	25 21	20 16	15 0
REGIMM	rs	TNEI	immediate
6	5	5	16

Description:

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. If the contents of general register rs are not equal to the sign-extended immediate, a trap exception occurs.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T: if GPR[rs] $\neq$(immediate$_{15}$)^{16} || immediate$_{15..0}$
TrapException
end

MIPS III operation:

T: if GPR[rs] $\neq$(immediate$_{15}$)^{48} || immediate$_{15..0}$
TrapException
end

Exceptions:

Trap exception

2.9. Coprocessor Instructions

The MIPS architecture provides four coprocessor units, or classes. Coprocessors are alternate execution units, which have separate register files from the processor.

MIPS I and II coprocessors have 2 register spaces, each of 32 registers of 32 bits. MIPS III implementations are 32 registers of 64 bits. The MIPS II subset of MIPS III does not use the odd locations of the register file. The MIPS II odd numbered registers reference the upper 32 bits of the MIPS III even register.

The first space, "coprocessor general registers," may be directly loaded from memory and stored into memory, and may be transferred between the coprocessor and processor. The second, "coprocessor control registers," may only be directly transferred between the coprocessor and processor. Coprocessor instructions may alter registers in either space.

Normally, by convention, coprocessor control register 0 is interpreted as a coprocessor implementation and revision register. However, the system control coprocessor uses coprocessor general register 15 for the processor/coprocessor implementation and revision register.

The low-order byte (bits 7..0) is interpreted as a coprocessor unit revision number. The second byte (bits 15..8) is interpreted as a coprocessor unit implementation descriptor. The contents of the high-order halfword of the register is not defined.

The revision number is a value of the form $y.x$ where y is a major revision number in bits 7..4 and x is a minor revision number in bits 3..0.

The revision number can distinguish some chip revisions. However, MIPS is free to change this register at any time and does not guarantee that changes to its chips will necessarily change the revision register, or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number to characterize the chip.

2.9.1. Coprocessor Unit and Implementation code Assignments

Coprocessor units are currently assigned as follows:

class		implementation	
number	description	number	description
0	System Control Coprocessor	0x01	MIPS R2000 CPU (MIPS I)
		0x02	MIPS R3000 CPU (MIPS I)
		0x03	MIPS R6000 CPU (MIPS II)
		0x04	MIPS R4000 CPU (MIPS II)
		0x05	LSI Logic xxxx (MIPS I)
		0x06	MIPS R6000A CPU (MIPS II)
1	Floating-point Coprocessor	0x00	software
		0x01	MIPS R2360 FPC (MIPS I)
		0x02	MIPS R2010 FPC (MIPS I)
		0x03	MIPS R3010 FPC (MIPS I)
		0x04	MIPS R6010 FPC (MIPS II)
		0x05	MIPS R4010 FPC (MIPS II)
		0x06	LSI Logic xxxx (MIPS I)
2	reserved		
3	reserved		

2.9.2. Coprocessor Load and Store

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 0
op	base	rt	offset
6	5	5	16

where:

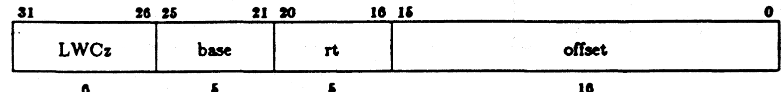
- op is the 6-bit operation code
- base is the 5-bit base register
- rt is the 5-bit source (for stores) or destination (for loads)
- offset is the 16-bit immediate offset

Description	op
Load Word to Coprocessor-z Store Word from Coprocessor-z	LWCz SWCz
Load Doubleword to Coprocessor-z Store Doubleword from Coprocessor-z	LDCz SDCz

LOAD WORD TO COPROCESSOR

Format:

LWCz rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into coprocessor register rt of coprocessor unit z.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is not valid for use with coprocessor unit 0.

In MIPS I implementations, the contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$byte \leftarrow vAddr_{1..0}$
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)

T+1: COPzLW (byte, rt, mem)

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$byte \leftarrow vAddr_{1..0}$
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
COPzLW (byte, rt, mem)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } (ReverseEndian \| 0^2))$
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
$byte \leftarrow vAddr_{2..0} \text{ xor } (BigEndianCPU \| 0^2)$
COPzLW (byte, rt, mem)

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

LOAD DOUBLEWORD TO COPROCESSOR

Format:

LDCz rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LDCz	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded into coprocessor register rt and rt+1 of coprocessor unit z.

If any of the three least significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid on MIPS I processors and causes a reserved instruction exception. In MIPS II implementations this instruction is undefined, when the least significant bit of rt is non-zero. On MIPS III processors, all values of the rt field are valid, as there are 32 64-bit registers.

This instruction is not valid for use with coprocessor unit 0.

The processor architecture does not define the ordering of the registers rt and rt+1 with respect to this instruction. Coprocessors may define the ordering as befits the application, and may choose to load either rt or rt+1 with the most-significant data loaded from memory and load the other register with the least-significant data. However, the ordering must be the same as selected for the SDCz instruction.

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
COPzLD (rt, mem)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
COPzLD (rt, mem)

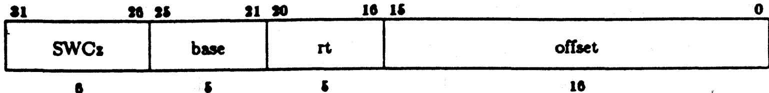
Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception
Reserved instruction exception (MIPS I only)

STORE WORD FROM COPROCESSOR

Format:

SWCz rt,offset(base)



Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of coprocessor register rt of coprocessor unit z are stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction is not valid for use with coprocessor unit 0.

MIPS I/II operation:

T: vAddr ← $((\text{offset}_{15})^{16} \parallel \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
byte ← $\text{vAddr}_{1..0}$
data ← COPzSW (byte, rt)
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \parallel \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \parallel (\text{pAddr}_{2..0}$ xor (ReverseEndian $\parallel 0^2$))
byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU $\parallel 0^2$)
data ← COPzSW (byte, rt)
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception

STORE DOUBLEWORD FROM COPROCESSOR

Format:

SDCz rt,offset(base)

31 … 26	25 … 21	20 … 16	15 … 0
SDCz	base	rt	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of coprocessor register rt of coprocessor unit z are stored at the memory location specified by the effective address.

If any of the three least significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid on MIPS I processors and causes a reserved instruction exception. In MIPS II implementations this instruction is undefined, when the least significant bit of rt is non-zero. On MIPS III processors, all values of the rt field are valid, as there are 32 64-bit registers.

This instruction is not valid for use with coprocessor unit 0.

The processor architecture does not define the ordering of the registers rt and rt+1 with respect to this instruction. Coprocessors may define the ordering as befits the application, and may choose to store either rt or rt+1 into the most-significant data stored into memory and store the other register into the least-significant data. However, the ordering must be the same as selected for the LDCz instruction.

MIPS II operation:

T: vAddr ← $((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← COPzSD (rt)
StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \,\|\, \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← COPzSD (rt)
StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception
Reserved instruction exception (MIPS I only)

2.9.3. Move To/From Coprocessor

Instruction Format:

31–26	25–21	20–16	15–11	10–0
COPz	sub	rt	rd	0
6	5	5	5	11

where:

COPz	is a 6-bit operation code and coprocessor unit number specifier
sub	is a 5-bit coprocessor sub-operation field
rt	is a 5-bit general register specifier
rd	is a 5-bit coprocessor register specifier

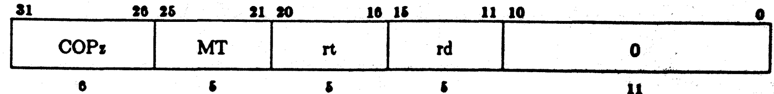
There are four opcodes of this type, one for each of the four coprocessor classes.

These coprocessor operations are moves between general and coprocessor registers of each class.

MOVE TO COPROCESSOR

Format:

MTCz rt,rd



Description:

The contents of general register rt are loaded into coprocessor register rd of coprocessor unit z.

MIPS I/II Operation:

T: data ← GPR[rt]

T+1: CPR[z, rd] ← data

MIPS III operation:

T: data ← $GPR[rt]_{31..0}$

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

MOVE FROM COPROCESSOR

Format:

MFCz rt,rd

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COPz	MF	rt	rd	0
6	5	5	5	11

Description:

The contents of coprocessor register rd of coprocessor unit z are loaded into general register rt.

MIPS I/II operation:

T: data ← CPR[z, rd]

T+1: GPR[rt] ← data

MIPS III operation:

T: if rd_0 = 0 then
  data ← CPR[z, $rd_{4..1}$ || $0]_{31..0}$
else
  data ← CPR[z, $rd_{4..1}$ || $0]_{63..32}$
endif

T+1: GPR[rt] ← $(data_{31})^{32}$ || data

Exceptions:

Coprocessor unusable exception

MOVE CONTROL TO COPROCESSOR

Format:

CTCz rt,rd

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COPz	CT	rt	rd	0
6	5	5	5	11

Description:

The contents of general register rt are loaded into coprocessor control register rd of coprocessor unit z.

This instruction is not valid for coprocessor unit 0.

MIPS I/II Operation:

T: data ← GPR[rt]

T+1: CCR[z, rd] ← data

MIPS III operation:

T: data ← GPR[rt]

T+1: CCR[z, rd] ← data

Exceptions:

Coprocessor unusable exception

MOVE CONTROL FROM COPROCESSOR

Format:

CFCz rt,rd

31–26	25–21	20–16	15–11	10–0
COPz	CF	rt	rd	0
6	5	5	5	11

Description:

The contents of coprocessor control register rd of coprocessor unit z are loaded into general register rt.

This instruction is not valid for coprocessor unit 0.

MIPS I/II operation:

T: data ← CCR[z, rd]

T+1: GPR[rt] ← data

MIPS III operation:

T: data ← $(CCR[z,rd]_{31})^{32}$ || CCR[z, rd]

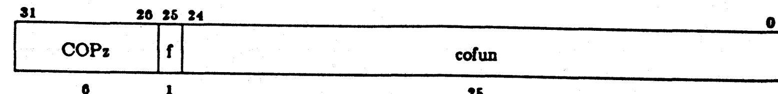
T+1: GPR[rt] ← data

Exceptions:

Coprocessor unusable exception

2.9.4. Coprocessor Operations

Instruction Format:



where:

COPz	is a 4-bit fixed field followed by a 2-bit code
f	is a 1-bit sub-operation specifier
cofun	is a 25-bit coprocessor function field

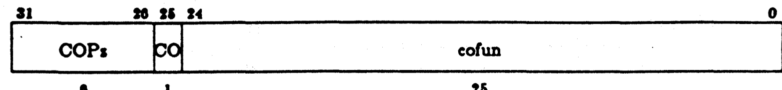
There are four opcodes of this type, one for each of the four coprocessor classes.

Details of the coprocessor operations are described in the Coprocessor Manuals.

COPROCESSOR OPERATION

Format:

COPz cofun



Description:

A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache/memory system.

Operation:

T: CoprocessorOperation (z, cofun)

Exceptions:

Coprocessor unusable exception
Coprocessor interrupt or Floating-Point Exception (R4000 coprocessor 1 only)

2.9.5. Branch on Coprocessor Condition

Instruction Format:

31 ... 26	25 ... 21	20 ... 16	15 ... 0
COPz	BC	br	offset
6	5	5	16

where:

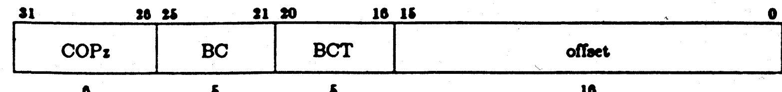
- COPz — is a 4-bit fixed field followed by a 2-bit code
- BC — is the 5-bit BC sub-opcode
- br — is a 5-bit branch condition specifier
- offset — is a 16-bit offset

Description	cond
Branch Coprocessor z True Branch Coprocessor z False	BCzT BCzF
Branch Coprocessor z True Likely Branch Coprocessor z False Likely	BCzTL BCzFL

BRANCH ON COPROCESSOR *z* TRUE

Format:

BC*z*T offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor *z*'s condition line, as sampled during the previous instruction, is true, the target address is branched to, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between a coprocessor instruction that changes the condition line and this instruction.

MIPS I/II operation:

T-1: condition ← COC[z]

T: target ← $(\text{offset}_{15})^{14}$ || offset || 0^2

T+1: if condition then
 PC ← PC + target
 endif

MIPS III operation:

T-1: condition ← COC[z]

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2

T+1: if condition then
 PC ← PC + target
 endif

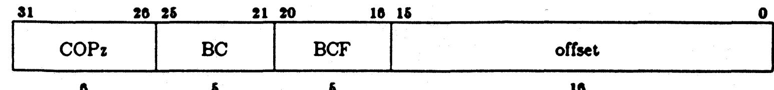
Exceptions:

Coprocessor unusable exception

BRANCH ON COPROCESSOR z FALSE

Format:

BCzF offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is false, the target address is branched to, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between a coprocessor instruction that changes the condition line and this instruction.

MIPS I/II operation:

T-1: condition ← not COC[z]

T: target ← $(offset_{15})^{14}$ || offset || 0^2

T+1: if condition then
    PC ← PC + target
  endif

MIPS III operation:

T-1: condition ← not COC[z]

T: target ← $(offset_{15})^{38}$ || offset || 0^2

T+1: if condition then
    PC ← PC + target
  endif

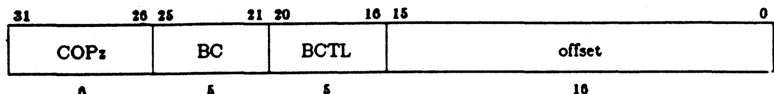
Exceptions:

Coprocessor unusable exception

BRANCH ON COPROCESSOR z TRUE LIKELY

Format:

BCzTL offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is true, the target address is branched to, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between a coprocessor instruction that changes the condition line and this instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

```
T-1:   condition ← COC[z]

T:     target ← (offset15)^14 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

MIPS III operation:

```
T-1:   condition ← COC[z]

T:     target ← (offset15)^38 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

Exceptions:

Coprocessor unusable exception

BRANCH ON COPROCESSOR z FALSE LIKELY

Format:

BCzFL offset

31 — 26	25 — 21	20 — 16	15 — 0
COPz	BC	BCFL	offset
6	5	5	16

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is false, the target address is branched to, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between a coprocessor instruction that changes the condition line and this instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid for MIPS I processors, but does not cause a reserved instruction exception.

MIPS II operation:

T-1: condition ← not COC[z]

T: target ← $(\text{offset}_{15})^{14}$ || offset || 0^2

T+1: if condition then
  PC ← PC + target
else
  NullifyCurrentInstruction
endif

MIPS III operation:

T-1: condition ← not COC[z]

T: target ← $(\text{offset}_{15})^{38}$ || offset || 0^2

T+1: if condition then
  PC ← PC + target
else
  NullifyCurrentInstruction
endif

Exceptions:

Coprocessor unusable exception

3. Floating Point Coprocessor

This chapter describes the MIPS floating-point coprocessor architecture, a set of instruction extensions to the MIPS R-Series processor architecture for the purpose of directly performing arithmetic operations on values in a floating-point representation.

As an architectural description, it provides a detailed specification of the instruction-level presentation of MIPS floating-point coprocessor designs, including instruction set, register formats, exception conditions and handling, but specifically excludes implementation details normally of interest only to hardware implementors.

Performance data which is pertinent to instruction scheduling and system performance prediction is included for current MIPS floating-point implementations.

Beyond the architectural constraints encompassed by this specification, substantial latitude is available to the implementor. General guidelines for efficient handling of concurrent execution and pipelining are presented for hardware designers. Hardware interface timing data is beyond the scope of this specification.

3.1. Floating-point coprocessor architecture

The MIPS floating-point coprocessor will be implemented in number of different ways. Initial and low-end implementations will rely on software floating-point, which use these floating-point instructions, but will implement them via a software exception handler. A board-level implementation of the MIPS floating-point coprocessor uses commercially-available floating-point VLSI devices to support a subset of the floating-point instructions in hardware, using hardware and/or software assistance to complete the instruction set. Given a VLSI implementation of the entire floating-point coprocessor, all of the capability of the floating-point instruction set may be implemented in hardware.

This MIPS floating-point coprocessor architecture permits each of these implementations to maintain a high degree of both upward and downward compatibility. Without requiring recompilation of user software, the floating-point coprocessor can be treated as a field-upgradable or replaceable unit. However, modest performance gains can be realized by tuning the generated code for particular implementations.

3.1.1. Instruction format

Floating-point instructions are mapped onto the MIPS coprocessor instructions, defining coprocessor unit number one (1) as the floating-point unit.

The basic operations are two-register and three-register operations, load and store operations to floating-point coprocessor registers, and moves between floating-point coprocessor and processor registers.

3.1.1.1. Floating-point loads, stores, and moves

All movement of data between the floating-point coprocessor and memory is accomplished by coprocessor load and store operations, which reference a single 32-bit word of the floating-point coprocessor general registers. These operations are unformated; no format conversions are performed and therefore no floating-point exceptions occur due to these operations.

MIPS II implementations provide coprocessor load and store operations, which reference a 64-bit doubleword directly, as a pair of floating-point coprocessor general registers. Like the single-word loads, these operations are unformated and perform no format conversions and incur no floating-point exceptions.

MIPS III implementations provide coprocessor load and store operations, which reference a 64-bit doubleword directly in a doubleword floating-point coprocessor general register. Like the single-word loads, these operations are unformated and perform no format conversions and incur no floating-point exceptions.

Data may also be directly moved between the floating-point coprocessor and the processor by move to coprocessor and move from coprocessor instructions. Like the floating-point load and store operations, move to/from operations perform no format conversions and never cause floating-point exceptions.

An additional set of 32 coprocessor registers are available, called "floating-point control registers" for which the only data movement operations supported are moves to and from processor general registers.

3.1.1.2. Floating-point operations

The floating-point unit's operation set includes floating-point add, subtract, multiply, divide, square root, convert between fixed-point and floating-point format, convert between floating-point formats, and floating-point compare. These operations satisfy IEEE Standard 754's requirements for accuracy. Specifically, these operations obtain a result which is identical to performing the result with infinite precision and then rounding to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for conversion functions, mixed-format operations are not provided.

3.1.1.3. Floating-point comparisons

The following table is derived from Table 4 in the IEEE standard, and in a systematic order, describes all twenty-six predicates named in the standard. An additional six predicates are listed, which round out the set of possible predicates, based on the conditions tested by a comparison.

Note that invalid operation exceptions occur only when comparisons include the characters "<" or ">," but not "?" in the *ad hoc* form of the predicate. The nomenclature for the predicates below leaves some types of comparisons unnamed, such as "test for equality and trap on unordered," because unlike the ordering tests, the tests for equality do not raise the invalid operation exception when one of the two operands is "Not a Number" and the "?" character is not present in the predicate.

The instructions specify two floating-point registers and one of the comparison conditions listed in the following table. Half of the conditions are generated directly by the comparison operation; the remainder are the logical negation of the first half.

Predicates			Relations				Invalid operation exception if unordered
Mnemonic	*ad hoc*	FORTRAN	greater than	less than	equal	unord-ered	
F	false		F	F	F	F	No
UN	?		F	F	F	T	No
EQ	=	.EQ.	F	F	T	F	No
UEQ	?=	.UE.	F	F	T	T	No
OLT	NOT(?>=)	.NOT. .UG.	F	T	F	F	No
ULT	?<	.UL.	F	T	F	T	No
OLE	NOT(?>)	.NOT. .UG.	F	T	T	F	No
ULE	?<=	.ULE.	F	T	T	T	No
OGT	NOT(?<=)	.NOT. .ULE.	T	F	F	F	No
UGT	?>	.UGT.	T	F	F	T	No
OGE	NOT(?<)	.NOT. .UL.	T	F	T	F	No
UGE	?>=	.UGE.	T	F	T	T	No
OGL	NOT(?=)		T	T	F	F	No
NEQ	NOT(=)	.NE.	T	T	F	T	No
OR	NOT(?)		T	T	T	F	No
T	true		T	T	T	T	No
SF			F	F	F	F	Yes
NGLE	NOT(<=>)	.NOT. .LEG.	F	F	F	T	Yes
SEQ			F	F	T	F	Yes
NGL	NOT(<>)	.NOT. .LG.	F	F	T	T	Yes
LT	<	.LT.	F	T	F	F	Yes
NGE	NOT(>=)	.NOT. .GE.	F	T	F	T	Yes
LE	<=	.LE.	F	T	T	F	Yes
NGT	NOT(>)	.NOT. .GT.	F	T	T	T	Yes
GT	>	.GT.	T	F	F	F	Yes
NLE	NOT(<=)	.NOT. .LE.	T	F	F	T	Yes
GE	>=	.GE.	T	F	T	F	Yes
NLT	NOT(<)	.NOT. .LT.	T	F	T	T	Yes
GL	<>	.LG.	T	T	F	F	Yes
SNE			T	T	F	T	Yes
GLE	<=>	.LEG.	T	T	T	F	Yes
ST			T	T	T	T	Yes

3.1.2. Coprocessor registers

The coprocessor architecture provides access to up to thirty-two registers through coprocessor load/store instructions and move to/from processor register instructions which address coprocessor general registers. The thirty-two coprocessor general registers are used to represent directly addressable floating-point registers.

An additional thirty-two words can be accessed through move to/from processor register instructions that address coprocessor control registers. These coprocessor control registers are used for saving and restoring other coprocessor state information.

In MIPS II implementations there are two views of the thirty-two coprocessor general registers: from the standpoint of the processor, which has no intrinsic representation of coprocessor registers, these registers are simply thirty-two single-word registers. From the standpoint of the floating-point coprocessor, collections of the single-word registers form floating-point registers, on which floating-point operations are performed.

In MIPS III implementations there are thirty-two, double-word, coprocessor general registers: from the standpoint of the processor, which has no intrinsic representation of coprocessor registers, these registers are simply

thirty-two double-word registers.

The thirty-two coprocessor control registers provide control and status registers by which exceptions and operational modes are controlled and made visible.

The MIPS I and MIPS II floating-point coprocessor contains sixteen floating-point registers, whereas the MIPS III floating-point coprocessor contains thirty-two floating-point registers. These are intended to provide a sufficient number of floating-point registers to support allocation of scalar floating-point values in registers and to permit overlapping and scheduling of floating-point operations. Each register can hold one value of a single-precision or double-precision format floating-point number.

In the MIPS I and MIPS II implementation this is achieved by combining the state held in two adjacent coprocessor general registers. Half of the registers can hold one value of an extended-precision or quad-precision format floating-point number, by using the state held in the adjacent floating-point register. Thus, these longer registers are formed from a collection of four adjacent coprocessor general registers. Single-precision floating-point operations (as opposed to loads, stores, and moves) leave the odd half of the result register undefined.

On MIPS II implementations that load or store doubleword quantities directly, these operations reference the sixteen floating-point registers in such a way that double-precision quantities are consistently represented in memory on both little-endian and big-endian machines. As will be seen in the detailed description of these instructions, this requires that doubleword stores be mapped differently for little-endian and bit-endian machines.

The coprocessor control registers contain a register for determining configuration and revision information for the coprocessor, and control and status registers for the coprocessor. These are primarily involved with diagnostic software, exception handling, state saving and restoring, and control of rounding modes.

IEEE Standard 754 requires handling five types of exceptions: Invalid operation, Division by zero, Overflow, Underflow, Inexact, which can be enabled and disabled individually. The standard also requires that status information be maintained for disabled exceptions to indicate whether or not these conditions have occurred.

The status registers indicate any operation that was started and then abandoned because of a floating-point exception. Status registers further indicate the cause of an internal exception.

Additional status bits provide implementation hooks for software implementation of some of the more complex functions of the floating-point coprocessor.

3.1.2.1. MIPS I and II Register layout - processor viewpoint

Regardless of the byte ordering of the MIPS processor, from the viewpoint of the MIPS processor, the coprocessor registers appear as follows:

FGR Number	Usage
0	FPR 0 (least)
1	FPR 0 (less)
2	FPR 0 (more) or 2 (least)
3	FPR 0 (most) or 2 (less)
4	FPR 4 (least)
5	FPR 4 (less)
6	FPR 4 (more) or 6 (least)
7	FPR 4 (most) or 6 (less)
8	FPR 8 (least)
9	FPR 8 (less)
10	FPR 8 (more) or 10 (least)
11	FPR 8 (most) or 10 (less)
12	FPR 12 (least)
13	FPR 12 (less)
14	FPR 12 (more) or 14 (least)
15	FPR 12 (most) or 14 (less)
16	FPR 16 (least)
17	FPR 16 (less)
18	FPR 16 (more) or 18 (least)
19	FPR 16 (most) or 18 (less)
20	FPR 20 (least)
21	FPR 20 (less)
22	FPR 20 (more) or 22 (least)
23	FPR 20 (most) or 22 (less)
24	FPR 24 (least)
25	FPR 24 (less)
26	FPR 24 (more) or 26 (least)
27	FPR 24 (most) or 26 (less)
28	FPR 28 (least)
29	FPR 28 (less)
30	FPR 28 (more) or 30 (least)
31	FPR 28 (most) or 30 (less)

FCR Number	Usage
0	Coprocessor implementation and revision register
1-29	Reserved
30	Exception instruction register (optional)
31	Rounding mode, cause, trap enables, and flags

Coprocessor registers may be read or written by instructions executing in either kernel or user mode.

3.1.2.2. MIPS I and II Register format - coprocessor viewpoint

Regardless of the byte ordering of the MIPS processor, internally to the floating-point coprocessor, the floating-point registers are assembled as:

FPR	FGR registers referenced			
number	(most)	(more)	(less)	(least)
0	FGR[3]	FGR[2]	FGR[1]	FGR[0]
2			FGR[3]	FGR[2]
4	FGR[7]	FGR[6]	FGR[5]	FGR[4]
6			FGR[7]	FGR[6]
8	FGR[11]	FGR[10]	FGR[9]	FGR[8]
10			FGR[11]	FGR[10]
12	FGR[15]	FGR[14]	FGR[13]	FGR[12]
14			FGR[15]	FGR[14]
16	FGR[19]	FGR[18]	FGR[17]	FGR[16]
18			FGR[19]	FGR[18]
20	FGR[23]	FGR[22]	FGR[21]	FGR[20]
22			FGR[23]	FGR[22]
24	FGR[27]	FGR[26]	FGR[25]	FGR[24]
26			FGR[27]	FGR[26]
28	FGR[31]	FGR[30]	FGR[29]	FGR[28]
30			FGR[31]	FGR[30]

All floating-point register numbers are even; odd register numbers are invalid (but not checked by the hardware). Each of the registers may contain values in a number of formats; When the register number is divisible by four it may contain a value in any defined format, while the remaining registers may only contain single- or double-precision floating-point and single-precision fixed point values.

Formats that require less than the full width of the floating-point register are right-justified. Extended-precision format, while it only has 80 significant bits, uses a 128-bit format to hold the value; the unused region is placed in the center of the format to simplify field packing and unpacking in an implementation that provides both extended and quad formats.

All fields of the floating-point register are packed into words, double-words, or quad-words so that load and store operations require no field-shuffling or tag bits.

3.1.2.3. MIPS III Register format - coprocessor viewpoint

Regardless of the byte ordering of the MIPS processor, internally to the floating-point coprocessor, the floating-point registers are assembled as:

FPR	FGR registers referenced			
number	(most)	(least)		
0	FGR[1]	FGR[0]		
1		FGR[1]		
2	FGR[3]	FGR[2]		
3		FGR[3]		
4	FGR[5]	FGR[4]		
5		FGR[5]		
30	FGR[31]	FGR[30]		
31		FGR[31]		

All 32 floating-point register numbers valid for single and double precision values. For extended and quad formats, the floating-point register numbers must be even.

Formats that require less than the full width of the floating-point register are right-justified. Extended-precision format, while it only has 80 significant bits, uses a 128-bit format to hold the value; the unused region is placed in the center of the format to simplify field packing and unpacking in an implementation that provides both extended and quad formats.

All fields of the floating-point register are packed into words, double-words, or quad-words so that load and store operations require no field-shuffling or tag bits.

3.1.2.3.1. Floating-point formats

Numbers in the single, double, extended, and quad floating-point formats are composed of three fields:

A 1-bit sign, s.

A biased exponent, $e=E+bias$, and

A fraction, $f=.b_1b_2 \cdots b_{p-1}$

Numbers in the extended floating-point format also explicitly represent the single, integer-weighted bit to the left of the binary point, i, which is "hidden" in the single and double precision formats.

The range of the unbiased exponent E includes every integer between two values E_{min} and E_{max} inclusive, and also two other reserved values: $E_{min}-1$ to encode ± 0 and denormalized numbers, and $E_{max}+1$ to encode $\pm\infty$ and NaNs. For single and double-precision formats, each representable nonzero numerical value has just one encoding.

Extended- and quad-precision formats permit unnormalized values, and so do not share this property with single and double precision formats. However, all extended- and quad-precision floating-point results will produce normalized values with unbiased exponents between the values E_{min} and E_{max} and will produce zero and denormalized values with unbiased exponents of $E_{min}-1$. This property ensures a single encoding for each representable nonzero numerical result.

For single-precision and double-precision formats, the value of a number, v, is determined by the following:

(1) if $E=E_{max}+1$ and $f\neq 0$, then v is NaN, regardless of s.

(2) if $E=E_{max}+1$ and $f=0$, then $v=(-1)^s\infty$.

(3) if $E_{min}\leq E\leq E_{max}$, then $v=(-1)^s 2^E(1.f)$.

(4) if $E=E_{min}-1$ and $f\neq 0$, then $v=(-1)^s 2^{E_{min}}(0.f)$.

(5) if $E=E_{min}-1$ and $f=0$, then $v=(-1)^s 0$.

For extended-precision and quad-precision formats, the value of a number, v, is determined by the following:

(1) if $E=E_{max}+1$ and $f\neq 0$, then v is NaN, regardless of s.

(2) if $E=E_{max}+1$ and $f=0$, then $v=(-1)^s\infty$.

(3) if $E_{min}\leq E\leq E_{max}$, then $v=(-1)^s 2^E(i.f)$.

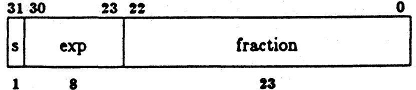
(4) if $E=E_{min}-1$, then $v=(-1)^s 2^{E_{min}}(i.f)$.

For all floating-point formats, if v is NaN, the most significant bit of f determines whether the value is a signaling or quiet NaN. v is a signaling NaN if the most significant bit of f is set; otherwise v is a quiet NaN.

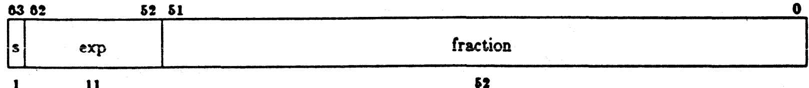
The format parameters in the preceding description have the following values:

Parameter	Format			
	Single	Double	Extended	Quad
p	24	53	64	112
E_{max}	+127	+1023	+16383	+16383
E_{min}	-126	-1022	-16382	-16382
exponent *bias*	+127	+1023	+16383	+16383
exponent width in bits	8	11	15	15
integer bit	hidden	hidden	1	1
fraction width in bits	24	53	63	111
format width in bits	32	64	80	128

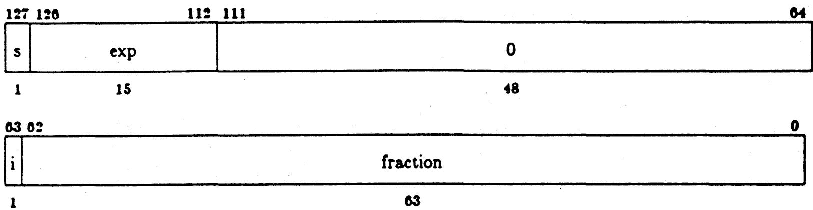
Single-precision floating-point format:



Double-precision floating-point format:

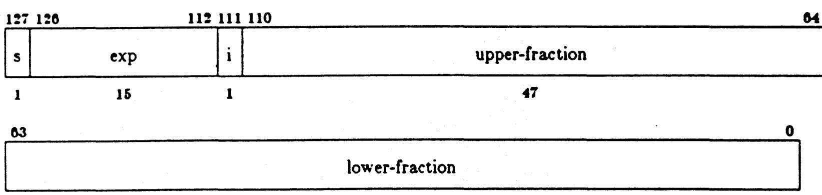


Extended-precision floating-point format:



Extended precision operations are undefined for data with bits 111..64 nonzero.

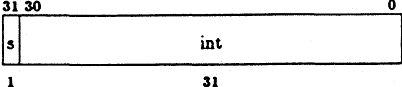
Quad-precision floating-point format:



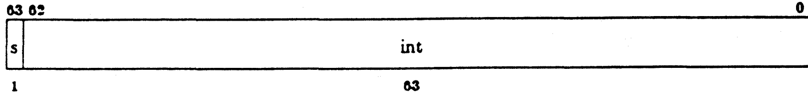
3.1.2.3.2. Binary fixed-point format

Binary fixed-point values are held in two's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set.

MIPS I/II single binary fixed-point format:



MIPS III single binary fixed-point format:



3.1.2.3.3. Implementation and revision register

Floating-point coprocessor control register zero contains values that can be used to identify the exact implementation and revision number of the floating-point unit. This information can be used to identify the set of operations supported by the coprocessor, or to assist in the complete testing or diagnosis of systems incorporating floating-point coprocessors.

Only the low-order two bytes of this register are defined. Bits 15 through 8 identify the implementation, and bits 7 through 0 identify the revision number, according to the following table:

implementation	
number	description
0	none (software)
1	R2360 (Multiple-chip VLSI, board-level controller)
2	R2010 (Single-chip VLSI)
3	R3010 (Single-chip VLSI)
4	R6010 (Multiple-chip VLSI with VLSI controller)
5	R4010 (Single-chip VLSI)

The revision number is a value of the form *y.z* where *y* is a major revision number in bits 7..4 and *z* is a minor revision number in bits 3..0.

The revision number can distinguish some chip revisions. However, MIPS is free to change this register at any time and does not guarantee that changes to its chips will necessarily change the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number to characterize the chip.

3.1.2.3.4. Exception instruction register (optional)

Floating-point coprocessor implementations may differ in their handling of exceptions. The recommended procedure is to assert floating-point exceptions in a logically precise manner; that is, the processor's EPC (Exception Program Counter) register contains the address of the floating-point instruction that was the cause of the floating-point exception. If this procedure is followed, this register **should not** be implemented.

Floating-point exceptions may be implemented in an imprecise manner; in this scheme, the exception instruction register is required, as the address of the instruction which caused a floating-point exception is not known. The exception instruction register contains floating-point instructions that have committed execution (whose addresses are no longer held in the processor's program counter chain), and which have caused an exception to occur. The register holds instructions in the same format in which they occur in memory.

Reading this register will cause all previous instructions which have not been completed in the floating-point coprocessor's pipeline to be completed. If necessary, an exception may be taken as the pipeline is emptied, and the instruction may be re-executed after the exception is serviced.

The contents of this register is undefined while no exceptions are pending. Specifically, this means that it is permissible (but not required) for an implementation to modify the exception instruction register on each floating-point operation, provided that when an exception occurs, the instruction that caused it is present in this register.

This register is only to be written into for purposes of state restoration. When writing a value into this register, the coprocessor must not be actively executing floating-point operations (which can be assured by reading this register or the control and status register first). The only valid state that can be restored is a valid instruction. The result of writing any other value, or of writing a value while floating-point operations are still in progress is not defined.

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	ft	fs	fd	function
6	5	5	5	5	6

3.1.2.3.5. Control and status register

Floating-point coprocessor control register 31 contains status and control information and is accessible by instructions running in either kernel or user mode. It controls the arithmetic rounding mode and the enabling of user-mode traps, and indicates exceptions that occurred in the most recently executed instruction, and any exceptions that may have occurred without being trapped.

Reading this register will cause all previous instructions that have not been completed in the floating-point coprocessor's pipeline to be completed. If necessary, an exception may be taken as the pipeline is emptied, and the instruction may be re-executed after the exception is serviced.

The contents of this register are unpredictable and undefined after a processor reset or a power-up event. Software should initialize this register.

31 ... 25	24	23	22 ... 18	17 ... 12	11 ... 7	6 ... 2	1 0
0	FS	c	0	cause	enables	flags	RM
7	1	1	5	6	5	5	2

cause:

17	16	15	14	13	12
E	V	Z	O	U	I
1	1	1	1	1	1

enables:

11	10	9	8	7
V	Z	O	U	I
1	1	1	1	1

flags:

6	5	4	3	2
V	Z	O	U	I
1	1	1	1	1

The FS bit is implemented on R4000 processors only. When FS is set, denormalized results are flushed to zero instead of causing an unimplemented operation exception.

The RM field controls the rounding mode for all floating-point operations. The rounding modes are described in the table below:

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

IEEE 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and optionally invoke an exception handler when the exception occurs. These features are implemented in the MIPS architecture with the cause, enable, and flag fields of the control status register. The flag bits implement IEEE 754 exception status flags, and the cause and enable bits implement exception handling.

The flag bits are cumulative and indicate that an exception was raised on some operation since the time they were explicitly reset. Flag bits are set to 1 if an IEEE 754 exception is raised, and unchanged otherwise. The flag bits are never cleared as a side effect of floating-point operations, but may be set or cleared by writing a new value into the status register, using a "move to coprocessor control" instruction.

The cause bits are logically an extension of the coprocessor 0 Cause register; they specify the cause of floating-point exceptions. The cause bits specify the exceptions raised by the last floating-point operation and cause an interrupt or exception if the corresponding enable bit is set.

The cause bits are written by each floating-point operation (but not by loads, stores, or moves). Unimplemented is set to 1 if software emulation is required, 0 otherwise. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence of an IEEE 754 exception.

A floating-point interrupt or exception is generated any time a cause bit and the corresponding enable bit are both set. A floating-point operation that sets an enabled cause bit will cause an immediate interrupt or exception, as will setting both cause and enable bits with CTC1. The R2000, R3000, and R6000 processors use external interrupts to cause a trap, whereas the R4000 processor uses an exception. R4000 floating-point exceptions cannot be disabled.

There is no enable for unimplemented. Setting unimplemented always generates a floating-point interrupt.

When a floating-point interrupt or exception is taken, no results are stored. The only state affected are the cause and flag bits. In an exception handler, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the cause field. Before returning from a floating-point interrupt or exception, or doing a CTC1, software must first clear the enabled cause bits to prevent the interrupt from being retaken. Thus user-mode programs can never observe enabled cause bits set. If this information is required in a user-mode handler, then it must be passed somewhere other than the status register.

The appropriate flag bits are set by the operation when a user-mode exception handler is invoked. This is not implemented in hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

For a floating-point operation that sets only non-enabled cause bits, no interrupt occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous

floating-point operation can be determined by reading the cause field.

The meanings of each bit in the cause field are given below. If multiple exceptions occur together on one instruction, each appropriate bit will be set.

Cause field bit	Description
E	Unimplemented operation
V	Invalid operation
Z	Division by zero
I	Inexact exception
O	Overflow exception
U	Underflow exception

The unimplemented operation exception is normally invisible to user-mode code. No implementation is required to provide them when the conditions under which the trap may occur are properly handled in hardware. It is included in order to permit implementations that do not handle the full set of operations and exceptions without software assistance. Specifically, certain commercially-available floating-point chips may be used, for which the set of operations and formats supported and details of exception handling differ from IEEE Standard 754, by using these optional exceptions to maintain compatibility.

The five IEEE 754 standard exceptions are listed below:

Field	Description
V	Invalid operation exception
Z	Division by zero exception
O	Overflow exception
U	Underflow exception
I	Inexact result exception

Each of the five exceptions is associated with a trap under user control, which is enabled by setting one of the five bits of the enable field, shown above.

The floating-point compare instruction places the condition that was detected into the "c" bit of the control and status register, so that the state of the condition line may be saved and restored. The "c" bit is set if the condition is true, and cleared if the condition is false, and is affected only by compare and move to control register instructions.

3.1.3. Save and restore state

In a MIPS I implementation, thirty-two single-word coprocessor loads or stores will save or restore the coprocessor's floating-point register state in memory. In a MIPS II implementation, sixteen doubleword coprocessors loads or stores would be required. The MIPS III implementation would require thirty-two doubleword coprocessors loads or stores. The remaining control and status information can be saved or restored through "move to/from coprocessor control register" instructions, and saving and restoring the processor registers. Normally, the control and status registers are saved first and restored last.

When coprocessor control register 31 is read, and the coprocessor is executing one or more floating-point instructions, the instructions in progress are completed or reported as exceptions. The architecture requires that no more than one of these pending instructions may cause an exception. If one of the pending instructions cannot be completed, the instruction is placed in the "exception register," if present, and information that indicates the type of exception is placed in the "control and status register." State information in the status word indicates that exceptions are pending when state is restored.

Writing a zero value to the cause field of control register 31 clears all pending exceptions, thus permitting normal processing to be restarted after the floating-point register state is restored.

3.1.4. Exception trap processing

When a floating-point exception trap is taken, the Cause register indicates that the floating-point coprocessor is the cause of the exception trap. For R4000 processors, the FPE exception code is used; for other systems a dedicated external interrupt code is typically used. The cause bits of the floating-point control status register indicate the reason for the floating-point exception; these bits are in effect an extension of the system coprocessor Cause register. Assuming that the floating-point coprocessor is implemented with precise exception handling, the MIPS processor's EPC register contains the address of the instruction that caused the exception. By use of this register, the operation and the operands of the instruction can be retrieved from memory.

When floating-point exception traps are imprecise, the MIPS processor's EPC register does not contain the address of the instruction that caused the exception, unlike most other exception traps. Normally, the EPC register contains an address that is a successor (by one or more instructions) to the offending instruction itself. In this case, the EIR contains the instruction that caused the exception.

This arrangement permits the execution of floating-point operations in the coprocessor to occur in parallel with execution of fixed-point processor operations and, in some implementations, in parallel with certain floating-point load and store instructions.

For each IEEE 754 standard exception, a flag bit is provided that is set on any occurrence of the corresponding exception condition with no corresponding exception trap signaled. It may be reset by writing a new value into the status register. The flags may be saved and restored individually, or as a group, by software. When no exception trap is signaled, a default action is taken by the floating-point coprocessor, which provides a substitute value for the original, exceptional, result of the floating-point operation. The default action taken depends on the type of exception, and in the case of the Overflow exception, the current rounding mode.

Field	Description	Rounding mode	Default action
V	Invalid operation		supply a quiet NaN
Z	Division by zero		supply a properly signed ∞
O	Overflow exception	RN	Modify overflow values to ∞ with the sign of the intermediate result
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result
		RP	Modify negative overflows to the format's most negative finite number; modify positive overflows to +∞
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to −∞
U	Underflow exception	.	supply a rounded result
I	Inexact exception		supply a rounded result

3.1.4.1. Invalid operation exception

The invalid operation exception is signaled if one or both of the operands are invalid for an implemented operation. The result, when the exceptions occurs without a trap, is a quiet NaN. The invalid operations are

(1) Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty)+(-\infty)$ or $(-\infty)-(-\infty)$

(2) Multiplication: $0 \times \infty$, with any signs

(3) Division: $\frac{0}{0}$ or $\frac{\infty}{\infty}$, with any signs

(4) Square root: $\sqrt{x}$, where x is less than zero

(5) Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format

(6) Comparison of predicates involving "<" or ">" without "?," when the operands are "unordered"

(7) Any operation on a signaling NaN

Software may simulate this exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE 754-specified functions implemented in software, such as Remainder: x REM y, where y is zero or x is infinite; conversion of a floating-point number to a decimal format whose value causes and overflow or is infinity or NaN; and transcendental functions, such as ln(−5) or $\cos^{-1}(3)$.

3.1.4.2. Division-by-zero exception

The division by zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. The result, when no trap occurs, is a correctly signed ∞.

If division by zero traps are enabled, the result register is not modified, and the source registers are preserved.

Software may simulate this exception for other operations that produce a signed infinity, such as ln(0), $sec(\frac{\pi}{2})$, $csc(0)$ or 0^{-1}.

3.1.4.3. Overflow exception

The overflow exception is signaled when what would have been the magnitude of the rounded floating-point result, were the exponent range unbounded, is larger than the destination format's largest finite number. The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.

If overflow traps are enabled, the result register is not modified, and the source registers are preserved.

3.1.4.4. Underflow exception

Two related events contribute to underflow. One is the creation of a tiny non-zero result between $\pm 2^{E_{min}}$ which, because it is tiny, may cause some other exception later. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers. IEEE Standard 754 permits a choice in how these events are detected, but requires that they must be detected the same way for all operations.

IEEE Standard 754 specifies that "tininess" may be detected either: "after rounding" (when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E_{min}}$), or "before rounding" (when a nonzero result computed as though the exponent range and the precision were unbounded would lie strictly between $\pm 2^{E_{min}}$). The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy may be detected as either "denormalization loss" (when the delivered result differs from what would have been computed if the exponent range were unbounded), or "inexact result" (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded). The MIPS architecture requires that loss of accuracy be detected as inexact result.

When an underflow trap is not enabled, underflow is signaled (via the underflow flag) only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or $\pm 2^{E_{min}}$. When an underflow trap is enabled, underflow is signaled when tininess is detected regardless of loss of accuracy.

If underflow traps are enabled, the result register is not modified, and the source registers are preserved.

3.1.4.5. Inexact exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception is signaled. The rounded or overflowed result is delivered to the destination register, when no inexact trap occurs. If inexact exception traps are enabled, the result register is not modified, and the source registers are preserved.

3.1.4.6. Unimplemented operation exception

If an operation is specified that the hardware may not perform, due to an implementation restriction on the supported operations or supported formats, an unimplemented operation exception may be signaled, which always causes a trap, for which there are no corresponding enable or flag bits. The trap cannot be disabled.

This exception is raised at the execution of the unimplemented instruction. The instruction may be emulated in software, possibly using implemented floating-point unit instructions to accomplish the emulation. Normal instruction execution may then be restarted.

This exception is also raised when an attempt is made to execute an instruction with an operation code or format code which has been reserved for future architectural definition. The unimplemented instruction trap is not optional, since the current definition contains codes of this kind.

This exception may be signaled when unusual operands or result conditions are detected, for which the implemented hardware cannot properly handle the condition. These may include (but are not limited to), denormalized operands or results, NaN operands, trapped overflow or underflow conditions. The use of this exception for such conditions is optional.

3.1.4.7. Trap handlers for the five IEEE Standard 754 exceptions

IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions. The trap handler may compute or specify a substitute result to be placed in the destination register of the operation. The trap handler may determine what exceptions occurred during the operation, the operation that was being performed, and the destination's format by examining a copy of the operation exception register returned by the floating-point coprocessor, or by retrieving the instruction via the processor's EPC register. On overflow or underflow exceptions (except for conversions) and on inexact exceptions, the trap handler has access to the correctly rounded result, by examining the source registers, and simulating the operation in software. On overflow or underflow exceptions encountered on floating-point conversions, and on invalid operation and divide-by-zero exceptions, the trap handler has access to the operand values by examining the source registers of the instruction.

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

3.2. Floating-point instruction set

The MIPS floating-point instruction set, as described in this chapter, provides an architecture-level definition of the MIPS floating-point instructions. Matters which are explicitly stated as undefined must be considered to be implementation-dependent and should therefore not be relied upon when, as is usually the case, transportability between implementations is required.

3.2.1. Instruction Formats and Notational Conventions

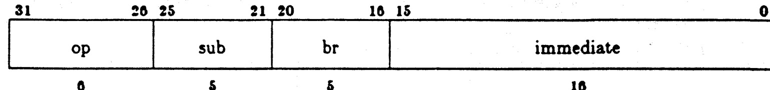
Every instruction consists of a single word (32 bits) aligned on a word boundary.

There are only four instruction formats:

I-type (Immediate):

31 ... 26	25 ... 21	20 ... 16	15 ... 0
op	base	ft	immediate
6	5	5	16

B-type (Branch):



M-type (Move):

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 0
op	sub	rt	fs	0
6	5	5	5	11

R-type (Register):

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
op	sub	ft	fs	fd	function
6	5	5	5	5	6

where:

op	is a 6-bit operation code
sub	is a 5-bit sub-operation code
br	is a 5-bit branch code
rt	is a 5-bit source/destination general register specifier
ft	is a 5-bit source/destination float register specifier
fs	is a 5-bit source register specifier
fd	is a 5-bit destination register specifier
immediate	is a 16-bit address/branch displacement
function	is a 6-bit function code
0	undefined if non-zero

This document adopts the notational convention that all variable subfields in an instruction format (such as fs, ft, immediate, etc.) have lower case names. Note that the field named fs, ft, and fd here are named rd, rt, and

sa in cpu instructions,

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

The two instruction subfields op and function have constant 6-bit values for specific instructions. These values are given upper case mnemonic names in the main body of this document. For example, op = COP1 and function = FADD in the floating-point add instruction. In some cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. For example, LWC*z* (Load Coprocessor *z*) represents 4 different 6-bit opcodes, each composed of the fixed 4-bit subfield LWC concatenated with a variable 2-bit subfield *z* which designates one of the 4 coprocessor classes.

The actual bit encoding for all the mnemonics is specified in chapter 8.

Under the sub-heading "Operation," the operation performed by each instruction is described, using the notation described in Chapter 2. In addition the following are defined:

symbol	meaning
FGR[*x*]	CPR[1, *x*], floating-point coprocessor general register *x* as viewed by the processor.
FPR[*x*]	Floating-point register *x* assembled from one to four floating-point coprocessor registers.
FCR[*x*]	CCR[1, *x*], floating-point control register *x*.

The description of the immediate causes and manner of handling of exceptions is omitted from the instruction descriptions in this chapter. The exceptions that may occur due to the execution of each instruction is explicitly listed, and the causes and handling of these exceptions is detailed in chapter 5.

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by the five-bit sub field:

Code	Mnemonic	Size	Format
16	S	single	binary floating-point
17	D	double	binary floating-point
18	E	extended	binary floating-point
19	Q	quad	binary floating-point
20	W	single	32-bit binary fixed-point
21	L	longword	64-bit binary fixed-point
22-31	-	-	reserved

The low-order six bits of the coprocessor instruction indicate the floating-point operation to be performed:

Code	Mnemonic	Operation
0	ADD	Add
1	SUB	Subtract
2	MUL	Multiply
3	DIV	Divide
4	SQRT	Square root
5	ABS	Absolute value
6	MOV	Move
7	NEG	Negate
8	ROUND.L	Convert to single fixed-point, rounded to nearest
9	TRUNC.L	Convert to single fixed-point, rounded toward zero
10	CEIL.L	Convert to single fixed-point, rounded to +∞
11	FLOOR.L	Convert to single fixed-point, rounded to −∞
12	ROUND.W	Convert to single fixed-point, rounded to nearest
13	TRUNC.W	Convert to single fixed-point, rounded toward zero
14	CEIL.W	Convert to single fixed-point, rounded to +∞
15	FLOOR.W	Convert to single fixed-point, rounded to −∞
16-31	-	reserved
32	CVT.S	Convert to single floating-point
33	CVT.D	Convert to double floating-point
34	CVT.E	Convert to extended floating-point
35	CVT.Q	Convert to quad floating-point
36	CVT.W	Convert to 32-bit binary fixed-point
37	CVT.L	Convert to 64-bit binary fixed-point
38-47	-	reserved
48-63	C	Floating-point compare

Each operation is valid only for certain formats. Implementations may support some of these formats and operations only through emulation, but only need support combinations that are valid, which are marked with a "•" in the table below. Those combinations marked with a "†" are not currently specified by this architecture, but must cause an unimplemented operation trap, to maintain compatibility with future architecture extensions. Entries which are blank are not valid, and therefore the result of such instructions are not defined.

operation	source format					
	Single	Double	Extended	Quad	Word	Longword
ADD	•	•	•	•	†	†
SUB	•	•	•	•	†	†
MUL	•	•	•	•	†	†
DIV	•	•	•	•	†	†
SQRT	•	•	•	•	†	†
ABS	•	•	•	•	†	†
MOV	•	•	•	•		
NEG	•	•	•	•	†	†
TRUNC.L	•	•	•	•		
ROUND.L	•	•	•	•		
CEIL.L	•	•	•	•		
FLOOR.L	•	•	•	•		
TRUNC.W	•	•	•	•		
ROUND.W	•	•	•	•		
CEIL.W	•	•	•	•		
FLOOR.W	•	•	•	•		
CVT.S		•	•	•	•	•
CVT.D	•		•	•	•	•
CVT.E	•	•		•	•	•
CVT.Q	•	•	•		•	•
CVT.W	•	•	•	•		
CVT.L	•	•	•	•		
C	•	•	•	•	†	†

The coprocessor branch on condition true/false instructions can be used to logically negate any predicate. Thus, the 32 possible conditions require only 16 distinct comparisons, as given in the table below.

Condition			Relations				Invalid operation exception if unordered
Mnemonic		code	greater than	less than	equal	unord-ered	
true	false						
F	T	0	F	F	F	F	No
UN	OR	1	F	F	F	T	No
EQ	NEQ	2	F	F	T	F	No
UEQ	OGL	3	F	F	T	T	No
OLT	UGE	4	F	T	F	F	No
ULT	OGE	5	F	T	F	T	No
OLE	UGT	6	F	T	T	F	No
ULE	OGT	7	F	T	T	T	No
SF	ST	8	F	F	F	F	Yes
NGLE	GLE	9	F	F	F	T	Yes
SEQ	SNE	10	F	F	T	F	Yes
NGL	GL	11	F	F	T	T	Yes
LT	NLT	12	F	T	F	F	Yes
NGE	GE	13	F	T	F	T	Yes
LE	NLE	14	F	T	T	F	Yes
NGT	GT	15	F	T	T	T	Yes

3.2.2. Load and Store to/from Floating-point coprocessor

In the MIPS I instruction set, all loads are implemented with a latency of at most one instruction. That is, the instruction immediately following a load normally cannot use the contents of the register which will be loaded with the data being fetched from storage.

In the MIPS II and MIPS III instruction set, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware will interlock, requiring additional real cycles, so scheduling load delay slots is still desirable, though not required for functional code.

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 0
op	base	ft	offset
6	5	5	16

where:

op	is the 6-bit operation code
base	is the 5-bit base register specifier
ft	is the 5-bit source (for stores) or destination (for loads) coprocessor register specifier
offset	is the 16-bit signed immediate offset

Description	op
Load Word	LWC1
Store Word	SWC1
Load Doubleword	LDC1
Store Doubleword	SDC1

All coprocessor loads and stores reference aligned data items. Thus, for word loads and stores, the access type field is always WORD, and the low-order two bits of the address must always be zero. For doubleword loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero. Regardless of byte-numbering order (endian-ness), the address specifies the byte which has the smallest byte address of each byte in the addressed field. For big-endian machines, this is the leftmost byte; for little-endian machines, this is the rightmost byte.

LOAD WORD TO FLOATING-POINT COPROCESSOR

Format:

LWC1 ft,offset(base)

31–26	25–21	20–16	15–0
LWC1	base	ft	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The contents of the word at the memory location specified by the effective address are loaded into register ft of the floating-point coprocessor.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, LWC1 loads either the high or low half of the 16 even floating-point registers. When FR is set, LWC1 loads the low 32 bits of any floating-point register.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

In MIPS I implementations, the contents of general register ft is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
CPR[z, ft] ← undefined

T+1: CPR[z, ft] ← mem

MIPS II operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
CPR[z, ft] ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)

MIPS III operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
$pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } (ReverseEndian \| 0^2))$
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
$byte \leftarrow vAddr_{2..0} \text{ xor } (BigEndianCPU \| 0^2)$
if SR_{26}=1 then
  $CPR[z, ft] \leftarrow undefined^{32} \| mem_{31+8*byte..8*byte}$
elseif ft_0=0 then
  $CPR[z, ft_{4..1} \| 0] \leftarrow CPR[z, ft_{4..1} \| 0]_{63..32} \| mem_{31+8*byte..8*byte}$
else
  $CPR[z, ft_{4..1} \| 0] \leftarrow mem_{31+8*byte..8*byte} \| CPR[z, ft_{4..1} \| 0]_{31..0}$
endif

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LOAD DOUBLEWORD TO FLOATING-POINT COPROCESSOR

Format:

LDC1 ft,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LDC1	base	ft	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

For MIPS II processors, the contents of the doubleword at the memory location specified by the effective address are loaded into registers ft and ft+1 of the floating-point coprocessor. This instruction is not valid, and is undefined, when the least significant bit of ft is non-zero.

For MIPS III processors, the contents of the doubleword at the memory location specified by the effective address are loaded into the 64-bit register ft of the floating-point coprocessor. The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, this instruction is not defined when the least significant bit of ft is non-zero. When FR is set, ft may specify either odd or even registers.

If any of the three least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction causes a reserved instruction exception on MIPS I processors.

MIPS II operation:

```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      if BigEndianCPU = 1 then
         CPR[z, ft+1] ← LoadMemory (uncached, WORD, pAddr+0, vAddr+0, DATA)
         CPR[z, ft+0] ← LoadMemory (uncached, WORD, pAddr+4, vAddr+4, DATA)
      else
         CPR[z, ft+0] ← LoadMemory (uncached, WORD, pAddr+0, vAddr+0, DATA)
         CPR[z, ft+1] ← LoadMemory (uncached, WORD, pAddr+4, vAddr+4, DATA)
      endif
```

MIPS III operation:

```
T:    vAddr ← ((offset15)^48 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
      if SR26=1 then
         CPR[z, ft] ← data
      else
         CPR[z, ft4..1 || 0] ← data
      endif
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I only)

STORE WORD FROM FLOATING-POINT COPROCESSOR

Format:

SWC1 ft,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
SWC1	base	ft	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The contents of register ft from the floating-point coprocessor are stored at the memory location specified by the effective address.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, SWC1 stores either the high or low half of the 16 even floating-point registers. When FR is set, SWC1 stores the low 32 bits of any floating-point register.

If either of the two least significant bits of the effective address is non-zero, an address error exception takes place.

MIPS I/II operation:

T: vAddr ← $((\text{offset}_{15})^{16} \,\|\, \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← CPR[z, ft]
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

MIPS III operation:

T: vAddr ← $((\text{offset}_{15})^{48} \,\|\, \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
pAddr ← $\text{pAddr}_{PSIZE-1..3} \,\|\, (\text{pAddr}_{2..0}$ xor (ReverseEndian $\|\, 0^2$))
byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU $\|\, 0^2$)
if SR_{26}=1 then
  data ← $\text{CPR}[z, ft]_{63-8*byte..0} \,\|\, 0^{8*byte}$
elseif ft_0=0 then
  data ← $\text{CPR}[z, ft_{4..1} \,\|\, 0]_{63-8*byte..0} \,\|\, 0^{8*byte}$
else
  data ← $0^{32-8*byte} \,\|\, \text{CPR}[z, ft_{4..1} \,\|\, 0]_{63..32-8*byte}$
endif
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

STORE DOUBLEWORD FROM FLOATING-POINT COPROCESSOR

Format:

SDC1 ft,offset(base)

31 - 26	25 - 21	20 - 16	15 - 0
SDC1	base	ft	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

For MIPS II processors, the contents of registers ft and ft+1 from the floating-point coprocessor are stored at the memory location specified by the effective address. This instruction is not valid, and is undefined, when the least significant bit of ft is non-zero.

For MIPS III processors, the 64-bit register ft is stored to the contents of the doubleword at the memory location specified by the effective address. The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, this instruction is not defined when the least significant bit of ft is non-zero. When FR is set, ft may specify either odd or even registers.

If any of the three least significant bits of the effective address is non-zero, an address error exception takes place.

This instruction causes a reserved instruction exception on MIPS I processors.

MIPS II operation:

```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      if BigEndianCPU = 1 then
         StoreMemory (uncached, WORD, CPR[z, ft+1], pAddr+0, vAddr+0, DATA)
         StoreMemory (uncached, WORD, CPR[z, ft+0], pAddr+4, vAddr+4, DATA)
      else
         StoreMemory (uncached, WORD, CPR[z, ft+0], pAddr+0, vAddr+0, DATA)
         StoreMemory (uncached, WORD, CPR[z, ft+1], pAddr+4, vAddr+4, DATA)
      endif
```

MIPS III operation:

```
T:    vAddr ← ((offset15)^48 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      if SR26=1 then
         data ← CPR[z, ft]
      else
         data ← CPR[z, ft4..1 || 0]
      endif
      StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (MIPS I only)

3.2.3. Move To/From Floating-point Coprocessor

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
COP1	sub	rt	fs	0
6	5	5	5	11

where:

COP1	is a 6-bit operation code and coprocessor unit number specifier
sub	is a 5-bit coprocessor sub-operation field
rt	is a 5-bit general register source/destination specifier
fs	is a 5-bit coprocessor source/destination register specifier

These coprocessor operations are moves between general processor and floating-point coprocessor registers.

MOVE WORD TO FLOATING-POINT COPROCESSOR

Format:

MTC1 rt,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	MT	rt	fs	0	0
6	5	5	5	5	6

Description:

The contents of register rt is loaded into register fs of the floating-point coprocessor.

The contents of floating-point register fs is undefined for time "T" of the instruction immediately following this load instruction.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, MTC1 loads either the high or low half of the 16 even floating-point registers. When FR is set, MTC1 loads the low 32 bits of any floating-point register.

MIPS I/II operation:

```
T:     data ← GPR[rt]

T+1:   CPR[z, fs] ← data
```

MIPS III operation:

```
T:     data ← GPR[rt]31..0

T+1:   if SR26=1 then
          CPR[z, fs] ← undefined^32 || data
       elseif fs0=0 then
          CPR[z, fs4..1 || 0] ← CPR[z, fs4..1 || 0]63..32 || data
       else
          CPR[z, fs4..1 || 0] ← data || CPR[z, fs4..1 || 0]31..0
       endif
```

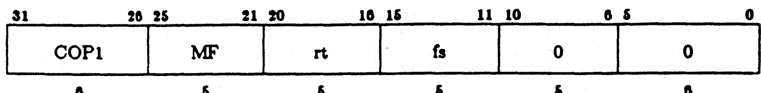
Exceptions:

Coprocessor unusable

MOVE WORD FROM FLOATING-POINT COPROCESSOR

Format:

MFC1 rt,fs



Description:

The contents of register fs from the floating-point coprocessor is stored into processor register rt.

The contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, MFC1 stores either the high or low half of the 16 even floating-point registers. When FR is set, MFC1 stores the low 32 bits of any floating-point register.

MIPS I/II operation:

T: data ← CPR[z, fs]

T+1: GPR[rt] ← data

MIPS III operation:

```
T:    if fs0=0 then
         data ← CPR[z, fs4..1 || 0]31..0
      else
         data ← CPR[z, fs4..1 || 0]63..32
      endif
```

T+1: GPR[rt] ← $(data_{31})^{32}$ || data

Exceptions:

Coprocessor unusable

MOVE DOUBLEWORD TO FLOATING-POINT COPROCESSOR

Format:

DMTC1 rt,fs

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1	DMT	rt	fs	0	0
6	5	5	5	5	6

Description:

The contents of register rt is loaded into register fs of the floating-point coprocessor.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, this instruction is not defined when the least significant bit of fs is non-zero. When FR is set, fs may specify either odd or even registers.

The contents of floating-point register fs is undefined for time "T" of the instruction immediately following this load instruction.

This instruction is defined only in MIPS III implementations.

MIPS III operation:

```
T:      data ← GPR[rt]

T+1:    if SR26=1 then
           CPR[z, fs] ← data
        else
           CPR[z, fs4..1 || 0] ← data
        endif
```

Exceptions:

Coprocessor unusable

MOVE DOUBLEWORD FROM FLOATING-POINT COPROCESSOR

Format:

DMFC1 rt,fs

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	DMF	rt	fs	0	0
6	5	5	5	5	6

Description:

The contents of register fs from the floating-point coprocessor is stored into processor register rt.

The FR bit of the Status register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When clear, this instruction is not defined when the least significant bit of fs is non-zero. When FR is set, fs may specify either odd or even registers.

The contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

This instruction is defined only in MIPS III implementations.

MIPS III operation:

```
T:      if SR26=1 then
            data ← CPR[z, fs]
        else
            data ← CPR[z, fs4..1 || 0]
        endif

T+1:    GPR[rt] ← data
```

Exceptions:

Coprocessor unusable

3.2.4. Move To/From Floating-point Coprocessor Control Registers

Instruction Format:

31–26	25–21	20–16	15–11	10–0
COP1	sub	rt	fs	0
6	5	5	5	11

where:

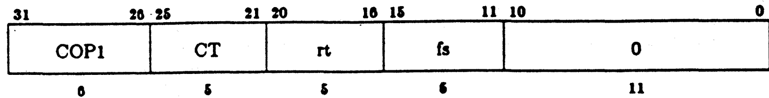
COP1	is a 6-bit operation code and coprocessor unit number specifier
sub	is a 5-bit coprocessor sub-operation field
rt	is a 5-bit general register specifier
fs	is a 5-bit coprocessor register specifier

These coprocessor operations are moves between general processor and floating-point coprocessor control registers.

MOVE CONTROL WORD TO FLOATING-POINT COPROCESSOR

Format:

CTC1 rt,fs



Description:

The contents of register rt is loaded into control register fs of the floating-point coprocessor.

This operation is only defined when fs equals 0 or 31, or when an EIR is present, equal to 30.

Writing to control register 31, the floating-point control status register, will cause an interrupt or exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs.

The contents of floating-point control register fs is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I/II operation:

T: temp ← GPR[rt];

T+1: FCR[fs] ← temp;
COC[1] ← $FCR[31]_{23}$

MIPS III operation:

T: temp ← $GPR[rt]_{31..0}$;

T+1: FCR[fs] ← temp;
COC[1] ← $FCR[31]_{23}$

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

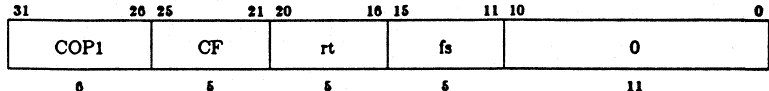
Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Division by zero exception
Inexact exception
Overflow exception
Underflow exception

MOVE CONTROL WORD FROM FLOATING-POINT COPROCESSOR

Format:

CFC1 rt,fs



Description:

The contents of control register fs from the floating-point coprocessor is stored into processor register rt.

This operation is only defined when fs equals 0 or 31, or when an EIR is present, equal to 30.

The contents of general register rt is undefined for time "T" of the instruction immediately following this load instruction.

MIPS I/II operation:

T: temp ← FCR[fs];

T+1: GPR[rt] ← temp;

MIPS III operation:

T: temp ← FCR[fs];

T+1: GPR[rt] ← $(temp_{31})^{32}$ || temp;

Exceptions:

Coprocessor unusable

3.2.5. Computational Instructions

Instruction Format:

31–26	25–21	20–16	15–11	10–6	5–0
COP1	fmt	ft	fs	fd	function
6	5	5	5	5	6

where:

COP1	is a 6-bit major operation code
fmt	is a 5-bit format specifier
fs	is a 5-bit source1 register
ft	is a 5-bit source2 register
fd	is a 5-bit destination register
function	is a 6-bit function field

Description	func
Floating-point Add	ADD
Floating-point Subtract	SUB
Floating-point Multiply	MUL
Floating-point Divide	DIV
Floating-point Square root	SQRT
Floating-point Absolute Value	ABS
Floating-point Move	MOV
Floating-point Negate	NEG
Floating-point Convert	CVT
Floating-point Compare	C

In the following pages, the notation FGR refers to floating-point coprocessor general registers 0 through 31, and FPR refers to floating-point registers 0 through 30. FPR's are formed by concatenation of floating-point coprocessor registers.

The following routines are used in the description of the floating-point operations to get the value of an FPR or to change the value of an FGR:

MIPS I/II operation:

```
value ← ValueFPR(fpr, fmt):
   /* undefined for odd fpr */
   case fmt of
      S, W:
         value ← FGR[fpr + 0]
      D:
         /* undefined for fpr not even */
         value ← FGR[fpr + 1] || FGR[fpr + 0]
      E, Q:
         /* undefined for fpr not a multiple of four */
         value ← FGR[fpr + 3] || FGR[fpr + 2] || FGR[fpr + 1] || FGR[fpr + 0]
   end

StoreFPR (fpr, fmt, value):
   /* undefined for odd fpr */
   case fmt of
      S, W:
         FGR[fpr + 1] ← undefined
         FGR[fpr + 0] ← value
      D:
         FGR[fpr + 1] ← value63..32
         FGR[fpr + 0] ← value31..0
      E, Q:
         /* undefined for fpr not a multiple of four */
         FGR[fpr + 3] ← value127..96
         FGR[fpr + 2] ← value95..64
         FGR[fpr + 1] ← value63..32
         FGR[fpr + 0] ← value31..0
   end
```

MIPS III operation:

```
value ← ValueFPR(fpr, fmt):

   case fmt of
      S, W:
         value ← FGR[fpr]31..0
      D, L:
         value ← FGR[fpr]
      E, Q:
         /* undefined for odd fpr */
         value ← FGR[fpr + 1] || FGR[fpr + 0]
      W:
         value ← FGR[fpr]
   end

StoreFPR (fpr, fmt, value):

   case fmt of
      S, W:
         FGR[fpr] ← undefined^32 || value
      D, L:
         FGR[fpr] ← value
      E, Q:
         /* undefined for odd fpr */
         FGR[fpr + 1] ← value127..64
         FGR[fpr + 0] ← value63..0
   end
```

FLOATING-POINT ADD

Format:

ADD.fmt fd,fs,ft

31–26	25–21	20–16	15–11	10–6	5–0
COP1	fmt	ft	fs	fd	ADD
6	5	5	5	5	6

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically added. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR(fs, fmt) + ValueFPR(ft, fmt));
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

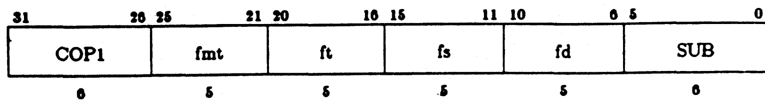
Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT SUBTRACT

Format:

SUB.fmt fd,fs,ft



31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	ft	fs	fd	SUB
6	5	5	5	5	6

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically subtracted. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) – ValueFPR(ft, fmt));

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT MULTIPLY

Format:

MUL.fmt   fd,fs,ft

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	ft	fs	fd	MUL
6	5	5	5	5	6

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically multiplied. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR(fs, fmt) * ValueFPR(ft, fmt));
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT DIVIDE

Format:

DIV.fmt fd,fs,ft

31–26	25–21	20–16	15–11	10–6	5–0
COP1	fmt	ft	fs	fd	DIV
6	5	5	5	5	6

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically divided. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt));

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Division by zero exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT SQUARE ROOT

Format:

SQRT.fmt fd,fs

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	0	fs	fd	SQRT
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified format and the positive arithmetic square root is taken. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. If the value of fs corresponds to –0, the result will be –0. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

This instruction is not implemented on MIPS I processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception

FLOATING-POINT ABSOLUTE VALUE

Format:

ABS.fmt fd,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	0	fs	fd	ABS
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs are interpreted in the specified format and the arithmetic absolute value is taken. The result is placed in the floating-point register specified by fd.

The absolute value operation is arithmetic; a NaN operand signals invalid operation.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception

FLOATING-POINT MOVE

Format:

MOV.fmt fd,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	0	fs	fd	MOV
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified format and is copied into floating-point register specified by fd.

The move operation is non-arithmetic; no IEEE 754 exceptions occur as a result of the instruction.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

```
T:    StoreFPR(fd, fmt, ValueFPR(fs, fmt));
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

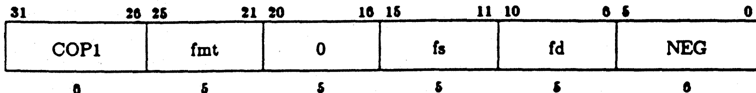
Coprocessor Exceptions:

Unimplemented operation exception

FLOATING-POINT NEGATE

Format:

NEG.fmt fd,fs



Description:

The contents of the floating-point register specified by fs are interpreted in the specified format and the arithmetic negation is taken. The result is placed in the floating-point register specified by fd.

The negate operation is arithmetic; a NaN operand signals invalid operation.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception

FLOATING-POINT CONVERT TO SINGLE FLOATING-POINT FORMAT

Format:

CVT.S.fmt fd,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	0	fs	fd	CVT.S
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single binary floating-point format. The result is placed in the floating-point register specified by fd.

Rounding occurs according to the currently specified rounding mode.

This instruction is valid only for conversion from double, extended, or quad floating-point format, or from 32-bit or 64-bit fixed-point format. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of the source register specifier is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT CONVERT TO DOUBLE FLOATING-POINT FORMAT

Format:

CVT.D.fmt fd,fs

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	0	fs	fd	CVT.D
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the double binary floating-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversion from single, extended, or quad floating-point format, or from 32-bit or 64-bit fixed-point format.

If the single floating-point or single fixed-point format is specified, the operation is exact. If the extended or quad-precision format is specified, rounding occurs according to the currently specified rounding mode. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of the source register specifier is set, as the register numbers specify aligned coprocessor general registers.

Operation:

T: StoreFPR(fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

FLOATING-POINT CONVERT TO EXTENDED FLOATING-POINT FORMAT

Format:

CVT.E.fmt fd,fs

31–26	25–21	20–16	15–11	10–6	5–0
COP1	fmt	0	fs	fd	CVT.E
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the extended binary floating-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversion from single, double, or quad floating-point format, or from 32-bit or 64-bit fixed-point format.

If the single or double-precision floating-point or single fixed-point format is specified, the operation is exact. If the quad-precision floating-point format is specified, the result is rounded according to the current rounding mode.

For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers. The operation is not defined if bit 1 (MIPS I and II) or bit 0 (MIPS III) of the destination register specifier is set, as the register number specifies an aligned-quad of adjacent coprocessor general registers.

If a quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of the source register specifier is set, as the register number specifies an aligned-quad of adjacent coprocessor general registers.

This instruction is not currently implemented by any R-Series processors.

Operation:

T: StoreFPR(fd, E, ConvertFmt(ValueFPR(fs, fmt), fmt, E))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

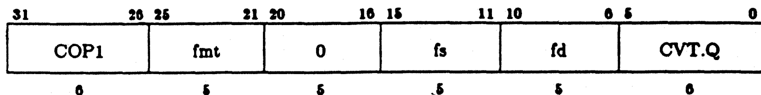
Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception

FLOATING-POINT CONVERT TO QUAD FLOATING-POINT FORMAT

Format:

CVT.Q.fmt fd,fs



Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the quad-precision floating-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversion from single, double or extended floating-point formats, or from 32-bit or 64-bit fixed-point format.

The convert to quad-precision operation is always exact.

For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers. The operation is not defined if bit 1 (MIPS I and II) or bit 0 (MIPS III) of the destination register specifier is set, as the register number specifies an aligned-quad of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of the source register specifier is set, as the register numbers specify aligned coprocessor general registers.

This instruction is not currently implemented by any R-Series processors.

Operation:

T: StoreFPR(fd, E, ConvertFmt(ValueFPR(fs, fmt), fmt, E))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception

FLOATING-POINT CONVERT TO SINGLE FIXED-POINT FORMAT

Format:

CVT.W.fmt fd,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	0	fs	fd	CVT.W
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. For MIPS I and MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{31}-1$ is returned.

Operation:

T: StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT CONVERT TO LONG FIXED-POINT FORMAT

Format:

CVT.L.fmt fd,fs

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
COP1	fmt	0	fs	fd	CVT.L
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined when bit 0 of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I and MIPS II processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT ROUND TO SINGLE FIXED-POINT FORMAT

Format:

ROUND.W.fmt fd,fs

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1	fmt	0	fs	fd	ROUND.W
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (0).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. For MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS II) or bit 0 (MIPS III) of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{31}-1$ is returned.

This instruction is not implemented on MIPS I processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT TRUNCATE TO SINGLE FIXED-POINT FORMAT

Format:

TRUNC.W.fmt fd,fs

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1	fmt	0	fs	fd	TRUNC.W
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (1).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. For MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS II) or bit 0 (MIPS III) of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{31}-1$ is returned.

This instruction is not implemented on MIPS I processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT CEILING TO SINGLE FIXED-POINT FORMAT

Format:

CEIL.W.fmt fd,fs

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	0	fs	fd	CEIL.W
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $+\infty$ (2).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. For MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS II) or bit 0 (MIPS III) of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{31}-1$ is returned.

This instruction is not implemented on MIPS I processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT FLOOR TO SINGLE FIXED-POINT FORMAT

Format:

FLOOR.W.fmt fd,fs

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	0	fs	fd	FLOOR.W
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $-\infty$ (3).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. For MIPS II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS II) or bit 0 (MIPS III) of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{31}-1$ is returned.

This instruction is not implemented on MIPS I processors, and will cause the unimplemented operation exception to occur.

Operation:

```
T:    StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT ROUND TO LONG FIXED-POINT FORMAT

Format:

ROUND.L.fmt fd,fs

31–26	25–21	20–16	15–11	10–6	5–0
COP1	fmt	0	fs	fd	ROUND.L
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (0).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined when bit 0 of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I and MIPS II processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

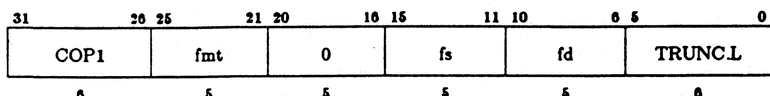
Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT TRUNCATE TO LONG FIXED-POINT FORMAT

Format:

TRUNC.L.fmt fd,fs



Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (1).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined when bit 0 of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause the unimplemented operation exception to occur.

Operation:

T: StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT CEILING TO LONG FIXED-POINT FORMAT

Format:

CEIL.L.fmt fd,fs

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	fmt	0	fs	fd	CEIL.L
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $+\infty$ (2).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined when bit 0 of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause the unimplemented operation exception to occur.

Operation:

```
T:      StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT FLOOR TO LONG FIXED-POINT FORMAT

Format:

FLOOR.L.fmt fd,fs

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1	fmt	0	fs	fd	FLOOR.L
6	5	5	5	5	6

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $-\infty$ (3).

This instruction is valid only for conversion from a single, double, extended or quad precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined when bit 0 of the source register specification is set, as the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, Invalid operation is raised. If Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I and MIPS II processors, and will cause the unimplemented operation exception to occur.

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOATING-POINT COMPARE

Format:

C.cond.fmt fs,ft

31–26	25–21	20–16	15–11	10–6	5–4	3–0
COP1	fmt	ft	fs	0	FC	cond
6	5	5	5	5	2	4

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically compared. A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is a "Not a Number," and the high-order bit of the condition field is set, an invalid operation trap is taken. After a one instruction delay, the condition is available for testing with "branch on floating-point coprocessor condition" instructions.

Comparisons are exact and neither overflow nor underflow. Four mutually exclusive relations are possible results: "less than," "equal," "greater than," and "unordered." The last case arises when one or both of the operands are NaN; every NaN compares "unordered" with everything, including itself. Comparisons ignore the sign of zero, so +0=−0.

This instruction is valid only for single, double, extended, and quad precision floating-point formats. For MIPS I and II processors the operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers.

If extended or quad-precision format is specified, the operation is not defined when bit 1 (MIPS I and II) or bit 0 (MIPS III) of any register specification is set, as the register numbers specify aligned coprocessor general registers.

Operation:

```
T:    if NaN(ValueFPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
          less ← false
          equal ← false
          unordered ← true
          if cond_3 then
              signal InvalidOperationException
          endif
      else
          less ← ValueFPR(fs, fmt) < ValueFPR(ft, fmt)
          equal ← ValueFPR(fs, fmt) = ValueFPR(ft, fmt)
          unordered ← false
      endif
      condition ← (cond_2 and less) or
          (cond_1 and equal) or
          (cond_0 and unordered)
      FCR[31]_23 ← condition
      COC[1] ← condition
```

Exceptions:

Coprocessor unusable
Coprocessor interrupt (R2000, R3000, and R6000) or Floating-Point Exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception

3.2.6. Branch on Floating-point Coprocessor Condition

Instruction Format:

31 — 26	25 — 21	20 — 16	15 — 0
COP1	BC	br	offset
6	5	5	16

where:

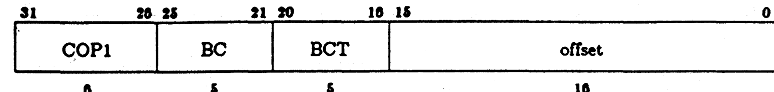
bc   is the 5-bit sub-opcode for coprocessor branches
br   is a 5-bit branch condition specifier
offset is a 16-bit offset

Description	cond
Floating-point Coprocessor branch if True Floating-point Coprocessor branch if False	BC1T BC1F
Floating-point Coprocessor branch if True Likely Floating-point Coprocessor branch if False Likely	BC1TL BC1FL

BRANCH ON FLOATING-POINT COPROCESSOR TRUE

Format:

BC1T offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32 bits. If the contents of the floating-point coprocessor's condition line is true, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

T-1: condition ← COC[1]

T: target ← $(offset_{15})^{14}$ || offset || 0^2

T+1: if condition then
PC ← PC + target
endif

MIPS III operation:

T-1: condition ← COC[1]

T: target ← $(offset_{15})^{38}$ || offset || 0^2

T+1: if condition then
PC ← PC + target
endif

Exceptions:

Coprocessor unusable exception

BRANCH ON FLOATING-POINT COPROCESSOR FALSE

Format:

BC1F offset

31 ... 26	25 ... 21	20 ... 16	15 ... 0
COP1	BC	BCF	offset
6	5	5	16

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 to 32 bits. If the contents of the floating-point coprocessor's condition line is false, the target address is branched to, with a delay of one instruction.

MIPS I/II operation:

```
T-1:    condition ← not COC[1]

T:      target ← (offset15)^14 || offset || 0^2

T+1:    if condition then
            PC ← PC + target
        endif
```

MIPS III operation:

```
T-1:    condition ← not COC[1]

T:      target ← (offset15)^38 || offset || 0^2

T+1:    if condition then
            PC ← PC + target
        endif
```

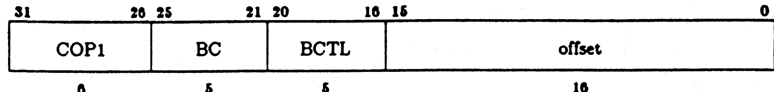
Exceptions:

Coprocessor unusable exception

BRANCH ON FLOATING-POINT COPROCESSOR TRUE LIKELY

Format:

BC1TL offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32 bits. If the contents of the floating-point coprocessor's condition line is true, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid on MIPS I processors.

MIPS II operation:

```
T-1:   condition ← COC[1]

T:     target ← (offset15)^14 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

MIPS III operation:

```
T-1:   condition ← COC[1]

T:     target ← (offset15)^38 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

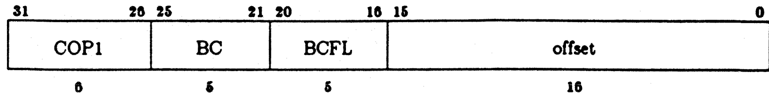
Exceptions:

Coprocessor unusable exception

BRANCH ON FLOATING-POINT COPROCESSOR FALSE LIKELY

Format:

BC1FL offset



Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32 bits. If the contents of the floating-point coprocessor's condition line is false, the target address is branched to, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

This instruction is not valid on MIPS I processors.

MIPS II operation:

```
T-1:   condition ← not COC[1]

T:     target ← (offset_15)^14 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

MIPS III operation:

```
T-1:   condition ← not COC[1]

T:     target ← (offset_15)^38 || offset || 0^2

T+1:   if condition then
          PC ← PC + target
       else
          NullifyCurrentInstruction
       endif
```

Exceptions:

Coprocessor unusable exception

3.3. Compatibility with the IEEE 754 floating-point standard

The MIPS floating-point coprocessor fully conforms to the requirements of ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic." In addition to conforming to the requirements of the IEEE standard, the MIPS floating-point coprocessor architecture fully supports the standard's recommendations. In certain fairly obscure cases, the IEEE standard's recommendations are incomplete, ambiguous, or left to the implementors' discretion. In the following section, the interpretations chosen for the MIPS floating-point architecture are described.

3.3.1. Interpretation of the standard

IEEE Standard 754 gives the implementor choices in the detection of underflow conditions. The MIPS floating-point architecture requires that *tininess* be detected *after rounding*, and that *loss of accuracy* be detected as *inexact result*.

IEEE Standard 754 does not define, when an exception condition occurs, how the cause field is set when traps are disabled, or how the flag bits are set when traps are enabled. The MIPS floating-point architecture requires that the cause field is loaded (set or cleared), and the flag bits set, regardless of whether traps are enabled. (Note: setting the flag bits when a trap is taken is implemented in software emulation, not in hardware.)

IEEE Standard 754 specifies that an inexact exception may occur concurrently with an overflow or underflow exception, and that the overflow or underflow exception trap take priority. It further requires that the inexact trap be taken when an operation overflows while the overflow trap is disabled. The MIPS floating-point architecture specifies that both the inexact exception and the overflow or underflow exception are signaled in these cases. A floating-point trap occurs if either exception is enabled; software is responsible for passing control to the appropriate trap handler.

IEEE Standard 754 specifies that a quiet NaN be generated when an invalid operation occurs with the exception trap disabled, but does not further specify the value generated. The MIPS floating-point architecture specifies that in such cases, the NaN generated shall have a mantissa field of all ones, except for the high-order fractional significand bit (excluding the explicit integer bit in the extended and quad formats). The sign bit shall be positive, and the explicit integer bit, if present, shall be set. A NaN is defined for the word format (i.e. integers). It is used as a result of converting floating-point NaN and Infinities to fixed-point. When the invalid operation exception occurs due to one or more of the operands being signaling NaNs, a new quiet NaN is generated according to the rules above. These values are listed in the following table:

Format	Generated NaN value
single	7fbfffff
double	7ff7ffff ffffffff
extended	7fff0000 00000000 bfffffff ffffffff
quad	7fffbfff ffffffff ffffffff ffffffff
word (MIPS I/II)	7fffffff
word (MIPS III)	7fffffff ffffffff

Note that R3010 and R6010 hardware never generates NaNs, but instead use the unimplemented exception and let software do it. The R4000 does generate the default quiet NaN for operations such as 0 / 0, ∞—∞, etc.

There is nothing in the architecture or software that detects and responds to the Integer NaN pattern. It exists on output-only, to satisfy IEEE754.

3.3.2. Software assistance for IEEE 754 standard compatibility

The standard does not require that all floating-point operations be performed in high-performance hardware, and it does not specify the instruction-set presentation.

When little performance advantage is realized by performing an operation in hardware, the MIPS architecture permits simplifying the hardware and performing the operation using software assistance. Operations which occur with low dynamic frequency may then be implemented in software, while still providing hardware implementations of frequent operations.

The most complex part of IEEE Standard 754 involves fully supporting the required and recommended exceptional conditions that arise in floating-point computation, such as overflow, underflow, and invalid operation. Here again, exception traps are also employed to avoid handling all exceptional conditions in hardware, when applicable. Exceptions that occur with low dynamic frequency are then handled using software assistance.

The MIPS architecture provides the necessary information and interrupts for trapping on exception conditions, but relies extensively on software to implement the IEEE 754 recommendations for support of floating-point exception trap handlers.

3.3.3. IEEE 754 exception trapping

The IEEE 754 floating-point standard makes recommendations on information to be made available during a floating-point exception trap handler. This information often includes the original operand values or other information which must be computed in hardware unless the original operand values are retained.

All of the information that the trap handler must determine can be derived from the state of the floating-point coprocessor are the time of the trap. However, in order to provide significant simplifications in the complexity of the hardware, some computation may need to be performed within the trap handler of an associated software "envelope" to determine the information.

Exception traps may be handled in either a precise or imprecise manner; when the imprecise scheme is employed, while the information recommended by IEEE Standard 754 is made available to the trap handler, other information that is useful for debugging, such as the address of the floating-point instruction which caused the trap, is not available. Special measures must be taken in generating object code when, for example, source-level debuggers are desired in which the statement in which in the floating-point operation was performed is to be identified.

3.3.4. IEEE 754 format compatibility

IEEE Standard 754 requires a 32-bit floating-point format (single), and recommends a 64-bit floating-point format (double). Extended formats are recommended for each format, at least 43 bits for single extended, and 79 bits for double extended, but suggests that support be provided for an extended form of the largest standard format which is implemented; the single extended format is not required or recommended when the double format is provided.

The MIPS floating-point unit uses IEEE Standard 754 single-precision and double-precision floating-point formats, and an IEEE 754-compatible double-extended-precision floating-point format. In the rest of this document, the double-extended format will be called simply the extended format. The single-extended-precision floating-point format is not implemented in the MIPS floating-point architecture.

While the MIPS architecture includes definitions of extended and quad precisions, both are to date unimplemented, either in software and hardware. They may be implemented in software at a later date.

IEEE Standard 754 does not specify the exact format of the extended format. The MIPS floating-point unit uses an 80-bit format for the double extended format that meets the recommendations of IEEE Standard 754 and is compatible with the 80-bit representation used on the Motorola MC68881. The double extended format requires four words of storage in memory and in coprocessor registers. The format is compatible with a 128-bit quad format, not encompassed by IEEE Standard 754, but similar in range and precision to quad formats on IBM and DEC VAX computers.

3.3.5. Compatibility of commercially available devices

The use of commercially available devices for performing floating-point computation that do not fully conform to the standard limits the level of compatibility that can be achieved.

A specific problem is that the closest that the Weitek WTL1164/1165 devices come to implementing the FMOV and FNEG operations is $+0+x$ and $+0-x$, respectively. There are deviations between the result generated by these operations when the operands are -0 for FMOV and $+0$ for FNEG. Because there is no conforming operation provided by the Weitek devices, these deviations cause the operation to be signaled as unimplemented when the result is a zero, requiring that these operations be emulated in software. MIPS VLSI implementations will implement the requirements and recommendations of IEEE Standard 754 precisely.

Some devices may define underflow differently than the requirements of the MIPS floating-point architecture *(after rounding, inexact result)*. When using these devices, operations that produce underflow or denormalized results may use the unimplemented operation exception to guarantee that, as seen by software, underflow is detected the same way in all implementations of the MIPS floating-point architecture.

3.4. Floating-point coprocessor implementation

All of the single, double, and word format floating-point instructions described are available within the MIPS user instruction set. The Coprocessor Unusable exception may be used on some implementations of some MIPS systems to implement these instructions using only software, and the unimplemented operation exception may be used in an implementation that combines software and hardware components.

3.4.1. Software implementation

If no floating-point hardware at all is present, the coprocessor unusable trap is taken on each instruction that refers to the floating-point coprocessor. The instructions are emulated by software routines.

Such implementations incur significant overhead upon execution of even the simplest of these instructions, such as floating-point loads and stores. This overhead is permissible only for systems that have low performance requirements for floating-point operations.

To minimize interrupt latency while retaining modest overhead, load and store coprocessor instructions may be interpreted with interruptions turned off, while the longer floating-point operation instructions can be interpreted after restoring sufficient CPU state to execute with interruptions on.

An alternative emulation strategy may employ "peeking" at the instructions following the interpreted coprocessor instruction and immediately interpreting the following instructions if they are also floating-point coprocessor instructions. While this requires saving enough state to antipate further exceptions as these following instructions are fetched, it may permit a substantial reduction in emulation overhead when floating-point coprocessor operations occur in clusters.

3.4.2. Hardware implementations

While this document defines an architectural level description of the MIPS floating-point coprocessor, it leaves several implementation choices available to designers. Without violating architectural constraints, designers may choose which operations and formats are supported in hardware, the degree and manner of pipelining, and what exceptions are handled in hardware.

3.4.2.1. Optional implementation of operations, formats and exceptions

A theoretically minimal hardware implementation which completely conforms to this architecture specification need not implement any floating-point operations at all, so long as all required registers are implemented. The unimplemented operation exception would be triggered on all floating-point operations, while load, store, move, and branch instructions would be executed directly. Of course, an implementation of this sort has little value, since the exception mechanism is more expensive than that of subroutine calls, and all operations are implemented in software.

A practical minimal implementation might support only the add, subtract, multiply, divide, absolute value, negate, move, and convert operations, and only support the single and double-precision floating-point and single fixed-point formats. The remaining operations and formats may be supported by trapping on unimplemented instructions.

Denormalized numbers may be handled by taking the unimplemented operation exception when they occur, and computing the result in software. The same technique can be applied to NaN handling. Note that these traps are defined optionally, and are provided for the benefit of the designer working at this implementation level.

A more ambitious implementation may extend the hardware to support a extended or even quad floating-point format, requiring wider internal register files and data paths. Externally, the coprocessor register format is the same, due to the use of register quads for long formats.

3.4.2.2. Pipelining and concurrent execution

The MIPS coprocessor architecture permits parallel execution of operations in the coprocessor and the processor, and provides for resynchronization, when required, using interlocks. While pipelining of floating-point

operations is not required of any implementation, the coprocessor and floating-point unit architecture enables implementations to permit pipelining or parallelism with moderate complexity and cost.

The floating-point unit is designed to provide the capability to perform pipelined operations without the need for complex hardware control and optimization techniques. MIPS software employs techniques to optimize the scheduling of instructions, taking into account implementation-dependent or -independent delays.

3.4.2.2.1. Precise exception traps

Floating-point operations, of necessity, require greater latency, than most operations performed by the MIPS R-Series processor. In order to cause the processor to report floating-point exception traps precisely (with the address of the instruction which caused the exception available), the trap must be reported within the relatively short time permitted by the processor.

Normally, this time is much shorter than the time required to perform the operation. If the execution pipeline is permitted to continue while the operation is performed, the processor will have continued past the point at which the trap may be reported precisely by the time the operation completes. A simple solution for making floating-point exceptions precise would be to stall the processor until the operation is completed and all exceptions can be determined.

This would, of course, remove all opportunities for pipelining and overlapping of floating-point loads, stores, and operations as well as processor operations. As such, it would have a significant performance impact.

However, it is often the case that an operation can be determined not to be exception-causing by examination of just a few bits of the operands, or by performing simple operations that are fast enough to be completed in time to appropriately interlock the processor so that it can later report a precise exception trap. This is accomplished by performing simple checks on a subset of the bits of the floating-point operands, ensuring that the dynamic frequency of such interlocks are low. For example, in a single-precision add, if the biased exponent field of both of the operands is less than 192, floating-point overflow will not occur. It is important to note that this is a sufficient but not necessary condition to ensure that floating-point overflow does not occur; the sum of two values whose exponent fields are both greater than 192 may or may not lead to an overflow. The key idea is to make these simple operations "pessimistic," in that they correctly predict all cases in which the operation may cause an exception trap, but they may incorrectly predict a trap when none actually occurs.

For all exception traps, save the inexact exception trap, these pessimistic predictions are easy to produce. For the inexact exception trap, it is assumed that the inexact exception trap is predicted whenever it is enabled. The efficiency of this technique depends on how often the prediction is pessimistic, as in this case, pipelining or overlapping of other operations is eliminated.

3.4.2.2.2. Imprecise Exception Traps

Because of the benefits of using the above technique, imprecise exceptions are not recommended as an implementation strategy, but they are permitted by the architecture. Imprecise exceptions will reduce the ability of software to debug code containing floating-point coprocessor operations, and may cause increased overhead in handling exception traps.

Only one exception trap may be reported from an instruction sequence. The single exception register requires that all operation pipelining be eliminated when any exceptions are possible. This permits implementations to reduce the pipeline control complexity, at the expense of reduced performance when result exceptions may occur. Note that underflow/overflow/inexact exceptions must be predicted in advance, when the traps are enabled, and denormalized results must be handled in hardware or predicted because two pipelined instructions may each produce an exceptional result. The mechanism for performing this determination is similar to that of the precise exception case, except that the pipeline may advance past the instruction while the determination is made, so long as no additional floating-point instructions are permitted to execute in the meantime.

A less aggressively pipelined machine may perform only one floating-point operations at any time, but may permit load and store operations to execute concurrently. This requires an interlock against further use of the single result register specified in the executing operation, and any modification of the source registers, during the period that exception traps may be pending for in the operation. If all such traps are disabled and default

dispositions assigned to the destination when exceptions occur, the source register conflicts may be ignored by the hardware.

3.4.2.3. Simplifying exception handling

Some implementations can reduce the complexity of hardware control by supporting only those operations that use a single pass through available floating-point hardware data paths. In this case, exceptional conditions arise that can require additional passes of processing or software assistance. The unimplemented operation exception trap has been specifically intended for this case.

Except for the inexact flag, each of the "flag bits," which indicate exceptions for which default actions have been taken, may be emulated in software with only a small overall performance penalty. The inexact exception will occur too often for it to be reasonably handled by causing a unimplemented operation exception whenever it occurs.

3.4.3. Standard coprocessor exception trap handler

Provided that the architectural guidelines here are followed, a standard coprocessor exception trap handler will be able to complete a partial implementation, providing software assistance in a manner consistent with the architecture for the optional exception traps and for subset implementations of the floating-point operation and format repertoire.

A standard floating-point instruction set emulator will be able to provide for the absence of floating-point hardware and for continued operation upon failure or removal of the floating-point coprocessor.

3.4.4. Operation timing

Machine interlocks will ensure that software will not depend on the specific timing of floating-point operations to ensure functionality. Therefore, the data below is not strictly architectural, but is clearly relevant to the problem of code generation, given that performance is dependent on the scheduling of floating-point operations with other instructions. MIPS compilers and assemblers use this data to optimize or schedule floating-point operations in an attempt to make the best possible use of computing resources.

The table below gives the minimum latency, in processor clock cycles for each of the floating-point operations for the configurations currently implemented. These latency calculations presume that the result of the operation is immediately used in a succeeding operation. None of these implementations provide extended or quad operations except through software interpretation.

operation	clock cycles											
	R2360			R2010, R3010			R4000			R6010		
fmt	S	D	W	S	D	W	S	D	W	S	D	W
ADD.fmt	11	13	γ	2	2	γ	4	4	γ	3	3-4ξ	γ
SUB.fmt	11	13	γ	2	2	γ	4	4	γ	3	3-4ξ	γ
MUL.fmt	11	15	γ	4	5	γ	7	8	γ	4χ	6χ	γ
DIV.fmt	36	67	γ	12	19	γ	23	36	γ	14χ	24χ	γ
SQRT.fmt	†	†	γ	†	†	γ	54	112	γ	23χ	42χ	γ
ABS.fmt	11	13	γ	1	1	γ	2	2	γ	2	2	γ
MOV.fmt	11	13	γ	1	1	γ	1	1	γ	2	2	γ
NEG.fmt	11	13	γ	1	1	γ	2	2	γ	2	2	γ
ROUND.W.fmt	†	†	γ	†	†	γ	4	4	γ	3	3-4ξ	γ
TRUNC.W.fmt	†	†	γ	†	†	γ	4	4	γ	3	3-4ξ	γ
CEIL.W.fmt	†	†	γ	†	†	γ	4	4	γ	3	3-4ξ	γ
FLOOR.W.fmt	†	†	γ	†	†	γ	4	4	γ	3	3-4ξ	γ
CVT.S.fmt	γ	13	11	γ	2	3	γ	4	6	γ	3-4ξ	3
CVT.D.fmt	13	γ	13	1	γ	3	2	γ	5	3	γ	3
CVT.W.fmt	11	13	γ	2	2	γ	4	4	γ	3	3-4ξ	γ
C.fmt.cond	11α	13α	γ	2α	2α	γ	3α	3α	γ	2α	2α	γ
BC1T		1			1			1			1	
BC1F		1			1			1			1	
BC1TL		δ			δ			1			1	
BC1FL		δ			δ			1			1	
LWC1		2α			2α			3			2-3β	
SWC1		·1			1			1			1-2β	
LDC1		α†			α†			3			2-3β	
SDC1		†			†			1			2	
MTC1		2α			2α			3α			2α	
MFC1		2			2			3			2	
CTC1		2α			2α			3α			6α	
CFC1		2			2			3			2	

α Software *must* schedule operations to avoid reading the floating-point register that is the target of a floating-point load or move to floating-point unit instruction less than 2 instructions later, and must schedule a floating-point branch instruction 2 or more instructions after a floating-point compare instruction.

β Use smaller figure for most operations; larger figure is the next instruction is a store: *SWC1, SDC1, SW, SWL, SWR, SH, SHU, SB, SBU* or move-to-or-from coprocessor: *MTCz, CTCz, MFCz, CFCz*

γ These operations are illegal.

δ These operations are not available on MIPS I processors.

† These operations are only provided through software interpretation.

ξ Operation latency depends on the preceeding instructions and the operation that produced this instruction's operands.

χ If the R6010 is programmed to use the 60MHz latencies for the B3110 chip, the latencies are: MUL.S 4; MUL.D 5; DIV.S 12; DIV.D 20; SQRT.S 20; SQRT.D 36.

3.4.4.1. Pipelining and overlapping

The floating-point architecture permits pipelining of operations and overlapping of floating-point operations with floating-point loads, stores, moves and with other processor operations. Current implementations use these features to varying degrees.

The R2360 floating-point board permits fixed-point operations to continue execution after issuing a floating-point operation. An arbitrary number of non-floating-point instructions may therefore be overlapped with any floating-point operation. However, floating-point instructions execute serially, and floating-point loads and stores will stall for execution until the coprocessor is no longer executing a floating-point operation.

The R2010 and R3010 floating-point chips permit loads and stores to execute concurrently with floating-point operations, provided that they do not modify or use the result registers of the executing operation. Fixed-point operations may also execute concurrently with floating-point operations. The R2010 and R3010 floating-point chips also will permit floating-point operations to be overlapped with each other in certain combinations. In order to provide precise floating-point exceptions, certain combinations of operands which would appear to possibly cause an exception will selectively eliminate concurrent operations.

The R6010 floating-point controller provides precise floating-point operations, in the form of the R2010, R3010, and R4000 designs. Loads and stores as well as fixed-point can execute concurrently with floating-point operations. Multiplies, divides, and square roots execute in one floating-point unit, and the other operations execute in another. Operations in the same unit cannot overlap, but operations in different units can do so. The R6010 lacks bypassing in some places, and so the latency of most operations is one cycle longer than the computation time of the floating-point unit.

The R4000 implements floating-point arithmetic together with the processor. Both the processor and floating-point use more pipelining than other MIPS processors, and thus the latency of R4000 operations are longer when measured in cycles, but are comparable or significantly faster when measured in absolute time. The longer cycle counts do make it profitable to partially pipeline some of the floating-point units. R4000 floating-point is fully bypassed.

The R4000 floating-point implements separate units for multiply, divide, and one for everything else (generally called the adder). Multiplies and divides can overlap with adder operations, but use the adder on their final cycles, which imposes some limitations. The multiply unit can begin a new double precision multiply every 4 cycles, and a new single precision multiply every 3 cycles. The adder generally can begin a new operation one cycle before the last one completes; a floating-point add or subtract therefore can start every 3 cycles. For further details, refer to the R4000 specification.

3.4.5. Software implementation of IEEE 754 standard operations

Some of the operations required or recommended by IEEE Standard 754 are not provided directly in hardware. These operations are not implemented in the floating-point instruction set either because of their high complexity, low frequency of use, or redundancy with the set of implemented instructions.

Code skeletons for the implementation of operations of these categories are given below.

3.4.5.1. Remainder

The "remainder" function is accomplished by repeated magnitude subtraction of a scaled form of the divisor, until the dividend/remainder is one-half of the divisor or until the magnitude is less than one-half of the magnitude of the divisor. The scaling of the divisor ensures that each subtraction step is exact; thus the remainder function itself is always exact.

3.4.5.2. Round to integer

The "round to integer in floating-point format" function may be implemented by adding a bias that causes normal rounding to occur at the end of the floating-point fraction, and then subtracting it back again. The code example is given for single-precision; double precision is similar. As always with IEEE 754 floating point, handling NaNs and –0 requires care.

```
/* Single-precision round to integer using current rounding mode */
/* Operand is in f12 */
/* Result placed in f0 */
rint:
        li.d      $f4, 4503599627370496.0    /* 2^52 */
        abs.d     $f2, $f12
        c.olt.d   $f2, $f4                   /* if |arg|≥2^52 or arg is NaN */
        mfc1      t0, $f13
        bc1f      4f                         /* then done */
         mov.d    $f0, $f12
        /* < 2^52 */
        bgez      t0, 2f                     /* if input negative, negate result */
         sll      t1, t0, 1
        /* negative */
        beq       t1, 0, 3f                  /* possible -0 */
         nop
1:      sub.d     $f0, $f12, $f4
        j         ra
         add.d    $f0, $f4
2:      /* positive */
        add.d     $f0, $f12, $f4             /* bias by 2^52 to force
                                                non-integer bits off the end */
        j         ra
         sub.d    $f0, $f4                   /* unbias */
3:      /* msw = 80000000 */
        mfc1      t1, $f12                   /* if -0, return -0 */
        nop
        bne       t1, 0, 1b                  /* if negative denorm, process that */
         nop
4:      j         ra
         nop
```

3.4.5.3. Convert between binary and decimal

The details of this operation are beyond the scope of this document. 64-bit integer arithmetic can be used to obtain the accuracy required by IEEE Standard 754-1985.

3.4.5.4. Copy sign

The copysign operation can be performed using integer compares and the absolute value and negation operations. Special attention must be paid to negative zero, as it has negative sign, but zero value.

```
/* Single-precision copysign */
/* Operands in f0, f2 */
/* Result placed in f0 */
        mfc1    t0, f2
        bgez    t0, 1f
        abs.s   f0
        neg.s   f0
1:
```

3.4.5.5. Scale binary

This conceptually simple operation is significantly complicated by the need to handle NaNs, infinities, denormalized numbers, zeros, etc. For this reason, when $E_{min} \leq scale \leq E_{max}$ it is best to use a suitably constructed multiply. For scale values outside of this range, extensive case analysis is required. A complete code sequence is too lengthy to include here.

3.4.5.6. Log binary

This operation is performed by moving the operand to the processor, where shift and add operations perform the basic operation. As usual, much of the implementation involves handling NaNs, infinities, zeros, and signaling exceptions if necessary.

3.4.5.7. Next after

This operation is performed by comparing the two floating-point values to determine the direction to compute the neighbor, then moving the operand to the processor, where single-precision or multiple-precision add operations perform the basic operation.

3.4.5.8. Finite

```
/* Single-precision finite */
/* Operands in f0 */
/* Result placed in v0 */
        mfc1    v0, f0
        nop
        sll     v0, 1
        srl     v0, 23+1
        sltu    v0, v0, 255
```

3.4.5.9. Is NaN

This operation is provided by using the "unordered" predicates of the floating-point compare operation.

3.4.5.10. Arithmetic inequality

This operation is available as the floating-point compare operation.

3.4.5.11. Class

This operation is performed by moving the operand to the processor, where fixed-point shifts and comparisons can classify the floating-point value.

3.4.6. Software implementation of IEEE 754 standard trap handlers

In order to fully conform to the recommended capabilities of IEEE Standard 754 with respect to trapping on exceptional conditions, software assistance may be required. In general, this assistance is provided by using the software floating-point implementation, which is complete in these respects, to re-execute the operation and handle these exceptions. The unimplemented exception trap leaves the original operand values available in the

floating-point registers for all floating-point exception traps.

Beyond providing a full implementation of the MIPS floating-point architecture, software provides a mechanism for passing all of the information recommended by IEEE Standard 754 to user-written trap handlers. This software is responsible for calculating biased results for the case of the overflow and underflow handlers, as well as replacing default results, or results calculated by the trap handler into the original destination registers of the operation.

3.4.7. Implementation in single-chip VLSI

Single-chip VLSI implementations include the R2010 and R3010 floating-point coprocessors. R4000 processors implement the floating-point coprocessor on the same chip as the cpu. The R6010 implements multiply, divide, and square root in one chip, and all other functions in a second chip.

IEEE Standard 754 includes features which are relatively costly to implement in hardware; these include denormalized numbers, NaN handling and IEEE 754 exception handling. Most of these features are newly developed for the standard and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these features can be implemented with assistance provided by software, maintaining full compatibility with the standard.

In the specific cases where the device's exception handling diverges from the standard, one of the optional exception classes is indicated, so that software may take the appropriate corrective action.

This section details the conditions under which the optional exception traps are employed, and the action taken by software under these conditions.

3.4.7.1. Unimplemented instruction

An attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the unimplemented cause bit and traps. The operand and destination registers are undisturbed.

The instruction is emulated in software. Any of the IEEE 754 exceptions may arise from the emulated operation; these exceptions are in turn simulated.

3.4.7.1.1. Extended or quad-precision

VLSI implementations do not support the extended or quad-precision formats; when an operation in this class is attempted, the unimplemented operation cause bit is set, which causes a trap. To the VLSI these operations are simply unimplemented instructions.

3.4.7.1.2. Square root

The R2010 and R3010 implementations do not support the square root operation; when an operation in this class is attempted, the unimplemented operation cause bit is set, which causes a trap. To the VLSI these operations are simply unimplemented instructions.

In the case of single, double precision square-root operations, implemented floating-point unit instructions may be used to compute the result.

3.4.7.1.3. Denormalized operand

When one or both of the source registers contain denormalized numbers, the unimplemented operation cause bit is set, which causes a trap.

The operation is then simulated in software. By scaling the operands, implemented floating-point operations may be employed to simulate the operation.

3.4.7.1.4. Quiet Not a Number operand

Except for comparisons, when one or both of the source registers contain quiet NaN values, the unimplemented operation cause bit is set, which causes a trap. The original source values are still available to permit simulation of the operation in the trap handler.

3.4.7.2. Invalid operation exception

Invalid operation occurs in the following cases:

(1) One or both operands is a signaling NaN (except in the MOV.fmt instructions)

(2) Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty)+(-\infty)$ or $(-\infty)-(-\infty)$

(3) Multiplication: $0\times\infty$, with any signs

(4) Division: $\frac{0}{0}$ or $\frac{\infty}{\infty}$, with any signs

(5) Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format.

(6) Comparison of predicates involving "<" or ">" without "?," when the operands are "unordered"

If invalid is enabled, VLSI sets the invalid cause bit and traps. For R2010, R3010, and R6010 coprocessors, if invalid is not enabled, VLSI sets the unimplemented bit and traps. For R4000 processors, if invalid is not enabled, VLSI returns the default quiet NaN.

3.4.7.3. Division-by-zero exception

The division by zero exception occurs on a divide operation if the divisor is zero and the dividend is a finite nonzero number. The division-by-zero cause bit is set. When division-by-zero is enabled, a trap occurs and the operands and destination are not modified. If division-by-zero is not enabled, VLSI also sets the division-by-zero flag and the result is a correctly signed ∞.

3.4.7.4. Overflow

Operation overflow sets the overflow cause bit. When overflow is enabled, a trap occurs and the operands and destination are not modified. If overflow is not enabled, VLSI also sets the overflow flag, and the result is determined by the rounding mode and the sign of the intermediate result, in accordance with IEEE Standard 754.

3.4.7.5. Underflow exception

IEEE Standard 754 provides for some variation between systems for the detection of the underflow condition, but require that for a particular system, underflow is always detected the same way for all operations and formats. As described in chapter 4, section 1, there are two conditions that affect the determination of the underflow condition; "tininess" and "loss of accuracy." VLSI implementations detect tininess "after rounding," and need not detect loss of accuracy. The underflow exception is never raised; instead tininess sets the unimplemented operation cause bit, which causes a trap. Software calculates a properly rounded denormalized result, and possibly signals underflow.

3.4.7.5.1. Denormalized result

When the a result is generated that can only be represented as a denormalized number in the specified format, the unimplemented operation cause bit is set, which causes a trap.

In this event, software performs the operation, from the beginning, using the contents of the source operands. If an underflow or inexact exception is called for, it is simulated in software.

3.4.7.6. Inexact exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact cause bit is set. When enabled, this causes a trap and no result is stored. If not enabled, the rounded or overflowed result is delivered to the destination register.

3.4.8. Implementation using commercially available floating-point devices

To allow the use of commercially available floating-point computational devices, such as the Weitek WTL1164/WTL1165, or BIT B3110/B3120, loopholes are provided in the architecture to that the manner of

exception handling provided by the devices, which is not entirely compatible with IEEE Standard 754, can be employed in a logically transparent manner.

In the specific cases where the device's exception handling diverges from the standard, one of the optional exception classes is indicated, so that software may take the appropriate corrective action.

This section details the conditions under which the optional exception trap are employed, and the action taken by software under these conditions.

3.4.8.1. Invalid operation exception

The invalid operation exception is signaled if one or both of the operands are invalid for an implemented operation. The result, when the exceptions occurs without a trap, is a quiet NaN. When trapped, the exception leaves the original operand values undisturbed, and copies the instruction into the exception instruction register.

All conditions under which the invalid operation exception are to be signaled are detected by the Weitek devices:

(1) Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty)-(-\infty)$

(2) Multiplication: $0\times\infty$, with any signs

(3) Division: $\frac{0}{0}$ or $\frac{\infty}{\infty}$, with any signs

(4) Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format (Weitek devices indicate an "underflow," "overflow," or "NaN" condition in this case.)

(5) Comparison of predicates involving "<" or ">" without "?," when the operands are "unordered" (Weitek devices indicates "unordered" condition in this case.)

The BIT devices likewise detect all cases in which the invalid operation exception are to be signaled:

(1) Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty)-(-\infty)$

(2) Multiplication: $0\times\infty$, with any signs

(3) Division: $\frac{0}{0}$ or $\frac{\infty}{\infty}$, with any signs

(4) Square root: any negative value (except -0)

(5) Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format (BIT devices set "UF," "OV," or "NaN" flags in this case.)

(6) Comparison of predicates involving "<" or ">" without "?," when the operands are "unordered" (BIT devices set "NaN" flag in this case.)

3.4.8.2. Division-by-zero exception

The division by zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. The result, when no trap occurs, is a correctly signed ∞. Weitek and BIT devices detect this exception and indicate it on the condition and flags outputs. When trapped, the exception leaves the original operand values undisturbed, and copies the instruction into the exception instruction register.

3.4.8.3. Overflow exception

The Weitek and BIT devices will only handle untrapped overflow exceptions; if the overflow exception trap is enabled, the result generated by the Weitek or BIT devices must not be placed into the result register. Software then calculates the correct result and simulates the trapped overflow exception.

If the overflow exception trap is not enabled, the coprocessor can use the default result provided by the Weitek or BIT devices. The result is determined by the rounding mode and the sign of the intermediate result.

3.4.8.4. Underflow exception

The Weitek and BIT devices do not handle trapped or untrapped underflow exceptions; the coprocessor instead uses the "unimplemented operation" exception when this condition occurs, and suppresses the result provided by the Weitek and BIT devices. Software then calculates the correct result and simulates the underflow exception, which may be either trapped or untrapped.

IEEE Standard 754 provides for some variation between systems for the detection of the underflow condition, but require that for a particular system, underflow is always detected the same way for all operations and formats. As described in chapter 4, section 1, there are two conditions that affect the determination of the underflow condition; "tininess" and "loss of accuracy." The Weitek devices detect tininess "after rounding." Whenever a result is tiny, these devices produce a value can be converted with a "wrap" operation which takes the truncated, biased value and produces a zero or denormalized result. The "loss of accuracy" condition relating to untrapped overflow provided by the Weitek devices is "inexact result." It is therefore required that operations implemented in software determine underflow by detecting tininess "after rounding," but, because the "wrapped" result will not be used, either method of detecting loss of accuracy is acceptable in hardware provided the unimplemented operation exception is signaled whenever an underflow is detected on devices that detect underflow "before rounding," so that the corner cases that differ will be computed in software. Detection of loss of accuracy by "inexact result" is usually a simpler matter than the detection of "denormalization loss," so most implementations should choose "inexact result." An implementation that uses devices that detect underflow as "denormalization loss" must use the unimplemented operation exception for all denormalized results, unless the "inexact result" condition can be otherwise determined.

3.4.8.5. Inexact exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception is signaled. The rounded or overflowed result is delivered to the destination register, if no trap occurs. Weitek devices detect this exception and indicate it on the condition output. BIT devices detect this exception and indicate it on the INX flag.

3.4.8.6. Unimplemented operation exception

This exception is raised when an attempt is made to execute an instruction with an operation code or format code which has been reserved for future architectural definition.

The exception leaves the original operand values undisturbed, and copies the instruction into the exception instruction register, when present.

The instruction is emulated in software. Any of the IEEE 754 exceptions may arise from the emulated operation; these exceptions are in turn simulated.

3.4.8.6.1. extended- and quad-precision

The Weitek and BIT devices do not support the extended- or quad-precision formats; when an operation in this class is attempted, the unimplemented operation exception is signaled.

3.4.8.6.2. square root

The Weitek devices do not support the square root operation; when an operation in this class is attempted, the unimplemented operation exception is signaled. In the case of single and double precision square-root operations, implemented floating-point unit instructions may be used to compute the result.

The BIT devices implement this function directly.

3.4.8.6.3. denormalized operand

When one or both of the source registers contain denormalized numbers, the Weitek devices indicate this event by the contents of the condition output. While the devices show which operand was denormalized, this information is not made available in the control and status register. When this event occurs, a unimplemented operation exception is signaled, which always causes a trap.

The BIT devices detect incoming denormalized operands and set the DX or DY flags. The DEN bit of the interrupt enable register should be set to make this case cause an interrupt, as the correct results is not generated by the BIT multiplier.

The operation is then simulated in software. By scaling the operands, implemented floating-point operations may be employed to simulate the operation.

3.4.8.6.4. Not a Number operand

When one or both of the source registers contain NaN values, the Weitek and BIT devices indicate this event by the contents of the condition or flag outputs. While the Weitek device show which operand was a NaN, the BIT device does not, so this information is not made available in the control and status register. When this event occurs, the unimplemented operation exception is signaled, which always causes a trap. The original source values are still available to permit simulation of the operation in the trap handler.

If one of the operands is a signaling NaN, the invalid operation exception is simulated by software.

3.4.8.6.5. Denormalised result

When the Weitek devices indicate an "underflow" or "underflow + inexact" condition on any operation but conversion, the unimplemented operation exception is signaled, which always causes a trap.

When the BIT devices are used, the "UF" flag causes the signaling of the unimplemented operation exception, which causes a trap. The wrapped result generated by the BIT devices is ignored, and not placed into the floating-point registers.

In this event, software performs the operation, from the beginning, using the contents of the source operands. If an underflow or inexact exception is called for, it is simulated in software.

4. System Control Coprocessor

The system control coprocessor translates virtual addresses into physical addresses, and manages exceptions and transitions between kernel and user states. It also controls the cache subsystem and provides diagnostic control and error recovery facilities. In some processors, a generic system timer facility is provided for interval timing, time-keeping, process accounting and time-slicing. All special, privileged registers and instructions associated with these facilities are encapsulated within the coprocessor mechanism as coprocessor zero.

Access to these registers and functions may be restricted by setting coprocessor zero to an "unusable" state. When the processor is executing in kernel mode, the usability of coprocessor zero is ignored; thus requests to manipulate coprocessor zero from kernel mode are always granted. The kernel may grant unrestricted access to system control coprocessor registers by setting coprocessor zero to a usable state.

In this chapter, the system control coprocessor registers and operations through which exceptions are handled and serviced are described. Chapter 5 details, for each type of exception, the conditions which cause it, how the exception system handles it, and how the exception may be serviced by an operating system.

4.1. System Control Coprocessor Compatibility

The MIPS I, MIPS II, and MIPS III architectures define the operation of the processor and floating-point coprocessor. However, except for the user-mode virtual address space, the MIPS system control coprocessor is outside of these specifications; each implementation may be different, and in general the operating system kernel for one processor series will not operate on another processor series. This book documents the current implementations, which are the R2000, R3000, R6000, and the R4000 series of processors. We use the labels MIPS I, etc. when a feature is tied to a particular processor/floating-point architecture definition, and the labels R2000, etc. when a feature is tied to a particular implementation.

4.2. MIPS III

MIPS III processors execute either in MIPS II compatibility mode, or in MIPS III native mode. In MIPS II mode the address space is limited to 2^{32} bytes, and MIPS III instructions cause reserved instruction exceptions (except in kernel mode). MIPS II operation on MIPS III processors is defined on the full MIPS III processor's state, and in general keeps 32-bit data in sign-extended form in 64-bit containers. This minimizes the hardware required to implement both MIPS II and MIPS III together.

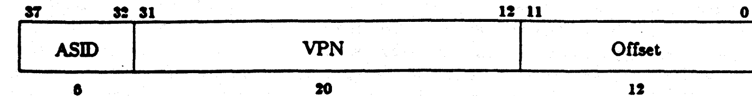
In MIPS III mode an extended address space of VSIZE bits per mode is available. VSIZE is implementation-dependent, and may range from 36 to 62 bits. Also available are the MIPS III instructions, which operate on 64-bit data.

MIPS II or MIPS III operation is selected independently for kernel, user, and supervisor modes in the Status register (the KX, SX, and UX bits). Thus a MIPS III kernel may support both MIPS II and MIPS III user programs.

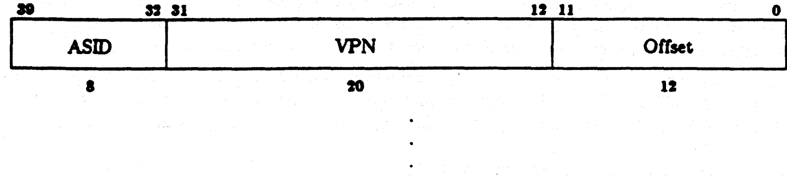
4.3. Memory System Architecture

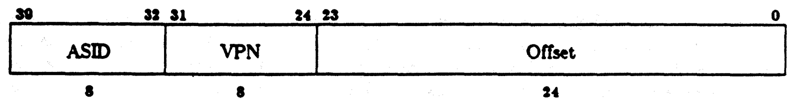
The virtual memory system provides for the logical expansion of the physical memory space of the machine by translating addresses composed in a large virtual address space into the physical memory space of the machine. The number of bits in a physical address is PSIZE. For R2000 and R3000 processors, PSIZE = 32, and virtual address mapping uses 4096-byte pages; thus, mapping affects only the most-significant 20 bits of a 32-bit virtual address, namely the virtual page number (VPN); the remaining 12 bits (Offset) are passed along unchanged. For R6000 processors, PSIZE = 36, pages are 16384 bytes, with an 18-bit VPN and a 14-bit offset. For R4000 processors, PSIZE = 36, and page sizes are variable from 4K bytes to 16M bytes. The virtual address is extended with an address space identifier (ASID) to reduce the frequency of TLB flushing on context switch. The size of the ASID field is 6 bits in R2000 and R3000 processors and 8 bits in R4000 and R6000 processors. The ASID is contained in the system control coprocessor's EntryHi register (described later).

Extended virtual address (R2000 and R3000):

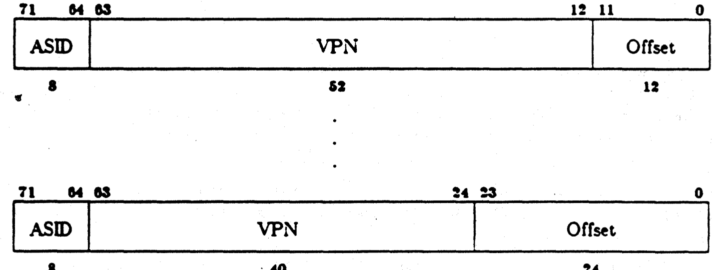


Extended virtual address (MIPS II R4000 4KB-16MB):





Extended virtual address (MIPS III R4000 4KB-16MB):



The manner in which addresses are mapped depends on the execution state of the processor; it is different for kernel and user mode states. To simplify the management of user state from within the kernel, (and to simplify the hardware mapping mechanism), the user-mode address space is a proper subset of the kernel-mode address space.

4.3.1. MIPS I and II User-mode Virtual Addressing

MIPS I and II processors operating in user mode provide a single, uniform virtual address space of 2^{31} bytes. All valid user-mode virtual addresses have the most-significant bit cleared; an attempt to reference an address with the most-significant bit set causes an Address Error exception.

The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. The mapping of these extended virtual addresses to physical addresses need not be one-to-one; two virtual addresses are permitted to map to the same physical address. However, software may impose certain restrictions to avoid coherency problems in virtual index caches.

The figure below shows the MIPS I and II user-mode virtual address space:

Address	Region
0xffffffff – 0x80000000	Unavailable (address error)
0x7fffffff – 0x00000000	User virtual address space (useg) Mapped, 2 GB

4.3.2. R2000, R3000, R6000 Kernel-mode Virtual Addressing

For R2000, R3000, or R6000 processors operating in kernel mode, four distinct virtual address spaces are simultaneously available, differentiated by high-order bits of the virtual address.

When the most-significant bit of the virtual address is cleared, the virtual address space selected covers the full 2^{31} bytes of the current user address space (kuseg). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses.

When the most-significant three bits of the virtual address are "100," the virtual address space selected is a 2^{29}-byte kernel physical space (kseg0). References to this space are not mapped; the physical address selected is defined by subtracting 0x80000000 from the virtual address. Caches are enabled for accesses to these addresses.

When the most-significant three bits of the virtual address are "101," the virtual address space selected is a 2^{29}-byte kernel physical space (kseg1). References to this space are not mapped; the physical address selected is defined by subtracting 0xa0000000 from the virtual address. Caches are disabled for accesses to these addresses; physical memory (or memory-mapped I/O device registers) are accessed directly on these references.

When the most-significant two bits of the virtual address are "11," the virtual address space selected is a 2^{30}-byte kernel virtual space (kseg2). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses.

The figure below shows the R2000, R3000, R6000 kernel-mode virtual address space:

Address	Space
0xffffffff 0xc0000000	Kernel virtual address space (kseg2) Mapped, 1 GB
0xbfffffff 0xa0000000	Uncached kernel physical address space (kseg1) Unmapped, 0.5 GB
0x9fffffff 0x80000000	Cached kernel physical address space (kseg0) Unmapped, 0.5 GB
0x7fffffff 0x00000000	User virtual address space (kuseg) Mapped, 2 GB

4.3.3. MIPS II R4000 Supervisor-mode Virtual Addressing

R4000 processors support a supervisor mode in addition to user and kernel modes. Supervisor mode is intended for layered operating system implementations where a true "kernel" runs in R4000 kernel mode, and the rest of the operating system runs in supervisor mode. In particular, it is intended for building highly secure systems. Because kernel mode has unrestricted access to the processor and its memory, the program that runs there must be completely trusted and should be subject to stringent verification. Verification is simpler when the kernel mode program is small, which is accomplished by running the bulk of the operating system in supervisor mode with no access to the system coprocessor. Supervisor mode and the supervisor address space may be ignored when they are not appropriate. The supervisor address space is then simply part of kernel mapped space.

When KSU = 01 and EXL = 0 and ERL = 0 in the Status register, the processor is executing in supervisor-mode. If SX = 0 then the virtual address space is 2.5G bytes, divided into two regions, differentiated by high-order bits of the virtual address.

When the most-significant bit of the virtual address is cleared, the virtual address space selected covers the full 2^{31} bytes of the current user address space (suseg). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. TLB misses on references to suseg select the TLB refill or XTLB refill exception vector based on the UX bit of the Status register. The base register used to form this virtual address must also have bit 31 clear, or the calculation is not defined, and may result in an exception, or access to a different location.

When the most-significant three bits of the virtual address are "110," the virtual address space selected is a 2^{29}-byte supervisor virtual space (sseg). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. TLB misses on references to sseg use the TLB refill exception vector because SX = 1 in the Status register. The base register used to form this virtual address must also have bits 31..29 equal to "110," or the calculation is not defined, and may result in an exception, or access to a different location.

The figure below shows the R4000 supervisor-mode virtual address space:

Address	Region
0xffffffff 0xe0000000	(address error)
0xdfffffff 0xc0000000	Supervisor virtual address space (sseg) Mapped, 0.5 GB
0xbfffffff 0xa0000000	(address error)
0x9fffffff 0x80000000	(address error)
0x7fffffff 0x00000000	User virtual address space (suseg) Mapped, 2 GB

4.3.4. MIPS II R4000 Kernel-mode Virtual Addressing

When a R4000 processor is operating in kernel mode (KSU = 00 or EXL = 1 or ERL = 1 in the Status register) and 32-bit mode is selected (KX = 0 in the Status register), the virtual address space is 2^{32} bytes, divided into five regions, differentiated by high-order bits of the virtual address.

When the most-significant bit of the virtual address is cleared, the virtual address space selected covers the full 2^{31} bytes of the current user address space (kuseg). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. TLB misses on references to kuseg select the TLB refill or XTLB refill exception vector based on the UX bit of the Status register. The base register used to form this virtual address must also have bit 31 clear, or the calculation is not defined, and may result in an exception, or access to a different location.

When the most-significant bit of the virtual address is set, bits 30..29 further select the address space. In this situation, the virtual address calculation

$$vAddr \leftarrow ((offset_{15})^{16} \,\|\, offset_{15..0}) + GPR[base]$$

must not overflow from bits 28..0 into bits 31..29; if $vAddr_{31..29} \neq GPR[base]_{31..29}$ then the virtual address is not defined and may result in an exception, or access to a different location.

When the most-significant three bits of the virtual address are "100," the virtual address space selected is a 2^{29}-byte kernel physical space (kseg0). References to this space are not mapped; the physical address selected is defined by subtracting 0x80000000 from the virtual address. Cacheability and coherency are controlled by the K0C field of the Config register.

When the most-significant three bits of the virtual address are "101," the virtual address space selected is a 2^{29}-byte kernel physical space (kseg1). References to this space are not mapped; the physical address selected

is defined by subtracting 0xa0000000 from the virtual address. Caches are disabled for accesses to these addresses; physical memory (or memory-mapped I/O device registers) are accessed directly on these references.

When the most-significant three bits of the virtual address are "110," the virtual address space selected is a 2^{29}-byte supervisor virtual space (ksseg). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. TLB misses on references to ksseg select the TLB refill or XTLB refill exception vector based on the SX bit of the Status register.

When the most-significant three bits of the virtual address are "111," the virtual address space selected is a 2^{29}-byte kernel virtual space (kseg3). The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. TLB misses on references to kseg3 use the TLB refill exception vector because KX = 1 in the Status register.

As a special feature for the ECC handler, if the ERL bit of the Status register is set, the user address region becomes a 2^{31}-byte unmapped, uncached space, similar to kseg1. This allows the ECC exception code to operate uncached using R0 as a base register.

The figure below shows the R4000 kernel-mode virtual address space:

Address range	Space
0xffffffff – 0xe0000000	Kernel virtual address space (kseg3) Mapped, 0.5 GB
0xdfffffff – 0xc0000000	Supervisor virtual address space (ksseg) Mapped, 0.5 GB
0xbfffffff – 0xa0000000	Uncached kernel physical address space (kseg1) Unmapped, 0.5 GB
0x9fffffff – 0x80000000	Cached kernel physical address space (kseg0) Unmapped, 0.5 GB
0x7fffffff – 0x00000000	User virtual address space (kuseg) Mapped, 2 GB

4.3.5. MIPS III User-mode Virtual Addressing

(At present, only R4000 processors implement MIPS III addressing.)

When KSU = 10 and EXL = 0 and ERL = 0 in the Status register, the processor is executing in user-mode and the following rules define addressing.

When UX = 0 in the SR, user-mode addressing is compatible with MIPS I/II: if $vAddr_{31}=0$ the address references a 2^{31}-byte user address space, otherwise an address exception occurs. Bits 63..32 are cleared when accessing the TLB. The TLB Refill exception vector is used for TLB misses.

When UX = 1 in the SR, MIPS III processors operating in user mode provide a single, uniform virtual address space of at least 2^{36} bytes, and as large as 2^{62} bytes (xuseg). Different MIPS III implementations may choose to implement different virtual address space sizes, so long as they trap on addresses larger than implemented. When VSIZE bits are implemented, all valid user-mode virtual addresses have bits 63..VSIZE zero; an attempt to reference an address with $vAddr_{63..VSIZE} \neq 0$ causes an Address Error exception. The Extended addressing TLB Refill exception vector is used for TLB misses.

When UX = 1 in the SR, the virtual address calculation

$$vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$$

must not overflow from bits 61..0 into bits 63..62; if $vAddr_{63..62} \neq GPR[base]_{63..62}$ then the virtual address is not defined and may result in an exception, or access to a different location. For compatibility with MIPS I implementations, a similar restriction is *not* imposed on $vAddr_{31}$ when UX = 0; the addition of the offset to a base register with bit 31 set is permitted to yield $vAddr_{31}=0$, although this practice is discouraged in new applications.

The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. The mapping of these extended virtual addresses to physical addresses need not be one-to-one; two virtual addresses are permitted to map to the same physical address. However, software may impose certain restrictions to avoid coherency problems in virtual index caches.

The figure below shows the MIPS III user-mode virtual address space:

$2^{64}-1$	Unavailable (address error)
$2^{VSIZE}+0$	
$2^{VSIZE}-1$	User virtual address space (xuseg) Mapped
0	

4.3.6. MIPS III Supervisor-mode Virtual Addressing

When KSU = 01 and EXL = 0 and ERL = 0 in the Status register, the processor is executing in supervisor-mode and the following rules define addressing.

When SX = 0 in the Status register, supervisor-mode addressing is compatible with MIPS II: if $vAddr_{31}=0$ then bits 63..32 must also be zero, and the address references a 2^{31}-byte user address space; if bits 31..29 are "110," then bits 63..32 must be all ones, and the address references a 2^{29}-byte supervisor address space (sseg). Other address references cause Address Error exceptions.

When SX = 1 in the SR, MIPS III processors operating in supervisor mode provide two virtual address spaces of at least 2^{36} bytes, and as large as 2^{62} bytes, as well as a 2^{29}-byte space.

The virtual address calculation

$$vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$$

must not overflow from bits 61..0 into bits 63..62; if $vAddr_{63..62} \neq GPR[base]_{63..62}$ then the virtual address is not defined and may result in an exception, or access to a different location. Likewise, overflow from bits 28..0 to 63..29 is not defined for references to ksseg.

When bits 63..62 of the virtual address are "00," the virtual address space selected is the 2^{VSIZE} bytes of the current user address space (xsuseg). $vAddr_{61..VSIZE} \neq 0$ causes an address error.

When bits 63..62 of the virtual address are "01," the virtual address space selected is the 2^{VSIZE} bytes of the current supervisor address space (xsseg). $vAddr_{61..VSIZE} \neq 0$ causes an address error.

When bits 63..62 of the virtual address are "11," and bits 31..29 equal to "110," the address references a separate 2^{29}-byte supervisor address space (sseg).

While the MIPS II user and kernel address spaces are subsets of their corresponding MIPS III equivalents, there are actually separate, disjoint MIPS II and MIPS III supervisor address spaces. Both may be used simultaneously.

The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses. The mapping of these extended virtual addresses to physical addresses need not be one-to-one; two virtual addresses are permitted to map to the same physical address.

The figure below shows the MIPS III supervisor-mode virtual address space:

Address	Region
$2^{64}-1$	address error
$2^{64}-2^{32}$+0xe0000000	
$2^{64}-2^{32}$+0xdfffffff	Supervisor MIPS II virtual address space (sseg) Mapped, 0.5 GB
$2^{64}-2^{32}$+0xc0000000	
	Unavailable (address error)
$1\times2^{62}+2^{VSIZE}+0$	
$1\times2^{62}+2^{VSIZE}-1$	Supervisor virtual address space (xsseg) Mapped
$1\times2^{62}+0$	
$1\times2^{62}-1$	address error
$2^{VSIZE}+0$	
$2^{VSIZE}-1$	User virtual address space (xsuseg) Mapped
0	

4.3.7. MIPS III R4000 Kernel-mode Virtual Addressing

When KSU = 00 or EXL = 1 or ERL = 1 in the Status register, the processor is executing in kernel-mode and the following rules define addressing for MIPS III R4000 processors.

When KX = 0 in the Status register, kernel-mode addressing is compatible with MIPS II: if $vAddr_{31}=0$ then bits 63..32 must also be zero, and the address references a 2^{31}-byte user address space; if bit 31 is set, then bits 63..32 must also be set, and the address references the 32-bit kernel or supervisor address spaces. Because bits $vAddr_{63..32}$ must be the same as $vAddr_{31}$, the kernel is prohibited from using an address calculation with 32-bit two's complement overflow. There are two such cases, which are in error: first, an offset with bit 15 clear and a base register with bit 31 clear where the sum has bit 31 set; second, an offset with bit 15 set and base with bit 31 set where the sum has bit 31 clear.

When KX = 1 in the Status register, extended addressing in kernel-mode is enabled and four distinct virtual address spaces are simultaneously available, differentiated by the high-order bits of the virtual address. The virtual address calculation

$$vAddr \leftarrow ((offset_{15})^{48} \,\|\, offset_{15..0}) + GPR[base]$$

must not overflow from bits 61..0 into bits 63..62; if $vAddr_{63..62} \neq GPR[base]_{63..62}$ then the virtual address is not defined and may result in an exception, or access to a different location.

When bits 63..62 of the virtual address are "00," the virtual address space selected is the 2^{VSIZE} bytes of the current user address space. $vAddr_{61..VSIZE} \neq 0$ causes an address error. The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses.

When bits 63..62 of the virtual address are "01," the virtual address space selected is the 2^{VSIZE} bytes of the current supervisor address space. $vAddr_{61..VSIZE} \neq 0$ causes an address error. The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses.

When bits 63..62 of the virtual address are "10," the virtual address space selected is a 2^{PSIZE} byte kernel physical space. $vAddr_{58..PSIZE} \neq 0$ causes an address error. References to this space are not mapped; the physical address selected is taken directly from bits PSIZE-1..0 of the virtual address, and the cache algorithm is specified by bits $vAddr_{61..59}$ (see EntryLo for the cache algorithm values).

When bits 63..62 of the virtual address are "11," the virtual address space selected is either a $2^{VSIZE}-2^{31}$-byte kernel virtual space (when $vAddr_{61..VSIZE}=0$), or a 2^{30} region compatible with the 32-bit address model (when $vAddr_{61..31}=-1$). Other addresses cause an address error. The virtual address is extended with the contents of the address space identifier field to form unique virtual addresses.

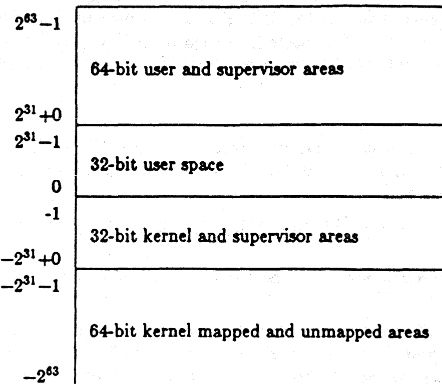
The compatibility space is divided into four 0.5 GB regions, just as in MIPS II mode. The region from 0xffff_ffff_8000_0000 to 0xffff_ffff_9fff_ffff (kseg0) is unmapped and cached, using the cache algorithm specified in the Config register; the physical address is $0^7 \parallel vAddr_{29..0}$. The region from 0xffff_ffff_a000_0000 to 0xffff_ffff_bfff_ffff (kseg1) is unmapped and uncached; the physical address is $0^7 \parallel vAddr_{29..0}$. The region from 0xffff_ffff_c000_0000 to 0xffff_ffff_dfff_ffff is the MIPS II supervisor virtual address space. The region from 0xffff_ffff_e000_0000 to 0xffff_ffff_ffff_ffff is the MIPS II kernel mapped space.

As a special feature for the ECC handler, if the ERL bit of the Status register is set, the user address region becomes a 2^{31}-byte unmapped, uncached space, similar to kseg1. This allows the ECC exception code to operate uncached using R0 as a base register.

The figure below shows the MIPS III kernel-mode virtual address space, where addresses are unsigned:

Address	Region
$2^{64}-1$	Kernel virtual address space (kseg3) Mapped, 0.5 GB
$2^{64}-2^{32}$+0xe0000000	
$2^{64}-2^{32}$+0xdfffffff	Supervisor MIPS II virtual address space (ksseg) Mapped, 0.5 GB
$2^{64}-2^{32}$+0xc0000000	
$2^{64}-2^{32}$+0xbfffffff	Uncached kernel physical address space (kseg1) Unmapped, 0.5 GB
$2^{64}-2^{32}$+0xa0000000	
$2^{64}-2^{32}$+0x9fffffff	Cached kernel physical address space (kseg0) Unmapped, 0.5 GB
$2^{64}-2^{32}$+0x80000000	
$2^{64}-2^{32}$+0x7fffffff	address error
$3\times2^{62}+2^{VSIZE}-2^{31}+0$	
$3\times2^{62}+2^{VSIZE}-2^{31}-1$	Kernel virtual address space (xkseg) Mapped
$3\times2^{62}+0$	
$3\times2^{62}-1$	Kernel physical address space (see below) Unmapped
$2\times2^{62}+0$	
$2\times2^{62}-1$	address error
$1\times2^{62}+2^{VSIZE}+0$	
$1\times2^{62}+2^{VSIZE}-1$	Supervisor MIPS III virtual address space (xksseg) Mapped
$1\times2^{62}+0$	
$1\times2^{62}-1$	address error
$2^{VSIZE}+0$	
$2^{VSIZE}-1$	User virtual address space (xkuseg) Mapped
0	

It may be helpful to think of addresses as signed values, in which case the 32-bit address space can be seen as the center subset of the 64-bit address space:



For example, for VSIZE=40, a virtual address is of the form:

63 62	61 ... 40	39 ... 0
R	FILL	vAddr
2	22	40

where:

vAddr	is the virtual address within the region
FILL	address error if not 0 or -1 (see above)
R	is the region
	0 user
	1 supervisor
	2 kernel unmapped
	3 kernel mapped and 32-bit compatible space

For kernel unmapped space with PSIZE = 36, the address has the form:

63 62	61 59	58 ... 36	35 ... 0
2	C	0	pAddr
2	3	23	36

where:

pAddr	is the virtual address within the region
C	is the cache algorithm (see EntryLo)
0	address error if not zero

4.3.8. Translation Lookaside Buffer

Mapped virtual addresses are translated into physical addresses using a translation lookaside buffer (TLB). There are two distinct implementations of the TLB, the first, as used on R2000, R3000, and R4000 processors is a fully-associative on-chip TLB. The second, as used on R6000 processors, is a two-set associative in-cache TLB, implemented using the R6000 two-way set associative secondary cache.

4.3.8.1. On-chip TLB

The TLB contains an implementation-dependent number of entries (TLBSIZE; R2000 and R3000 implementations have 64 entries; R4000 implementation has 48), each of which are simultaneously checked for a match with the extended virtual address.

For R2000 and R3000 processors each TLB entry maps one page. For R4000 processors, each TLB entry maps an even-odd page pair.

The page size is 4K bytes on R2000 and R3000 processors. The R4000 page size is variable in the range 4K to 16M bytes.

A virtual address matches a TLB entry when the virtual page number (VPN) field of the virtual address equals the VPN field of the entry, and either the Global (G) bit of the TLB entry is set, or the address space identifier (ASID) field of the virtual address (as held in the EntryHi register) matches the ASID field of the TLB entry. While the Valid (V) bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

The operation of the TLB is not defined if more than one entry in the TLB matches. If one TLB entry matches, the physical address and access control bits (N, D, and V) are retrieved; otherwise a TLB refill exception occurs. If the access control bits (D and V) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs.

The R4000 page size is controlled on a per-entry basis by a bit-mask. The page size can vary from 4K to 16M bytes.

The format of each TLB entry is as follows:

R2000, R3000:

63 … 44	43 … 38	37 … 32
VPN	ASID	-
20	6	6

31 … 12	11	10	9	8	7 … 0
PFN	N	D	V	G	-
20	1	1	1	1	8

MIPS II R4000:

127–121	120–109	108–96
-	MASK	-
7	12	13

95–77	76	75–72	71–64
VPN2	G	-	ASID
19	1	4	8

63–62	61–38	37–35	34	33	32
-	PFN	C	D	V	-
2	24	3	1	1	1

31–30	29–6	5–3	2	1	0
-	PFN	C	D	V	-
2	24	3	1	1	1

MIPS III R4000:

255–217	216–205	204–192
-	MASK	-
39	12	13

191–190	189–168	167–141	140	139–136	135–128
R	-	VPN2	G	-	ASID
2	22	27	1	4	8

127–94	93–70	69–67	66	65	64
-	PFN	C	D	V	-
34	24	3	1	1	1

63–30	29–6	5–3	2	1	0
-	PFN	C	D	V	-
34	24	3	1	1	1

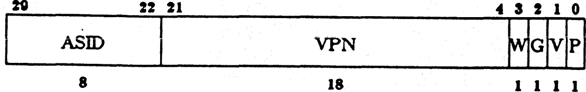
where:

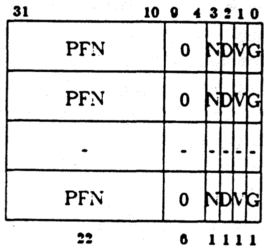
MASK	is the comparison mask
R	is the Region (00 → user, 01 → supervisor, 11 → kernel) used to match $vAddr_{63..62}$
VPN	is the Virtual Page Number (upper bits of virtual address)
VPN2	is the Virtual Page Number / 2
ASID	is the Address Space Identifier
PFN	is the Page Frame Number (upper bits of physical address)
N	if set, page is uncached
C	is the cache algorithm for the page (see EntryLo)
D	if set, page is dirty (writable)
V	if set, entry is valid
G	if set, page is global (ignore ASID in match logic)
-	these bits are not stored in the TLB

4.3.8.2. In-cache TLB

The TLB in R6000 processors is contained in a reserved portion of the secondary cache, and is updated by software. Unlike the fully-associative on-chip TLB, which is consulted on each access to memory, the in-cache TLB is consulted only when virtual-address cache tags do not match on a memory access. For each cache line in the in-cache TLB, the corresponding virtual-address cache tag holds the VPN and ASID fields to determine whether the set of TLB entries is valid.

The format of a group of LINESIZE entries is as follows (R6000):





where:

ASID	is the Address Space Identifier
VPN	is the Virtual Page Number (upper bits of virtual address)
W	is writable
G	is global
V	is valid
P	is parity over cache tag
PFN	is the Page Frame Number (upper bits of physical address)
N	if set, page is uncached
D	if set, page is dirty (writable)
V	if set, entry is valid
G	if set, page is global (ignore ASID in match logic)
0	is unused (ignored on write, zero when read)

4.3.9. Cache Memory

MIPS processors are normally configured with both an instruction cache and a data cache, and in some cases a secondary cache as well. A configuration may have variable cache sizes, within implementation-dependent minimum and maximum cache size limits:

implementation	minimum cache size	maximum cache size
R2000	4 kilobytes	64 kilobytes
R3000	4 kilobytes	256 kilobytes
R4000 (I-cache)	8 kilobytes	32 kilobytes
R4000 (D-cache)	8 kilobytes	32 kilobytes
R4000 (S-cache)	128 kilobytes	4 megabytes
R6000 (S-cache)	512 kilobytes	2 megabytes

When a processor has both an instruction cache and data cache, the two caches need not be the same size.

R4000 processors have I and D caches on-chip, and so are fixed in size for a given implementation, but implementations with multiple configurations are expected.

R6000 processors use a combined instruction-and-data second-level cache, which is two-set associative, and contains 512K or 2M bytes. The primary cache size is determined by logic outside the R6000 processor.

In the pseudocode descriptions of the cache below, the following implementation-dependent variables are used: CACHESIZE is the number of words in the cache; CACHEBITS is the base-two log of the number of bytes in the cache, as listed in the table below:

cache size (bytes)	CACHESIZE (words)	CACHEBITS
4 KB	1024	12
8 KB	2048	13
16 KB	4096	14
32 KB	8192	15
64 KB	16384	16
128 KB	32768	17
256 KB	65536	18
512 KB	131072	19
1 MB	262144	20
2 MB	524288	21
4 MB	1048576	22

R4000 and R6000 processors use a single tag for multiple data words. The set of data words and its tag is a "cache line." In the definitions for these processors, the following implementation-dependent variables are used: LINESIZE is the number of words in a cache line; LINEBITS is the base-two log of the number of bytes in a cache line, as in the table below:

line size (bytes)	LINESIZE (words)	LINEBITS
16	4	4
32	8	5
64	16	6
128	32	7
256	64	8

4.3.9.1. R2000 caches

For R2000 processors, both instruction and data cache lines consists of a single 32-bit word, a cache tag, and a valid bit. The data field is protected by 4 bits of parity, and the tag field is protected by 3 bits of parity. The low bits of the physical address select a single cache line ("direct mapped"). The caches are accessed for cached instruction and data fetches. A cache hit occurs when the cache tag and physical address match and the valid bit is set. On cache miss the processor reads one word from memory and refills the cache. Word stores to cached addresses write both the cache and memory ("write through") and cannot miss. Partial-word stores to cached addresses, such as generated by SB, SH, and sometimes SWL and SWR instructions, invalidate the addressed cache line unconditionally.

The format of the cache word is as follows:

59 56	55 53	52	51 32	31 0
DataP	TagP	V	PFN	Data
4	3	1	20	32

where:

Data	is the cache data
PFN	is the Page Frame Number (upper bits of physical address)
V	if set, entry is valid
TagP	is parity over the V and PFN fields
DataP	is parity over the Data field

4.3.9.2. R3000 caches

The caches of R3000 processors are similar to those of the R2000, with two additional features, both of which are configuration options selected when the processor is reset. If store partial mode is selected, partial-word stores are two cycle operations that first read the addressed cache line, and on a hit write updated data and send a full word write to memory. On a miss the cache is not modified and a partial-word write is sent to memory. When the cache is "isolated" (see description of the IsC bit of the Status register), partial-word stores continue to unconditionally invalidate. Second, if multi-word refill mode is selected, a cache miss will read from main memory, and write into the cache, CACHEREFILL words, where CACHEREFILL may be set to 4, 8, 16, or 32 words.

The format of the cache word is as follows:

59 56	55 53	52	51 32	31 0
DataP	TagP	V	PFN	Data
4	3	1	20	32

where:

Data	is the cache data
PFN	is the Page Frame Number (upper bits of physical address)
V	if set, entry is valid
TagP	is parity over the V and PFN fields
DataP	is parity over the Data field

4.3.9.3. R4000 primary instruction cache

The instruction cache is indexed with a virtual address and organized as blocks of 16 or 32 bytes of data with a 25-bit tag. The tag holds a 24-bit physical address, and a single valid bit. Byte parity is used on the instruction data, and a single parity bit is used for the tag. The format of a 32-byte cache line is as follows:

25	24	23 ... 0
P	V	PTag
1	1	24

71 ... 64	63 ... 0
DataP	Data
DataP	Data
DataP	Data
DataP	Data
8	64

where:

PTag	is the physical tag (bits 35..12 of the physical address)
V	is the valid bit
Data	is the cache data
P	is even parity for the PTag and V fields
DataP	is even parity

4.3.9.4. R4000 primary data cache

The data cache is indexed with a virtual address and organized as blocks of 16 or 32 bytes of data with a 27-bit tag. The tag holds a 24-bit physical address, a two-bit cache line state, and a writeback bit. Byte parity is used on the data, a single parity bit is used for the tag, and the writeback bit has its own parity bit. The format of a 32-byte cache line is as follows:

28	27	26	25 24	23 ... 0
W'	W	P	CS	PTag
1	1	1	2	24

71 ... 64	63 ... 0
DataP	Data
DataP	Data
DataP	Data
DataP	Data
8	64

where:

PTag	is the physical tag (bits 35..12 of the physical address)
CS	is the cache state
W	is the Writeback bit (set if this data is modified)
Data	is the cache data
P	is even parity for the PTag and CS fields
W'	is even parity for the Writeback bit
DataP	is even parity for the Data

cache states:

0	is Invalid
1	is Shared (either Clean or Dirty)
2	is Clean Exclusive
3	is Dirty Exclusive

In R4000 processors without a secondary cache, only two states are used to indicate whether the cache line is valid or not (Invalid and Dirty Exclusive). In R4000 processors with a secondary cache, all four states are used to control whether loads and stores need to access the secondary cache for coherency purposes. When the primary cache is filled from the secondary, the secondary state is mapped into a primary cache state by folding the secondary states Shared and Dirty Shared into the primary state Shared. The Dirty Exclusive state allows the primary cache to be written without a secondary access. In all R4000 processors, the Writeback bit, and not the cache state, is used to indicate the primary contains modified data that must be written back to memory or the secondary cache when it is replaced.

Primary state	Secondary states	Action on Load	Action on Store
Invalid	all	miss	miss
Shared	Shared Dirty Shared Dirty Exclusive	none	Read secondary cache tag. If Dirty Exclusive, set primary state to Dirty Exclusive; otherwise if coherency algorithm is update on write, then send update and set secondary cache state to Dirty Shared; otherwise send invalidate and set primary and secondary states to Dirty Exclusive.
Clean Exclusive	Clean Exclusive Dirty Exclusive	none	Set data and secondary cache states to Dirty Exclusive.
Dirty Exclusive	Dirty Exclusive	none	none

4.3.9.5. R4000 secondary cache

R4000 processors support an optional external secondary cache. The secondary cache is writeback, direct-mapped, indexed with a physical address, checked with a physical tag, and organized as blocks of 32, 64, or 128 bytes of data with a 25-bit tag. The tag holds a 19-bit physical address, a three-bit cache line state, and a three-bit primary cache index. The tag is protected by a seven-bit error correction code. The tag holds bits 35..17 of the physical address. Lower order bits are not stored as they are implicit in the index used to select the block (minimum 32 bytes) within the the cache (128 KByte minimum). The format of the tag is as follows:

31 .. 25	24 .. 22	21 .. 19	18 .. 0
ECC	CS	PIdx	STag
7	3	3	19

where:

ECC	ECC for secondary tag
CS	is the cache state
PIdx	is primary cache index (bits 14..12 of the virtual address)
STag	is the physical tag (bits 35..17 of the physical address)

cache states:

0	is Invalid
1	is reserved
2	is reserved
3	is reserved
4	is Clean Exclusive
5	is Dirty Exclusive
6	is Shared
7	is Dirty Shared

The cache state indicates whether the cache line data and tag is valid, as well as whether the data is at least potentially present in other processors' caches (Shared vs. Exclusive), and whether the processor is responsible for eventually updating main memory (Clean vs. Dirty).

When a secondary cache is used, the primary caches must be a subset of the secondary cache. R4000 processors maintain the subset property by checking and invalidating the primary caches if necessary when a secondary cache line is replaced.

The PIdx field allows the processor to locate the primary cache blocks that may contain data from this secondary cache block. A second function of the PIdx field is to detect a cache alias. During a data reference to the S-cache, if the physical address tag matches, but the PIdx field does not match the appropriate bits in the virtual address, then the reference is made with a different virtual address than the one that caused the S-cache line to be created. This could create a cache alias and the processor signals this condition by taking a Virtual Coherency Exception.

Secondary cache states Shared and Dirty Shared are mapped to the primary cache state Shared.

4.3.9.6. R6000 cache

The primary instruction and data caches are organized into words containing 32 bits (4 bytes) of data, with parity for each byte. A cache line consists of 8 words (32 bytes) of data for the instruction cache, and 2 words (8 bytes) of data for the data cache, and associated with each line is 30 additional bits, called the primary cache tag. The virtual address is used to select a single cache line ("direct mapped"). The tag holds an eight-bit address space identifier (ASID), an 18-bit virtual page number (VPN), and four one-bit values denoting writable, global, valid, and tag parity. The format of a 32-byte instruction cache line is as follows:

29 22	21 4	3	2	1	0
ASID	VPN	W	G	V	P
8	18	1	1	1	1

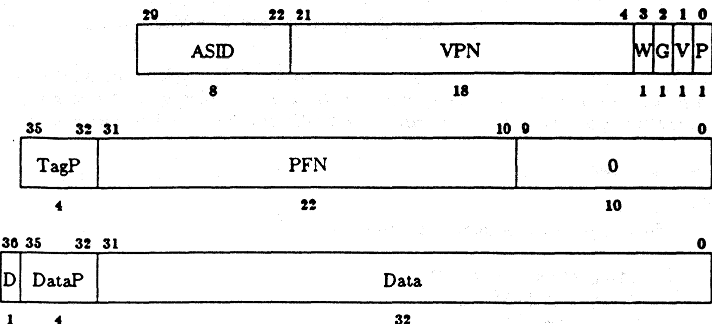
35 32	31 0
DataP	Data
DataP	Data
DataP	Data
DataP	Data
DataP	Data
DataP	Data
DataP	Data
DataP	Data
4	32

where:

ASID	is the Address Space Identifier
VPN	is the Virtual Page Number (upper bits of virtual address)
W	is writable
G	is global
V	is valid
P	is parity over cache tag
Data	is the cache data
DataP	is byte parity on the cache data

The first-level instruction and data caches hold virtual tags. Changes to the virtual-to-physical memory mapping, including the reassignment of ASIDs, may require that software invalidate the contents of the instruction and data caches.

The secondary cache holds both instructions and data, and is organized into words containing 32 bits (4 bytes) of data, with parity for each byte, and dirty bit per-word. A cache line consists of 32 words of data, and associated with each line are 30 additional bits, called the secondary cache tag. The format of a secondary cache line is as follows:



where:

ASID	is the Address Space Identifier
VPN	is the Virtual Page Number (upper bits of virtual address)
W	is writable
G	is global
V	is valid
P	is parity over cache tag
TagP	is byte parity on the physical tag
PFN	is the Page Frame Number (upper bits of physical address)
Data	is the cache data
DataP	is byte parity on the cache data
D	is a per-word dirty bit
0	is unused

The secondary cache consists of two associative sets, each of 256K bytes of data, for a total cache size of 512K bytes. The high 32K of the 512K bytes is reserved for the in-cache TLB and physical tags. Physical address that would otherwise access this portion of the cache are instead mapped to another 32K byte area of the cache. The secondary cache is indexed with a physical address and both virtual and physical tags are stored. A match on the virtual tag indicates a cache hit. A mismatch on the virtual tag indicates either a cache miss or an incorrect virtual to physical translation. The physical tags are then used to detect virtual address aliasing, and are checked after translating the virtual address to a physical address via the in-cache TLB. When the virtual-to-physical memory mapping is changed, including the reassignment of ASIDs, software must flush the contents of the secondary cache virtual tags. A subsequent reference to the same physical address will cause the virtual tag to be regenerated.

4.3.10. Functional Operation of Memory System

This section details the operation of the address translation and memory access functions used to describe instruction fetching and the load and store instructions in chapter 2. The functions: AddressTranslation, LoadMemory, and StoreMemory are listed below.

LoadMemory and StoreMemory in turn use the LoadMainMemory and StoreMainMemory functions to access main memory when required.

The descriptions here are at an *architectural* level, that is, features of the implementation that are managed transparently are not described in this section. However, these software-transparent features do affect the performance of the system, and the performance effects are briefly described in the following section.

4.3.10.1. R2000 processor

AddressTranslation:

```
(pAddr, uncached) ← AddressTranslation (vAddr, IorD):

   if vAddr_31 and not vAddr_30 then
      pAddr ← 0^3 || vAddr_28..0
      uncached ← vAddr_29
   else
      match ← i in 0..TLBSIZE-1 such that vAddr_31..12 = TLB[i]_63..44 and
         (CPR[0,EntryHi]_11..6 = TLB[i]_43..38 or TLB[i]_8)
      pAddr ← TLB[match]_31..12 || vAddr_11..0
      uncached ← TLB[match]_11
   endif
```

LoadMemory:

```
memoryWord ← LoadMemory (uncached, accessType, pAddr, vAddr, IorD):

   if uncached or cacheless then
      memoryWord ← LoadMainMemory (accessType, pAddr)
   else
      effCache ← CPR[0, SR]_17 xor IorD
      cacheEntry ← Cache[effCache, pAddr_CACHEBITS-1..2]
      if cacheEntry_51..32 = pAddr_31..12 and cacheEntry_52 then
         memoryWord ← cacheEntry_31..0
      else
         memoryWord ← LoadMainMemory (WORD, pAddr_31..2 || 0^2)
         cacheEntry ← 1^1 || pAddr_31..12 || memoryWord
         Cache[effCache, pAddr_CACHEBITS-1..2] ← cacheEntry
      endif
   endif
```

StoreMemory:

```
StoreMemory (uncached, accessType, memoryWord, pAddr, vAddr, IorD):

    if uncached or cacheless then
        StoreMainMemory (accessType, memoryWord, pAddr)
    else
        effCache ← CPR[0, SR]17 xor IorD
        validInCache ← accessType = WORD
        cacheEntry ← validInCache || pAddr31..12 || memoryWord
        Cache[effCache, pAddrCACHEBITS-1..2] ← cacheEntry
        StoreMainMemory (accessType, memoryWord, pAddr)
    endif
```

LoadMainMemory:

```
memoryWord ← LoadMainMemory (accessType, pAddr):

    byte ← pAddr1..0
    for i in 0..accessType
        if BigEndianMem = 1 then
            memoryWord31-8*(byte+i)..24-8*(byte+i) ← MainMemory[pAddr+i]
        else
            memoryWord7+8*(byte+i)..8*(byte+i) ← MainMemory[pAddr+i]
        endif
    endfor
```

StoreMainMemory:

```
StoreMainMemory (accessType, memoryWord, pAddr):

    byte ← pAddr1..0
    for i in 0..accessType
        if BigEndianMem = 1 then
            MainMemory[pAddr+i] ← memoryWord31-8*(byte+i)..24-8*(byte+i)
        else
            MainMemory[pAddr+i] ← memoryWord7+8*(byte+i)..8*(byte+i)
        endif
    endfor
```

4.3.10.2. R3000 processor

The functions AddressTranslation, LoadMainMemory, and StoreMainMemory are the same as the R2000 processor.

LoadMemory:

```
memoryWord ← LoadMemory (uncached, accessType, pAddr, vAddr, IorD):

   if uncached or cacheless then
      memoryWord ← LoadMainMemory (accessType, pAddr)
   else
      effCache ← CPR[0, SR]_17 xor IorD
      cacheEntry ← Cache[effCache, pAddr_CACHEBITS-1..2]
      if cacheEntry_51..32 = pAddr_31..12 and cacheEntry_52 then
         memoryWord ← cacheEntry_31..0
      else
         for i in 0..CACHEREFILL-1
            memoryWord ← LoadMainMemory (WORD, pAddr_31..LINEBITS || i || 0^2)
            cacheEntry ← 1^1 || pAddr_31..12 || memoryWord
            Cache[effCache, pAddr_CACHEBITS-1..LINEBITS || i] ← cacheEntry
         endfor
      endif
   endif
```

StoreMemory:

StoreMemory (uncached, accessType, memoryWord, pAddr, vAddr, IorD):

if uncached or cacheless then
  StoreMainMemory (accessType, memoryWord, pAddr)
else
  effCache ← $CPR[0, SR]_{17}$ xor IorD
  if partialWordCacheUpdate then
    cacheEntry ← Cache[effCache, $pAddr_{CACHEBITS-1..2}$]
    if $cacheEntry_{51..32}$ = $pAddr_{31..12}$ and $cacheEntry_{52}$ then
      byte ← $pAddr_{1..0}$
      if BigEndian then
        memoryWord ← $cacheEntry_{31..32-8*byte}$ ||
          $memoryWord_{31-8*byte..24-8*byte}$ ||
          $cacheEntry_{23-8*byte..0}$
      else
        memoryWord ← $cacheEntry_{31..8+8*byte}$ ||
          $memoryWord_{7+8*byte..8*byte}$ ||
          $cacheEntry_{8*byte-1..0}$
      endif
      accessType = WORD
      pAddr = $pAddr_{31..2}$ || 0^2
      cacheEntry ← $cacheEntry_{52..32}$ || memoryWord
      Cache[effCache, $pAddr_{CACHEBITS-1..2}$] ← cacheEntry
    endif
    StoreMainMemory (accessType, memoryWord, pAddr)
  else
    validInCache ← accessType = WORD
    cacheEntry ← validInCache || $pAddr_{31..12}$ || memoryWord
    Cache[effCache, $pAddr_{CACHEBITS-1..2}$] ← cacheEntry
    StoreMainMemory (accessType, memoryWord, pAddr)
  endif
endif

4.3.10.3. R4000 processor

This section to be completed.

4.3.10.4. R6000 processor

This section to be completed.

4.3.11. Transparent implementation features of the cache and virtual memory system

Functionally, a user of the R-Series processors can assume that all instructions and features operate as described. However, certain features of the implementation not otherwise detailed can have an effect on the performance of a system or application, even though the management of these features is entirely transparent to the functional description.

4.3.11.1. R2000

R2000 processors contain a small, two-entry micro-TLB, whose function is to translate instruction addresses from a virtual address to a physical address more quickly (less latency) than can be accomplished in the main 64-entry TLB. This micro-TLB holds two entries, fully-associative, with the least-recently-used entry replaced

on a miss in a single cycle. The micro-TLB is flushed when the EntryHi register is loaded.

4.3.11.2. R3000

R3000 processors contain a micro-TLB as well, with the same organization as the R2000. In addition, partial-word writes, when selected to operate in a read-modify-write manner, will require two cycles, rather than one.

4.3.11.3. R4000

R4000 processors also contain a micro-TLB, with the same organization as the R2000. A micro-TLB miss has a three cycle penalty. However, no penalty occurs for misses on the target of a taken branch or jump. Each micro-TLB maps 4KB, regardless of the page size use in the full TLB.

R4000 processors use the virtual address to index the cache in parallel with the TLB translation. At the end of the TLB translation, the cache tag is compared with the physical page number, and if they differ, a cache miss occurs in which the coherence state of the line is determined.

Because no virtual addresses are stored in R4000 caches, no cache flushing is required when the virtual to physical mapping is changed (e.g. on context switch).

The use of a virtual address to index the primary data cache does cause some problems that software must avoid. Two references to the same physical location that use different virtual addresses modulo the cache size will not operate coherently, because they will read and write different cache locations. For coherency, the software should align shared memory on a boundary greater than or equal to the largest primary data cache size. Then different virtual addresses will still reference the same cache location. When sharing cannot be restricted, cache invalidation can be used to effect coherency.

4.3.11.4. R6000

Both the primary and secondary caches of the R6000 contain virtual tags with an 8-bit address space identifier. When a virtual address is deleted, or re-assigned to a new physical address (for example, on address space identifier rollover), the virtual tags must be purged from the cache. For the primary caches, this is accomplished by flushing. For the secondary cache, the virtual tags are invalidated, but the physical tags and data remain valid.

Virtual addresses are used to index the primary caches. When the primary cache size is 16 kilobytes, this is not different from a physically-indexed cache, because the page size is also 16 kilobytes. To increase the data cache size to 64 kilobytes, software must use some method to solve the virtual address coherency problem. Virtual address coherency is not a problem for the instruction cache.

The secondary cache is accessed on its first attempt by a physical index that is predicted from the virtual index by a direct-mapped TLB-slice. This TLB-slice contains only enough bits of the physical page frame number to guess the physical index, not enough to check that the guess is correct. If the virtual tag in the secondary cache matches, it can be presumed that the guess is correct, otherwise an additional eight machines cycles are required to correct the guess, access the secondary cache a second time, and resume execution. The size of the TLB-slices are 8 entries for instruction accesses and 8 entries for data accesses.

4.4. System Control Coprocessor Registers

Registers within the system control coprocessor provide the path through which the virtual memory system's page mapping is examined and changed, and through which the operating modes (kernel vs. user mode, interrupts enabled or disabled, cache enabled or isolated, caches normal or switched) may be controlled and by which exceptions may be identified and handled. In this section, each of these registers are described in depth.

The EntryHi and EntryLo registers provide a data pathway through which the TLB is read, written, or probed. For R4000 processors, EntryLo is actually two registers, EntryLo0 and EntryLo1. R4000 processors also have a page mask register that holds the per-entry comparison mask to control the page size. These registers are loaded with relevant information when translation exceptions occur. These registers are only present for implementations with an on-chip TLB.

The Index and Random registers are used to access particular entries in the TLB for reading or writing; the Index register also identifies the matching entry when the TLB is associatively probed. These registers are only present for implementations with an on-chip TLB.

The Bad Virtual Address register and the Context register are both loaded with the virtual address for which a translation exception occurs. The Context register is specifically used within a fast, minimal TLB exception handler (called the TLB refill handler) for rapid access of Page Table Entries (PTE) from main memory. The Context register is only present for implementations with an on-chip TLB. The XContext register is similar to the Context register, but is intended for use with MIPS III addressing.

The Status and Cause registers provide means to alter operating modes and to identify the cause of exceptions.

The Exception Program Counter (EPC) register indicates the virtual address at which the most recent exception occurred.

The Error Exception Program Counter (ErrorEPC) register indicates the virtual address at which the most recent ECC, Reset, or NMI exception occurred. This register is present only on R4000 processors.

The Processor Revision Identification (PRId) register identifies the implementation and revision level of the processor and associated system control coprocessor.

The Timer Count and Compare registers implement a real-time cycle-count facility. These registers are present only on R4000 processors.

The WatchLo and WatchHi registers address reference traps for debugging. These registers are present only on R4000 processors.

The TagLo, TagHi, and CacheErr registers are used in the detection and correction of cache parity and ECC errors, and in cache diagnostic testing. The TagLo and TagHi registers can also be used for software virtual coherency exception handling. These registers are present only on R4000 processors.

On MIPS III processors, coprocessor 0 registers may be either 32 or 64 bits wide. 32-bit registers must be read and written with MFC0 and MTC0. 64-bit registers may be read and written with MFC0, MTC0, DMFC0, and DMTC0. On the R4000, the 64-bit registers are EntryHi, XContext, EPC, and ErrorEPC. In future versions of the R4000, EntryLo0 and EntryLo1 may become 64-bit registers.

The system control coprocessor registers are numbered as follows:

Number	Mnemonic	Description
0	Index	Programmable pointer into TLB array (on-chip TLB only)
1	Random	Pseudo-random pointer into TLB array (read only) (on-chip TLB only)
2	EntryLo	Low half of TLB entry (R2000 and R3000 only)
2	EntryLo0	Low half of TLB entry for even VPN (R4000 only)
3	EntryLo1	Low half of TLB entry for odd VPN (R4000 only)
4	Context	Pointer to kernel virtual PTE table (on-chip TLB only)
5	PageMask	TLB Page Mask (R4000 only)
6	Wired	Number of wired TLB entires (R4000 only)
7	Error	Parity control/status register (R6000 only)
8	BadVAddr	Bad virtual address
9	Count	Timer Count (R4000 only)
10	EntryHi	High half of TLB entry (on-chip TLB only)
10	ASID	Address Space identifier (in-cache TLB only)
11	Compare	Timer Compare (R4000 only)
12	SR	Status Register
13	Cause	Cause of last exception
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16	Config	Configuration Register (R4000 only)
17	LLAddr	Load Linked address (R4000 only)
18	WatchLo	Memory reference trap address low bits (R4000 only)
19	WatchHi	Memory reference trap address high bits (R4000 only)
20	XContext	Context Register for MIPS III addressing (R4000 only)
21-25	-	unused
26	ECC	S-cache ECC and Primary Parity (R4000 only)
27	CacheErr	Cache Error and Status register (R4000 only)
28	TagLo	Cache Tag register (R4000 only)
29	TagHi	Cache Tag register (reserved)
30	ErrorEPC	Error Exception Program Counter (R4000 only)
31	-	unused

4.4.1. EntryHi

The EntryHi register is a read/write register that is used to access an on-chip TLB. In addition, the EntryHi register contains the Address Space Identifier (ASID) used to match the virtual address with a TLB entry when virtual addresses are presented for translation.

The EntryHi register also holds the contents of the high-order bits of an TLB entry when performing TLB Read and Write operations. When a TLB refill, TLB invalid, or TLB modified exception occurs, the EntryHi register is loaded with the Virtual Page Number and the Address Space Identifier of the virtual address that failed to have a matching TLB entry.

EntryHi is also used by the TLBP, TLBW, and TLBWI instructions. It is also written by the TLBR instruction.

The format of the register is as follows.

R2000, R3000:

31 — 12	11 — 6	5 — 0
VPN	ASID	0
20	6	6

MIPS II R4000:

31 — 13	12 — 8	7 — 0
VPN2	0	ASID
19	5	8

MIPS III R4000:

63 62	61 — 40	39 — 13	12 — 8	7 — 0
R	FILL	VPN2	0	ASID
2	22	27	5	8

where:

R	is the Region (00 → user, 01 → supervisor, 11 → kernel) used to match $vAddr_{63..62}$
VPN	is the Virtual Page Number (upper bits of virtual address)
VPN2	is the Virtual Page Number / 2
ASID	is the Address Space Identifier
0	reserved for future use: 0 on read; should be 0 on write
FILL	reserved for future use: 0 on read; should be 0 or -1 on write

4.4.2. PageMask

The PageMask register is a read/write register that is used when reading or writing an on-chip TLB. TLB Write and Read operations use this register as a source or destination. When virtual addresses are presented for translation, the corresponding bits in the TLB specify whether virtual address bits 24..13 participate in the comparison. This implements a variable page size on a per-entry basis.

R4000:

31 ... 25	24 ... 13	12 ... 0
0	MASK	0
7	12	13

where:

MASK is the page comparison mask
0 reserved for future use: 0 on read; should be 0 on write

The following table gives MASK values for the full range of page sizes. The operation of the TLB when MASK is not one of these values is undefined.

Page size	Bit											
	24	23	22	21	20	19	18	17	16	15	14	13
4K	0	0	0	0	0	0	0	0	0	0	0	0
16K	0	0	0	0	0	0	0	0	0	0	1	1
64K	0	0	0	0	0	0	0	0	1	1	1	1
256K	0	0	0	0	0	0	1	1	1	1	1	1
1M	0	0	0	0	1	1	1	1	1	1	1	1
4M	0	0	1	1	1	1	1	1	1	1	1	1
16M	1	1	1	1	1	1	1	1	1	1	1	1

4.4.3. EntryLo

On R2000 and R3000 processors, the EntryLo register is a 32-bit read/write register that is used to access an on-chip TLB. EntryLo holds the contents of the low-order 32 bits of an TLB entry when performing TLB Read and Write operations.

On R4000 processors, EntryLo consists of two registers, EntryLo0 for even virtual pages and EntryLo1 for odd virtual pages. Each register is 64 bits for MIPS III, 32 bits otherwise.

The format of the register is as follows:

EntryLo, R2000 and R3000:

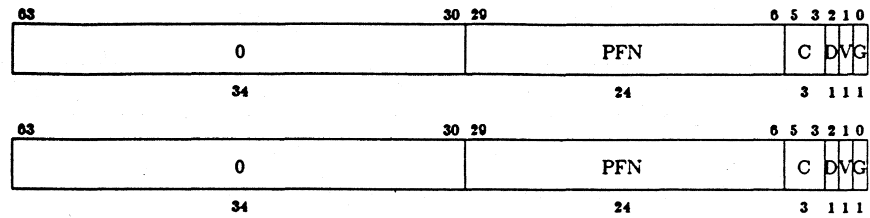
31 ... 12	11	10	9	8	7 ... 0
PFN	N	D	V	G	unused
20	1	1	1	1	8

EntryLo0 and EntryLo1, MIPS II R4000:

31 30	29 ... 6	5 ... 3	2	1	0
0	PFN	C	D	V	G
2	24	3	1	1	1

31 30	29 ... 6	5 ... 3	2	1	0
0	PFN	C	D	V	G
2	24	3	1	1	1

EntryLo0 and EntryLo1, MIPS III R4000:



where:

PFN	is the Page Frame Number (upper bits of physical address)
N	if set, page is uncached
C	specifies the cache algorithm to use
D	if set, page is dirty and writable
V	if set, entry is valid
G	if set in both Lo0 and Lo1, then ignore ASID
0	reserved for future use: 0 on read; should be 0 on write
unused	0 on read, ignored on write

The cache algorithm specifies whether references to this page should be cached, and if cached, selects between several cache coherency algorithms.

Cache algorithms:

0 reserved
1 reserved
2 uncached
3 cacheable non-coherent
4 cacheable coherent exclusive
5 cacheable coherent exclusive on write
6 cacheable coherent update on write
7 reserved for cacheable write-through

Further information on cache coherency algorithms can be found in the R4000 interface specification.

4.4.4. TLB Index

The TLB Index register is a read/write register of which 6 bits specify an entry in an on-chip TLB. The high-order bit of the register indicates the success or failure of an TLBP operation.

The TLB Index register is used to specify the entry in the TLB affected by the TLBRI and TLBWI instructions.

The format of the register is as follows:

R2000, R3000:

31	30 ... 14	13 ... 8	7 ... 0
P	0	Ind	0
1	17	6	8

R4000:

31	30 ... 6	5 ... 0
P	0	Ind
1	25	6

where:

P	is set if the last Probe operation is unsuccessful
Ind	TLB Index
0	must be zero

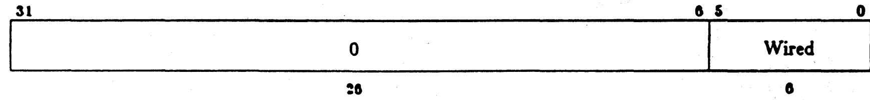
4.4.5. TLB Wired

The TLB Wired register is a read/write register that specifies the boundary between the wired and random entries of the TLB. This register exists on R4000 processors only. For R2000 and R3000 processors this boundary is fixed at 8.

This register is set to 0 upon system reset. Writing this register also sets Random to TLBENTRIES-1.

The format of the register is as follows:

R4000:



where:

Wired	TLB Wired boundary
0	must be zero

4.4.6. TLB Random

The TLB Random register is a read-only register of which 6 bits indexes an entry in an on-chip TLB.

On R2000 and R3000 processors, the value of this register changes (decrements) on each clock cycle of the machine, whether or not the processor executes an instruction on the clock cycle. On R4000 processors, this register decrements for each instruction executed. The values range between a lower bound set by the number of TLB entries reserved for exclusive use by the operating system (8 on R2000 and R3000, the contents of the TLB Wired register on R4000 processors) and an upper bound set by the total number of TLB entries (TLBENTRIES).

For R2000 and R3000 processors the upper bound (TLBENTRIES-1) is 63. For R4000 processors the upper bound (TLBENTRIES-1) is 39.

The TLB Random register is used to specify the entry in the TLB affected by the TLBWR instruction. It does not need to be read for this purpose, however, the register is readable in order to verify proper operation of the processor. To simplify testing, this register is set to the upper bound upon system reset. On the R4000, this register is also set to the upper bound when the Wired register is written.

The format of the register is as follows:

R2000, R3000:

31 — 14	13 — 8	7 — 0
0	Ran	0
18	6	8

R4000:

31 — 6	5 — 0
0	Ran
26	6

where:

Ran	TLB Random Index
0	must be zero

4.4.7. Bad Virtual Address

The Bad Virtual Address register is a read-only register that displays the most recently translated virtual address that failed to have a valid translation.

The format of the register is as follows:

MIPS I/II:

31 … 0
BadVAddr
32

MIPS III:

63 … 0
BadVAddr
64

where:

BadVAddr is the bad virtual address

4.4.8. Context

The Context register is a read/write register containing a pointer into a kernel virtual Page Table Entry (PTE) array. It is designed for use in the TLB refill handler.

The BadVPN field is not writable. It contains the VPN of the most recently translated virtual address that did not have a valid translation.

The PTEBase field is writable as well as readable, and indicates the base address of the PTE table of the current user address space.

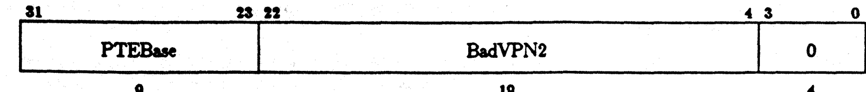
For R2000 and R3000 processors, the 19-bit BadVPN field contains bits 30..12 (user-segment virtual page number) of the BadVAddr register. Bit 31 is excluded because the TLB refill handler is only invoked on user-segment references. This format can be used directly as an address in a table of 4-byte PTEs. For other PTE and page sizes, shifting and masking this value produces an appropriate address.

For R4000 processors, the 19-bit BadVPN2 field contains bits 31..13 of the virtual address that caused the TLB miss. Bit 12 is excluded because a single TLB entry maps an even-odd page pair. This format can be used directly as an address in a table of pairs of 8-byte PTEs for a page size of 4K bytes. For other PTE and page sizes, shifting and masking this value produces an appropriate address. This register is intended for TLB refill for MIPS I/II (32-bit) address spaces. The XContext register should be used for TLB refill for MIPS III addressing.

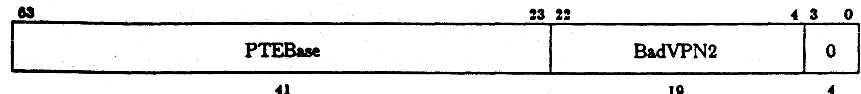
The format of the register is as follows for R2000 and R3000 processors:

31 … 21	20 … 2	1 … 0
PTEBase	BadVPN	0
11	19	2

MIPS II R4000:



MIPS III R4000:



where:

PTEBase	is the base address of the PTE
BadVPN	VPN of the failed virtual address
BadVPN2	VPN of the failed virtual address / 2
0	is unused (ignored on write, zero when read)

4.4.9. XContext

The XContext register is a read/write register containing a pointer into a kernel virtual Page Table Entry (PTE) array. It is designed for use in the XTLB refill handler, which loads TLB entries for references to a MIPS III address space.

The R and BadVPN2 fields are not writable. BadVPN2 contains bits 39..13 of the most recently translated virtual address that did not have a valid translation. R contains bits 63..62 of the address.

The PTEBase field is writable as well as readable, and indicates the base address of the PTE table of the current user address space.

This format can be used directly as an address in a table of 8-byte PTEs for a page size of 4K bytes. For other PTE and page sizes, shifting and masking this value produces an appropriate address.

The format of the register is as follows for R4000 processors (VSIZE = 40):

63 ... 33	32 31	30 ... 4	3 ... 0
PTEBase	R	BadVPN2	0
31	2	27	4

where:

PTEBase	is the base address of the page table
R	is the Region (00 → user, 01 → supervisor, 11 → kernel)
BadVPN2	VPN of the failed virtual address ($vAddr_{39..13}$)
0	is unused (ignored on write, zero when read)

4.4.10. Status

The status register (SR) is a read/write register that contains the kernel/user mode, interrupt enable, and diagnostic state of the processor.

For R2000, R3000, and R6000 processors, the contents of this register are undefined at reset, except for TS, SWc, KUc, and IEc, which are zero, and BEV, which is one. For R4000 processors, the contents of this register are undefined at reset, except for TS, which is zero, and ERL and BEV, which are one, and SR which distinquishes between Reset and NMI or Soft Reset.

For R2000, R3000, and R6000 processors, The SR contains a three-level stack (current, previous, and old) of the kernel/user mode bit (KU) and the interrupt enable (IE) bit. The stack is pushed when each exception is taken, and popped by the Restore From Exception (RFE) instruction. These bits may also be directly read or written.

For R4000 processors, the three-level stack is replaced by a base mode and base interrupt enable and two modifier bits: EXL and ERL. This change supports the new supervisor mode as well as fast TLB refill exceptions for the kernel address space.

The interrupt mask field (IM) is an 8-bit field that controls the enabling of each of eight interrupt conditions. An interrupt is taken if interrupts are enabled, and corresponding bits are set in both the interrupt mask field of the SR and the interrupt pending field of the Cause register. The actual width of this register is machine-dependent; see the description of the interrupt pending (IP) field of the Cause register.

The Coprocessor Usability (CU) field is a 4-bit field that individually controls the usability of the four coprocessors. Regardless of the setting of the CU_0 bit, coprocessor zero is always considered usable when in kernel mode.

For R6000 processors, only one of coprocessors 1, 2, or 3 may be present at any one time. Hence, only one of the CU_1, CU_2, or CU_3 bits should be on at any one time. There is only one CpCond input pin (coprocessor condition) and only one CpBusy input. The coprocessor instructions can be executed only if the corresponding CU bit is on.

On some processors, the RE bit (bit 25) is used to reverse the Endianess of the machine in User mode. R-Series processors are configured as either Little Endian or Big Endian at system reset. This selection is used in Kernel, Supervisor, and User mode when RE is clear. Setting this bit to one inverts the selection in User mode.

The Diagnostic Status (DS) field is an implementation-dependent 9-bit field that is used for self-testing and checking of the cache and virtual memory system.

The format of the register is as follows:

R2000, R3000, R6000:

31–28	27–26	25	24–16
CU	0	RE	DS
4	2	1	9

15–8	7–6	5	4	3	2	1	0
IM	0	KUo	IEo	KUp	IEp	KUc	IEc
8	2	1	1	1	1	1	1

where:

CU — controls the usability of each of the four coprocessor unit numbers (1 → usable; 0 → unusable). Coprocessor zero is always usable when in kernel mode, regardless of the setting of the CU_0 bit.

RE — Reverse endian in user mode (R6000 only).

DS — is an implementation-dependent diagnostic status field.

IM — Interrupt Mask: controls the enabling of each of the external, internal, coprocessor and software interrupts (0 → disabled; 1 → enabled). Bits 15..13 are unused on R6000 processors. See description of Cause register for further information.

KUo — is the old kernel/user mode (0 → kernel; 1 → user)

IEo — is the old interrupt enable (0 → disable; 1 → enable)

KUp — is the previous kernel/user mode (0 → kernel; 1 → user)

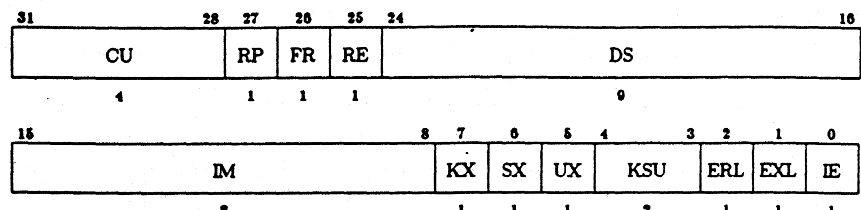
IEp — is the previous interrupt enable (0 → disable; 1 → enable)

KUc — is the current kernel/user mode (0 → kernel; 1 → user)

IEc — is the current interrupt enable (0 → disable; 1 → enable)

0 — reserved for future use: 0 on read; should be 0 on write

R4000:



where:

CU	controls the usability of each of the four coprocessor unit numbers (1 → usable; 0 → unusable). Coprocessor zero is always usable when in kernel mode, regardless of the setting of the CU_0 bit.
RP	enables reduced-power operation by reducing the clock frequency (0 → full speed; 1 → reduced clock). The clock divisor is programmable at boot time; see the R4000 Interface Specification.
KX	If set use the Extended addressing TLB refill exception vector address for TLB misses on kernel addresses.
SX	If set enables MIPS III opcodes in supervisor-mode and causes TLB misses on supervisor addresses to use the Extended TLB refill exception vector.
UX	If set enables MIPS III opcodes in user-mode and causes TLB misses on user addresses to use the Extended TLB refill exception vector. If clear implements MIPS II compatibility on virtual address translation; see MIPS III User-mode Virtual Addressing.
FR	enables additional floating-point registers (0 → 16 registers, 1 → 32 registers).
RE	Reverse endian in user mode.
DS	is an implementation-dependent diagnostic status field.
IM	Interrupt Mask: controls the enabling of each of the external, internal, coprocessor and software interrupts (0 → disabled; 1 → enabled). See description of Cause register for further information.
KSU	is the Mode (10 → user, 01 → supervisor, 00 → kernel)
ERL	is the Error Level (0 → normal, 1 → error)
EXL	is the Exception Level (0 → normal, 1 → exception)
IE	is the Interrupt Enable (0 → disable; 1 → enable)
0	reserved for future use: 0 on read; should be 0 on write

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

Accesses to the kernel address space are allowed when KSU = 00 or EXL = 1 or ERL = 1. Accesses to the supervisor address space are allowed when KSU ≠ 10 or EXL = 1 or ERL = 1. Access to the user address space is always allowed.

Interrupts are enabled when IE = 1 and EXL = 0 and ERL = 0.

When ERL = 1, the user address space is replaced by an unmapped, uncached space, so that R0-based addressing may be used in the ECC handler. Also, the split secondary cache mode is disabled, and further exceptions cause a Soft Reset. See the R4000 interface specification for details.

4.4.10.1. Diagnostic Status

Because the diagnostic facilities are heavily dependent on the characteristics of the cache and virtual memory system on the implementation, the layout of the diagnostic status field is implementation-dependent. Its normal use is for diagnostic code, and in certain cases for use by operating system diagnostic facilities (such as reporting parity errors), and on some machines, for relatively rare operations such as flushing caches. In normal operation, this field can and should be set to zero by operating system code.

4.4.10.1.1. R2000, R3000

For R2000 and R3000 processors, the diagnostic status bits: BEV, TS, PE, CM, PZ, SwC, and IsC, provide a complete fault detection capability, but are not intended to provide for extensive fault diagnosis.

The SwC bit controls the switching of control signals for instruction and data caches associated with the processor. When the bit is changed, cache control signals are altered so that the instruction and data caches are effectively interchanged. The processor must be executing from an uncached region and must not be executing loads or stores near the time of cache switching.

The IsC bit, when set, causes the cache currently being used as the data cache to be effectively isolated from the rest of the memory system, making cache diagnostics possible. (The selection of the "data" cache is dependent on the state of the SwC bit.) When the IsC bit is set, store operations affect only the cache (main memory writes are inhibited), and load operations return the data at the addressed location in the cache whether or not a cache miss occurs (main memory reads are inhibited). Uncached data references are not generally useful with IsC set, but their handling is as follows: uncached store operations affect neither the cache nor the main memory system, and uncached load operations return the data at the addressed location in the cache. IsC affects only data references; instruction fetches are not affected. This bit may also be used by operating system code to flush caches without causing associated main memory accesses.

The PZ bit, when set, causes zero to replace the normal outgoing parity bits which are generated on a store instruction, covering both cache data and tags. This permits the writing of incorrect parity bits in the caches, checking each parity tree individually within the cache diagnostics.

The CM bit, when the cache is isolated, indicates whether or not the most recent data cache load resulted in a cache miss. This bit is used by cache test programs to verify the proper functioning of the cache tag and parity bits.

The PE bit indicates whether a cache parity error has occurred. It is set on a cache parity error and reset by writing a one to this bit. Writing a zero to this bit does not affect its value. This bit is used to log cache parity errors in software, as otherwise, they are recovered from completely transparently. Within cache diagnostics, this bit is used to verify proper functioning of the cache parity bits and of the cache parity trees.

The TS bit is read-only and indicates that the TLB has shut down due to attempts to cause several entries in the TLB to be accessed simultaneously. This mechanism protects the TLB from hardware failures in the event of catastrophic software misuse of the TLB. When the TLB is in this state, all translations and PROBE accesses are inhibited, and have undefined effects. This state can be cleared only by asserting RESET.

The Bootstrap Exception Vectors (BEV) bit, when set, causes the TLB refill exception vector to be relocated to a virtual address of 0xbfc00100 and the general exception vector to 0xbfc00180. When cleared, these vectors are normally located at 0x80000000 (TLB refill) and 0x80000080 (general), respectively. This bit is used when diagnostic tests cause exceptions to occur prior to verifying proper operation of the cache and main memory system.

The format of the diagnostic status field is as follows (R2000, R3000 only):

24 23	22	21	20	19	18	17	16
0	BEV	TS	PE	CM	PZ	SwC	IsC
2	1	1	1	1	1	1	1

where:

- BEV controls the location of TLB refill and general exception vectors. (0 → normal; 1 → bootstrap).
- TS indicates that TLB shut-down has occurred.
- PE indicates that a cache Parity Error has occurred. This bit may be cleared by writing a one to this bit position.
- CM indicates whether a data cache miss has occurred while in cache test mode. (0 → hit; 1 → miss)
- PZ controls the zeroing of cache parity bits (0 → normal; 1 → parity forced to zero)
- SwC controls the switching of the data and instruction caches (0 → normal; 1 → switched)
- IsC controls isolation of cache (0 → normal; 1 → cache isolated)
- 0 is unused (ignored on write, zero when read)

4.4.10.1.2. R4000

The format of the diagnostic status field is as follows. All bits are read/write, except TS.

24 23	22	21	20	19	18	17	16
0	BEV	TS	SR	0	CH	CE	DE
2	1	1	1	1	1	1	1

where:

- BEV controls the location of TLB refill and general exception vectors. (0 → normal; 1 → bootstrap).
- TS indicates that TLB shutdown has occurred (read-only).
- SR indicates that a soft reset has occurred.
- CH "Hit" (tag match and valid state) or "miss" indication for last CACHE Hit Invalidate, Hit Writeback Invalidate, Hit Writeback, Hit Set Virtual, or Create Dirty Exclusive for a secondary cache.
- CE if set, the contents of the ECC register are used to set or modify the check bits of the caches; see the ECC register description.
- DE specifies that cache parity or ECC errors are not to cause exceptions.
- 0 reserved for future use

4.4.10.1.3. R6000

The format of the diagnostic status field is as follows (R6000 only):

24–23	22	21	20	19	18	17	16
0	BEV	0	CM1	CM0	PZ	ITP	MM
2	1	1	1	1	1	1	1

where:

- BEV controls the location of TLB refill and general exception vectors. (0 → normal; 1 → bootstrap).
- CM1 indicates that a miss on set 1 of the secondary cache occurred on the last load or store operation.
- CM0 indicates that a cache miss on set 0 of the secondary cache occurred on the last load or store operation.
- PZ controls the zeroing of cache parity bits (0 → normal; 1 → parity forced to zero)
- ITP inverts tag parity on writes (0 → normal; 1 → inverted parity)
- MM converts LWL/SWL/LWR/SWR to memory management instructions (0 → normal; 1 → memory management)
- 0 is unused (ignored on write, zero when read)

4.4.11. Cause

The Cause Register is a read/write register that describes the nature of the last exception. A 5-bit exception code indicates the cause of the exception and the remaining fields contain detail information relevant to the handling of certain types of exceptions.

For R6000 processors, see also the Error register.

The Branch Delay (BD) bit indicates whether the EPC has been adjusted to point at the branch instruction which precedes the next restartable instruction.

The Coprocessor Error (CE) field indicates, if the exception is "Coprocessor Unusable," the coprocessor unit number referenced by the instruction which caused the exception.

The Interrupt Pending (IP) field indicates which external, internal, coprocessor and software interrupts are pending. This field reflects the current status, and changes in response to external signals. The number and assignment of the IP (and IM) bits are implementation dependent.

R2000 and R3000 processors implement 6 external interrupts with $IP_{7..2}$, which are latched each cycle from input signals. $IP_{7..2}$ are read-only. IP_5 is typically used for the R2010/R3010 Floating-point coprocessor interrupt, IP_2 is typically used for system bus (I/O) interrupts. $IP_{1..0}$ are software interrupts, and may be written into to set or reset software interrupts.

R4000 processors implement 5 external interrupts with $IP_{6..2}$, but with several differences. Reading Cause returns the inclusive or of two internal registers for $IP_{6..2}$. One register is latched each cycle from input signals, as for R2000 and R3000 processors. The other register is is read and written by commands on the R4000 system interface port. On Reset, R4000 processors are configured either with IP_7 as a sixth external interrupt or as an internal interrupt that is set set when the Count register is equal to the Compare register. $IP_{1..0}$ are software-only interrupts, and may be written into to set or reset software interrupts. Floating-point exceptions use a separate exception code.

R6000 processors have three external interrupts, $IP_{4..2}$; IP_2 is used for system bus and interval timer interrupts, IP_3 is used for the floating-point coprocessor interrupt, and IP_4 is an unused spare. $IP_{1..0}$ are for software interrupts.

The format of this register has been designed so that the low-order 8 bits may be easily extracted and used as a word offset into a table for software interrupt vectoring.

Only $IP_{1..0}$ are read/write; all other bits are read only.

The format of the Cause register is as follows:

31	30	29 28	27 16
BD	0	CE	0
1	1	2	12

15 8	7	6 2	1 0
IP	0	ExcCode	0
8	1	5	2

where:

BD	indicates whether the last exception was taken while executing in a branch delay slot. (0 → normal; 1 → delay slot)
CE	indicates the coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IP	indicates whether an interrupt is pending
ExcCode	is the exception code field (described below)
0	is unused (ignored on write, zero when read)

The exception code field is coded as follows:

Number	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception (MIPS II processors only)
14	NCD	LDCz/SDCz to uncached address (R6000 only)
14	VCEI	Virtual Coherency Exception Instruction (R4000 only)
15	MC	Machine Check exception (R6000 only)
15	FPE	Floating Point exception (R4000 only)
16-22	-	Reserved for future use
23	WATCH	Reference to WatchHi/WatchLo address (R4000 only)
19-30	-	Reserved for future use
31	VCED	Virtual Coherency Exception Data (R4000 only)

4.4.12. Error

Error is a R6000-only control/status register for parity.

The format of the Error register is as follows:

31–20	19	18	17	16
0	IEXT	IRF	IDB	IIB
12	1	1	1	1

15–4	3	2	1	0
0	EXT	RF	DB	IB
12	1	1	1	1

where:

- **IEXT** 1 → causes the CPU to ignore the parity errors that are detected off chip and reported to the CPU by the External Parity Error signal. These include Tag parity errors and external coprocessor parity errors. Error register bits will be set to reflect the error but no exception will be generated.
- **IRF** 1 → causes the CPU to ignore parity errors that are detected from the Register File. These errors will not be propagated outside the chip because the parity is regenerated before it is sent out over the DataBus. Error register bits will be set to reflect the error but no exception will be generated.
- **IDB** 1 → causes the CPU to ignore parity errors that are detected off of the DataBus. This covers Load's and MFCz's. Error register bits will be set to reflect the error but no exception will be generated.
- **IIB** 1 → causes the CPU to ignore parity errors that are detected off of the IcacheBus. Error register bits will be set to reflect the error but no exception will be generated.
- **EXT** Set when an external parity error (external parity error chip input) is detected. Reset by writing '1' to this bit position.
- **RF** Set when a CPU register file parity error is detected. Reset by writing '1' to this bit position.
- **DB** Set when a parity error is detected on the DataBus during a Load or an MFCz. Reset by writing '1' to this bit position.
- **IB** Set when a parity error is detected on the IcacheBus (instruction fetch). Reset by writing '1' to this bit position.

4.4.13. EPC

The EPC register contains the address at which instruction processing may resume after servicing an exception. For synchronous exceptions, the EPC register contains either the virtual address of the instruction which was the direct cause of the exception, or when that instruction is in a branch delay slot, the EPC contains the virtual address of the immediately preceding branch or jump instruction and the Branch Delay bit in the Cause register is set.

If the exception is caused by recoverable, temporary conditions (such as a TLB miss), the EPC contains a virtual address at the instruction which caused the exception. Thus, after correcting the conditions, the EPC contains a point at which execution can be legitimately resumed.

On R4000 processors this register is read/write. On other processors this register is read-only.

The format of the register is as follows:

MIPS I/II:

31 ... 0
EPC
32

MIPS III:

63 ... 0
EPC
64

where:

EPC   is the Exception Program Counter

4.4.14. ErrorEPC

The ErrorEPC register is similar to the EPC, but is used on ECC and parity error exceptions. It is also used to store the PC on Reset, Soft Reset, and NMI exceptions. It is read/write and contains the virtual address at which instruction processing may resume after servicing an Error. The EPC register contains either the virtual address of the instruction which was the direct cause of the exception, or when that instruction is in a branch delay slot, the EPC contains the virtual address of the immediately preceding branch or jump instruction. There is no branch delay slot indication for ErrorEPC.

This register exists only on R4000 processors.

The format of the register is as follows:

MIPS II:

31 — 0
ErrorEPC
32

MIPS III:

63 — 0
ErrorEPC
64

where:

ErrorEPC is the Error Exception Program Counter

4.4.15. CacheErr

This register exists only on R4000 processors.

CacheErr is a 32-bit read-only register for correcting and logging ECC errors in the secondary cache and parity errors in the primary cache. The register provides the cache index and status bits indicating the source and nature of the error. It is loaded when a Cache Error exception is taken.

31	30	29	28	27	26	25	24	23 22	21 … 3	2 … 0
ER	EC	ED	ET	ES	EE	EB	EI	0	SIdx	PIdx
1	1	1	1	1	1	1	1	2	19	3

where:

- **ER** indicates the type of reference (0 → instruction; 1 → data)
- **EC** indicates in which cache level the error occurred (0 → primary; 1 → secondary)
- **ED** indicates whether a data field error occurred (0 → no error: 1 → error)
- **ET** indicates whether a tag field error occurred (0 → no error; 1 → error)
- **ES** indicates the error occurred accessing processor state (e.g. primary or secondary cache) in response to an external request (0 → internal reference; 1 → external reference)
- **EE** set when the error occurred on the SysAD bus
- **EB** set when a data error occurred in addition to the instruction error indicated by the rest of the bits, which will require flushing the data cache after fixing the instruction error (should be rare)
- **EI** set on a secondary data cache ECC error while refilling the primary cache on a store miss. In this case the ECC handler must first do a Index Store Tag to invalidate the incorrect data from the primary data cache.
- **SIdx** is $pAddr_{21..3}$ of the reference that encountered the error (which is not necessarily the same as the address of the doubleword in error, but which is sufficient to locate that doubleword in the secondary cache)
- **PIdx** is $vAddr_{14..12}$ of the doubleword in error (to be used with SIdx to construct a virtual index for the primary caches)
- **0** reserved for future use

4.4.16. TagLo/TagHi

The TagLo register exists only on R4000 processors. The TagHi register is reserved for future use. Uses of TagHi on the R4000 are undefined.

The TagLo and TagHi registers are 32-bit read/write registers used to hold either the primary cache tag and parity or secondary cache tag and ECC during cache initialization, cache diagnostics, or cache error handling. The Tag registers are written by the CACHE and MTC0 instructions.

The P and ECC fields are ignored on Index Store Tag operations. Parity and ECC are computed by the store. *Note: in future versions of the R4000, the CE bit of the Status register should control this.*

The format of these registers for secondary cache operations is as follows:

TagLo:

31 ... 13	12 ... 10	9 ... 7	6 ... 0
STagLo	SState	VIndex	ECC
19	3	3	7

TagHi:

31 ... 0
STagHi
32

The format of this register for primary cache operations is as follows:

TagLo:

31 ... 8	7 ... 6	5 ... 1	0
PTagLo	PState	undef	P
24	2	5	1

TagHi:

31 ... 0
PTagHi
32

where:

undef	undefined – reserved for future use
ECC	is ECC for the STag, SState, and VIndex fields
VIndex	is a 3-bit field specifying the virtual index of the associated primary cache line ($vAddr_{14..12}$)
SState	is a 3-bit field specifying the secondary cache state
STagLo	is an 19-bit field specifying the physical address bits 35..17
P	is a 1-bit field specifying the primary tag even parity bit
PState	is a 2-bit field specifying the primary cache state (see "R4000 primary caches")
PTagLo	is a 24-bit field specifying the physical address bits 35..12
	is currently unused, but reserved for extending the physical address size > 36 bits

PTagHi

STagHi is currently unused, but reserved for extending the physical address size > 36 bits

The ECC code used for the secondary cache tag is defined by the following matrix:

Check	6	54	32	10				
Data	22 32	22 10	11 98	11 76	1111 5432	11 1098	7654	3210
0	111	1100	1100	1100	0001	0001	0001	0001
1	010	0010	0100	0100	0010	0010	0010	1111
2	000	0001	1000	1000	0100	0000	1111	1111
3	100	0100	0010	0100	1000	0100	1111	0000
4	010	1000	0001	1000	0000	1111	0100	0010
5	000	0100	0100	0010	1111	1111	0000	0100
6	100	1000	1000	0001	1111	1000	1000	1000

4.4.17. ECC

This register exists only on R4000 processors.

The ECC register is an 8-bit read/write register used to read and write the secondary cache data ECC or the primary cache data parity bits of the primary caches for initialization, cache diagnostics, or cache error handling. (Tag ECC and parity is loaded and stored to and from the TagLo register.) The ECC register is loaded by the CACHE operation Index Load Tag. It is written into the primary data cache on store instructions instead of the computed parity when the CE bit of the Status register is set, substituted for the computed instruction parity for the CACHE operation Fill, and XOR'd into the computed ECC for the secondary cache for the primary data cache CACHE operations Index Writeback Invalidate, Hit Writeback, and Hit Writeback Invalidate.

The format of this register is as follows:

ECC:

31 … 8	7 … 0
0	ECC
24	8

where:

0 reserved for future use

ECC is an 8-bit field specifying the ECC bits read from or to be written to a secondary cache, or the even byte parity bits to be read from or written to a primary cache

The ECC code used for secondary cache data is defined by the following matrix:

Check		76		54											32		10	
Data	6666 3210	55 98	5555 7654	55 32	5544 1098	4444 7654	4444 3210	3333 9876	3333 5432	3322 1098	2222 7654	2222 3210	1111 9876	1111 5432	11 10	9876	54	3210
0	0001	0100	0001	0100	0000	0001	0001	0001	0001	0000	1111	1111	0000	0001	0101	0011	1100	1111
1	0010	0100	0011	1100	1111	0010	0010	0010	0010	1111	0000	0000	0000	0010	0100	0010	0101	1111
2	0100	1100	0010	0101	1111	0100	0100	0100	0100	0000	0000	1111	1111	0100	0100	0011	0100	0000
3	1000	0101	0011	0100	0000	1000	1000	1000	1000	1111	1111	0000	1111	1000	1100	0001	0100	0000
4	0000	1010	0100	1100	0001	1111	0000	1111	1111	0001	0001	0001	0001	0000	1000	1100	1000	0001
5	0000	1000	1100	1010	0010	1111	1111	0000	0000	0010	0010	0010	0010	1111	1000	1000	1100	0010
6	1111	1000	1000	1000	0100	0000	0000	0000	1111	0100	0100	0100	0100	1111	1100	1100	1010	0100
7	1111	1100	1100	1000	1000	0000	1111	1111	0000	1000	1000	1000	1000	0000	1010	0100	1000	1000

4.4.18. Processor Revision Identifier

The PRId register is a read-only register that contains information that identifies the implementation and revision level of the processor and associated system control coprocessor.

The format of the register is as follows:

31 ... 16	15 ... 8	7 ... 0
0	Imp	Rev
16	8	8

where:

Imp is the implementation identifier
Rev is the revision identifier

The revision number can distinguish some chip revisions. However, MIPS is free to change this register at any time and does not guarantee that changes to its chips will necessarily change the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number to characterize the chip.

Very early engineering versions of the R2000 processor do not have this register.

See section "Coprocessor Unit and Revision Assignments" for a table decoding the implementation and revision field.

4.4.19. Count

There are two registers used to implement the timer services. The first is the Count register, which increments at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made.

These registers are implemented on R4000 processors only. The R6020 bus interface chip provides registers with similar functionality.

This register is both readable and writable. It is writable for diagnostic purposes, and for system initialization to synchronize two processors operating in lock-step.

31 ... 0
Count
32

The rate at which the Count register is incremented is implementation dependent. On R4000 processors this register increments at half of the maximum instruction issue rate.

4.4.20. Compare

The second is the Compare register, which maintains a stable value (does not change on its own). When the value of the Count register equals the Compare register, IP_7 in the Cause register is set, which causes an interrupt on the next execution cycle in which the interrupt is enabled.

31 ... 0
Compare
32

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

For diagnostic purposes, this register is both readable and writable. Under normal use, the Compare register is only written.

4.4.21. LLAddr

LLAddr is a read/write coprocessor register that contains the physical address read by the most recent Load Linked instruction. This register exists for diagnostic purposes, and serves no function during normal operation.

LLAddr:

31 ... 0
PAddr
32

where:

PAddr bits 35..4 of the physical address

4.4.22. WatchLo and WatchHi

R4000 processors provide a debugging feature to detect references to a physical address. Loads or stores to the location specified by the WatchLo and WatchHi registers cause an Watch trap. The format of these registers is as follows:

WatchLo:

31 ... 3	2	1	0
PAddr0	0	R	W
29	1	1	1

WatchHi:

31 ... 4	3 ... 0
0	PAddr1
28	

where:

PAddr1	bits 35..32 of the physical address
PAddr0	bits 31..3 of the physical address
R	trap on read references
W	trap on write references

4.4.23. Config

The Config register specifies the various configuration options selected on R4000 processors. Some configuration options (bits 31..6) are set by the hardware during reset, and are included in this register read-only for software. For more information on these options, refer to the R4000 Interface Specification. Other configuration options are read-write and controlled only by software (bits 5..0).

On Reset, all read-write fields are undefined. This register should be initialized by software before the caches are used. The caches should be completely written back before changing block sizes, and re-initialized after any change is effected.

31	30 28	27 24	23 22	21	20	19 18	17	16
CM	EC	EP	SB	SS	SW	EW	SC	SM
1	3	4	2	1	1	2	1	1

15	14	13	12	11 9	8 6	5	4	3	2 0
BE	EM	EB	0	IC	DC	IB	DB	CU	K0
1	1	1	1	3	3	1	1	1	3

where:

- CM Master-Checker mode
- EC SYSAD port clock ratio
  - 0 → processor clock divided by 2
  - 1 → processor clock divided by 3
  - 2 → processor clock divided by 4
- EP Pattern for writeback data on SYSAD port
  - 0 → D
  - 1 → DDx
  - 2 → DDxx
  - 3 → DxDx
  - 4 → DDxxx
  - 5 → DDxxxx
  - 6 → DxxDxx
  - 7 → DDxxxxx
  - 8 → DxxxDxxx
- SB Block size for secondary cache
  - 0 → 4 words
  - 1 → 8 words
  - 2 → 16 words
  - 3 → 32 words
- SS Split secondary cache mode
  - 0 → instruction and data mixed in secondary cache
  - 1 → instruction and data separated by $SCAddr_{17}$
- SW Secondary cache port width
  - 0 → 128-bit data path to secondary cache
  - 1 → 64-bit data path to secondary cache
- EW Reserved for SYSAD port width
  - 0 → 64-bit SYSAD port
  - 1 → 32-bit SYSAD port
- SC Secondary cache present
  - 0 → secondary present
  - 1 → no secondary cache

SM	Use Dirty Shared coherency state 0 → Dirty Shared state enabled 1 → Dirty Shared state disabled
BE	BigEndianMem 0 → memory and kernel are little-endian 1 → memory and kernel are big-endian
EM	ECC mode enable
EB	Block ordering (if set then sequential, else sub-block)
IC	Instruction cache size is 2^{12+IC} bytes
DC	Data cache size is 2^{12+DC} bytes
IB	Instruction cache block size (read-write) 0 → 16 bytes 1 → 32 bytes
DB	Data cache block size (read-write) 0 → 16 bytes 1 → 32 bytes
CU	Update on Store Conditional (read-write) 0 → Store Conditional uses coherency algorithm specified by TLB 1 → Store Conditional uses cacheable coherent update on write
K0	kseg0 coherency algorithm (see EntryLo) (read-write)
0	reserved

4.5. System Control Coprocessor Instructions

Although load and store instructions to coprocessors are generally permitted elsewhere, load/store instructions (LWC0, SWC0) are not valid to the system control coprocessor. The move control to/from coprocessor instructions (CFC0, CTC0) are also not valid to this coprocessor. The move to/from coprocessor instructions (MFC0, MTC0, and on R4000 processors, DMFC0, and DMTC0) are the only valid mechanism for reading and writing the system control coprocessor registers.

For the 64-bit coprocessor zero registers (EntryHi, XContext, EPC, and ErrorEPC), MTC0 transfers the entire 64-bit GPR value. Bits 63..32 of the GPR should be equal to bit 31.

The system control coprocessor operations include instructions to directly read, write, and probe TLB entries, and to modify the operating modes in preparation for a return to user-mode or interrupt-enabled states.

4.5.1. Move To/From System Control Coprocessor

Instruction Format:

31 26	25 21	20 16	15 11	10 0
COP0	sub	rt	rd	0
6	5	5	5	11

where:

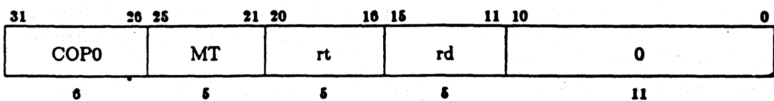
COP0	is the 6-bit Coprocessor Zero operation code
sub	is a 5-bit coprocessor sub-operation field
rt	is a 5-bit general register specifier
rd	is a 5-bit coprocessor register specifier
0	must be zero

These coprocessor operations are moves between general purpose and system control coprocessor registers.

MOVE TO SYSTEM CONTROL COPROCESSOR

Format:

MTC0 rt,rd



Description:

The contents of general register rt are loaded into coprocessor register rd of coprocessor unit 0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load and store instructions and TLB operations immediately prior to and after this instruction are undefined.

On MIPS III processors, for 64-bit coprocessor 0 registers, all 64 bits of rt are transferred, and this instruction is identical to DMTC0.

MIPS I/II operation:

T: data ← GPR[rt]

T+1: CPR[0, rd] ← data

MIPS III operation:

T: data ← GPR[rt]

T+1: CPR[0, rd] ← data

Exceptions:

Coprocessor unusable exception

MOVE FROM SYSTEM CONTROL COPROCESSOR

Format:

MFC0 rt,rd

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COP0	MF	rt	rd	0
6	5	5	5	11

Description:

The contents of coprocessor register rd of coprocessor zero are loaded into general register rt.

On MIPS III processors, only the low 32 bits of data are transferred; the high 32 bits of the result are written with the sign of the low bits.

MIPS I/II operation:

T: data ← CPR[0, rd]

T+1: GPR[rt] ← data

MIPS III operation:

T: data ← CPR[0, rd]

T+1: $GPR[rt] \leftarrow (data_{31})^{32} \| data_{31..0}$

Exceptions:

Coprocessor unusable exception

DOUBLE MOVE TO SYSTEM CONTROL COPROCESSOR

Format:

DMTC0 rt,rd

31–26	25–21	20–16	15–11	10–0
COP0	DMT	rt	rd	0
6	5	5	5	11

Description:

The contents of general register rt are loaded into coprocessor register rd of coprocessor unit 0.

This instruction is valid only on MIPS III processors. All 64 bits of the coprocessor 0 are written from the general register source. The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load and store instructions and TLB operations immediately prior to and after this instruction are undefined.

MIPS III operation:

T: data ← GPR[rt]

T+1: CPR[0, rd] ← data

Exceptions:

Coprocessor unusable exception

DOUBLE MOVE FROM SYSTEM CONTROL COPROCESSOR

Format:

DMFC0 rt,rd

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
COP0	DMF	rt	rd	0
6	5	5	5	11

Description:

The contents of coprocessor register rd of coprocessor zero are loaded into general register rt.

This instruction is valid only on MIPS III processors. All 64 bits of the general register destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

MIPS III operation:

T: data ← CPR[0, rd]

T+1: GPR[rt] ← data

Exceptions:

Coprocessor unusable exception

4.5.2. System Control Coprocessor Operations

Instruction Format:

31 — 26	25	24 — 6	5 — 0
COP0	CO	0	function
6	1	19	6

where:

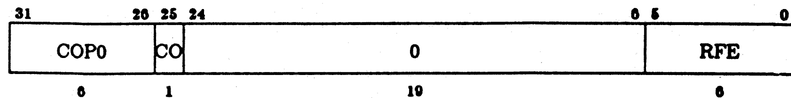
COP0	is the 6-bit coprocessor zero operation code
CO	specifies an internal coprocessor operation
0	must be zero
function	is a 5-bit function field

Description	function
Restore From Exception	RFE
Exception Return	ERET
Read TLB entry pointed to by index	TLBR
Write TLB entry pointed to by index	TLBWI
Write TLB entry pointed to by random	TLBWR
Probe TLB for matching entry	TLBP

RESTORE FROM EXCEPTION

Format:

RFE



Description:

RFE restores the "previous" interrupt mask and kernel/user-mode bits (IEp and KUp) of the Status register (SR) into the corresponding "current" status bits (IEc and KUc), and restores the "old" status bits (IEo and KUo) into the corresponding previous status bits (IEp and KUp). The old status bits remain unchanged.

The architecture does not specify the operation of memory references associated with load/store instructions immediately prior to an RFE instruction. Normally, the RFE instruction follows in the delay slot of a JR (jump register) instruction to restore the PC.

This instruction is not implemented on R4000 processors. Use ERET instead.

An RFE executed between a LL and SC also causes the SC to fail.

R2000, R3000, R6000 operation:

T: $SR \leftarrow SR_{31..4} \| SR_{5..2}$
$LLbit \leftarrow 0$

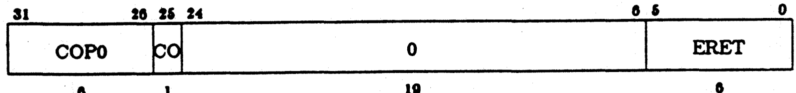
Exceptions:

Coprocessor unusable exception
Reserved instruction exception (R4000)

EXCEPTION RETURN

Format:

ERET



Description:

ERET is the R4000 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing an error trap (SR_2=1), then load the PC from the ErrorEPC and clear the ERL bit of the status register (SR_2). Otherwise (SR_2=0), load the PC from the EPC, and clear the EXL bit of the status register (SR_1).

An ERET executed between a LL and SC also causes the SC to fail.

R4000:

T: if SR_2=1 then
  PC ← ErrorEPC
  SR ← $SR_{31..3}$ || 0 || $SR_{1..0}$
else
  PC ← EPC
  SR ← $SR_{31..2}$ || 0 || SR_0
endif
LLbit ← 0

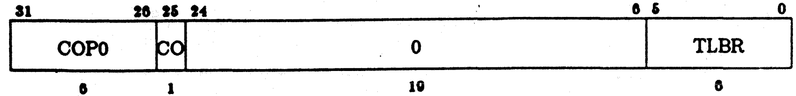
Exceptions:

Coprocessor unusable exception
Reserved instruction exception (non-R4000)

READ INDEXED TLB ENTRY

Format:

TLBR



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 operation is invalid (the architecture does not specify the result) if the contents of the TLB Index register is greater than the number of TLB entries in the processor.

This instruction is only valid for processors with an on-chip associative TLB. This instruction is not valid on R6000 processors.

Also, the G bit (controls ASID matching) read from the TLB is written into both EntryLo0 and EntryLo1.

R2000, R3000 operation:

T: EntryHi $\leftarrow$ $TLB[Index_{13..8}]_{63..32}$
EntryLo $\leftarrow$ $TLB[Index_{13..8}]_{31..0}$

MIPS II R4000 operation:

T: PageMask $\leftarrow$ $TLB[Index_{5..0}]_{128..96}$
EntryHi $\leftarrow$ $TLB[Index_{5..0}]_{95..64}$ and not $TLB[Index_{5..0}]_{128..96}$
EntryLo1 $\leftarrow$ $TLB[Index_{5..0}]_{63..32}$
EntryLo0 $\leftarrow$ $TLB[Index_{5..0}]_{31..0}$

MIPS III R4000 operation:

T: PageMask $\leftarrow$ $TLB[Index_{5..0}]_{255..192}$
EntryHi $\leftarrow$ $TLB[Index_{5..0}]_{191..128}$ and not $TLB[Index_{5..0}]_{255..192}$
EntryLo1 $\leftarrow$ $TLB[Index_{5..0}]_{127..65}$ $\|$ $TLB[Index_{5..0}]_{140}$
EntryLo0 $\leftarrow$ $TLB[Index_{5..0}]_{63..1}$ $\|$ $TLB[Index_{5..0}]_{140}$

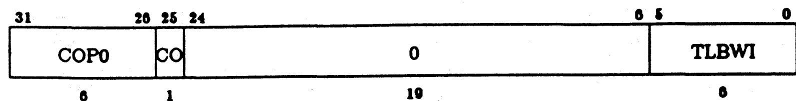
Exceptions:

Coprocessor unusable exception

WRITE INDEXED TLB ENTRY

Format:

TLBWI



Description:

The TLB entry pointed at by the contents of the TLB Index register is loaded with the contents of the EntryHi and EntryLo registers.

The operation is invalid (the architecture does not specify the result) if the contents of the TLB Index register is greater than the number of TLB entries in the processor.

On R4000 processors, the G bit of the TLB is written with the logical and of the G bits in EntryLo0 and EntryLo1.

This instruction is only valid for processors with an on-chip associative TLB. This instruction is not valid on R6000 processors.

R2000, R3000 operation:

T: $TLB[Index_{13..8}] \leftarrow EntryHi \,||\, EntryLo$

R4000 operation:

T: $TLB[Index_{5..0}] \leftarrow PageMask \,||\, (EntryHi \text{ and not } PageMask) \,||\, EntryLo1 \,||\, EntryLo0$

Exceptions:

Coprocessor unusable exception

WRITE RANDOM TLB ENTRY

Format:

TLBWR

31 … 26	25	24 … 6	5 … 0
COP0	CO	0	TLBWR
6	1	19	6

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.

On R4000 processors, the G bit of the TLB is written with the logical and of the G bits in EntryLo0 and EntryLo1.

This instruction is only valid for processors with an on-chip associative TLB. This instruction is not valid on R6000 processors.

R2000, R3000 operation:

T: $TLB[Random_{13..8}] \leftarrow EntryHi \,||\, EntryLo$

R4000 operation:

T: $TLB[Random_{5..0}] \leftarrow PageMask \,||\, (EntryHi \text{ and not } PageMask) \,||\, EntryLo1 \,||\, EntryLo0$

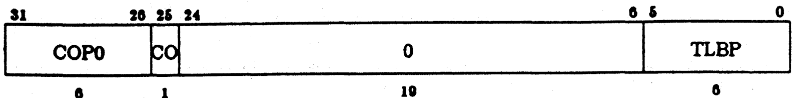
Exceptions:

Coprocessor unusable exception

PROBE TLB FOR MATCHING ENTRY

Format:

TLBP



Description:

The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the high-order bit of the Index register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

This instruction is only valid for processors with an on-chip associative TLB. This instruction is not valid on R6000 processors.

R2000, R3000 operation:

T: $Index \leftarrow 1 \| 0^{31}$
for i in 0..TLBEntries-1
if $(TLB[i]_{63..44} = EntryHi_{31..12})$ and $(TLB[i]_{8}$ or $(TLB[i]_{43..38} = EntryHi_{11..6}))$ then
$Index \leftarrow 0^{18} \| i_{5..0} \| 0^{8}$
endif
endfor

MIPS II R4000 operation:

T: $Index \leftarrow 1 \| 0^{25} \| 1^{6}$
for i in 0..TLBEntries-1
if $(TLB[i]_{95..77} = EntryHi_{31..12})$ and $(TLB[i]_{76}$ or $(TLB[i]_{71..64} = EntryHi_{7..0}))$ then
$Index \leftarrow 0^{26} \| i_{5..0}$
endif
endfor

MIPS III R4000 operation:

T: $Index \leftarrow 1 \| 0^{31}$
for i in 0..TLBEntries-1
if $((TLB[i]_{167..141}$ and not $(0^{15} \| TLB[i]_{216..205})) = EntryHi_{39..13}$ and not $(0^{15} \| TLB[i]_{216..205}))$
and $(TLB[i]_{140}$ or $(TLB[i]_{135..128} = EntryHi_{7..0}))$ then
$Index \leftarrow 0^{26} \| i_{5..0}$
endif
endfor

Exceptions:

Coprocessor unusable exception

CACHE

Format:

CACHE op,offset(base)

31 … 26	25 … 21	20 … 16	15 … 0
CACHE	base	op	offset
6	5	5	16

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The virtual address is translated to a physical address using the TLB. The 5-bit sub-opcode specifies a cache operation for that address.

This operation is only valid for R4000 processors. If coprocessor 0 is not usable (user or supervisor mode the coprocessor 0 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 when none is present, is undefined. The operation of this instruction on uncached addresses is undefined.

The "Index" operations use part of the virtual address to specify a cache block. The operation is performed unconditionally. For a primary cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, $vAddr_{CACHESIZE..BLOCKSIZE}$ specifies the block. For a secondary cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, $pAddr_{CACHESIZE..BLOCKSIZE}$ specifies the block. Index Load Tag also uses $vAddr_{BLOCKSIZE..3}$ to select the doubleword for reading ECC or parity. When the CE bit of the Status register is set, Hit Writeback, Hit Writeback Invalidate, Index Writeback Invalidate and Fill also use $vAddr_{BLOCKSIZE..3}$ to select the doubleword that has its ECC or parity modified.

The "Hit" operations access the specified cache as normal data references, and perform 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.

Writeback from a primary cache goes to the secondary if there is one, and to memory otherwise. Writeback from a secondary cache always goes to memory. A secondary writeback always writes the "most recent" data; the data comes from the primary data cache if present there and modified (the Writeback bit set), and otherwise from the specified secondary. The address to write is specified by the cache tag, and not the translated physical address (these can be different for "Index" operations).

TLB Refill and TLB Invalid exceptions can occur on any operation. For the Index operations, where the physical address is irrelevant, unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions.

Bits 17..16 of the instruction specify the cache as follows:

code	name	cache
0	I	primary instruction
1	D	primary data
2	SI	secondary instruction
3	SD	secondary data (or combined instruction/data)

Bits 20..18 of the instruction specify the operation as follows:

code	caches	name	operation
0	I, SI	Index Invalidate	Set the cache state of the cache block to Invalid.
0	D	Index Writeback Invalidate	Examine the cache state and Writeback bit (W bit) of the primary data cache block at the index specified by the virtual address. If the state is not Invalid and the W bit is set, then write back the block to the secondary cache (if present) or to memory (if no secondary cache). The address to write is taken from the primary cache tag. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR'd into the computed check bits during the write to the secondary cache for the addressed doubleword. Set the cache state of primary cache block to Invalid. The W bit is unchanged (and irrelevant because the state is Invalid).
0	SD	Index Writeback Invalidate	Examine the cache state of the secondary data cache block at the index specified by the physical address. If the state is Dirty Exclusive or Dirty Shared, then writeback the block to memory. The address to write is taken from the secondary cache tag. Like all secondary writebacks, the operation will write any modified data (W bit set) from the primary data cache. Unlike Hit Writeback Invalidate the operation does *not* invalidate or clear the W bit in the primary data cache. In all cases, set the secondary cache block state to Invalid.
1	all	Index Load Tag	Read the tag for the cache block at the specified index and place it into the TagLo and TagHi coprocessor zero registers, ignoring ECC and parity errors. Also load the data ECC or parity bits into the ECC register.
2	all	Index Store Tag	Write the tag for the cache block at the specified index from the TagLo and TagHi coprocessor zero registers. *Note: in future R4000s, the CE bit of the Status register should control whether computed ECC/parity or TagLo ECC and P fields are used.*

code	caches	name	operation
3	SD	Create Dirty Exclusive	This operation is used to avoid needlessly loading data from memory when writing new contents into an entire cache block. If the cache block is valid but does not contain the specified address (a valid miss) the secondary block is vacated: the data is written back to memory if dirty and then all matching blocks in both primary caches are invalidated. As usual during a secondary writeback, if the primary data cache contains modified data (matching blocks with W bit set) that modified data is written to memory. If the cache block is valid and does contain the specified physical address (a hit), then the operation cleans up the primary caches to avoid virtual alias problem: all blocks in both primary caches that match the secondary line are invalidated (without writeback). Note that the search for matching primary blocks uses the virtual index of the PIdx field of the secondary cache tag (the virtual index when the location was last used) and not the virtual index of the virtual address used in the operation (the virtual index where the location will now be used). If the secondary tag and address do not match (miss), or the tag and address do match (hit) and the block is in a shared state, send an invalidate for the specified address on the system interface. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive, and set the virtual index field from the virtual address. The CH bit in the Status register is set or cleared to indicate a hit or miss.
3	D	Create Dirty Exclusive	This operation is used to avoid needlessly loading data from secondary cache or memory when writing new contents into an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the secondary cache if present or memory otherwise. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive.

code	caches	name	operation
4	I, D	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid.
4	SI, SD	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid and also invalidate all matching blocks, if present, in the primary caches (the PIdx field of the secondary tag is used to determine the locations in the primaries to search). The CH bit in the Status register is set or cleared to indicate a hit or miss.
5	D	Hit Writeback Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the cache block invalid. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR'd into the computed check bits of during the write to the secondary cache for the addressed doubleword.
5	SD	Hit Writeback Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the secondary cache block and all matching blocks in both primary caches invalid. As usual with secondary writebacks, modified data in the primary data cache (matching block with the W bit set) is used during the writeback. The PIdx field of the secondary tag is used to determine the locations in the primaries to check for matching primary blocks. The CH bit in the Status register is set or cleared to indicate a hit or miss.
5	I	Fill	Fill the primary instruction cache block from secondary or 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.

code	caches	name	operation
6	D	Hit Writeback	If the cache block contains the specified address, and the Writeback bit is set, write back the data, and clear the Writeback bit. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR'd into the computed check bits during the write to the secondary cache for the addressed doubleword.
6	SD	Hit Writeback	If the cache block contains the specified address, and the cache state is Dirty Exclusive or Dirty Shared, write back the data to memory, and change the cache state to Clean Exclusive or Shared, respectively. The CH bit in the Status register is set or cleared to indicate a hit or miss. The writeback looks in the primary data cache for modified data, but does *not* invalidate or clear the Writeback bit in the primary data cache. This state, though perhaps not intuitive, is consistent since the primary block contains data that is at least as current as in memory or secondary cache. A subsequent writeback of the primary line without futher modification would be redundant, but not incorrect.
6	I	Hit Writeback	If the cache block contains the specified address, write back the data unconditionally. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR'd into the computed check bits during the write to the secondary cache for the addressed doubleword.
7	SI, SD	Hit Set Virtual	This operation is used to change the virtual index of secondary cache contents avoiding unnecessary memory operations. If the cache block contains the specified address, invalidate matching blocks in the primary caches at the index formed by concatenating PIdx in the secondary cache tag (not the virtual address of the operation) and $vAddr_{11..4}$, and then set the virtual index field of the secondary cache tag from the specified virtual address. Modified data in the primary data cache is not preserved by the operation and should be explcitly written back before this operation. The CH bit in the Status register is set or cleared to indicate a hit or miss.

R4000 operation:

T: $vAddr \leftarrow ((offset_{15})^{48} \,\|\, offset_{15..0}) + GPR[base]$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
CacheOp (op, vAddr, pAddr)

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception (R4000)
Reserved instruction exception (non-R4000)

FLUSH

Format:

LWR offset(base)

31–26	25–21	20–16	15–0
LWR	base	0	offset
6	5	5	16

Description:

This operation is only valid for R6000 processors. This operation is effected by using the LWR opcode with the MM bit of the status register set.

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

On R6000 processors, bits 17..7 are used to specify the cache line and $offset_0$ specifies which cache set in the secondary cache to which the operation occurs. When a 2MB secondary cache is used, bits 26..25 provide additional address bits.

If the cache line at the specified location is dirty, it is written to memory, and the cache line state is updated to reflect that the line is now clean/consistent.

Operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
FlushCaches (vAddr, $offset_0$)

Exceptions:

none

INVALIDATE

Format:

SWR offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
SWR	base	0	offset
6	5	5	16

Description:

This operation is only valid for R6000 processors. This operation is effected by using the SWR opcode with the MM bit of the status register set.

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

On an R6000 processor, bits 17..7 of the effective address specify the cache line to invalidate and $offset_0$ specifies the set. When a 2MB secondary cache is used, bits 26..25 provide additional address bits. The addressed virtual cache tag in the secondary cache is marked invalid.

At the same time, the effective address specifies cache line in the primary data cache ($vAddr_{13..3}$) or instruction cache ($vAddr_{15..5}$), and $offset_0$ specifies which of the two caches (0 → instruction cache; 1 → data cache) to which the operation occurs. The addressed virtual cache tag in the primary cache is invalidated.

Because this operation simultaneously invalidates a primary cache line and a secondary cache line, it is normally only well-defined when used to invalidate the virtual tags of the entire cache.

Operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
InvalidateCaches ($vAddr$, $offset_0$)

Exceptions:

none

LOAD FROM CACHE

Format:

LWL rt,offset(base)

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LWL	base	rt	offset
6	5	5	16

Description:

This operation is only valid on a processor with an in-cache TLB (R6000). This operation is effected by using the LWL opcode with the MM bit of the status register set.

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

$Offset_0$ (not the effective address) selects the secondary cache set accessed.

Bits 17..2 of the effective address select the word of the secondary cache for a 512KB secondary cache. The physical tags begin at 0x3E000 and the TLB entries begin at 0x3C000. Bits 26..25 and 17..2 select the word for a 2MB secondary cache.

The contents of the word at the TLB location specified by the offset and effective address are loaded into general register rt. This instruction is not interlocked; referencing rt in the next two instructions is undefined.

If the virtual tags for this access do not match in the secondary cache, the CM0 or CM1 bit in the status register (SR) is set. Regardless of whether the tags match, the data contained in the addressed location is loaded into general register rt.

This instruction must not be placed in a branch delay slot.

Operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \parallel offset_{15..0}) + GPR[base]$
$data \leftarrow LoadCache\ (vAddr, offset_0)$

T+2: $GPR[rt] \leftarrow data$

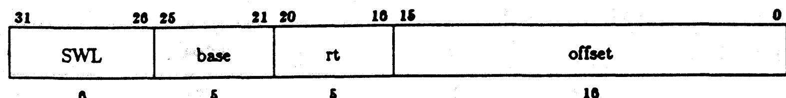
Exceptions:

none

STORE TO CACHE

Format:

SWL rt,offset(base)



Description:

This operation is only valid on a processor with an in-cache TLB (R6000). This operation is effected by using the SWL opcode with the MM bit of the status register set.

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address.

$Offset_0$ (not the effective address) selects the secondary cache set accessed. $Offset_1$ is the value to store in the Global bit of the virtual tag.

For a 512KB secondary cache, bits 17..2 of the effective address select the word of the secondary cache. The physical tags begin at 0x3E000 and the TLB entries begin at 0x3C000. For a 2MB secondary cache, bits 26..25 and 17..2 select the word.

The contents of general register rt are stored at the offset and set specified by the effective address and $offset_0$.

The corresponding virtual tag in the secondary cache is set with bits 31..14 of the virtual address, with the global bit set to $offset_1$, and the line is marked writable, valid, and not dirty.

This instruction must not be placed in a branch delay slot.

Operation:

T: $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
$data \leftarrow GPR[rt]$
$StoreCache\ (data, vAddr, offset_0, offset_1)$

Exceptions:

none

5. Exception Handling

The exception handling system is responsible for efficiently handling relatively infrequent events, such as translation misses, arithmetic overflow, I/O interrupts, and system calls. These events cause the interruption of the normal flow of execution; aborting instructions which cause exceptional conditions and all those which follow and have already begun executing, and a direct jump into a designated handler routine.

The architecture defines a minimal amount of additional state which is saved in coprocessor registers in order to facilitate the analysis of the cause of the exception, the servicing of the event which caused it, and the resumption of the original flow of execution, when applicable.

5.1. Exception operation

To handle an exception, the processor forces execution of a handler at a fixed address in kernel mode with interrupts disabled. To resume, the PC, operating mode, and interrupt enable must be restored, and thus it is this context that must be saved when an exception is taken.

When an exception occurs, the EPC is loaded with an appropriate restart location at which execution may resume after servicing the exception. The EPC also can be thought of as containing the address of the instruction that caused the exception, or if the instruction was executing in a branch delay slot, the address of the immediate predecessor of the instruction.

To save and restore the operating mode and interrupt enable, the R2000, R3000, and R6000 use a three-level stack for the KU and IE bits. The KUc and IEc bits always specify whether the machine is executing in kernel or user mode and whether interrupts are enabled or disabled. When an exception, other than Reset, is taken, the values of KUp, IEp, KUc, and IEc are saved in KUo, IEo, KUp and IEp, and KUc and IEc are cleared. Thus the machine begins operating in kernel mode with interrupts disabled. On return from exception (RFE instruction), the KUc, IEc, KUp, and IEp bits are restored from KUp, IEp, KUo, and IEo. The KUo and IEo bits exist to allow an exception to be taken in a first level exception handler. This single additional level of exception handling is intended for use in the TLB refill handler. This mechanism is not appropriate for nested interrupts, which may be implemented by software to save and restore the status register, EPC, and other context on a stack.

R2000 and R3000 Reset:

```
T:      undefined
        Random ← TLBENTRIES-1
        SR ← 0x00400000
        PC ← 0xbfc00000
```

R6000 Reset:

```
T:      undefined
        SR ← 0x00400000
        PC ← 0xbfc00000
```

R2000, R3000, and R6000 exceptions (except Reset):

T: Cause ← BD || 0 || CE || 0^{12} || $Cause_{15..8}$ || 0 || ExcCode || 0^2
EPC ← PC
SR ← $SR_{31..6}$ || $SR_{3..0}$ || 0^2
if SR_{22} = 1 then
  PC ← 0xbfc00100 + vector
else
  PC ← 0x80000000 + vector
endif

R4000 processors use a different mechanism for saving and restoring operating mode and interrupt status in order to support a supervisor mode and fast TLB refill for all address spaces. The 3 sets of KU and IE bits are replaced by a single interrupt enable (IE), a base operating mode (user, supervisor, kernel), an exception level (normal, exception), and an error level (normal, error). Interrupts are enabled with IE = 1 and both levels are normal. The operating mode is specified by the base mode when the exception level is normal, and is kernel when exception level is set. Returning from an exception consists of resetting the exception level to normal (see the ERET instruction).

R4000 Reset:

T: undefined
Random ← TLBENTRIES-1
Wired ← 0
Config ← $undefined^{11}$ || MC || MW || ST || $undefined^4$ || SW || $undefined^4$ || DC || IC || E
ErrorEPC ← PC
SR ← $SR_{31..23}$ || 1 || 0 || 0 || $SR_{19..3}$ || 1 || $SR_{1..0}$
PC ← 0xffff_ffff_bfc0_0000

R4000 Soft Reset and NMI:

T: ErrorEPC ← PC
SR ← $SR_{31..23}$ || 1 || 0 || 1 || $SR_{19..3}$ || 1 || $SR_{1..0}$
PC ← 0xffff_ffff_bfc0_0000

R4000 exceptions (except Reset, Soft Reset, NMI, and Cache Error):

T: Cause ← BD || 0 || CE || 0^{12} || $Cause_{15..8}$ || 0 || ExcCode || 0^2
if SR_1=0 then
  EPC ← PC
endif
SR ← $SR_{31..2}$ || 1 || SR_0
if SR_{22} = 1 then
  PC ← 0xffff_ffff_bfc0_0200 + vector
else
  PC ← 0xffff_ffff_8000_0000 + vector
endif

R4000 Cache Error exception:

T: ErrorEPC ← PC
CacheErr ← ER || EC || ED || ET || ES || EI || 0^4 || Index || 0^3
SR ← $SR_{31..3}$ || 1 || $SR_{1..0}$
if SR_{22} = 1 then
PC ← 0xffff_ffff_bfc0_0200 + 100
else
PC ← 0xffff_ffff_a000_0000 + 100
endif

5.2. Precision of Exceptions'

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

Interrupts, which are generated by external devices attached to the processor, have various meanings, depending on the system environment into which the MIPS processor in designed. Variations in the memory system design, such as possible inclusion of a write-buffer, can affect the meaning of bus error exceptions and the location and means of accessing relevant parameters to service them. Descriptions of these environments can be found in Chapter 3 and documents such as "MIPS R2300 CPU Board," and similar implementation-level documentation. As much as possible, this architectural description of the exception handling system details what state information can and cannot be relied upon over variations in implementations and environments.

In some cases, however, the characteristics of the machine's pipeline staging cannot guarantee that all state in the processor and associated system remains completely unchanged as a result of the possibly incomplete execution of instructions immediately following an instruction which causes an exception. Examples of these state changes include:

Instructions may have been read from memory and loaded into the instruction cache.

The multiply/divide registers, HI and LO may have been altered by a MULT/MULTU, DIV/DIVU, or MTHI/MTLO instruction.

Coprocessor zero registers may have been altered as a result of MTC0 instructions or COP0 operations.

When a bus error occurs on a cached, word, memory write operation, the cache has been updated to contain the data that was attempted to have been written.

These effects can normally be ignored, however, because in these cases, enough of the state of the machine is restored, so that execution may always properly resume after servicing the exception. In the case of the multiply/divide and coprocessor zero registers, so-called "reorganization constraints" prevent the use of instructions in an organization for which the hardware cannot guarantee interruptibility.

5.3. Exception Types

The table below lists each of the exception types which are handled by the processor, giving a interpretation of the meaning of each exception.

Name	Cause code	Description
Reset	-	The reset exception aborts the current execution stream and starts executing at the reset vector. A separate vector is provided for this exception.
Soft Reset	-	(R4000 only) The soft reset exception aborts the current execution stream and starts executing at the reset vector. Soft Reset is used to re-initialize the processor without going through the entire Reset hardware sequence.
NMI	-	(R4000 only) This is a non-maskable interrupt requested by external logic. The Reset vector is used for this interrupt.
TLB refill	TLBL TLBS	The referenced address did not match any TLB entry. A separate vector is provided for this exception. On R4000 processors this vector is used for all virtual address spaces when the Status register EXL bit is 0; for the R2000, R3000, and R6000 this exception is used only for references to the user address space (from either kernel or user-mode).
Extended addressing TLB refill	TLBL TLBS	The referenced address did not match any TLB entry and the referenced address space is using extended addressing (MIPS III). A separate vector is provided for this exception when the SR exception level (EXL) is 0.
TLB invalid	TLBL TLBS	Virtual-address reference that matches an invalid TLB entry.
TLB modified	Mod	An attempt to write to a virtual address that did not have D bit in the corresponding TLB entry set.
Bus error	IBE DBE	An external interrupt signaled by bus interface circuitry. A bus error is signaled for events such as bus time-out, bus parity errors, and invalid memory addresses or access types.
Address error	AdEL AdES	An attempt is made to load, fetch, or store a word not aligned on a word boundary or load or store a halfword not aligned on a halfword boundary, or load or store a doubleword not aligned on a doubleword boundary, or to reference a privileged virtual address.
Integer overflow	Ov	An add or subtract operation causes two's complement overflow.
Trap	Tr	A trap operation was executed with a true condition.
System call	Sys	Execution of a SYSCALL instruction.
Breakpoint	Bp	Execution of a BREAK instruction.
Reserved Instruction	RI	Execution of an instruction with a reserved major operation code (bits 31..26), or a SPECIAL instruction with a reserved minor operation code (bits 5..0).

Name	Cause code	Description
Coprocessor Unusable	CpU	Execution of a coprocessor instruction for which the corresponding coprocessor-usable bit was not set.
Floating Point	FPE	One of several floating-point exceptions. See chapter 3.
Interrupt	Int	One of several interrupt conditions. See the Cause register.
Machine Check	MC	(R6000 only) Fatal parity error detected.
Uncached LDC1/SDC1	NCD	(R6000 only) LDC1/SDC1 to an uncached address.
Virtual Coherency	VCEI VCED	(R4000 only) Different virtual indexes used in primary cache for the same physical location.
Cache error	-	(R4000 only) Parity error in primary cache, or ECC error in secondary cache. A separate vector is provided for this exception.
Watch	WATCH	(R4000 only) Reference to WatchHi/WatchLo address.

5.4. In-cache TLB Exceptions

This document does not currently include information on how the exception handling is affected by the inclusion of an in-cache TLB, rather than an on-chip TLB, as occurs in R6000 processors.

5.5. Exception vectors

The Reset, Soft Reset, and NMI exceptions are always vectored to 0xbfc00000. The address for other exceptions is a combination of a vector offset and a base address determined by the BEV bit of the SR.

Vector Base (except Reset, Soft Reset, NMI, and Cache Error):

BEV	R2000, R3000, and R6000 Vector Base	R4000 Vector Base
0	0x80000000	0xffff_ffff_8000_0000
1	0xbfc00100	0xffff_ffff_bfc0_0200

The vector base for the R4000 Cache Error exception is in kseg1 (0xffff_ffff_a000_0000) instead of kseg0 (0xffff_ffff_8000_0000) when BEV is zero. It is 0xffff_ffff_bfc0_0200 when BEV is set.

Vector Offset:

Exception	R2000, R3000, and R6000 Vector Offset	R4000 Vector Offset
TLB refill, EXL = 0	000	000
XTLB refill, EXL = 0	-	080
Cache Error	-	100
others	080	180

5.6. Priority of Exceptions

When multiple exceptions can occur for a single instruction, only one exception is reported, with priority given in the following order:

Reset
Soft Reset (R4000 only)
NMI (R4000 only)
Machine Check (R6000 only)
Address error – Instruction fetch
TLB refill – Instruction fetch
TLB invalid – Instruction fetch
Cache error – Instruction fetch (R4000 only)
Virtual Coherency – Instruction fetch (R4000 only)
Bus error – Instruction fetch
Integer overflow, Trap, System call, Breakpoint,
  Reserved Instruction, Coprocessor Unusable,
  or Floating Point Exception
Address error – Data access
TLB refill – Data access
TLB invalid – Data access
TLB modified – Data write
Cache error – Data access (R4000 only)
Watch (R4000 only)
Virtual Coherency – Data access (R4000 only)
Uncached LDC1/SDC1 (R6000 only)
Bus error – Data access
Interrupt

5.6.1. Reset

Cause:

The Reset exception occurs in response to the Reset pin. Execution begins at the reset vector when Reset is deasserted. This exception is not maskable.

Handling:

A special exception vector (0xbfc00000) is provided for this exception. This vector is located within the unmapped and uncached address space so that the cache and TLB need not be initialized to handle this exception.

The contents of all registers are undefined when this exception occurs, except for the Random register, which is initialized to TLBENTRIES-1, the Wired register, which is initialized to 0, and the Status register. For R2000, R3000, and R6000 processors, the Status register is undefined, except for TS, SWc, KUc, and IEc, which are zero, and BEV, which is one. For R4000 processors, the contents of the Status register are undefined, except for SR and TS, which are zero, and ERL and BEV, which are one.

Servicing:

The Reset exception is serviced by initializing all processor registers, coprocessor registers, the caches and the memory system, performing diagnostic tests, and bootstrapping the operating system. The reset exception vector is selected to appear within the uncached, unmapped memory space of the machine so that instructions may be fetched and executed while the cache and virtual memory system are still in an undefined state.

5.6.2. Soft Reset

Cause:

The Soft Reset exception occurs in response to the Soft Reset input signal. Execution begins at the reset vector when Soft Reset is deasserted. This exception is not maskable. It is implemented only on R4000 processors.

Handling:

The reset exception vector (0xbfc00000) is used for this exception. This vector is located within the unmapped and uncached address space so that the cache and TLB need not be initialized to handle this exception. The SR bit of the Status register is set to differentiate this exception from Reset.

Soft Reset primary purpose is to re-initialize the processor after a fatal error, such as a Master/Checker mismatch. Unlike NMI, all cache and bus state machines are reset by this exception. Like Reset it may be used on a processor in any state; the caches, TLB, and normal exception vectors need not be properly initialized. The contents of all registers are preserved when this exception occurs, except for the ErrorEPC register, which contains the restart PC, and the ERL bit of the Status register, which is set to one. Because Soft Reset may abort cache and bus operations, cache and memory state is undefined when this exception occurs.

Servicing:

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

5.6.3. Non-maskable Interrupt

Cause:

The NMI exception occurs in response to the falling edge of the Non-maskable Interrupt pin. This exception is not maskable; it occurs regardless of the settings of the EXL, ERL, and IE Status register bits. It is implemented only on R4000 processors.

Handling:

The Reset exception vector (0xbfc00000) is used for this exception. This vector is located within the unmapped and uncached address space so that the cache and TLB need not be initialized to handle this exception. The SR bit of the Status register is set to differentiate this exception from Reset.

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

Servicing:

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

5.6.4. TLB Refill and Extended addressing TLB Refill

Cause:

The TLB refill exception occurs when no TLB entry matches a reference to a mapped address space. This exception is not maskable.

Handling:

Two special vectors are provided for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The UX, SX, and KX bits of the Status register determine whether the user, supervisor, or kernel address spaces are 32-bit or 64-bit spaces. For R4000 processors all references use this vector when EXL = 0 in the Status register; for other implementations only references to the user address space (from either kernel or user mode) use this vector and references to the kernel address space use the common exception vector.

The "TLBL" or "TLBS" code in the Cause register is set, indicating whether the instruction, indicated by the EPC register and "BD" bit in the Cause register, caused the miss via an instruction reference or load or alternatively, via a store.

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 Address Space Identifier from which the translation fault occurred. The Random register normally contains a valid location in which to put a replacement TLB entry. The contents of the EntryLo register is undefined.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

To service this exception, the contents of the Context (or XContext) register is used as a virtual address to fetch a memory word containing the physical page frame and access control bits. The memory word is placed into the EntryLo register (EntryLo0/EntryLo1 on the R4000), and the EntryHi and EntryLo registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is also not resident in the TLB. This efficiently handled by allowing a TLB refill exception in the TLB refill handler. This second exception goes instead to the common exception vector because it is a reference to the kernel address space on R2000, R3000, and R6000 processors, and because the EXL bit of the Status register is set for R4000 processors.

5.6.5. TLB Invalid

Cause:

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

Handling:

The common exception vector is used for this exception. The "TLBL" or "TLBS" code in the Cause register is set, indicating whether the instruction, indicated by the EPC register and "BD" bit in the Cause register, caused the miss via an instruction reference or load or alternatively, via a store.

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 Address Space Identifier from which the translation fault occurred. The Random register normally contains a valid location in which to put a replacement TLB entry. The contents of the EntryLo register is undefined.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

The valid bit of a TLB entry is typically cleared when a virtual address does not exist, or when it exists, but is not in main memory (a page fault), or when a trap is desired on any reference to the page (for example to maintain a reference bit). After servicing the particular cause of this exception, the TLB entry is located with TLBP (TLB Probe), and replaced with an entry with the valid bit set.

5.6.6. TLB Modified

Cause:

The TLB modified exception occurs when a store operation's virtual address reference to memory matches a TLB entry which is marked valid but not dirty/writable. This exception is not maskable.

Handling:

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

When this exception occurs, the BadVAddr, Context, and EntryHi registers contain the virtual address that failed address translation. The EntryHi register also contains the Address Space Identifier from which the translation fault occurred. The contents of the EntryLo register is undefined.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

The kernel uses the the failing virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses, and if not permitted, a "Write Protection Violation" has occurred.

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

5.6.7. Bus Error

Cause:

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

This error occurs only for these events when they occur synchronously (cache miss refills, uncached references, and unbuffered writes); a bus error resulting from a buffered write transaction must instead be reported using the general interrupt mechanism.

Handling:

The common interrupt vector is used for this exception. The "IBE" or "DBE" code in the Cause register is set, signifying whether the instruction, indicated by the EPC register and "BD" bit in the Cause register, caused the exception via an instruction reference or alternatively, via a load or store.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

The physical address at which the fault occurred may be computed from information available in the system control coprocessor registers. If the "IBE" code in the Cause register is set (instruction fetch reference), the virtual address is contained in the EPC register. If the "DBE" code set (load or store reference), the instruction which caused the exception is located at the virtual address contained in the EPC register (or four plus the contents of the EPC register if the "BD" bit of the Cause register is set). The virtual address of the load or store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the EntryLo register to compute the physical page number.

The process executing at the time is handed a UNIX SIGBUS (bus error) signal. This error is usually fatal.

5.6.8. Address Error

Cause:

The Address Error exception occurs when an attempt is made to load, fetch or store a word which is not aligned on a word boundary, or to load or store a halfword which is not aligned on a halfword boundary, or to load or store a doubleword which is not aligned on a doubleword boundary, or to reference a kernel address space from user or supervisor mode, or a supervisor address space from user mode. This exception is not maskable.

Handling:

The common exception vector is used for this exception. The "AdEL" or "AdES" code in the Cause register is set, indicating whether the instruction, indicated by the EPC register and "BD" bit in the Cause register, caused the exception via an instruction reference or load or alternatively, via a store.

When this exception occurs, the BadVAddr register contains the virtual address that was not properly aligned or which referenced a protected address space. The contents of the VPN field of the Context and EntryHi registers is undefined. The contents of the EntryLo register is undefined.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

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

5.6.9. Integer overflow

Cause:

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

Handling:

The common exception vector is used for this exception. The "OV" code in the Cause register are set.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

Servicing:

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

5.6.10. Trap (MIPS II and III only)

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.

Handling:

The common exception vector is used for this exception. The "Tr" code in the Cause register are set.

The EPC points at the instruction which caused the exception, unless it is in a branch delay slot. If the instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it, and the BD bit of the Cause register is set.

This exception does not occur on R2000 and R3000 processors, as the instructions that cause this exception are not valid.

Servicing:

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

5.6.11. System Call

Cause:

The system call exception occurs when an attempt is made to execute the SYSCALL instruction. This exception is not maskable.

Handling:

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

The EPC points at the SYSCALL instruction, unless it is in a branch delay slot. If the SYSCALL instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it.

If the SYSCALL instruction is in a branch delay slot, the BD bit of the Status register (SR) is set, otherwise it is cleared.

Servicing:

Control is transferred to the applicable system routine. To resume execution, the EPC must be altered so that the SYSCALL instruction is not re-executed; this is accomplished by adding 4 to the EPC register before returning. Note that if a SYSCALL instruction is in a branch delay slot, a more complicated algorithm would be required.

5.6.12. Breakpoint

Cause:

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

Handling:

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

The EPC points at the BREAK instruction, unless it is in a branch delay slot. If the BREAK instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it.

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

Servicing:

Control is transferred to the applicable system routine. Additional distinctions may be made on the basis of the otherwise unused bits of the BREAK instruction (bits 25..6), by loading the contents of the instruction pointed at by the EPC register. (A value of 4 must be added to the contents of the EPC register to locate the instruction if it resides in a branch delay slot.)

To resume execution, the EPC must be altered so that the BREAK instruction is not re-executed; this is accomplished by adding 4 to the EPC register before returning. Note that if a BREAK instruction is in a branch delay slot, interpretation of the branch instruction would be required in order to resume execution.

5.6.13. Reserved Instruction

Cause:

The Reserved Instruction exception occurs when an attempt is made to execute an instruction whose major opcode (bits 31..26) is undefined or a SPECIAL instruction whose minor opcode (bits 5..0) is undefined. On R6000 and R4000 processors, this exception also occurs on REGIMM instruction whose minor opcode (bits 20..16) is undefined. On MIPS III processors, this exception also occurs on MIPS III opcodes when the processor is in user mode and UX = 0 in the Status register and when the processor is in supervisor mode and SX = 0 in the Status register. This exception is not maskable.

R6000 processors take this exception when an attempt is made to perform an uncached doubleword load. This permits such operations to be interpreted in software, as required.

This exception provides a mechanism to interpret instructions which are added to or removed from the MIPS processor architecture at a later time.

Handling:

The common exception vector is used for this exception. The "RI" code in the Cause register are set.

The EPC points at the reserved instruction, unless it is in a branch delay slot. If the reserved instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it.

Servicing:

In current systems, no instructions in the architecture are interpreted. The process executing at the time is handed a UNIX SIGILL/ILL_RESOP_FAULT (illegal instruction/reserved operand fault) signal. This error is usually fatal.

5.6.14. Coprocessor Unusable

Cause:

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for which the corresponding coprocessor unit has not been marked usable, or for coprocessor zero instructions, when the unit has not been marked usable and the process is executing in user mode. This exception is not maskable.

Handling:

The common exception vector is used for this exception. The "CpU" code in the Cause register is set.

The contents of the Coprocessor Usage Error field of the Coprocessor Control register indicates which of the four coprocessors was referenced.

The EPC points at the unusable coprocessor instruction, unless it is in a branch delay slot. If the unusable coprocessor instruction is in a branch delay slot, the EPC points at the branch instruction which precedes it.

Servicing:

The coprocessor unit to which an attempt was made to reference is identified from the Coprocessor Usage Error field. If the process is entitled to access, the coprocessor is marked usable and the corresponding user state is restored into the coprocessor.

If the process is entitled to access to the coprocessor, but it is known not to exist or to have failed, interpretation of the coprocessor instruction is possible. If the BD bit is set in the Cause register, the branch instruction must be interpreted; then the coprocessor instruction may be emulated and execution resumed with the EPC advanced past the coprocessor instruction.

If the process is not entitled to access to the coprocessor, the process executing at the time is handed a UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

5.6.15. Interrupt

Cause:

The Interrupt exception occurs when one of the eight interrupt conditions are asserted. The significance of these interrupts is implementation-dependent.

Each of the eight interrupts may be masked by clearing the corresponding bit in the IntMask field of the Status register. All of the eight interrupts may be masked at once by clearing the IEc bit of the Status register.

Handling:

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

The IP field of the Cause register indicates the current interrupt requests. It is possible that more than one of the bits will be set at once, or even that no bits are set (if an interrupt is asserted and then deasserted before this register is read).

Servicing:

If the interrupt is caused by one of the two software-generated exceptions (SW_1 or SW_0), the interrupt condition is cleared by setting the corresponding Cause register bit to zero.

If the interrupt is hardware-generated, the interrupt condition is cleared by alleviating the condition which is causing the corresponding interrupt pin to be asserted. The manner in which this is accomplished is implementation-dependent.

5.6.16. Machine Check (R6000 only)

Cause:

The Machine Check exception occurs when a hardware failure, such as a cache parity error occurs which cannot be recovered from completely transparently, i.e. requires software intervention, recovery, or reporting. This exception is not maskable.

Handling:

The common interrupt vector is used for this exception. The "MC" code in the Cause register is set.

The contents of implementation-dependent diagnostic status bits on the status register indicates the precise cause of the exception. It is possible that more than one of the bits may be pending at once.

Servicing:

The machine check condition is cleared by alleviating the condition which is causing the corresponding exception to be asserted. The manner in which this is accomplished is implementation-dependent.

5.6.17. Uncached LDCz/SDCz (R6000 only)

Cause:

The Uncached LDCz/SDCz exception occurs when a doubleword access is made to an uncached address. This exception is not maskable.

Handling:

The common exception vector is used for this exception. The "NCD" code in the Cause register is set.

Servicing:

The exception handler should emulate the doubleword access and resume continue execution.

5.6.18. Virtual Coherency (R4000 only)

Cause:

The Virtual Coherency Exception occurs when a primary cache miss hits in the secondary cache, but $vAddr_{CACHEBITS-1..12}$ was not equal to the corresponding bits of the PIdx field of the secondary cache tag, and the cache algorithm for the page (from the C field in the TLB) specifies the page is cached. This exception is not maskable.

Handling:

The common exception vector is used for this exception. The "VCEI" or "VCED" code in the Cause register is set for Instruction and Data cache misses respectively. The BadVAddr register indicates the virtual address that caused the exception.

Servicing:

The CACHE instruction may be used to determine the old virtual index, to remove the data from the primary caches at the old virtual index, and finally to write the PIdx field of the secondary cache with the new virtual index. At this point the program may be continued. Software can avoid the cost of this trap by using consistent virtual primary cache indexes to access the same physical data.

5.6.19. Cache Error (R4000 only)

Cause:

The Cache Error exception occurs when a secondary cache ECC error, a primary cache parity error, or a R4000 SYSAD bus parity error is detected. This exception is not maskable (but error detection may be disabled by the DE bit of the Status register).

Handling:

The processor sets the ERL bit in the Status register and saves the exception restart address in ErrorEPC, and then transfers to a special vector in uncached space (0xffff_ffff_a000_0100 if BEV = 0, otherwise 0xffff_ffff_bfc0_0300). No other registers are changed.

Servicing:

All errors should be logged. Single-bit ECC errors in the secondary cache can be corrected, using the CACHE instruction, and execution resumed with ERET. Cache parity errors and non-single-bit ECC errors in unmodified cache blocks can be corrected by invalidating the cache block, again using the CACHE instruction, then overwriting the old data via a cache miss, and resuming execution with ERET. Other errors are not correctable, and are likely to be fatal to the current process.

5.6.20. Watch (R4000 only)

Cause:

The Watch exception occurs when a load or store instruction references the physical address specified in the WatchLo/WatchHi system control coprocessor registers. The WatchLo register specifies whether loads, stores, both, or neither initiate this exception. The CACHE instruction never causes a WATCH exception. The exception is postponed while the EXL bit is set in the Status register. This exception is maskable only by setting EXL in the Status register.

Handling:

The common exception vector is used for this exception. The "WATCH" code in the Cause register is set.

Servicing:

This exception is intended as a debugging aid. Typically the exception handler will transfer control to a debugger, allowing the user to examine the situation. To continue, the Watch must be disabled for the execution of the faulting instruction and then re-enabled. Execution of the faulting instruction may be accomplished by interpretation, or by setting breakpoints.

5.6.21. Floating Point (R4000 only)

Cause:

The Floating Point Exception is used by the R4000 floating point coprocessor. Other implementations use one of the hardware interrupts for this exception. This exception is not maskable.

Handling:

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

The contents of the Floating Point Control Status register indicates the cause of this exception.

Servicing:

This exception is cleared by clearing the appropriate bit in the Floating Point Control Status register. For an unimplemented exception, the kernel should emulate the instruction. For other exceptions, the kernel should pass the exception to the user.

6. Hazards and Interlocks

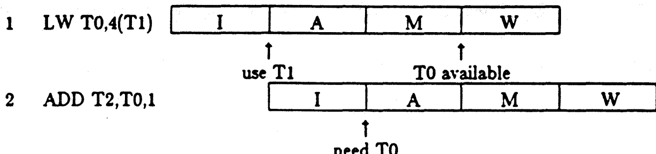
Certain combinations of instructions are not permitted, because the results of executing such combinations are unpredictable, in the face of events such as pipeline delays, cache misses, interrupts and exceptions. These constraints are noted in each of the relevant instructions, and is repeated here, collected together, for additional clarity.

The R-series architecture allows implementations to expose a few predefined hazards. These conditions need not be detected and corrected in hardware; instead software is responsible for avoiding the hazard. Future implementations will be upward-compatible and introduce no additional hazards in the CPU and floating-point coprocessor. Future implementations of coprocessor 0 may be different.

Most hazards result from instructions modifying and reading state in different pipeline stages. Such hazards are defined between pairs of instructions, not on a single instruction in isolation. Other hazards are associated with restartability of instructions in the presence of exceptions.

6.1. Introduction to pipeline hazards

Pipelining is an implementation technique that allows multiple instructions to be in various stages of execution simultaneously. Pipelining is key to high performance, but we need to consider what happens when a result is needed by an instruction in one pipe stage before it has been produced by an earlier instruction in a later pipe stage. For example, consider an idealized pipeline consisting of instruction cache, alu, data cache, and register write stages:



Here instruction 2 is trying to reference instruction 1's result before it is available. This occurs because instruction 1 takes more time to execute than it takes to fetch the next instruction (instruction 2).

There are several ways to solve this problem. One is to detect the situation in hardware and insert null cycles to separate the two instructions in the pipeline so that operands are available when they are needed. This is called "interlocking". The other is to do the same thing in software, using other instructions (a NOP if nothing else will do) to separate the instructions. In this case if software fails to separate the instructions the sequence executes incorrectly. We say there is a "hazard" that software must avoid.

The R-series architecture uses a combination of hardware interlocking and software hazard-avoidance to ensure programs execute correctly despite pipelining. In general, hardware interlocks are used for long latency operations (potentially many NOPs required) such as integer multiply/divide and floating point results, and software hazard-avoidance for low-latency operations such as the result of load instructions.

A common question about the R-series architecture is "How many NOPs are needed after instruction I?" This question is meaningless because pipeline hazards are defined between a pair of instructions, not by a single instruction. One way to represent such data would be a giant N×N matrix giving the number of hazard cycles, where N is the number of instructions. A better way requires a table with only N rows giving the pipeline stage where operands are used and results produced. The required distance between instruction α that produces a result used by instruction β is the stage number α produces a result minus the stage number β uses the result. For example a simple add instruction gets its operands on input to stage 2 and produces a result by its stage 3. The distance required between two dependent adds is then 3−2=1 instruction, which is the minimum spacing, and so there is no hazard. Between a load (result available by its stage 4) and a dependent add the required distance is 4−2=2 instructions. There is a hazard here which must be avoided by separating the load and add by one instruction, which leaves them two apart:

```
100:  LW    T0,4(T1)
104:  NOP
108:  ADD   T2,T0,1
```

Conversely, between an add and a load, the required distance is 3−2=1 instruction, so the following works:

```
200:  ADD   T1,T1,4
204:  LW    T0,0(T1)
```

The table below gives the pipeline stages for operands and results in the R2000/R3000 R-series architecture. Future implementations will be upward-compatible, introducing hardware interlocks if necessary.

To illustrate how important it is to consider a pair of instructions, consider the CTC1 instruction, which moves the contents of general-purpose register to the floating point control/status register (available in stage 4). The floating point status register controls rounding, exceptions, etc, so the required distance between CTC1 and a floating point (read in stage 2) op is 4−2=2 instructions:

```
300:  CTC1    T0, $31
304:  NOP
308:  ADD.D   F0, F2, F4
```

The floating point control/status register also contains the floating point compare result. This compare result is used by the BC1T and BC1F instructions, and it is used in stage 1, so the required distance is 4−1=3 instructions:

```
400:  CTC1   T0, $31
404:  NOP
408:  NOP
40C:  BC1T   L
```

Interlocks and stalls cannot be used to eliminate a hazard. For example,

```
500:  MUL.S   F0, F2, F4
504:  LWC1    F6, 0(T0)
508:  ADD.S   F0, F0, F6
```

the processor will stall on the ADD.S to wait for F0 to be computed by the MUL.S. This stall does not eliminate the LWC1/ADD.S hazard, and so the above code is in error.

6.2. Introduction to restartability hazards

The other class of hazards arise from the need to be able to transparently take an interrupt or exception between any pair of instructions. Most R-series implementations delay changing stored program state until a pipeline stage after all exceptions and interrupts are detected and effected (typically bypassing is used so that the results may be used before that time). When an instruction is aborted by an exception or interrupt, all subsequent instructions in the pipeline are also aborted; because these instructions have not yet affected stored program state, the exception handler sees the same stored program state as if there were no pipelining.

If an implementation updates program state in an early pipeline stage, then an exception detected in a later pipeline stage of a previously fetched instruction would not abort the update. In this case the effect of an aborted instruction would be part of the stored program state. If execution is resumed, the difference in program state may affect program operation, and thus the exception or interrupt was not transparent.

For example, R2000 and R3000 MULT, MULTU, DIV, and DIVU write the HI and LO registers in the A stage of the pipeline, instead of waiting until the W stage, where most instructions commit results. Consider

the following instruction sequence:

```
600:   MFLO    R5
604:   NOP
608:   MULTU   R6, R7
```

When no interrupt occurs during this sequence, the MFLO reads LO during its A-stage, and two cycles later the MULTU writes a new value into LO. Thus R5 receives the previous multiply/divide result. However, if an exception occurs such that MFLO is aborted in the W stage, the MULTU, which is in the A stage still completes and writes HI and LO. When the MFLO is restarted, it reads the new multiply result. The R-series architecture makes software responsible for avoiding this hazard by requiring at least two instructions between a read of HI or LO and an instruction that writes HI or LO.

Restartability also imposes a restriction on JALR. While R2000 and R3000 processors do wait delay writing the destination GPR until the W stage, an exception or interrupt in their branch delay slot sets the EPC such that these instructions will be re-executed. In order that this re-execution be transparent, the source register of JALR must be different from its destination register.

6.3. Hazards allowed by the R-series architecture

This section enumerates hazards allowed by implementations of the R-series architecture. The operation of programs that do not avoid these hazards is undefined. Coprocessor 0 hazards are implementation specific, and are addressed in a later section.

6.3.1. Load delay slot

All processor load instructions: *LBU, LB, LHU, LH, LW, LWL, LWR;* and move-from-coprocessor instructions: *MFCz, CFCz;* modify processor general registers in a late pipe stage, preventing their use as source register operands in the immediately following instruction.

In the same way, coprocessor load instructions: *LWCz, LDCz;* and move-to-coprocessor instructions: *MTCz, CTCz;* modify coprocessor general registers or coprocessor control registers in a late pipe stage, and cannot be used as source register operands in the immediately following instruction.

MIPS II processors, while for timing purposes have delayed load instructions, need not have the delay slots of processor and coprocessor load instructions filled with a NOP, as the hardware will interlock to provide the updated value of the processor or coprocessor target register to the next instruction. All move-to-and-from coprocessor instructions: *MFCz, CFCz, MTCz, CTCz* will still require a scheduled delay slot to be filled with a NOP or other instruction that does not use the target register.

6.3.2. Branch delay slot

Most R-series processors have a single program counter, yet have delayed branches. When an exception occurs that would normally require setting the EPC to an instruction in a branch delay slot, the EPC is instead backed up to the immediately preceding branch instruction. Thus, the branch instruction is executed a second time, upon return from the exception handler, which leads to the following constraint:

JALR cannot use a source register the same as the destination register.

If a branch instruction could itself be placed in a branch delay slot, no placement of the EPC could assure a properly restartable instruction flow. In addition, PC-relative branch operations are relative to the address of the instruction in the branch delay slot, and in order to be well-defined, this must be the instruction immediately following the branch. Each of these are sufficient reasons for the constraint:

The instruction in the delay slot of a PC changing instruction, *J, JR, JAL, JALR, BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL,* and on implementations other than the R2000 and R3000, *BEQL, BNEL, BLEZL, BGTZL, BLTZL, BGEZL, BLTZALL, BGEZALL,* cannot itself change the PC.

6.3.3. Setting up a coprocessor condition

A BCzT or BCzF instruction samples the coprocessor condition line during the instruction immediately preceding it. Thus, the coprocessor condition should not be changed immediately preceding the BCzT or BCzF instruction.

This constraint requires that the instruction executed immediately after a coprocessor condition setting (e.g. C.cond.fmt for coprocessor 1) cannot be BCzT or BCzF. If the condition setting occurs as a result of a MTCz or CTCz instruction, two other instructions must occur between this instruction and the BCzT or BCzF instruction that tests it.

6.3.4. No bypassing for HI and LO registers

To reduce the amount of bypassing that is required for the HI and LO registers, which contain the results of multiply and divide operations, it is required that the two instructions that follow an attempt to read the registers with either of the *MFHI, MFLO* instructions, must not modify the register being read. This permits the instructions that modify the HI and LO registers: *MULT, MULTU, DIV, DIVU, MTHI, MTLO;* to start modifying the registers as soon as it is verified that the instruction is valid, even if it is later determined that an exception will cause the instruction to be canceled for later re-execution. This leads to the constraints:

The two instructions executed after MFLO cannot be MTLO, MULT, MULTU, DIV, DIVU. The two instructions executed after MFHI cannot be MTHI, MULT, MULTU, DIV, DIVU.

6.3.5. Combinations of scheduling hazards

In some cases, scheduling hazards connect together to produce longer constraints. For example, the sequence: *CTC1 $rx,$31; BC1T label;* requires two NOPs between the instructions.

6.3.6. Additional requirements for R2360 floating-point board

The floating-point status register in the R2360 floating-point board is not fully bypassed, requiring the following additional constraints: (1) All attempts to read the floating-point status register must be performed on an "empty" pipeline. The MIPS assembler provides this constraint by duplicating all CFC1 instructions that read the floating-point status register. (2) Three NOPs are required between a CTC1 instruction that alters the contents of the floating-point condition and a subsequent BC1T or BC1F instruction. The MIPS assembler observes this constraint.

Exception handling in the R2360 floating-point board is imprecise. A floating-point exception may occur at any time after a floating-point operation is started, until the next floating-point operation, load, store, or move instruction occurs. If the location of the exception is important (as when using statically-defined exception handlers), additional instructions must be inserted when crossing such statically-defined boundaries, to empty the floating-point pipeline and report exceptions within the proper region of the program. Current MIPS compilers do not provide this facility.

6.4. R2000 and R3000 Pipeline

R2000 and R3000 processors internally use a 5-stage pipeline with each cycle further divided in halves named $\phi1$ and $\phi2$.

I1	I		R		A		M		W					
	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$				
I2			I		R		A		M		W			
			$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$		
I3					I		R		A		M			
					$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$	$\phi1$	$\phi2$		

where:

I $\phi1$	0	Use micro-TLB to translate instruction virtual address to physical (after branch decision in A $\phi1$).
I $\phi2$		Send physical address to instruction cache.
R $\phi1$	1	Instruction returns from instruction cache. Compare tags, check parity.
R $\phi2$		Read register file. Branch: calculate branch target address. Latch coprocessor condition input.
A $\phi1+\phi2$	2	Bypass operands from other pipeline stages. Calculate add, logical, shift, etc. results. Shift store data. Start integer multiply/divide, or floating point.
A $\phi1$	2	Branch: decide whether branch taken or not. Load or store: calculate virtual address.
A $\phi2$		Load or store: translate virtual address to physical using TLB.
M $\phi1$	3	Load or store: Send physical address to data cache.
M $\phi2$		Load or store: data returns from data cache. Compare tags, check parity, extract byte for loads. MTCz or MFCz: transfer data to or from coprocessor.
W $\phi1$	4	Write register file.

R2000 and R3000 Pipeline Stages				
instruction	source			destination
	rs/fs	rt/ft	other	
alu, shift	2	2		3
branch	2	2		
BLTZAL, BGEZAL	2			3
JAL				3
JR	2			
JALR	2			3
MFHI, MFLO			2	3
MTHI, MTLO	2			3
MULT, MULTU	2	2		14β
DIV, DIVU	2	2		37β
LB, LBU, LH, LHU, LW, LWCz	2			4α
MFCz, CFCz	2			4α
MTCz, CTCz		2		4α
BCzT, BCzF			1	
LWL, LWR	2	3		4α
SB, SH, SW, SWL, SWR, SWCz	2	2		
ADD.S, SUB.S, ADD.D, SUB.D	2	2		4β
MUL.S	2	2		6β
MUL.D	2	2		7β
DIV.S	2	2		14β
DIV.D	2	2		21β
MOV.S, MOV.D, NEG.S, NEG.D, ABS.S, ABS.D	2			3
CVT.D.S	2			3
CVT.W.S, CVT.W.D, CVT.S.D	2			4β
CVT.S.W, CVT.D.W	2			5β
C.xx.S, C.xx.D	2	2		3α

α Possible hazard on this result.

β Possible interlock on this result.

6.5. R6000 Pipeline

The R6000 pipeline is as follows:

I1	I	R/A	D	N	W		
I2		I	R/A	D	N	W	
I3			I	R/A	D	N	W

where:

I	0	Fetch instruction from instruction cache.
R/A	1	Read register file, bypass results from other stages, use ALU to calculate results or virtual address for loads and stores.
D	2	Load or store: Read or write primary data cache.
N	3	Detect primary cache misses and resolve exceptions. Load: Send data from data cache to processor.
W	4	Write register file.

R6000 Pipeline stages				
instruction	source			destination
	rs/fs	rt/ft	other	
alu	2	2		3
SLL, SRL, SRA		2		3
SLLV, SRLV, SRAV	2	2		3
branch, trap	2	2		
BLTZAL, BGEZAL	2			3
JAL				3
JR	2			
JALR	2			3
MFHI, MFLO			2	3
MTHI, MTLO	2			3
MULT	2	2		19β
MULTU	2	2		20β
DIV	2	2		40β
DIVU	2	2		39β
LB, LBU, LH, LHU, LW, LL, LWCz	2			4β
LDCz	2			4-5β
MFCz, CFCz	2			4α
MTCz, CTCz		2		4α
BCzT, BCzF			1	
LWL, LWR	2	3		4β
SB, SH, SW, SWL, SWR, SWCz	2	2		
SDCz	2	2-3		
SC	2	2		4β
ADD.S, SUB.S	2	2		5β
ADD.D, SUB.D	2-3	2-3		5-6β
MUL.S	2	2		6-7β
MUL.D	2-3	2-3		7-8β
DIV.S	2	2		14-17β
DIV.D	2-3	2-3		22-28β
MOV.S, NEG.S, ABS.S	2			4β
MOV.D, NEG.D, ABS.D	2-3			4-5β
CVT.D.S	2			4-5β
CVT.W.S, ROUND.W.S, TRUNC.W.S, FLOOR.W.S, CEIL.W.S, CVT.S.W, CVT.D.W	2			5β
CVT.W.D, ROUND.W.D, TRUNC.W.D, FLOOR.W.D, CEIL.W.D, CVT.S.D	2-3			5-6β
C.xx.S	2	2		3α
C.xx.D	2-3	2-3		3-4α

α Possible hazard on this result.

β Possible interlock on this result.

6.6. R4000 Pipeline

The R4000 pipeline is as follows:

I1	IF	IS	RF	EX	DF	DS	TC	WB		
I2		IF	IS	RF	EX	DF	DS	TC	WB	
I3			IF	IS	RF	EX	DF	DS	TC	WB

where:

IF	0	First half of instruction cache access.
IS	1	Second half of instruction cache access.
RF	2	Register file read, and tag check of instruction fetch.
EX	3	Calculate add, logical, shift, branch, etc. results, and virtual address for loads and stores.
DF	4	First half of data cache access. First half of TLB access.
DS	5	Second half of data cache access. Second half of TLB access.
TC	6	Tag check of data cache access.
WB	7	Register file write.

R4000 Pipeline stages

instruction	source			destination
	rs/fs	rt/ft	other	
alu	3	3		4
SLL, SRL, SRA		3		4
SLLV, SRLV, SRAV	3	3		5
branch, trap	3	3		
BLTZAL, BGEZAL	3			4
JAL				4
JR	3			
JALR	3			4
MFHI, MFLO			3	4
MTHI, MTLO	3			4
MULT, MULTU	3	3		15β
DIV, DIVU	3	3		79β
LB, LBU, LH, LHU, LW, LL, LWCz, LDCz	3			6β
MFCz, CFCz	3			6α
MTCz, CTCz		3		6α
BCzT, BCzF			3	
LWL, LWR	3	5		6β
SB, SH, SW, SWL, SWR, SWCz, SDCz	3	3		
SC	3	3		6β
ADD.S, SUB.S, ADD.D, SUB.D	3	3		7β
MUL.S	3	3		10β
MUL.D	3	3		11β
DIV.S	3	3		26β
DIV.D	3	3		39β
MOV.S, MOV.D	3			4
NEG.S, NEG.D, ABS.S, ABS.D	3			5β
CVT.D.S	3			5β
CVT.W.S, CVT.W.D, ROUND.W.S, ROUND.W.D, TRUNC.W.S, TRUNC.W.D, FLOOR.W.S, FLOOR.W.D, CEIL.W.S, CEIL.W.D	3			7β
CVT.S.D	3			7β
CVT.S.W	3			9β
CVT.D.W	3			8β
C.xx.S, C.xx.D	3	3		6β

α Possible hazard on this result.

β Possible interlock on this result.

6.7. Coprocessor 0 Hazards

Many coprocessor 0 registers contain data that affects instruction fetch, instruction execution, address translation, exceptions, interrupts, etc. Therefore the hazards associated with MTC0 instructions are complex and depend on which fields are being modified by the write.

6.7.1. R3000 coprocessor 0 Hazards

Most coprocessor 0 registers contain data that has side-effects on the behavior of following instructions. Such side-effects are not predictable on the one, two, or three instructions immediately following a *MTC0* instruction that modifies that data.

In particular, following a MTC0 instruction that turns on the usability of a coprocessor, the next two instructions must not depend on that coprocessor being usable, that is, they must not be instructions for that coprocessor unit. Similarly, following a MTC0 instruction that turns off the usability of a coprocessor, the next two instructions must not depend on that coprocessor being *unusable*.

Also, a MTC0 that enables or disables interrupts does not take effect until the third following instruction.

For R2000 and R3000 processors, the coprocessor 0 instructions TLBWI and TLBWR write the TLB in the same pipeline stage as loads and stores read it, so there are no hazards between these instructions and loads and stores. However, instruction fetches do access the micro-TLB and TLB in a different pipe stage, causing a delay of two instructions before the map change is effected. Also, the micro-TLB is not flushed by TLBWI and TLBWR (it is flushed by loading EntryHi), which can further delay the effect. For these reasons, it is recommended that the move to EntryHi and TLB write be executed from unmapped space.

6.7.2. R6000 memory management hazards

The R6000 memory management instructions (LWL/SWL/LWR/SWR with the MM Status register bit set) cannot be placed in a branch or jump delay slot.

6.7.3. R4000 coprocessor 0 hazards

operation	source		destination	
	name	stage	name	stage
MTC0	gpr rt	3	cpr rd	7
MFC0	cpr rd	4	gpr rt	7
TLBR	Index, TLB	5-7	PageMask, EntryHi, EntryLo0, EntryLo1	8
TLBWI TLBWR	Index or Random, PageMask, EntryHi, EntryLo0, EntryLo1	5-7	TLB	8
TLBP	PageMask, EntryHi	3-6	Index	7
ERET	EPC or ErrorEPC, Status, TLB	4	Status.EXL, Status.ERL	4-8$^{\alpha}$
			LLbit	7
CACHE Index Load Tag	-		TagLo, TagHi, ECC	8$^{\beta}$
CACHE Index Store Tag	TagLo, TagHi, ECC	7	-	
CACHE Hit ops	-		Status.CH	8
instruction fetch	EntryHi.ASID, Status.KSU, Status.EXL, Status.ERL, Status.RE, Config.K0C, Config.IB	0		
	Config.SB	3		
	TLB	2		
instruction fetch exception	-		EPC, Status	8
			Cause, BadVAddr, Context	3
coprocessor usable test	Status.CU, Status.KSU, Status.EXL, Status.ERL	2		
interrupt	Cause.IP, Status.IM, Status.IE, Status.EXL, Status.ERL	3		
load/store	EntryHi.ASID, Status.KSU, Status.EXL, Status.ERL, Status.RE, Config.K0C, Config.DB, TLB	4		
	Config.SB	7		
	WatchHi, WatchLo	4-5		
load/store exception	-		EPC, Status, Cause, BadVAddr, Context	8
TLB shutdown	-		Status.TS	7

α Status.EXL and Status.ERL are permanently cleared in stage 8, but the effect of clearing them is visible to instruction fetch starting in stage 4.

β Only one instruction needs to separate Index Load Tag and MFC0 Tag, even though the above would imply three instructions.

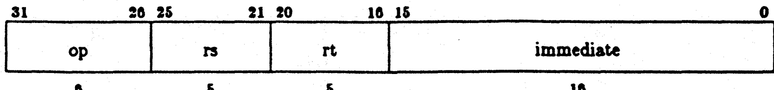
The instruction following a MTC0 must not be a MFC0.

The five instructions following a MTC0 to Status that changes KSU and sets EXL or ERL may be executed in the new mode, and not kernel mode. This can be avoided by setting EXL first, leaving KSU set to kernel, and later changing KSU.

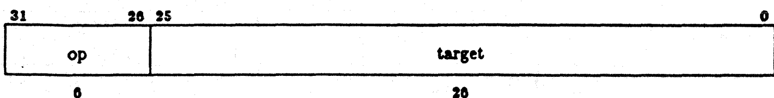
There must two non-load and non-CACHE instructions between a store and a CACHE instruction directed to the same primary cache line as the store.

7. Instruction Encoding

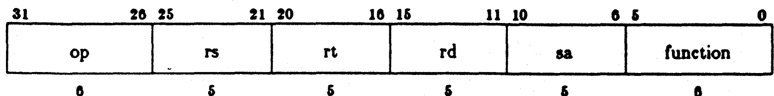
I-type (Immediate):



J-type (Jump):



R-type (Register):



where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier or function code
rt	is a 5-bit source/destination register specifier or function code
immediate	is a 16-bit immediate, branch displacement or address displacement
target	is a 26-bit jump target address
rd	is a 5-bit destination register specifier
sa	is a 5-bit shift amount
function	is a 6-bit function code

† Operation codes marked with a dagger cause reserved instruction exceptions in all current implementations and are reserved for future versions of the architecture.

α Operation codes marked with an alpha cause reserved instruction exceptions in MIPS I implementations, and are valid for MIPS II implementations.

β Operation codes marked with a beta are not valid for MIPS I implementations, and are valid for MIPS II implementations. MIPS I implementations do not take a reserved instruction exception on these opcodes.

γ Operation codes marked with a gamma are not valid for MIPS I implementations, and for MIPS II implementations 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 R4000 processors with coprocessor 0 enabled, and cause a reserved instruction exception on other processors.

ξ Operation codes marked with a xi are not valid on R4000 processors.

χ Operation codes marked with a chi are valid only on R4000 processors.

7.1. MIPS I and MIPS II opcode assignments

op

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	BEQLα	BNELα	BLEZLα	BGTZLα
3	†	†	†	†	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	†
5	SB	SH	SWL	SW	†	†	SWR	CACHEδ
6	LL	LWC1	LWC2	LWC3	†	LDC1α	LDC2α	LDC3α
7	SC	SWC1	SWC2	SWC3	†	SDC1α	SDC2α	SDC3α

SPECIAL function

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSCALL	BREAK	†	SYNCα
2	MFHI	MTHI	MFLO	MTLO	†	†	†	†
3	MULT	MULTU	DIV	DIVU	†	†	†	†
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	†	†	†	†
6	TGEα	TGEUα	TLTα	TLTUα	TEQα	†	TNEα	†
7	†	†	†	†	†	†	†	†

REGIMM rt

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZLβ	BGEZLβ	γ	γ	γ	γ
1	TGEIβ	TGEIUβ	TLTIβ	TLTIUβ	TEQIβ	γ	TNEIβ	γ
2	BLTZAL	BGEZAL	BLTZALLβ	BGEZALLβ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

COPz rs

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	γ	CF	γ	MT	γ	CT	γ
1	BC	†	†	†	†	†	†	†
2	CO							
3								

COPz rt

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFLβ	BCTLβ	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

COP0 function

5..3 \ 2..0	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFEξ							
3	ERETχ							
4								
5								
6								
7								

7.2. MIPS III opcode assignments

ε Operation codes marked with a epsilon are valid only for MIPS III implementations. These opcodes cause a reserved instruction exception on MIPS I and II implementations.

op (rows: bits 31..29; columns: bits 28..26)

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	†	BEQLα	BNELα	BLEZLα	BGTZLα
3	DADDIε	DADDIUε	LDL	LDR	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDL	SDR	SWR	CACHEδ
6	LL	LWC1	LWC2	†	LLDε	LDC1α	LDC2α	LDε
7	SC	SWC1	SWC2	†	SCDε	SDC1α	SDC2α	SDε

SPECIAL function (rows: bits 5..3; columns: bits 2..0)

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSCALL	BREAK	†	SYNCα
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	†	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGEα	TGEUα	TLTα	TLTUα	TEQα	†	TNEα	†
7	DSLLε	†	DSRLε	DSRAε	DSLL32ε	†	DSRL32ε	DSRA32ε

REGIMM rt (rows: bits 20..19; columns: bits 18..16)

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZLβ	BGEZLβ	γ	γ	γ	γ
1	TGEIβ	TGEIUβ	TLTIβ	TLTIUβ	TEQIβ	γ	TNEIβ	γ
2	BLTZAL	BGEZAL	BLTZALLβ	BGEZALLβ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

COPz rs (rows: bits 25..24; columns: bits 23..21)

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	DMFε	CF	γ	MT	DMTε	CT	γ
1	BC	†	†	†	†	†	†	†
2	CO							
3								

20..19 \ 18..16 (COPz rt)	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFLβ	BCTLβ	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

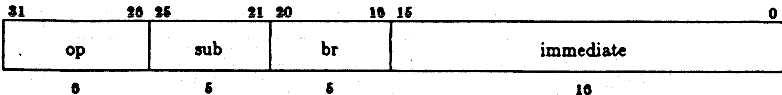
5..3 \ 2..0 (COP0 function)	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFEξ							
3	ERETχ							
4								
5								
6								
7								

8. Floating Point Instruction Encoding

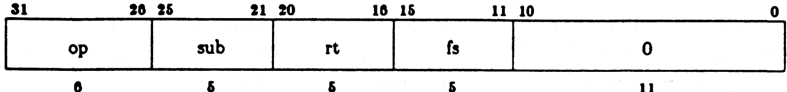
I-type (Immediate):

31 — 26	25 — 21	20 — 16	15 — 0
op	base	ft	immediate
6	5	5	16

B-type (Branch):



M-type (Move):



R-type (Register):

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
op	sub	ft	fs	fd	function
6	5	5	5	5	6

where:

op	is a 6-bit operation code
sub	is a 5-bit sub-operation code
br	is a 5-bit branch code
rt	is a 5-bit source/destination general register specifier
ft	is a 5-bit source/destination float register specifier
fs	is a 5-bit source register specifier
fd	is a 5-bit destination register specifier
immediate	is a 16-bit address/branch displacement
function	is a 6-bit function code
0	undefined if non-zero

† Operation codes marked with a dagger cause reserved or unimplemented operation exceptions in all current implementations and are reserved for future versions of the architecture.

α Operation codes marked with an alpha cause reserved or unimplemented operation exceptions in MIPS I implementations, and are valid for MIPS II implementations.

β Operation codes marked with a beta are not valid for MIPS I implementations, and are valid for MIPS II implementations.

γ Operation codes marked with a gamma are not valid for MIPS I implementations, and for MIPS II implementations cause a reserved instruction exception. They are reserved for future versions of the architecture.

8.1. MIPS I and MIPS II opcode assignments

op

31..29 \ 28..26	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1α		
7		SWC1				SDC1α		

sub

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	γ	CF	γ	MT	γ	CT	γ
1	BC	†	†	†	†	†	†	†
2	S	D	E†	Q†	W	†	†	†
3	†	†	†	†	†	†	†	†

br

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFLβ	BCTLβ	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

function

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRTα	ABS	MOV	NEG
1	†	†	†	†	ROUND.Wα	TRUNC.Wα	CEIL.Wα	FLOOR.Wα
2	†	†	†	†	†	†	†	†
3	†	†	†	†	†	†	†	†
4	CVT.S	CVT.D	CVT.E†	CVT.Q†	CVT.W	†	†	†
5	†	†	†	†	†	†	†	†
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

8.2. MIPS III opcode assignments

δ Operation codes marked with a delta are valid only for MIPS III implementations. These opcodes cause a reserved instruction exception on MIPS I and II implementations.

op (28..26 across, 31..29 down)

31..29 \ 28..26	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1α		
7		SWC1				SDC1α		

sub (23..21 across, 25..24 down)

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	DMFδ	CF	γ	MT	DMTδ	CT	γ
1	BC	†	†	†	†	†	†	†
2	S	D	E†	Q†	W	Lδ	†	†
3	†	†	†	†	†	†	†	†

br (18..16 across, 20..19 down)

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFLβ	BCTLβ	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

function (2..0 across, 5..3 down)

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRTα	ABS	MOV	NEG
1	ROUND.Lδ	TRUNC.Lδ	CEIL.Lδ	FLOOR.Lδ	ROUND.Wα	TRUNC.Wα	CEIL.Wα	FLOOR.Wα
2	†	†	†	†	†	†	†	†
3	†	†	†	†	†	†	†	†
4	CVT.S	CVT.D	CVT.E†	CVT.Q†	CVT.W	CVT.Lδ	†	†
5	†	†	†	†	†	†	†	†
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

Index

A

Access type ... 1-2-5, 1-3-24
Accuracy ... 1-3-2
Add, arithmetic ... 1-2-44—1-2-48, 1-2-55—1-2-57, 1-2-60—1-2-61
Address Error exception ... 1-2-5, 1-2-8, 1-2-12, 1-2-16, 1-2-25, 1-2-27, 1-2-30, 1-2-33, 1-2-35, 1-2-41, 1-2-43, 1-2-142, 1-2-145, 1-3-25, 1-3-27, 1-3-29—1-3-30, 1-4-3, 1-4-8, 1-4-11, 1-4-13, 1-5-10
Address translation ... 1-2-4
And, logical ... 1-2-3, 1-2-44, 1-2-51, 1-2-55, 1-2-60

B

BadVAddr system coprocessor register ... 1-4-38, 1-5-8, 1-5-10, 1-5-14
BigEndianMem ... 1-2-3
Branch ... 1-2-6, 1-2-99, 1-2-105, 1-2-107, 1-2-112, 1-2-115, 1-2-120, 1-2-122, 1-2-153, 1-2-157, 1-3-67, 1-3-71
Branch and link ... 1-2-105, 1-2-113—1-2-114
Branch and link likely ... 1-2-121—1-2-122
Branch delay slot ... 1-2-99, 1-4-46, 1-4-49—1-4-50, 1-4-79—1-5-1, 1-6-3
Branch likely ... 1-2-115—1-2-120, 1-2-156—1-2-157, 1-3-70
Break ... 1-2-126
Breakpoint ... 1-2-126
Breakpoint exception ... 1-5-12
Bus Error exception ... 1-5-3, 1-5-10

C

Cache ... 1-1-5, 1-2-3, 1-2-25, 1-2-27, 1-2-41, 1-2-43 1-4-18—1-4-21, 1-4-23—1-4-24, 1-4-26, 1-4-29, 1-5-6
Cache Error exception ... 1-5-3, 1-5-15
Cache, errors ... 1-4-51—1-4-52, 1-4-54, 1-5-3, 1-5-14—1-5-15
Cache, flushing ... 1-4-77
Cache, instruction ... 1-4-71
Cache, invalidating ... 1-4-78
Cache, load ... 1-4-79
Cache, parameters ... 1-4-57
Cache, store ... 1-4-80
Cache, virtual index ... 1-4-30, 1-5-14
Cache, virtual tag ... 1-4-30
CacheErr system coprocessor register ... 1-4-51
Caches, enabling ... 1-4-4, 1-4-6, 1-4-8, 1-4-11, 1-4-13, 1-4-35
Caches, isolating ... 1-4-43
Caches, parity ... 1-4-43
Caches, swapping ... 1-4-43
Cause, floating-point exception ... 1-3-12
Cause system coprocessor register ... 1-1-13, 1-3-15, 1-4-46, 1-4-55, 1-5-2, 1-5-13
CCR, coprocessor control register ... 1-2-3
COC, coprocessor condition ... 1-2-3
Coherency ... 1-1-5, 1-2-25, 1-2-41, 1-2-43, 1-4-3, 1-4-6, 1-4-8, 1-4-11, 1-4-13, 1-4-21, 1-4-30, 1-4-35, 1-5-14
Compare system coprocessor register ... 1-4-55
Comparisons, floating-point ... 1-3-2
Comprisons, integer ... 1-2-49—1-2-50, 1-2-64—1-2-65, 1-2-107—1-2-122, 1-2-128—1-2-129, 1-2-131—1-2-139
Compatibility ... 1-3-1
Computational instructions ... 1-2-6, 1-2-44, 1-3-40
Config system coprocessor register ... 1-4-57
Context system coprocessor register ... 1-4-38
Control registers, coprocessor ... 1-3-3
Coprocessor ... 1-1-1, 1-2-6, 1-2-140, 1-2-142—1-2-157, 1-4-60—1-4-63
Coprocessor Unusable exception ... 1-5-13
Count system coprocessor register ... 1-4-55
CPR, coprocessor control register ... 1-2-3
CPR, coprocessor general register ... 1-2-3
CPR, coprocessor register ... 1-3-20

D

Data format ... 1-1-1, 1-3-7, 1-3-10
Directed rounding ... 1-3-13
Div, integer division ... 1-2-3
Divide ... 1-2-86, 1-2-89—1-2-90, 1-2-93—1-2-94

E

ECC System coprocessor register ... 1-4-54
Enables, floating-point exception ... 1-3-4, 1-3-12
Endianess ... 1-1-1, 1-2-5, 1-3-4, 1-3-6, 1-3-24, 1-4-40, 1-4-58
EntryHi system coprocessor register ... 1-4-33
EntryLo system coprocessor register ... 1-4-34
EPC system coprocessor register ... 1-4-49
Error system coprocessor register ... 1-4-48
Error EPC system coprocessor register ... 1-4-50

Exception, Address Error ... 1-2-5, 1-2-8, 1-2-12—1-2-16, 1-2-25, 1-2-27, 1-2-30—1-2-33, 1-2-35, 1-2-41, 1-2-43, 1-2-142—1-2-145, 1-3-25, 1-3-27, 1-3-29—1-3-30, 1-4-3, 1-4-8—1-4-11, 1-4-13, 1-5-10
Exception, Breakpoint ... 1-5-12
Exception, Bus Error ... 1-5-3, 1-5-10
Exception, Cache Error ... 1-5-3, 1-5-15
Exception, Coprocessor Unusable ... 1-3-77, 1-4-40
Exception, Floating Point ... 1-5-15
Exception, instruction register ... 1-3-11
Exception, Integer Overflow ... 1-2-45, 1-2-47, 1-2-56, 1-2-58, 1-2-60, 1-2-62, 1-5-11
Exception, Interrupt ... 1-5-13
Exception, Machine Check ... 1-5-14
Exception, NMI ... 1-5-7
Exception priority ... 1-5-6
Exception, Reserved Instruction ... 1-4-1, 1-5-12
Exception, Reset ... 1-5-6
Exception, Soft Reset ... 1-5-7
Exception, System Call ... 1-5-11
Exception, TLB Invalid ... 1-5-9
Exception, TLB Modified ... 1-5-9
Exception, TLB Refill ... 1-5-8
Exception, Trap ... 1-5-11
Exception types ... 1-5-4
Exception, Uncached LDCz/SDCz ... 1-5-14
Exception Vectors ... 1-5-5
Exception, Virtual Coherency ... 1-5-14
Exception, Watch ... 1-5-15
Exclusive Or, logical ... 1-2-3, 1-2-44, 1-2-53, 1-2-55, 1-2-68

F
Fetch ... 1-2-6
Flags, floating-point exception ... 1-3-4, 1-3-12, 1-3-79
Floating Point exception ... 1-5-15
Floating-point control/status register ... 1-3-12
Format, floating-point ... 1-3-7
Format, instruction 1-2-1, 1-3-1
Format, mixed ... 1-3-2
Format, operand ... 1-3-2
Format, unnormalized ... 1-3-7

G
General register ... 1-1-4
General register, coprocessor ... 1-3-3
GPR, general purpose register ... 1-2-3, 1-3-20

H
Hazards ... 1-2-95, 1-2-96, 1-2-99, 1-5-3, 1-6-1
HI, multiply/divide register ... 1-2-3, 1-2-86, 1-2-95, 1-2-97, 1-6-4

I
IEEE Standard 754 ... 1-3-2, 1-3-4, 1-3-13, 1-3-17, 1-3-73, 1-3-81—1-3-83, 1-3-85—1-3-87
Immediate operand instructions ... 1-2-44, 1-2-54, 1-2-124, 1-2-134—1-2-139
Implementations ... 1-3-1
Implementation choices 1-3-77
Implementation dependencies ... 1-2-1, 1-2-86, 1-3-19, 1-3-74
Imprecise exception traps ... 1-3-11, 1-3-78
Instruction, ABS.fmt ... 1-3-48
Instruction, ADD ... 1-2-56
Instruction, ADD.fmt ... 1-3-43
Instruction, ADDI ... 1-2-45
Instruction, ADDIU ... 1-2-46
Instruction, ADDU ... 1-2-57
Instruction, AND ... 1-2-66
Instruction, ANDI ... 1-2-51
Instruction, BC1F ... 1-3-69
Instruction, BC1FL ... 1-3-71
Instruction, BC1T ... 1-3-68
Instruction, BC1TL ... 1-3-70
Instruction, BCzF ... 1-2-155
Instruction, BCzFL ... 1-2-157
Instruction, BCzT ... 1-2-154
Instruction, BCzTL ... 1-2-156
Instruction, BEQ ... 1-2-107
Instruction, BEQL ... 1-2-115
Instruction, BGEZ ... 1-2-112
Instruction, BGEZAL ... 1-2-114
Instruction, BGEZALL ... 1-2-122
Instruction, BGEZL ... 1-2-120
Instruction, BGTZ ... 1-2-110
Instruction, BGTZL ... 1-2-118
Instruction, BLEZ ... 1-2-109
Instruction, BLEZL ... 1-2-117
Instruction, BLTZ ... 1-2-111
Instruction, BLTZAL ... 1-2-113
Instruction, BLTZALL ... 1-2-121
Instruction, BLTZL ... 1-2-119
Instruction, BNE ... 1-2-108
Instruction, BNEL ... 1-2-116
Instruction, BREAK ... 1-2-126

Instruction, CACHE ... 1-4-71
Instruction, C.cond.fmt ... 1-3-65
Instruction, CELL.L.fmt ... 1-3-63
Instruction, CELL.W.fmt ... 1-3-58
Instruction, CFC1 ... 1-3-39
Instruction, CFCz ... 1-2-150
Instruction, COPz ... 1-2-152
Instruction, CTC1 ... 1-3-38
Instruction, CTCz ... 1-2-149
Instruction, CVT.D.fmt ... 1-3-52
Instruction, CVT.E.fmt ... 1-3-53
Instruction, CVT.L.fmt ... 1-3-56
Instruction, CVT.Q.fmt ... 1-3-54
Instruction, CVT.S.fmt ... 1-3-51
Instruction, CVT.W.fmt ... 1-3-55
Instruction, DADD ... 1-2-60
Instruction, DADDI ... 1-2-47
Instruction, DADDIU ... 1-2-48
Instruction, DADDU ... 1-2-61
Instruction, DDIV ... 1-2-93
Instruction, DDIVU ... 1-2-94
Instruction, DIV ... 1-2-89
Instruction, DIV.fmt ... 1-3-46
Instruction, DIVU ... 1-2-90
Instruction, DMFC0 ... 1-4-63
Instruction, DMFC1 ... 1-3-36
Instruction, DMTC0 ... 1-4-62
Instruction, DMTC1 ... 1-3-35
Instruction, DMULT ... 1-2-91
Instruction, DMULTU ... 1-2-92
Instruction, DSLL ... 1-2-77
Instruction, DSLL32 ... 1-2-78
Instruction, DSLLV ... 1-2-83
Instruction, DSRA ... 1-2-81
Instruction, DSRA32 ... 1-2-82
Instruction, DSRAV ... 1-2-85
Instruction, DSRL ... 1-2-79
Instruction, DSRL32 ... 1-2-80
Instruction, DSRLV ... 1-2-84
Instruction, DSUB ... 1-2-62
Instruction, DSUBU ... 1-2-63
Instruction, ERET ... 1-4-66
Instruction fetch ... 1-2-6, 1-5-3
Instruction, FLOOR.L.fmt ... 1-3-64
Instruction, FLOOR.W.fmt ... 1-3-60
Instruction format ... 1-3-19, 1-7-1
Instruction, J ... 1-2-101
Instruction, JAL ... 1-2-102
Instruction, JALR ... 1-2-104
Instruction, JR ... 1-2-103
Instruction, LB ... 1-2-10
Instruction, LBU ... 1-2-11
Instruction, LD ... 1-2-16
Instruction, LDC1 ... 1-3-27
Instruction, LDCz ... 1-2-143
Instruction, LDL ... 1-2-21
Instruction, LDR ... 1-2-23
Instruction, LH ... 1-2-12
Instruction, LHU ... 1-2-13
Instruction, LL ... 1-2-25
Instruction, LLD ... 1-2-27
Instruction, LUI ... 1-2-54
Instruction, LW ... 1-2-14
Instruction, LWC1 ... 1-3-25
Instruction, LWCz ... 1-2-142
Instruction, LWL ... 1-2-17, 1-4-79
Instruction, LWR ... 1-2-19, 1-4-77
Instruction, LWU ... 1-2-15
Instruction, MFC0 ... 1-4-61
Instruction, MFC1 ... 1-3-34
Instruction, MFCz ... 1-2-148
Instruction, MFHI ... 1-2-95
Instruction, MFLO ... 1-2-96
Instruction, MOV.fmt ... 1-3-49
Instruction, MTC0 ... 1-4-60
Instruction, MTC1 ... 1-3-33
Instruction, MTCz ... 1-2-147
Instruction, MTHI ... 1-2-97
Instruction, MTLO ... 1-2-98
Instruction, MUL.fmt ... 1-3-45
Instruction, MULT ... 1-2-88
Instruction, MULTU ... 1-2-88
Instruction, NEG.fmt ... 1-3-50
Instruction, NOR ... 1-2-69
Instruction, OR ... 1-2-67
Instruction, ORI ... 1-2-52
Instruction, RFE ... 1-4-65
Instruction, ROUND.L.fmt ... 1-3-61
Instruction, ROUND.W.fmt ... 1-3-57
Instruction, SB ... 1-2-29
Instruction, SC ... 1-2-41
Instruction, SCD ... 1-2-43
Instruction, SD ... 1-2-32
Instruction, SDC1 ... 1-3-30
Instruction, SDCz ... 1-2-145
Instruction, SDL ... 1-2-37
Instruction, SDR ... 1-2-39
Instruction, SH ... 1-2-30

Instruction, SLL ... 1-2-71
Instruction, SLLV ... 1-2-74
Instruction, SLT ... 1-2-64
Instruction, SLTI ... 1-2-49
Instruction, SLTIU ... 1-2-50
Instruction, SLTU ... 1-2-65
Instruction, SQRT.fmt ... 1-3-47
Instruction, SRA ... 1-2-73
Instruction, SRAV ... 1-2-76
Instruction, SRL ... 1-2-72
Instruction, SRLV ... 1-2-75
Instruction, SUB ... 1-2-58
Instruction, SUB.fmt ... 1-3-44
Instruction, SUBU ... 1-2-59
Instruction, SW ... 1-2-31
Instruction, SWC1 ... 1-3-29
Instruction, SWCz ... 1-2-144
Instruction, SWL ... 1-2-33, 1-4-80
Instruction, SWR ... 1-2-35, 1-4-78
Instruction, SYNC ... 1-2-127
Instruction, SYSCALL ... 1-2-125
Instruction, TEQ ... 1-2-132
Instruction, TEQI ... 1-2-138
Instruction, TGE ... 1-2-128
Instruction, TGEI ... 1-2-134
Instruction, TGEIU ... 1-2-135
Instruction, TGEU ... 1-2-129
Instruction, TLBP ... 1-4-70
Instruction, TLBR ... 1-4-67
Instruction, TLBWI ... 1-4-68
Instruction, TLBWR ... 1-4-69
Instruction, TLT ... 1-2-130
Instruction, TLTI ... 1-2-136
Instruction, TLTIU ... 1-2-137
Instruction, TLTU ... 1-2-131
Instruction, TNE ... 1-2-133
Instruction, TNEI ... 1-2-139
Instruction, TRUNC.L.fmt ... 1-3-62
Instruction, TRUNC.W.fmt ... 1-3-58
Instruction, XOR ... 1-2-68
Instruction, XORI ... 1-2-53
Integer overflow exception ... 1-2-45, 1-2-47, 1-2-56, 1-2-58, 1-2-60, 1-2-62, 1-5-11
Interlock ... 1-2-8, 1-2-86, 1-3-24, 1-3-77—1-3-79, 1-4-79, 1-6-3
Interrupt exception ... 1-5-13
Interrupts ... 1-3-38, 1-4-40, 1-4-42, 1-4-46, 1-4-55—1-4-56, 1-4-65—1-4-66, 1-5-1—1-5-3, 1-5-13
Invalid operation exception ... 1-3-2

J
Jump ... 1-2-6, 1-2-99, 1-2-101, 1-2-103
Jump and link ... 1-2-99, 1-2-102, 1-2-104

K
Kernel mode ... 1-4-1, 1-4-4, 1-4-6, 1-4-10, 1-4-40, 1-4-42

L
Link ... 1-2-99, 1-2-102, 1-2-104—1-2-105, 1-2-113—1-2-114, 1-2-121—1-2-122
LLAddr system coprocessor register ... 1-4-56
LLbit, load linked bit ... 1-2-3, 1-2-25, 1-2-27, 1-2-41, 1-2-43, 1-4-65, 1-4-66
LO, multiply/divide register ... 1-2-3, 1-2-86, 1-2-96, 1-2-98, 1-6-4
Load ... 1-2-6, 1-2-8, 1-2-10—1-2-17, 1-2-19, 1-2-21, 1-2-23, 1-2-25, 1-2-27, 1-2-141—1-2-143, 1-3-24, 1-4-77, 1-4-79
Load delay slot ... 1-2-8, 1-3-24, 1-6-3

M
Machine Check exception ... 1-5-14
Memory, cache ... 1-2-3
Memory, main ... 1-2-3
MIPS II ... 1-1-1, 1-1-5, 1-2-8, 1-2-25, 1-2-41, 1-2-115—1-2-122, 1-2-127—1-2-139, 1-2-143, 1-2-145, 1-2-156—1-2-157, 1-3-3—1-3-4, 1-3-27, 1-3-30, 1-3-47, 1-3-57—1-3-60, 1-3-70—1-3-71, 1-4-1, 1-5-11, 1-6-3, 1-7-1, 1-8-1
MIPS III ... 1-1-1, 1-2-4, 1-2-15—1-2-16, 1-2-21, 1-2-23, 1-2-27, 1-2-32, 1-2-37, 1-2-39, 1-2-43, 1-2-47—1-2-48, 1-2-60—1-2-63, 1-2-77—1-2-84, 1-2-91—1-2-94, 1-2-140, 1-2-143, 1-2-145, 1-3-1, 1-3-3—1-3-4, 1-3-6, 1-3-10, 1-3-14, 1-3-25, 1-3-27, 1-3-30, 1-3-35—1-3-36, 1-3-42, 1-3-56, 1-3-61—1-3-64, 1-3-73, 1-4-1, 1-4-8—1-4-10, 1-4-31, 1-4-39, 1-4-42, 1-4-60—1-4-63, 1-7-4, 1-8-3
Mod, modulo ... 1-2-3
Multiply ... 1-2-86—1-2-88, 1-2-91—1-2-92
Multiprocessor ... 1-1-1, 1-1-5, 1-2-127

N
NMI ... 1-5-2
NMI exception ... 1-5-7
Nor, logical ... 1-2-3, 1-2-55, 1-2-69
Notation ... 1-2-2—1-2-3, 1-3-20

O
Opcode assignments ... 1-7-2, 1-7-4
Or, logical ... 1-2-3, 1-2-44, 1-2-52, 1-2-55, 1-2-67

Ordering, byte ... 1-1-1, 1-2-5, 1-3-4, 1-3-6, 1-3-25, 1-4-40, 1-4-58
Overflow, integer ... 1-2-45, 1-2-47, 1-2-56, 1-2-58, 1-2-60, 1-2-62, 1-5-11

P

Page Mask system coprocessor register ... 1-4-34
PC, program counter ... 1-2-3
Pipelining ... 1-3-77—1-3-78
Precise exception traps ... 1-3-78
PRId system coprocessor register ... 1-4-55

R

R2010 floating-point implementation VLSI ... 1-3-10, 1-3-81
R2360 floating-point implementation board ... 1-3-10, 1-3-81
R3010 floating-point implementation VLSI ... 1-3-10, 1-3-81
R4000 floating-point implementation VLSI ... 1-3-81
R4010 floating-point implementation VLSI ... 1-3-10
R6010 floating-point implementation VLSI ... 1-3-10, 1-3-81
Register, format ... 1-3-6—1-3-7
Register, justification ... 1-3-6—1-3-7
Registers, coprocessor control ... 1-3-3
Registers, coprocessor general ... 1-3-3
Registers, floating-point ... 1-3-3
Reserved Instruction exception ... 1-4-1, 1-5-12
Reset ... 1-3-12, 1-4-19, 1-4-36—1-4-37, 1-4-40, 1-4-43, 1-4-57, 1-5-1—1-5-2, 1-5-5—1-5-6
Reset exception ... 1-5-6
Rounding Mode ... 1-3-13

S

Scheduling, instruction ... 1-3-78
Set on less than ... 1-2-44, 1-2-49—1-2-50, 1-2-55, 1-2-64—1-2-65
Shift ... 1-2-70
Shift, arithmetic ... 1-2-70, 1-2-73, 1-2-76, 1-2-81—1-2-82, 1-2-85
Shift, logical ... 1-2-70—1-2-72, 1-2-74—1-2-75, 1-2-77—1-2-80, 1-2-83—1-2-84
Shift, variable ... 1-2-74—1-2-76, 1-2-83—1-2-85
Soft Reset ... 1-5-2
Soft Reset exception ... 1-5-7
Software implementation ... 1-3-77, 1-3-81, 1-3-83
Special ... 1-2-7, 1-2-55—1-2-70
Status system coprocessor register ... 1-4-40
Store ... 1-2-6, 1-2-8, 1-2-29—1-2-33, 1-2-35, 1-2-37, 1-2-39, 1-2-41, 1-2-43, 1-2-141, 1-2-144—1-2-145, 1-3-24, 1-4-78, 1-4-80
Subroutine call ... 1-2-102, 1-2-104, 1-2-113—1-2-114, 1-2-121—1-2-122
Subtract, arithmetic ... 1-2-55, 1-2-58—1-2-59, 1-2-62—1-2-63
Supervisor mode ... 1-4-1, 1-4-5, 1-4-9, 1-4-42
Synchronization ... 1-2-8, 1-2-25, 1-2-27, 1-2-41, 1-2-43, 1-2-127
Syscall ... 1-2-125
Systemcall ... 1-2-125
System Call exception ... 1-5-11
System coprocessor register, BadVAddr ... 1-4-38, 1-5-8—1-5-10, 1-4-14
System coprocessor register, CacheErr ... 1-4-51
System coprocessor register, Cause ... 1-3-13, 1-3-15, 1-4-46, 1-4-55, 1-5-2, 1-5-13
System coprocessor register, Compare ... 1-4-55
System coprocessor register, Config. 1-4-57
System coprocessor register, Context ... 1-4-38
System coprocessor register, Count ... 1-4-55
System coprocessor register, ECC ... 1-4-54
System coprocessor register, EntryHi ... 1-4-33
System coprocessor register, EntryLo ... 1-4-34
System coprocessor register, EPC ... 1-4-49
System coprocessor register, Error ... 1-4-48
System coprocessor register, Error EPC ... 1-4-50
System coprocessor register, LLAddr ... 1-4-56
System coprocessor register, Page Mask ... 1-4-34
System coprocessor register, PRId ... 1-4-55
System coprocessor register, Status ... 1-4-40
System coprocessor register, TagHi ... 1-4-52
System coprocessor register, TagLo ... 1-4-52
System coprocessor register, TLB Index ... 1-4-36
System coprocessor register, TLB Random ... 1-4-37
System coprocessor register, TLB Wired ... 1-3-36
System coprocessor register, WatchHi ... 1-4-56
System coprocessor register, WatchLo ... 1-4-56
System coprocessor register, XContext ... 1-4-39

T

TagHi system coprocessor register ... 1-4-52
TagLo system coprocessor register ... 1-4-52
TLB, associative ... 1-4-30, 1-4-32—1-4-34, 1-4-36—1-4-39, 1-4-43, 1-4-67—1-4-70, 1-5-1—1-5-2, 1-5-6, 1-5-8—1-5-9, 1-6-11

TLB, flushing ... 1-4-1
TLB, in-cache ... 1-4-14, 1-4-17, 1-4-25, 1-5-5, 1-5-8—1-5-9
TLB Index system coprocessor register ... 1-4-36
TLB Invalid exception ... 1-5-9
TLB, micro ... 1-4-29—1-4-30, 1-6-11
TLB Modified exception ... 1-5-9
TLB, on-chip ... 1-4-14
TLB Random system coprocessor register ... 1-4-37
TLB Refill exception ... 1-5-8
TLB, shutdown ... 1-4-43
TLB, slice ... 1-4-30
TLB, translation lookaside buffer ... 1-2-3
TLB Wired system coprocessor register ... 1-4-36
Trap ... 1-2-128—1-2-139
Trap exception ... 1-5-11

U
Unaligned ... 1-2-17, 1-2-19, 1-2-21, 1-2-23, 1-2-33, 1-2-35, 1-2-37, 1-2-39
Unaligned load/store ... 1-2-8
Uncache LDCz/SDCz exception ... 1-5-14
Undefined ... 1-2-1, 1-2-10—1-2-14, 1-2-17, 1-2-19, 1-2-25, 1-2-27, 1-2-41, 1-2-43, 1-2-87—1-2-94, 1-2-97—1-2-99, 1-2-104, 1-2-142—1-2-143, 1-2-145, 1-3-4, 1-3-9, 1-3-11—1-3-12, 1-3-19, 1-3-25, 1-3-27, 1-3-30, 1-3-41—1-3-42, 1-4-5—1-4-6, 1-4-8—1-4-10, 1-4-14, 1-4-34, 1-4-43, 1-4-60, 1-4-62—1-4-63, 1-4-71, 1-4-79, 1-5-6, 1-5-8—1-5-10, 1-6-3, 1-8-1
Unimplemented operation exception ... 1-3-77, 1-3-79
Unsigned ... 1-2-46, 1-2-48, 1-2-50, 1-2-55, 1-2-57, 1-2-59, 1-2-61, 1-2-63, 1-2-65, 1-2-86, 1-2-88, 1-2-90, 1-2-92, 1-2-94
User mode ... 1-4-1, 1-4-3, 1-4-8, 1-4-40, 1-4-42

V
Virtual Coherency exception ... 1-5-14
Virtual index ... 1-4-3, 1-4-8, 1-4-30, 1-5-14
Virtual tag ... 1-4-30

W
Watch exception ... 1-5-15
WatchHi system coprocessor register ... 1-4-56
WatchLo system coprocessor register ... 1-4-56

X
XContext system coprocessor register ... 1-4-39

μPD30360

(V_R3600)

User's Manual

BEFORE READING THIS MANUAL

Target reader:

This manual is designed for engineers already familiar with the functions of the uPD30360 (V_R3600™) and intending to design an application system using these functions.

Objectives:

This manual is designed to provide information regarding the architecture of the V_R3600 to the user engineers in the configuration shown in the next paragraph.

Configuration:

This manual is configured as follows:

- Outline
- CPU Architecture
- FPU Architecture

How to read:

This manual is designed on the assumption that the user is already familiar with basic knowledges on electronics, logic circuits, and microcomputers.

The word "V_R3600" stands for the following meanings

In Part 2 : CPU of V_R3600

In Part 3 : FPU of V_R3600

To become familiar with the V_R3600 functions:

⟶ Read cover to cover according to "Table of Contents".

For details regarding instruction functions for the V_R3600:

⟶ Refer to "Instruction Set" of Part 2 and Part 3.

Examples:

- Cuation and Notes:
  Describes contents which require a special attention.

- **Remarks:**
  Provides supplemental information for the text.

- Numeric expression:
  Decimal ... xxxx
  Hexadecimal ... xxxxH

Part 1 Overview

Chapter 1 Overview

1.1 V_R3600 Features

The uPD30360 (V_R3600) is an RISC type microprocessor integrating the CPU and FPU on a single chip. The V_R3000A™ is used as the main CPU, and the V_R3010A™ is used as the FPU.
The CPU block consists of two tightly coupled processors. The first processor is a complete 32-bit RISC type CPU. The second processor is the system control co-processor (CP0). It consists of the high-speed translation buffer function (TLB: Translation Lookaside Buff) and the control register. The system control co-processor supports cache memories independently for instruction and data.
The FPU block operates as the floating point operation co-processor for the CPU and executes arithmetic operation for floating point format value. The FPU conforms to ANSI/IEEE standard specifications 754-1985 "IEEE Binary Floating Point Operation Specifications".
These are the V_R3600 features:

o V_R3000A and V_R3010A equivalent are mounted on a single chip.

o Functionally compatible with V_R3000A and V_R3010A in addition to software compatibility.

o Fully 32-bit operation
  . 32 built-in 32-bit registers
  . All instructions and addresses are 32-bit.

o Built-in cache controller
  . High band-width memory interface separately processes the external instruction cache and the data cache from 4K bytes to 256K bytes.
    Both caches are accessed in single CPU cycle.
  . All cache control mechanisms are internally provided.

o Built-in memory control unit

. Fully associative translation lookaside buffer (TLB) can execute high speed virtual-physical memory mapping in the 4G-byte virtual address space.

o $\overline{\text{FpInt}}$ signal internally connected/disconnected mode

. $\overline{\text{FpInt}}$ connected mode

The internal pin number, to connect the interrupt signal ($\overline{\text{FpInt}}$) from the FPU to the CPU, is fixed (internally connected) to $\overline{\text{Int3}}$. The FpInt signal is also externally output.

. $\overline{\text{FpInt}}$ disconnected mode

The $\overline{\text{FpInt}}$ signal is output. The interrupt signal from the FPU to be connected to the CPU can be externally selected. $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ are not internally connected.

1.2 Functional Description

<table>
<tr><th>Pin name</th><th>Input/output</th><th>Function</th></tr>
<tr><td>AccTyp (1:0)
(Access types 1-0)</td><td>Output</td><td>This signal basically indicates the data size for accessing the main memory (excluding the cache memory).
(1) In the stall cycle and when AccTyp(2) is low
<table>
<tr><th>AccTyp(1)</th><th>AccTyp(0)</th><th>Data size</th></tr>
<tr><td>0</td><td>0</td><td>Byte</td></tr>
<tr><td>0</td><td>1</td><td>Half word</td></tr>
<tr><td>1</td><td>0</td><td>3 bytes</td></tr>
<tr><td>1</td><td>1</td><td>Word</td></tr>
</table>
(2) In the stall cycle and when AccTyp(2) is high AccTyp(0) indicates the reason for entering main memory read cycle, as indicated below:
<table>
<tr><th>AccTyp(0)</th><th>Cause</th></tr>
<tr><td>0</td><td>Data cache miss hit</td></tr>
<tr><td>1</td><td>Instruction cache miss hit</td></tr>
</table>
(3) When $\overline{\text{DispParRexEnd}}$ is asserted, when resetting, AccTyp(1) indicates parity error when cache an incorrect stall occurs.
<table>
<tr><th>AccTyp(1)</th><th>Cause</th></tr>
<tr><td>0</td><td>No parity error</td></tr>
<tr><td>1</td><td>Parity error</td></tr>
</table>
</td></tr>
<tr><td>AccTyp(2)
(Access type 2)</td><td>Output</td><td>This signal indicates the kind of memory area (including cache memories) in the stall cycle, as indicated below.
<table>
<tr><th>AccTyp(2)</th><th>Memory area</th></tr>
<tr><td>0</td><td>Uncached area</td></tr>
<tr><td>1</td><td>Cached area</td></tr>
</table>
</td></tr>
</table>

(cont'd)

<table>
<tr><th>Pin name</th><th>Input/output</th><th>Function</th></tr>
<tr><td>$\overline{\text{Int}}$ (5:0)
(Interrupt request)</td><td>Input</td><td>These pins are used for maskable interrupt request signals. These pins are also used for setting the operation mode, when resetting the V_R3600.
The $\overline{\text{Int3}}$ condition changes by a change in the Mode pin value.
<table>
<tr><th rowspan="2">Mode</th><th colspan="2">$\overline{\text{Int3}}$ pin condition</th></tr>
<tr><th>During reset</th><th>During normal operation</th></tr>
<tr><td>1</td><td>Not connected to $\overline{\text{FpInt}}$
Input is valid</td><td>Internally connected to $\overline{\text{FpInt}}$
Input is invalid</td></tr>
<tr><td>0</td><td>Not connected to $\overline{\text{FpInt}}$</td><td>Not connected to $\overline{\text{FpInt}}$</td></tr>
</table></td></tr>
<tr><td>$\overline{\text{IRd}}$ (2:1)
(Instruction cache read)</td><td>Output</td><td>This signal is output during instruction cache read. This signal is used as output enable or strobe for cache RAM.</td></tr>
<tr><td>$\overline{\text{IWr}}$ (2:1)
(Instruction cache write)</td><td>Output</td><td>This signal is output during instruction cache write. This signal is used as write enable or strobe for cache RAM.</td></tr>
<tr><td>$\overline{\text{MemRd}}$
(Memory read)</td><td>Output</td><td>This signal is asserted, when the V_R3600 reads the memory.</td></tr>
<tr><td>$\overline{\text{MemWr}}$
(Memory write)</td><td>Output</td><td>This signal is asserted, when the V_R3600 writes the memory.</td></tr>
<tr><td>Mode
($\overline{\text{FpInt}}$ internal connection)</td><td>Input</td><td>This signal selects the internal connection mode as indicated below.
The Mode signal can be changed only when $\overline{\text{Reset}}$ = "0".
<table>
<tr><th>Mode</th><th>Mode</th></tr>
<tr><td>1 (V_{DD})</td><td>$\overline{\text{FpInt}}$ connected mode</td></tr>
<tr><td>0 (Gnd)</td><td>$\overline{\text{FpInt}}$ disconnected mode</td></tr>
</table></td></tr>
<tr><td>RdBusy
(Read busy)</td><td>Input</td><td>The V_R3600 asserts this signal in response to a read request, when the memory is busy. The pipeline remains in the stall cycle, until this signal is deasserted.</td></tr>
</table>

(Cont'd)

Pin name	Input/output	Function
$\overline{\text{Reset}}$ (Reset)	Input	This signal initializes the V_R3600. In order to allow the PLL in the FPU to operate normally, this signal must be asserted for 3000 clocks or for 200us. The pipeline remains in stall until this signal is deasserted. This pin is internally connected to the internal FPU. When this signal is deasserted, the setting must be synchronized to the rising edge of the clock.
$\overline{\text{Run}}$ (Run)	Output	This signal is asserted when the V_R3600 is in the run cycle, and deasserted when in the stall cycle. This pin is internally connected to the internal FPU.
$\overline{\text{SysOut}}$ (Sync clock)	Output	This signal reflects the internal clock of the V_R3600, and is used for generating the system clock.
AdrLo (17:0) (Lower address 17-0)	Output	Lower 18 bits of the address, used to access the external memory (including cache memory). When accessing cache memory, only the upper 16 bits are used. Since, AdrLo (17:12) and Tag (17:12) are overlapped, the cache size can be selected from 4K bytes to 256K bytes. If ExCache is asserted, when resetting, AdrLo (17:16) serves as the most significant bit of the cache address. If ExCache is deasserted, AdrLo (17:16) serves as CpCond (3:2), which is input. If $\overline{\text{MPAdrDisable}}$ is asserted when resetting, these pins become Hi-Z condition during multi-processor stall.
$\overline{\text{BusError}}$	Input	This pin indicates bus error (such as caused by bus time out) to the V_R3600. However, when writing via the write buffer, the address at which bus error occurred has already been off the V_R3600. In such a case, bus error must be processed, not by the bus error signal, but by the interrupt signal.
Clk2xPhi	Input	This pin inputs the double frequency clock used to determine the positions of the internal phases; phase 1 and phase 2. This pin is internally connected to the FPU.
Clk2xRd	Input	This pin inputs the double frequency clock used to determine the cache RAM enable time interval. This pin is internally connected to the FPU.

(Cont'd)

Pin name	Input/output	Function
Clk2xSmp	Input	This pin inputs the double frequency clock, used to determine the sampling point for input data. This pin is internally connected to the FPU.
Clk2xSys	Input	This pin inputs the double frequency clock used for generating $\overline{\text{SysOut}}$. This pin is internally connected to the FPU.
CpBusy (Coprocessor busy)	Output*	This signal is set to high when the FPU needs a longer instruction execution time. If this signal becomes high, the V_R3600 enters the stall cycle. This stall cycle continues until this signal becomes low. Since the V_R3600 contains the FPU and the CpBusy signal is internally connected to the FPU, normally, this signal is not used.
CpCond (1) (Coprocessor condition 1)	Output*	This signal is conditional branch status, handed from the FPU to the CPU. Since the V_R3600 contains the FPU and the CpCond1 signal is internally connected to the FPU, normally, this signal is not used.
CpCond (0) (Coprocessor condition 0)	Input	This signal is conditional branch status handed from the coprocessor to the V_R3600.
$\overline{\text{CpSync}}$ (Coprocessor sync clock)	Output	This signal is used for synchronization with the coprocessor. Since the V_R3600 contains the FPU and the $\overline{\text{CpSync}}$ signal is internally connected to the FPU, normally, this signal is not used.
D (31:0) (Data bus)	Input/output	32-bit bi-directional data bus. This bus is internally connected to the FPU.
DP (3:0) (Data parity)	Input/output	DP provides even parity to 4-byte data of D (31:0). These pins are internally connected to the internal FPU.
DClk (Data cache read)	Output	Address latch signal used for accessing the data cache. This signal is output in every cycle.
$\overline{\text{DRd}}$ (2:1) (Data cache read)	Output	This signal is output when reading the data cache. This signal uses the cache RAM output as an enable signal.

*: Direction of signal is reversed with respect to the V_R3000/3000A.

(Cont'd)

<table>
<tr><th>Pin name</th><th>Input/output</th><th>Function</th></tr>
<tr><td>$\overline{DWr}$ (2:1)
(Data cache read)</td><td>Output</td><td>This signal is output when writing the data cache.
This signal is used as the cache RAM write enable or strobe signal.</td></tr>
<tr><td>$\overline{Exception}$
(Exception)</td><td>Output</td><td>This signal is asserted when the V_R3600 generates an exception and the pipeline stops.
This signal is used to stop the pipeline for the coprocessor.
This signal is internally connected to the internal FPU.</td></tr>
<tr><td>$\overline{FpInt}$
(FPU interrupt)</td><td>Output</td><td>This pin outputs the interrupt from the internal FPU,so that the internal FPU interrupt can be input to either one of $\overline{Int}$ (5:0). Whether or not this pin is internally connected to $\overline{Int3}$ signal depends on the Mode pin.
<table>
<tr><th rowspan="2">Mode</th><th colspan="2">$\overline{FpInt}$, $\overline{Int3}$ signal connection condition</th></tr>
<tr><th>During reset</th><th>During normal operation</th></tr>
<tr><td>1</td><td>Not connected</td><td>Internally connected</td></tr>
<tr><td>0</td><td>Not connected</td><td>Not connected</td></tr>
</table></td></tr>
<tr><td>IClk
(instruction cache latch clock)</td><td>Output</td><td>Address latch signal used for accessing the instruction cache.
This signal is output in every cycle.</td></tr>
<tr><td>Tag (31:12)
(Tag bus)</td><td>Input/output</td><td>This bus inputs/outputs a tag, when accessing the cache. When reading (loading) the cache memory, the V_R3600 compares the upper 20 bits of the address with the tag input to determine cache hit/miss hit.
When accessing the main memory, the upper 16 bits, in conjunction with AdrLo (17:0), provide 32-bit physical address.</td></tr>
<tr><td>TagP (2:0)
(Tag parity)</td><td>Input/output</td><td>TagP provides even parity for Tag (31:12) and TagV. If parity error is detected when reading cache, the V_R3600 determines it as a cache miss.</td></tr>
<tr><td>TagV
(Tag valid)</td><td>Input/output</td><td>This pin is used for transaction of the valid bit with the V_R3600 cache. This signal is asserted (output), only when writing a 32-bit word during cache write. Otherwise, this pin is deasserted.
This pin becomes input, when reading cache indicating a cache hit.</td></tr>
</table>

(Cont'd)

Pin name	Input/output	Function
$\overline{\text{WrBusy}}$ (Write busy)	Input	When the main memory is busy, this pin is asserted in response to a write request from the V_R3600. The stall cycle continues until this signal is deasserted.
$\overline{\text{XEn}}$ (Read enable)	Output	This signal serves as a read enable signal, when a read buffer is externally connected.

1.3 Mode Selection and Pin Condition

The V_R3600 can be set either in the $\overline{\text{FpInt}}$ signal internally connected or disconnected mode by the Mode pin value.

(1) $\overline{\text{FpInt}}$ connected mode (Mode = 1)

In this mode, the internal connection condition of $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ changes depending on the condition of the $\overline{\text{Reset}}$ signal. When $\overline{\text{Reset}}$ = "0", $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ signals are not internally connected. $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ signals are internally connected when $\overline{\text{Reset}}$ = "1". This mode is used for a system in which an interrupt from the FPU to the CPU is received by $\overline{\text{Int3}}$, that is, $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ are to be connected.

(2) $\overline{\text{FpInt}}$ disconnected mode (Mode = 0)

In this mode, $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ are not connected, regardless of the $\overline{\text{Reset}}$ signal condition. This mode is used in a system in which $\overline{\text{FpInt}}$ is connected other than to $\overline{\text{Int3}}$ interrupt pin. The method to externally connect the $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ pins is the same as for the V_R3000™, the V_R3000A, V_R3010™, and the V_R3010A. In addition, $\overline{\text{FpInt}}$ and $\overline{\text{Int3}}$ can be externally connected.

1.4 Internal Block Diagram

The V_R3600 consists of these three main blocks; CPU, FPU, and the mode controller. The CPU is equivalent to the V_R3000A, the FPU to the V_R3010A. The mode controller controls signals for each mode. Figure 1-1 shows the internal connections for the V_R3600.

Fig. 1-1 Internal Block Diagram

CPU

FPU

Int3 — *C

Reset — *A

*B — Reset

CpBusy — FpBusy

CpCond1 — FpCond

Run — Run

Exception — Exception

CpSync — FpSync

Data31_0 — Data31_0

Clk2xSys — Clk2xSys

Clk2xSmp — Clk2xSmp

Clk2xRd — Clk2xRd

Clk2xPhi — Clk2xPhi

Int5,4, 2_0

RdBusy

*D ← FpInt

WrBusy

Open — FpPresent

BusError

P110n — GND

SysOut

CpCond0

FpSysIn

Tag31_12

FpSysOut

TagP2_0

TagV

DataP3_0

AdrLo17_0

AccType2_0

MemRd

MemWr

XEn

IRd2_1

IWr2_1

DRd2_1

DWr2_1

IClk

DClk

*A *B *C *D

Mode controller

Inside

Outside

V_R3000A signal

Reset

Int3

FpInt

Mode control signal (Mode)

1.5 System Configuration

Fig. 1-2 System Configuration Example

Data Bus
Tag Bus
AdrLo Bus

Tag
TagV
TagP
AdrLo
Data
DataP

Latch
IClK
DClk
Latch

V_R3600

Data
Tag
IAdr
[15:2]
I-Cache
OE
WE

DAdr
[15:2]
Tag
Data
D-Cache
OE
WE

$\overline{IRd}$
$\overline{Iwr}$
$\overline{DRd}$
$\overline{DWr}$

Memory
Interface

$\overline{Reset}$
$\overline{XEn}$
$\overline{CpSync}$
$\overline{SysOut}$
$\overline{Run}$
AccType[2:0]
$\overline{Exception}$
$\overline{MemRd}$
CpBusy
$\overline{MemWr}$
CpCond1
RdBusy
CpCond0
$\overline{WrBusy}$
CpCond[0]
$\overline{Int[5:0]}$
$\overline{BusError}$

Clocks
Clk2xSys
Clk2xSmp
Clk2xRd
Clk2xPhi

Part 2 CPU Architekture

Chapter 1 Overview CPU Architecture

The V_R3600 is a 32-bit microprocessor which employs RISC architecture instead of the CISC architecture adopted by the V60™/V70™/V80™. This chapter outlines the RISC architecture.

1.1 What Is RISC?

Existing microprocessors have evolved from 8-bit architecture to the 16-bit architecture, and then to the 32-bit architecture by adding instructions and addressing modes that provide powerful support to high-level languages and operating systems. Computers based on these microprocessors are known as Complex Instruction Set Computers (CISCs). The CISC controls almost all interpretation of sophisticated instructions and addressing modes by using a microprogram closely related to the hardware. In general, the CISC, equipped with such sophisticated instructions and addressing modes, requires a large quantity of hardware and is highly complicated. In a sense, therefore, the CISC evolution history can be said to be the history of improvement in the integration scale on the LSI.
In the meantime, as a result of tracing system programs and object modules, to investigate how the software actually uses the resources of a processor, it has been recently revealed that only several simple instructions are used with dominant frequency.
Moreover, the ordinary compiler hardly outputs sophisticated instructions and abundant addressing modes, which are the CISC features. Based on the results of these statistical analyses, Reduced Instruction Set Computers (RISCs) have emerged in recent years.
Unlike the existing CISC, the RISC executes almost all instructions within 1 clock period by providing only the basic addressing modes and instructions and by making the lengths of the instructions equal. Consequently, the processor is not controlled by the microprogram, and therefore, the processor construction can be made simple. As a result, the quantity of hardware required can be dramatically reduced. In addition,

because sophisticated instructions and addressing modes, that provide a powerful support to high-level languages and operating systems, controls, including low-level controls, are dependent upon the software developed by the user. To appreciate the full performances of the processor, therefore, the hardware, firmware, and software functions must be appropriately divided.

The RISC architecture does not employ a particularly unique, new method. However, computers employing this architecture have increasingly become popular in recent years, primarily because, since compilers that accomplish optimization at "microcode" level have been developed, high-level languages have been increasingly used for programming. The second reason is that the high-speed RAM can now be used as an instruction cache, thanks to rapid progress achieved in semiconductor memory and mounting technologies. As a consequence, the user can fully appreciate the processor resources with a program described in a high-level language.

The RISC architecture is only a technique to draw out the maximum performances of microprocessors. Its main purpose is not to decrease the number of instructions or addressing modes. The driving force behind the RISC processor development is nothing more than improvement in performance.

1.2 Defining Performance

The performance of a processor can be defined as the time required to accomplish a specific task (or program or algorithm, or benchmark) and can be expressed as the product of three factors:

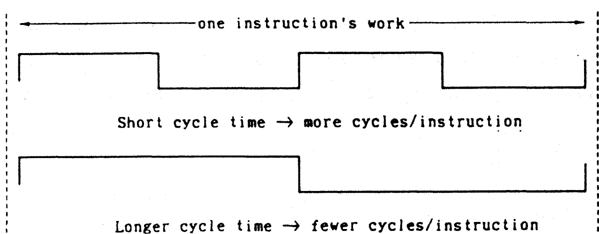
```
Time per Task = C x T x I
where         C = Cycles per Instructions
              T = Time per Cycle (clock speed)
              I = Instructions per Task
```

Performance can be improved by reducing any of these three factors. RISC-type designs strive to improve performance by minimizing the first two factors. However, changes that reduce the cycles/instruction and time/cycle factors tend to increase the instructions/task factor: this tendency has been the focus of most criticisms leveled at RISC. However, the use of optimizing compilers and other techniques mitigate this tendency. The sections that follow discuss each of the three performance-related factors and typical techniques used in RISC-type designs to minimize each factor.

1.2.1 Time per instruction

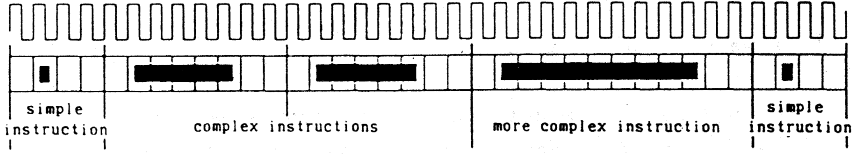
The time required to execute an instruction is the product of the first two factors (C and T) in the equation developed in the preceding section. These two factors are complementary: increasing the clock speed (reducing the time per cycle) decreases the amount of work that can be accomplished within a cycle. Thus, fast clock rates (short cycle times) tend to increase the number of cycles required to perform an instruction as illustrated in the following Figure:

Fig. 1-1 Cycle Time and Cycle/Instruction



In most processors, it makes little difference whether cycle time is short and instructions require many cycles, or cycle time is long with instructions requiring few cycles — it's the total time/instruction (time/cycle X cycles/instruction) that is significant. Typically, the cycle time is chosen to allow execution of the most simple operations (or sub-operations) in a single cycle, and execution of other, more complex operations in multiple cycles. Thus, the instruction stream in a typical CISC processor might look like this:

Fig. 1-2 CISC Instruction Stream



Executing the most simple instructions in the above example requires four cycles and executing the more complex instruction requires eight or twelve cycles. This approach would seem to achieve a rather efficient utilization of time: simple instructions are executed quickly and more complicated instructions are given additional time to execute. Each instruction is given just the amount of time it needs — no more and no less. This technique has one very damaging drawback,

however, that makes it unsuitable in RISC-type designs: it greatly complicates the use of instruction pipelines. Instruction pipelines are an essential technique used to reduce the cycles/instruction factor, and the gains that pipelines can provide are negated by instruction sets where the cycles/instruction factor is variable. The advantages of instruction pipelines and the impact that their use has on the design of instruction sets are discussed in the sections that follow.

1.2.2 Cycles per instruction (C)

If the work that each instruction performs is simple and straightforward, then the time required to execute each instruction can be shortened and the number of cycles reduced. The goal of RISC designs is to achieve an execution rate of one machine cycle per instruction. Techniques that allow this goal to be approached include:

(1) Instruction pipelines
(2) Load/Store architecture
(3) Delayed load instructions
(4) Delayed branch instructions

(1) Instruction Pipelines

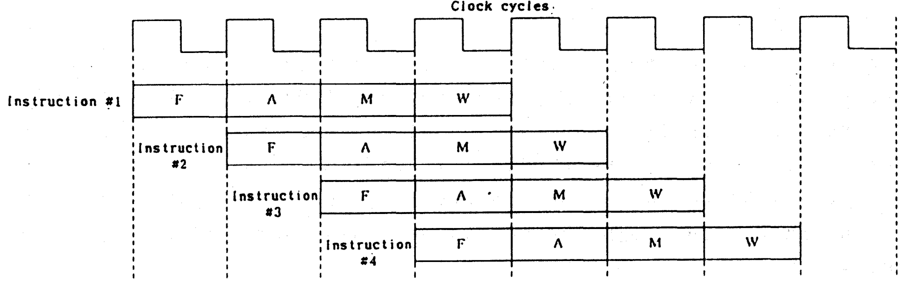
One way to reduce the number of cycles required to execute an instruction is to overlap the execution of multiple instructions. Instruction pipelines work by dividing the execution of each instruction into several discrete portions and then executing multiple instructions simultaneously. For example, the execution of an instruction might be subdivided into four portions as shown below:

Fig. 1-3 Steps of Instruction Execution

Cycle #1	Cycle #2	Cycle #3	Cycle #4
Fetch Instruction (F)	ALU Operation (A)	Access Memory (M)	Write Results (W)

In this example, four clock cycles are required to execute an instruction. An instruction pipeline, however, can potentially reduce the number of cycles/instruction by a factor equal to the depth of the pipeline. For example, in the following figure, each instruction still requires a total of four clock cycles to execute. However, if a four-level instruction pipeline is used, a new instruction can be initiated at each clock cycle and the effective execution rate is one cycle per instruction. The instruction pipeline technique can be likened to an assembly line — the instruction progress from one specialized stage to the next until it is completed just as an automobile might move along an assembly line. This is in contrast to the non-pipeline, microcoded approach where all the work is done by one general unit, which is less capable at each individual task.

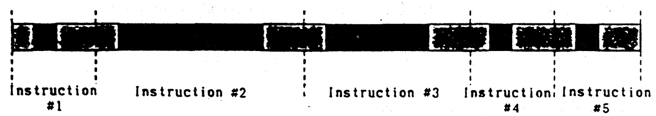
Fig. 1-4 Pipeline Works



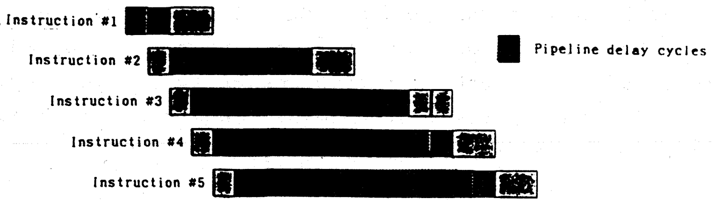
Note that the previous paragraph stated that a pipeline can potentially reduce the number of cycles/instruction by a factor equal to the depth of the pipeline. Fulfilling this potential requires that the pipeline always be filled with useful instructions and that nothing delay the advance of instructions through the pipeline. These requirements impose certain demands on the architecture. For example, consider the earlier example of serially executing an instruction stream where each instruction can require a different number of clock cycles. The following figure illustrates how this instruction stream might look as it proceeds through a pipeline:

Fig. 1-5 Serial Processing and Pipelined Processing

(a) Serial Instruction Execution



(b) Pipelined Instruction Execution



In this example, the darkly-shaded cycles indicate those where the instructions require the use of the same resources (for example, the ALU, shifters, or registers). Competition for these resources blocks the progression of the instructions through the pipeline and causes delay cycles to be inserted for many of the instructions until the required resources become available. The

pipeline technique shortens the average number of cycles/instruction in this example, but the gains are greatly reduced by the delay cycles that must be added.

The negative effect of the variable execution times is actually much worse than the simple preceding example might indicate. Management of an instruction pipeline requires proper and efficient handling of events such as branches, exceptions or interrupts that can completely disrupt the flow of instructions. If the instruction stream can include a variety of different instruction lengths and a mixture of delay and normal cycles, pipeline management becomes very complex. Additionally, such a varied, complex instruction stream makes it almost impossible for a compiler to schedule instructions so as to reduce or eliminate delays. It is for these reason that a primary goal of RISC designs is to define an instruction set where execution of all, or most, instructions requires a uniform number of cycles and, ideally, to achieve a rate of execution of one cycle/instruction.

(2) Load/Store Architecture

The discussion of the instruction pipeline illustrated how each instruction can be subdivided into several discrete parts which then permit the processor to execute multiple instructions in parallel. For this technique to work efficiently, the time required to execute each instruction sub-part should be approximately equal. If one part requires an excessive length of time, then all the cycles must be made longer or additional cycles must be used for the longer operation.

Instructions that perform operations on operands in memory tend to increase either the cycle time or the number of cycles/instruction. Such instructions require additional time for execution to calculate the addresses of the operands, read the required operands from memory, calculate the result, and store the results of the operation back to memory. To eliminate the negative impact of such instructions, RISC designs implement

a Load/Store architecture in which all operations are performed on operands held in processor registers, and main memory is accessed only by load and store instructions. This approach produces several benefits:

(a) reducing the number of memory accesses eases memory bandwidth requirements

(b) limiting all operations to registers helps simplify the instruction set

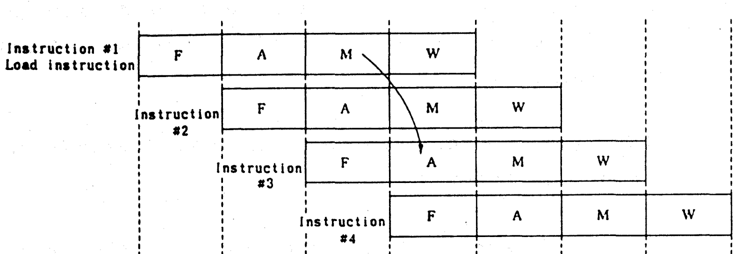
(c) eliminating memory operations makes it easier for compilers to optimize register allocation — this further reduces memory accesses and also reduces the instructions/task factor.

All of these factors help RISC designs approach their goal of executing one cycle/instruction. However, two classes of instructions still inhibit reaching this goal — load instructions and branch instructions. The sections that follow discuss how RISC designs overcome obstacles raised by these classes of instructions.

(3) Delayed Load Instructions

Load instructions read operands from memory into processor registers for subsequent operation by other instructions. Because memory typically operates at much slower speeds than processor clock rates, the loaded operand is not immediately available to subsequent instructions in a processor with an instruction pipeline. This data dependency situation is illustrated in the following figure:

Fig. 1-6 Delayed Load Instruction



In this illustration, the operand loaded by instruction #1 is not available in time for use in the "A" cycle of instruction #2. One way to handle this dependency is to delay the pipeline by inserting additional clock cycles into the execution of instruction #2 until the loaded data becomes available. This approach would obviously introduce delays that would increase the cycles/instruction factor.

The technique used in many RISC designs to handle this data dependency is to recognize and make visible to compilers the fact that all load instructions inherently have a latency or load delay. In the preceding illustration, there is a load delay or latency of one instruction. The instruction that immediately follows the load is described as being in the load delay slot. If the instruction that is in this slot does not require the data from the load, then no delay of the pipeline is required.

If the existence of this load delay is made visible to software, a compiler can arrange instructions to ensure that there is no data dependency between a load instruction and the instruction in the load delay slot. The simplest way of ensuring that there is no data dependency is to insert a NOP (No Operation) instruction to fill the slot:

```
Load    R1,A
Load    R2,B
NOP        ← this instruction fills the delay slot
Add     R3,R1,R2
```

Although it eliminates the need for hardware-controlled pipeline stalls in this case, filling the delay slot with NOP instructions still is not a very efficient use of the pipeline stream since the NOP instructions increase the code size and perform no useful work. (In practice, however, this technique need not have much negative impact on performance, especially if the delay is only one cycle.)

A more effective solution to handling the data dependency situation is to fill the load delay slot with a useful instruction. Good optimizing compilers can usually accomplish this, especially if the load delay is only one instruction. The following figure illustrates how a compiler might rearrange instructions to handle a potential data dependency:

```
# Consider the code for C:=A+B; F:=D;
  Load    R1,A
  Load    R2,B
  Add     R3,R1,R2 ←— this instruction stalls because R2 data
                       is not available
  Load    R4,D
   ...     ...
# An alternative code sequence (where delay length = 1)
  Load    R1,A
  Load    R2,B
  Load    R4,D
  Add     R3,R1,R2 ←— no stall since R2 data is available
   ...     ...
```

Since the Add (Add R3, R1, R2) instruction does not depend on the availability of the data from the third Load instruction (Load R4, D), the delay slot (for Load R2, B) can be filled with a usable instruction (Load R4, D) and the pipeline can be fully utilized.

(4) Delayed Branch Instructions

Branch instructions usually delay the instruction pipeline because the processor must calculate the effective destination of the branch and fetch that instruction. When a cache access requires an entire cycle, and the fetched branch instruction specifies the target address, it is impossible to perform this fetch (of the destination instruction) without delaying the pipeline for at least one pipe stage (one cycle). Conditional branches may cause further delays because they require the calculation of a condition, as well as the target address. The following figure illustrates a delay of one pipeline stage while the instruction at the destination address is calculated and fetched:

Fig. 1-7 Delayed Branch Instructions

Instead of stalling the instruction pipeline to wait for the instruction at the target address, RISC designs typically use an approach similar to that used with Load instructions: Branch instructions are delayed and do not take effect until after one or more instructions immediately following the Branch instruction have been executed. The instruction or instructions in this branch delay slot are always executed, as illustrated in the following Figure.

Fig. 1-8 Branch Delay Slot Execution

Instruction #1 Branch instruction	F	A	M	W		
Instruction #2		F	A	M	W	
Instruction #3			F	A	M	W

With this approach, the inherent delay associated with the branch instructions is made visible to the software, and compilers attempt to fill the branch delay slot with useful instructions. This task is usually not too difficult if there is only a one instruction delay as is the case in the following code example:

Table 1-1 Coding Example

Typical CISC-type code	RISC-type code with delayed branches	RISC-type code with delay slots filled	
A: move s0, a0	A: move s0, a0	A: move s0, a0	
move s1, a1	move s1, a1		
addiu s0, s0, 1	addiu s0, s0, 1	addiu s0, s0, 1	
beq s0, zero, D	beq s0, zero, D	beq s0, zero, D	
	nop (delay slot)	move a1, s1	
B: move a0, s0	B: move a0, s0	move a0, s0	Inst-
move a1, s1	move a1, s1		ruc-
			tion
C: jal x	C: jal x	C: jal x	copy
	nop (delay slot)	move a1, s1	
addiu s0, s0, 1	addiu s0, s0, 1	addiu s0, s0, 1	
bne s0, zero, B	bne s0, zero, B	bne s0, zero, C	
	nop (delay slot)	move a0, s0	

If the branch delay slot cannot be filled with any useful instructions, NOP instructions can be inserted to keep the instruction pipeline filled. Usually, however, a compiler can fill the slot with useful instructions. The preceding example illustrates two different techniques used to fill the slot:

(a) Often, an instruction that occurs before the branch can be executed after the branch without affecting the logic or the Branch instruction itself. Thus, in the example, the move s1, a1 and move a1, s1 instructions can be moved from their original positions to the delay slots without changing the logic of the program.

(b) The original target instruction of the bne instruction was the move a0, s0 instruction at label B:. In the example, this instruction is duplicated in the delay slot following the bne instruction and the target of the bne instruction is changed to be the instruction at label C: Note that while this technique increases the static number of instructions by one, it does not increase the dynamic instruction count: the same number of instructions are executed during the program as in the CISC-type code illustrated in the example.

1.2.3 Time per cycle (T)

The time required to perform a single machine cycle is determined by such factors as:

(1) Instruction decode time.
(2) Instruction operation time.
(3) Instruction access time (memory bandwidth).
(4) Architectural simplicity.

Many of the same design approaches that reduce the number of cycles/instruction also help reduce the time/cycle. For example, dividing up an instruction's execution into several discrete stages to implement the instruction pipeline can also result

in reducing the time required to execute a cycle.

(1) Instruction Decode Time

The time required to decode instructions is partly related to the number of instructions in the instruction set and the variety of instruction formats supported. Thus, simple, uniform RISC instruction sets minimize the instruction decode circuitry and time requirements. For example, if the instruction formats are uniform, with consistent use of bit fields within the instructions, then the processor can decode multiple fields simultaneously to speed the process. In addition to providing instructions only to perform simple tasks, RISC designs also reduce the number of options such as addressing modes to further reduce the number of possible instruction formats.

(2) Instruction Operation Time

For CISC architectures, instruction operation time is usually measured in multiples of cycles. RISC designs, however, strive to make all instructions execute within a single cycle and, further, to make that cycle time as short as possible. Many of the techniques discussed earlier under the category of reducing the number of cycles/instruction also help reduce instruction operation time. For example, the time required for register-to-register operations is much less than the time needed to operate on memory operands. Thus, the load/store architectural approach described earlier also helps reduce the cycle time.

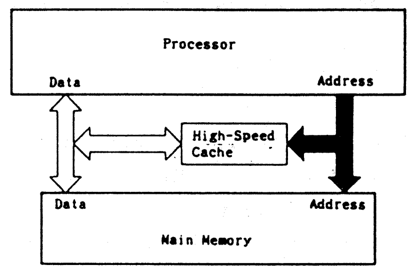
(3) Instruction Access Time (Memory Bandwidth)

The time needed to access (fetch) an instruction is largely a function of the memory system supported and often becomes the limiting factor in RISC-type designs because of the high rate at which instructions can be executed. While the load/store architecture (discussed earlier in this chapter) common to RISC designs helps reduce memory bandwidth requirements, achieving an

execution rate of one cycle/instruction is impossible unless the memory system can deliver instructions at the cycle rate of the processor. A variety of techniques are used to obtain the required memory bandwidth needed to support the high-performance RISC designs. Two common techniques are:

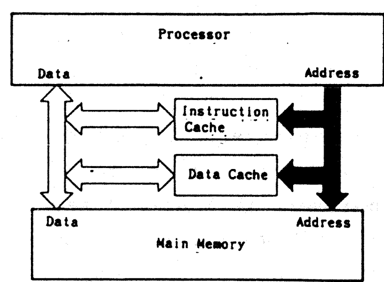
(a) Supporting hierarchical memory systems using high-speed cache memory to provide the primary, re-usable pool of instructions and data that are frequently accessed by the processor. Figure 1-9 illustrates the functional position of cache memory in a hierarchical memory system.

Fig. 1-9 Functional Position of Cache in a Hierarchical Memory System



(b) Supporting separate caches for instructions and data to double the effective cache memory bandwidth. The access time of the cache memory devices can be the factor limiting the processor's throughput; the use of separate caches lets the processor alternate accesses between instruction cache and data cache. Figure 1-10 illustrates a memory system with separate caches for instructions and data.

Fig. 1-10 A V_R3600 System with a High-performance Memory System



The use of separate caches for data and instructions has an additional benefit beyond decreasing the access time: the locality of a set of instructions or a set of data is typically much higher than that of a mixture of instructions and data. Therefore, for most programs, data and instructions held in separate caches are more likely to be re-usable than if a common, shared cache were used.

Another technique that helps minimize the time required to fetch an instruction is to require that all instructions be of a uniform length (a fixed number of bits) and that they always be aligned on a regular boundary. For example, many RISC processors define all instructions to be 32 bits wide and require that they be aligned on word boundaries. This approach eliminates the possibility of a single instruction extending across a word boundary (requiring multiple fetches) or across a memory management boundary (requiring multiple address translations).

(4) Overall Architectural Simplicity

The general simplicity of RISC architectures allows streamlining of the entire machine's organization. As a result, the overhead on each instruction can be reduced and the clock cycle can be shortened, as designers are able to focus on optimizing a small number of critical processor features. The general simplicity of the machine also allows the use of more aggressive semiconductor

process technologies in the manufacture of the processor. More aggressive process technologies provide the potential for faster performance.

1.2.4 Instructions per task (I)

This factor of the performance equation is the one where RISC designs are most vulnerable and has been the source of most of the criticisms that were initially directed at RISC designs. Since RISC processors implement the more complex operations performed by CISC processors using a series of simple instructions, the total number of instructions required to perform a given task tends to increase as the complexity of the instruction set decreases. Therefore, a given program or algorithm written using the instruction set for a RISC processor tends to have more instructions than the same task written using the instruction set for a CISC processor.

However, advances in RISC techniques has done much to mitigate this negative tendency and, for many algorithms, the dynamic instruction count for good RISC processors is not significantly different than for CISC processors. The primary techniques that help reduce the instructions/task factor are:

(1) Optimizing compilers
(2) Operating system support

(1) Optimizing Compilers

Reliance on high-level languages (HLL) has been increasing for many years while the importance of assembly language programming has diminished. This trend has led to an emphasis on the use of efficient compilers to convert HLL instructions to machine instructions. Primary measures of a compiler's efficiency are the compactness of the code it generates and the execution time of that code. Modern, optimizing compilers have evolved to provide great efficiency in the HLL-to-machine language translation.

There is nothing about optimizing compilers that is inherently RISC-oriented; many of the techniques they use were developed before the current generation of RISC architectures arrived and are applied to RISC and CISC machines alike. There is, however, a synergistic relationship between optimizing compilers and RISC architectures — compilers can do their best job of optimization with a RISC architecture, and RISC-type computers, in many cases, rely on compilers to obtain their full performance capabilities.

During the development of more efficient compilers, an analysis of instruction streams revealed that most time was spent executing simple instructions and performing load and store operations — more complex instructions were little used. It was also learned that compilers produced code that was often a narrow subset of a processor's architecture: complex instructions and features were not usable by compilers.

It might seem illogical that people writing compilers would end up ignoring the most powerful instructions and preferring the simpler ones, but it happens because the powerful instructions are hard for a compiler to use or because the instructions don't precisely fit the HLL requirements. A compiler prefers instructions that perform simple, well-defined operations with minimum side-effects. Since these characteristics are typical of a RISC instruction set, there is a natural match between RISC architectures and efficient, optimizing compilers. This match makes it easier for compilers to choose the most effective sequences of a machine's instructions to accomplish the tasks described by a high-level language.

Optimizing Techniques An examination of some of the techniques that compilers use to optimize programs will make the match between compilers and RISC architectures more apparent.

(a) Register allocation

The compiler allocates processor registers to hold frequently used data and thus reduce the number of load/store operations. The following simple example illustrates how careful register allocation can reduce the number of instructions required to perform a task:

```
# task is A:= B+C
  Load   R1,B
  Load   R2,C
  Add    R3,R1,R2
  Store  R3,A
# If A, B, and C are allocated to registers
  Add    Ra,Rb,Rc
```

In this simple example, the two Load instructions are eliminated since the required values are already available in registers and the Store instruction is not needed since the compiler will hold the result of the Add in a register for future use.

(b) Redundancy elimination

The compiler looks for opportunities to re-use results and thus eliminate redundant computations.

(c) Loop optimization

A compiler optimizes loop operations by recognizing variables and expressions that don't change during a loop and moving them outside the loop.

(d) Replace slow operations with faster ones

A compiler searches for situations where slow operations, such as special case of a multiply or divide, can be replaced with faster operations, such as shift and add instructions.

(e) Strength reduction

This technique consists of replacing "expensive" operations with cheaper ones. For example, multi-dimension arrays are often indexed using a combination of several multiplication and addition operations. Strength reduction might simplify the index calculation by using a previously calculated address and a simple addition operation.

(f) Pipeline scheduling

The compiler schedules and reorganizes instructions to ensure that pipeline delay slots are filled with useful instructions as illustrated earlier in the description of load and branch delays.

Again note that none of the techniques described above are uniquely linked to RISC architectures. However, the simplicity of a RISC machine makes it inherently easier for a compiler to discover optimization opportunities and implement these optimizations with a clear view of their effects.

Optimization Levels The development of optimizing compilers has produced its own terminology. This section describes the terms commonly used to categorize the various levels of optimization performed by compilers. The optimization techniques used can be divided into four levels according to their scope and degree of difficulty:

(a) Peephole optimization

Peephole optimization attempts to make improvements in code size or performance within a narrow context. An example of this level is replacing slower operations with faster ones.

(b) Local optimization

Local optimization makes decisions based on views of multiple-instruction sequences. An example of this level is to examine sequences of instructions to determine the best prologue/epilogue to use as the entry/exit code for a

function. Other examples include keeping values in registers over short periods of time, and eliminating branch instructions whose target is another branch instruction.

(c) Global optimization

Global optimization optimizes program control flow by enhancing branch and loop structures and by performing strength reduction.

(d) Inter-procedural optimization

This level is rarely performed because techniques like the following are just being developed:

- Allocating registers to maximize their life between procedures.

- Merging procedures and converting appropriate procedures to in-line code to reduce overhead.

(2) Operating System Support

The performance gains obtained by providing support for operating systems are often subtle and not as easily defined or measured as with some of the other RISC techniques. While CISC architectures typically provide elaborate support for operating systems, the RISC approach emphasizes appropriate support. The appropriateness is based on a rigorous evaluation of the performance gains that can be obtained by the support of any particular function. The guiding principles are to avoid unnecessary complexity unless justified by statistics of actual usage, and to simplify and streamline operations required most frequently by operating systems.

The learning path here parallels the one traveled during exploration of compiler efficiencies — trying to put features supporting high-level languages into hardware often frustrated compiler writers. Similarly, putting special features into

hardware to support operating systems does not always match the real needs of operating systems. With compilers, it was learned that the special instructions intended to simplify support of high-level languages were not often used by compilers. Similarly, it has been found that special hardware features for operating systems may also miss their mark. Often, the most efficient way of supporting an operating system is to just provide it with raw speed and with simple, minimal controls.

The paragraphs that follow illustrate some of the subtle ways in which RISC-type designs can supply appropriate operating system support to enhance performance without adding unacceptable complexity to the hardware:

(a) Virtual Memory System

Translation Lookaside Buffers (TLBs) provide the virtual-to-physical address translation that is essential to implementing a powerful operating system. While there is nothing about TLBs that is RISC-specific, the chip area gained by overall simplification of the processor can be used to implement (larger) on-chip TLBs. An on-chip TLB enhances performance by eliminating the cycle(s) otherwise required to transfer the virtual address to an external TLB.

(b) Modes and Protection

Operating systems require some mechanisms for controlling user access to system and processor resources. CISC processors often provide a variety of operating modes and protection mechanisms. Multiple modes and protection schemes add complexity to the hardware, however, and experience teaches that there is seldom a complete match between these mechanisms and operating system requirements. The RISC approach is to supply limited control and protection mechanisms: a simple kernel/privileged, user/unprivileged mode differentiation is usually sufficient. More elaborate schemes can then be implemented as needed in the kernel software.

(c) Interrupts and Traps

Many CISC processors provide extensive hardware support for responding to interrupts and traps by saving a lot of state information and by generating numerous vector addresses to which control is transferred in response to exceptions. This support adds complexity to the hardware but does not necessarily produce corresponding simplification of the operating system's tasks. For example, many operating systems do not really need or use numerous distinct exception vector addresses: instead, they first execute a common interrupt handler which then does the work to determine the specific processing needed for the exception. The operating system itself might then determine what state information (if any) needs to be saved. This approach results in simplified hardware and lets the appropriate complexity be provided by the operating system as needed.

(d) Special-function Instructions

Note that we have made no mention of special instructions to simplify and support operating system activities. Once again, the rule of simplicity and appropriateness argues against the inclusion of special instructions. Even in cases where significant time is spent in an operating system, the bulk of the time is spent executing general code rather than performing special functions. Thus, it is more efficient to let the operating system use the standard, simple, non-specialized instructions to perform all of its functions.

1.3 Hidden Benefits of RISC Design

Some of the important benefits that results from the RISC design techniques are not attributable to the architectural characteristics adopted to enhance performance but are a result of the overall reduction in complexity: the simpler design allows both chip-area resources and human resources to be applied to features that enhance performance.

(1) Shorter Design Cycle

The simplified architectures of RISC processors can be implemented more quickly: it is much easier to implement and debug a streamlined, simplified architecture with no microcode than a complex, microcoded architecture. CISC processors have such a long design cycle that they are often not fully debugged until the technology in which they were designed is obsolete. The shorter time required to design and implement RISC processors lets them make use of the best available technologies.

(2) Smaller Chip Size

The simplicity of RISC processors also frees scarce chip-area resources for performance-critical structures like larger register files, translation-lookaside-buffers (TLB's) coprocessors, and fast multiply-divide units. These additional resources help these processors obtain an even greater performance edge.

(3) User (Programmer) Benefits

Somewhat surprisingly, simplicity in architecture also helps the user:

(a) The uniform instruction set is easier to use.

(b) There is a closer correlation between instruction count and cycle count making it much easier to measure the

true impact of code optimization activities.

(c) Programmers can have a higher confidence in hardware correctness.

1.4 Rules of Description

Before getting in touch with the main subject, this section describes the rules of description and figure representation in this manual.

1.4.1 Representation of numeric values

Numeric values are treated in the text as follows:

Binary xxxx
Decimal xxxx
Hexadecimal ..... xxxxH

For example, the numeric value 118 is in decimal notation, which is represented as 0111 0110 in binary notation, and as 76H in hexadecimal notation.

Remarks: K, M, and G of K bytes, M bytes, and G bytes are the powers of 2 and respectively mean kilc, mega, and giga.

1.4.2 Representation of data

The data in memory is basically in units of 8 bits or byte. The position of a byte in the main memory can be specified by an address. To represent data exceeding 1 byte, the information can be placed over several contiguous bytes.
These contiguous bytes must have equally contiguous addresses. The term "word" means contiguous 4 bytes (32 bits). Therefore, the term "half word" means contiguous 2 bytes (16 bits), and "double word" means contiguous 8 bytes (64 bits).

(1) Representation of byte

Generally, the 8 bits of 1 byte are assigned serial numbers from the right toward the left, starting from 0. If this byte represents an integer such as 118 (=0111 0110B=76H), the bit at the rightmost position (i.e., bit 0) is called the least significant bit (LSB) and the bit at the leftmost position (bit 7) is referred to as "the most significant bit (MSB)".

Fig. 1-11 Bit Position in Byte (Bit Number)

7	6	5	4	3	2	1	0

The value of each bit is as shown in Fig. 1-12 when the numeric value 118 is set in this byte.

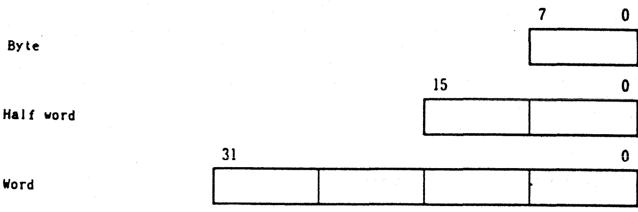
Fig. 1-12 Byte with Numeric Value Set

7							0
0	1	1	1	0	1	1	0

(2) Representation of Plural Bytes

More than one bytes, e.g., x bytes are represented as shown in Fig. 1-13. Note, however, that the byte sequence differs depending on the endianness. In the case of the big endian, the most significant byte is placed at the rightmost position and the least significant byte is placed at the leftmost position. In the case of the little endian, the most significant byte is placed at the leftmost position and the least significant bit is placed at the rightmost position.
For details, refer to CHAPTER 2 V_R3600 CPU ARCHITECTURE.

Fig. 1-13 Representation of Plural Bytes (Horizontal)



(3) Representation of Bit

A bit is represented by a bit number. To represent several bits (or a bit string) in data, an expression such as "bits 5-3" is used to indicate 3-bit data of bits 3 through 5.
In the byte that stores the numeric value 118 above, bits 5-3 are '110' in binary notation.

(4) Representation of Register

A register is a location in which data different from those of the memory can be stored. To show a register in a figure, the bit 0 of the register is shown at the rightmost position.
Figure 1-14 shows a 32-bit register. When data is loaded from the memory to this register, the data is stored in the register in the same image as in the memory. Likewise, when data is stored from the register to the memory, the data is stored in the same image as in the register.

Fig. 1-14 Representation of 32-bit Register

31 24 23 16 15 8 7 0

1.4.3 Representation of memory

(1) Address

The position of a byte is treated as an address in the memory and address space.

(2) Representation of memory

To express the memory or its addresses in figure, the lower addresses are in principle shown on the lower part of the figure. Therefore, the lowest part of the figure indicates the position of the least significant address, and the higher the position on the figure, the higher the address.

(3) Data in memory

If the address of data is a multiple of 2, 4, or 8, it is said that the data is at the boundary of half word, word, or double word.

1.4.4 Terms

The special terms used throughout this manual are as follows:

RFU: Field reserved by NEC. Any data written to this field is ignored. When this field is read, zero is returned.

*: RFU: Reserved for Future Use

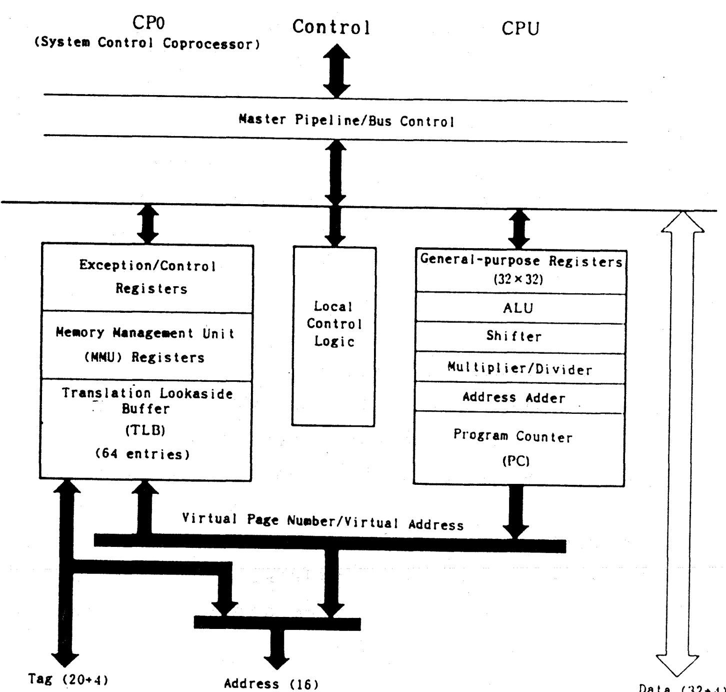
As additional or supplementary information, the following description may be provided:

*: Footnote
Note: Important information
Remarks: Supplement

Chapter 2 V_R3600 CPU Architecture

The V_R3600 Processor consists of two tightly-coupled processors implemented on a single chip. The first processor is a full 32-bit RISC CPU. The second processor is a system control coprocessor (CP0), containing a TLB (Translation Lookaside Buffer) and control registers to support a virtual memory subsystem and separate caches for instructions and data. Figure 2-1 shows the functions incorporated within the V_R3600.

Fig. 2-1 V_R3600 CPU Functional Block Diagram



2.1 V_R3600 Processor Features

- Software compatible with R3000A developed by MIPS Computer Systems, Inc.

- Full 32-bit Operation
  The V_R3600 contains thirty-two 32-bit registers, and all instructions and addresses are 32 bits.

- Efficient Pipelining
  The CPU's 5-stage pipeline design assists in obtaining an execution rate approaching one instruction per cycle. Pipeline stalls and exceptional events are handled precisely and efficiently.

- On-chip Cache Control
  The V_R3600 provides a high-bandwidth memory interface that handles separate external Instruction and Data caches ranging in size from 4 to 64 Kbytes each. Both caches are accessed during a single CPU cycle. All cache control logic is on chip.

- On-chip Memory Management Unit
  A fully-associative, 64-entry Translation Lookaside Buffer (TLB) provides fast address translation for virtual-to-physical memory mapping of the 4-Gbyte virtual address space.

- Coprocessor Interface
  The V_R3600 generates all addresses and handles memory interface control for up to three additional tightly-coupled external coprocessors.

2.2 V_R3600 CPU Registers

The V_R3600 CPU provides 32 general purpose 32-bit registers, a 32-bit Program Counter, and two 32-bit registers that hold the results of integer multiply and divide operations. The CPU registers are shown in Figure 2-2 and are described in detail later in this chapter. Note that there is no Program Status Word (PSW) register shown in this figure: the functions traditionally provided by a PSW register are instead provided in the Status and Cause registers incorporated within the System Control Coprocessor (CP0).

Fig. 2-2 V_R3600 CPU Registers

General Registers

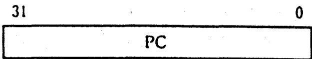
31	0
r0	
r1	
r2	
⋮	
r29	
r30	
r31	

Multiply/Divide Registers

31	0
HI	

31	0
LO	

Program Counter



2.3 Instruction Set Overview

All V_R3600 instructions are 32 bits long and there are only three instruction formats as shown in Figure 2-3. This approach simplifies instruction decoding. More complicated (and less frequently used) operations and addressing modes can be synthesized by compiler using sequences of simple instructions.

Fig. 2-3 V_R3600 Instruction Formats

I-Type (Immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-Type (Jump)

R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

Remarks: The variable subfields are as follows:

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

The V_R3600 instruction set can be divided into the following groups:

(1) Load/Store instruction

Load/Store instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is base register plus 16-bit, signed immediate offset.

(2) Computational instruction

Computational instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands and the result are registers) and I-type (one operand is a 16-bit immediate) formats.

(3) Jump and Branch instruction

Jump and Branch instructions change the control flow of a program. Jumps are always to a paged absolute address formed by combining a 26-bit target with four bits of the Program Counter (J-type format, for subroutine calls) or 32-bit register addresses (R-type, for returns and dispatches). Branches have 16-bit offsets relative to the program counter (I-type). Jump and Link instructions save a return address in Register 31.

(4) Coprocessor instruction

Coprocessor instructions perform operations in the coprocessors. Coprocessor Loads and Stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPA ARCHITECTURE in CHAPTER 7)

(5) Coprocessor 0 instruction

Coprocessor 0 instructions perform operations on the System Control Coprocessor (CP0) registers to manipulate the memory management and exception handling facilities of the processor.

(6) Special instruction

Special instructions perform a variety of tasks, including movement of data between special and general registers, system calls, and breakpoint. They are always R-type.

Table 2-1 lists the instruction set of the V_R3600 Processor. A more detailed summary is provided in CHAPTER 3 INSTRUCTION SET SUMMARY and a complete description of each instruction is contained in CHAPTER 7 V_R3600 PROCESSOR INSTRUCTION SET DETAILS.

Table 2-1 V

op	Description	op	Description
Load/Store Instructions		**Shift Instructions**	
LB	Load Byte	SRL	Shift Right Logical
LBU	Load Byte Unsigned	SRA	Shift Right Arithmetic
LH	Load Halfword	SLLV	Shift Left Logical Variable
LHU	Load Halfword Unsigned	SRLV	Shift Right Logical Variable
LW	Load Word	SRAV	Shift Right Arithmetic Variable
LWL	Load Word Left	**Special Instruction**	
LWR	Load Word Right	SYSCALL	System Call
SB	Store Byte	BREAK	Break
SH	Store Halfword	**Multiply/Divide Instructions**	
SW	Store Word	MULT	Multiply
SWL	Store Word Left	MULTU	Multiply Unsigned
SWR	Store Word Right	DIV	Divide
Arithmetic Instructions (ALU Immediate)		DIVU	Divide Unsigned
ADDI	Add Immediate	MFHI	Move From HI
ADDIU	Add Immediate Unsigned	MTHI	Move To HI
SLTI	Set on Less Than Immediate	MFLO	Move From LO
SLTIU	Set on Less Than Immediate	MTLO	Move To LO
	Unsigned	**Jump/Branch Instructions**	
ANDI	AND Immediate	J	Jump
ORI	OR Immediate	JAL	Jump And Link
XORI	Exclusive OR Immediate	JR	Jump to Register
LUI	Load Upper Immediate	JALR	Jump And Link Register
Arithmetic Instructions (3-operand, register-type)		BEQ	Branch on Equal
ADD	Add	BNE	Branch on Not Equal
ADDU	Add Unsigned	BLEZ	Branch on Less than or Equal to
SUB	Subtract		Zero
SUBU	Subtract Unsigned	BGTZ	Branch on Greater Than Zero
SLT	Set on Less Than	BLTZ	Branch on Less Than Zero
SLTU	Set on Less Than Unsigned	BGEZ	Branch on Greater Than or Equal
AND	AND		to Zero
OR	OR	BLTZAL	Branch on Less Than Zero And
XOR	Exclusive OR		Link
NOR	NOR	BGEZAL	Branch on Greater than or Equal
Shift Instructions			to Zero And Link
SLL	Shift Left Logical		

Table 2-1 V_R3600 Instruction Summary (2/2)

op	Description	op	Description
Coprocessor Instructions		**System Control Coprocessor (CP0) Instructions**	
LWCz	Load Word from Coprocessor	MTC0	Move to CP0
SWCz	Store Word to Coprocessor	MFC0	Move From CP0
MTCz	Move To Coprocessor	TLBR	Read indexed TLB entry
MFCz	Move From Coprocessor	TLBWI	Write Indexed TLB entry
CTCz	Move Control to Coprocessor	TLBWR	Write Random TLB entry
CFCz	Move Control From Coprocessor	TLBP	Probe TLB for matching entry
COPz	Coprocessor Operation	RFE	Restore From Exception
BCzT	Branch on Coprocessor z True		
BCzF	Branch on Coprocessor z False		

2.4 V_R3600 Processor Programming Model

This section describes organization of data in registers and in memory and the set of general registers available. It also gives a summary description of all the V_R3600 CPU registers.

2.4.1 Data formats and addressing

The V_R3600 defines a 32-bit word, a 16-bit half word and an 8-bit byte. The byte ordering is configurable (configuration occurs during hardware reset) into either big-endian or little-endian byte ordering:

- When configured as a big-endian system, byte 0 is always the most significant (leftmost) byte.

- When configured as a little-endian system, byte 0 is always the least significant (rightmost) byte.

Figure 2-4 shows the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

Fig. 2-4 Addresses of Bytes within Words

(a) Big Endian

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

. Most significant byte is at lowest address.
. Word is addressed by byte address of most significant byte.

(b) Little Endian

Higher Address	31 ... 24	23 ... 16	15 ... 8	7 ... 0	Word Address
	11	10	9	8	8
	7	6	5	4	4
Lower Address	3	2	1	0	0

. Least significant byte is at lowest address.
. Word is addressed by byte address of least significant byte.

The V_R3600 uses byte addressing, with alignment constraints, for half word and word accesses; half word accesses must be aligned on an even type boundary and word accesses must be aligned on a byte boundary divisible by four.

As shown in Figure 2-4, the address of a multiple-byte data item is the address of the most-significant byte on a big-endian configuration, and is the address of the least-significant byte on a little-endian configuration.

Special instructions are provided for addressing words that are not aligned on 4-byte (word) boundaries (Load/Store-Word-Left/Right; LWL, LWR, SWL, SWR). These instructions are used in pairs to provide addressing of misaligned words with one additional instruction cycle over that required for aligned words. Figure 2-5 shows the bytes accessed when addressing a misaligned word with a byte address of 3 for each of the two conventions.

Fig. 2-5 Misaligned Word: Byte Addresses

Big Endian

	31 ... 24	23 ... 16	15 ... 8	7 ... 0
Higher Address	4	5	6	
↑ Lower Address				3

Little Endian

	31 ... 24	23 ... 16	15 ... 8	7 ... 0
Higher Address		6	5	4
↑ Lower Address	3			

2.4.2 V_R3600 CPU general registers

Figure 2-6 shows the V_R3600 CPU registers. There are 32 general registers, each consisting of a single word (32 bits). The 32 general registers are treated symmetrically, with two exceptions: r0 is hardwired to a zero value, and r31 is the link register for Jump And Link instructions.

Register r0 may be specified as a target register for any instruction when the result of the operation is discarded. The register maintains a value of zero under all conditions when used as a source register.

The two Multiply/Divide registers (HI, LO) store the double-word, 64-bit result of multiply operations and the quotient and remainder of divide operations.

Note: In addition to the CPU's general registers, the system control coprocessor (CP0) has a number of special purpose registers that are used in conjunction with the memory management system and during exception processing. Refer to CHAPTER 5 MEMORY MANAGEMENT SYSTEM for a description of the memory management registers and to CHAPTER 6 EXCEPTION PROCESSING for a discussion of the exception handling registers.

Fig. 2-6 V_R3600 CPU Registers

General-Purpose Registers

31 … 0
r0 (=0)
r1
r2
. . .
r29
r30
r31

Multiply/Divide Registers

31 … 0
HI
LO

Program Counter

31 … 0
PC

2.5 V_R3600 System Control Coprocessor (CP0)

The V_R3600 can operate with up to four tightly-coupled coprocessors (designated CP0 through CP3). The System Control Coprocessor (or CP0), is incorporated on the V_R3600 chip and supports the virtual memory system and exception handling functions of the V_R3600. The virtual memory system is implemented using a Translation Lookaside Buffer and a group of programmable registers as shown in Figure 2-7.

Fig. 2-7 The CP0 Registers

Memory Management

Entry HI Register | Entry LO Register

63
TLB
0

Index Register
Random Register

Exception Handling

Status Register | Cause Register

Context Register | EPC Register

BadVAddr Register | PRId Register

The CP0 registers shown in Figure 2-7 are used to manipulate the memory management and exception handling capabilities of the V_R3600. Table 2-2 provides a brief description of each register.

Table 2-2 System Control Coprocessor (CP0) Registers

No.	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudo-random pointer into TLB array
2	EntryLo	Low half of a TLB entry
4	Context	Pointer into kernel's virtual Page Table Entry array
8	BadVAddr	Most recent bad virtual address
10	EntryHi	High half of a TLB entry
12	SR	Mode, interrupt enables, and diagnostic status info
13	Cause	Indicates nature of last exception
14	EPC	Exception Program Counter
15	PRid	Processor revision identification

2.6 V_R3600 Pipeline Architecture

The execution of a single V_R3600 instruction consists of five primary steps:

(1) IF — Fetch the instruction (I-Cache).
(2) RD — Read any required operands from CPU registers while decoding the instruction.
(3) ALU — Perform the required operation on instruction operands.
(4) MEM — Access memory (D-Cache).
(5) WB — Write back results to register file.

Each of these steps requires approximately one CPU cycle as shown in Figure 2-8 (parts of some operations lap over into another cycle while other operations require only 1/2 cycle).

Fig. 2-8 Instruction Execution Sequence

Instruction Execution

IF	RD		ALU	MEM	WB
I-Cache		RF	OP	D-Cache	WB
			←one cycle→		

The V_R3600 uses a 5-stage pipeline to achieve an instruction execution rate approaching one instruction per CPU cycle. Thus, execution of five instructions at a time are overlapped as shown in Figure 2-9.

Fig. 2-9 V_R3600 Instruction Pipeline (5-deep)

IF	RD	ALU	MEM	WB				
	IF	RD	ALU	MEM	WB			
		IF	RD	ALU	MEM	WB		
			IF	RD	ALU	MEM	WB	
				IF	RD	ALU	MEM	WB
				Current CPU Cycle				

Instruction Flow
↓

This pipeline operates efficiently because different CPU resources (address and data bus accesses, ALU operations, register accesses, and so on) are utilized on a non-interfering basis. Refer to CHAPTER 3 INSTRUCTION SET SUMMARY for a detailed discussion of the instruction pipeline.

2.7 Memory Management System

The V_R3600 has an addressing range of 4 Gbytes. However, since most V_R3600 systems implement a physical memory smaller than 4 Gbytes, the V_R3600 provides for the logical expansion of memory space by translating addresses composed in a large virtual address space into available physical memory addresses. The 4 Gbyte address space is divided into 2 Gbytes for users and 2 Gbytes for the kernel.

2.7.1 The TLB (Translation Lookaside Buffer)

Virtual memory mapping is assisted by the Translation Lookaside Buffer (TLB). The on-chip TLB provides very fast virtual memory access and is well-matched to the requirements of multi-tasking operating systems. The fully-associative TLB contains 64 entries, each of which maps a 4-Kbyte page, with controls for read/write access, cacheability, and process identification. The TLB allows each user to access up to 2 Gbytes of virtual address space.

2.7.2 V_R3600 operating modes

The V_R3600 has two operating modes: User mode and Kernel mode. The V_R3600 normally operates in the User mode until an exception is detected forcing it into the Kernel mode. It remains in the Kernel mode until a Restore From Exception (RFE) instruction is executed. The manner in which memory addresses are translated or mapped depends on the operating mode of the V_R3600. Figure 2-10 shows the virtual address space for the two operating modes.

Fig. 2-10 V_R3600 Virtual Addressing

User Mode

Address	Segment	Size / Mapping
7FFF FFFFH – 0	kuseg	2 GB TLB Mapped

Kernel Mode

Address	Segment	Size / Mapping
FFFF FFFFH – C000 0000H	kseg2	1 GB TLB Mapped
A000 0000H	kseg1	0.5 GB TLB Unmapped
8000 0000H	kseg0	0.5 GB TLB Unmapped
7FFF FFFFH	kuseg	2 GB TLB Mapped

Remarks: kseg means Kernel segment and kuseg means Kernel User segment.

(1) User Mode

In this mode, a single, uniform virtual address space (kuseg) of 2 Gbyte is available. Each virtual address is extended with a 6-bit process identifier field to form unique virtual addresses for up to 64 user processes. All references to this segment are mapped through the TLB. Use of the cache is determined by bit settings for each page within the TLB entries.

(2) Kernel Mode

Four separate segments are defined in this mode:

(a) kuseg

When in the Kernel mode, references to this segment are treated just like User mode references, thus streamlining kernel access to user data

(b) kseg0

References to this 512-Mbyte segment use cache memory but are not mapped through the TLB. Instead, they always map to the first 0.5 Gbytes of physical memory.

(c) kseg1

References to this 512-Mbyte segment are not mapped through the TLB and do not use the cache. Instead, they are hard-mapped into the same 0.5-Gbyte segment of physical memory space as kseg0.

(d) kseg2

References to this 1-Gbyte segment are always mapped through the TLB, and use of the cache is determined by bit settings within the TLB entries.

2.8 Memory System Hierarchy

The high performance capabilities of the V_R3600 Processor demand system configurations incorporating techniques frequently employed in large, mainframe computers but seldom encountered in systems based on more traditional microprocessors.

A primary goal of RISC machines is to achieve an instruction execution rate of one instruction per CPU cycle. The V_R3600 approaches this goal by means of a compact and uniform instruction set, a deep instruction pipeline (as described above), and careful adaptation to optimizing compilers. Many of the advantages obtained from these techniques can, however, be negated by an inefficient memory system.

Figure 2-11 illustrates memory in a simple microprocessor system. In this system, the CPU outputs addresses to memory and reads instructions and data from memory or writes data to memory. The memory space is completely undifferentiated: instructions, data, and I/O devices are all treated the same. In such a system, a primary limiting performance factor is memory bandwidth.

Fig. 2-11 A Simple Microprocessor Memory System

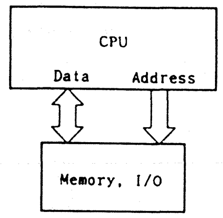
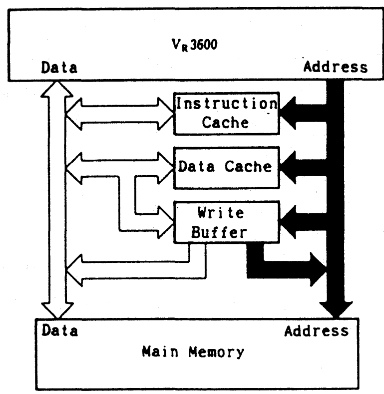


Figure 2-12 illustrates a memory system that supports the significantly greater memory bandwidth required to take full advantage of the V_R3600's performance capabilities. The key features of this system are:

Fig. 2-12 V_R3600 System with a High-Performance Memory System



(1) External Cache Memory

Local, high-speed memory (called cache memory) is used to hold instructions and data that is repetitively accessed by the CPU (for example, within a program loop) and thus reduces the number of references that must be made to the slower speed main memory. Some microprocessors provide a limited amount of cache memory on the CPU chip itself. The external caches supported by the V_R3600 can be much larger; while a small cache can improve performance of some programs, significant improvements for a wide range of programs require large caches.

(2) Separate Caches for Data and Instructions

Even with high-speed caches, memory speed can still be a limiting factor because of the fast cycle time of a high-performance microprocessor. The V_R3600 supports separate caches for instructions and data and alternates accesses of the two caches during each CPU cycle. Thus, the processor can obtain data and instructions at the cycle rate of the CPU using caches constructed with commercially available static RAM devices.

(3) Write Buffer

In order to ensure data consistency, all data that is written to the data cache must also be written out to main memory. To relieve the CPU of this responsibility (and the inherent performance burden) the V_R3600 supports an interface to a write buffer. The uPD31311 Write Buffer captures data and addresses output by the CPU and ensures that the data is passed on to main memory.

Chapter 3 V_R600 Instruction Set Summary

This chapter provides an overview of the V_R3600 instruction set by presenting each category of instructions in a tabular summary form. It also provides a detailed discussion of the instruction pipeline. Refer to CHAPTER 7 V_R3600 PROCESSOR INSTRUCTION SET DETAILS for a detailed description of each instruction.

3.1 Instruction Formats

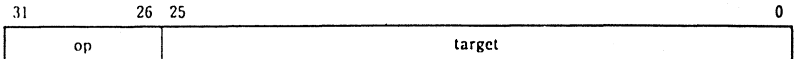
Every V_R3600 instruction consists of a single word (32 bits) aligned on a word boundary. There are only three instruction formats as shown in Figure 3-1. This approach simplifies instruction decoding. More complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler.

Fig. 3-1 V_R3600 Instruction Formats

I-Type (Immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-Type (Jump)



R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

Remarks: The variable subfields are as follows:

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

3.2 Instruction Notational Conventions

In this manual, all variable sub-fields in an instruction format (such as rs, rt, immediate, and so on) are shown in lower-case names.

For the sake of clarity, an alias is sometimes used for a variable sub-field in the formats of specific instructions. For example, rs = base in the format for Load and Store instructions. Such an alias is always lower case, since it refers to a variable sub-field.

Instruction opcodes are shown in all upper case.

The actual bit encoding for all the mnemonics is specified at the end of CHAPTER 7.

3.3 Load and Store Instructions

Load/Store instructions move data between memory and general registers. They are all I-type instructions. The only addressing mode directly supported is base register plus 16-bit signed immediate offset.

All load operations have a latency of one instruction. That is, the data being loaded from memory into a register is not available to the instruction that immediately follows the load instruction: the data is available to the second instruction after the load instruction. An exception is the target register for the "load word left" and "load word right" instructions, which may be specified as the same register used as the destination of a load instruction that immediately precedes it. (Refer to CHAPTER 4 V_R3600 INSTRUCTION PIPELINE for a detailed discussion of load instruction latency.)

The Load/Store instruction opcode determines the access type which indicates the size of the data item to be loaded or stored as shown in Table 3-2. Regardless of access type or byte-numbering order (endian-ness), the address specifies the byte which has the smallest byte address of all the bytes in the addressed field. For a beg-endian machine, this is the most significant byte; for a little-endian machine, this is the least significant byte.

The bytes within the addressed word that are used can be determined directly from the access type and the two low-order bits of the address, as shown in Table 3-2. Note that certain combinations of access type and low-order address bits can never occur: only the combinations shown in Table 3-1 are permissible.

Table 3-1 Byte Specifications for Loads/Stores

Acess Type (1 0)	Low-Order Address Bits (1 0)	Low-Order Address Bits: Big-Endian (31)			(0)	Little-Endian (31)			(0)
1 1 (word)	0 0	0	1	2	3	3	2	1	0
1 0 triple-byte	0 0	0	1	2			2	1	0
	0 1		1	2	3	3	2	1	
0 1 (halfword)	0 0	0	1					1	0
	1 0			2	3	3	2		
0 0 (byte)	0 0	0							0
	0 1		1					1	
	1 0			2			2		
	1 1				3	3			

Table 3-2 and 3-3 summarize the V_R3600 Load and Store instructions.

Table 3-2 Load Instruction Summary

Instruction	Format and Description (op / base / rt / offset)
Load Byte	LB rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Sign-extend contents of addressed byte and load into rt.
Load Byte Unsigned	LBU rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Zero-extend contents of addressed byte and load into rt.
Load Halfword	LH rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Sign-extend contents of addressed halfword and load into rt.
Load Halfword Unsigned	LHU rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address Zero-extend contents of addressed halfword and load into rt.
Load Word	LW rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Load contents of addressed word into register rt.
Load Word Left	LWL rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift addressed word left so that addressed byte is leftmost byte of a word. Merge bytes from memory with contents of register rt and load result into register rt.
Load Word Right	LWR rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift addressed word right so that addressed byte is rightmost byte of a word, into register rt.

Table 3-3 Store Instruction Summary

Instruction	Format and Description (op | base | rt | offset)
Store Byte	SB rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant byte of register rt at addressed location.
Store Halfword	SH rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant halfword of register rt at address location.
Store Word	SW rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant word of register rt at addressed location.
Store Word Left	SWL rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift contents of register rt left so that leftmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.
Store Word Right	SWR rt, offset(base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift contents of register rt right so that leftmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.

3.4 Computational Instructions

Computational instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats. There are four categories of computational instructions:

- ALU Immediate instructions are summarized in Table 3-4.
- 3-Operand Register-Type instructions are summarized in Table 3-5.
- Shift instructions are summarized in Table 3-6.
- Multiply/Divide instructions are summarized in Table 3-7.

Table 3-4 ALU Immediate Instruction Summary

Instruction	Format and Description
ADD Immediate	ADDI rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt. Trap on two's complement overflow.
ADD Immediate Unsigned	ADDIU rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt. Do not trap on overflow.
Set on Less Than Immediate	SLTI rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as signed 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt.
Set on Less Than Unsigned Immediate	SLTIU rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as unsigned 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt.
AND Immediate	ANDI rt, rs, immediate Zero-extend 16-bit immediate, AND with contents of register rs and place result in register rt.
OR Immediate	ORI rt, rs, immediate Zero-extend 16-bit immediate, OR with contents of register rs and place result in register rt.
Exclusive OR Immediate	XORI rt, rs, immediate Zero-extend 16-bit immediate, exclusive OR with contents of register rs and place result in register rt.
Load Upper Immediate	LUI rt, immediate Shift 16-bit immediate left 16 bits. Set least significant 16 bits of word to zeroes. Store result in register rt.

Table 3-5 3-Operand Register-Type Instruction Summary

Instruction	Format and Description [illegible] [illegible] rt rd [illegible]
Add	ADD rd, rs, rt Add contents of registers rs and rt and place 32-bit result in register rd. Trap on two's complement overflow.
Add Unsigned	ADDU rd, rs, rt Add contents of registers rs and rt and place 32-bit result in register rd. Do not trap on overflow.
Subtract	SUB rd, rs, rt Subtract contents of registers rt from rs and place 32-bit result in register rd. Trap on two's complement overflow.
Subtract Unsigned	SUBU rd, rs, rt Subtract contents of registers rt from rs and place 32-bit result in register rd. Do not trap on overflow.
Set on Less Than	SLT rd, rs, rt Compare contents of register rt to register rs (as signed 32-bit integers). If register rs is less than rt, result = 1; otherwise, result = 0.
Set on Less Than Unsigned	SLTU rd, rs, rt Compare contents of register rt to register rs (as unsigned 32-bit integers). If register rs is less than rt, result = 1; otherwise, result = 0.
AND	AND rd, rs, rt Bitwise AND contents of registers rs and rt and place result in register rd.
OR	OR rd, rs, rt Bitwise OR contents of registers rs and rt and place result in register rd.
Exclusive OR	XOR rd, rs, rt Bitwise exclusive OR contents of registers rs and rt and place result in register rd.
NOR	NOR rd, rs, rt Bitwise NOR contents of registers rs and rt and place result in register rd.

Table 3-6 Shift Instruction Summary

(a) SLL, SRL, SRA

Instruction	Format and Description (op | 0 | rt | rd | sa | funct)
Shift Left Logical	SLL rd, rt, sa Shift contents of register rt left by sa bits, inserting zeroes into low order bits. Place 32-bit result in register rd.
Shift Right Logical	SRL rd, rt, sa Shift contents of register rt right by sa bits, inserting zeroes into high order bits. Place 32-bit result in register rd.
Shift Right Arithmetic	SRA rd, rt, sa Shift contents of register rt right by sa bits, sign-extending the high order bits. Place 32-bit result in register rd.

(b) SLLV, SRLV, SRAV

Instruction	Format and Description (op | rs | rt | rd | 0 | funct)
Shift Left Logical Variable	SLLV rd, rt, rs Shift contents of register rt left. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into low order bits of rt and place 32-bit result in register rd.
Shift Right Logical Variable	SRLV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into high order bits of rt and place 32-bit result in register rd.
Shift Right Arithmetic Variable	SRAV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Sign-extend the high order bits of rt and place 32-bit result in register rd.

Table 3-7 Multiply/Divide Instruction Summary

(a) MULT, MULTU, DIV, DIVU

Instruction	Format and Description (op | rs | rt | 0 | funct)
Multiply	MULT rs, rt Multiply contents of registers rs and rt as twos complement values. Place 64-bit result in special registers HI/LO.
Multiply Unsigned	MULTU rs, rt Multiply contents of registers rs and rt as unsigned values. Place 64-bit result in special registers HI/LO.
Divide	DIV rs, rt Divide contents of register rs by rt treating operands as twos complements values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI.
Divide Unsigned	DIVU rs, rt Divide contents of register rs by rt treating operands as unsigned values. Place the 32-bit quotient in special register LO, and the 32-bit remainder in HI.

(b) MFHI, MFLO

Instruction	Format and Description (op | 0 | rd | 0 | funct)
Move From HI	MFHI rd Move contents of special register HI to register rd.
Move From LO	MFLO rd Move contents of special register LO to register rd.

(c) MTHI, MTLO

Instruction	Format and Description (op, [illegible], 0, funct)
Move To HI	MTHI rd Move contents of register rd to special register HI.
Move To LO	MTLO rd Move contents of register rd to special register LO.

3.5 Jump and Branch Instructions

Jump and Branch instructions change the control flow of a program. All Jump and Branch instructions occur with a one instruction delay: that is, the instruction immediately following the jump or branch is always executed while the target instruction is being fetched from storage. Refer to 4.2 The Delayed Instruction Slot for a detailed discussion of the delayed Dump and Branch instructions.

The J-type instruction format is used for both jumps and jump-and-links for sub-routine calls. In this format, the 26-bit target address is shifted left two bits, and combined with the high-order 4 bits of the current program counter to form a 32-bit absolute address.

The R-type instruction format which takes a 32-bit byte address contained in a register is used for returns, dispatches, and cross-page jumps.

Branches have 16-bit offsets relative to the program counter (I-type). Jump-and-Link and Branch-and-Link instructions save a return address in Register r31.

Table 3-8 summarizes the V_R3600 Jump Instructions and Table 3-9 summarizes the Branch instructions.

Table 3-8 Jump Instruction Summary

Instruction	Format and Description (op | target)
Jump	J target Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay.
Jump And Link	JAL target Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay. Place address of instruction following delay slot r31 (link register).

(b) JR

Instruction	Format and Description (op | rs | 0 | funct)
Jump Register	JR rs Jump to address contained in register rs with a one instruction delay.

(c) JALR

Instruction	Format and Description (op | rs | 0 | rd | 0 | funct)
Jump And Link Register	JALR rs, rd Jump to address contained in register rs with a one instruction delay. Place address of instruction following delay slot in rd.

Table 3-9 Branch Instruction Summary

(a) BEQ, BNE

Instruction	Format and Description (op | rs | rt | offset)
Branch on Equal	BEQ rs, rt, offset Branch to target address if register rs = rt.
Branch on Not Equal	BNE rs, rt, offset Branch to target address if register rs ≠ rt.

(b) BLEZ, BGTZ

Instruction	Format and Description (op | rs | 0 | offset)
Branch on Less than or Equal Zero	BLEZ rs, offset Branch to target address if register rs less than or = 0.
Branch on Greater Than Zero	BGTZ rs, offset Branch to target address if register rs greater than 0.

*: All Branch Instruction target addresses are computed as follows:
Add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32-bits). All branches occur with a delay of one instruction.

(c) BLTZ, BGEZ, BLTZAL, BGEZAL

Instruction	Format and Description [op] [[illegible]] [funct] [offset]
Branch on Less Than Zero	BLTZ rs, offset Branch to target address if register rs less than 0.
Branch on Greater than or Equal Zero	BGEZ rs, offset Branch to target address if register rs greater than or = to 0.
Branch on Less than Zero And Link	BLTZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs less than 0.
Branch on Greater than or Equal Zero And Link	BGEZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs is greater than or = to 0.

*: All Branch Instruction target addresses are computed as follows:
Add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32-bits). All branches occur with a delay of one instruction.

3.6 Special Instructions

The two Special instructions let software initiate traps. They are always R-type.
Table 3-10 summarizes the Special instructions.

Table 3-10 Special Instructions

(a) SYSCALL

Instruction	Format and Description (op | 0 | funct)
System Call	SYSCALL Initiates system call trap, immediately transferring control to exception handler.

(b) BREAK

Instruction	Format and Description (op | code | funct)
Break point	BREAK Initiates breakpoint trap, immediately transferring control to exception handler.

3.7 Coprocessor Instructions

Coprocessor instructions perform operations in the coprocessors. Coprocessor Loads and Stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see coprocessor manuals). Table 3-11 summarizes the Coprocessor instructions.

Table 3-11 V_R3600 Coprocessor Instruction Summary

(a) LWCz, SWCz

Instruction	Format and Description (op | base | rt | offset)
Load Word on Coprocessor	LWCz rt, offset(base) Sign-extend 16-bit offset and add to base to form address. Load contents of addressed word into coprocessor register rt of coprocessor unit z.
Store Word from coprocessor	SWCz rt, offset(base) Sign-extend 16-bit offset and add to base to form address. Store contents of coprocessor register rt from coprocessor unit z at addressed memory word.

(b) MTCz, MFCz, CTCz, CFCz

Instruction	Format and Description (op | funct | offset)
Move To Coprocessor	MTCz rt, rd Move contents of CPU register rt into coprocessor register rd of coprocessor unit z.
Move From Coprocessor	MFCz rt, rd Move contents of coprocessor register rd from coprocessor unit z to CPU register rt.
Move Control To Coprocessor	CTCz rt, rd Move contents of CPU register rt into coprocessor control register rd of coprocessor unit z.
Move Control From Coprocessor	CFCz rt, rd Move contents of control register rd of coprocessor unit z into CPU register rt.

(c) COPz

Instruction	Format and Description
Coprocessor Operation	COPz cofun Coprocessor z performs an operation. The state of the V_R3600 is not modified by a coprocessor operation.

(d) BCzT, BCzF

Instruction	Format and Description (op | funct | offset)
Branch on Coprocessor z True	BCzT offset Compute a branch target address by adding address of instruction in the 16-bit offset (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor z's condition line is true.
Branch on Coprocessor z False	BCzF offset Compute a branch target address by adding address of instruction in the 16-bit offset (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor z's condition line is false.

3.8 System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions perform operations on the System Control Coprocessor (CP0) registers to manipulate the memory management and exception handling facilities of the processor. Table 3-12 summarizes the instructions available to work with CP0.

Table 3-12 System Control Coprocessor (CP0) Instruction Summary

(a) MTC0, MFC0

Instruction	Format and Description (op / funct / rt / rd / 0)
Move To CP0	MTC0 rt, rd Load contents of CPU register rt into register rd of CP0.
Move From CP0	MFC0 rt, rd Load contents of CP0 register rd into CPU register rt.

(b) TLBR, TLBWI, TLBWR, TLBP, RFE

Instruction	Format and Description
Read Indexed TLB Entry	TLBR Load EntryHi and EntryLo registers with TLB entry pointed at by Index register.
Write Indexed TLB Entry	TLBWI Load TLB entry pointed at by Index register with contents of EntryHi and EntryLo registers.
Write Random TLB Entry	TLBWR Load TLB entry pointed at by Random register with contents of EntryHi and EntryLo registers.
Probe TLB for Matching Entry	TLBP Load Index register with address of TLB entry whose contents match EntryHi and EntryLo. If no TLB entry matches, set high-order bit of Index register.
Restore From Exception	RFE Restore previous interrupt mask and mode bits of Status register into current status bits. Restore old status bits into previous status bits.

Chapter 4 V$_R$3600 Instruction Pipeline

4.1 Pipeline Processing

The execution of a single instruction consists of five primary steps or pipe stages:

(1) IF - Instruction Fetch. Access the TLB and calculate the instruction address required to read an instruction from the I-Cache. Note that the instruction is not actually read into the processor until the beginning (phase 1) of the RD pipe stage.

(2) RD - Read any required operands from CPU registers (RF = Register Fetch) while decoding the instruction.

(3) ALU - Perform the required operation on instruction operands.

(4) MEM - Access memory (D-Cache) if required (for a Load or Store instruction).

(5) WB - Write back ALU results or value loaded from D-cache to register file.

Each of these steps requires approximately one CPU cycle as shown in Figure 4-1 (parts of some operations lap over into another cycle while other operations require only 1/2 cycle).

Fig. 4-1 Instruction Execution Sequence

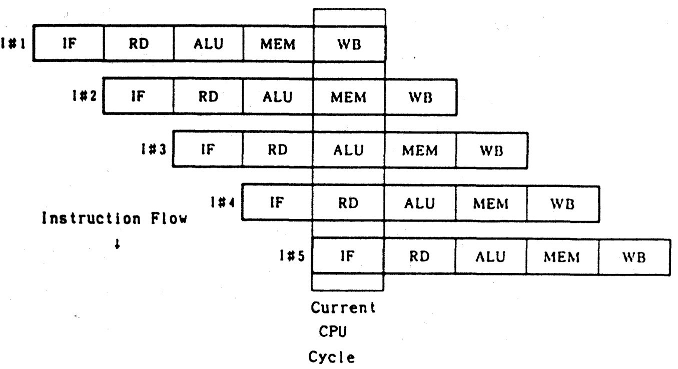
Instruction Execution

IF		RD		ALU		MEM	WB
	I-Cache		RF	OP		D-Cache	WB
TLB					TLB		

←one cycle→

To achieve an instruction execution rate approaching one instruction per CPU cycle, a five-instruction pipeline is utilized. Thus, five instructions at a time are executed in an overlapped fashion as shown in Figure 4-2.

Fig. 4-2 V_R3600 Instruction Pipeline (5-deep)



4.2 The Delayed Instruction Slot

The V_R3600 uses a number of techniques internally to enable execution of all instructions in a single cycle; however, there are two categories of instructions whose special requirements could disturb the smooth flow of instructions through the pipeline.

(1) Delayed Loads

Load instructions have a delay, or latency, of one cycle before the data being loaded is available to another instruction.

(2) Delayed Jumps and Branches

Jump and Branch instructions also have a delay of one cycle while they fetch the instruction and the target address if the branch is taken.

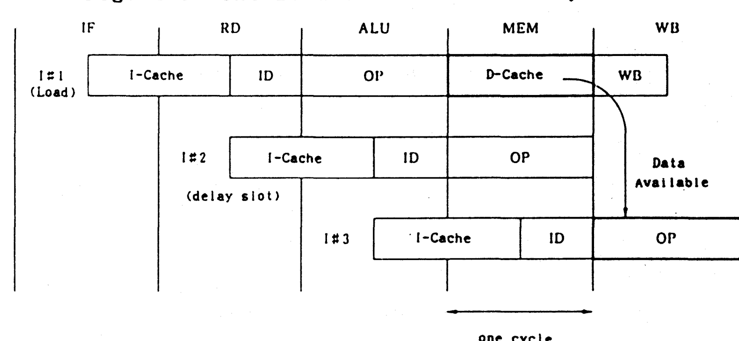
One technique for dealing with the delay inherent with these instructions would be to stall the flow of instructions through the pipeline whenever a load, jump, or branch is executed. However, in addition to the negative impact that this technique would have on instruction throughput, it would also complicate the pipeline logic, exception processing, and system synchronization.

The technique used in the V_R3600 is to continue execution despite the delay. Loads, jumps, and branches do not interrupt the normal flow of instructions through the pipeline; the processor always executes the instruction immediately following one of these "delayed" instructions. Instead of having the processor deal with pipeline delays, the V_R3600 turns over the responsibility for dealing with delayed instructions to software. Thus, an assembler can insert an appropriate instruction immediately following a delayed instruction and has the responsibility of ensuring that the inserted instruction will not be affected by the delay.

(1) Delayed Loads

Figure 4-3 shows three instructions in the V_R3600 pipeline. Instruction 1 (I#1) is a Load instruction. The data from the load is not available until the end of the I#1 MEM cycle - too late to be used by I#2 during its ALU cycle, but available to I#3 for its ALU cycle. Therefore, software must ensure that I#2 does not depend on data loaded by I#1. Usually, a compiler can reorganize instructions so that something useful is executed during the delay slot or, if no other instruction is available, can insert a NOP (no operation) instruction in the slot.

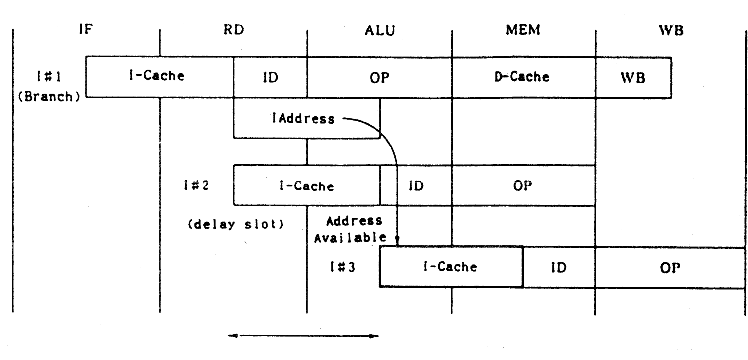
Fig. 4-3 The Load Instruction Delay Slot



(2) Delayed Jumps and Branches

Figure 4-4 also shows three instructions in the V_R3600 pipeline. Instruction 1 (I#1) in this case is a Branch instruction. I#1 must calculate a branch target address, and that address is not available until the beginning of the ALU cycle of I#1 - too late for the I-Cache access of I#2 but available to I#3 for its I-Cache access. The instruction in the delay slot (I#2) will always be executed before the branch or jump actually occurs.

Fig. 4-4 The Jump/Branch Instruction Delay Slot



An assembler has several possibilities for utilizing the branch delay slot productively:

(a) It can insert an instruction that logically precedes the branch instruction in the delay slot since the instruction immediately following the jump/branch effectively belongs to the block preceding the transfer instruction.

(b) It can replicate the instruction that is the target of the branch/jump into the delay slot provided that no side-effects occur if the branch falls through.

(c) It can move an instruction up from below the branch into the delay slot, provided that no side-effects occur if the branch is taken.

(d) If no other instruction is available, it can insert a NOP instruction in the delay slot.

Chapter 5 Memory Management System

The V_R3600 provides a full-featured memory management unit (MMU) utilizing an on-chip Translation Lookaside Buffer (TLB) to provide very fast virtual memory accesses. This chapter describes the operation of the TLB and the CP0 registers that provide the software interface to the TLB. The memory mapping scheme supported by the V_R3600 to translate virtual addresses to physical addresses is also described in detail.

5.1 Memory System Architecture

The V_R3600's virtual memory system logically expands the CPU's physical memory space by translating addresses composed in a large virtual address space into the physical memory space of the V_R3600.

Figure 5-1 shows the form of an V_R3600 virtual address. The most significant 20 bits of a 32-bit virtual address are called the virtual page number, or VPN. The VPN allows mapping of 4 Kbyte pages, while the least significant 12 bits (the offset within a page) are passed along unchanged to form the physical address. The three most significant bits of VPN (bits 31..29) further define how addresses are mapped according to whether the V_R3600 is in user mode or kernel mode (these modes are described in the paragraphs that follow).

Fig. 5-1 Virtual Address Format

31 … 12	11 … 0
VPN	Offset

31	30	29	
0	X	X	kuseg
1	0	0	kseg0
1	0	1	kseg1
1	1	X	kseg2

A 6-bit process identifier field is appended to each virtual address to form unique virtual addresses for up to 64 processes. The mapping of these extended, process-unique virtual addresses to physical addresses need not be one-to-one; virtual addresses of two or more different processes may map to the same physical address.

5.1.1 Privilege states

The V_R3600 provides two privilege states: the Kernel mode, which is analogous to the "supervisory" mode provided by many machines, and the User mode, where non-supervisory programs are executed. The V_R3600 enters the Kernel mode whenever an exception is detected and remains in the Kernel mode until a Restore From Exception (rfe) instruction is executed.

Address mapping is different for Kernel and User modes. To simplify the management of user state from within the Kernel, the user-mode address space is a subset of the Kernel-mode address space. Figure 5-2 shows the virtual-to-physical memory map for both the User mode and Kernel mode segments.

Fig. 5-2 The V_R3600 Virtual Memory Map

Virtual

FFFF FFFFH

kseg2

1 Gbyte

Kernel
Mapped
Cacheable

Any

C000 0000H
BFFF FFFFH

kseg1

512 Mbytes

Kernel
Unmapped
Uncached

A000 0000H
9FFF FFFFH

kseg0

512 Mbytes

Kernel
Unmapped
Cacheable

8000 0000H
7FFF FFFFH

kuseg

User
Mapped
Cacheable

2 Gbytes

Any

0000 0000H

Physical

FFFF FFFFH

Memory

3.5 Gbytes

2000 0000H
1FFF FFFFH

Memory

512 Mbytes

0000 0000H

5.1.2 User-mode virtual addressing

When the processor is operating in User mode, a single, uniform virtual address space (kuseg) of 2 Gbytes is available for users. All valid User-mode virtual addresses have the most-significant bit cleared to 0. An attempt to reference an address with the most-significant bit set while in the User mode causes an Address Error exception. (See CHAPTER 6).

The 2 Gbyte User segment starts at address zero. The TLB maps all references to kuseg identically from Kernel and User modes and controls access cacheability. (The N bit in a TLB entry determines whether the reference will be cached.)

Kuseg is typically used to hold user code and data, and the current user process typically resides in kuseg.

5.1.3 Kernel-mode virtual addressing

When the processor is operating in Kernel mode, three distinct virtual address spaces (in addition to kuseg) are simultaneously available. The three segments dedicated to the kernel are:

kseg0: This cached, unmapped segment starts at virtual address 0x8000_0000 and is 512 Mbytes long.

kseg1: This uncached, unmapped segment begins at virtual address 0xa000_0000 and is 512 Mbytes long.

kseg2: This kernel-mapped, cacheable segment begins at virtual address 0x000_0000 and is 1 Gbytes long.

(1) Kseg0

When the most-significant three bits of the virtual address are "100," the virtual address space selected is a 512-Mbyte kernel physical space (kseg0). The V_R3600 direct-maps references within

kseg0 onto the first 512 Mbytes of physical address space. These references use cache memory, but they do not use TLB entries. Thus, kseg0 is typically used for kernel executable code and some kernel data.

(2) Ksegl

When the most-significant three bits of the virtual address are "101," the virtual address space selected is a 512-Mbyte kernel physical space (ksegl). The processor directly maps ksegl onto the first 512 Mbytes of physical space and uses no TLB entries. Unlike kseg0, ksegl uses uncached references. An operating system typically uses ksegl for I/O registers, ROM code, and disk buffers.

(3) Kseg2

When the most-significant two bits of the virtual address are "11," the virtual address space selected is a 1024 Mbyte kernel virtual space (kseg2). Like kuseg, kseg2 uses TLB entries to map virtual addresses to arbitrary physical ones, with or without caching. (The N bit in a TLB entry determines whether the reference will be cached.) An operating system typically uses kseg2 for stacks and per-process data that it must remap on context switches, for user page tables (memory map), and for some dynamically allocated data areas. Kseg2 allows selective caching and mapping on a per-page basis, rather than requiring an all or nothing approach.

5.2 Virtual Memory and the TLB

Mapped virtual addresses are translated into physical addresses using a Translation Lookaside Buffer (TLB). The TLB is a fully associative memory device that holds 64 entries to provide mapping of 64 4Kbyte pages. When address mapping is indicated (that is, when the access is in kuseg or kseg2), each TLB entry is simultaneously checked for a match with the extended virtual address.

The CPU supports up to four coprocessors. Coprocessor 0 (CP0), which is called the System Control Coprocessor, is implemented as an integral part of the V_R3600. CP0 supports address translation, exception handling, and other "privileged" operations. It consists of the 64-entry TLB plus the ten registers shown in Figure 5-3. The sections that follow describe how each of the four TLB-related registers is used. (Note: CP0 functions and registers associated with exception handling are described in CHAPTER 6).

Fig. 5-3 The CP0 Registers & the TLB

Memory Management

EntryHI Register | EntryLO Register

63
TLB
8
7
("Safe" entries)
(See Random Register)
0

Index Register

Random Register

Exception Handling

Status Register | Cause Register

Context Register | EPC Register

BadVAddr Register | PRId Register

Used with Exception Processing.

5.2.1 TLB entries

Each TLB entry is 64 bits wide and its format is shown in Figure 5-4. Each of the fields of a TLB entry has a corresponding field in the EntryHI/EntryLO register pair described next. Refer to Figure 5-5 and 5-6 for a description of each of the TLB entry fields.

Fig. 5-4 Format of a TLB Entry

VPN	PID	0	PFN	N	D	V	G	0
20	6	6	20	1	1	1	1	8

(Bit 63 at left, bit 0 at right.)

5.2.2 EntryHI & EntryLO registers

These two registers provide the data pathway through which the TLB is read, written, or probed. When address translation exceptions occur, these registers are loaded with relevant information about the address that caused the exception. The format of the EntryHI and EntryLO register pair is the same as the format of TLB entry and is illustrated in Figure 5-4.

EntryLo is the natural form of a Page Table Entry (PTE); however, since PTEs are always loaded by system software, not by the V_R3600 hardware, an operating system can use another format for memory-resident PTEs.

Fig. 5-5 TLB EntryHI Register

63 ... 44	43 ... 38	37 ... 32
VPN	PID	RFU
20	6	6

VPN	Virtual Page Number. Bits 31 .. 12 of virtual address.
PID	Process ID field. A6-bit field which lets multiple processes share the TLB while each process has a distinct mapping of otherwise identical page numbers.
RFU	Reserved. Currently ignores writes, returns zero when read.

Fig. 5-6 TLB EntryLO Register

31 … 12	11	10	9	8	7 … 0
PFN	N	D	V	G	RFU
20	1	1	1	1	8

PFN	Page Frame Number. Bits 31 .. 12 of the physical address. The V_R3600 maps a virtual page to the PFN.
N	Non-cacheable. If this bit is set, the page is marked as non-cacheable and the V_R3600 directly accesses main memory instead of first accessing the cache.
D	Dirty. If this bit is set, the page is marked as "dirty" and therefore writable. This bit is actually a "write-protect" bit that software can use to prevent alteration of data. If an entry is accessed for a write operation when the D bit is cleared, the V_R3600 causes a TLB Mod trap. The TLB entry is not modified on such a trap.
V	Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS Miss occurs.
G	Global. If this bit is set, the V_R3600 ignores the PID match requirement for valid translation. In kseg2, the Global bit lets the kernel access all mapped data without requiring it to save or restore PID (Process ID) values.
RFU	Reserved. Currently ignores writes, returns zero when read.

5.2.3 Virtual address translation

During virtual-to-physical address translation, the V_R3600 compares the PID and the highest 20 bits (the VPN) of the virtual address to the contents of the TLB. Figure 5-7 illustrates the TLB address translation process.

A virtual address matches a TLB entry when the virtual page number (VPN) field of the virtual address equals the VPN field of the entry, and either the Global (G) bit of the TLB entry is set, or the process identifier (PID) field of the virtual address (as held in the EntryHI register) matches the PID field of the TLB entry. While the Valid (V) bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

If a TLB entry matches, the physical address and access control bits (N, D, and V) are retrieved from the matching TLB entry. Otherwise, a TLB miss (or UTLB miss) exception occurs. If the access control bits (D and V) indicate that the access is not valid, a TLB modification or TLB miss exception occurs. If the N bit is set, the physical address that is retrieved is used to access main memory, bypassing the cache.

Fig. 5-7 TLB Address Translation

Input Virtual Address

VPN & PID

MSB=1 — Yes → User Mode — Yes → Address Error Exception

User Mode — No → (continue)

MSB=1 — No →

VPN Match — No → MSB=1

VPN Match — Yes →

G=1 — No → PID Match — No → MSB=1

PID Match — Yes → (continue)

G=1 — Yes →

V=1 — No → TLB Miss Exception

V=1 — Yes →

D=1 — No → Write — Yes → TLB Mod Exception

Write — No → (continue)

D=1 — Yes →

N=1 — Yes → Access Main Memory

N=1 — No → Access Cache

MSB=1 — Yes → TLB Miss Exception

MSB=1 — No → UTLB Miss Exception

Output Physical Address

5.2.4 The index register

The Index register is a 32-bit, read/write register, which has 6 bits that index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (tlbp) instruction (described later in this chapter).

The Index register also specifies the TLB entry that will be affected by the TLB Read (tlbr) and TLB Write Index (tlbwi) instructions. Figure 5-8 shows the format of the Index register.

Fig. 5-8 The Index Register

31	30 — 14	13 — 8	7 — 0
P	RFU	Index	RFU
1	17	6	8

P	Probe failure. Set to 1 when the last TLBProbe (tlbp) instruction was unsuccessful.
Index	Index to the TLB entry that will be affected by the TLBRead and TLBWrite instructions.
RFU	Reserved. Currently ignores writes, returns zero when read.

5.2.5 The random register

The Random register is a 32-bit register. The format of the Random register is shown in Figure 5-9. The six-bit Random field indexes a random entry in the TLB. The V_R3600 decrements the Random field during every machine cycle but constrains the value of this field to the TLB indexes from 63 to 8 (the counter wraps around from 8 to 63, skipping values 7 through 0).

The TLB Write Random (tlbwr) instruction is used to write the TLB entry that this register indexes. The first eight entries (0 to 7) are the "safe" entries because a tlbwr instruction can never replace the contents of these entries. Typically, these eight entries are reserved for use by the operating system.

Although normal operations never require it, the contents of this register can be read to verify proper operation of the process. To further simplify testing, the Random field is set to a value of 63 when the V_R3600 is reset.

Fig. 5-9 The Random Register

31 — 14	13 — 8	7 — 0
RFU	Random	RFU
18	6	8

Random	A random Index (with a value ranging from 8 to 63) to a TLB entry.
RFU	Reserved. Currently ignores writes, returns zero when read.

5.2.6 TLB instructions

The instructions that the V_R3600 provides for working with the TLB are listed in Table 5-1 and described briefly below.

Table 5-1 TLB Instruction

Op Code	Description
tlbp	Translation Lookaside Buffer Probe
tlbr	Translation Lookaside Buffer Read
tlbwi	Translation Lookaside Buffer Write Index
tlbwr	Translation Lookaside Buffer Write Random

Translation Lookaside Buffer Probe (tlbp)

This instruction probes the TLB to see if an entry matches the EntryHI register contents. If a match exists, the V_R3600 loads the Index register with the index of the entry that matches the EntryHI register. When no match exists, the V_R3600 sets the high order bit (the P bit) of the Index register.

Translation Lookaside Buffer Read (tlbr)

This instruction loads the EntryHI and EntryLO registers with the contents of the TLB entry specified by the contents of the Index register.

Translation Lookaside Buffer Write Index (tlbwi)

This instruction loads the specified TLB entry with the contents of the EntryHI and EntryLO registers. The contents of the Index register specify the TLB entry.

Translation Lookaside Buffer Write Random (tlbwr)

This instruction loads a pseudo-randomly specified TLB entry with the contents of the EntryHI and EntryLO registers. The contents of the Random register specify the TLB entry.

Chapter 6 Exception Processing

This chapter describes how the V_R3600 Processor handles exceptions and also describes the system control coprocessor (CP0) registers used during exception processing.

When the V_R3600 detects an exception, the normal sequence of instruction execution is suspended; the processor exits User mode and is forced into Kernel mode where it can respond to the abnormal or asynchronous event. All events that can initiate exception processing are described in this chapter. Table 6-1 lists the exceptions that the V_R3600 recognizes.

The V_R3600's exception handling system efficiently handles machine exceptions, including Translation Lookaside Buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. All of these events interrupt the normal execution flow; the V_R3600 aborts the instruction causing the exception and also aborts all those following in the instruction pipeline which have already begun execution. The V_R3600 then performs a direct jump into a designated exception handler routine.

When an exception occurs, the V_R3600 loads the EPC (Exception Program Counter) with an appropriate restart location where execution may resume after the exception has been serviced. The restart location in the EPC is the address of the instruction which caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot.

6.1 The Exception Handling List

Table 6-1 V_R3600 Exception

Exception	Mnemonic	Cause
Reset	Reset	Assertion of the V_R3600's $\overline{\text{Reset}}$ signal causes an exception that transfers control to the special vector at virtual address 0xbfc00000.
UTLB miss	UTLB	User TLB miss. A reference is made (in either User mode or Kernel mode) to a page in kuseg that has no matching TLB entry.
TLB miss	TLBL (load) TLBS (store)	A referenced TLB entry's Valid bit isn't set or there is a reference to a kseg2 page that has no matching TLB entry.
TLB modified	Mod	During a store instruction, the Valid bit is set but the Dirty bit is not set.
Bus error	IBE (Instruction) DBE (data)	Assertion of the V_R3600's $\overline{\text{BERR}}$* signal due to such external events as bus timeout, backplane bus parity errors, invalid physical addresses or invalid access types.
Address Error	AdEL (load) AdES (store)	Attempt to load, fetch, or store an unaligned word; that is, a word or halfword at an address not evenly divisible by 4 or 2 respectively. Also caused by reference to a virtual address with most significant bit set while in User mode.
Overflow	Ov	Twos complement overflow during add or subtract.
System call	Sys	Execution of the syscall instruction.
Breakpoint	Bp	Execution of the break instruction.
Reserved Instruction	RI	Execution of an instruction with an undefined or reserved major operation code (bits 31..26), or a special instruction whose minor opcode (bits 5..0) is undefined.
Coprocessor Unusable	CpU	Execution of a coprocessor instruction when the CU (Coprocessor Usable) bit is not set for the target coprocessor.
Interrupt	Int	Assertion of one of the V_R3600's six hardware interrupt inputs or setting of one of the two software interrupt bits in the Cause register.

6.2 The Exception Handling Registers

The exception related information is stored in the six Exception Handling Registers, when an exception is occurred. Software can examine these registers during exception processing to determine such things as the cause of an exception, and the state of the CPU at the time of an exception. Each of these registers is described in detail in the paragraphs that follow.

(a) the Cause register

(b) the EPC (Exception Program Counter) register

(c) the Status register

(d) the BadVaddr (Bad Virtual Address) register

(e) the Context register

(f) the PRId (Processor Revision Identifier) register

Two other registers, the Index register and the Random register, are used to implement the V_R3600 virtual memory management system and may also contain information of interest when handling exceptions related to virtual memory errors. Refer to CHAPTER 5 MEMORY MANAGEMENT SYSTEM for a description of these two registers.

Fig. 6-1 The CP0 Exception Handling Registers

Memory Management

EntryHI Register
EntryLO Register
63
TLB
8
7
("Safe" entries)
(See Random Register)
0
Index Register
Random Register
Used with Virtual Memory (TLB)

Exception Handling

Status Register
Cause Register
Context Register
EPC Register
BadVAddr Register
PRId Register

6.2.1 The cause register

This contents of this 32-bit register describe the last exception. A 4-bit exception code (ExcCode) indicates the cause as listed in Table 6-2. The remaining fields contain detailed information specific to certain exceptions. All bits in the register, with the exception of the Sw bits, are read-only. The Sw bits can be written into to set or reset software interrupts. The format for the Cause register is shown in Figure 6-2.

Fig. 6-2 The Cause Register

31	30	29 28	27 … 16	15 … 10	9 8	7 6	5 … 2	1 0
BD	RFU	CE[1..0]	RFU	IP [5..0]	Sw[1..0]	RFU	ExcCode	RFU
1	1	2	12	6	2	2	4	2

BD	Branch Delay	Set of 1 if last exception was taken while executing in a branch delay slot.
CE	Coprocessor Error	Indicates the unit number referenced when a Coprocessor Unusable Exception is taken. (See Table 6-3)
IP	Interrupts Pending	Indicates the external interrupts that are pending. IP[5..0] = Interrupt [5..0]
Sw	Software Interrupts	Indicates which of the two software interrupts are pending. This field may be written into to set or reset software interrupts.
ExcCode	Exception Code field	Described in Table 6-2.
RFU		Reserved. Currently ignores writes, returns zero when read.

Table 6-2 The ExcCode Field

The Cause Register ExcCode Field		
Number	Mnemonic	Description
0	Int	External interrupt
1	Mod	TLB modification exception
2	TLBL	TLB miss exception (Load or Instruction fetch)
3	TLBS	TLB miss exception (store)
4	AdEL	Address error exception (Load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (for an instruction fetch)
7	DBE	Bus error exception (for a data load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor unusable exception
12	Ov	Arithmetic overflow exception
13-15	-	reserved

Table 6-3 CE Field

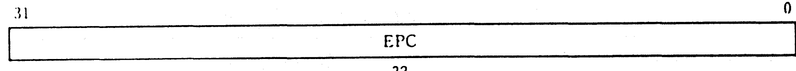
CE bit (1,0)	Coprocessor Unit No.
0,0	0
0,1	1
1,0	2
1,1	3

6.2.2 The EPC (Exception Program Counter) register

The 32-bit, read-only EPC register contains the address where processing can resume after an exception has been serviced.

This register contains the virtual address of the instruction that caused the exception. When that instruction resides in a branch delay slot, the EPC register contains the virtual address of the immediately preceding Branch or Jump instruction. The V_R3600 also sets the Cause register's BD bit if the exception occurred in the branch delay slot. The EPC register format is shown in Figure 6-3.

Fig. 6-3 The EPC Register



6.2.3 The status register

This register contains all major status bits. Any exception puts the system in Kernel mode. All bits in the Status register, with the exception of the TS (TLB Shutdown) bit are readable and writable; the TS bit is read-only. Figure 6-4 shows the format of the Status register and summarizes the functions performed by each bit. Additional details on the function of each Status register bit are provided in the paragraphs that follow.

Fig. 6-4 The Status Register

Bits	31–28	27–26	25	24–23	22	21	20	19	18	17	16	15–8	7–6	5	4	3	2	1	0
Field	CU 3..0	RFU	RE	RFU	BEV	TS	PE	CM	PZ	SwC	IsC	IntMask Int5-0Sw1.0	RFU	KUo	IEo	KUp	IEp	KUc	IEc
Width	4	2	1	2	1	1	1	1	1	1	1	8	2	1	1	1	1	1	1

CU	Coprocessor Usability. These bits control usability of the four possible coprocessors: Cu3, Cu2, Cu1, and Cu0. If a CU bit is set (=1), that coprocessor is usable.
RE	RE reverses the Endianess set during initialization by reset.
BEV*	Bootstrap Exception Vector. If set to 1, causes the V_R3600 to use the alternate, bootstrap vectors for UTLB Miss and general exceptions.
TS*	TLB Shutdown. Set to 1 if V_R3600 has disabled TLB due to catastrophic error. Cleared only by $\overline{\text{Reset}}$.
PE*	Parity Error. Set to 1 if cache parity error occurs. Reset by writing a 1 to this bit.
CM*	Cache Miss. Set to 1 if most recent D-Cache load resulted in a miss (only when the D-Cache is isolated).
PZ*	Parity Zero. When set to 1, causes zero to replace normal outgoing parity bits.
SwC*	Swap Caches. Controls switching of control signals for I-Cache and D-Cache.
IsC*	Isolate Cache. When set to 1, isolates D-Cache from main memory system.
IntMask	Interrupt Mask. When a bit is set to 1, the corresponding hardware interrupt [$\overline{\text{Int5..0}}$] or software interrupt [Sw1..0] is enabled.
KUo	Kernel/User mode, old. Set to 0 if Kernel, 1 if User.
IEo	Interrupt Enable, old. Set to 1 to enable, 0 to disable.
KUp	Kernel/User mode, previous. Set to 0 if Kernel, 1 if User.
IEp	Interrupt Enable, previous. Set to 1 to enable, 0 to disable.

*: Indicates primary use is for diagnostics and testing.

(cont'd)

KUc	Kernel/User mode, current. Set to 0 if Kernel, 1 if User.
IEc	Interrupt Enable, current. Set to 1 to enable, 0 to disable.
RFU	Reserved. Currently ignores writes, returns zero when read.

(1) CU (Coprocessor Usable)

CU controls the usability of each of the possible four coprocessors (Cu3..Cu0). Thus, software can control access of processes to the coprocessors. If a bit is set to 1, the corresponding coprocessor is usable, if a bit is cleared (0), the coprocessor is marked as unusable. All coprocessor instructions require that the target coprocessor be marked usable or a Coprocessor Unusable Exception occurs. Note that the System Control Coprocessor (CP0) is always considered usable when the V_R3600 is operating in Kernel mode regardless of the setting of the Cu0 bit.

(2) RE (Reversal Endianess)

RE reverses the Endianess set during initialization by reset.

(3) BEV (Bootstrap Exception Vectors)

BEV controls location of UTLB miss and general exception vectors during bootstrap (immediately following reset). When this bit is set to 0, the normal exception vectors are used; when the bit is set to 1, bootstrap vector locations are used. This alternate set of vectors can be used when diagnostic tests cause exceptions to occur prior to verifying proper operation of the cache and main memory system. (Refer to 6.3 Exception Description Details later in this chapter for a description of the exception vectors.)

(4) TS (TLB Shutdown)

This read-only bit is intended for use by diagnostics and indicates that the V_R3600 has shut down the TLB due to attempts to access several TLB entries simultaneously. This mechanism protects the TLB from catastrophic hardware failures in the event of software misuse of the TLB — specifically, when two or more TLB entries have the same VPN (Virtual Page Number) and PID (Process ID). When the TLB is in this state, all address translations and TLB probe access are inhibited and have undefined effects. This state can be cleared only by asserting $\overline{\text{Reset}}$.

(5) PE (Parity Error)

This bit is set if a cache parity error has occurred. Since the V_R3600 transparently recovers from parity errors (by taking a cache miss and accessing main memory) this bit is intended for diagnostic purposes. Software can use this bit to log cache parity errors, and diagnostics can use it to verify proper functioning of the cache parity bits and cache parity trees. To clear this bit, write a one to PE; writing a zero to this bit does not affect its value.

(6) CM (Cache Miss)

This bit is set if the most recent D-Cache load resulted in a cache miss and is intended for use by diagnostic programs to verify the proper functioning of the cache tag and parity bits. This bit setting only takes effect when in the "isolated cache" mode. See the IsC bit.

(7) PZ (Parity Zero)

If this bit is set, outgoing parity bits (for both cache data and tags) for store instructions are set to zero.

(8) SwC (Swap Caches)

This bit controls swapping of the control signals for the data cache (D-Cache) and instruction cache (I-Cache). 0 means normal; 1 means switched.) Cache swapping can be used to implement cache flushing mechanisms and to perform cache testing and diagnostics.

(9) IsC (Isolate Cache)

Setting this bit isolates the D-Cache from the main memory system. (0 means normal; 1 means D-Cache isolated.) Cache isolation can be used to implement cache flushing mechanisms and to perform cache testing and diagnostics.

(10) IntMask (Interrupt Mask)

These bits allow individual enabling/disabling of each of the eight interrupt classes -- six hardware interrupts and two software interrupts. A 0 in a bit position disables that interrupt and a 1 enables the interrupt. All interrupts can be disabled by clearing the Interrupt Enable bit(s) IEo/IEp/IEc described below.

(11) KUo/KUp/KUc (Kernel/User mode: Old/Previous/Current)

These three bits comprise a 3-level stack showing the old/previous/current mode (0 means Kernel; 1 means User). Manipulation and use of these bits during exception processing is described in the section that follows.

(12) IEo/IEp/IEc (Interrupt Enable: Old/Previous/Current)

These three bits comprise a 3-level stack showing the old/previous/current interrupt enable settings (0 means disable; 1 means enable). Manipulation and use of these bits during exception processing is described in the section that follows.

6.2.4 Status register mode bits and exception processing

When the V_R3600 responds to an exception it saves the current Kernel/User mode(KUc) and current interrupt enable mode (IEc) bits of the Status register into the previous mode bits (KUp and IEp). The previous mode bits (KUp and IEp) are saved into the old mode bits (KUo and IEo). The current mode bits (KUc and IEc) are cleared to cause the processor to enter the Kernel operating mode and turn off interrupts.

This three-level set of mode bits lets the V_R3600 respond to two levels of exceptions before software must save the contents of the Status register. Figure 6-5 shows how the V_R3600 manipulates the Status register during exception recognition.

Fig. 6-5 The Status Register and Exception Recognition

	5	4	3	2	1	0
	KUo	IEo	KUp	IEp	KUc	IEc

Exception recognition

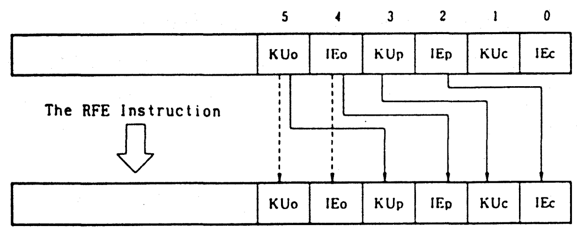
	KUo	IEo	KUp	IEp	KUc	IEc
	← KUp	← IEp	← KUc	← IEc	0	0

After an exception handler has completed execution, the V_R3600 must return to the system context that existed prior to the exception (if possible). The Restore From Exception (RFE) instruction provides the mechanism for this return.

The Restore From Exception (RFE) instruction restores control to a process that an exception pre-empted. When the RFE instruction is executed, it restores the "previous" interrupt mask (IEp) bit and Kernel/User mode (KUp) bit in the Status register into the corresponding "current" status bits (IEc and KUc). It also restores the "old" status bits (IEo and KUo) into the corresponding previous status bits (IEp and KUp). The old status

bits (IEo and KUo) remain unchanged. The actions of the RFE instruction are illustrated in Figure 6-6.

Fig. 6-6 Restoring from Exceptions

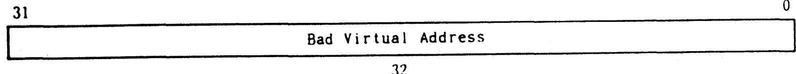


6.2.5 BadVAddr register

The BadVAddr register saves the entire bad virtual address for any addressing exception: AdEL or AdES. Figure 6-7 illustrates the organization of the register.

Note: This register does not save any information for bus errors since these are not addressing errors.

Fig. 6-7 The BadVAddr Register



6.2.6 Context register

The Context register duplicates some of the information provided in the BadVAddr register, but provides the information in a form that may be more useful for a software TLB exception handler. It is designed for use in a UTLB miss handler, which loads TLB entries for normal user-mode references.

The Context register can be used to hold a pointer into the Page Table Entry (PTE). An operating system sets the PTE base field in the register, as needed. Normally, an operating system uses the Context register to address the current user process's page map, which resides in the kernel-mapped segment kseg2. Note that the use of this register is solely for the convenience of the operating system.

For all addressing exceptions (except bus errors), this register holds the Virtual Page Number (VPN) from the most recent virtual address for which the translation was invalid. Figure 6-8 shows the format of the Context register.

Fig. 6-8 The Context Register

31 .. 21	20 .. 2	1 .. 0
PTEBase	BadVPN	RFU
11	19	2

PTEBase	Holds the base for the Page Table Entry (set by software).
BadVPN	Holds the failing Virtual Page Number (set by hardware). This field is read-only and contains bits 30..12 (user-segment VPN) of the BadVAddr register.
RFU	Unused; ignored on writes, zero when read.

6.2.7 Processor revision identifier register

This 32-bit read-only register contains information that identifies the implementation and revision level of the Processor and System Control Coprocessor. The format of the register is shown in Figure 6-9.

Fig. 6-9 The Processor Revision Identifier Register

31 … 16	15 … 8	7 … 0
RFU	Imp	Rev
16	8	8

Imp	Implementation identifier (02H).
Rev	Revision identifier.
RFU	Reserved. Currently ignores writes, returns zero when read.

6.3 Exception Description Details

This part of CHAPTER 6 describes the following exceptions — their cause, handling, and servicing.

- Address Error Exception
- Breakpoint Exception
- Bus Error Exception
- Coprocessor Unusable Exception
- Interrupt Exception
- Overflow Exception
- Reserved Instruction Exception
- Reset Exception
- System Call Exception
- TLB Miss Exception

Note: You cannot mask machine exceptions.

Exception Vector Locations

The V_R3600 uses three different addresses for exception vectors:

(1) The RESET exception vector is at address BFC0 0000H.

(2) The UTLB Miss exception vector is at address 8000 0000H.

(3) The General exception vector which is used for all other types of exceptions is at address 8000 0080H.

Caution: If the BEV (Bootstrap Exception Vector) bit in the Status Register is set to 1, the UTLB Miss vector address is changed to BFC0 0100H and the General exception vector is changed to BFC0 0180H.

6.3.1 Address error exception

Cause: This exception occurs when an attempt is made to load, fetch, or store a word that is not aligned on a word boundary. Attempts to load or store a half-word that is not aligned on a half-word boundary also cause this exception. The exception also occurs in User mode if a reference is made to a virtual address whose most significant bit is set — a kernel address. This exception is not maskable.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. When the exception occurs, the V_R3600 sets the ADEL or ADES code in the Cause register's ExcCode field to indicate whether the address error occurred during an instruction fetch or a load operation (ADEL) or a store operation (ADES).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively and clears the KUc and IEc bits.

When this exception occurs, the BadVAddr register contains the virtual address that was not properly aligned or that improperly addressed kernel data while in User mode. The contents of the VPN field of the Context and EntryHi registers are undefined.

Servicing: A kernel should hand the executing process a segmentation violation signal.

Caution: Such an error is usually fatal although an alignment error might be handled by simulating the instruction that caused the error.

6.3.2 Breakpoint exception

Cause: This exception occurs when the V_R3600 executes the BREAK instruction. This exception is not maskable.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception and sets the BP code in the Cause register's ExcCode field.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the Kuo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The EPC register points at the BREAK instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the BREAK instruction and sets the BD bit of the Cause register.

Servicing: Transfer control to the applicable system routine. Unused bits of the BREAK instruction (bits 26..6) can be used to pass additional information. To examine these bits, load the contents of the instruction pointed at by the EPC register.

To resume execution, change the EPC register so that the V_R3600 does not execute the BREAK instruction again.

Caution 1: If the instruction resides in the branch delay slot, add four to the contents of the EPC register to find the instruction.

2: If a BREAK instruction is in the branch delay slot, the branch instruction must be interpreted in order to resume execution.

6.3.3 Bus error exception

Cause: This exception occurs when the $\overline{\text{Bus Error}}$ input to the CPU is asserted by external logic. For example, events like bus time-outs, backplane bus parity errors, and invalid physical memory addresses or access types can signal this exception. This exception is not maskable.

This exception is used for synchronously occurring events such as cache miss refills, uncached references, and unbuffered writes. The general interrupt mechanism must be used to report a bus error that results from asynchronous events such as a buffered write transaction.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. When the exception occurs, the V_R3600 sets the IBE or DBE code in the Cause register's ExcCode field to indicate whether the error occurred during an instruction fetch reference (IBE) or during a data load or store reference (DBE).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: The physical address where the fault occurred can be computed from the information in the CP0 registers:

- If the Cause register's IBE code is set (showing an instruction fetch reference), the virtual address resides in the EPC register.

- If the Cause register's DBE exception code is set (specifying a load or store reference), the instruction that caused the exception is at the virtual address contained in the EPC register. Interpret the instruction pointed to by EPC to get the virtual address of the load or store reference and then use the TLB Probe (TLBP) instruction and read EntryLo to compute the physical page number.

Caution 1: If the BD of the cause register is set, add four to the contents of the EPC register.

2: A kernel should hand the executing process a bus error when this exception occurs. Such an error is usually fatal.

6.3.4 Coprocessor unusable exception

Cause: This exception occurs due to an attempt to execute a coprocessor instruction when the corresponding coprocessor unit has not been marked usable (the appropriate CU bit in the Status register has not been set). For CP0 instructions, this exception occurs when the unit has not been marked usable and the process is executing in User mode: CP0 is always usable from Kernel mode regardless of the setting of the Cu0 bit in the Status register. This exception is not maskable.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. It sets the CpU code in the Cause register's ExcCode field. Only one coprocessor can fail at a time.

The contents of the Cause register's CE (Coprocessor Error) field show which of the four coprocessors (3, 2, 1, or 0) the V_R3600 referenced when the exception occurred.

The EPC register points at the coprocessor instruction that

caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the coprocessor instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: To identify the coprocessor unit that was reference, examine the contents of the Cause register's CE field. If the process is entitled to access, mark the coprocessor usable and restore the corresponding user state to the coprocessor.

If the process is entitled to access to the coprocessor, but the coprocessor is known not to exist or to have failed, the system could interpret the coprocessor instruction. If the BD bit is set in the Cause register, the branch instruction must be interpreted; then, the coprocessor instruction could be emulated with the EPC register advanced past the coprocessor instruction.

Caution: If the process is not entitled to access to the coprocessor, the process executing at the time should be handed an illegal instruction/privileged instruction fault signal. Such an error is usually fatal.

6.3.5 Interrupt exception

Cause: This exception occurs when one of eight interrupt conditions (software generates two, hardware generates six) occurs.

Each of the eight external interrupts can be individually masked by clearing the corresponding bit in the IntMask field of the Status register. All eight of the interrupts can be masked at once by clearing the IE bit in the Status register.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. The V_R3600 sets the Int code in the Cause register's ExcCode field.

The IP field in the Cause register show which of six external interrupts are pending, and the SW field in the Cause register shows which of two software interrupts are pending. More than one interrupt can be pending at a time.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: If software generates the interrupt, clear the interrupt condition by setting the corresponding Cause register bit (SW1:0) to zero.

If external hardware generates the interrupt, clear the interrupt condition by alleviating conditions that assert the interrupt signal ($\overline{\text{Int5-Int0}}$).

6.3.6 Overflow exception

Cause: This exception occurs when an ADD ADDI, SUB, or SUBI instruction results in two's complement overflow. This exception is not maskable.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. The V_R3600 sets the OV code in the ExcCode field of the Cause register.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: A kernel should hand the executing process a floating point exception or integer overflow error when this exception occurs. Such an error is usually fatal.

6.3.7 Reserved instruction exception

Cause: This exception occurs when the V_R3600 executes an instruction whose major opcode (bits 31..26) is undefined or a SPECIAL instruction whose minor opcode (bits 5..0) is undefined.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. It sets the RI code of the Cause register's ExcCode field.

The EPC register points at the reserved instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the reserved instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

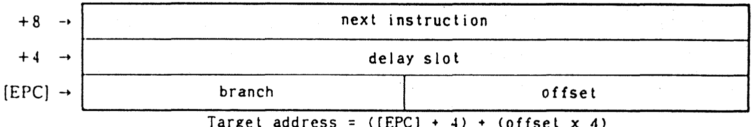
Servicing: If instruction interpretation is not implemented, the kernel should hand the executing process an illegal instruction/reserved operand fault signal.

An operating system can interpret the undefined instruction and pass control to a routine that implements the instruction in software. If the undefined instruction is in the branch delay slot, the routine that implements the instruction is responsible for simulating the branch instruction after the undefined

instruction has been "executed". Simulation of the branch instruction includes determining if the conditions of the branch were met and transferring control to the branch target address (if required) or to the instruction following the delay slot if the branch is not taken. If the branch is not taken, the next instruction's address is = [EPC] + 8. If the branch is taken, the branch target address is calculated as shown below:

Caution: **An error that the kernel hand the executing process an illegal instruction/reserved operation fault signal is fatal.**

Fig. 6-10 Branch Target Address



Note that the target address is relative to the address of the instruction in the delay slot, not the address of the branch instruction. Refer to the descriptions of branch instruction for details on how branch target addresses are calculated.

6.3.8 Reset exception

Cause: This exception occurs when the V_R3600's $\overline{\text{RESET}}$ signal is asserted and then de-asserted.

Handling: The V_R3600 provides a special interrupt vector (BFC0 0000H) for this exception. The Reset vector resides in the V_R3600's unmapped and uncached address space; therefore the hardware need not initialize the Translation Lookaside Buffer (TLB) or the cache to handle this exception. The processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the V_R3600 are undefined when this exception occurs except for the following:

- The TS, SWc, KUc, and IEc bits of the Status register are cleared to zero.

- The BEV bit of the Status register is set to 1.

- The Random register is initialized to 63.

Servicing: The Reset exception is serviced by initializing all processor registers, coprocessor registers, the caches, and the memory system. Typically, diagnostics would then be executed and the operating system bootstrapped. The Reset exception vector is selected to appear within the uncached, unmapped memory space of the machine so that instructions can be fetched and executed while the cache and virtual memory system are still in an undefined state.

6.3.9 System call exception

Cause: This exception occurs when the V_R3600 executes a SYSCALL instruction.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception and sets the Sys code in the Cause register's ExcCode field.

The EPC register points at the SYSCALL instruction that caused the exception, unless the SYSCALL instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the SYSCALL instruction and the BD bit of the Cause register is set.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: The operating system transfers control to the applicable system routine. To resume execution, alter the EPC register so that the SYSCALL instruction does not execute again. To do this, add four to the EPC register before returning.

Note: If a SYSCALL instruction is in a branch delay slot, the branch instruction must be interpreted in order to resume execution.

6.3.10 TLB miss exceptions

There are three different types of TLB misses that can occur:

(a) If the input Virtual Page Number (VPN) does not match the VPN of any TLB entry, or if the Process Identifier (PID) in EntryHi does not match the TLB entry's PID (and the Global bit is not set), a miss occurs. For kuseg, a UTLB Miss occurs. For kseg2, a TLB Miss occrs.

(b) If everything matches, but the Valid bit of the matching TLB entry is not set, a TLB Miss occurs.

(c) If the dirty bit in a matching TLB entry is not set and the access is a write, a TLB MOD exception occurs.

Figure 6-11 (a simplified version of TLB address translation figure used in CHAPTER 5) illustrates how the three different kinds of TLB miss exceptions are generated. Each of the exceptions is described in detail in the pages that follow.

Fig. 6-11 TLB Miss Exceptions

Input Virtual Address

VPN & PID

VPN Match — No / Yes

G=1 — No / Yes

PID Match — No / Yes

V=1 — No / Yes

D=1 — No / Yes

Write — Yes / No

MSB=1 — Yes / No

TLB Mod Exception

Output Physical Address

TLB Miss Exception

UTLB Miss Exception

(1) TLB Miss Exception

Cause: This exception occurs when a Kernel mode virtual address reference to memory is not mapped, when a User mode virtual address reference to memory matches an invalid TLB entry, or when a Kernel mode reference to user memory space matches an invalid TLB entry.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception. When the exception occurs, the V_R3600 sets the TLBL or TLBS code in the Cause register's ExcCode field to indicate whether the miss was due to an instruction fetch or a load operation (TLBL) or a store operation (TLBS).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

When this exception occurs, the BadVAddr, Context, and EntryHi registers contain the virtual address that failed address translation. The PID field of EntryHi remains unchanged by this exception. The Random register normally specifies the pseudo-random location where the V_R3600 can put a replacement TLB entry.

Servicing: The failing virtual address or virtual page number identifies the corresponding PTE. The operating system should load EntryLo with the appropriate PTE that contains the physical page frame and access control bits and also write the contents of EntryLo and EntryHi into the TLB.

Servicing Multiple (nested) TLB Misses

Within a UTLB Miss handler, the virtual address that specifies the PTE contains physical address and access control information that might not be mapped in the TLB. Then, a TLB Miss exception occurs. You can recognize this case by noting that the EPC register points within the UTLB Miss handler. The operating system might interpret the event as an address error (when the virtual address falls outside the valid region for the process) or as a TLB Miss on the page mapping table.

This second TLB miss obscures the contents of the BadVAddr, Context, and EntryHi registers as they were within the UTLB Miss handler. As a result, the exact virtual address whose translation caused the first fault is not known unless the UTLB Miss handler specifically saved this address. You can only observe the failing PTE virtual address. The BadVAddr register now contains the original contents of the Context register within the UTLB Miss handler, which is the PTE address for the original faulting address.

If the operating system interprets the exception as a TLB Miss on the page mapping table, it constructs a TLB entry to map the page table and writes the entry into the TLB. Then, the operating system can determine the original faulting virtual page number, but not the complete address. The operating system uses this information to fetch the PTE that contains the physical address and access control information. It also writes this information into the TLB.

The UTLB Miss handler must save the EPC in a way that allows the second miss to find it. The EPC register information that the UTLB Miss handler saved gives the correct address at which to resume execution. The "old" KUo and IEo bits of the Status register contain the correct mode after the V_R3600 services a double miss.

Note: You neither need nor want to return to the ULTB Miss handler at this point.

(2) TLB Modified Exception

Cause: This exception occurs when a store operation's virtual address reference to memory matches a TLB entry that is marked valid, but not marked dirty. This exception is not maskable.

Handling: The V_R3600 branches to the General Exception vector (8000 0080H) for this exception and sets the Mod exception code in the Cause register's ExcCode field.

When this exception occurs, the BadVAddr, Context, and EntryHi registers contain the virtual address that failed address translation. EntryHi also contains the PID from which the translation fault occurred.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing: A kernel should use the failing virtual address or virtual page number to identify the corresponding access control information. The identified page might or might not permit write accesses. (Typically, software maintains the "real" write protection in unused hardware bits.) If the page does not permit write access, a "Write Protection Violation" occurs.

If the page does permit write accesses, the kernel should mark the page frame as dirty in its own data structures. Use the

TLBProbe (tlbp) instruction to put the index of the TLB entry that must be altered in the Index register. Then load the EntryLo register with a word that contains the physical page frame and access control bits (with the data bit D set). Finally, use the TLBWrite Indexed (tlbwi) instruction to write EntryHi and EntryLo into the TLB.

(3) UTLB Miss Exception

Cause: This exception occurs from Use, or Kernel mode references to user memory space when no TLB entry matches both the VPN and the PID. Invalid entries cause a TLB Miss rather than a UTLB Miss. This exception is not maskable.

Handling: The V_R3600 uses the special ULTB Miss interrupt vector (8000 0000H) for this exception. When the exception occurs, the V_R3600 sets the TLBL or TLBS code in the Cause register's ExcCode field to indicate whether the miss was due to an instruction fetch or a load operation (TLBL) or a store operation (TLBS).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The V_R3600 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The virtual address that failed translation is held in the BadVAddr, Context, and EntryHi registers. The EntryHi register also contains the PID (Process Identifier) from which the translation fault occurred. The Random register contains a valid psuedo-random location in which to put a replacement TLB entry.

Servicing: The contents of the Context register can be used as the virtual address of the memory word that contains the physical page frame and the access control bits—a Page Table Entry (PTE)--for the failing reference. An operating system should put the memory word in EntryLo and write the contents of EntryHi and EntryLo into the TLB by using a TLB Write Random (tlbwr) assembly instruction.

The PTE virtual address might be on a page that is not resident in the TLB. Therefore, before an operating system can reference the PTE virtual address, it should save the EPC register's contents in a general register reserved for kernel use or in a physical memory location. If the reference is not mapped in the TLB, a TLB Miss exception would occur within the UTLB Miss handler.

Chapter 7 Instruction Set Details

This chapter provides a detailed description of the operation of each V_R3600 instruction. The instructions are listed in alphabetical order.

Refer to PART 3 FPU ARCHITECTURE for a detailed description of the V_R3600 FPU instructions.

The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and manner of handling exceptions is omitted from the instruction descriptions in this chapter. Refer to CHAPTER 6 EXCEPTION HANDLING for detailed descriptions of exceptions and handling.

Refer to APPENDIX A for the constant fields of each instruction.

7.1 Instruction Classes

V_R3600 instructions are divided into the following classes:

(1) Load/Store instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is base register + 16-bit immediate offset.

(2) Computational instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.

(3) Jump and Branch instructions change the control flow of a program. Jump are always to absolute 26-bit word addresses (J-type format), or 32-bit register address (R-type). Branches have 16-bit offsets relative to the program counter (I-type) performing the addition of 16-bit offsets to the program counter (instruction address in the delay slot). Jump and Link instructions (such as JAL, JALR, BLTZAL, BGEZAL, etc.) save a return address in Register 31.

(4) Coprocessor instructions perform operations in the coprocessors. Coprocessor Loads and Stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPU instructions). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.

(5) Special instructions perform a variety of tasks, including movement of data between special and general registers, syscall, and breakpoint. They are always R-type.

7.2 Instruction Formats

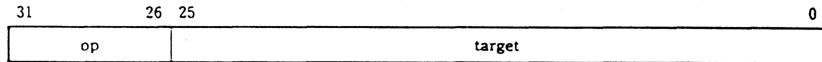
Every V_R3600 instruction consists of a single word (32 bits) aligned on a word boundary and there are only three instruction formats as shown in Figure 7-1.

Fig. 7-1 V_R3600 Instruction Formats

I-Type (Immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-Type (Jump)



R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

Remarks: The variable subfields are shown as follows.

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

7.3 Instruction Notation Conventions

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

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

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 7-2.

Table 7-1 V_R3600 Instruction Operation Notations

Symbol	Meaning
←	Assignment
‖	Bit string concatenation
x^y	Replication of bit value x into a y-bit string. Note that x is always a single-bit value.
$x_{y...z}$	Selection of bits y through z of bit string x. Little-endian bit notation is always used. If y is less than z, this expression is an empty (zero length) bit string.
+	Two's complement addition
-	Two's complement subtraction
*	Two's complement multiplication
div	Two's complement integer division
mod	Two's complement modulo
<	Two's complement less than comparison
and	Bitwise logic AND
or	Bitwise logic OR
xor	Bitwise logic XOR
nor	Bitwise logic NOR
GPR[x]	General Register x. The contents of GPR[0] is always zero. Attempts to alter the contents of GPR[0] have no effect.
CPR[z,x]	Coprocessor unit z, general register x.
CCR[z,x]	Coprocessor unit z, control register x.
COC[z]	Coprocessor unit z, control signal.
Big Endian Mem	Big Endian mode selected during reset. Indicates the Endianess in the memory interface or during Kernel mode execution.
Reverse Endian	Reverses the Endianess for the Load/Store instructions. Usable only in the User mode, and set by RE bit for the status register. ReverseEndian can be calculated by (SR_{25} and UserMode).

(cont'd)

Symbol	Meaning
BigEndian CPU	The Endianess for the Load/Store instructions. This may be reversed in the User mode by setting SR_{25}. BigEndianCPU can be calculated by (BigEndianMem xor ReverseEndian).
T + i	Indicates the time steps (CPU cycles) between operations. Thus, operations identified as occurring at T + 1 are performed during the cycle following the one where the instruction was initiated. This type of operation occurs with loads, stores, jumps, branches and coprocessor instructions.
vAddr	Virtual address
pAddr	Physical address

Instruction Notation Examples

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

Example #1:

GPR[rt] immediate 0^{16}

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

Example #2:

$(immediate_{15})^{16}$ $immediate_{15...0}$

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

7.4 Instruction Class Summary

7.4.1 Load and store instructions

All load operations have a latency of one instruction. That is, the instruction immediately following a load cannot use the contents of the register which will be loaded with the data being fetched from storage. An exception is that the target register for the load word left (LWL) and load word right (LWR) instructions may be specified as the same register used as the destination of a load instruction that immediately precedes it.

In the load/store operation descriptions, the functions listed in Table 7-2 are used to summarize the handling of virtual addresses and physical memory.

Table 7-2 Load/Store Common Functions

Function	Description
Addr Translation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the entry for the page containing the virtual address is not present in the TLB (Translation Lookaside Buffer).
Load Memory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the AccTyp (1:0) signal 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 into the cache.
Store Memory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data into the word containing the specified physical address. The low-order two bits of the address and the AccTyp (1:0) signal indicate which of the four bytes within the data word should be stored.

The AccTyp (1:0) signal indicates the size of the data item to be loaded or stored as shown in Table 7-3. Regardless of access type or byte-numbering order (endianess), the address specifies the byte which has the smallest byte address of the bytes in the addressed field. For a big-endian machine, this is the leftmost

byte and contains the sign for a two's complement number; for a little-endian machine, this is the rightmost byte and contains the lowest precision byte.

Table 7-3 Access Type Specifications for Loads/Stores

AccTyp (1:0)		Meaning
Mnemonic	Value	
WORD	3	word (32 bits)
TRIPLE-BYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

The bytes within the addressed word which are used can be determined directly from the access type and the two low-order bits of the address, as shown in Table 7-4.

Table 7-4 Byte Specifications for Loads/Stores

AccTyp		Low-Order Address bit		Low-Order Address bit							
				Big-Endian				Little-Endian			
1	0	1	0	31			0	31			0
1 (word)	1	0	0	0	1	2	3	3	2	1	0
1 (triple-byte)	0	0	0	0	1	2			2	1	0
		0	1		1	2	3	3	2	1	
0 (halfword)	1	0	0	0	1					1	0
		1	0			2	3	3	2		
0 (byte)	0	0	0	0							0
		0	1		1					1	
		1	0			2			2		
		1	1				3	3			

7.4.2 Jump and branch instructions

All jump and branch instructions are implemented with a delay of exactly one instruction. That is, the instruction immediately following a jump or branch (i.e., occupying the delay slot) is always executed while the target instruction is being fetched from storage. It is not valid for a delay slot to be occupied itself by a jump or branch instruction; however, this error is not detected, and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the V_R3600 sets the EPC register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are re-executed.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 may not be used as a source register.

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

7.4.3 Coprocessor instructions

The V_R series™ architecture provides four coprocessor units, or classes. Coprocessors are alternate execution units, which have separate register files from the V_R3600 processor. Each coprocessor has 2 register spaces, each with thirty-two 32-bit registers. The first space, coprocessor general registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor. The second, coprocessor control registers, may only have their contents transferred directly between the coprocessor and processor. Coprocessor instructions may alter registers in either space.

Normally, by convention, coprocessor control register 0 is interpreted as a coprocessor revision register. However, the system control coprocessor (CP0) uses coprocessor general register 15 for the processor/coprocessor revision register. The register's low-order byte (bits 7..0) is interpreted as a coprocessor unit implementation descriptor. The second byte (bits 15..8) is interpreted as a coprocessor unit revision number. The contents of the high-order halfword of the register are not defined.

7.4.4 System control coprocessor (CP0) instructions

There are some special limitations imposed on operations involving the System Control Coprocessor (CP0) that is incorporated within the V_R3600. Although load and store instructions to transfer data to and from coprocessors and move control to/from coprocessor instructions are generally permitted by the V_R3600 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 reading from and writing to the CP0 registers.

Several coprocessor operation instructions are defined for CP0 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.

7.5 Instructions

ADD

Add

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

Format:

ADD rd, rs, rt

Description:

The contents of general register rs and the contents of general register rt are added to form a 32-bit result. The result is placed into general register rd.

An overflow exception occurs if the two highest order carry-out bits differ (two's complement overflow).

Operation:

T :	GPR[rd] ← GPR[rs] + GPR[rt] ;

Exceptions:

Overflow exception

ADDI Add Immediate

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

Format:

ADDI rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form a 32-bit result. The result is placed into general register rt.

An overflow exception occurs if the two highest order carry-out bits differ (two's complement overflow).

Operation:

T: $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \| immediate_{15..0}$;

Exception:

Overflow exception

ADDIU Add Immediate Unsigned

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

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form a 32-bit result. The result is placed into general register rt. No overflow exception occurs under any circumstances.

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

Operation:

T:	$GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \parallel immediate_{15..0}$

Exception:

None.

ADDU

Add Unsigned

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

Format:

ADDU rd, rs, rt

Description:

The contents of general register rs and the contents of general register rt are added to form a 32-bit result. The result is placed into general register rd.

No overflow exception occurs under any circumstances.

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

Operation:

T:	GPR[rd] ← GPR[rs] + GPR[rt] ;

Exception:

None.

AND **And**

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

Format:

AND rd, rs, rt

Description:

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

Operation:

T:	GPR[rd] ← GPR[rs] and GPR[rt]

Exception:

None.

ANDI — And Immediate

31–26	25–21	20–16	15–0
ANDI	rs	rt	immediate
6	5	5	16

Format:

ANDI rt, rs, immediate

Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical AND operation. The result is placed into general register rt.

Operation:

T:	$GPR[rt] \leftarrow 0^{16} \parallel (immediate \text{ and } GPR[rs]_{15..0})$;

Exception:

None.

BCzF — Branch On Coprocessor z False

31–26	25–21	20–16	15–0
COPz	BC	BCF	offset
6	5	5	16

Format:

BCzF offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower two bits of the 16-bit offset. If the coprocessor z's condition signal (CpCond=0) is false, then the program branches to the target address, with a delay of one instruction.

Operation:

T−1 :	condition ← not COC[z]
T :	target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
T+1 :	if condition then
	PC ← PC + target
	endif

Exception:

Coprocessor unusable exception

BCzT　　　　Branch On Coprocessor z True

31　　26	25　　21	20　　16	15　　0
COPz	BC	BCT	offset
6	5	5	16

Format:

BCzT offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. If the coprocessor z's condition signal (CpCond=1) is true, then the program branches to the target address, with a delay of one instruction.

Operation:

T−1 :	condition ← COC[z]
T :	target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
T+1 :	if condition then
	PC ← PC + target
	endif

Exception:

Coprocessor unusable exception

BEQ Branch On Equal

31 — 26	25 — 21	20 — 16	15 — 0
BEQ	rs	rt	offset
6	5	5	16

Format:

BEQ rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

T : target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
condition ← (GPR[rs] = GPR[rt])
T+1 : if condition then
PC ← PC + target
endif

Exception:

None.

BGEZ

Branch On Greater Than Or Equal To Zero

31 26	25 21	20 16	15 0
BCOND	rs	BGEZ	offset
6	5	5	16

Format:

BGEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

Operation:

T: $target \leftarrow (offset_{15})^{14} \parallel offset \parallel 0^2$

$condition \leftarrow (GPR[rs]_{31} = 0)$

T+1: if condition then

PC ← PC + target

endif

Exception:

None.

BGEZAL

Branch On Greater Than
Or Equal To Zero And Link

31 — 26	25 — 21	20 — 16	15 — 0
BCOND	rs	BGEZAL	offset
6	5	5	16

Format:

BGEZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. Unconditonally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however.

Operation:

```
T :      target ← (offset15)^14 || offset || 0^2
           condition ← (GPR[rs]31 = 0)
         GPR[31] ← PC + 8
T+1 :    if condition then
           PC ← PC + target
         endif
```

Exception:

None.

BGTZ **Branch On Greater Than Zero**

31 26	25 21	20 16	15 0
BGTZ	rs	0	offset
6	5	5	16

Format:

BGTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. The contents of general register rs are compared to zero. If the contents of general register rs have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

T :	target ← $(offset_{15})^{14}$ ∥ offset ∥ 0^2
	condition ← $(GPR[rs]_{31} = 0)$ and $(GPR[rs] \neq GPR[r0])$
T + 1 :	if condition then
	PC ← PC + target
	endif

Exception:

None.

BLEZ

Branch On Less Than Or Equal To Zero

31–26	25–21	20–16	15–0
BLEZ	rs	0	offset
6	5	5	16

Format:

BLEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. The contents of general register rs is compared to zero. If the contents of general register rs have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T :      target ← (offset15)^14 || offset || 0^2
             condition ← (GPR[rs]31 = 1 ) and (GPR[rs] ≠ GPR[r0])
T + 1 :  if condition then
             PC ← PC + target
         endif
```

Exception:

None.

BLTZ — Branch On Less Than Zero

31 – 26	25 – 21	20 – 16	15 – 0
BCOND	rs	BLTZ	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction.

Operation:

T :	target ← $(\text{offset}_{15})^{14}$ ‖ offset ‖ 0^2
	condition ← $(\text{GPR}[rs]_{31} = 1)$
T + 1 :	if condition then
	PC ← PC + target
	endif

Exception:

None.

BLTZAL

Branch On Less Than Zero And Link

31 — 26	25 — 21	20 — 16	15 — 0
BCOND	rs	BLTZAL	offset
6	5	5	16

Format:

BLTZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however.

Operation:

T :	target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
	condition ← $(GPR[rs]_{31} = 1)$
	GPR[31] ← PC + 8
T+1 :	if condition then
	PC ← PC + target
	endif

Exception:

None.

BNE — Branch On Not Equal

31 … 26	25 … 21	20 … 16	15 … 0
BNE	rs	rt	offset
6	5	5	16

Format:

BNE rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 18-bit offset, appended zeros to the lower 2 bits of the 16-bit offset. The contents of general register rs and the contents of general register rt are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T :      target ← (offset15)^14 || offset || 0^2
             condition ← (GPR[rs] ≠ GPR[rt])
T + 1 :  if condition then
             PC ← PC + target
         endif
```

Exception:

None.

BREAK Break

31 — 26	25 — 6	5 — 0
SPECIAL	code	BREAK
6	20	6

Format:

BREAK code

Description:

A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    BreakpointException ;
```

Exception:

Breakpoint exception

CFCz

Move Control From Coprocessor

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
COPz	CF	rt	rd	0
6	5	5	5	11

Format:

CFCz rt, rd

Description:

The contents of coprocessor control register rd of coprocessor unit z are loaded into general register rt.

Operation:

T :	data ← CCR[z, rd]
T+1 :	GPR[rt] ← data

Exception:

Coprocessor unusable exception

COPz Coprocessor Operation

31 ... 26	25	24 ... 0
COPz	CO	cofun
6	1	25

Format:

COPz cofun

Description:

A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache/memory system.

Operation:

T:	Coprocessor Operation(z.cofun);

Exception:

Coprocessor unusable exception

CTCz **Move Control to Coprocessor**

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COPz	CT	rt	rd	0
6	5	5	5	11

Format:

CTCz rt, rd

Description:

The contents of general register rt are loaded into control register rd of coprocessor unit z.

Operation:

T :	data ← GPR[rt]
T+1 :	CCR[z, rd] ← data

Exception:

Coprocessor unusable exception

DIV Divide

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL	rs	rt	0	DIV
6	5	5	10	6

Format:

DIV rs, rt

Description:

The contents of general register rs are divided by the contents of general register rt, treating both operands as 32-bit two's complement values. No overflow exception occurs under any circumstances.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI. The MFHI and MFLO instructions are interlocked so that any attempt to read them before operations have completed will cause execution of instructions to be delayed until the operation finishes.

Divide operations are performed by a separate, autonomous execution unit within the V_R3600. After a divide operation is started, execution of other instructions may continue in parallel. The multiply/divide unit continues to operate during cache miss and other delaying cycles in which no instructions are executed.

DIV

Divide
(Cont'd)

Operation:

T−2 :	LO ← undefined
	HI ← undefined
T−1 :	LO ← undefined
	HI ← undefined
T :	LO ← GPR[rs] div GPR[rt]
	HI ← GPR[rs] mod GPR[rt]

Exception:

None.

DIVU Divide Unsigned

31 — 26	25 — 21	20 — 16	15 — 6	5 — 0
SPECIAL	rs	rt	0	DIVU
6	5	5	10	6

Format:

DIVU rs, rt

Description:

The contents of general register rs are divided by the contents of general register rt, treating both operands as 32-bit two's unsigned values. No overflow exception occurs under any circumstances.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI. The MFHI and MFLO instructions are interlocked so that any attempt to read them before operations have completed will cause execution of instructions to be delayed until the operation finishes.

Divide operations are performed by a separate, autonomous execution unit within the V_R3600. After a divide operation is started, execution of other instructions may continue in parallel. The multiply/divide unit continues to operate during cache miss and other delaying cycles in which no instructions are executed.

DIVU

Divide Unsigned
(Cont'd)

Operation:

T−2 :	LO ← undefined
	HI ← undefined
T−1 :	LO ← undefined
	HI ← undefined
T :	LO ← (0 ‖ GPR[rs]) div (0 ‖ GPR[rt])
	HI ← (0 ‖ GPR[rs]) mod (0 ‖ GPR[rt])

Exception:

None.

J **Jump**

31 — 26	25 — 0
J	target
6	26

Format:

J target

Description:

The 26-bit target address is shifted left two bits, combined with the high order 4 bits of the current program counter, and the program unconditionally jumps to the calculated address, with a delay of one instruction.

Operation:

T :	temp ← target
T + 1 :	$PC \leftarrow PC_{31...28} \parallel temp \parallel 0^2$

Exception:

None.

JAL — Jump And Link

31 … 26	25 … 0
J	target
6	26

Format:

JAL target

Description:

The 26-bit target address is shifted left two bits, combined with the high order 4 bits of the current program counter, and the program unconditionally jumps to the calculated address, with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, r31.

Operation:

T:	temp ← target
	GPR[31] ← PC + 8
T+1:	$PC \leftarrow PC_{31...28} \| temp \| 0^2$

Exception:

None.

JALR — Jump And Link Register

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	0	rd	0	JALR
6	5	5	5	5	6

Format:

JALR rs
JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general register rs, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register rd. The default value of rd, if omitted in the assembly language instruction, is 31.

Register specifiers rs and rd may not be equal, because such an instruction does not have the same effect when re-executed. However, an attempt to execute this instruction is not trapped; the result of executing such an instruction is undefined.

Operation:

T:	temp ← GPR[rs]
	GPR[rd] ← PC + 8
T+1:	PC ← temp

JALR — Jump And Link Register (Cont'd)

Exception:

None.

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

JR **Jump Register**

31–26	25–21	20–6	5–0
SPECIAL	rs	0	JR
6	5	15	6

Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register rs, with a delay of one instruction.

Operation:

T :	temp ← GPR[rs]
T + 1 :	PC ← temp

Exception:

None.

Since instructions must be word-aligned, a Jump Register instruction must specify a target register (rs) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

LB Load Byte

31 26	25 21	20 16	15 0
LB	base	rt	offset
6	5	5	16

Format:

LB rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register rt.

Operation:

T :	vAddress ← $(\text{offset}_{15})^{16}$ ‖ $\text{offset}_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
	byte ← $\text{vAddr}_{1..0}$ xor BigEndianCPU^2
	GPR[rt] ← undefined
T+1 :	GPR[rt] ← $(\text{mem}_{7+8*\text{byte}})^{24}$ ‖ $\text{mem}_{7+8*\text{byte}..8*\text{byte}}$

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LBU — Load Byte Unsigned

31 — 26	25 — 21	20 — 16	15 — 0
LBU	base	rt	offset
6	5	5	16

Format:

LBU rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register rt.

Operation:

T :	vAddress ← $(offset_{15})^{16}$ ‖ $offset_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
	byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
	GPR[rt] ← undefined
T+1 :	GPR[rt] ← 0^{24} ‖ $mem_{7+8*byte..8*byte}$

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LH Load Halfword

31 — 26	25 — 21	20 — 16	15 — 0
LH	base	rt	offset
6	5	5	16

Format:

LH rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register rt.

If the least significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
T :     vAddress ← (offset15)^16 || offset15..0 + GPR[base]
        (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
        mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
        byte ← vAddr1..0 xor (BigEndianCPU || 0)
        GPR[rt] ← undefined
T+1 :   GPR[rt] ← (mem15+8*byte)^16 || mem15+8*byte..8*byte
```

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LHU — Load Halfword Unsigned

31–26	25–21	20–16	15–0
LHU	base	rt	offset
6	5	5	16

Format:

LHU rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general resister rt.

If the least significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

T:	vAddress ← $(\text{offset}_{15})^{16}$ ‖ $\text{offset}_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
	byte ← $\text{vAddr}_{1..0}$ xor (BigEndianCPU ‖ 0)
	GPR[rt] ← undefined
T+1:	GPR[rt] ← 0^{16} ‖ $\text{mem}_{15+8*\text{byte}..8*\text{byte}}$

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LUI — Load Upper Immediate

31 — 26	25 — 21	20 — 16	15 — 0
LUI	0	rt	immediate
6	5	5	16

Format:

LUI rt, immediate

Description:

The 16-bit immediate is shifted left 16 bits and concatenated to 16 bits of zeroes. The result is placed into general register rt.

Operation:

T:	GPR[rt] ← immediate ∥ 0^{16}

Exception:

None.

LW | Load Word

31–26	25–21	20–16	15–0
LW	base	rt	offset
6	5	5	16

Format:

LW rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the word at the memory location specified by the effective address are loaded into general register rt.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

T :	vAddr ← $(offset_{15})^{16}$ ‖ $offset_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
	GPR[rt] ← undefined
T+1 :	GPR[rt] ← mem

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LWCz — Load Word To Coprocessor

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LWCz	base	rt	offset
6	5	5	16

Format:

LWCz rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of the word at the memory location specified by the effective address are loaded into coprocessor register rt of coprocessor unit z.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

T :	vAddr ← $(offset_{15})^{16}$ ‖ $offset_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
	CPR[z, rt] ← undefined
T + 1 :	CPR[z, rt] ← mem

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception
Coprocessor unusable exception

LWL — Load Word Left

31–26	25–21	20–16	15–0
LWL	base	rt	offset
6	5	5	16

Format:

LWL rt, offset (base)

Description:

This instruction can be used in combination with LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWL loads the left portion of the register from the appropriate part of the high-order word; LWR loads the right portion of the register from the appropriate part of the low-order word.

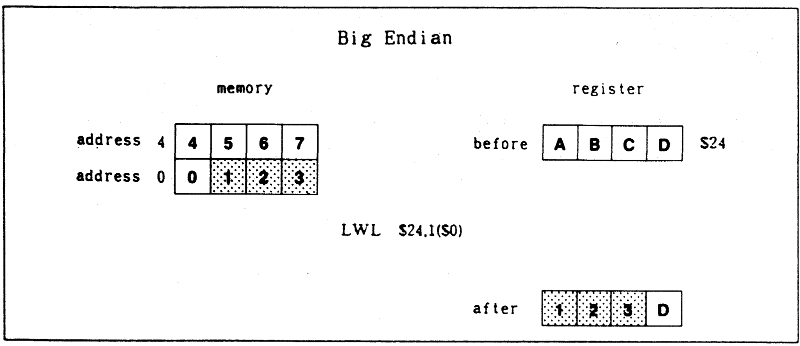
The LWL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a 32-bit unsigned effective address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified.

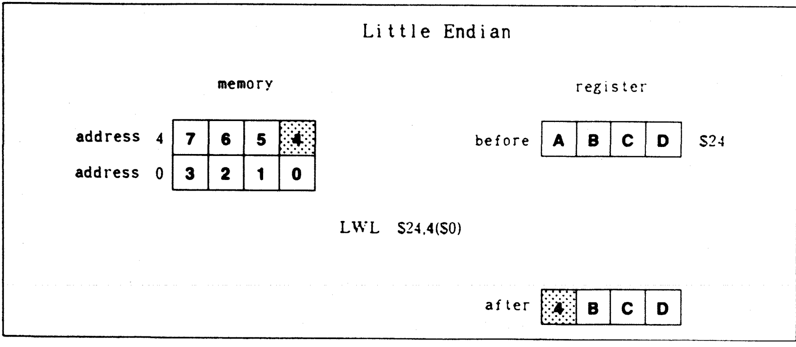
The contents of general register rt are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register rt and a following LWL instruction which also specifies register rt.

Address error exceptions due to byte alignment are suppressed by this instruction.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it proceeds toward the low-order byte of the word in memory

and the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the word in memory. The low-order (right-most) byte(s) of the register will not be changed.





LWL

Load Word Left
(Cont'd)

Operation:

T :	$vAddr \leftarrow (offset_{15})^{16} \| offset_{15..0} + GPR[base]$
	$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
	$byte \leftarrow vAddr_{1..0}$ xor $BigEndianCPU^2$
	if BigEndianMem = 0 then
	$pAddr \leftarrow pAddr_{31..2} \| 0^2$
	endif
	$mem \leftarrow LoadMemory(uncached, byte, pAddr, vAddr, DATA)$
T+1 :	$GPR[rt] \leftarrow mem_{7+8*byte..0} \| GPR[rt]_{23-8*byte..0}$

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

LWR Load Word Right

31 — 26	25 — 21	20 — 16	15 — 0
LWR	base	rt	offset
6	5	5	16

Format:

LWR rt, offset (base)

Description:

This instruction can be used in combination with LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWR loads the right portion of the register from the appropriate part of the low-order word; LWL loads the left portion of the register from the appropriate part of the high-order word.

The LWR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a 32-bit unsigned effective address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified.

The contents of general register rt are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register rt and a following LWR instruction which also specifies register rt.

Address error exceptions due to byte alignment are suppressed by this instruction.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it proceeds toward the high-order byte of the word in memory

and the high-order byte of the register, loading bytes from memory into the register until it reaches the high-order byte of the word in memory. The high-order (left-most) byte(s) of the register will not be changed.

Big Endian

memory

address 4	4	5	6	7
address 0	0	1	2	3

register

before	A	B	C	D	$24

LWR $24,4($0)

after	A	B	C	4	$24

Little Endian

memory

address 4	7	6	5	4
address 0	3	2	1	0

register

before	A	B	C	D	$24

LWR $24,1($0)

after	A	3	2	1	$24

LWR

Load Word Right
(Cont'd)

Operation:

T :	vAddr ← $(offset_{15})^{16}$ ‖ $offset_{15..0}$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
	if BigEndianMem = 1 then
	pAddr ← $pAddr_{31..2}$ ‖ 0^2
	endif
	mem ← LoadMemory (uncached, WORD-byte, pAddr, vAddr, DATA)
T+1 :	GPR[rt] ← $mem_{31..32-8*byte}$ ‖ $mem_{31..8*byte}$

Exception:

UTLB miss fault
TLB miss fault
Bus error exception
Address error exception

MFC0 — Move From System Control Coprocessor

31–26	25–21	20–16	15–11	10–0
COP0	MF	rt	rd	0
6	5	5	5	11

Format:

MFC0 rt, rd

Description:

The contents of coprocessor register rd of System Control Coprocessor (CP0) are loaded into general register rt.

Operation:

T :	data ← CPR[0, rd]
T+1 :	GPR[rt] ← data

Exception:

Coprocessor unusable exception

MFCz **Move From Coprocessor**

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
COPz	MF	rt	rd	0
6	5	5	5	11

Format:

MFCz rt, rd

Description:

The contents of coprocessor register rd of coprocessor unit z are loaded into general register rt.

Operation:

T :	data ← CPR[z, rd]
T + 1 :	GPR[rt] ← data

Exception:

Coprocessor unusable exception

MFHI **Move From HI**

31–26	25–16	15–11	10–6	5–0
SPECIAL	0	rd	0	MFHI
6	10	5	5	6

Format:

MFHI rd

Description:

The contents of special register HI are loaded into general register rd.

To ensure proper operation in the event of interruptions, the two instructions which follow an MFHI instruction may not be any of the instructions which modify the HI register: MULT, MULTU, DIV, DIVU, MTHI.

Operation:

T:	GPR[rd] ← HI

Exception:

None.

MFLO Move From LO

31 26	25 16	15 11	10 6	5 0
SPECIAL	0	rd	0	MFLO
6	10	5	5	6

Format:

MFLO rd

Description:

The contents of special register LO are loaded into general register rd.

To ensure proper operation in the event of interruptions, the two instructions which follow an MFLO instruction may not be any of the instructions which modify the LO register: MULT, MULTU, DIV, DIVU, MTLO.

Operation:

T:	GPR[rd] ← LO

Exception:

None.

MTC0 | Move To System Control Coprocessor

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
COP0	MT	rt	rd	0
6	5	5	5	11

Format:

MTC0 rt, rd

Description:

The contents of general register rt are loaded into coprocessor register rd of the System Control Coprocessor (CP0).

Because the state of the virtual address translation system may be altered by this instruction, the operation of load and store instructions and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
T:      data ← GPR[rt]
T+1 :   CPR[0,rd] ← data
```

Exception:

Coprocessor unusable exception

MTCz **Move To Coprocessor**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
COPz	MT	rt	rd	0
6	5	5	5	11

Format:

MTCz rt, rd

Description:

The contents of general register rt are loaded into coprocessor register rd of coprocessor unit z.

Operation:

T :	data ← GPR[rt]
T+1 :	CPR[z, rd] ← data

Exception:

Coprocessor unusable exception

MTHI — Move To HI

31–26	25–21	20–6	5–0
SPECIAL	rs	0	MTHI
6	5	15	6

Format:

MTHI rs

Description:

The contents of general register rs are loaded into special register HI.

If an MTHI operation is executed following an MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register LO are undefined.

Operation:

```
T-2 :  HI ← undefined
T-1 :  HI ← undefined
T :    HI ← GPR[rs]
```

Exception:

None.

MTLO

Move To LO

31 26	25 21	20 6	5 0
SPECIAL	rs	0	MTLO
6	5	15	6

Format:

MTLO rs

Description:

The contents of general register rs are loaded into special register LO.

If an MTLO operation is executed following an MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register HI are undefined.

Operation:

T−2 :	LO ← undefined
T−1 :	LO ← undefined
T :	LO ← GPR[rs]

Exception:

None.

MULT Multiply

31 26	25 21	20 16	15 6	5 0
SPECIAL	rs	rt	0	MULT
6	5	5	10	6

Format:

MULT rs, rt

Description:

The contents of general register rs and the contents of general register rt are multiplied, treating both operands as 32-bit two's complement values. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI. The MFHI and MFLO instructions are interlocked so that any attempt to read them before operations have completed will cause execution of instructions to be delayed until the operation finishes.

Multiply operations are performed by a separate, autonomous execution unit within the V_R3600. After a multiply operation is started, execution of other instructions may continue in parallel. The multiply/divide unit continues to operate during cache miss and other delaying cycles in which no instructions are executed.

MULT

Multiply (Cont'd)

Operation:

T − 2 :	LO ← undefined
	HI ← undefined
T − 1 :	LO ← undefined
	HI ← undefined
T :	t ← GPR[rs] * GPR[rt]
	LO ← $t_{31..0}$
	HI ← $t_{63..32}$

Exception:

None.

MULTU — Multiply Unsigned

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	MULTU
6	5	5	10	6

Format:

MULTU rs, rt

Description:

The contents of general register rs and the contents of general register rt are multiplied, treating both operands as 32-bit unsigned values. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI. The MFHI and MFLO instructions are interlocked so that any attempt to read them before operations have completed will cause execution of instructions to be delayed until the operation finishes.

Multiply operations are performed by a separate, autonomous execution unit within the V_R3600. After a multiply operation is started, execution of other instructions may continue in parallel. The multiply/divide unit continues to operate during cache miss and other delaying cycles in which no instructions are executed.

MULTU

Multiply Unsigned
(Cont'd)

Operation:

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 ← $t_{63..32}$

Exception:

None.

NOR Nor

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	NOR
6	5	5	5	5	6

Format:

NOR rd, rs, rt

Description:

The contents of general register rs are combined with the contents of general register rt in a bit-wise logical NOR operation. The result is placed into general register rd.

Operation:

T:	GPR[rd] ← GPR[rs] nor GPR[rt]

Exception:

None.

OR Or

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

Format:

OR rd, rs, rt

Description:

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

Operation:

T:	GPR[rd] ← GPR[rs] or GPR[rt]

Exception:

None.

ORI Or Immediate

31 — 26	25 — 21	20 — 16	15 — 0
ORI	rs	rt	immediate
6	5	5	16

Format:

ORI rt, rs, immediate

Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical OR operation. The result is placed into general register rt.

Operation:

T:	$GPR[rt] \leftarrow GPR[rs]_{31..16} \| (immediate\ or\ GPR[rs]_{15..0})$

Exception:

None.

RFE Restore From Exception

31–26	25	24–5	4–0
COP0	CO	0	RFE
6	1	20	5

Format:

RFE

Description:

Restores the previous interrupt mask and kernel/user-mode bits (IEp and KUp) of the Status register (SR) into the corresponding current status bits (IEc and KUc), and restores the old status bits (IEo and KUo) into the corresponding status bits (IEp and KUp). The old status bits remain unchanged.

The operation of memory references associated with load/store instructions immediately prior to an RFE instruction are unspecified. Normally, the RFE instruction follows in the delay slot of a JR (jump register) instruction to restore the PC.

Operation:

T:	$SR \leftarrow SR_{31..4} \parallel SR_{5..2}$

Exception:

Coprocessor unusable exception

SB Store Byte

31 26	25 21	20 16	15 0
SB	base	rt	offset
6	5	5	16

Format:

SB rt, offset (base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The least significant byte of the contents of register rt is stored at the effective address.

Operation:

T: $vAddr \leftarrow (offset_{15})^{16} \parallel offset_{15..0} + GPR[base]$

$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

$pAddr \leftarrow pAddr_{PSIZE-1..2} \parallel (pAddr_{1..0}\ xor\ ReverseEndian^2)$

$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$

$data \leftarrow GPR[rt]_{31-8*byte..0} \parallel 0^{8*byte}$

StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception

SH Store Halfword

31 – 26	25 – 21	20 – 16	15 – 0
SH	base	rt	offset
6	5	5	16

Format:

SH rt, offset (base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The least significant halfword of the contents of register rt is stored at the effective address.

If the least significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

T: $vAddr \leftarrow (offset_{15})^{16} \| offset_{15..0} + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ (ReverseEndian \| 0))$
$byte \leftarrow vAddr_{1..0}\ xor\ (BigEndianCPU \| 0)$
$data \leftarrow GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception

SLL **Shift Left Logical**

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	0	rt	rd	sa	SLL
6	5	5	5	5	6

Format:

SLL rd, rt, sa

Description:

The contents of general register rt are shifted left by sa bits, inserting zeroes into the low-order bits. The 32-bit result is placed in register rd.

Operation:

T:	$GPR[rd] \leftarrow GPR[rt]_{31-sa..0} \parallel 0^{sa}$

Exception:

None.

SLLV Shift Left Logical Variable

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

Format:

SLLV rd, rt, rs

Description:

The contents of general register rt are shifted left by the number of bits specified by the low-order 5 bits of the contents of general register rs, inserting zeroes into the low order bits. The 32-bit result is placed in register rd.

Operation:

T: $s \leftarrow GPR[rs]_{4..0}$
$GPR[rd] \leftarrow GPR[rt]_{(31-s)..0} \| 0^{s}$

Exception:

None.

SLT Set On Less Than

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

Format:

SLT rd, rs, rt

Description:

The contents of general register rt are compared with the contents of general register rs. Considering both quantities as signed 32-bit integers, if the contents of general register rs are less than the contents of general register rt, the result is set to one, otherwise the result is set to zero. The result is placed into general register rd.

No overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:      if GPR[rs] < GPR[rt] then
            GPR[rd] ← 0^31 || 1
        else
            GPR[rd] ← 0^32
        endif
```

Exception:

None.

SLTI Set On Less Than Immediate

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

Format:

SLTI rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and compared with the contents of general register rs. Considering both quantities as signed 32-bit integers, if rs is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:    if GPR[rs] < ((immediate_15)^16 || immediate_15..0) then
          GPR[rt] ← 0^31 || 1
      else
          GPR[rt] ← 0^32
      endif
```

Exception:

None.

SLTIU

Set On Less Than
Immediate Unsigned

31 — 26	25 — 21	20 — 16	15 — 0
SLTIU	rs	rt	immediate
6	5	5	16

Format:

SLTIU rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and compared with the contents of general register rs. Considering both quantities as unsigned 32-bit integers, if rs is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflow.

Operation:

T:	if $(0 \parallel GPR[rs]) < (0 \parallel (immediate_{15})^{16} \parallel immediate_{15..0})$
	then
	    $GPR[rt] \leftarrow 0^{31} \parallel 1$
	else
	    $GPR[rt] \leftarrow 0^{32}$
	endif

Exception:

None.

SLTU Set On Less Than Unsigned

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

Format:

SLTU rd, rs, rt

Description:

The contents of general register rt are compared with the contents of general register rs. Considering both quantities as unsigned 32-bit integers, if the contents of general register rs are less than the contents of general register rt, the result is set to one, otherwise the result is set to zero. The result is placed into general rd.

No overflow exception occurs under any circumstances. The comparison valid even if the subtraction used during the comparison overflows.

Operation:

```
T:    if (0 || GPR[rs]) < (0 || GPR[rt]) then
          GPR[rd] ← 0^31 || 1
      else
          GPR[rd] ← 0^32
      endif
```

Exception:

None.

SRA Shift Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL	0	rt	rd	sa	SRA
6	5	5	5	5	6

Format:

SRA rd, rt, sa

Description:

The contents of general register rt are shifted right by sa bits, sign-extending the high-order bits. The 32-bit result is placed in register rd.

Operation:

T: $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} \parallel GPR[rt]_{31..sa}$

Exception:

None.

SRAV — Shift Right arithmetic Variable

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

Format:

SRAV rd, rt, rs

Description:

The contents of general register rt are shifted right by the member of bit specified by the low-order 5 bits of the contents of general register rs, sign-extending the high order bits. The 32-bit result is placed in register rd.

Operation:

T: $s \leftarrow GPR[rs]_{4..0}$

$GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \| GPR[rt]_{31..s}$

Exception:

None.

SRL Shift Right Logical

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL	0	rt	rd	sa	SRL
6	5	5	5	5	6

Format:

SRL rd, rt, sa

Description:

The contents of general register rt are shifted right by sa bits, inserting zeroes into the high order bits. The 32-bit result is placed in register rd.

Operation:

T: $GPR[rd] \leftarrow 0^{sa} \parallel GPR[rt]_{31..sa}$

Exception:

None.

SRLV

Shift Right Logical Variable

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

Format:

SRLV rd, rt, rs

Description:

The contents of general register rt are shifted right by the number of bits specified by the low-order 5 bits of the contents of general register rs, inserting zeroes into the high order bits. The 32-bit result is placed in register rd.

Operation:

T:	$s \leftarrow GPR[rs]_{4..0}$
	$GPR[rd] \leftarrow 0^{s} \parallel GPR[rt]_{31..s}$

Exception:

None.

SUB Substract

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

Format:

SUB rd, rs, rt

Description:

The contents of general register rt are subtracted from the contents of general register rs to form a 32-bit result. The result is placed into general register rd.

An overflow exception occurs if the two highest order carry-out bits differ (two's complement overflow).

Operation:

T:	GPR[rd] ← GPR[rs] − GPR[rt]

Exception:

Overflow exception

SUBU

Subtract Unsigned

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

Format:

SUBU rd, rs, rt

Description:

The contents of general register rt are subtracted from the contents of general register rs to form a 32-bit result. The result is placed into general register rd.

No overflow exception occurs under any circumstances.

Note that the only difference between this instruction and the SUB instruction is that SUBU never causes an overflow exception.

Operation:

T:	GPR[rd] ← GPR[rs] − GPR[rt]

Exception:

None.

SW Store Word

31 — 26	25 — 21	20 — 16	15 — 0
SW	base	rt	offset
6	5	5	16

Format:

SW rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of general register rt are stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

T: $vAddr \leftarrow (offset_{15})^{16} \| offset_{15..0} + GPR[base]$
$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
$data \leftarrow GPR[rt]$
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception

SWCz   Store Word From Coprocessor

31 — 26	25 — 21	20 — 16	15 — 0
SWCz	base	rt	offset
6	5	5	16

Format:

SWCz rt, offset (base)

Description:

The 16-bit offset is added to the contents of general register base to form a 32-bit effective address. The contents of coprocessor register rt of coprocessor unit z are stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

T: vAddr ← $((\text{offset}_{15})^{16} \| \text{offset}_{15..0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
data ← CPR[z, rt]
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception
Coprocessor unsable exception

SWL — Store Word Left

31–26	25–21	20–16	15–0
SWL	base	rt	offset
6	5	5	16

Format:

SWL rt, offset (base)

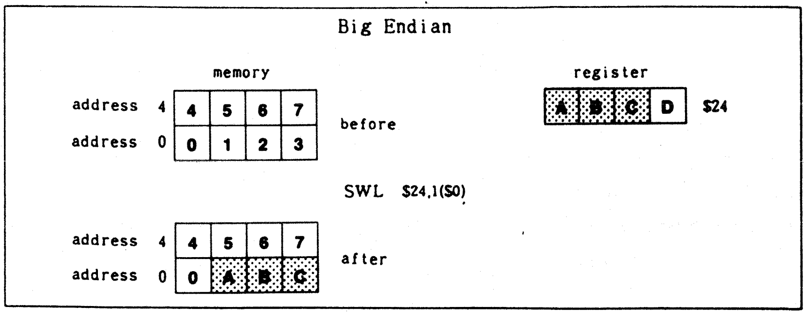
Description:

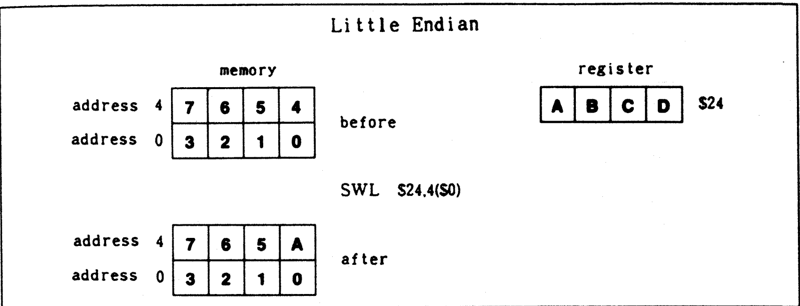
This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWL stores the left portion of the register into the appropriate part of the high-order word of memory; SWR stores the right portion of the register into the appropriate part of the low-order word.

The SWL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a 32-bit unsigned effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the high-order byte of the register and copies it to the specified byte in memory, then it proceeds toward the low-order byte of the register and the low-order byte of the word in memory, copying bytes from register to memory until it reaches the low-order byte of the word in memory.

SWL — Store Word Left (Cont'd)





Operation:

T : $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$

$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ ReverseEndian^2)$

if $BigEndianMem = 0$ then

  $pAddr \leftarrow pAddr_{31..2} \| 0^2$

endif

$byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$

$data \leftarrow 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte}$

StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)

SWL Store Word Left (Cont'd)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception

SWR Store Word Right

31 — 26	25 — 21	20 — 16	15 — 0
SWR	base	rt	offset
6	5	5	16

Format:

SWR rt, offset (base)

Description:

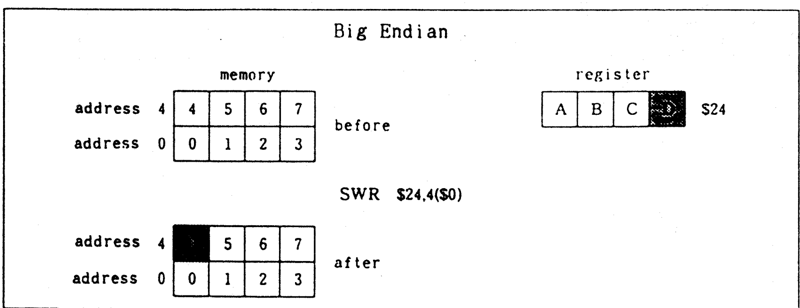
This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low-order word of memory; SWL stores the left portion of the register into the appropriate part of the low-order of word.

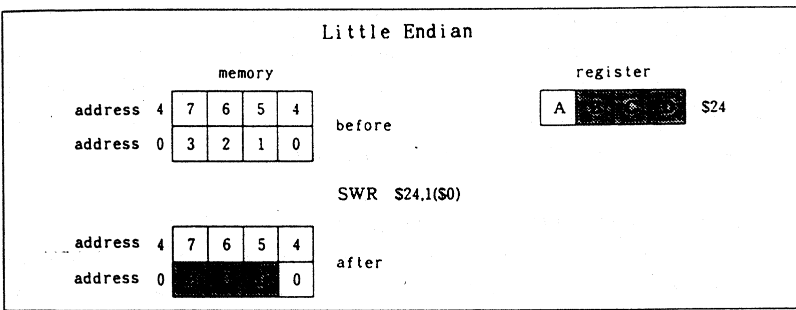
The SWR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a 32-bit unsigned effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the low-order (right-most) byte of the register and copies it to the specified byte in memory, then it proceeds toward the high-order byte of the register and the high-order byte of the word in memory, copying bytes from register to memory until it reaches the high-order byte of the word in memory.

SWR

Store Word Right (Cont'd)





Operation:

T : $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$

$(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$

$pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0} \text{ xor } ReverseEndian^2)$

if $BigEndianMem = 1$ then

  $pAddr \leftarrow pAddr_{31..2} \| 0^2$

endif

$byte \leftarrow vAddr_{1..0} \text{ xor } BigEndianCPU^2$

$data \leftarrow GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$

StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)

SWR

Store Word Right (Cont'd)

Exception:

UTLB miss fault
TLB miss fault
TLB modification fault
Bus error exception
Address error exception

SYSCALL System Call

31 — 26	25 — 6	5 — 0
SPECIAL	0	SYSCALL
6	20	6

Format:

SYSCALL

Description:

A system call trap occurs, immediately and unconditionally transferring control to the exception handler.

Operation:

```
SystemCallException
```

Exception:

System Call trap

TLBP — Probe TLB For Matching Entry

31 – 26	25	24 – 5	4 – 0
COP0	CO	0	TLBP
6	1	20	5

Format:

TLBP

Description:

The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHI and EntryLO registers. If no TLB entry matches, the high-order bit of the Index register is set.

If more than one TLB entry matches, the results of this instruction are not specified. Additionally, the operation of memory references associated with the instruction immediately following a TLBP instruction is unspecified.

Operation:

T: $Index \leftarrow 1 \| 0^{31}$
for i in 0..TLBEntries − 1
if $((TLB[i]_{63..44} = EntryHi_{31..12})$ and
$(TLB[i]_8$ or $(TLB[i]_{43..38} = EntryHi_{11..6})))$ then
$Index \leftarrow 0^{18} \| i_{5..0} \| 0^8$
endif
endfor

Exception:

Coprocessor unusable exception

TLBR Read Indexed TLB Entry

31 ... 26	25	24 ... 5	4 ... 0
COP0	CO	0	TLBR
6	1	20	5

Format:

TLBR

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 operation is invalid (and the results are unspecified) if the contents of the TLB Index register are greater than the number of TLB entries in the processor.

Operation:

T: $EntryHi \leftarrow TLB[Index_{13..8}]_{63..32}$
$EntryLo \leftarrow TLB[Index_{13..8}]_{31..0}$

Exception:

Coprocessor unusable exception

TLBWI Write Indexed TLB Entry

31–26	25	24–5	4–0
COP0	CO	0	TLBWI
6	1	20	5

Format:

TLBWI

Description:

The TLB entry pointed at by the contents of the TLB Index register is loaded with the contents of the EntryHI and EntryLO registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB Index register are greater than the number of TLB entries in the processor.

Operation:

T:	$TLB[Index_{13..8}] \leftarrow EntryHi \parallel EntryLo$

Exception:

Coprocessor unusable exception

TLBWR Write Random TLB Entry

31 ... 26	25	24 ... 5	4 ... 0
COP0	CO	0	TLBWR
6	1	20	5

Format:

TLBWR

Description:

The TLB entry pointed at by the contents of the TLB Random register is loaded with the contents of the EntryHI and EntryLO registers.

Operation:

T:	$TLB[Random_{13..8}] \leftarrow EntryHi \parallel EntryLo$

Exception:

Coprocessor unusable exception

XOR — Exclusive Or

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

Format:

XOR rd, rs, rt

Description:

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

Operation:

T:	GPR[rd] ← GPR[rs] xor GPR[rt]

Exception:

None.

XORI   Exclusive Or Immediate

31 — 26	25 — 21	20 — 16	15 — 0
XORI	rs	rt	immediate
6	5	5	16

Format:

XORI rt, rs, immediate

Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical exclusive-OR operation. The result is placed into general register rt.

Operation:

T:	GPR[rt] ← GPR[rs] xor (0^{16} ‖ immediate)

Exception:

None.

Part 3 FPU Architecture

Chapter 1 V_RFPU Overview

The V_R3600 Floating-Point Unit (FPU) operates as a coprocessor for the V_R3600 Processor and extends the V_R3600's instruction set to perform arithmetic operations on values in floating-point representations. The V_R3600 FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic." Figure 1-1 illustrates the functional organization of the FPU.

Fig. 1-1 V_R3600 FPU Functional Block Diagram

cache data

Data Bus

instructions

Control unit

operands

Register unit

exponent part

fraction

11 11 11

53 53 53

A B result

Exponent unit

A B result

Add unit

53 53 56

A B result

Divide unit

53 53 56

A B result

Multiply unit

1.1 V_R3600 FPU Features

- V_R3600 FPU is software compatible with V_R3010™ and V_R3010A.

- Full 64-bit Operation. The V_R3600 FPU contains sixteen, 64-bit registers that can each be used to hold single-precision or double-precision values. The FPU also includes a 32-bit status/control register that provides access to all IEEE-Standard exception handling capabilities.

- Load/Store Instruction Set. Like the V_R3600 Processor, the V_R3600 FPU uses a load/store-oriented instruction set, with single-cycle loads and stores. Floating-point operations are started in a single cycle and their execution is overlapped with other fixed point or floating-point operations.

- Tightly-coupled Coprocessor Interface - the FPU connects to the V_R3600 Processor to form a tightly-coupled unit with a seamless integration of floating-point and fixed-point instruction sets. Since each unit receives and executes instructions in parallel, some floating-point instructions can execute at the same single-cycle per instruction rate as fixed point-instructions.

1.2 V_R3600 FPU Programming Model

This section describes the organization of data in registers and in memory and the set of general registers available. This section also gives a summary description of all the V_R3600 FPU registers.

The V_R3600 FPU provides three types of registers as shown in Figure 6.2:

(1) Floating-Point General-Purpose Registers (FGR)
(2) Floating-Point Registers (FPR)
(3) Floating-Point Control Registers (FCR)

Fig. 1-2 V_R3600 FPU Registers

Floating-Point Registers (FPR)

Floating-Point General-Purpose Registers (FGR)

FPR		31 (FGR) 0
FPR0	least	FGR0
	most	FGR1
FPR2	least	FGR2
	most	FGR3
. . .		. . .
FPR28	least	FGR28
	most	FGR29
FPR30	least	FGR30
	most	FGR31

Floating-Point Control Registers (FCR)

Control/Status Register (bits 31 … 0)

Implementation/Revision Register (bits 31 … 0)

Floating-Point General-Purpose Registers (FGR) are directly addressable, physical registers. The FPU provides thirty-two 32-bit FGRs.

Floating-Point Registers (FPR) are logical registers used to store data values during floating-point operations. Each of the 16 FPRs is 64 bits wide and is formed by concatenating two

adjacent FGRs. Depending on the requirements of an operation, FPRs hold either single- or double-precision floating-point values.

Floating-Point Control Registers are used for rounding mode control, exception handling, and state saving. The FCRs include the Control/Status register and the Implementation/Revision register.

(1) Floating-Point General Registers

The 32 Floating-Point General Registers (FGRs) on the FPU are directly addressable 32-bit registers used in floating-point operations and individually accessible via move, load, and store instructions. The FGRs are listed in Table 1-1, and the Floating-Point Registers (FPRs) that are logically formed by the general registers are described in the section that follow.

Table 1-1 Floating-Point General Registers

FGR Number	Usage
0	FPR 0 (least)
1	FPR 0 (most)
2	FPR 2 (least)
3	FPR 2 (most)
.	.
.	.
.	.
28	FPR 28 (least)
29	FPR 28 (most)
30	FPR 30 (least)
31	FPR 30 (most)

(2) Floating-Point Registers

The V_R3600 provides 16 Floating-Point Registers (FPR). These logical 64-bit registers hold floating-point values during floating-point operations and are physically formed from the

General-Purpose Registers (FGR).

The FPRs hold values in either single- or double-precision floating-point format. Only even numbers are used to address FPRs: odd FPR register numbers are invalid. During single-precision floating-point operations, only the even-numbered (least) general registers are used, and during double-precision floating-point operations, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting Floating-Point Register 0 (FPR0) addresses adjacent Floating-Point General-Purpose Registers FGR0 and FGR1.

(3) Floating-Point Control Registers

The FPU coprocessor implements two Floating-Point Control Registers (FCRs). These registers can be accessed only by Move operations and contain the following:

- The Control/Status register (FCR31), is used to control and monitor exceptions, hold result of compare operations and establish rounding modes; and

- The Implementation/Revision register (FCR0), holds revision information about the FPU.

(a) Control/Status Register (Read and Write)

The Control/Status register, FCR31, contains control and status data and can be accessed by instructions running in either Kernel or User mode. It control the arithmetic rounding mode and the enabling of exceptions. It also indicates exceptions that occurred in the most recently executed instruction, and all exceptions that have occurred since the register was cleared. Figure 1-3 shows the bit assignments.

Fig. 1-3 Control/Status Register Bit Assignments

31–24	23	22–18	17–12	11–7	6–2	1–0
RFU	C	RFU	Exceptions EVZOUI	TrapEnable VZOUI	StickyBits VZOUI	RM
8	1	5	6	5	5	2

C	Condition bit. Set/cleared to reflect result of Compare instruction and drives the FPU's CpCond output signal.
Exceptions	These bits are set to indicate any exceptions that occurred during the most recent instruction.
Trap Enable	Trap Enables. These bits enable assertion of the FpInt signal if the corresponding Exception bit is set during a floating-point operation.
Sticky bits	These bits are set if an exception occurs and are reset only by explicitly loading new settings into this register (with a Move instruction)
RM	Rounding Mode. These two bits specify which of the four rounding modes is to be used by the FPU.
RFU	Reserved. Currently ignores writes, undefined when read.

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

The bits in the Control/Status register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. This register must only be written to when the FPU is not actively executing floating-point operations: this can be assured by first reading the contents of the register to empty the pipeline.

(i) Control/Status Register Condition Bit

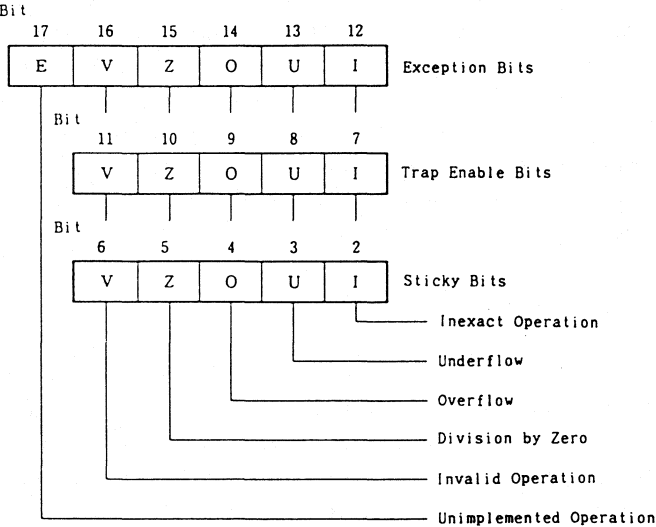
(i) Control/Status Register Condition Bit

Bit 23 of the Control/Status register is the Condition bit. When a floating-point Compare operation takes place, the detected condition is placed at bit 23, so that the state of the condition line may be saved or restored. The "C" bit is set (1) if the condition is true and cleared (0) if the condition is false. Bit 23 is affected only by Compare and Move Control To FPU instructions.

(ii) Control/Status Register Exception Bits

Bits 17:12 in the Control/Status register contains Exception bits as shown in Figure 1-4 that reflect the results of the most recently executed instruction. These bits are appropriately set or cleared after each floating-point operation. Exception bits are set for instructions that cause one of the five IEEE standard exceptions or the Unimplemented Operation exception.

Fig. 1-4 Control/Status Register Exception/
Sticky/Trap Enable Bits



If two exceptions occur together in one instruction, both of the appropriate bits in the exception bit field will be set. When an exception occurs, both the corresponding Exception and Sticky bits are set. Refer to CHAPTER 3, FLOATING POINT EXCEPTIONS, for a complete description of floating-point exceptions.

The Unimplemented Operation exception is not one of the standard IEEE-defined floating-point exceptions. It is provided to permit software implementation of IEEE standard operations and exceptions that are not fully supported by the FPU. Note that trapping on this exception cannot be disabled: there is no Trap Enable bit for E.

(iii) Control/Status Register Sticky Bits

The Sticky bits shown in Figure 1-4 hold the accumulated or accrued exception bits required by the IEEE standard for trap disabled operation. These bits are set whenever an FPU operation result causes one of the corresponding Exception bits to be set. However, unlike the Exception bits, the Sticky bits are never cleared as a side-effect of floating-point operations; they can be cleared only by writing a new value into the Control/Status register, using the Move Control To Coprocessor 1 (CTC1) instruction.

(iv) Control/Status Register Trap Enable Bits

The Trap Enable bits shown in Figure 1-4 are used to enable a user trap when an exception occurs during a floating-point operation. If the Trap Enable bit corresponding to the exception is set it causes assertion of the FPU's $\overline{\text{FpInt}}$ signal. The V_R3600 responds to the $\overline{\text{FpInt}}$ signal by taking an interrupt exception which can then be used to implement trap handling of the FPU exception.

(v) Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the Control/Status register comprise the Rounding Mode (RM) field. These bits specify the rounding mode that the FPU will use for all floating-point operations as shown in Table 1-2.

Table 1-2 Rounding Mode Bit Decoding

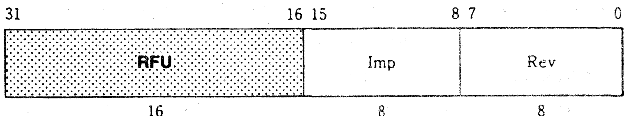
RM Bits	Mnemonic	Rounding Mode Description
00	RN	Rounds result to nearest representable value; round to value with least significant bit zero when the two nearest representable values are equally near.
01	RZ	Rounds result toward zero; round to value closest to and not greater in magnitude than the infinitely precise result.
10	RP	Rounds toward $+\infty$; round to value closest to and not less than the infinitely precise result.
11	RM	Rounds toward $-\infty$; round to value closest to and not greater than the infinitely precise result.

(b) Implementation and Revision Register (Read Only)

The FPU control register zero (FCR0) contains values that define the implementation and revision number of the V_R3600. This information can be used to determine the coprocessor revision and performance level and can also used by diagnostic software. NOTE: This register is intended to assist users in identifying version-specific characteristics of the FPU. However, due to the variety of levels at which design changes may be implemented to the silicon, the revision information cannot be guaranteed with every revision of the device nor assured to follow a completely predictable numerical sequence.

Only the low-order bytes of the implementation and revision register are defined. Bits 15 through 8 identify the implementation and bits 7 through 0 identify the revision number as shown in Figure 1-5.

Fig. 1-5 Implementation/Revision Register



Imp	;	Implementation: (03H)
Rev	;	Revision of FPU
RFU	;	Unused; ignored on writes, zero when read.

1.3 Floating-Point Formats

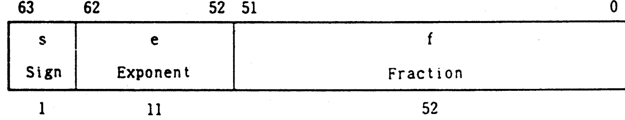
The V_R3600 performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit format has a 24-bit signed-magnitude fraction field and an 8-bit exponent, as shown in Figure 1-6.

Fig. 1-6 Single-Precision Floating-Point Format

31	30 … 23	22 … 0
s Sign	e Exponent	f Fraction
1	8	23

The 64-bit format has a 53-bit signed-magnitude fraction field and an 11-bit exponent, as shown in Figure 1-7.

Fig. 1-7 Double-Precision Floating-Point Format



Numbers in the single-precision and double-precision floating-point formats (extended and quad formats are not supported by V_R3600) are composed of three fields:

- A 1-bit sign: s,
- A biased exponent: e = E + bias, and
- A fraction: $f = b_1 b_2 \ldots b_{D-1}$

The range of the unbiased exponent E includes every integer between two values E_{min} and E_{max} inclusive, and also two other reserved values: $E_{min} - 1$ to encode ± 0 and denormalized numbers, and $E_{max} + 1$ to encode $\pm\infty$ and NaNs (Not a Number). For single- and double-precision formats, each representable non-zero numerical value has just one encoding.

For single-and double-precision formats, the value of a number, v, is determined by the equations shown in Table 1-3.

Table 1-3 Equations for Calculating Values in Floating-Point Format

(1)	if $E = E_{max} + 1$ and $f \neq 0$, then v is NaN, regardless of s.
(2)	if $E = E_{max} + 1$ and $f = 0$, then $v = (-1)^s \infty$.
(3)	if $E_{min} \leqq E \leqq E_{max}$, then $v = (-1)^s 2^E (1.f)$.
(4)	if $E = E_{min} - 1$ and $f \neq 0$, then $v = (-1)^s 2^E (0.f)$.
(5)	if $E = E_{min} - 1$ and $f = 0$, then $v = (-1)^s 0$.

For all floating-point formats, if v is NaN, the most significant bit of f determines whether the value is a signaling or quiet NaN. v is a signaling NaN if the most significant bit of f is set; otherwise v is a quiet NaN.

Table 1-4 defines the values for the format parameters in the preceding description.

Table 1-4 Floating-Point Format Parameter Values

Parameter	Single	Double
P	24	53
E_{max}	+127	+1023
E_{min}	-126	-1022
exponent bias	+127	+1023
exponent width in bits	8	11
integer bit	hidden	hidden
fraction width in bits	23	52
format width in bits	32	64

1.4 Number Definitions

This subsection contains a definition of the following number types specified in the IEEE 754 standard:

(1) Normalized Numbers
(2) Denormalized Numbers
(3) Infinity
(4) Zero

For more information, refer to the ANSI/IEEE Std 754-1985 IEEE Standard for Binary Floating-point Arithmetic.

(1) Normalized Numbers

Most floating-point calculations are performed on normalized numbers. For single-precision operations, normalized numbers have a biased exponent that ranges from 1 to 254 (-126 to +127 unbiased) and a normalized fraction field, meaning that the leftmost, or hidden, bit is one. In decimal notation, this allows representation of a range of positive and negative numbers from approximately 10^{38} to 10^{-38}, with accuracy to 7 decimal places.

(2) Denormalized Numbers

Denormalized numbers have a zero exponent and a denormalized (hidden bit equal to zero) non-zero fraction field.

(3) Infinity

Infinity has an exponent of all ones and a fraction field equal to zero. Both positive and negative infinity are supported.

(4) Zero

Zero has an exponent of zero, a hidden bit equal to zero, and a value of zero in the fraction field. Both +0 and -0 are supported.

1.5 Coprocessor Operation

The FPU continually monitors the V_R3600 Processor instruction stream. If an instruction does not apply to the coprocessor, it is ignored; if an instruction does apply to the coprocessor, the FPU executes that instruction and transfers necessary result and exception data synchronously to the V_R3600 main processor.

The FPU performs three types of operations:

(1) Loads and Stores;
(2) Moves;
(3) Two- and three register floating-point operations.

(1) Load and Store Operations

Load and Store operations load/store data between memory. These operations perform no format conversions and cause no floating-point exceptions. Load and Store operations reference a single 32-bit word of either the Floating-Point General Registers (FGR) or the Floating-Point Control Registers (FCR).

(2) Move Operations

Move operations move data between the V_R3600 Processor registers and the V_R3600 FPU register. These operations perform no format conversions and cause no floating-point exceptions. Move operations reference a single 32-bit word of either the Floating-Point General Registers (FGR) or the Floating-Point Control Registers (FCR)

(3) Floating-Point Operations

The V_R3600 supports the following single- and double-precision format floating-point operations:

- Add

- Subtract
- Multiply
- Divide
- Absolute Value
- Move
- Negate
- Compare

In addition, the V_R3600 supports conversions between single- and double-precision floating-point formats and fixed-point formats. Refer to CHAPTER 2, INSTRUCTION SET SUMMARY & INSTRUCTION PIPELINE for a complete description of all the FPU instructions.

Exceptions

The V_R3600 FPU supports all five IEEE standard exceptions:

- Invalid Operation
- Inexact Operation
- Division by Zero
- Overflow
- Underflow

The FPU also supports the optional, Unimplemented Operation exception that allows unimplemented instructions to trap to software emulation routines. For more information on exceptions, refer to CHAPTER 3, FLOATING POINT EXCEPTIONS.

1.6 Instruction Set Overview

All V_R3600 instructions are 32 bits long and they can be divided into the following groups:

(1) Load/Store and Move instructions move data between memory, the main processor and the FPU general registers.
(2) Computational instructions perform arithmetic operations on floating point values in the FPU registers.
(3) Conversion instructions perform conversion operations between the various data formats.
(4) Compare instructions perform comparisons of the contents of registers and set a condition bit based on the results.

Table 1-5 lists the instruction set of the V_R3600 FPU. A more detailed summary is contained in CHAPTER 2 INSTRUCTION SET SUMMARY & INSTRUCTION PIPELINE and complete description of each instruction is provided in CHAPTER 4, INSTRUCTION SET.

Table 1-5 V_R3600 Instruction Summary

OP	Description	OP	Description
Load/Store/Move Instructions		CVT.W.fmt	Floating-point Convert to fixed-point
LWC1	Load Word To FPU	Computational Instructions	
SWC1	Store Word From FPU	ADD.fmt	Floating-point Add
MTC1	Move Word To FPU	SUB.fmt	Floating-point Subtract
MFC1	Move Word From FPU	MUL.fmt	Floating-point Multiply
CTC1	Move Control Word To FPU	DIV.fmt	Floating-point Divide
CFC1	Move Control Word From FPU	ABS.fmt	Floating-point Absolute value
Conversion Instructions		MOV.fmt	Floating-point Move
CVT.S.fmt	Floating-point Convert to Single FP	NEG.fmt	Floating-point Negate
CVT.D.fmt	Floating-point Convert to Double FP	Compare Instructions	
		C.cond.fmt	Floating-point Compare

1.7 Pipeline Architecture

The V_R3600 FPU provides an instruction pipeline that parallels that of the V_R3600 Processor. The FPU, however, has a 6-stage pipeline instead of the 5-stage pipeline of the V_R3600 the additional FPU pipe stage is used to provide efficient coordination of exception responses between the FPU and main processor.

The execution of a single V_R3600 FPU instruction consists of six primary steps:

1) IF—Instruction Fetch. The main processor calculates the instruction address required to read an instruction from the I-Cache. No action is required of the FPU during this pipe stage since the main processor is responsible for address generation.

2) RD—The instruction is present on the data bus during phase 1 of this pipe stage, and the FPU decodes the data on the bus to determine if it is an instruction for the FPU.

3) ALU—If the instruction is an FPU instruction, instruction execution commences during this pipe stage.

4) MEM—If this is a coprocessor load or store instruction, the FPU presents or captures the data during phase 2 of this pipe stage.

5) WB—The FPU uses this pipe stage solely to deal with exceptions.

6) FWB—The FPU uses this stage to write back ALU results to its register file. This stage is the equivalent of the WB stage in the V_R3600 main processor.

Each of these steps requires approximately one FPU cycle as shown in Figure 1-8 (parts of some operations spill over into another cycle while other operations require only 1/2 cycle).

Fig. 1-8 Instruction Execution Sequence

IF	RD	ALU	MEM	WB	FWB
I-Cache	RF	OP	D-Cache	exception	FpWB

one cycle

The V_R3600 uses a 6-stage pipeline to achieve an instruction execution rate approaching one instruction per FPU cycle. Thus, execution of six instructions at a time are overlapped as shown in Figure 1-9.

Fig. 1-9 V_R3600 Instruction Pipeline

					Current Cycle					
IF	RD	ALU	MEM	WB	FWB					
	IF	RD	ALU	MEM	WB	FWB				
		IF	RD	ALU	MEM	WB	FWB			
			IF	RD	ALU	MEM	WB	FWB		
				IF	RD	ALU	MEM	WB	FWB	
					IF	RD	ALU	MEM	WB	FWB

Instruction Flow ↓

This pipeline operates efficiently because different FPU resources (address and data bus accesses, ALU operations, register accesses, and so on) are utilized on a non-interfering basis. Refer to CHAPTER 2 INSTRUCTION SET SUMMARY & INSTRUCTION PIPELINE for a detailed discussion of the instruction pipeline.

Chapter 2 Instruction Set Summary & Instruction Pipeline

This chapter provides a summary of the V_R3600 FPU 's instruction set and also includes a detailed discussion of the FPU 's instruction pipeline that permits overlapping of instructions to increase the effective instruction execution rate.

2.1 Instruction Set Summary

The floating point instructions supported by the V_R3600 FPU are all implemented using the coprocessor unit 1 (COP1) operation instructions of the V_R3600 Processor instruction set. The basic operations performed by the FPU are:

- Load and store operations from/to the FPU registers
- Moves between FPU and CPU registers
- Computational operations including floating-point add, subtract, multiply divide and convert instructions
- Floating point comparisons

Remarks: The branch on coprocessor 1 condition (BC1T/BC1F) operations are also COP1 operations and are described in this chapter: however, these instructions are actually implemented entirely by the V_R3600 Processor using the CpCond input from the FPU.

(1) Load, Store, and Move Instructions

All movement of data between the V_R3600 FPU and memory is accomplished by load word to coprocessor 1 (LWC1) and store word to coprocessor 1 (SWC1) instructions which reference a single 32-bit word of the FPU's general registers. These loads and stores are unformatted; no format conversions are performed and therefore no floating-point exceptions occur due to these operations.

Data may also be directly moved between the FPU and the V_R3600

Processor by move to coprocessor 1 (MTC1) and move from coprocessor 1 (MFC1) instructions. Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

The load and move to operations have a latency of one instruction. That is, the data being loaded from memory or the CPU into an FPU register is not available to the instruction that immediately follows the load instruction: the data is available to the second instruction after the load instruction. (Refer to 2.2, Instruction Pipeline for a detailed discussion of load instruction latency.)

Table 2-1 and Table 2-2 summarize the Load/Store and Move instructions, respectively.

Table 2-1 Load and Store Instruction Summary

Instruction	Format and Description [op | base | ft | offset]
Load Word to FPU (coprocessor 1)	LWC1 ft, offset (base) Sign-extend 16-bit offset and add to contents of CPU register base to form address. Load contents of addressed word into FPU general register ft.
Store Word from FPU (coprocessor 1)	SWC1 ft, offset (base) Sign-extend 16-bit offset and add to contents of CPU register base to form address. Store 32-bit contents of FPU general register ft at addressed location.

Table 2-2 Move Instruction Summary

Instruction	Format and Description [COP1 | 0 | sub | rt | fs | 0]
Move Word to FPU (coprocessor 1)	MTC1 rt, fs Move contents of CPU register rt into FPU register fs.
Move Word from FPU (coprocessor 1)	MFC1 rt, fs Move contents of FPU general fs into CPU register ft.
Move Control Word to FPU	CTC1 rt, fs Move contents of CPU register rt into FPU control register fs.
Move Control Word from FPU (coprocessor 1)	CFC1 rt, fs Move contents of FPU control register fs into CPU register rt.

(2) Floating Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values in registers. There are four categories of computational instructions summarized in Table 2-3.

(a) 3-Operand Register-Type instructions that perform floating-point addition, subtraction, multiplication, and division operations.

(b) 2-Operand Register-Type instructions that perform floating point absolute value, move, and negate operations

(c) Convert instructions that perform conversions between the various data formats

(d) Compare instructions that perform comparisons of the contents of two registers and set or clear a condition signal based on the result of the comparison.

In the instruction formats shown in Table 2-3, the fmt term appended to the instruction op code is the data format specifier: s specifies Single-precision binary floating-point, d specifies Double-precision binary floating-point, and w specifies binary fixed-point. For example, an ADD.d specifies that the operands for the addition operation are double-precision binary floating-point values.

Caution: **When fmt is single-precision or binary fixed point, the odd register of the destination is undefined.**

Table 2-3 Computational Instruction Summary (1/2)

(a) ADD, SUB, MUL, DIV

Instruction	Format and Description COP1 | 1 | fmt | rt | fs | fd | funct
Floating-point Add	ADD.fmt fd, fs, ft Interpret contents of FPU registers ft and ft in specified format (fmt) and add arithmetically. Place rounded result in FPU register fd.
Floating-point Subtract	SUB.fmt fd, fs, ft Interpret contents of FPU registers fs and ft in specified format (fmt) and arithmetically subtract ft from fs. Place result in FPU register fd.
Floating-point Multiply	MUL.fmt fd, fs, ft Interpret contents of FPU registers fs and ft in specified format (fmt) and arithmetically multiply ft and fs. Place result in FPU register fd.
Floating-point Divide	DIV.fmt fd, fs, ft Interpret contents of FPU registers fs and ft in specified format (fmt) and arithmetically divide fs by ft. Place rounded result in register fd.

Table 2-3 Computational Instruction Summary (2/2)

(b) ABS, MOV, NEG, CVT.S, CVT.D, CVT.T, CVT.W

Instruction	Format and Description (COP1 | 1 | fmt | 0 | fs | fd | funct)
Floating-point Absolute Value	ABS.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and take arithmetic absolute value. Place result in FPU register fd.
Floating-point Move	MOV.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and copy into FPU register fd.
Floating-point Negate	NEG.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and take arithmetic negation. Place result in FPU fd.
Floating-point Convert to Single FP Format	CVT.S.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and arithmetically convert to the single binary floating point format. Place rounded result in FPU register fd.
Floating-point Convert to Double FP Format	CVT.D.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and arithmetically convert to the double binary floating point format. Place rounded result in FPU register fd.
Floating-point Convert to Single Fixed-Point Format	CVT.W.fmt fd, fs Interpret contents of FPU register fs in specified format (fmt) and arithmetically convert to the single fixed-point format. Place result in FPU register fd.

(c) C.cond

Instruction	Format and Description COP1 | 1 | fmt | ft | fs | 0 | fc | cond
Floating-point Compare	C.cond. fmt fs, ft Interpret contents of FPU registers fs and ft in specified format (fmt) and arithmetically compare. The result is determined by the comparison and the specified condition (cond). After a one instruction delay, the condition is available for testing by the V_R3600 with the branch on floating-point coprocessor condition (BC1T, BC1F) instructions.

(3) Floating Point Relational Operations

The Floating-point Compare (C.fmt.cond) instructions interpret the contents of two FPU registers (fs, ft) in the specified format (fmt) and arithmetically compares them. A result is determined based on the comparison and conditions (cond) specified in the instruction. Table 2-4 lists the conditions that can be specified for the Compare instruction and Table 2-5 summarizes the floating-point relational operations that are performed.

Table 2-4 Relational Mnemonic Definitions

Mne-monic	Definition	Mne-monic	Definition
F	False (FpCond=0)	T	True (FpCond=1)
UN	Unordered	OR	Ordered
EQ	Equal	NEQ	Not Equal
UEQ	Unordered or Equal	OLG	Ordered or Less Than or Greater Than
OLT	Ordered Less Than	UGE	Unordered or Greater Than or Equal
ULT	Unordered or Less Than	OGE	Ordered Greater Than
OLE	Ordered Less than or Equal	UGT	Unordered or Greater Than
ULE	Unordered or Less than or Equal	OGT	Ordered Greater Than
SF	Signaling False	ST	Signaling True
NGLE	Not Greater than or Less than or Equal	GLE	Greater Than, or Less Than or Equal
SEQ	Signaling Equal	SNE	Signaling Not Equal
NGL	Not Greater than or Less than	GL	Greater Than or Less Than
LT	Less Than	NLT	Not Less Than
NGE	Not Greater than or Equal	GE	Greater Than or Equal
LE	Less than or Equal	NLE	Not Less Than or Equal
NGT	Not Greater Than	GT	Greater Than

Table 2-5 is derived from the similar table in the IEEE floating point standard and describes the 26 predicates named in the standard. The table includes six additional predicates (for a total of 32) to round out the set of possible predicates based on the conditions tested by a comparison. Four mutually exclusive relations are possible: less than, equal, greater than, and unordered. Note that invalid operation exceptions occur only when comparisons include the less than (<) or greater than (>) characters but not the unordered (?) character in the ad hoc form of the predicate.

Table 2-5 Floating Point Relational Operators

PREDICATES			RELATIONS				Invalid Operation Exception if Unordered
Condition Mnemonic	ad hoc	FORTRAN	Greater Than	Less Than	Equal	Un-ordered	
F	false		F	F	F	F	no
UN	?		F	F	F	T	no
EQ	=	.EQ.	F	F	T	F	no
UEQ	? =	.UE.	F	F	T	T	no
OLT	NOT(?>=)	.NOT..UGE.	F	T	F	F	no
ULT	?<	.UL.	F	T	F	T	no
OLE	NOT(?>)	.NOT..UG.	F	T	T	F	no
ULE	?<=	.ULE.	F	T	T	T	no
OGT	NOT(?<=)	.NOT..ULE.	T	F	F	F	no
UGT	?>	.UGT.	T	F	F	T	no
OGE	NOT(?<)	.NOT..UL.	T	F	T	F	no
UGE	?>=	.UGE.	T	F	T	T	no
OLG	NOT(? =)		T	T	F	F	no
NEQ	NOT(=)	.NE.	T	T	F	T	no
OR	NOT(?)		T	T	T	F	no
T	true		T	T	T	T	no
SF			F	F	F	F	yes
NGLE	NOT(<=>)	.NOT..LGE.	F	F	F	T	yes
SEQ			F	F	T	F	yes
NGL	NOT(<>)	.NOT..LG.	F	F	T	T	yes
LT	<	.LT.	F	T	F	F	yes
NGE	NOT(>=)	.NOT..GE.	F	T	F	T	yes
LE	<=	.LE.	F	T	T	F	yes
NGT	NOT(>)	.NOT..GT.	F	T	T	T	yes
GT	>	.GT.	T	F	F	F	yes
NLE	NOT(<=)	.NOT..LE.	T	F	F	T	yes
GE	>=	.GE.	T	F	T	F	yes
NLT	NOT(<)	.NOT..LT.	T	F	T	T	yes
GL	< >	.LG.	T	T	F	F	yes
SNE			T	T	F	T	yes
GLE	<=>	.LEG.	T	T	T	F	yes
ST			T	T	T	T	yes

(4) Branch Instructions

Table 2-6 summarizes the two branch instructions that can be used to test the result of the FPU Compare (C.cond) instructions.
In this table, the phrase delay slot refers to the instruction immediately following the branch instruction. Refer to the PART 2 CPU ARCHITECTURE for a discussion of the branch delay slot.

Table 2-6 Branch Instructions

Instruction	Format and Description COP1 | [illegible] | sub | br | offset
Branch on FPU True	BC1T Compute a branch target address by adding address of instruction in the delay slot and the 16-bit offset (shifted left two bits and sing-extended to 32 bits). Branch to the target address (with a delay of one instruction) if the FPU's CpCond signal is true.
Branch on FPU False	BC1F Compute a branch target address by adding address of instruction in the delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if the FPU's CpCond signal is false.

2.2 The Instruction Pipeline

2.2.1 The pipeline processing

The V_R3600 FPU provides an instruction pipeline that parallels that of the V_R3600 Processor. The FPU, however, has a 6-stage pipeline instead of the 5-stage pipeline of the V_R3600; the additional FPU pipe stage is used to provide efficient coordination of exception responses between the FPU and main processor. Figure 2-1 illustrates the six stages of the FPU instruction pipeline.

Fig. 2-1 Instruction Execution Sequence

IF	RD		ALU	MEM	WB	FWB
I-Cache		RF	OP	D-Cache	Exception	FpWB

one cycle

The six stages of the FPU instruction pipeline are used as follows:

1) IF - Instruction Fetch. The CPU calculates the instruction address required to read an instruction from the I-Cache. The instruction address is generated and output during phase 2 of this pipe stage. No action is required of the FPU during this pipe stage since the main processor is responsible for address generation. Note that the instruction is not actually read into the processor until the beginning (phase 1) of the RD pipe stage.

2) RD - The instruction is present on data bus during phase 1 of this pipe stage and the FPU decodes the data on the bus to determine if it is an instruction for the FPU. The FPU reads any required operands from its registers (RF = Register Fetch) while decoding the instruction.

3) ALU - If the instruction is one for the FPU, execution commences during this pipe stage. If the instruction causes an exception, the FPU notifies the V_R3600 main processor of the exception during this pipe stage by asserting the $\overline{\text{FpInt}}$ signal. If the FPU determines that it requires additional time to complete this instruction, it initiates a stall during this pipe stage.

4) MEM - If this is a coprocessor load or store instruction, the FPU presents or captures the data during phase 2 of this pipe stage. If an interrupt is taken by the main processor, it notifies the FPU during phase 2 of this pipe stage (via the $\overline{\text{Exception}}$ signal).

5) WB - If the instruction that is currently in the write back (WB) stage caused an exception, the main processor notifies the FPU by asserting the $\overline{\text{Exception}}$ signal during this pipe stage. Thus, the FPU uses this pipe stage solely to deal with exceptions.

6) FWB - The FPU uses this stage to write back ALU results to its register file. This stage is the equivalent of the WB stage in the V_R3600 main processor.

Figure 2-2 illustrates how the six instructions would be overlapped in the FPU pipeline.

Fig. 2-2 V_R3600 FPU Instruction Pipeline

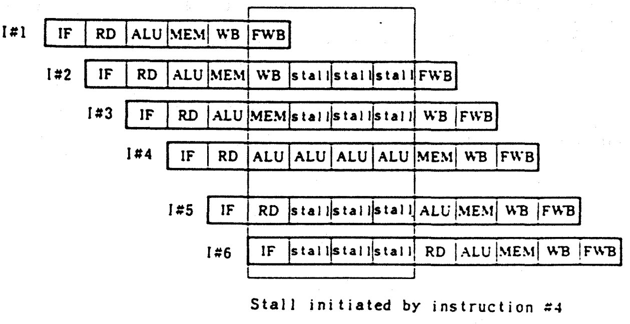
IF	RD	ALU	MEM	WB	FWB					
	IF	RD	ALU	MEM	WB	FWB				
		IF	RD	ALU	MEM	WB	FWB			
			IF	RD	ALU	MEM	WB	FWB		
				IF	RD	ALU	MEM	WB	FWB	
					IF	RD	ALU	MEM	WB	FWB

Instruction Flow ↓

Current CPU Cycle

This figure presumes that each instruction can be completed in a single cycle. Most FPU instructions, however, require more than one cycle to execute. Therefore, the FPU must stall the pipeline if an instruction's execution cannot proceed because of register or resource conflicts. Figure 2-3 illustrates the effect of a three-cycle stall on the FPU pipeline.

Fig. 2-3 Pipeline Stall



To mitigates the performance impact that would result from frequently stalling the instruction pipeline, the FPU allows overlapping of instructions so that instruction execution can proceed so long as there are no resource conflicts, data dependencies, or exception conditions. The sections that follow describe and illustrate the timing and overlapping of FPU instruction.

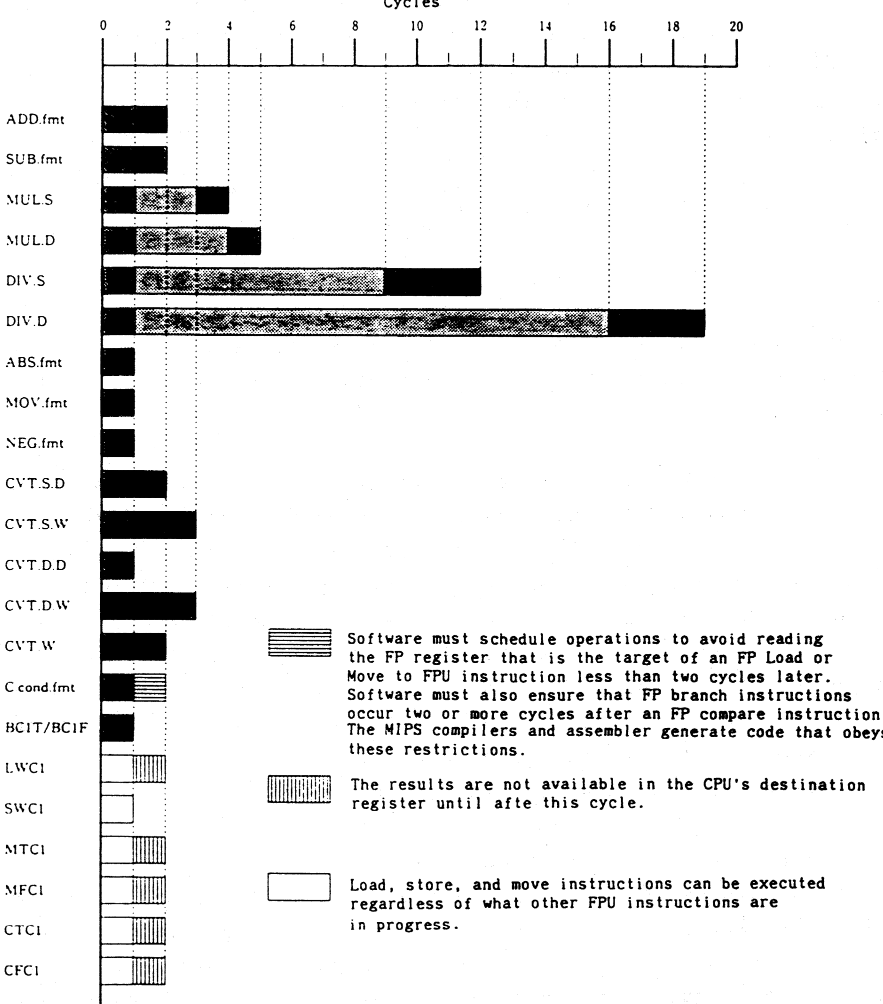
2.2.2 Instruction Execution Times

Unlike the V_R3600 Processor which executes almost all instructions in a single cycle, the time required to execute FPU instructions ranges from one cycle to 19 cycles. Figure 2-4 illustrates the number of cycles required to execute each of the FPU instructions.

In Figure 2-4, the cycles of an instruction's execution time that are darkly shaded ▓ require exclusive access to an FPU resource (such as buses or ALU) that precludes the concurrent use by another instruction and therefore prohibits overlapping execution of another FPU instruction. (Note that load and store operations can be overlapped with these cycles.) Those instruction cycles that are lightly shaded ░, however, are placing minimal demands on the FPU resources, and other instructions can be overlapped (with some restrictions) to obtain simultaneous execution of instructions without stalling the instruction pipeline.

For example, an instruction such as DIV.D that requires a large number of cycles to complete could begin execution, and another instruction such as ADD.D could be initiated and completed while the DIV.D instruction is still being executed. Note that only one multiply instruction can be running at a time and only one divide instruction can run at a time.

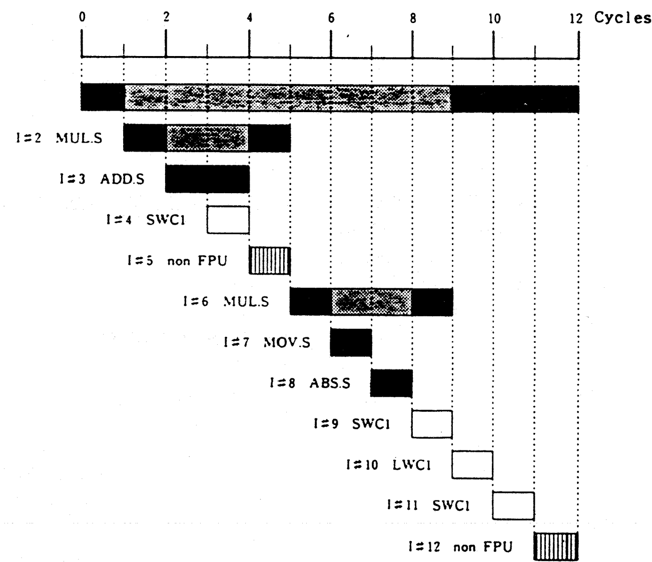
Fig. 2-4 Instruction Execution Times



2.2.3 Overlapping FPU instructions

Figure 2-5 illustrates the overlapping of several FPU (and non-FPU) instructions. In this figure, the first instruction (DIV.S) requires, a total of 12 cycles for execution but only the first cycle and last three cycles preclude the simultaneous execution of other FPU instructions. Similarly, the second instruction (MUL.S) has 2 cycles in the middle of its total of 4 required cycles that can be used to advance the execution of the third (ADD.S) and fourth instructions shown in the figure.

Fig. 2-5 Overlapping FPU Instructions



non FPU: Instructions excepting V_R3600 FPU

Note that although processing of a single instruction consists of six pipe stages, the FPU does not require that an instruction actually be completed within six cycles to avoid stalling the instruction pipelines. If a subsequent instruction does not

require FPU resources being used by a preceding instruction and has no data dependencies on preceding uncompleted instructions, then execution continues.

Fig. 2-6 Overlapped Instructions in the FPU Pipeline

Cycles: 0, 2, 4, 6, 8, 10, 12

I=1	IF	RD	ALU	MEM	WB	FWB								
I=2		IF	RD	ALU	MEM	WB	FWB							
I=3			IF	RD	ALU	MEM	WB	FWB						
I=4				IF	RD	ALU	MEM	WB	FWB					
I=5					IF	RD	ALU	MEM	WB	FWB				
I=6						IF	RD	ALU	MEM	WB	FWB			
I=7							IF	RD	ALU	MEM	WB	FWB		
I=8								IF	RD	ALU	MEM	WB	FWB	
I=9									IF	RD	ALU	MEM	WB	FWB

I=1 DIV.S
I=2 MUL.S
I=3 ADD.S
I=4 SWC1
I=5 non FPU
I=6 MUL.S
I=7 MOV.S
I=8 ABS.S
I=9 SWC1
I=10 LWC1
I=11 non FPU
I=12 non FPU

non FPU: Instructions excepting V_R3600 FPU

Figure 2-6 illustrates the progression of the FPU instruction pipeline with some overlapped FPU instructions. The first instruction (DIV.S) in this figure requires eight additional cycles beyond its FWB pipe stage before it is completed. The pipeline need not be stalled, however, because the way in which

the FPU instructions are overlapped avoids resource conflicts.

Figure 2-6 also presumes that there are no data dependencies between the instructions that would stall the pipeline. For example, if any instruction before I#13 required the results of the DIV.S (I#1) instruction, then the pipeline would be stalled until those results were available.

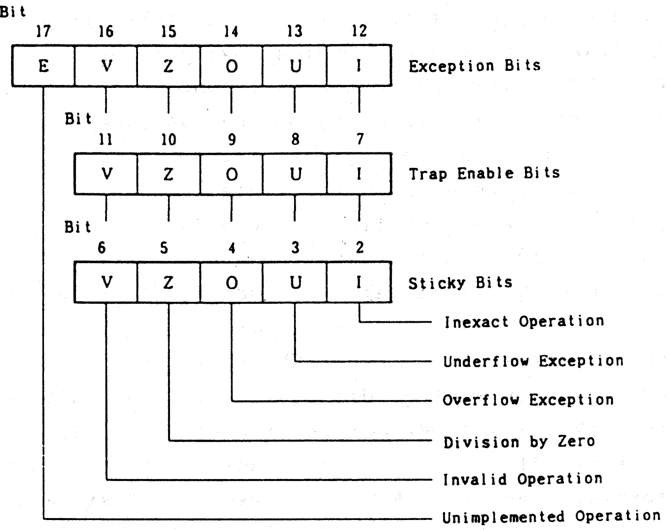
Chapter 3 Floating Point Exceptions

The chapter describes how the V_R3600 FPU handles floating point exceptions. A floating point exception occurs whenever the FPU cannot handle the operands or results of a floating point operation in the normal way. The FPU responds either by generating an interrupt to initiate a software trap or by setting a status flag. The Control/Status register described in CHAPTER 1 V_R3600 FPU OVERVIEW contains a trap enable bit for each exception type that determines whether an exception will cause the FPU to initiate a trap or just set a status flag. If a trap is taken, the FPU remains in the state found at the beginning of the operation, and a software exception handling routine is executed. If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE exceptions -- inexact (I), overflow (O), underflow (U), divide by zero (Z), and invalid operation (V) -- with exception bits, trap enables, and sticky bits (status flags). The FPU adds a sixth exception type, unimplemented operation (E), to be used in those cases where the FPU itself cannot implement the standard floating-point operation, including cases where the FPU cannot determine the correct exception behavior. This exception indicates that a software implementation must be used. The unimplemented operation exception has no trap enable or sticky bit; whenever this exception occurs, an unimplemented exception trap is taken (if the FPU's interrupt input to the V_R3600 is enabled).

Figure 3-1 illustrates the Control/Status register bits used to support exceptions.

Fig. 3-1 Control/Status Register Exception/ Sticky/Trap Enable Bits



Each of the five IEEE exceptions (V, Z, O, U, I) is associated with a trap under user control which is enabled by setting one of the five TrapEnable bits. When an exception occurs, both the corresponding Exception and Sticky bits are set. If the corresponding TrapEnable bit is set, the FPU generates an interrupt to the V_R3600 processor and the subsequent exception processing allows a trap to be taken.

3.1 Exception Trap Processing

When a floating-point exception trap is taken, the V_R3600 Processor's Cause register indicates that an external interrupt from the FPU is the cause of the exception and the V_R3600's EPC (Exception Program Counter) contains the address of the instruction that caused the exception trap.

For each IEEE standard exception, a status flag (Sticky bit) is provided that is set on any occurrence of the corresponding exception condition with no corresponding exception trap signaled. The Sticky bits may be reset by writing a new value into the Control/Status register and may be saved and restored individually, or as a group, by software.

When no exception trap is signaled, a default action is taken by the FPU, which provides a substitute value for the original, exceptional, result of the floating-point operation. The default action taken depends on the type of exception, and in the case of the Overflow exception, the current rounding mode. Table 3-1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 3-1 Exception Default Actions

Exception		Rounding Mode	Default Action (no exception trap signaled)
V	Invalid Operation	—	Supply a quiet NaN.
Z	Division by zero	—	Supply a properly signed ∞.
O	Overflow	RN	Modify overflow values to ∞ with the sign of the intermediate result.
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result.
		RP	Modify negative overflows to the format's most negative finite number. Modify positive overflows to + ∞.
		RM	Modify positive overflows to the format's largest finite number. Modify negative overflows to - ∞.
U	Underflow	—	Generate an Unimplemented exception.
I	Inexact	—	Supply a rounded result.

The FPU internally detects eight different conditions that can cause exceptions. When the FPU encounters one of these unusual situations, it will cause either an IEEE exception or an Unimplemented Operation exception (E). Table 3-2 lists the exception-causing situations and contrasts the behavior of the FPU with the IEEE standard's requirements.

Table 3-2 Exception-causing Conditions

FPU internal result	IEEE Stndrd	Trap Enab.	Trap Disab.	Note
Inexact result	I	I	I	loss of accuracy
Exponent overflow	O I	O I	O I	normalized exponent > Emax
Divide by zero	Z	Z	Z	zero is (exponent = Emin-1, mantissa =0)
Overflow on convert	V	V	E	source out of integer range
Signaling NaN source	V	V	E	quiet NaN source produces quiet NaN result
Invalid operation	V	V	E	0/0 etc.
Exponent underflow	U	E	E	normalized exponent < Emin
Denormalized source	none	E	E	exponent = Emin-1 and mantissa<>0

Remarks: Standard specifies inexact exception on overflow only if overflow trap is disabled.

The sections that follow describe the conditions that cause the FPU to generate each of its six exceptions and details the FPU 's response to each of these exception-causing situations.

(1) Inexact Exception (I)

The FPU generates this exception if the rounded result of an operation is not exact or if it overflows.

The FPU usually examines the operands of floating point operations before execution actually begins to determine (based on the exponent values of the operands) if the operation can possibly cause an exception. If there is a possibility of an instruction causing an exception trap, then the FPU uses a coprocessor stall mechanism to execute the instruction. It is impossible, however, for the FPU to predetermine if an instruction will produce an inexact result. Therefore, if inexact exception traps are enabled, the FPU uses the coprocessor stall mechanism to execute all floating point operations that require more than one cycle. Since this mode of execution can impact performance, inexact exception traps should be enabled only when necessary.

Trap Enabled Results: If inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

Trap Disabled Results: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

(2) Invalid Operation Exception (V)

The invalid operation exception is signaled if one or both of the operands are invalid for an implemented operation. The invalid operations are:

1) Addition or subtraction: magnitude subtraction of infinities, such as:
   $(+\infty) - (+\infty)$

2) Multiplication: 0 times ∞, with any signs

3) Division: 0 ÷ 0, or ∞ ÷ ∞, with any signs

4) Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format

5) Comparison of predicates involving <or> without?, when the operands are "unordered"

6) Any arithmetic operation on a signaling NaN. Note that a move (MOV) operation is not considered to be an arithmetic operation, but that ABS and NEG are considered to be arithmetic operations and will cause this exception if one or both operands is a signaling NaN.

Software may simulate this exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE-specified functions implemented in software, such as Remainder: x REM y, where y is zero or x is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow or is infinity or NaN; and transcendental functions, such as ln (-5) or $\cos^{-1}(3)$. Refer to CHAPTER 4 INSTRUCTION SET DETAILS for examples or routines to handle these cases.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: The FPU always signals an Unimplemented exception because it does not create the NaN that the Standard specifies should be returned under these circumstances.

(3) Division-by-Zero Exception (Z)

The division by zero exception is signaled on a divide operation if the divisor is zero and the dividend is a finite non-zero number.

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

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

(4) Overflow Exception (O)

The overflow exception is signaled when what would have been the magnitude of the rounded floating-point result, were the exponent range unbounded, is larger than the destination format's largest finite number. (This exception also sets the Inexact exception and sticky bits.)

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

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (as listed in Table 3-1).

(5) Underflow Exception (U)

The FPU never generates an Underflow exception and never sets the U bit in either the Exceptions field or Sticky field of the Control/Status register. If the FPU detects a condition that could be either an underflow or a loss of accuracy, it generates an Unimplemented exception.

(6) Unimplemented Operation Exception (E)

The FPU generates this exception when it attempts to execute an instruction with an OpCode (bits 31-26) or format code (bits 24-21) which has been reserved for future use.

This exception is not maskable: the trap is always enabled. When an Unimplemented Operation is signaled, an interrupt is sent to the V_R3600 Processor so that the operation can be emulated in software. When the operation is emulated in software, any of the IEEE exceptions may arise; these exceptions must, in turn, be simulated.

This exception is also generated when any of the following exceptions are detected by the FPU:

- o Denormalized Operand
- o Not-a-Number (NaN) Operand
- o Invalid operation with trap disabled
- o Denormalized Result
- o Underflow

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot be disabled.

3.2 Saving and Restoring State

Thirty-two coprocessor load or store instructions will save or restore the FPU's floating-point register state in memory. The contents of the Control/Status register can be saved using the "move to/from coprocessor control register" instructions (CTC1/CFC1). Normally, the Control/Status register contents are saved first and restored last.

If the Control/Status register is read when the coprocessor is executing one or more floating-point instructions, the instructions in progress (in the pipeline) are completed before the contents of the register are moved to the main processor. If an exception occurs during one of the in-progress instructions, that exception is written into the Control/Status register Exceptions field.

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

All of the bits in the Exceptions field can be cleared by writing a zero value to this field. This permits restarting of normal processing after the Control/Status register state is restored.

Chapter 4 Instruction Set Details

This chapter provides a detailed description of the operation of each V_R3600 instruction. The instructions are listed alphabetically.

4.1 Instruction Set Summary

4.1.1 Instruction formats

There are four basic instruction format types:

(1) I-Type, or Immediate instructions, which include Load and Store operations,

(2) M-Type, or Move instructions,

(3) R-Type, or Register instructions, which include the two- and three-register Floating-Point operations, and

(4) B-Type, or Branch instructions, which include Branch operations according to the CpCond(1) signal.

The instruction description subsections that follow show how the four basic instruction formats are used by:

(1) Load and Store instructions,

(2) Move instructions,

(3) Floating-Point Computational instructions, and

(4) Branch instructions.

4.1.2 Instruction notational conventions

In this appendix, all variable subfields in an instruction format (such as fs, ft, immediate, and so on) are shown with lower-case names. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

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

In some instructions, however, the two instruction subfields op and function have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. In the floating-point instruction, for example, we use op = COP1 and function = ADD. In some cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Actual bit encoding for mnemonics is shown in APPENDIX B INSTRUCTION CODES.

In the instruction description 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 4-1.

Table 4-1 Instruction Operation Notations

Symbol	Meaning
←	Assignment
‖	Bit string concatenation
x^y	Replication of bit value x into a y-bit string. Note that x is always a single-bit value.
$x_{y..z}$	Selection of bits y through z of bit string x. Little-endian bit notation is always used. If y is less than z, this expression is an empty (zero length) bit string.
+	Two's complement addition
-	Two's complement subtraction
*	Two's complement multiplication
div	Two's complement integer division
mod	Two's complement modulo
/	Floating-point division
<	Two's complement less than comparison
and	Bitwise logic AND
or	Bitwise logic OR
xor	Bitwise logic XOR
nor	Bitwise logic NOR
GPR[x]	CPU General Register x. Note that the contents of GPR[0] are always zero: attempts to alter GPR[0] contents have no effect.
FGR[x]	FPU General Register x, as viewed by the V_R3600 processor.
FPR[x]	FPU Floating-Point register x. Each FPR is assembled from two FGRs.
FCR[x]	FPU Control Register x.
COC[1]	FPU Control Signal (CpConbd 1).
T + i	Indicates the time steps (CPU cycles) between operations. Thus, operations identified as occurring at T + i are performed during the cycle following the one where the instruction was initiated. This type of operation occurs with loads, stores, jumps, branches and coprocessor instructions.
vAddress	Virtual address
pAddress	Physical address

o Instruction Notation Examples

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

Example #1:

$GPR[ft] \leftarrow immediate \; || \; 0^{16}$

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to GPR ft.

Example #2:

$(immediate_{15})^{16} \; || \; immediate_{15..0}$

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

4.1.3 Load and store instructions

All loads operations have a latency of one instruction. That is, the instruction immediately following a load cannot use the contents of the register which will be loaded with the data being fetched from storage.

In the load/store operation descriptions, the functions listed in Table 4-2 are used to summarize the handling of virtual addresses and physical memory.

Table 4-2 Load/Store Common Functions

Function	Description
Address Translation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the entry for the page containing the virtual address is not present in the TLB (Translation Lookaside Buffer).
Load Memory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of each the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
Store Memory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data into the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of the four bytes within the data word should be stored.

o Load and Store Instruction Format

Figure 4-1 shows the I-Type instruction format used by load and store operations.

Fig. 4-1 Load and Store Instruction Format

I-Type (Immediate)

31 — 26	25 — 21	20 — 16	15 — 0
op	base	ft	offset
6	5	5	16

Remarks: The variable subfields are shown as follows:

op	is a 6-bit operation code
base	is the 5-bit V_R3600 CPU base register specifier
ft	is a 5-bit source (for stores) or destination (for loads) FPU register
offset	is the 16-bit signed immediate offset

All coprocessor loads and stores reference aligned full word data items. Thus, the access type field is always WORD, and the low-order two bits of the address must always be zero. The address specifies the smallest byte address of each byte in the address field.

4.1.4 Computational instructions

Computational instructions include all of the arithmetic floating-point operations performed by the FPU.

Figure 4-2 shows the R-Type instruction format used for computational operations.

Fig. 4-2 Computational Instruction Format

R-Type (Register)

31 – 26	25	24 – 21	20 – 16	15 – 11	10 – 6	5 – 0
op	1	fmt	ft	fs	fd	funct
6	1	4	5	5	5	6

Remarks: The variable subfields are shown as follows:

op	is a 6-bit major operation code
fmt	is a 4-bit format specifier
fs	is a 5-bit source 1 register
ft	is a 5-bit source 2 register
fd	is a 5-bit destination register
funct	is a 6-bit function field

The four format code bits of a floating-point instruction specify which operand format is used in the instruction. Decoding for this field is shown in Table 4-3.

Table 4-3 Format Field Decoding

Code	Mnemonic	Size	Format
0	S	single	binary floating-point
1	D	double	binary floating-point
2-3	-	-	reserved
4	W	single	binary fixed-point
5-15	-	-	reserved

The six low-order function bits of coprocessor instruction indicate which floating-point operation is to be performed. Table 4-4 lists all floating-point instructions.

Table 4-4 Floating-Point Instructions and Operations

Code	Mnemonic	Operation
0	ADD.fmt	Add
1	SUB.fmt	Subtract
2	MUL.fmt	Multiply
3	DIV.fmt	Divide
4	-	reserved
5	ABS.fmt	Absolute value
6	MOV.fmt	Move
7	NEG.fmt	Negate
8-31	-	reserved
32	CVT.S.fmt	Convert to single floating-point
33	CVT.D.fmt	Convert to double floating-point
34-35	-	reserved
36	CVT.W.fmt	Convert to binary fixed-point
37-47	-	reserved
48-63	C.fmt	Floating-point compare

The following routines are used in the description of the floating-point operations to get the value of an FPR or to change the value of an FGR: When the format is single-precision or integer, the odd register of the destinations is undefined.

```
value ← ValueFPR(fp, fmt) :
    case fmt of
        S :  value ← FGR[fpr]
        D :  value ← FGR[fpr + 1] || FGR[fpr]
        W :  value ← FGR[fpr]
    end
StoreFPR(fpr, fmt, value) :
    case fmt of
        S : FGR[fpr + 1] ← undefined
            FGR[fpr] ← value
        D : FGR[fpr + 1] ← value63...32
            FGR[fpr] ← value31..0
        W : FGR[fpr] ← value
            FGR[fpr + 1] ← undefined
    end
```

4.1.5 Move instructions

Move instructions include move operations between CPU and FPU. The instruction format of MOV.fmt, which is the move operation between the FPU registers, is R-Type.

Figure 4-3 shows the M-Type instruction format used for move operations.

Fig. 4-3 Move Instruction Format

M-Type (move)

31 26	25	24 21	20 16	15 11	10 0
op	0	sub	rt	fs	0
6	1	4	5	5	11

Remarks: The variable subfields are shown as follows:

op	is a 6-bit major operation code
sub	is a 4-bit sub-op code
rt	is a 5-bit CPU register
fs	is a 5-bit FPU register

4.1.6 Branch instructions

Branch instructions include branch operations performed according to the CpCond(1) signal status.

Figure 4-4 shows the B-Type instruction format used for branch operations.

Fig. 4-4 Branch Instruction Format

B-Type (branch)

31–26	25	24–21	20–16	15–0
op	0	sub	br	offset
6	1	4	5	16

Remarks: The variable subfields are shown as follows:

op	is a 6-bit major operation code
sub	is a 4-bit sub-op code
br	is a 5-bit branch operation code
offset	is the 16-bit branch address offset

4.2 Instruction Details

This section provides a detailed description of the operation of each V_R3600 instruction. The instructions are listed alphabetically. The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction.

ABS.fmt Floating-Point Absolute Value

31 – 26	25	24 – 21	20 – 16	15 – 11	10 – 6	5 – 0
COP1	1	fmt	0	fs	fd	ABS
6	1	4	5	5	5	6

Format:

ABS.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified format and the arithmetic absolute value is taken. The result is placed in the floating-point register specified by fd.

The absolute value is always exact.

On the FPU, this operation is valid only for single- and double-precision floating-point formats. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

```
T :      StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt))) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

ADD.fmt — Floating-Point Add

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	ft	fs	fd	ADD
6	1	4	5	5	5	6

Format:

ADD.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified format and arithmetically added. The result is rounded as if calculated to infinite precision and then rounded to the specified format (fmt), according to the current rounding mode. The result is placed in the floating-point register (FPR) specified by fd.

This instruction is valid on the FPU only for single- and double-precision floating-point format. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent FPU general registers (FGR).

Operation:

```
T:      StoreFPR(fd.fmt.ValueFPR(fs.fmt) + ValueFPR(ft.fmt)) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception
Inexact Exception
Overflow Exception
Underflow Exception

BC1F

Branch On FPU False
(coprocessor 1)

31–26	25	24–21	20–16	15–0
COP1	0	BC	BCF	offset
6	1	4	5	16

Format:

BC1F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32 bits. If the FPU's condition signal (FpCond) to the V_R3600 Processor is false, the program branches to the target address, with a delay of one instruction.

Operation:

T − 1 :	condition ← not COC[1]
T :	target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
T + 1 :	if condition then
	PC ← PC + target
	endif

Exceptions:

Coprocessor unusable exception

BC1T

Branch On FPU True
(coprocessor 1)

31–26	25	24–21	20–16	15–0
COP1	0	BC	BCT	offset
6	1	4	5	16

Format:

BC1T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32 bits. If the FPU's condition signal (FpCond) to the V_R3600 Processor is true, the program branches to the target address, with a delay of one instruction.

Operation:

T − 1 :	condition ← COC[1]
T :	target ← $(offset_{15})^{14}$ ‖ offset ‖ 0^2
T + 1 :	if condition then
	PC ← PC + target
	endif

Exceptions:

Coprocessor unusable exception

C.cond.fmt — Floating-Point Compare

31–26	25	24–21	20–16	15–11	10–6	5–4	3–0
COP1	1	fmt	ft	fs	0	fc	cond
6	1	4	5	5	5	2	4

Format:

C.cond.fmt fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified source format and arithmetically compared. A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is a NaN, and the low-order bit of the condition is set, an invalid operation trap is taken. After a one-instruction delay, the condition is available for testing with "branch on FPU coprocessor condition" instructions.

Comparisons are exact and neither overflow nor underflow. Four mutually exclusive relations are possible results: less than, equal, greater than, and unordered. The last case arises when one or both of the operands are NaN; every NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 = -0.

On the FPU, this operation is valid only for double- or single-precision floating-point formats. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

C.cond.fmt

Floating-Point Compare (Cont'd)

Operation:

```
T :      if NaN(ValueFPR(fs.fmt)) or NaN(ValueFPR(ft.fmt)) then
           less ← false
           equal ← false
           unordered ← true
           if cond3 then
               single InvalidOperationException
           endif
         else
           less ← ValueFPR(fs.fmt) < ValueFPR(ft.fmt)
           equal ← ValueFPR(fs.fmt) = ValueFPR(ft.fmt)
           unordered ← false
         endif
T+1 :    condition ← (cond2 and less) or
           (cond1 and equal) or
           (cond0 and unordered)
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

CFC1

Move Control Word From FPU (coprocessor 1)

31–26	25	24–21	20–16	15–11	10–0
COP1	0	CF	rt	fs	0
6	1	4	5	5	11

Format:

CFC1 rt, fs

Description:

The contents of the FPU's control register fs are loaded into V_R3600 Processor's general register rt.

Operation:

```
T:      temp ← FCR[fs] ;
T+1 :   GPR[rt] ← temp ;
```

Exceptions:

Coprocessor unusable exception

CTC1 — Move Control Word To FPU (coprocessor 1)

31–26	25	24–21	20–16	15–11	10–0
COP1	0	CT	rt	fs	0
6	1	4	5	5	11

Format:

CTC1 rt, fs

Description:

The contents of V_R3600 Processor's general register rt are loaded into the FPU's control (FCR) register fs.

Operation:

T:	temp ← GPR[rt];
T+1:	FCR[fs] ← temp;
	COC[1] ← $FCR[31]_{23}$

Exceptions:

Coprocessor unusable exception

CVT.D.fmt

Floating-Point Convert to Double Floating-Point Format

31 — 26	25	24 — 21	20 — 16	15 — 11	10 — 6	5 — 0
COP1	1	fmt	0	fs	fd	CVT.D
6	1	4	5	5	5	6

Format:

CVT.D.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified source format and arithmetically converted to the double binary floating-point format. The result is placed in the FPU register specified by fd.

Rounding occurs according to the currently specified rounding mode.

On the FPU, this operation is valid only for conversion from a single fixed-point or floating-point format. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

```
T:    StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt), fmt, D))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

CVT.S.fmt — Floating-Point Convert to Single Floating-Point Format

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	0	fs	fd	CVT.S
6	1	4	5	5	5	6

Format:

CVT.S.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified source format and arithmetically converted to the single binary floating-point format. The result is placed in the FPU register specified by fd.

Rounding occurs according to the currently specified rounding mode.

On the FPU, this operation is valid only for conversion from double-precision floating-point or fixed-point formats. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

```
T:      StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

CVT.S.fmt

Floating-Point Convert to Single Floating-Point Format (Cont'd)

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

CVT.W.fmt

Floating-Point Convert to Fixed-Point Format

31 ... 26	25	24 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
COP1	1	fmt	0	fs	fd	CVT.W
6	1	4	5	5	5	6

Format:

CVT.W.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified source format and arithmetically converted to the single fixed-point format. The result is placed in the FPU register specified by fd.

On the FPU, this operation is valid only for conversion from single- or double-precision floating-point formats. For double-precision format, this operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

T:	StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

DIV.fmt   Floating-Point Divide

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	ft	fs	fd	DIV
6	1	4	5	5	5	6

Format:

DIV.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified format and arithmetically divided. The result is rounded as if calculated to infinite precision and then rounded to the specified format (fmt), according to the current rounding mode. The result is placed in the floating-point register (FPR) specified by fd.

This instruction is valid on the FPU only for single- and double-precision floating-point format. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent FPU general registers (FGR).

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) / ValueFPR (ft, fmt)) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

DIV.fmt

Floating-Point Divide (Cont'd)

Floating-point Exceptions:

Unimplemented Operation Exception
Inexact Exception
Divide by Zero Exceptions
Invalid Operation Exception
Overflow Exception

LWC1 | Load Word to FPU (coprocessor 1)

31 — 26	25 — 21	20 — 16	15 — 0
LWC1	base	ft	offset
6	5	5	16

Format:

LWC1 ft, offset (base)

Description:

The 16-bit offset is signing-extended and added to the contents if the V_R3600 Processor's general register base to form a 32-bit unsigned effective address. The contents of the word at the effective address memory location is loaded into the FPU's general register (FGR) at location ft.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

T :	vAddr ← $((\text{offset}_{15})^{16} \parallel \text{offset}_{15..0})$ + GPR[base]
	(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
	mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
	FGR[ft] ← undefined
T + 1 :	FGR[ft] ← mem

Exceptions:

Coprocessor unusable exception
Bus error exception
Address error exception

MFC1

Move Word From FPU
(coprocessor 1)

31–26	25	24–21	20–16	15–11	10–0
COP1	0	MF	rt	fs	0
6	1	4	5	5	11

Format:

MFC1 rt, fs

Description:

The contents of the FPU general register at location fs are loaded into V_R3600 Processor's general register rt.

Operation:

T :	temp ← FGR[fs]
T+1 :	GPR[rt] ← temp

Exceptions:

Coprocessor unusable exception

MOV.fmt — Floating-Point Move

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	0	fs	fd	MOV
6	1	4	5	5	5	6

Format:

MOV.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified format and are copied into the FPU register specified by fd.

The Move operation is always exact.

On the FPU, this operation is valid only for single- and double-precision floating-point formats. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

```
T :    StoreFPR(fd.fmt.ValueFPR(fs.fmt)) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception

MTC1

Move Word To FPU (coprocessor 1)

31–26	25	24–21	20–16	15–11	10–0
COP1	0	MT	rt	fs	0
6	1	4	5	5	11

Format:

MTC1 rt, fs

Description:

The contents of V_R3600 Processor's general register rt are loaded into the FPU's general register at location fs.

Operation:

T :	temp ← GPR[rt]
T+1 :	FGR[fs] ← temp

Exceptions:

Coprocessor unusable exception

MUL.fmt Floating-Point Multiply

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	ft	fs	fd	MUL
6	1	4	5	5	5	6

Format:

MUL.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified format and arithmetically multiplied. The result is rounded as if calculated to infinite precision and then rounded to the specified format (fmt), according to the current rounding mode. The result is placed in the floating-point register (FPR) specified by fd.

This instruction is valid on the FPU only for single- and double-precision floating-point format. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent FPU general registers (FGR).

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) * ValueFPR (ft, fmt)) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception
Inexact Exception
Overflow Exception
Underflow Exception

NEG.fmt Floating-Point Negate

31–26	25	24–21	20–16	15–11	10–6	5–0
COP1	1	fmt	0	fs	fd	NEG
6	1	4	5	5	5	6

Format:

NEG.fmt fd, fs

Description:

The contents of the FPU register specified by fs are interpreted in the specified format and the arithmetic negation is taken (the polarity of the sign-bit is changed).
The result is placed in the FPU register specified by fd. If the register contains a signaling NaN, an invalid Operation Exception is generated.

The negated value is always exact.

On the FPU, this operation is valid only for single- and double-precision floating-point formats. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent coprocessor general registers (FGR).

Operation:

T :	StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt)))

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception

SUB.fmt Floating-Point Subtract

31 26	25	24 21	20 16	15 11	10 6	5 0
COP1	1	fmt	ft	fs	fd	SUB
6	1	4	5	5	5	6

Format:

SUB.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified format and arithmetically subtracted. The result is rounded as if calculated to infinite precision and then rounded to the specified format (fmt), according to the current rounding mode. The result is placed in the floating-point register (FPR) specified by fd.

This instruction is valid on the FPU only for single- and double-precision floating-point format. This operation is not defined if bit 0 of any register specification is set, as the register numbers specify an even-odd pair of adjacent FPU general registers (FGR).

Operation:

```
T:    StoreFPR(fd.fmt, ValueFPR(fs.fmt) - ValueFPR(ft.fmt)) ;
```

Exceptions:

Coprocessor unusable exception
Coprocessor Exception Trap

SUB.fmt

Floating-Point Subtract (Cont'd)

Floating-Point Exceptions:

Unimplemented Operation Exception
Invalid Operation Exception
Inexact Exception
Overflow Exception
Underflow Exception

SWC1 — Store Word from FPU (coprocessor 1)

31–26	25–21	20–16	15–0
SWC1	base	ft	offset
6	5	5	16

Format:

SWC1 ft, offset (base)

Description:

The 16-bit offset is sign-extended and added to the contents of the V_R3600 Processor's general register base to form a 32-bit unsigned effective address. The contents of the FPU's general register (FGR) at location ft is stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

```
T:  vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
    (pAddr, uncached) ← AddressTranslation(vAddr, DATA)
    data ← FGR[ft]
    StoreMemory(uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable exception
Bus error exception
Address error exception

Appendix A VR3600 CPU Instruction Opcode Bit Encoding

Opcode

Bit 31-29 \ Bits 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	*	*	*	*
3	*	*	*	*	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	SB	SH	SWL	SW	*	*	*	*
6	LWC0	LWC1	LWC2	LWC3	*	*	*	*
7	SWC0	SWC1	SWC2	SWC3	*	*	*	*

SPECIAL

Bit 5-3 \ Bits 2..0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	*
2	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	MULT	MULTU	DIV	DIVU	*	*	*	*
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

BCOND

Bit 20-19 \ Bits 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						
3								

*: Reserved for future versions of the architecture

COPz rs

Bit 25 - 24 / Bits 23..21

Bit 25 - 24	0	1	2	3	4	5	6	7
0	MF		CF		MT		CT	
1	BC							
2	CO							
3	CO							

COPz rt

Bit 20 - 19 / Bits 18..16

Bit 20 - 19	0	1	2	3	4	5	6	7
0	BCF	BCT						
1								
2								
3								

COP0 function [Bit 25="1"]

Bit 4 - 3 / Bits 2..0

Bit 4 - 3	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFE							
3								

Appendix B V_R3600 FPU Instruction Opcode Bit Encoding

opcode

Bit 31 - 29 \ Bits 28..26	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				*		
7		SWC1				*		

sub [Bit 25="0"]

Bit 24 \ Bits 23..21	0	1	2	3	4	5	6	7
0	MF	*	CF	*	MT	*	CT	*
1	BC	*	*	*	*	*	*	*

fmt [Bit 25="1"]

Bit 24 \ Bits 23..21	0	1	2	3	4	5	6	7
0	Single	Double	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*

br

Bit 20 - 19 \ Bits 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*

function

Bit 5-3	Bits 2..0: 0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	*	ABS	MOV	NEG
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*
4	CVT.S	CVT.S	*	*	CVT.W	*	*	*
5	*	*	*	*	*	*	*	*
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

*: Causes exceptions and reserved for future versions of the architecture.

© 1991 MIPS Computer Systems, Inc. All Rights Reserved.

RISCompiler,RISCwindows, and RISC/os are Trademarks of
MIPS Computer Systems, Inc.

UNIX is a Trademark of AT&T

Motif is a Trademark of OSF

MIPS Computer Systems, Inc.
950 DeGuigne Drive
Sunnyvale, CA 94086

NEC Customer Services

NEC's Commitment to Information

Our offices throughout Europe are always at your service for comprehensive support. Here are some of the technical services we provide:

- INSECT
- Seminars
- Update service
- Hotline
- Mailing list
- University program

INSECT

INformation **S**ystem and **E**lectronic **C**a**T**alog.

This is an on-line information service. Via a telephone link you can call up the latest data on all VLSI devices available from NEC. This includes enhancements, news and the most recent application know-how.

The service is free to our customers and other interested parties. For a menu-driven guest session, you can dial in to INSECT via the international packet switching network – in Germany this is DATEX-P – using of these numbers
45 21 10 13 020/030
and responding to the request for USERNAME and PASSWORD simply with "customer".

Seminars

No-one is more aware than NEC of the difference that a brief intensive training course can make to your mastery of advanced and often complex devices. We hold regular workshops and seminars at local NEC offices, in our Düsseldorf headquarters, or on customer premises. For information on NEC workshops and seminars, please contact your nearest office.

Update Service

When you buy an evaluation package from NEC, you become automatically entitled to one year's free updates for both hardware and software. All updates reach you fast and reliably via a courier service. In a field where rapid changes are the norm, you can be thus be sure of working with the most up-to-date development tools.

Hotlines

NEC's offices located throughout Europe are responsible for technical support and customer services. On the back cover of this brochure you can check which office is most convenient for you to contact. You are also welcome to contact our European headquarters in Düsseldorf directly.

Mailing list

Our engineering staff produces frequent additions to the available technical documentation in the form of application notes, product news and technical letters. If you would like to be included on our mailing list for this documentation, please inform us.

University Program

Many of our products, because of their complexity and dedicated application support through EB tools, are interesting subjects for graduate studies. NEC is always ready to discuss this possibility.

European Distributors

BELGIUM

ACAL NV S.A.
LOZENBERG 4
1932 ZAVENTEM
TEL.: (09) 32 27 20 59 83
FAX: (09) 32 27 25 10 14

DENMARK

MER-EL A/S
VED KLAEDEBO 18
2970 HOERSHOLM
TEL.: (42) 57 10 00
FAX: (42) 57 22 99

FINLAND

OY COMDAX AB
ITAELAHDENKATU 23A
PI 1
00210 HELSINKI
TEL.: (0) 67 02 77
FAX: (0) 6 92 23 26

FRANCE

ASAP COMPOSANTS
3, RUE FRANCOIS GEOFFRE
78190 TRAPPES
TEL.: (1) 30 12 20 00

CCI ELECTRONIQUE
5, AVENUE MARCELLIN BERTHELOT
BP 92
92164 ANTONY
TEL.: (1) 46 66 21 82
FAX: (1) 42 37 24 30

CGE COMPOSANTS
6 AVENUE MARECHAL JUIN
ZI GRANGE DAME ROSE
BP 55
92363 MEUDON LA FORET
TEL.: (1) 40 94 84 00
FAX: (1) 46 30 01 29

Especially for MICROWAVE and FIBER OPTIC COMPONENTS:
MILLIMONDES
PARC TECHNOLOGIQUE
BAT. COPERNIC
18/22 AV. EDOUARD HERIOT
92356 LE PLESSIS ROBINSON
TEL.: (1) 45 37 12 30
FAX: (1) 46 32 81 06

GERMANY

BIT-ELECTRONIC AG
DINGOLFINGER STRASSE 6
8000 MÜNCHEN 80
TEL.: (0 89) 41 80 07-0
FAX: (0 89) 41 80 07-20

DATA MODUL AG
LANDSBERGER STRASSE 320
8000 MÜNCHEN 21
TEL.: (0 89) 56 01 70
FAX: (0 89) 56 01 71 19

GLEICHMANN + CO.
ELECTRONICS GMBH
WORMSER STRASSE 34
6710 FRANKENTHAL
TEL.: (0 62 33) 2 50 56
FAX: (0 62 33) 2 02 98

GLYN GMBH
AM WÖRTZGARTEN 8
6270 IDSTEIN/TS.
TEL.: (0 61 26) 59 02 22
FAX: (0 61 26) 59 01 11

MICROSCAN GMBH
ÜBERSEERING 31
2000 HAMBURG 60
TEL.: (0 40) 6 32 00 30
FAX: (0 40) 6 32 00-349

REIN ELEKTRONIK GMBH
LÖTSCHWEG 66
4054 NETTETAL 1
TEL.: (0 21 53) 73 31 11
FAX: (0 21 53) 73 31 10

ULTRATRONIK GMBH
GEWERBESTRASSE 4
8036 HERRSCHING
TEL.: (0 81 52) 37 09-0
FAX: (0 81 52) 51 83

UNIELECTRONIC
VERTRIEBS GMBH
IM GEFIERTH 11A
6072 DREIEICH 1 B. FRANKFURT
TEL.: (0 61 03) 3 51 75
FAX: (0 61 03) 3 59 48

Especially for DISCRETE Products:
ROEDERSTEIN GMBH
UNTERNEHMENSBEREICH HALBLEITER
LUDMILLASTRASSE 23-25
8300 LANDSHUT
TEL.: (08 71) 8 61
FAX: (08 71) 8 62 91

Especially for GAS LASER:
GERHARD FRANCK OPTRONIK GMBH
BERZELIUSSTRASSE 89
2000 HAMBURG 74
TEL.: (0 40) 7 33 60 50
FAX: (0 40) 73 36 05 30

Especially for MICROWAVE:
OMECON ELECTRONIC GMBH
RAIFFEISENSTRASSE 12
8150 HOLZKIRCHEN
TEL.: (0 80 24) 6 40 80
FAX: (0 80 24) 64 08 70

ITALY

ADELSY S.R.L.
VIA DEL FONDITORE, 5
LOCALITA ROVERI
40127 BOLOGNA
TEL.: (0 51) 53 21 19
FAX: (0 51) 6 01 00 76

CLAITRON S.P.A.
VIA GALLARATE, 211
20151 MILANO
TEL.: (02) 33 40 40 00
FAX: (02) 38 00 14 14

DIS.EL. S.P.A.
VIA ALA DI STURA 71
10148 TORINO
TEL.: (0 11) 2 91 93 00
FAX: (0 11) 2 91 93 80

MELCHIONI S.P.A.
VIA COLLETTA, 37
20135 MILANO
TEL.: (02) 5 79 41
FAX: (02) 55 18 19 14

European Distributors

NETHERLANDS
ACAL AURIEMA B.V.
DOORNAKKERSWEG 26
5642 MP EINDHOVEN
TEL.: (0 40) 81 65 65
FAX: (0 40) 81 18 15

MALCHUS B.V.
FOKKERSTRAAT 511 - 513
3125 BD SCHIEDAM
TEL.: (10) 4 27 77 77
FAX: (0 10) 4 15 44 66

INNOCIRCUIT BENELUX B.V.
POSTBUS 48
3100 AA SCHIEDAM
TEL.: (10) 4 27 77 80
FAX: (0 10) 4 15 44 66

Especially for GAS LASERS:
FAIRLIGHT
TECHNISCHE & WETENSCHAPPELIJKE
APPARATEN B.V.
MARSHALLWEG 45
3068 JN ROTTERDAM
TEL.: (0 10) 4 20 64 44
FAX: (0 10) 4 20 65 11

NORWAY
JAKOB HATTELAND
ELECTRONIC A/S
P.B. 25
5578 NEDRE VATS
TEL.: (47) 6 31 11
FAX: (47) 6 53 39

PORTUGAL
COMELTA S.A.
RUA DE CABO VERDE
LOTE 113 LOJA
2685 SACAVEM
LISBOA
TEL.: (1) 9 42 40 81
FAX: (1) 9 42 41 55

SPAIN
COMELTA DA
EMILIO MUNOZ 41, NAVE 1-1-2
28037 MADRID
TEL.: (1) 3 27 06 14
FAX: (1) 3 27 05 40

VENCO
CARRETERA DEL MIG, 75
L'HOSPITALET DE LLOBREGAT
08907 BARCELONA
TEL.: (3) 2 63 33 54
FAX: (3) 2 63 33 23

Especially for MICROWAVE and FIBER OPTIC COMPONENT:
IBERCOM
RAMON GOMEZ DE LA SERNA, 1-2A
28035 MADRID
TEL.: (1) 3 73 62 34
FAX: (1) 3 73 54 64

SWEDEN
NAX AB KOMPONENTBOLAGET
BOX 41 15
17104 SOLNA
TEL.: (8) 98 51 40
FAX: (8) 7 64 54 51

Especially for CIRCUIT COMPONENTS and RELAYS:
ERICSSON PROCOMP
16481 KISTA, STOCKHOLM
TEL.: (8) 7 57 50 00
FAX: (8) 7 57 41 10

Especially for MICROWAVE and FIBER OPTIC COMPONENTS:
SANGUS AB
BOX 5004
16205 VÄLLINGBY
TEL.: (8) 38 02 10
FAX: (8) 38 27 35

SWITZERLAND
MEMOTEC AG
GASWERKSTRASSE 32
4901 LANGENTHAL
TEL.: (63) 28 11 22
FAX: (63) 22 35 06

Especially for RELAYS:
ERNI & CO AG
ELEKTROINDUSTRIE
8306 BRUETTISELLEN
TEL.: (1) 8 35 35 35
FAX: (1) 8 33 49 66

UNITED KINGDOM
2001 ELECTRONIC COMPONENTS
WOOLMERS WAY
STEVENAGE
HERTS
SG1 3AJ
TEL.: (4 38) 74 20 01
FAX: (4 38) 74 20 02

ANZAC COMPONENTS LIMITED
822, YEOVIL ROAD
SLOUGH TRADING ESTATE
SLOUGH
BERSHIRE
SL1 4JA
TEL.: (62 86) 44 11
FAX: (7 53) 69 18 72

CELDIS LIMITED
37, LOVEROCK ROAD
READING
BERKSHIRE
RG3 1EL
TEL.: (7 34) 58 51 71
FAX: (7 34) 50 99 33

IMPULSE ELECTRONICS LIMITED
HAMMOND HOUSE
CATERHAM
SURREY
CR3 6XG
TEL.: (8 83) 34 64 33
FAX: (8 83) 34 60 61

S.T.C. MULTICOMPONENTS LIMITED
EDINBURGH WAY
HARLOW
ESSEX
CM20 2DF
TEL.: (2 79) 44 11 44
FAX: (2 79) 44 34 17