NEC

User's Manual

V_R5432™

64-Bit MIPS® RISC Microprocessor

Volume 2

μPD30541GD

Document No.U15397EU5V0UMJ1
Date Published: May 2001 CP (K)

© **NEC Electronics Inc.** 2000
Printed in U.S.A.

NEC

VR5432 Microprocessor User’s Manual
Document Number U15397EU5V0UMJ1

Revision History

February 1999: First release
August 1999: Version 2, Preliminary
December 1999: Version 3, Preliminary update
February 2000: Version 4, Document release
May 2000: Version 5, Document corrections

NEC, the NEC logo, VR Series, VR3000, VR4000, VR4300, VR5000, VR5432 and VR10000 are registered trademarks of NEC Corporation. All other product, brand, or trade names used in this publication are the trademarks or registered trademarks of their respective trademark owners.

In North America: No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics Inc. (NECEL). The information in this document is subject to change without notice. All devices sold by NECEL are covered by the provisions appearing in NECEL's Terms and Conditions of Sales only, including the limitation of liability, warranty, and patent provisions. NECEL makes no warranty, express, statutory, implied or by description, regarding information set forth herein or regarding the freedom of the described devices from patent infringement. NECEL assumes no responsibility for any errors that may appear in this document. NECEL makes no commitments to update or to keep current information contained in this document. The devices listed in this document are not suitable for use in applications such as, but not limited to, aircraft control systems, aerospace equipment, submarine cables, nuclear reactor control systems and life support systems. “Standard” quality grade devices are recommended for computers, office equipment, communication equipment, test and measurement equipment, machine tools, industrial robots, audio and visual equipment, and other consumer products. For automotive and transportation equipment, traffic control systems, anti-disaster and anti-crime systems, it is recommended that the customer contact the responsible NECEL salesperson to determine the reliability requirements for any such application and any cost adder. NECEL does not recommend or approve use of any of its products in life support devices or systems or in any application where failure could result in injury or death. If customers wish to use NECEL devices in applications not intended by NECEL, customers must contact the responsible NECEL salespeople to determine NECEL's willingness to support a given application.

Contents

Volume 1

Preface

The VR5432™ microprocessor is an NEC VR Series™ RISC (reduced instruction set computer) microprocessor that implements the high-performance 64-bit MIPS® IV architecture. This manual describes the architecture and hardware functions of the VR5432 microprocessor.

Legend

Data significance:	Higher on left and lower on right
Active-high signal name:	XXX
Active-low signal name:	XXX*
Numeric representation:	binary ... XXXX or $XXXX_2$
	decimal ... XXXX
	hexadecimal ... 0xXXXX

Prefixes representing an exponent of 2 (for address space or memory capacity):

K (kilo)	$2^{10} = 1024$
M (mega)	$2^{20} = 1024^2$
G (giga)	$2^{30} = 1024^3$
T (tera)	$2^{40} = 1024^4$

Manual Overview

The manual is divided into two volumes. Volume 1 is the user manual, containing processor architectural and functional information and instructions. Volume 2 contains the instruction set information and appendixes.

Volume 1 (U13751E)

Chapter 1: Introduction provides an overview of the device features, CPU, Floating-Point Unit (FPU), and pipeline.

Chapter 2: Signal Descriptions discusses the pin configuration and functions of the VR5432 processor signals.

Chapter 3: Pipeline describes the dual-issue instruction pipeline stages, delays, and interlock and exception handling.

Chapter 4: Memory Management Unit discusses the processor's virtual and physical address spaces, the virtual-to-physical address translation, the translation lookaside buffer (TLB) process, and the system control coprocessor registers that provide the software interface to the TLB.

Chapter 5: Cache Organization and Operation describes the cache memory's place in the VR5432 memory configuration and individual cache organization.

Chapter 6: CPU Exceptions describes the processor's exception types, registers, vector offsets, processing handling, and interrupts.

Chapter 7: Floating-Point Unit describes the FPU coprocessor, including the programming model, instruction set and formats, and the pipeline.

Chapter 8: Floating-Point Exceptions discusses FPU exception types, exception trap processing, exception flags, saving and restoring states when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

Chapter 9: Bus Interface describes how the processor accesses the external resources needed to satisfy cache misses and uncached operations, while permitting an external agent access to some of the processor's internal resources.

Chapter 10: System Interface Transactions (Native Mode) describes processor and external requests in the native system interface protocol of the VR5432 processor.

Chapter 11: System Interface Protocols (Native Mode) contains a cycle-by-cycle description of the system interface protocols for each type of processor and external request in the native protocol of the VR5432 processor.

Chapter 12: System Interface Transactions (R43K Mode) This section describes processor and external requests as they occur in R43K (VR4300 compatibility) mode.

Chapter 13: System Interface Protocols (R43K Mode) contains a cycle-by-cycle description of the system interface protocols for each type of processor and external request in R43K mode.

Chapter 14: Initialization Interface describes the processor reset and initialization signals.

Chapter 15: Clock Interface describes the basic system clocks, SysClock and PClock, and Phase-Locked Loop (PLL) and Bypass PLL modes.

Volume 2 (U15397E)

Chapter 16: Instruction Set Overview discusses the general attributes of the CPU, FPU, multimedia, and debugging instructions of the MIPS IV instruction set architecture (ISA) utilized by the VR5432 processor.

Chapter 17: CPU Instruction Set describes the details of the CPU instructions.

Chapter 18: Floating-Point Unit Instruction Set describes the details of the FPU instructions.

Chapter 19: Multimedia Instruction Set describes the details of the multimedia instructions.

Chapter 20: Debug and Test Features describes the VR5432 processor's debug and test functions, Debug mode, and debug instructions.

Appendix A: Sublock Order describes how a block of data elements (bytes, halfwords, words, or doublewords) can be retrieved from storage in sequential or nonsequential (sub-block) order.

Appendix B: Comparing the VR4300, VR5000, and VR5432 Processors delineates each processor's attributes.

Appendix C: PLL Analog Power Filtering illustrates the phase-locked loop circuit configuration.

Appendix D: Instruction Hazards identifies the VR5432 instruction hazards that occur with certain instruction and event combinations (such as pipeline delays, cache misses, interrupts, and exceptions).

Related Documents

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

Product	Data Sheet	User's Manual	
		Hardware Architecture	Instruction Set
VR5432	U13504E	U13751E	U15397E
VR5000	U12031E	U11761E	U12754E
VR10000	U12703E	U10278E	U12754E

Instruction Set Overview

16

This chapter provides an overview of the instruction set architecture (ISA) utilized by the VR5432 processor. For detailed information on each instruction type, refer to the following chapters.

- Chapter 17, CPU Instruction Set, on page 3
- Chapter 18, Floating-Point Unit Instruction Set, on page 569
- Chapter 19, Multimedia Instruction Set, on page 677
- Chapter 20, Debug and Test Features, on page 737

16.1 Instruction Set Architecture

The VR5432 processor executes the MIPS IV instruction set (a superset of the MIPS III instruction set) plus instructions added by NEC specifically for VR5432 implementation. As Figure 16-1 illustrates, each new architecture level (or version) includes the former levels. Therefore, a processor implementing MIPS IV can also run MIPS I, MIPS II, or MIPS I II binary programs without change.

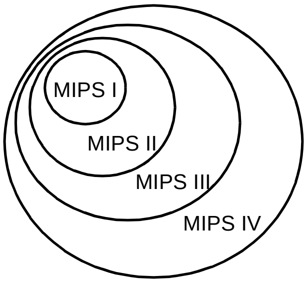


Figure 16-1 MIPS Architecture Extensions

The MIPS IV instruction set for the VR5432 processor utilizes the following instruction types.

- CPU instructions
- Floating-point instructions
- Multimedia instruction
- Test and debug instructions

In earlier MIPS architectures, coprocessor instructions were implementation dependent. In the MIPS IV architecture, the Coprocessor 3 instruction formats have been used for extensions to the floating-point instruction set. In the VR5432 implementation, the Coprocessor 2 instruction formats have been used for implementation-specific instruction set extensions. The new MIPS IV, VR5432 processor-specific instructions are summarized and briefly explained in Section 16.8.

16.2 Instruction Formats

Each instruction consists of a single 32-bit word aligned on a word boundary. There are three instruction formats—immediate (I-type), jump (J-type), and Register (R-type). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats. See the subsequent instruction chapters for details on the formats of each instruction type.

16.3 Load and Store Instructions

Load and Store instructions are immediate (I-type) instructions that transfer data between the memory system and the general-purpose register sets in the CPU and coprocessors. There are separate instructions for different purposes: transferring variously sized fields, treating loaded data as signed or unsigned integers, accessing unaligned fields, selecting the addressing mode, and providing atomic memory updates (read-modify-write cycles).

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address among the bytes forming the object. For big-endian ordering, this is the most-significant byte; for little-endian ordering, this is the least-significant byte.

Except for the few specialized instructions listed in Table 17-2, Load and Store instructions must access naturally aligned objects. An attempt to load or store an object at an address that is not an even multiple of the size of the object will cause an Address Error exception.

Load and Store operations have been added in each revision of the architecture:

MIPS II

- 64-bit coprocessor transfers
- Atomic update

MIPS III

- 64-bit CPU transfer
- Unsigned word load for the CPU

MIPS IV: Register + r egister addressing mode for the FPU

Table 16-1 and Table 16-2 tabulate the supported Load and Store operations and indicate the MIPS architecture level at which each operation was first supported. The instructions themselves are listed in the following sections.

Table 16-1 Load/Store Operations Using Register + Offset Addressing Mode

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

Table 16-2 Load/Store Operations Using Register + Register Addressing Mode

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

16.3.1 Delayed Load Instructions

The MIPS I architecture defines delayed loads; an instruction scheduling restriction requires that an instruction immediately following a load into register Rn cannot use Rn as a source register. The time between the Load instruction and the time the data is available is the “load delay slot.” If no useful instruction can be put into the load delay slot, then a null operation (assembler mnemonic NOP) must be inserted.

In MIPS II, this instruction scheduling restriction is removed. Programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra read cycles. Most processors cannot actually load data quickly enough for immediate use and the processor will be forced to wait until the data is available. Scheduling load delay slots can be desirable, both for performance and compatibility with earlier VR Series processors. However, the scheduling of load delay slots is not required for correct operation of the processor.

16.3.2 Defining Access Types

Access type indicates the size of a VR5432 processor data item to be loaded or stored, as set by the Load or Store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Table 16-3). Only the combinations shown in Table 16-3 are permissible; other combinations cause Address Error exceptions.

Table 16-3   Byte Access within a Doublew o r

Access Type Mnemonic (Value)	**Low-Order Address Bits**			**Bytes Accessed**															
				Big Endian (63-----------31------------0) Byte								**Little Endian (63-----------31------------0) Byte**							
	2	**1**	**0**																
Doubleword (7)	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Septibyte (6)	0	0	0	0	1	2	3	4	5	6	▨	▨	6	5	4	3	2	1	0
	0	0	1	▨	1	2	3	4	5	6	7	7	6	5	4	3	2	1	▨
Sextibyte (5)	0	0	0	0	1	2	3	4	5	▨	▨	▨	▨	5	4	3	2	1	0
	0	1	0	▨	▨	2	3	4	5	6	7	7	6	5	4	3	2	▨	▨
Quintibyte (4)	0	0	0	0	1	2	3	4	▨	▨	▨	▨	▨	▨	4	3	2	1	0
	0	1	1	▨	▨	▨	3	4	5	6	7	7	6	5	4	3	▨	▨	▨
Word (3)	0	0	0	0	1	2	3	▨	▨	▨	▨	▨	▨	▨	▨	3	2	1	0
	1	0	0	▨	▨	▨	▨	4	5	6	7	7	6	5	4	▨	▨	▨	▨
Triplebyte (2)	0	0	0	0	1	2	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	2	1	0
	0	0	1	▨	1	2	3	▨	▨	▨	▨	▨	▨	▨	▨	3	2	1	▨
	1	0	0	▨	▨	▨	▨	4	5	6	▨	▨	6	5	4	▨	▨	▨	▨
	1	0	1	▨	▨	▨	▨	▨	5	6	7	7	6	5	▨	▨	▨	▨	▨
Halfword (1)	0	0	0	0	1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	0
	0	1	0	▨	▨	2	3	▨	▨	▨	▨	▨	▨	▨	▨	3	2	▨	▨
	1	0	0	▨	▨	▨	▨	4	5	▨	▨	▨	▨	5	4	▨	▨	▨	▨
	1	1	0	▨	▨	▨	▨	▨	▨	6	7	7	6	▨	▨	▨	▨	▨	▨
Byte (0)	0	0	0	0	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	0
	0	0	1	▨	1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	▨
	0	1	0	▨	▨	2	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	2	▨	▨
	0	1	1	▨	▨	▨	3	▨	▨	▨	▨	▨	▨	▨	▨	3	▨	▨	▨
	1	0	0	▨	▨	▨	▨	4	▨	▨	▨	▨	▨	▨	4	▨	▨	▨	▨
	1	0	1	▨	▨	▨	▨	▨	5	▨	▨	▨	▨	5	▨	▨	▨	▨	▨
	1	1	0	▨	▨	▨	▨	▨	▨	6	▨	▨	6	▨	▨	▨	▨	▨	▨
	1	1	1	▨	▨	▨	▨	▨	▨	▨	7	7	▨	▨	▨	▨	▨	▨	▨

Table 16-4 Access Type Specifications for Load/Store Instru c t i o n

Access Type	SysCmd (2:0)	Meaning
Doubleword	7	8 bytes (64 bits)
Septibyte	6	7 bytes (56 bits)
Sextibyte	5	6 bytes (48 bits)
Quintibyte	4	5 bytes (40 bits)
Word	3	4 bytes (32 bits)
Triplebyte	2	3 bytes (24 bits)
Halfword	1	2 bytes (16 bits)
Byte	0	1 byte (8 bits)

16.4 Computational Instructions

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

Two's-complement arithmetic is performed on integers represented in two's-complement notation. There are signed versions of add, subtract, multiply, and divide operations. There are add and subtract operations, called "unsigned," that are actually modulo arithmetic without overflow detection. There are unsigned versions of multiply and divide. There is a full complement of shift and logical operations.

MIPS I provides 32-bit integers and 32-bit arithmetic. MI PSIII adds 64-bit integers and provides separate Arithmetic and Shift instructions for 64-bit operands. Logical operations are not sensitive to the width of the register.

Computational instructions perform the following operations on register values:

- Arithmeti
- Logical
- Rotate
- Shift
- Multipl
- Divide
- Multiply-accumulat
- Parallel operations on packed bytes

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

- ALU immediate instruction
- Three-operand register-type instructions
- Rotate and Shift instructions
- Multiply and Divide instructions
- Multiply-accumulate instructions
- Packed byte instructions

16.4.1 64-Bit Operations

The VR5432 microprocessor has a 64-bit architecture that supports 32-bit operands. These operands must be sign extended. Opcodes are available for 32-bit operands for all of the basic arithmetic and logical instructions, such as: ADD, ADDU, SUB, SUBU, ADDI, SLL, SRA, and SLLV. Operations that don’t use sign-extended 32-bit values correctly are unpredictable. In addition, 32-bit data is stored sign extended in a 64-bit register.

16.5 Jump and Branch Instructions

All Jump and Branch instructions have a delay slot of exactly one instruction. That is, the instruction immediately following a Jump or Branch instruction (the instruction occupying the delay slot) is executed while the target instruction is being fetched from the cache. A Jump or Branch instruction cannot be used in a delay slot; however, if they are used, the error is not detected and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of the instruction while it is in a delay slot, the hardware sets a virtual address to the EPC register at the point of the Jump or Branch instruction that precedes it. When exception or interrupt processing is complete and the program is restored, both the Jump and Branch instruction and the instruction in the delay slot are re-executed.

Because Jump and Branch instructions may be re-executed after exception or interrupt processing, register 31 (the register in which the link address is stored) should not be used as a source register in Jump, Link/Branch, and Link instructions.

Because instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register that contains an address where the low-order two bits are zero. If these low-order two bits are not zero, an Address Error Exception instruction at the Jump destination is fetched.

16.5.1 Jump Instructions

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

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

16.5.2 Branch Instructions

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

If a conditional Branch Likely instruction is not taken, the instruction in the delay slot is nullified (i.e., discarded without affecting any data).

16.6 Special Instructions

Special instructions allow the software to initiate traps both conditionally and unconditionally. These instructions can cause System Call (SysCall), Breakpoint (Break), and Trap (Trap) conditions in the processor. SysCall and Break are unconditional, while Trap can specify a condition such as a Branch instruction. The Synchronize (Sync) instruction allows the software to ensure that all pending operations are complete. In the VR5432 processor implementation, the Sync instruction is executed as an NOP.

16.7 Coprocessor Instructions

Coprocessors are alternate execution units with register files separate from the CPU. The MIPS architecture provides an abstraction for up to 4 coprocessor units, numbered 0 to 3. Each architecture level defines some of these coprocessors, as shown in Table 16-5. Coprocessor 0 is always used for system control and Coprocessor 1 is used for the floating-point unit. Other coprocessors are architecturally valid, but do not have a reserved use. Some coprocessors are not defined and their opcodes are either reserved or used for other purposes.

Table 16-5 Coprocessor Definition and Use in the MIPS Architect u r

	MIPS Architecture Level			
Coprocessor	**I**	**II**	**III**	**IV**
0	Sys. control	Sys. control	Sys. control	Sys. control
1	FPU	FP	FP	FPU
2	Unused	Unused	Unused	Unused
3	Unused	Unused	Not defined	FPU (COP1X)

The coprocessors may have two register sets, Coprocessor general-purpose registers and coprocessor control registers, with each set containing up to 32 registers. Coprocessor computational instructions may alter registers in either set.

System control for all MIPS processors is implemented as Coproce ssor0 (CP0), the system control coprocessor. It provides the processor control, memory management, and exception handling functions. The CP0 instructions are specific to each CPU and are documented with the CPU-specific information.

If a system includes a floating-point unit, it is implemented as coproces sor1 (CP1). In MIPS I V, the FPU also uses the computation opcode space for Coprocessor unit 3, renamed COP1X. The FPU instructions are documented in Chapter 18.

The coprocessor instructions are divided into two main groups:

- Load and Store instructions that are reserved in the main opcode space
- Coprocessor-specific operations that are defined entirely by the coprocessor

16.7.1 Coprocessor Load and Store

Load and Store instructions are not defined for CP0; the Move to/from Coprocessor instructions provide the only way to write and read the CP0 registers.

16.7.2 Coprocessor Operations

Up to four coprocessors and their instructions are shown generically for coprocessor z. Within the operation main opcode, the coprocessor has further coprocessor-specific instructions encoded.

Table 16-6 Coprocessor Operation Instr u c t i o n

Mnemonic	Description	Defined in MIPS...
COPz	Coprocessor z Operation	I

16.8 Implementation-Specific Instructions

16.8.1 Overview

The MIPS IV instructions added by NEC for the VR5432 processor enable the MIPS architecture to compete in the high-end numeric processing market, which has traditionally been dominated by vector architectures.

Compound Multiply-Add instructions are included, taking advantage of the fact that most floating-point computations use the chained multiply-add paradigm. The intermediate multiply result is rounded before the addition is performed.

A register + register addressing indexed mode for floating-point loads and stores eliminates the extra integer required in many array accesses. However, issuing a register + register load causes a one-cycle stall in the pipeline, which makes it useful only for compatibility with other MIPS IV implementations. Register + register indexed addressing for integer memory operations is not supported.

A set of four conditional move operators allows floating-point arithmetic IF statements to be represented without branches. THEN and ELSE clauses are computed unconditionally and the results are placed in a temporary register. Conditional move operators then transfer the temporary results to their true register. Conditional moves must be able to test both integer and floating-point conditions in order to supply the full range of IF statements. Integer tests are performed by comparing a general-purpose register against a zero value.

Because floating-point conditional moves test the floating-point condition codes, the VR5432 processor provides eight condition codes to give the compiler increased flexibility in scheduling the comparison and the conditional moves.

Table 16-7 lists the new instructions that complete the MIPS IV instruction set; these instructions are described in Section 16.8.2 on page 333.

Table 16-7   MIPS IV Instruction Additions

Instruction	Definition
BC1F	Branch on FP condition code false
BC1T	Branch on FP condition code true
BC1FL	Branch on FP condition code false likely
BC1TL	Branch on FP condition code true likely
C.cond.fmt (cc)	Floating-point compare
LDXC1	Load doubleword indexed to COP1
LWXC1	Load word indexed to COP1
MADD.fmt	Floating-point multiply-add
MOVF	Move conditional on FP condition code false
MOVN	Move on register not equal to zero
MOVT	Move conditional on FP condition code true
MOVZ	Move on register equal to zero
MOVF.fmt	FP move conditional on condition code false
MOVN.fmt	FP move on register not equal to zero
MOVT.fmt	FP move conditional on condition code true
MOVZ.fmt	FP move conditional on register equal to zero
MSUB.fmt	Floating-point multiply-subtract
NMADD.fmt	Floating-point negative multiply-add
NMSUB.fmt	Floating-point negative multiply-subtract
PREFX	Prefetch indexed — register + register
PREF	Prefetch — register + offset
RECIP.fmt	Reciprocal
RSQRT.fmt	Reciprocal square root
SDXC1	Store doubleword indexed from COP1
SWXC1	Store word indexed from COP1

16.8.2 Implementation-Specific Instruction Descriptions

This section describes the new instructions listed in Table 16-7.

16.8.2.1 Branch on floating-point Coprocessor instructions

BC1T: Branch on FP condition True

BC1F: Branch on FP condition False

BC1TL: Branch on FP condition True Likely

BC1FL: Branch on FP condition False Likely

The four Branch instructions are upwardly compatible extensions of the Branch on floating-point coprocessor instructions of the MIPS instruction set. The BC1T and BC1F instructions are extensions of MIPS I. BC1TL and BC1FL are extensions of MIPS III. These instructions test one of eight floating-point condition codes. This encoding is upwardly compatible with previous MIPS architectures.

The 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 64 bits. If the contents of the floating-point condition code specified in the instruction are equal to the test value, the target address is branched to with a delay of one instruction. If the conditional branch is not taken and the nullify delay bit in the instruction is set, the instruction in the branch delay slot is nullified.

16.8.2.2 Floating-point Compare instructions

C.cond.fmt: Compares the contents of two FPU registers

The contents of the two FPU source registers specified in the instruction are interpreted and arithmetically compared. A result is determined based on the comparison and the conditions specified in the instruction.

16.8.2.3 Indexed floating-point Load instructions

LWXC1: Load word indexed to Coprocessor 1

LDXC1: Load doubleword indexed to Coprocessor 1

The two indexed floating-point Load instructions are exclusive to the MIPS IV instruction set and transfer floating-point data types from memory to the floating-point registers using the register + register addressing mode. There are no indexed loads to general-purpose registers. The contents of the general-purpose register specified by the base are added to the contents of the general-purpose register specified by the index to form a virtual address. The contents of the word or doubleword specified by the effective address are loaded into the floating-point register specified in the instruction.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits, an Address Error exception occurs. Also, if the address is not aligned, an Address Error exception occurs.

16.8.2.4 Integer conditional Move instructions

MOVT: Move conditional on condition code True

MOVF: Move conditional on condition code False

MOVN: Move conditional on register not equal to zero

MOVZ: Move conditional on register equal to zero

The four-integer Move instructions are exclusive to the MIPS IV instruction set and are used to test a condition code or a general-purpose register and then conditionally perform an integer move. The value of the floating-point condition code specified in the instruction by the 3-bit condition code specifier, or the value of the register indicated by the 5-bit general-purpose register specifier, is compared to zero. If the result indicates that the move should be performed, the contents of the specified source register are copied into the specified destination register.

16.8.2.5 Floating-point Multiply-Add instructions

MADD: Floating-point Multiply-Add

MSUB: Floating-point Multiply-Subtract

NMADD: Floating-point Negative Multiply-Add

NMSUB: Floating-point Negative Multiply-Subtract

These four instructions are exclusive to the MIPS IV instruction set and accomplish two floating-point operations with one instruction. Each of these four instructions performs intermediate rounding.

16.8.2.6 Floating-point conditional Move instructions

MOVT.fmt: Floating-point conditional move on condition code True

MOVF.fmt: Floating-point conditional move on condition code False

MOVN.fmt: Floating-point conditional move on register not equal to zero

MOVZ.fmt: Floating-point conditional move on register equal to zero

The four floating-point Conditional Move instructions are exclusive to the MIPS IV instruction set and are used to test a condition code or a general-purpose register and then conditionally perform a floating-point move. The value of the floating-point condition code specified by the 3-bit condition code specifier, or the value of the register indicated by the 5-bit general-purpose register specifier, is compared to zero. If the result indicates that the move should be performed, the contents of the specified source register are copied into the specified destination register. All of these conditional floating-point move operations are non-arithmetic. Consequently, no IEEE-754 exceptions occur as a result of these instructions.

16.8.2.7 Prefetch instructions

PREF: Register + offset format

PREFX: Register + register format

The two Prefetch instructions are exclusive to the MIPS IV instruction set and allow the compiler to issue instructions early so the corresponding data can be fetched and placed as close as possible to the CPU. Each instruction contains a 5-bit “hint” field that gives the coherency status of the line being prefetched. The line can be shared, exclusive clean, or exclusive dirty. The contents of the general-purpose register specified by the base are added either to the 16-bit sign-extended offset or to the contents of the general-purpose register specified by the index to form a virtual address. This address and “hint” field are sent to the cache controller and a memory access is initiated.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits, an Address Error exception occurs. The Prefetch instruction never generates TLB-related exceptions. The PREF instruction is considered a standard processor instruction, while the PREFX instruction is considered a standard Coprocessor 1 instruction.

16.8.2.8 Reciprocal instructions

RECIP.fmt: Reciprocal

RSQRT.fmt: Reciprocal Square Root

The Reciprocal instruction performs a reciprocal on a floating-point value. The reciprocal of the value in the floating-point source register is placed in a destination register.

The Reciprocal Square Root instruction performs a reciprocal square root on a floating-point value. The reciprocal of the positive square root of a value in the floating-point source register is placed in a destination register.

The VR5432 meets full IEEE accuracy requirements for the RECIP and RSQRT instructions. On the VR5432 microprocessor, the RECIP instruction has the same latency as a DIV instruction, but an RSQRT is faster than a SQRT followed by a RECIP.

16.8.2.9 Indexed floating-point Store instructions

SWXC1: Store word indexed from Coprocessor 1

SDXC1: Store doubleword indexed from Coprocessor 1

The two indexed floating-point Store instructions are exclusive to the MIPS IV instruction set and transfer floating-point data types from the floating-point registers to memory using the register + register addressing mode. There are no indexed stores from general-purpose registers. The contents of the general-purpose register specified by the base are added to the contents of the general-purpose register specified by the index to form a virtual address. The contents of the floating-point register specified in the instruction are stored to the memory location specified by the effective address.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits, an Address Error exception occurs. Also, if the address is not aligned, an Address Error exception occurs.

16.9 Integer Rotate Instructions

The VR5432 processor adds a set of Rotate instructions that are not part of the standard MIPS instruction set.

Table 16-8 Rotate Instructi o n

Instruction	**Definition**
DROR	Doubleword rotate right
DROR32	Doubleword rotate right plus 32
DRORV	Doubleword rotate right variable
ROR	Rotate right
RORV	Rotate right variable

16.10 Integer Multiply-Accumulate Instructions

The VR5432 processor includes a set of Multiply-Accumulate instructions that are not part of the standard MIPS instruction set. These instructions use half of the HI and LO registers together as a 64-bit accumulator, with the upper 32 bits of the accumulator mapped to the lower 32 bits of HI and the lower 32 bits of the accumulator mapped to the lower 32 bits of LO. These instructions perform no underflow or overflow detection and produce no exceptions. Table 16-9 lists these instructions.

Table 16-9 Multiply-Accumulate Instruction Set Extensions

Instruction	**Definition**
MACC	Multiply, accumulate, and move LO
MACCHI	Multiply, accumulate, and move HI
MACCHIU	Unsigned multiply, accumulate, and move HI
MACCU	Unsigned multiply, accumulate, and move LO
MSAC	Multiply, negate, accumulate, and move LO
MSACHI	Multiply, negate, accumulate, and move HI
MSACHIU	Unsigned multiply, negate, accumulate, and move HI
MSACU	Unsigned multiply, negate, accumulate, and move LO
MUL	Multiply and move LO
MULHI	Multiply and move HI
MULHIU	Unsigned multiply and move HI
MULS	Multiply, negate, and move LO
MULSHI	Multiply, negate, and move HI
MULSHIU	Unsigned multiply, negate, and move HI
MULSU	Unsigned multiply, negate, and move LO
MULU	Unsigned multiply and move LO

Table 16-10 Multiply-Accumulate Instruction Latency and Repeat Rat e

Instruction	Latency	Repeat Rate
MACC, MACCHI, MACCHIU, MACCU	3	1
MSAC, MSACHI, MSACHIU, MSACU	3	1
MUL, MULHI, MULHIU, MULU	3	1
MULS, MULSHI, MULSHIU, MULSU	3	1

16.11 Multimedia Extensions

The VR5432 adds a set of instructions to operate on packed vectors of eight 8-bit unsigned integers. These instructions are described in Chapter 19.

Table 16-11 Multimedia Extensions

Instruction	Definition
ADD.OB	Vector add
ALNI.OB	Vector align
AND.OB	Vector AND
C.EQ.OB	Vector compare equal
C.LE.OB	Vector compare less than or equal
C.LT.OB	Vector compare less than
MAX.OB	Vector maximum
MIN.OB	Vector minimum
MUL.OB	Vector multiply
MULA.OB	Vector multiply-accumulate
MULS.OB	Vector multiply, negate, and accumulate
MULSL.OB	Vector multiply, negate, and load accumulator
NOR.OB	Vector NOR
OR.OB	Vector OR
PICKF.OB	Vector pick false
PICKT.OB	Vector pick true
RZU.OB	Vector scale, round, and clamp accumulator
SHFL.MIXH.OB	Vector element shuffle
SHFL.MIXL.OB	Vector element shuffle
SHFL.PACH.OB	Vector element shuffle

Table 16-11 Multimedia Extensions (continued)

Instruction	Definition
SHFL.PACL.OB	Vector element shuffle
SLL.OB	Vector shift left logical
SRL.OB	Vector shift right logical
SUB.OB	Vector subtract
XOR.OB	Vector XOR

16.12 Debugging Instructions

The VR5432 processor adds a set of instructions to control the on-chip debugging features described in Chapter 20.

Table 16-12 Debug Instructi o n

Instruction	Definition
DBREAK	Debug break
DRET	Debug return
MFDR	Move from Debug register
MTDR	Move to Debug register

16.12.1 Instruction Notation Conventions

In the following instruction set chapters, all variable subfields in instruction formats (such as *rs, rt, fs, ft, immediate*, and so on) are shown in lowercase.

For clarity, sometimes an alias is used for a variable subfield in the formats of specific instructions. For example, *rs* = *base* in the format for Load and Store instructions. Such an alias is always lowercase, since it refers to a variable subfield.

In some instructions, the instruction subfields *op* and *function* have fixed 6-bit values. These instructions use an uppercase mnemonic. For instance, in the floating-point ADD instruction, *op* = COP1 and *function* = FADD. In other cases, a single field has both fixed and variable subfields, so the name contains both uppercase and lowercase characters. The actual encodings of all the mnemonics and the codes in the function fields are shown in the instruction chapters. The operation executed by each instruction is described in pseudocode notation, as described in Table 16-13.

Table 16-13 Instruction Operation Notatio n

Symbol	Meaning
$\leftarrow$	Substitution assignment
$\|\|$	Bit string concatenation
x^y	Repetition of bit string *x* with a *y*-bit string. *x* is always a single-bit value.
$x_{y...z}$	Selection of bits *y* through *z* for bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	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 remainder
/	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-purpose register *x*. GPR (0) always reads as zero. Attempts to modify the contents of GPR (0) have no effect.
CPR[z,x]	Coprocessor unit *z*, general-purpose register *x*
CCR[*z*,*x*]	Coprocessor unit *z*, control register *x*
COC[*z*]	Coprocessor unit *z* condition signal
BigEndianMem	Endian mode as configured at reset (0 $\rightarrow$ Little, 1 $\rightarrow$ Big). Specifies the byte order of the memory interface (see LoadMemory and StoreMemory), and the byte order of Kernel and Supervisor modes. Controlled by the *BE* bit in the Configuration register, which can only be modified during reset initialization.
ReverseEndian	Signal to reverse the byte order of Load and Store instructions. This feature is available in User mode only, and is enabled by setting the *RE* bit of the Status register.

Table 16-13 Instruction Operation Notations (continued)

Symbol	**Meaning**
BigEndianCPU	Endian mode for Load and Store instructions (0 → Little, 1 → Big). In User mode, byte order can be reversed by setting the *RE* bit. The byte order is also affected by the *BE* bit in the Configuration register. BigEndianCPU is calculated as BigEndianMem XOR ReverseEndian.
LLbit	Bit showing synchronized state of instructions. Set by LL instruction, cleared by ERET instruction, and read by SC instruction.
T+*i*:	Indicates the time steps between operations. Each statement within a time step is defined to be executed in sequential order (instruction execution order may be changed by conditional branch and loop). Operations marked *T + i:* are executed at instruction cycle *i* from the start of execution of the instruction. Thus, an instruction that starts at time *j* executes operations marked T + *i:* at the time of the *i* + *j*th cycle. The order is not defined for instructions executed at the same time of operations.

The examples in Figure 16-2 illustrate the application of some of the instruction notations.

Example #1
GPR[rt] $\leftarrow$ immediate || 0^{16}
Sixteen zero bits are concatenated with an immediate value (typically 16 bits) and the 32-bit string is assigned to general-purpose register rt.
Example #2
$(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$
Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and th result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign-extended value.
Example #3
CPR[1, ft] $\leftarrow$ data
Data is assigned to general-purpose register ft of CP1 (Floating-Point General-Purpose register FGR).

Figure 16-2 Instruction Notation Examples

CPU Instruction Set

17

17.1 Introduction

This chapter describes the instruction set architecture (ISA) for the central processing unit (CPU) in the MIPS IV architecture. (For a general overview of the VR5432 MIPS IV instruction set, see Chapter 16.) The CPU architecture defines the nonprivileged instructions that execute in User mode. It does not define privileged instructions providing processor control executed by the implementation-specific system control processor. Instructions for the floating-point unit are described in Chapter 18.

17.2 Functional Instruction Groups

CPU instructions are divided into the following functional instruction groups:

- Load and Store
- Arithmetic and Logic Unit (ALU)
- Jump and Branch
- Miscellaneous
- Coprocessor

17.2.1 Load and Store Instructions

The instructions in Table 17-1 transfer data in bytes, halfwords, words, and doublewords. Signed and unsigned integers of different sizes are supported by load operations that either sign extend or zero extend the data loaded into the register. Load and Store instructions are not defined for CP0; the Move to/from coprocessor instructions provide the only way to write and read the CP0 registers.

Table 17-1 Normal CPU Load/Store Instructions

Mnemonic	Description	Defined in MIPS...
LB	Load Byte	I
LBU	Load Byte Unsigned	I
SB	Store Byte	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
SH	Store Halfword	I
LW	Load Word	I
LWU	Load Word Unsigned	III
SW	Store Word	I
LD	Load Doubleword	III
SD	Store Doubleword	III

Unaligned words and doublewords can be loaded or stored in only two instructions by using a pair of special instructions (Table 17-2). The Load instructions read the left-side or right-side bytes (left or right side of the register) from an aligned word and merge them into the correct bytes of the destination register. MIPS I, though it prohibits other use of loaded data in the load delay slot, permits LWL and LWR instructions targeting the same destination register to be executed sequentially. Store instructions select the correct bytes from a source register and update only those bytes in an aligned memory word (or doubleword).

Table 17-2 Unaligned CPU Load/Store Instructions

Mnemonic	**Description**	**Defined in MIPS...**
LWL	Load Word Left	I
LWR	Load Word Right	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III

17.2.1.1 Atomic update Load and Store instructions

Paired instructions, Load Linked and Store Conditional, can be used to perform an atomic read-modify-write access of word and doubleword cached memory locations. 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. The individual instruction descriptions describe how to use them.

Table 17-3 Atomic Update CPU Load/Store Instructions

Mnemonic	**Description**	**Defined in MIPS...**
LL	Load Linked Word	II
SC	Store Conditional Word	II
LLD	Load Linked Doubleword	III
SCD	Store Conditional Doubleword	III

17.2.2 Computational Instructions

17.2.2.1 Multiply and Divide instructions

The Multiply and Divide instructions produce twice as many result bits as is typical with other processors and they deliver their results into the HI and LO special registers. Multiply produces a full-width product twice the width of the input operands; the low half is put in LO and the high half is put in HI. Divide produces both a quotient in LO and a remainder in HI. The results are accessed by instructions that transfer data between HI/LO and the general-purpose registers.

Table 17-4 Multiply/Divide Instructions

Mnemonic	Description	Defined in MIPS...
MULT	Multiply Word	I
MULTU	Multiply Unsigned Word	I
DIV	Divide Word	I
DIVU	Divide Unsigned Word	I
DMULT	Doubleword Multiply	III
DMULTU	Doubleword Multiply Unsigned	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I

Cycle Timing for Computational Instructions

The VR5432A processor performs most computational instructions with the exception of Multiply and Divide instructions in a single processor cycle (PCycle). Multiply and Divide instructions require multiple iterations in the functional units and require multiple processor cycles to execute. Also, Divide and some Multiply instructions require the use of the MFLO and MFHI instructions to move the result back to the general register file. Since Multiply and Divide instructions can be executed in parallel with other nondependent instructions, it is desirable to schedule nondependent operations to gain performance. The VR5432A will automatically interlock the pipe when a dependency on a multicycle instruction is detected.

Table 17-5 gives the number of processor cycles (PCycles) required to execute and resolve a stall between Multiply or Divide instructions, and a subsequent dependent instruction.

Table 17-5 Multiply and Divide Instruction Latency and Repeat Rates

Instruction	**Latency[1]/Repeat Rate (Cycles)/(Cycles)**	
	Word	**Long**
DIV / DIVU / DDIV / DDIVU	42/42	74/74
MACC / MACCHI / MACCHIU / MACCU	3/1	
MSAC / MSACHI / MSACHIU / MSACU	3/1	
MUL / MULHI / MULHIU / MULU	3/1	
MULS / MULSHI / MULSHIU / MULSU	3/1	
MULT / MULTU / DMULT / DMULTU	3/1	4/2

Note:
1. Latency of the accumulator for back-to-back Multiply-accumulate instructions is 1 cycle.

17.2.2.2 ALU instructions

Some Arithmetic and Logical instructions operate on one operand from a register and the other from a 16-bit immediate value in the instruction word. The immediate operand is treated as signed for the Arithmetic and Compare instructions, and as logical (zero extended to register length) for the Logical instructions.

Table 17-6 ALU Instructions With an Immediate Op e r a n

Mnemonic	Description	Defined in MIPS...
ADDI	Add Immediate Word	I
ADDIU	Add Immediate Unsigned Word	I
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
ANDI	AND Immediate	I
ORI	OR Immediate	I
XORI	Exclusive OR Immediate	I
LUI	Load Upper Immediate	I
DADDI	Doubleword Add Immediate	III
DADDIU	Doubleword Add Immediate Unsigned	III

Table 17-7   Three-Operand ALU Instructions

Mnemonic	Description	Defined in MIPS...
ADD	Add Word	I
ADDU	Add Unsigned Word	I
SUB	Subtract Word	I
SUBU	Subtract Unsigned Word	I
DADD	Doubleword Add	III
DADDU	Doubleword Add Unsigned	III
DSUB	Doubleword Subtract	III
DSUBU	Doubleword Subtract Unsigned	III
SLT	Set on Less Than	I
SLTU	Set on Less Than Unsigned	I
AND	AND	I
OR	OR	I
XOR	Exclusive OR	I
NOR	NOR	I

17.2.2.3 Shift instructions

There are Shift instructions that take the shift amount from a 5-bit field in the instruction word and Shift instructions that take a shift amount from the low-order bits of a general-purpose register. The instructions with a fixed shift amount are limited to a 5-bit shift count, so there are separate instructions for doubleword shifts of 0–31 bits and 32–63 bits.

Table 17-8 Shift Instructions

Mnemonic	Description	Defined in MIPS...
SLL	Shift Word Left Logical	I
SRL	Shift Word Right Logical	I
SRA	Shift Word Right Arithmetic	I
SLLV	Shift Word Left Logical Variable	I
SRLV	Shift Word Right Logical Variable	I
SRAV	Shift Word Right Arithmetic Variable	I
DSLL	Doubleword Shift Left Logical	III
DSRL	Doubleword Shift Right Logical	III
DSRA	Doubleword Shift Right Arithmetic	III
DSLL32	Doubleword Shift Left Logical + 32	III
DSRL32	Doubleword Shift Right Logical + 32	III
DSRA32	Doubleword Shift Right Arithmetic + 32	III
DSLLV	Doubleword Shift Left Logical Variable	III
DSRLV	Doubleword Shift Right Logical Variable	III
DSRAV	Doubleword Shift Right Arithmetic Variable	III

17.2.3 Jump and Branch Instructions

Table 17-9 Jump Instructions Jumping Within a 256 MB Reg i o

Mnemonic	Description	Defined in MIPS...
J	Jump	I
JAL	Jump and Link	I

Table 17-10 Jump Instructions to Absolute Address

Mnemonic	Description	Defined in MIPS...
JR	Jump Register	I
JALR	Jump and Link Register	I

Table 17-11 PC-Relative Conditional Branches Comparing Two Registers

Mnemonic	Description	Defined in MIPS...
BEQ	Branch on Equal	I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II

Table 17-12 PC-Relative Conditional Branches Comparing Against Zer

Mnemonic	Description	Defined in MIPS...
BLTZ	Branch on Less Than Zero	I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero and Link	I
BGEZAL	Branch on Greater Than or Equal to Zero and Link	I
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero and Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	II

17.2.4 Miscellaneous Instructions

17.2.4.1 Exception instructions

Exception instructions cause exceptions that will transfer control to a software exception handler in the kernel. System Call and Breakpoint instructions cause exceptions unconditionally. Trap instructions cause exceptions based upon the result of a comparison.

Table 17-13 System Call and Breakpoint Instructions

Mnemonic	Description	Defined in MIPS...
SYSCALL	System Call	I
BREAK	Breakpoint	I

Table 17-14 Trap-on-Condition Instructions Comparing Two Register

Mnemonic	Description	Defined in MIPS...
TGE	Trap if Greater Than or Equal	II
TGEU	Trap if Greater Than or Equal Unsigned	II
TLT	Trap if Less Than	II
TLT	Trap if Less Than Unsigned	II
TEQ	Trap if Equal	II
TNE	Trap if Not Equal	II

Table 17-15 Trap-on-Condition Instructions Comparing an Immediate

Mnemonic	Description	Defined in MIPS...
TGEI	Trap if Greater Than or Equal Immediate	II
TGEIU	Trap if Greater Than or Equal Unsigned Immediate	II
TLTI	Trap if Less Than Immediate	II
TLTIU	Trap if Less Than Unsigned Immediate	II
TEQI	Trap if Equal Immediate	II
TNEI	Trap if Not Equal Immediate	II

17.2.4.2 Conditional Move instructions

Instructions were added in MIPS IV to move one CPU general-purpose register to another, based on the value in a third general-purpose register.

Table 17-16 CPU Conditional Move Instructio n

Mnemonic	Description	Defined in MIPS...
MOVN	Move Conditional on Not Zero	IV
MOVZ	Move Conditional on Zero	IV

17.3 System Control Coprocessor (CP0) Instructions

There are some limitations imposed on operations involving a CP0 that is incorporated within the CPU. Although Load and Store instructions to transfer data to and from coprocessors and to exchange control codes to and from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status because it has responsibility for exception handling and memory management. Therefore, the coprocessor transfer instructions are the only valid way of writing to and reading from the CP0 registers.

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

17.4 CPU Instructions

This section describes in detail each function of the CPU instructions in 32- or 64-bit mode. Exceptions that may occur are listed at the end of each instruction's description. For details regarding CPU exceptions and exception processing, refer to Chapter 6.

ADD Add ADD

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

Format:

ADD rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are added to the contents of general-purpose register *rt*. The result is stored in general-purpose register *rd*. In 64-bit mode, the operands must be sign-extended, 32-bit values.

An Integer Overflow exception occurs if the carries-out of bits 30 and 31 differ (two’s-complement overflow). The contents of destination register *rd* are not modified when an Integer Overflow exception occurs.

Operation:

```
32    T:    GPR[rd] ← GPR[rs] + GPR[rt]

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

Exceptions:

Integer Overflow exception

ADDI Add Immediate ADDI

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

Format:

ADDI rt, rs, immediate (MIPS I format)

Description:

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

An Integer Overflow exception occurs if the carries-out of bits 30 and 31 differ (two's-complement overflow). The contents of destination register *rt* are not modified when an Integer Overflow exception occurs.

Operation:

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

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

Exceptions:

Integer Overflow exception

ADDIU **Add Immediate Unsigned** ADDIU

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

Format:

ADDIU rt, rs, immediate (MIPS I format)

Description:

The 16-bit *immediate* is sign extended and added to the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt.* No Integer Overflow exception occurs under any circumstance. In 64-bit mode, the operand must be sign-extended, 32-bit values.

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

Operation:

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

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

Exceptions:

None

ADDU **Add Unsigned** ADDU

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

Format:

ADDU rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are added to the contents of general-purpose register *rt*. The result is stored in general-purpose register *rd*. No Integer Overflow exception occurs under any circumstance. In 64-bit mode, the operands must be sign-extended, 32-bit values.

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

Operation:

```
32    T:    GPR[rd] ← GPR[rs] + GPR[rt]

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

Exceptions:

None

AND AND AND

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

Format:

AND rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are bitwise ANDed with the contents of general-purpose register *rt*. The result is stored in general-purpose register *rd*.

Operation:

```
32    T:    GPR[rd] ← GPR[rs] and GPR[rt]

64    T:    GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

ANDI — AND Immediate — ANDI

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

Format:

ANDI rt, rs, immediate (MIPS I format)

Description:

The 16-bit *immediate* is zero extended and bitwise ANDed with the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt*.

Operation:

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

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

Exceptions:

None

BEQ **Branch on Equal** BEQ

31 26	25 21	20 16	15 0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format:

BEQ rs, rt, offset (MIPS I format)

Description:

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

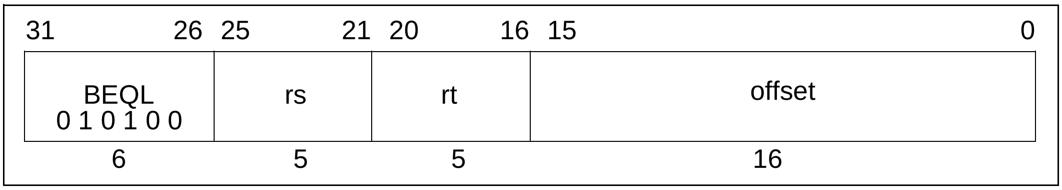
Operation:

```
32   T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
           condition ← (GPR[rs] = GPR[rt])
     T+1:  if condition then
                PC ← PC + target
           endif
64   T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
           condition ← (GPR[rs] = GPR[rt])
     T+1:  if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BEQL **Branch on Equal Likely** BEQL



Format:

BEQL rs, rt, offset (MIPS II format)

Description:

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

If it does not branch, the instruction in the delay slot is discarded.

Operation:

```
32    T:    $target \leftarrow (offset_{15})^{14} \| offset \| 0^2$
            $condition \leftarrow (GPR[rs] = GPR[rt])$
      T+1:  if condition then
                        $PC \leftarrow PC + target$
            else
                        NullifyCurrentInstruction
            endif
64    T:    $target \leftarrow (offset_{15})^{46} \| offset \| 0^2$
            $condition \leftarrow (GPR[rs] = GPR[rt])$
      T+1:  if condition then
                        $PC \leftarrow PC + target$
            else
                        NullifyCurrentInstruction
            endif
```

Exceptions:

None

BGEZ — Branch on Greater Than or Equal to Zero — BGEZ

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	BGEZ 0 0 0 0 1	offset
6	5	5	16

Format:

BGEZ rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are equal to or greater than 0, then the program branches to the branch address with a delay of one instruction.

Operation:

```
32    T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
            condition ← $(GPR[rs]_{31} = 0)$
      T+1:  if condition then
                PC ← PC + target
            endif
64    T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
            condition ← $(GPR[rs]_{63} = 0)$
      T+1:  if condition then
                PC ← PC + target
            endif
```

Exceptions:

None

BGEZAL — Branch on Greater Than or Equal to Zero and Link — BGEZAL

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	BGEZAL 1 0 0 0 1	offset
6	5	5	16

Format:

BGEZAL rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are equal to or greater than 0, then the program branches to the branch address, with a delay of one instruction.

Usually, general-purpose register *r31* should not be specified as general-purpose register *rs*, because the contents of *rs* are overwritten by storing the link address, and then it may not be re-executable. An attempt to use *r31* does not cause an exception, however.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 0)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 0)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BGEZALL — Branch on Greater Than or Equal to Zero and Link Likely — BGEZALL

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	BGEZALL 1 0 0 1 1	offset
6	5	5	16

Format:

BGEZALL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are equal to or greater than 0, then the program branches to the branch address, with a delay of one instruction. When it does not branch, the instruction in the delay slot is discarded. Usually, general-purpose register *r31* should not be specified as general-purpose register *rs*, because the contents of *rs* are overwritten by storing the link address, and then it may not be re-executable. An attempt to use *r31* does not cause an exception, however.

Operation:

```
32   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
64   T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
           condition ← $(GPR[rs]_{63} = 0)$
           GPR[31] ← PC + 8
     T+1:  if condition then
               PC ← PC + target
           else
               NullifyCurrentInstruction
           endif
```

Exceptions:

None

BGEZL — Branch on Greater Than or Equal to Zero Likely — BGEZL

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	BGEZL 0 0 0 1 1	offset
6	5	5	16

Format:

BGEZL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are equal to or greater than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the delay slot is discarded.

Operation:

```
32   T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
           condition ← $(GPR[rs]_{31} = 0)$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64   T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
           condition ← $(GPR[rs]_{63} = 0)$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

Exceptions:

None

BGTZ Branch on Greater Than Zero BGTZ

31 26	25 21	20 16	15 0
BGTZ 0 0 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZ rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are greater than 0, then the program branches to the branch address, with a delay of one instruction.

Operation:

```
32   T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
           condition ← $(GPR[rs]_{31} = 0)$ and $(GPR[rs] \neq 0^{32})$
     T+1:  if condition then
                PC ← PC + target
           endif
64   T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
           condition ← $(GPR[rs]_{63} = 0)$ and $(GPR[rs] \neq 0^{64})$
     T+1:  if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BGTZL — Branch on Greater Than Zero Likely — BGTZL

31 … 26	25 … 21	20 … 16	15 … 0
BGTZL 0 1 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are greater than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the delay slot is discarded.

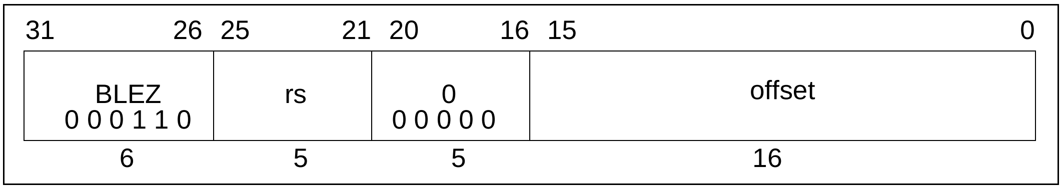
Operation:

```
32   T:   $target \leftarrow (offset_{15})^{14} \| offset \| 0^2$
          $condition \leftarrow (GPR[rs]_{31} = 0)$ and $(GPR[rs] \neq 0^{32})$
     T+1: if condition then
                PC ← PC + target
          else
                NullifyCurrentInstruction
          endif
64   T:   $target \leftarrow (offset_{15})^{46} \| offset \| 0^2$
          $condition \leftarrow (GPR[rs]_{63} = 0)$ and $(GPR[rs] \neq 0^{64})$
     T+1: if condition then
                PC ← PC + target
          else
                NullifyCurrentInstruction
          endif
```

Exceptions:

None

BLEZ Branch on Less Than or Equal to Zero BLEZ



Format:

BLEZ rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are equal to or less than 0, then the program branches to the branch address, with a delay of one instruction.

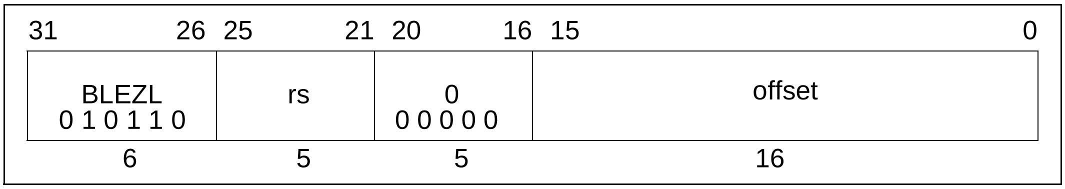
Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 1)$ or $(\text{GPR[rs]} = 0^{32})$
     T+1:  if condition then
                PC ← PC + target
           endif
64   T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 1)$ and $(\text{GPR[rs]} = 0^{64})$
     T+1:  if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BLEZL — Branch On Less Than or Equal to Zero Likely — BLEZL



Format:

BLEZL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are equal to or less than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

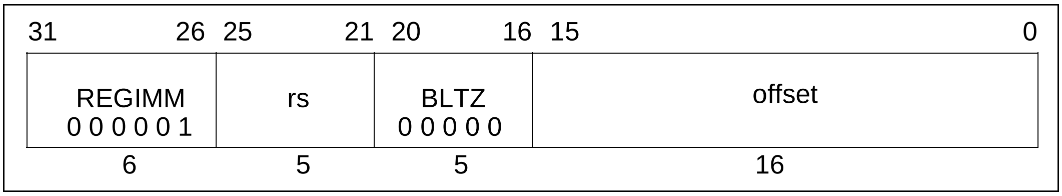
Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 1)$ or $(\text{GPR[rs]} = 0^{32})$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64   T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 1)$ and $(\text{GPR[rs]} = 0^{64})$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

Exceptions:

None

BLTZ — Branch on Less Than Zero — BLTZ



Format:

BLTZ rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. If the contents of general-purpose register *rs* are less than 0, then the program branches to the branch address, with a delay of one instruction.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 1)$
     T+1:  if condition then
                   PC ← PC + target
           endif
64   T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 1)$
     T+1:  if condition then
                   PC ← PC + target
           endif
```

Exceptions:

None

BLTZAL — Branch on Less Than Zero and Link — BLTZAL

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM 0 0 0 0 0 1	rs	BLTZAL 1 0 0 0 0	offset
6	5	5	16

Format:

BLTZAL rs, offset (MIPS I format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are less than 0, then the program branches to the branch address, with a delay of one instruction.

Usually, general-purpose register *r31* should not be specified as general-purpose register *rs*, because the contents of *rs* are overwritten by storing the link address, and then it is not re-executable. An attempt to use *r31* does not generate an exception, however.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 1)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                 PC ← PC + target
            endif
64    T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 1)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                 PC ← PC + target
            endif
```

Exceptions:

None

BLTZALL — Branch on Less Than Zero and Link Likely — BLTZALL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZALL 1 0 0 1 0	offset
6	5	5	16

Format:

BLTZALL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended.
Unconditionally, the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are smaller than 0, then the program branches to the branch address, with a delay of one instruction.
If it does not branch, the instruction in the branch delay slot is discarded.
Usually, general-purpose register *r31* should not be specified as general-purpose register *rs*, because the contents of *rs* are overwritten by storing the link address, and then it is not re-executable. An attempt to use *r31* does not cause an exception, however.

Operation:

```
32   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
          else
              NullifyCurrentInstruction
          endif
64   T:   target ← $(offset_{15})^{46}$ || offset || $0^2$
          condition ← $(GPR[rs]_{63} = 1)$
          GPR[31] ← PC + 8
     T+1: if condition then
              PC ← PC + target
          else
              NullifyCurrentInstruction
          endif
```

Exceptions:

None

BLTZL — Branch on Less Than Zero Likely — BLTZL

31 — 26	25 — 21	20 — 16	15 — 0
REGIMM 0 0 0 0 0 1	rs	BLTZL 0 0 0 1 0	offset
6	5	5	16

Format:

BLTZL rs, offset (MIPS II format)

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign extended. Unconditionally, the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are less than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 1)$
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
64    T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 1)$
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
```

Exceptions:

None

BNE — Branch on Not Equal — BNE

31 … 26	25 … 21	20 … 16	15 … 0
BNE 0 0 0 1 0 1	rs	rt	offset
6	5	5	16

Format:

BNE rs, rt, offset (MIPS I format)

Description:

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

Operation:

```
32    T:     $target \leftarrow (offset_{15})^{14} \| offset \| 0^2$
             $condition \leftarrow (GPR[rs] \neq GPR[rt])$
      T+1:   if condition then
                  $PC \leftarrow PC + target$
             endif
64    T: $target \leftarrow (offset_{15})^{46} \| offset \| 0^2$
             $condition \leftarrow (GPR[rs] \neq GPR[rt])$
      T+1: if condition then
                  $PC \leftarrow PC + target$
             endif
```

Exceptions:

None

BNEL Branch on Not Equal Likely BNEL

31 26	25 21	20 16	15 0
BNEL 0 1 0 1 0 1	rs	rt	offset
6	5	5	16

Format:

BNEL rs, rt, offset (MIPS II format)

Description:

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

If it does not branch, the instruction in the branch delay slot is discarded.

Operation:

```
32    T:    target $\leftarrow$ $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition $\leftarrow$ (GPR[rs] $\neq$ GPR[rt])
      T+1:  if condition then
                  PC $\leftarrow$ PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target $\leftarrow$ $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition $\leftarrow$ (GPR[rs] $\neq$ GPR[rt])
      T+1:  if condition then
                  PC $\leftarrow$ PC + target
            else
                  NullifyCurrentInstruction
            endif
```

Exceptions:

None

BREAK **Breakpoint** BREAK

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	code	BREAK 0 0 1 1 0 1
6	20	6

Format:

BREAK (MIPS I format)

Description:

A Breakpoint exception occurs after execution of this instruction, transferring control to the exception handler.

The code field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64  T: BreakpointException
```

Exceptions:

Breakpoint exception

CACHE **Cache Operation** CACHE

31 26	25 21	20 16	15 0
CACHE 1 0 1 1 1 1	base	op	offset
6	5	5	16

Format:

CACHE op, offset (*base*) (MIPS III format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

The index operation uses part of the virtual address to specify a cache block.

For a cache of 32 KB with 32 bytes per tag, $vAddr_{13:5}$ specifies the block. Bit 0 of the virtual address is used to specify the associativity.

Index Load Tag uses $vAddr_{LINEBITS... 3}$ to select the doubleword for reading parity. When the *CE* bit of the Status register is set, Hit WriteBack, Hit WriteBack Invalidate, Index WriteBack Invalidate, and Fill also use $vAddr_{LINEBITS ... 3}$ to select the doubleword that has its parity modified. This operation is performed unconditionally.

The hit operation accesses the specified cache as normal data references and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.

During a write-back operation, modified data in the cache (i.e., “dirty” data) is written to main memory. The address to be written is specified by the cache tag and not the translated physical address.

CACHE — Cache Operation (continued) — CACHE

Bits 17...16 of the instruction specify the cache as follows:

Code	Name	Cache
00	I	Instruction
01	D	Data
10	—	Reserved
11	—	Reserved

Bits 20 to 18 (this value is listed under the Code column) of the instruction specify the operation as follows:

Code	Cache	Name	Operation
000	I	Index Invalidate	Set the cache state of the cache block to Invalid and Unlocked.
000	D	Index WriteBack Invalidate	Examine the cache state of the data cache block at the index specified by the virtual address. If the state is Dirty and not Invalid, writes the block back to memory. The address to write is taken from the cache tag. Set the cache state of the cache block to Invalid. May be used to unlock a cache block.

Cache Operation (continued)

Code	Cache	Name	Operation
001	All	Index Load Tag	Reads the tag for the cache block at the specified index and places it into the TagLo and TagHi CP0 registers, ignoring any parity errors. In addition, the data parity from the specified doubleword is loaded into the PErr register.
010	I, D	Index Store Tag	Write the tag for the cache block at the specified index from the TagLo and TagHi CP0 registers, including the parity bit (*P*) from the TagLo register.
011	D	Create Dirty	This operation is used to avoid loading data needlessly from memory when writing new contents into an entire cache block. If the cache block does not contain the specified address and the block is dirty, write it back to memory. In all cases, set the cache block tag to the specified physical address and set the cache state to Dirty.

CACHE — Cache Operation (continued) — CACHE

Code	Cache	Name	Operation
100	I, D	Hit Invalidate	If the cache block contains the specified address, mark the cache block Invalid.
101	D	Hit WriteBack Invalidate	If the cache block contains the specified address, write the data back if it is dirty. In all cases, mark the cache block Invalid.
101	I	Fill	Fill the instruction cache block from memory.
110	D	Hit WriteBack	If the cache block contains the specified address and its state is Dirty, write back the data and clear the state to not Dirty.
111	D	Fetch and Lock	This operation is used to lock a cache block. If the cache block does not contain the specified address, fill it from memory, writing the original block back to memory using the tag address if the block was dirty. In all cases, set the cache block tag to the specified physical address and set the cache state to Locked.
111	I	Fetch and Lock	This operation is used to lock a cache block. If the cache block does not contain the specified address, fill it from memory. In all cases, set the cache block tag to the specified physical address and set the cache state to Locked.

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag), unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified exceptions.

If CP0 is not enabled (i.e., the CP0 enable bit in the Status register is clear in User or Supervisor mode) and this instruction is executed, a Coprocessor Unusable exception is taken. The operation of this instruction on any operation/cache combination not listed in the table is undefined. The operation of this instruction on uncached addresses is also undefined.

The processor only fills the I-cache line using the cache instruction “Fill” when the data is not stored in the cache.

Cache Operation (continued)

Operation:

```
32, 64T:   vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15...0})$ + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           CacheOp (op, vAddr, pAddr)
```

Exceptions:

Coprocessor Unusable exception

CFC1 — Move Control Word from FPU (Coprocessor 1) — CFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	CF 0 0 0 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CFC1 rt, fs (MIPS I format)

Description:

The contents of the floating-point control register *fs* are loaded into general-purpose register *rt*, with sign extension if the destination register is 64 bits.

This instruction is only defined when *fs* equals 0 or 31.

For MIPS I, MIPS II, and MIPS III, the contents of general-purpose register *rt* are undefined while the instruction immediately following this Load instruction is being executed.

Operation:

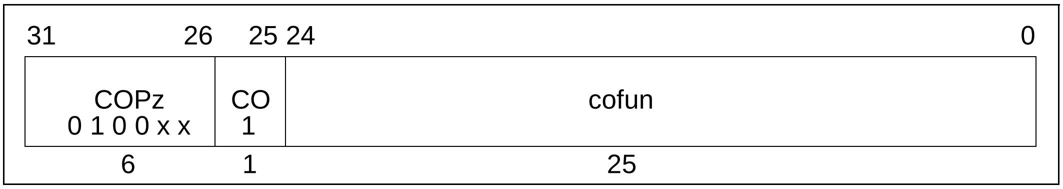
```
32    T: temp ← FCR[fs]
      T+1: GPR[rt] ← temp

64    T: temp ← FCR[fs]
      T+1: GPR[rt] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

Coprocessor Unusable exception

COPz Coprocessor z Operation COPz



Format:

COPz cofun (MIPS I format)

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 states within the processor, cache, or main memory.

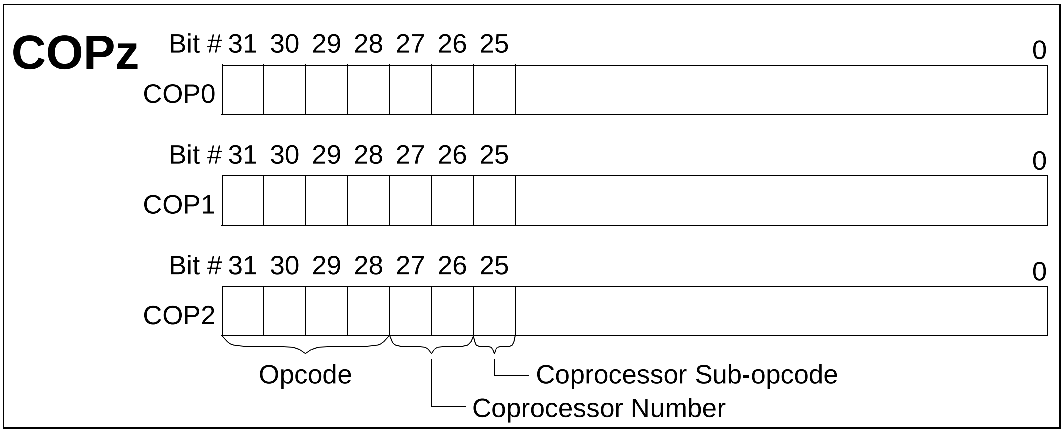
Operation:

```
32, 64T:     CoprocessorOperation (z, cofun)
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception (CP1 only)

Opcode Bit Encoding:


COPz
Bit # 31 30 29 28 27 26 25
0
COP0
Bit # 31 30 29 28 27 26 25
0
COP1
Bit # 31 30 29 28 27 26 25
0
COP2
Opcode
Coprocessor Sub-opcode
Coprocessor Number

CTC1 — Move Control Word to FPU (Coprocessor 1) — CTC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	CT 0 0 1 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CTC1 rt, fs (MIPS I format)

Description:

The contents of general-purpose register *rt* are stored in floating-point control register *fs*. This instruction is defined only if *fs* is 0 or 31.

If any cause bit of the Floating-Point Control/Status register (FCR31) and its corresponding enable bit are set by writing data to FCR31, the Floating-Point exception occurs. The data is written to the register before the exception occurs.

For MIPS I, MIPS II, and MIPS III, the contents of the Floating-Point Control register *fs* are undefined while the instruction immediately following this instruction is executed.

Operation:

```
32    T:      temp ← GPR[rt]
      T+1:    FCR[fs] ← temp
              COC[1] ← FCR[31]$_{23}$
64    T:      temp ← GPR[rt]$_{31...0}$
      T+1:    FCR[fs] ← temp
              COC[1] ← FCR[31]$_{23}$
```

CTC1 Move Control Word to FPU (Coprocessor 1) (continued) CTC1

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Division by Zero exception
Inexact Operation exception
Overflow exception
Underflow exception

DADD — Doubleword Add — DADD

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

Format:

DADD rd, rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rs* and the contents of general-purpose register *rt* are added, and the result is stored in general-purpose register *rd*. An Integer Overflow exception occurs if the carries-out of bits 62 and 63 differ (two's-complement overflow). The contents of the destination register *rd* are not modified when an Integer Overflow exception occurs.

This operation is only defined for 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64    T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Integer Overflow exception
Reserved Instruction exception

DADDI **Doubleword Add Immediate** DADDI

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

Format:

DADDI rt, rs, immediate (MIPS III format)

Description:

The 16-bit *immediate* is sign extended and added to the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt*. An Integer Overflow exception occurs if the carries-out of bits 62 and 63 differ (two's-complement overflow). The contents of the destination register *rt* are not modified when an Integer Overflow exception occurs.

This operation is only defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Integer Overflow exception
Reserved Instruction exception

DADDIU — Doubleword Add Immediate Unsigned — DADDIU

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

Format:

DADDIU rt, rs, immediate (MIPS III format)

Description:

The 16-bit *immediate* is sign extended and added to the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt*.

This operation is only defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

The only difference between this instruction and the DADDI instruction is that the DADDIU instruction never causes an Integer Overflow exception.

Operation:

64	$\text{T: GPR [rt]} \leftarrow \text{GPR[rs]} + (\text{immediate}_{15})^{48} \,\|\, \text{immediate}_{15...0}$

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DADDU Doubleword Add Unsigned DADDU

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

Format:

DADDU rd, rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rs* and the contents of general-purpose register *rt* are added, and the result is stored in general-purpose register *rd*.

This operation is only defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

The only difference between this instruction and the DADD instruction is that the DADDU instruction never causes an Integer Overflow exception.

Operation:

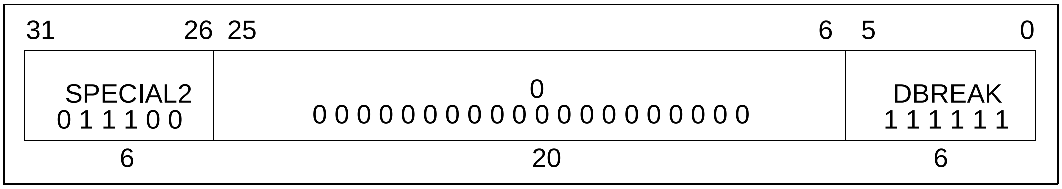
```
64    T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Note: Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DBREAK **Debug Break** DBREAK



Format:

DBREAK (VR5432 format)

Description:

The DBREAK instruction forces entry into Debug mode by causing a trap to the Debug Exception vector address (0xFFFF FFFF BFC0 1000). This instruction may only be executed in User, Supervisor, or Kernel mode. Execution in Debug mode results in undefined behavior.

Execution transitions to Debug mode at an instruction boundary, the program counter (PC) is saved in the DEPC register, and execution is redirected to the 64-bit Debug Exception vector (location 0xFFFF FFFF BFC0 1000).

Before the processor enters Debug mode, all instructions are flushed from the pipeline and all outstanding external bus transactions are completed. There may be a delay entering Debug mode to allow the pipeline flush and to allow all outstanding external transactions to complete. The processor stalls during this time.

The processor will not enter Debug mode at a branch delay slot instruction boundary. Instead, it stops either at the Branch instruction or the target of the branch. If a software or hardware breakpoint occurs for the branch delay slot instruction, the breakpoint occurs at the corresponding Branch instruction. If a single-step break is executed on a Branch instruction, both the branch and its delay slot are executed.

If the *DME* bit in the Status register is not set, a Reserved Instruction exception will occur when DBREAK is issued.

DBREAK Debug Break DBREAK

(Continued)

Operation:

```
32, 64   T:   DBreakOperation ()
```

Exceptions:

Reserved Instruction exception

DDIV Doubleword Divide DDIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DDIV 0 1 1 1 1 0
6	5	5	10	6

Format:

DDIV rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt*, treating both operands as signed integers. An Integer Overflow exception never occurs, and the result of this operation is undefined when the divisor is zero.

This instruction is usually executed after additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the doubleword result is stored in special register LO, and the remainder word of the doubleword result is stored in special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain correct results, insert two or more additional instructions between MFHI or MFLO and the DDIV instruction.

This operation is only defined in 64-bit mode and 32-bit Kernel mode. Execution in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64      T–2:    LO      ← undefined
                HI      ← undefined
        T–1:    LO      ← undefined
                HI      ← undefined
        T:      LO      ← GPR[rs] div GPR[rt]
                HI      ← GPR[rs] mod GPR[rt]
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DDIVU — Doubleword Divide Unsigned — DDIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DDIVU 0 1 1 1 1 1
6	5	5	10	6

Format:

DDIVU rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt*, treating both operands as unsigned integers. An Integer Overflow exception never occurs, and the result of this operation is undefined when the divisor is zero.

This instruction is usually executed after instructions to check for a zero divisor.

When the operation completes, the quotient (doubleword) is stored into special register LO and the remainder (doubleword) is stored into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain correct results, insert two or more instructions between MFHI or MFLO and the DDIVU instruction.

This operation is only defined for the VR5432 operating in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DIV **Divide** DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIV 0 1 1 0 1 0
6	5	5	10	6

Format:

DIV rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt*, treating both operands as signed integers. An Overflow exception never occurs, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the result must be sign-extended, 32-bit values.

This instruction is usually executed after instructions to check for a zero divisor and for overflow

When the operation completes, the quotient (doubleword) is stored into special register LO and the remainder (doubleword) is stored into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain correct results, insert two or more additional instructions between MFHI or MFLO and the DIV instructions.

DIV Divide (continued) DIV

Operation:

```
32    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      LO    ← GPR[rs] div GPR[rt]
              HI    ← GPR[rs] mod GPR[rt]
64    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      q     ← $GPR[rs]_{31...0}$ div $GPR[rt]_{31...0}$
              r     ← $GPR[rs]_{31...0}$ mod $GPR[rt]_{31...0}$
              LO    ← $(q_{31})^{32} || q_{31...0}$
              HI    ← $(r_{31})^{32} || r_{31...0}$
```

Exceptions:

None

DIVU **Divide Unsigned** DIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIVU 0 1 1 0 1 1
6	5	5	10	6

Format:

DIVU rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt*, treating both operands as unsigned integers. An Integer Overflow exception never occurs, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the result must be sign-extended, 32-bit values.

This instruction is usually executed after instructions to check for a zero divisor.

When the operation completes, the quotient (doubleword) is stored into special register LO and the remainder (doubleword) is stored into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain correct results, insert two or more additional instructions between MFHI or MFLO and the DIVU instruction.

DIVU **Divide Unsigned (continued)** DIVU

Operation:

```
32   T–2:   LO   ← undefined
            HI   ← undefined
     T–1:   LO   ← undefined
            HI   ← undefined
     T:     LO   ← (0 || GPR[rs]) div (0 || GPR[rt])
            HI   ← (0 || GPR[rs]) mod (0 || GPR[rt])
64   T–2:   LO   ← undefined
            HI   ← undefined
     T–1:   LO   ← undefined
            HI   ← undefined
     T:     q    ← (0 || $GPR[rs]_{31...0}$) div (0 || $GPR[rt]_{31...0}$)
            r    ← (0 || $GPR[rs]_{31...0}$) mod (0 || $GPR[rt]_{31...0}$)
            LO   ← $(q_{31})^{32}$ || $q_{31...0}$
            HI   ← $(r_{31})^{32}$ || $r_{31...0}$
```

Exceptions:

None

DMFC0 — Doubleword Move from System Control Coprocessor — DMFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	DMF 0 0 0 0 1	rt	rd	0 0 0 0 0 0 0 0 0 0 0 00
6	5	5	5	11

Format:

DMFC0 rt, rd (MIPS III format)

Description:

The contents of coprocessor register *rd* of CP0 are stored in general-purpose register *rt.*

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

The contents of the source coprocessor register *rd* are written to the 64-bit destination general-purpose register *rt*. The operation of a DMFC0 instruction on a 32-bit register of CP0 is undefined.

Operation:

```
64          T:      data ←CPR[0,rd]
            T+1:    GPR[rt] ← data
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception (64-/32-bit User mode and Supervisor mode if CP0 is disabled)

Reserved Instruction exception (32-bit User or Supervisor mode)

DMTC0 — Doubleword Move to System Control Coprocessor — DMTC0

31 26	25 21	20 16	15	0
COP0 010000	DMT 00101	rt	rd	0 00000000000
6	5	5		16

Format:

DMTC0 rt, rd (MIPS III format)

Description:

The contents of general-purpose register *rt* are loaded into coprocessor register *rd* of CP0.

This operation is defined in 64-bit mode or in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

The contents of the source general-purpose register *rd* are written to the 64-bit destination coprocessor register *rt*. The operation of a DMTC0 instruction on a 32-bit register of CP0 is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of Load instructions, Store instructions, and TLB operations for the instructions immediately before and after this instruction are undefined.

Operation:

```
64        T:     data ← GPR[rt]
          T+1:   CPR[0, rd] ← data
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception (64-/32-bit User and Supervisor mode if CP0 is disabled)

Reserved Instruction exception (32-bit User or Supervisor mode)

DMULT **Doubleword Multiply** DMULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMULT 0 1 1 1 0 0
6	5	5	10	6

Format:

DMULT rs, rt (MIPS III format)

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as signed integers. An Integer Overflow exception never occurs.

When the operation completes, the low-order doubleword is stored into special register LO and the high-order doubleword is stored into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the DMULT instruction.

This operation is only defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64    T–2:  LO      ← undefined
            HI      ← undefined
      T–1:  LO      ← undefined
            HI      ← undefined
      T:    t       ← GPR[rs] * GPR[rt]
            LO      ← $t_{63...0}$
            H I     ← $t_{127...64}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DMULTU **Doubleword Multiply Unsigned** DMULTU

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMULTU 0 1 1 1 0 1
6	5	5	10	6

Format:

DMULTU rs, rt (MIPS III format)

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as unsigned integers. An Overflow exception never occurs.

When the operation completes, the low-order doubleword is stored into special register LO, and the high-order doubleword is stored into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the DMULTU instruction.

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← (0 || GPR[rs]) * (0 || GPR[rt])
             LO ← $t_{63...0}$
             HI ← $t_{127...64}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DRET **Debug Return** DRET

31 … 26	25 … 6	5 … 0
SPECIAL2 0 1 1 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DRET 1 1 1 1 1 0
6	20	6

Format:

DRET (VR5432 format)

Description:

The DRET instruction returns from Debug mode to the mode in effect (User, Supervisor, or Kernel mode) when the last debug break event has occurred. Control is passed to the instruction pointed to by the Debug Exception PC (DEPC) register. Unlike most jumps and branches, the execution of which also executes the next instruction (the one in the delay slot), DRET does not execute a delay slot instruction. The DRET instruction must not be placed in a branch delay slot.

Operation:

```
32, 64   T:   DRetOperation ()
```

Exceptions:

None

DROR — Doubleword Rotate Right — DROR

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

Format:

DROR rd, rt, sa (VR5432 format)

Description:

The contents of general-purpose register *rt* are rotated right by *sa* bits, and the result is stored in general-purpose register *rd*.

Operation:

```
32, 64 T:    GPR[rd] ← GPR[rt]$_{sa-1...0}$ || GPR[rt]$_{63...sa}$
```

Exceptions:

None

DROR32 — Doubleword Rotate Right Plus 32 — DROR32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	1 0 0 0 0 1	rt	rd	sa	DROR32 1 1 1 1 1 0
6	5	5	5	5	6

Format:

DROR32 rd, rt, sa (VR5432 format)

Description:

The contents of general-purpose register *rt* are rotated right by *sa* + 32 bits, and the result is stored in general-purpose register *rd*.

Operation:

```
32, 64 T:    $s = sa + 32$

             $GPR[rd] \leftarrow GPR[rt]_{s-1...0} \| GPR[rt]_{63...s}$
```

Exceptions:

None

DRORV — Doubleword Rotate Right Variable — DRORV

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

Format:

DRORV rd, rt, rs (VR5432 format)

Description:

The contents of general-purpose register *rt* are rotated right by the number of bits specified by the low-order five bits of general-purpose register *rs*. The result is stored in general-purpose register *rd*.

Operation:

```
32, 64 T:   $s \leftarrow GPR[rs]_{4...0}$
            $GPR[rd] \leftarrow GPR[rt]_{s-1...0} \| GPR[rt]_{63...s}$
```

Exceptions:

None

DSLL Doubleword Shift Left Logical DSLL

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

Format:

DSLL rd, rt, sa (MIPS III format)

Description:

The contents of general-purpose register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSLLV — Doubleword Shift Left Logical Variable — DSLLV

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

Format:

DSLLV rd, rt, rs (MIPS III format)

Description:

The contents of general-purpose register *rt* are shifted left by the number of bits specified by the low-order six bits contained in general-purpose register *rs*, inserting zeros into the low-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSLL32 — Doubleword Shift Left Logical Plus 32 — DSLL32

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSLL32 1 1 1 1 0 0
6	5	5	5	5	6

Format:

DSLL32 rd, rt, sa (MIPS III format)

Description:

The contents of general-purpose register *rt* are shifted left by 32 + *sa* bits, inserting zeros into the low-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64        T:    $s \leftarrow 1 \| sa$
                $GPR[rd] \leftarrow GPR[rt]_{(63-s)...0} \| 0^{s}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRA — Doubleword Shift Right Arithmetic — DSRA

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

Format:

DSRA rd, rt, sa (MIPS III format)

Description:

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

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64      T:   $s \leftarrow 0 \,||\, sa$
             $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63...s}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRAV — Doubleword Shift Right Arithmetic Variable — DSRAV

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

Format:

DSRAV rd, rt, rs (MIPS III format)

Description:

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

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRA32 **Doubleword Shift Right Arithmetic Plus 32** DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRA32 1 1 1 1 1 1
6	5	5	5	5	6

Format:

DSRA32 rd, rt, sa (MIPS III format)

Description:

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

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64      T:   $s \leftarrow 1 \,||\, sa$
             $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63...s}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRL — Doubleword Shift Right Logical — DSRL

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL 1 1 1 0 1 0
6	5	5	5	5	6

Format:

DSRL rd, rt, sa (MIPS III format)

Description:

The contents of general-purpose register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and in 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRLV Doubleword Shift Right Logical Variable DSRLV

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

Format:

DSRLV rd, rt, rs (MIPS III format)

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the low-order six bits of general-purpose register *rs,* inserting zeros into the high-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSRL32 **Doubleword Shift Right Logical Plus 32** DSRL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL32 1 1 1 1 1 0
6	5	5	5	5	6

Format:

DSRL32 rd, rt, sa

Description:

The contents of general-purpose register *rt* are shifted right by 32 + *sa* bits, inserting zeros into the high-order bits. The result is stored in general-purpose register *rd*.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64          T:     $s \leftarrow 1 \| sa$
                   $GPR[rd] \leftarrow 0^{s} \| GPR[rt]_{63...s}$
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

DSUB **Doubleword Subtract** DSUB

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

Format:

DSUB rd, rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*, and the result is stored in general-purpose register *rd*.

An Integer Overflow exception takes place if the carries-out of bits 62 and 63 differ (a two’s-complement overflow). The contents of destination register *rd* are not modified when an Integer Overflow exception occurs.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64        T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Integer Overflow exception
Reserved Instruction exception

DSUBU Doubleword Subtract Unsigned DSUBU

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

Format:

DSUBU rd, rs, rt (MIPS III format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*, and the result is stored in general-purpose register *rd*.

The only difference between this instruction and the DSUB instruction is that the DSUBU instruction never causes an Integer Overflow exception.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
64          T:     GPR[rd] ← GPR[rs] – GPR[rt]
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Reserved Instruction exception

ERET Return from Exception ERET

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ERET 0 1 1 0 0 0
6	1	19	6

Format:

ERET (MIPS III format)

Description:

ERET is for returning from an interrupt, exception, or error exception. Unlike a Branch or Jump instruction, ERET does not execute a delay slot instruction.

The ERET instruction must not itself be placed in a branch delay slot.

If the *ERL* bit of the Status register is set (SR_2 = 1), load the contents of the ErrorEPC register to the PC and clear the *ERL* bit to zero. Otherwise (SR_2 = 0), load the PC from the *EPC*, and clear the *EXL* bit of the Status register to zero (SR_1 = 0).

An ERET instruction executed between an LL instruction and an SC instruction causes the SC instruction to fail, since the ERET instruction clears the *LL* bit to zero.

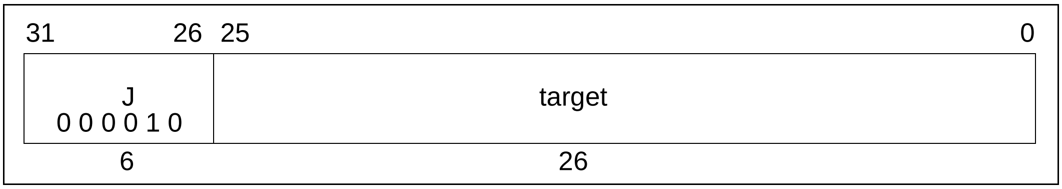
Operation:

```
32, 64    T:  if $SR_2 = 1$ then
                  PC $\leftarrow$ ErrorEPC
                  SR $\leftarrow SR_{31...3} \| 0 \| SR_{1...0}$
              else
                  PC $\leftarrow$ EPC
                  SR $\leftarrow SR_{31...2} \| 0 \| SR_0$
              endif
              LLbit $\leftarrow$ 0
```

Exceptions:

Coprocessor Unusable exception

J Jump J



Format:

J target (MIPS I format)

Description:

The 26-bit target is shifted left two bits and combined with the high-order four bits of the address of the delay slot to calculate the target address. The program unconditionally jumps to this calculated address with a delay of one instruction.

Because instructions must be word aligned, a J instruction must specify an address where the low-order two bits are zero. If these low-order two bits are not zero, an Address Error exception will occur when the Jump target instruction is fetched.

Operation:

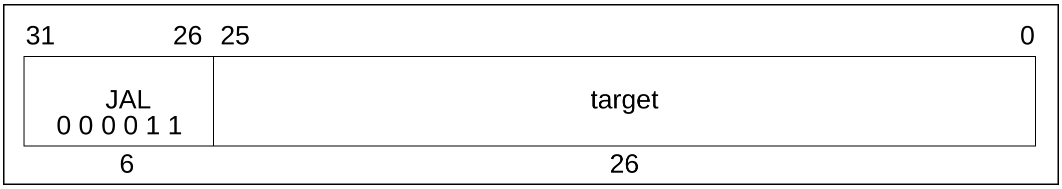
```
32        T:     $temp \leftarrow target$
          T+1:   $PC \leftarrow PC_{31...28} \,||\, temp \,||\, 0^2$

64        T:     $temp \leftarrow target$
          T+1:   $PC \leftarrow PC_{63...28} \,||\, temp \,||\, 0^2$
```

Exceptions:

Address Error exception

JAL Jump and Link JAL



Format:

JAL target (MIPS I format)

Description:

The 26-bit target is shifted left two bits and combined with the high-order four bits of the address of the delay slot to calculate the address. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.

Because instructions must be word aligned, a JAL instruction must specify an address where the low-order two bits are zero. If these low-order two bits are not zero, an Address Error exception will occur when the Jump target instruction is fetched.

Operation:

```
32    T:    $temp \leftarrow target$
            $GPR[31] \leftarrow PC + 8$
      T+1:  $PC \leftarrow PC_{31...28} || temp || 0^2$
64    T:    $temp \leftarrow target$
            $GPR[31] \leftarrow PC + 8$
      T+1:  $PC \leftarrow PC_{63...28} || temp || 0^2$
```

Exceptions:

Address Error exception

JALR — Jump and Link Register — JALR

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

Format:

JALR rs	(MIPS I format, rd = 31 implied)
JALR rd, rs	(MIPS I format)

Description:

The program unconditionally jumps to the address contained in general-purpose register *rs*, with a delay of one instruction. The address of the instruction after the delay slot is stored in general-purpose register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register numbers *rs* and *rd* should not be equal, because such an instruction does not have the same effect when re-executed. If they are equal, the contents of *rs* are destroyed by storing a link address. However, if an attempt is made to execute this instruction, an exception will not occur, and the result of executing such an instruction is undefined.

Because instructions must be word aligned, a JALR instruction must specify a target register (*rs*) that contains an address where the low-order two bits are zero. If these low-order two bits are not zero, an Address Error exception will occur when the Jump target instruction is fetched.

Operation:

```
32, 64    T:        temp ← GPR [rs]
                    GPR[rd] ← PC + 8
          T+1:      PC ← temp
```

Exceptions:

Address Error exception

JR Jump Register JR

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	JR 0 0 1 0 0 0
6	5	15	6

Format:

JR rs (MIPS I format)

Description:

The program unconditionally jumps to the address contained in general-purpose register *rs*, with a delay of one instruction.

Because instructions must be word aligned, a JR instruction must specify a target register (*rs*) that contains an address where the low-order two bits are zero. If these low-order two bits are not zero, an Address Error exception will occur when the Jump target instruction is fetched.

Operation:

```
32, 64    T:        temp ← GPR[rs]
          T+1:      PC ← temp
```

Exceptions:

Address Error exception

LB — Load Byte — LB

31 … 26	25 … 21	20 … 16	15 … 0
LB 1 0 0 0 0 0	base	rt	offset
6	5	5	16

Format:

LB rt, offset (*base*) (MIPS I format)

Description:

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

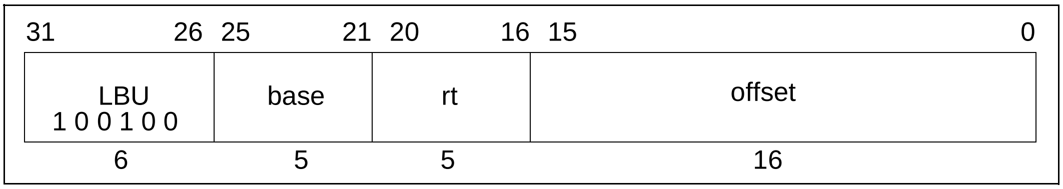
Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $ReverseEndian^3)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}$ xor $BigEndianCPU^3$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{24} \| mem_{7+8*byte...8*byte}$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $ReverseEndian^3)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}$ xor $BigEndianCPU^3$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{56} \| mem_{7+8*byte...8*byte}$
```

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LBU Load Byte Unsigned LBU



Format:

LBU rt, offset(base) (MIPS I format)

Description:

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

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1 ... 3} \| (pAddr_{2...0}$ xor $ReverseEndian^3)$
            $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
            $byte \leftarrow vAddr_{2...0}$ xor $BigEndianCPU^3$
            $GPR[rt] \leftarrow 0^{24} \| mem_{7+8*byte...8*byte}$
64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $ReverseEndian^3)$
            $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
            $byte \leftarrow vAddr_{2...0}$ xor $BigEndianCPU^3$
            $GPR[rt] \leftarrow 0^{56} \| mem_{7+8*byte...8*byte}$
```

Exceptions:

TLB Miss exception	TLB Invalid exception
Bus Error exception	Address Error exception

LD — Load Doubleword — LD

31 … 26	25 … 21	20 … 16	15 … 0
LD 1 1 0 1 1 1	base	rt	offset
6	5	5	16

Format:

LD rt, offset (base) (MIPS III format)

Description:

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

If any of the low-order three bits of the address are not zero, an Address Error exception occurs.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

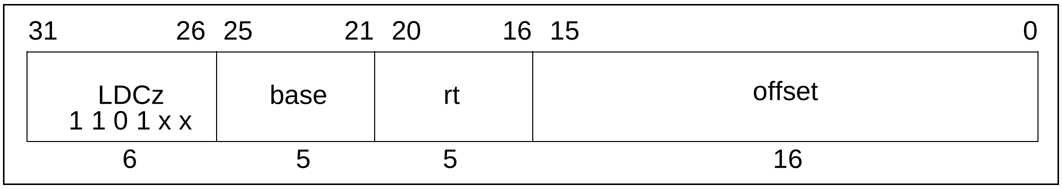
```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15...0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

LDCz Load Doubleword to Coprocessor z LDCz



Format:

LDCz rt, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The processor loads a doubleword from the addressed memory location to CPz. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If any of the low-order three bits of the address are not zero, an Address Error exception takes place.

This instruction is not valid for use with CP0.

When CP1 is specified, the *FR* bit of the Status register equals zero and the least-significant bit in the *rt* field is not zero; the operation of the instruction is undefined. If the *FR* bit equals one, an odd or even register is specified by *rt*.

LDCz — Load Doubleword to Coprocessor z (continued) — LDCz

Operation:

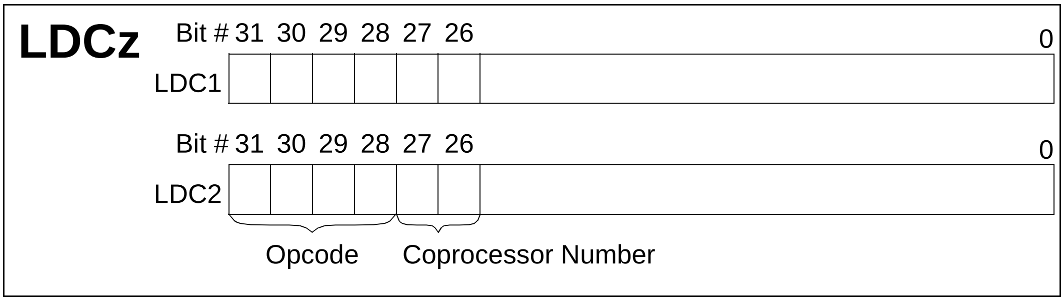
```
32    T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15...0}$) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)

64    T:   vAddr ← ((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 Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Coprocessor Unusable exception

Opcode Bit Encoding:



LDL **Load Doubleword Left** LDL

31 26	25 21	20 16	15 0
LDL 0 1 1 0 1 0	base	rt	offset
6	5	5	16

Format:

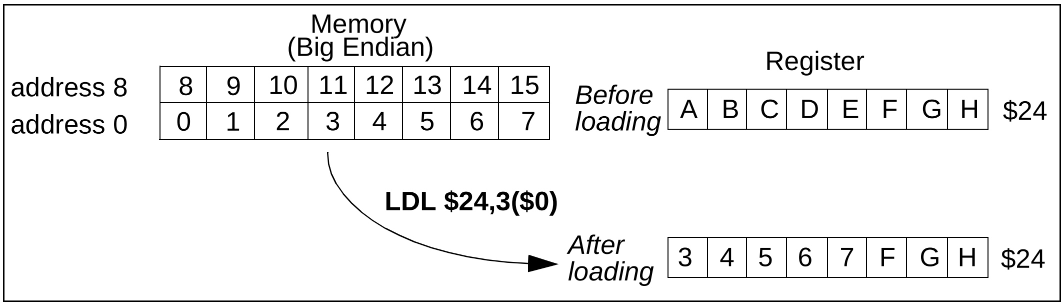
LDL rt, offset (base) (MIPS III format)

Description:

This instruction is used in combination with the LDR instruction to load the doubleword data in the memory that is not at the word boundary to general-purpose register *rt*. The LDL instruction loads the higher portion of the data to the register, while the LDR instruction loads the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address that can specify any byte. Of the doubleword data in the memory where the most-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the higher portion of general-purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 8.

In other words, first the addressed byte is stored to the most-significant byte position of general-purpose register *rt*. If there is data of the low-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of general-purpose register *rt* is repeated. The remaining low-order byte is not affected.



LDL　　Load Doubleword Left (continued)　　LDL

The contents of general-purpose register *rt* are internally bypassed within the processor, so that no NOP instruction is needed between an immediately preceding Load instruction that targets general-purpose register *rt* and a subsequent LDL (or LDR) instruction.

The Address Error exception does not occur even if the specified address is not at the doubleword boundary.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

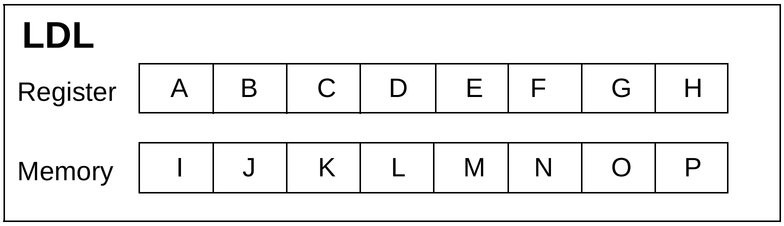
Operation:

```
64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1...3}$ || (pAddr$_{2...0}$ xor ReverseEndian$^3$)
            if BigEndianMem = 0 then
                        pAddr ← pAddr$_{PSIZE-1...3}$ || 0$^3$
            endif
            byte ← vAddr$_{2...0}$ xor BigEndianCPU$^3$
            mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
            GPR[rt] ← mem$_{7+8*byte...0}$ || GPR[rt]$_{55-8*byte...0}$
```

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LDL — Load Doubleword Left (continued) — LDL

The relationship between the address given to the LDL instruction and the result (bytes for registers) is shown below:



vAddr$_{2...0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G H	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

Note: *Type*: Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a doubleword)

Offset: pAddr$_{2...0}$ Output to memory

LEM	Little-endian memory (BigEndianMem = 0)
BEM	Big-endian memory (BigEndianMem = 1)

LDL **Load Doubleword Left (continued)** LDL

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

LDR **Load Doubleword Right** LDR

31 26	25 21	20 16	15 0
LDR 0 1 1 0 1 1	base	rt	offset
6	5	5	16

Format:

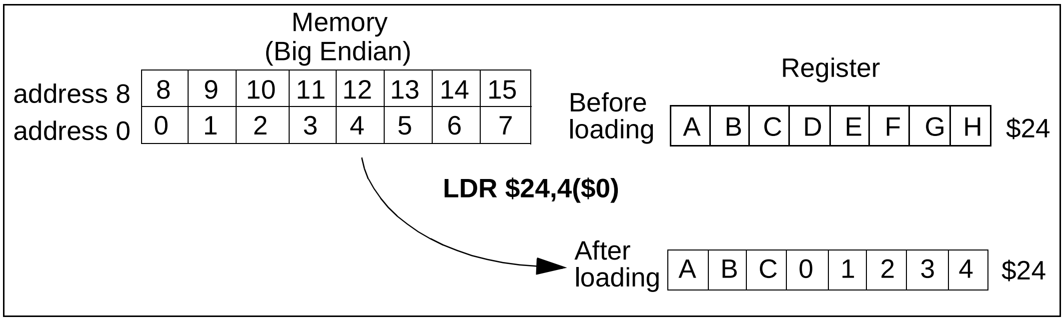
LDR rt, offset (base)   (MIPS III format)

Description:

This instruction is combined with the LDL instruction to load the word data in the memory that is not at the word boundary to general-purpose register *rt*. The LDL instruction loads the higher portion of the data to the register, while the LDR instruction loads the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address that can specify any byte. Of the word data in memory where the least-significant byte is specified by the generated address, only the data at the same doubleword boundary as the target address is loaded and stored to the lower portion of general-purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 8.

In other words, first the addressed byte is stored to the least-significant byte position of general-purpose register *rt*. If there is data of the high-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of general-purpose register *rt* is repeated. The remaining high-order byte is not affected.



LDR Load Doubleword Right (continued) LDR

The contents of general-purpose register *rt* are bypassed within the processor so that no NOP instruction is needed between an immediately preceding Load instruction that targets general-purpose register *rt* and a subsequent LDR (or LDL) instruction.

The Address Error exception does not occur even if the specified address is not located at the doubleword boundary.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

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

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LDR Load Doubleword Right (continued) LDR

The relationship between the address given to the LDR instruction and the result (bytes for registers) is shown below:

LDR

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2..0}$	Destination	Type	LEM	BEM	Destination	Type	LEM	BEM
0	I J K L M N O P	7	0	0	A B C D E F G I	0	7	0
1	A I J K L M N O	6	1	0	A B C D E F I J	1	6	0
2	A B I J K L M N	5	2	0	A B C D E I J K	2	5	0
3	A B C I J K L M	4	3	0	A B C D I J K L	3	4	0
4	A B C D I J K L	3	4	0	A B C I J K L M	4	3	0
5	A B C D E I J K	2	5	0	A B I J K L M N	5	2	0
6	A B C D E F I J	1	6	0	A I J K L M N O	6	1	0
7	A B C D E F G I	0	7	0	I J K L M N O P	7	0	0

Note:	*Type*:	Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a doubleword)
	Offset:	$pAddr_{2...0}$ Output to memory
	LEM	Little-endian memory (BigEndianMem = 0)
	BEM	Big-endian memory (BigEndianMem = 1)

LDR — Load Doubleword Right (continued) — LDR

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

LH Load Halfword LH

31 26	25 21	20 16	15 0
LH 1 0 0 0 0 1	base	rt	offset
6	5	5	16

Format:

LH rt, offset (base) (MIPS I format)

Description:

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

If the least-significant bit of the address is not zero, an Address Error exception occurs.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ (ReverseEndian^{2} \| 0))$
          $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}\ xor\ (BigEndianCPU^{2} \| 0)$
          $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte...8*\ byte}$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ (ReverseEndian^{2} \| 0))$
          $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}\ xor\ (BigEndianCP\ ^{2} \| 0)$
          $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte...8*\ byte}$
```

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LHU — Load Halfword Unsigned — LHU

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LHU 1 0 0 1 0 1	base	rt	offset
6	5	5	16

Format:

LHU rt, offset (base)   (MIPS I format)

Description:

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

If the least-significant bit of the address is not zero, an Address Error exception occurs.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $(ReverseEndian^2 \| 0))$
            $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
            $byte \leftarrow vAddr_{2...0}$ xor $(BigEndianCPU^2 \| 0)$
            $GPR[rt] \leftarrow 0^{16} \| mem_{15+8*byte...8*byte}$

64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ 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 Miss exception	TLB Invalid exception
Bus Error exception	Address Error exception

LL Load Linked LL

31 26	25 21	20 16	15 0
LL 1 1 0 0 0 0	base	rt	offset
6	5	5	16

Format:

LL rt, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general-purpose register *rt*. In 64-bit mode, the loaded word is sign extended. In addition, the specified physical address of the memory is stored in the LLAddr register, and sets the *LL* bit to 1. Afterward, the processor checks whether the address stored in the LLAddr register has been rewritten by the other processors or devices.

This instruction is provided for compatibility with MIPS implementations that implement multiprocessing facilities; however, the VR5432 does not implement these facilities.

Load Linked (LL) and Store Conditional (SC) instructions can be used to update memory atomically:

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

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

This instruction is available in User mode; it is not necessary to enable CP0.

LL **Load Linked (continued)** LL

If the specified address is in the noncache area, the operation of the LL instruction is undefined. A cache miss that occurs between the LL and SC instructions hinders execution of the SC instruction. Usually, therefore, one should not use a Load or Store instruction between the LL and SC instructions. Otherwise, the operation of the SC instruction is not guaranteed. If an exception frequently occurs, the exception also hinders execution of the SC instruction. It is therefore necessary to disable the exception temporarily.

If either of the low-order two bits of the address is not zero, an Address Error exception takes place.

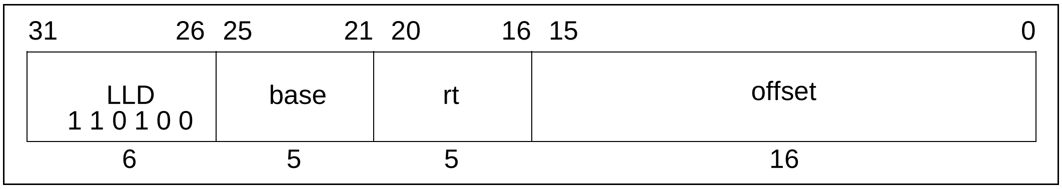
Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ (ReverseEndian \| 0^2))$
          $mem \leftarrow LoadMemory (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}\ xor\ (BigEndianCPU \| 0^2)$
          $GPR[rt] \leftarrow mem_{31+8*byte...8*byte}$
          $LLbit \leftarrow 1$
          $LLAddr \leftarrow pAddr$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ (ReverseEndian \| 0^2))$
          $mem \leftarrow LoadMemory (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2...0}\ xor\ (BigEndianCPU \| 0^2)$
          $GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \| mem_{31+8*byte...8*byte}$
          $LLbit \leftarrow 1$
          $LLAddr \leftarrow pAddr$
```

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LLD Load Linked Doubleword LLD



Format:

LLD rt, offset (base) (MIPS III format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of the doubleword at the memory location specified by the address are loaded into general-purpose register *rt*. In addition, the specified physical address of the memory is stored in the LLAddr register, and sets the *LL* bit to 1. Afterward, the processor checks whether the address stored in the LLAddr register has been rewritten by the other processors or devices.

This instruction is provided for compatibility with MIPS implementations that implement multiprocessing facilities; the VR5432 does not implement these facilities.

The Load Linked Doubleword (LLD) instruction and the Store Conditional Doubleword (SCD) instruction can be used to update the memory atomically:

```
L1:
    LL      T1, (T0)
    DADD    T2, T1, 1
    SCD     T2, (T0)
    BEQ     T2, 0, L1
    NOP
```

This atomically increments the doubleword addressed by T0. Changing the DADD instruction to an OR instruction changes this to an atomic bit set.

LLD — Load Linked Doubleword (continued) — LLD

If the specified address is in a noncache area, the operation of the LLD instruction is undefined. If a data cache miss occurs between the LLD and SCD instructions, the operation of the SCD instruction is not guaranteed. Therefore, do not use a Load or Store instruction between the LLD and SCD instructions. An exception also causes the operation of the SCD instruction to not be guaranteed, so it is necessary to disable exceptions temporarily.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15...0}$) + GPR[base]
                  (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
            LLAddr ← pAddr
64    T:    vAddr ← ((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
            LLAddr ← pAddr
```

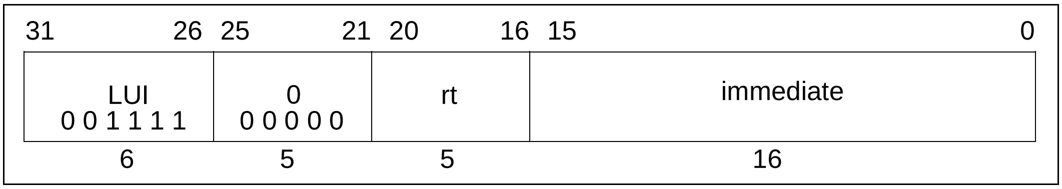
Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LLD Load Linked Doubleword (continued) LLD

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

LUI Load Upper Immediate LUI



Format:

LUI rt, immediate (MIPS I format)

Description:

The 16-bit *immediate* is shifted left 16 bits and extended on the right with 16 bits of zeros. The result is placed into general-purpose register *rt*. In 64-bit mode, the 32-bit result is sign extended to 64 bits.

Operation:

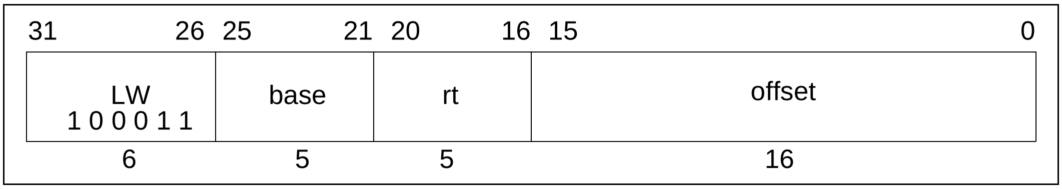
```
32    T:    GPR[rt] ← immediate || $0^{16}$

64    T:    GPR[rt] ← $(\text{immediate}_{15})^{32}$ || immediate || $0^{16}$
```

Exceptions:

None

LW — Load Word — LW



Format:

LW rt, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general-purpose register *rt*. In 64-bit mode, the loaded word is sign extended to 64 bits.

If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

Operation:

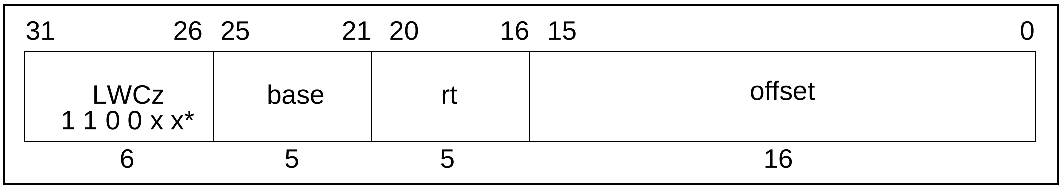
```
32   T:   vAddr $\leftarrow$ $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15...0})$ + GPR[base]
          (pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA)
          mem $\leftarrow$ LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          GPR[rt] $\leftarrow$ mem

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

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LWCz Load Word to Coprocessor z LWCz



Format:

LWCz rt, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The processor loads a word at the addressed memory location to general-purpose register *rt* of CPz. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

This instruction is not valid for use with CP0.

LWCz — Load Word to Coprocessor z (continued) — LWCz

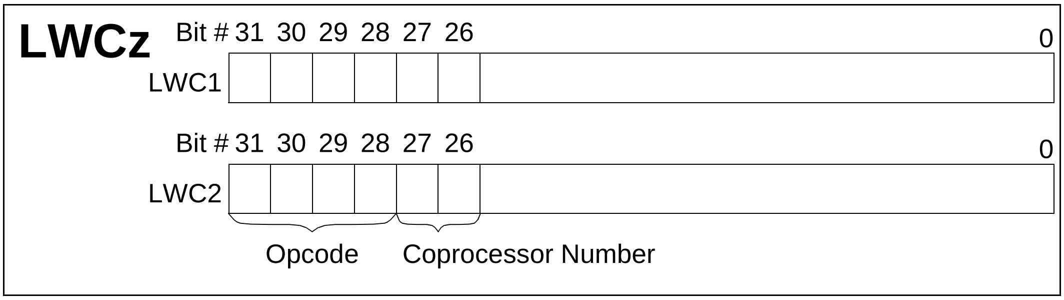
Operation:

```
32   T:   vAddr ← ((offset_15)^16 || offset_15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr_PSIZE-1...3 || (pAddr_2...0 xor (ReverseEndian || 0^2))
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          byte ← vAddr_2...0 xor (BigEndianCPU || 0^2)
          COPzLW (byte, rt, mem)
64   T:   vAddr ← ((offset_15)^48 || offset_15...0) + GPR[base]
          (pAddr, uncached)← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr_PSIZE-1...3 || (pAddr_2...0 xor (ReverseEndian || 0^2))
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          byte ← vAddr_2...0 xor (BigEndianCPU || 0^2)
          COPzLW (byte, rt, mem)
```

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Coprocessor Unusable exception

Opcode Bit Encoding:



LWL **Load Word Left** LWL

31 26	25 21	20 16	15 0
LWL 1 0 0 0 1 0	base	rt	offset
6	5	5	16

Format:

LWL rt, offset (base) (MIPS I format)

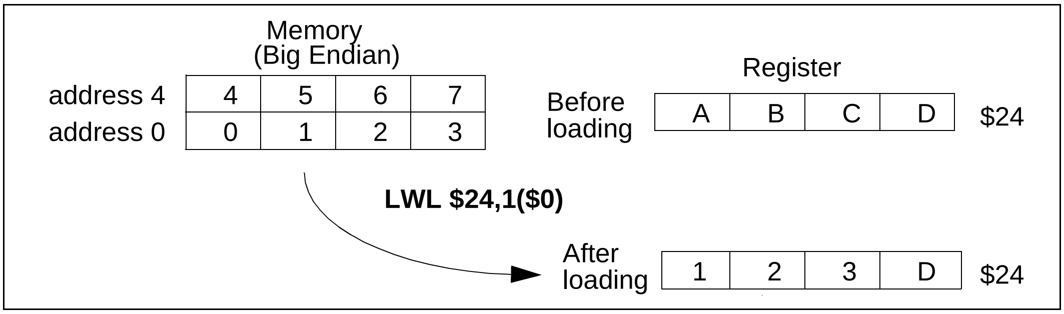
Description:

This instruction is combined with the LWR instruction to load word data in memory that is not at a word boundary to general-purpose register *rt*. The LWL instruction loads the higher portion of the data to the register, while the LWR instruction loads the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address that can specify any byte. Of the word data in the memory where the most-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the higher portion of general-purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 4.

In other words, first the addressed byte is stored to the most-significant byte position of general-purpose register *rt*. If there is data of the high-order byte that follows the same word boundary, the operation to store this data to the next byte of general-purpose register *rt* is repeated.

The remaining higher byte is not affected.



LWL Load Word Left (continued) LWL

The contents of general-purpose register *rt* are bypassed within the processor, so that no NOP instruction is needed between an immediately preceding Load instruction that targets general-purpose register *rt* and a subsequent LWL (or LWR) instruction.

The Address Error Exception does not occur, even if the specified address is not located at the word boundary.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \mid\mid offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \mid\mid (pAddr_{2...0}$ xor $ReverseEndian^3)$
          if BigEndianMem = 0 then
                  $pAddr \leftarrow pAddr_{PSIZE-1...2} \mid\mid 0^2$
          endif
          $byte \leftarrow vAddr_{1...0}$ xor $BigEndianCPU^2$
          $word \leftarrow vAddr_2$ xor $BigEndianCPU$
          $mem \leftarrow LoadMemory (uncached, byte, pAddr, vAddr, DATA)$
          $temp \leftarrow mem_{32*word+8*byte+7} \mid\mid GPR[rt]_{23-8*byte...0}$
          $GRP[rt] \leftarrow temp$

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \mid\mid offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \mid\mid (pAddr_{2...0}$ xor $ReverseEndian^3)$
          if BigEndianMem = 0 then
                  $pAddr \leftarrow pAddr_{PSIZE-1...2} \mid\mid 0^2$
          endif
          $byte \leftarrow vAddr_{1...0}$ xor $BigEndianCPU^2$
          $word \leftarrow vAddr_2$ xor $BigEndianCPU$
          $mem \leftarrow LoadMemory (uncached, byte, pAddr, vAddr, DATA)$
          $temp \leftarrow mem_{32*word+8*byte+7} \mid\mid GPR[rt]_{23-8*byte...0}$
          $GPR[rt] \leftarrow (temp_{31})^{32} \mid\mid temp$
```

LWL — Load Word Left (continued) — LWL

The relationship between the address given to the LWL instruction and the result (bytes for registers) is shown below:

LWL

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2...0}$	Destination	Type	LEM	BEM	Destination	Type	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

Note:	*Type*:	Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a double-word)
	Offset:	$pAddr_{2...0}$ Output to memory
	LEM	Little-endian memory (BigEndianMem = 0)
	BEM	Big-endian memory (BigEndianMem = 1)
	S:	Sign extension of destination bit 31

LWL Load Word Left (continued) LWL

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LWR　　Load Word Right　　LWR

31　　26	25　　21	20　　16	15　　0
LWR 1 0 0 1 1 0	base	rt	offset
6	5	5	16

Format:

LWR rt, offset (base)　　(MIPS I format)

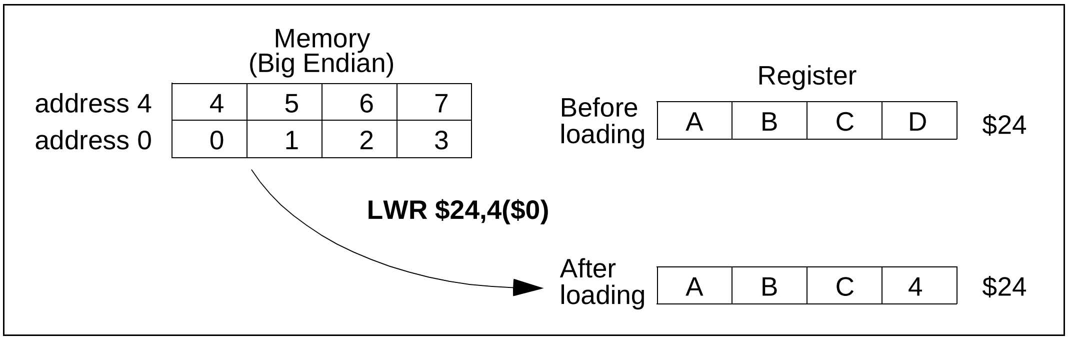
Description:

This instruction is combined with the LWL instruction to load the word data in the memory that is not at the word boundary to general-purpose register *rt*. The LWL instruction loads the higher portion of the data to the register, while the LWR instruction loads the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address that can specify any byte. Of the word data in the memory where the least-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the lower portion of general-purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 4.

In other words, first the addressed byte is stored to the least-significant byte position of general-purpose register *rt*. If there is data of the high-order byte that follows the same word boundary, the operation to store this data to the next byte of general-purpose register *rt* is repeated.

The remaining high-order byte is not affected.



LWR Load Word Right (continued) LWR

The contents of general-purpose register *rt* are bypassed within the processor, so that no NOP instruction is needed between an immediately preceding Load instruction that targets general-purpose register *rt* and a following LDL (or LWR) instruction.

The Address Error exception does not occur even if the specified address is not located at the word boundary.

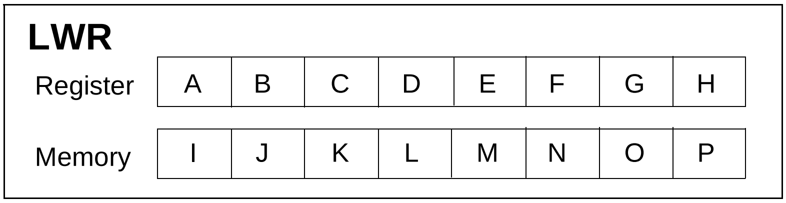
Operation:

```
32    T:    vAddr ← $((offset_{15})^{16}$ || $offset_{15...0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $pAddr_{PSIZE-1...3}$ || $(pAddr_{2...0}$ xor $ReverseEndian^3)$
            if BigEndianMem = 1 then
                    pAddr ← $pAddr_{PSIZE-31...3}$ || $0^3$
            endif
            byte ← $vAddr_{1...0}$ xor $BigEndianCPU^2$
            word ← $vAddr_2$ xor BigEndianCPU
            mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
            temp ← $mem_{31...32-8*byte...0}$ || $mem_{31+32*word-32*word+8*byte}$
            GPR[rt] ← temp

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

LWR — Load Word Right (continued) — LWR

The relationship between the address given to the LWR instruction and the result (bytes for registers) is shown below:



	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2...0}$	Destination	Type	LEM	BEM	Destination	Type	LEM	BEM
0	S S S S M N O P	3	0	4	X X X X E F G I	0	7	0
1	X X X X E M N O	2	1	4	X X X X E F I J	1	6	0
2	X X X X E F M N	1	2	4	X X X X E I J K	2	5	0
3	X X X X E F G M	0	3	4	S S S S I J K L	3	4	0
4	S S S S I J K L	3	4	0	X X X X E F G M	0	3	4
5	X X X X E I J K	2	5	0	X X X X E F M N	1	2	4
6	X X X X E F I J	1	6	0	X X X X E M N O	2	1	4
7	X X X X E F G I	0	7	0	S S S S M N O P	3	0	4

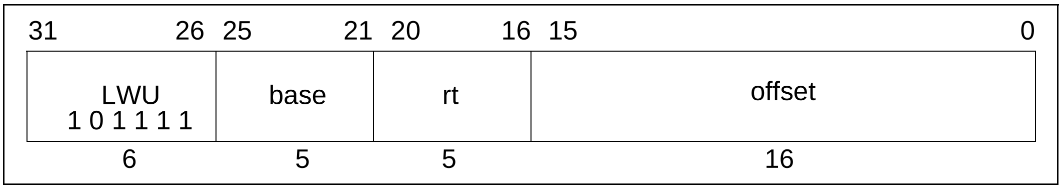
Note:		
	Type:	Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a double-word)
		*Offset*pAddr$_{2...0}$ Output to memory
		*LEM*Little-endian memory (BigEndianMem = 0)
		*BEM*Big-endian memory (BigEndianMem = 1)
	S:	Sign extension of destination bit 31
	X:	Not affected

LWR — Load Word Right (continued) — LWR

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LWU Load Word Unsigned LWU



Format:

LWU rt, offset (base) (MIPS III format)

Description:

The 16-bit *offset* is sign extended and added to the contents of the general-purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general-purpose register *rt*. The loaded word is zero-extended in 64-bit mode.

If either of the low-order two bits of the effective address is not zero, an Address Error exception occurs.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
32    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)
            GPR[rt] ← mem
64    T:    vAddr ← $((\text{offset}_{15})^{48} \parallel \text{offset}_{15...0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← $0^{32} \parallel$ mem
```

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LWU — Load Word Unsigned (continued) — LWU

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

MACC — Multiply, Accumulate, and Move LO — MACC

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

Format:

MACC rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is added to the signed contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MACC instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MACCHI **Multiply, Accumulate, and Move HI** MACCHI

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

Format:

MACCHI rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is added to the signed contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MACCHI instruction.

Operation:

```
32, 64    T:    $HI_{31..0} || LO_{31...0} \leftarrow (HI_{31...0} || LO_{31...0}) + (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} || LO_{31...0}) + (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MACCHIU Unsigned Multiply, Accumulate, and Move HI MACCHIU

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

Format:

MACCHIU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is added to the unsigned contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MACCHIU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MACCU — Unsigned Multiply, Accumulate, and Move LO — MACCU

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

Format:

MACCU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is added to the unsigned contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MACCU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...0}) + (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MFC0 — Move from System Control Coprocessor — MFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFC0 rt, rd (MIPS I format)

Description:

The contents of general-purpose register *rd* of the CP0 are loaded into general-purpose register *rt*.

Operation:

```
32    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← data

64    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← $(data_{31})^{32}$ || $data_{31...0}$
```

Exceptions:

Coprocessor Unusable exception (64-/32-bit User and Supervisor mode if CP0 is disabled)

MFCz Move from Coprocessor z MFCz

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 x x*	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFCz rt, rd (MIPS I format)

Description:

The contents of general-purpose register *rd* of CPz are loaded into general-purpose register *rt*.

Operation:

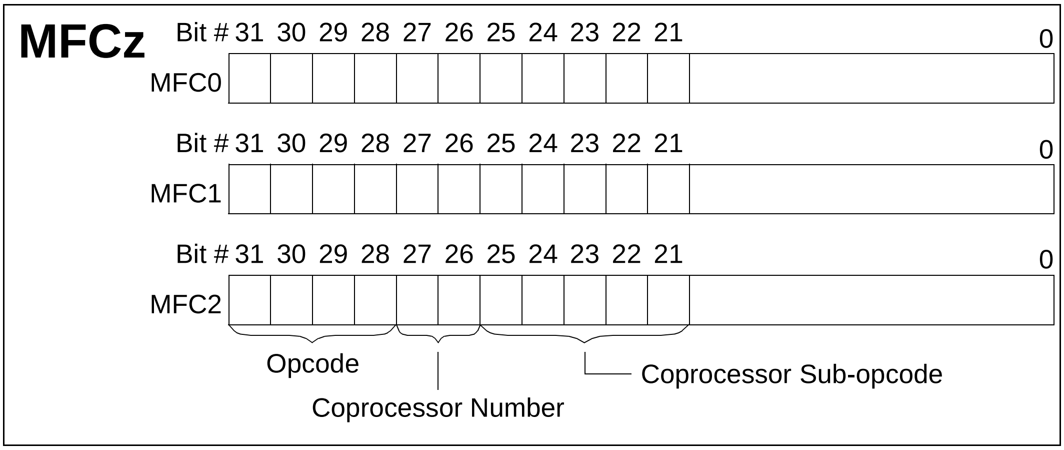
```
32      T:      data ← CPR[z,rd]
        T+1:    GPR[rt] ← data
64      T:      if $rd_0$ = 0 then
                    data ← $CPR[z, rd_{4...1} || 0]_{31...0}$
                else
                    data ← $CPR[z, rd_{4...1} || 0]_{63...32}$
                endif
        T+1:    GPR[rt] ← $(data_{31})^{32}$ || data
```

Exceptions:

Coprocessor Unusable exception

MFCz Move from Coprocessor z (continued) MFCz

Opcode Bit Encoding:


MFCz
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MFC0
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MFC1
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MFC2
Opcode
Coprocessor Number
Coprocessor Sub-opcode

MFDR — Move from Debug Register — MFDR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 0 1 1 1 0 0	MFDR 0 0 0 0 0	rt	dr	0 0 0 0 0 0	Debug Move 1 1 1 1 0 1
6	5	5	5	5	6

Format:

MFDR rt, dr (VR5432 format)

Description:

The contents of debug register *dr* are loaded into general-purpose register *rt.*

Operation:

```
32, 64 T:     GPR[rt] ← DEBUG[dr]
```

Exceptions:

None

MFHI **Move from HI** MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFHI 0 1 0 0 0 0
6	10	5	5	6

Format:

MFHI rd (MIPS I format)

Description:

The contents of special register HI are loaded into general-purpose register *rd*.

To ensure proper operation in the event of interrupts, the two instructions that follow an MFHI instruction may not be any of the instructions that modify the HI register: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MAC, MACC, MACCHI, MACCHIU, MACCU, MTHI, MUL, MULHI, MULHIU, MULT, MULTU, or MULU.

Operation:

```
32, 64    T:        GPR[rd] ← HI
```

Exceptions:

None

MFLO **Move from LO** MFLO

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFLO 0 1 0 0 1 0
6	10	5	5	6

Format:

MFLO rd (MIPS I format)

Description:

The contents of special register LO are loaded into general-purpose register *rd*.

To ensure proper operation in the event of interruptions, the two instructions that follow an MFLO instruction may not be any of the instructions that modify the L register: DDIV, DDIVU, DIV, DIVU, DMAC, DMULT, DMULTU, MAC, MACC, MACCHI, MACCHIU, MACCU, MTLO, MUL, MULHI, MULHIU, MULT, MULTU, or MULU.

Operation:

```
32, 64      T:          GPR[rd] ← LO
```

Exceptions:

None

MFPC — Move from Performance Counter — MFPC

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 0 1 0 0 0 0	MFPC 0 0 0 0 0	rt	CP0 Move 1 1 0 0 1	0 0 0 0 0 0	reg	1
6	5	5	5	5	5	1

Format:

MFPC rt, reg (VR5432 format)

Description:

The contents of Performance Counter *reg* are loaded into general-purpose register *rt*.

Operation:

```
32,64  T:     GPR[rt] ← CPR[0,reg]
```

Exceptions:

Coprocessor Unusable exception

MFPS — Move from Performance Event Specifier — MFPS

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 0 1 0 0 0 0	MFPS 0 0 0 0 0	rt	CP0 Move 1 1 0 0 1	0 0 0 0 0 0	reg	0
6	5	5	5	5	5	1

Format:

MFPS rt, reg (VR5432 format)

Description:

The contents of performance event specifier *reg* are loaded into general-purpose register *rt.*

Operation:

```
32,64  T:    GPR[rt] ← CPR[0,reg]
```

Exceptions:

Coprocessor Unusable exception

MOVN Move Conditional on Not Zero MOVN

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

Format:

MOVN rd, rs, rt (MIPS IV format)

Description:

If the value in general-purpose register *rt* is not equal to zero, then the contents of general-purpose register *rs* are placed into general-purpose register *rd*.

Operation:

```
if GPR[rt] ≠ 0 then
        GPR[rd] ← GPR[rs]
endif
```

Note: The nonzero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Exceptions:

Reserved Instruction exception

MOVZ **Move Conditional on Zero** MOVZ

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

Format:

MOVZ rd, rs, rt (MIPS IV format)

Description:

If the value in general-purpose register *rt* is equal to zero, then the contents of general-purpose register *rs* are placed into general-purpose register *rd*.

Operation:

```
if GPR[rt] = 0 then
        GPR[rd] ← GPR[rs]
endif
```

Note: The nonzero value tested here is the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Exceptions:

Reserved Instruction exception

MSAC — Multiply, Negate, Accumulate, and Move LO — MSAC

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

Format:

MSAC rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is subtracted from the signed contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MSAC instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0})$ - (GPR[rs] * GPR[rt])
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...}$ ) - (GPR[rs] * GPR[rt]))$_{31..0}$
```

Exceptions:

None

MSACHI — Multiply, Negate, Accumulate, and Move HI — MSACHI

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

Format:

MSACHI rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is subtracted from the signed contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MSACHI instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0}) - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...}\ ) - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MSACHIU — Unsigned Multiply, Negate, Accumulate, and Move HI — MSACHIU

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

Format:

MSACHIU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is subtracted from the unsigned contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MSACHIU instruction.

Operation:

32, 64 T: $HI_{31..0} \,||\, LO_{31...0} \leftarrow (HI_{31...0} \,||\, LO_{31...0}) - (GPR[rs] * GPR[rt])$

$GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \,||\, LO_{31...}\ ) - (GPR[rs] * GPR[rt]))_{63..32}$

Exceptions:

None

MSACU — Unsigned Multiply, Negate, Accumulate, and Move LO — MSACU

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

Format:

MSACU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is subtracted from the unsigned contents of the 64-bit accumulator formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MSACU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow (HI_{31...0} \| LO_{31...0})$ - (GPR[rs] * GPR[rt])
                $GPR[rd]_{31..0} \leftarrow ((HI_{31...0} \| LO_{31...}$ ) - (GPR[rs] * GPR[rt]))$_{31..0}$
```

Exceptions:

None

MTC0 — Move to System Control Coprocessor — MTC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MT 0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 00
6	5	5C	5	11

Format:

MTC0 rt, rd (MIPS I format)

Description:

The contents of general-purpose register *rt* are loaded into general-purpose register *rd* of CP0.

Because the contents of the TLB may be altered by this instruction, the operation of Load and Store instructions and TLB operations for the instructions immediately before and after this instruction are undefined.

Operation:

```
32, 64   T:       data ← GPR[rt]
         T+1:     CPR[0, rd] ← data
```

Exceptions:

Coprocessor Unusable exception

MTCz — Move to Coprocessor z — MTCz

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COPz 0 1 0 0 x x*	MT 0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MTCz rt, rd (MIPS I format)

Description:

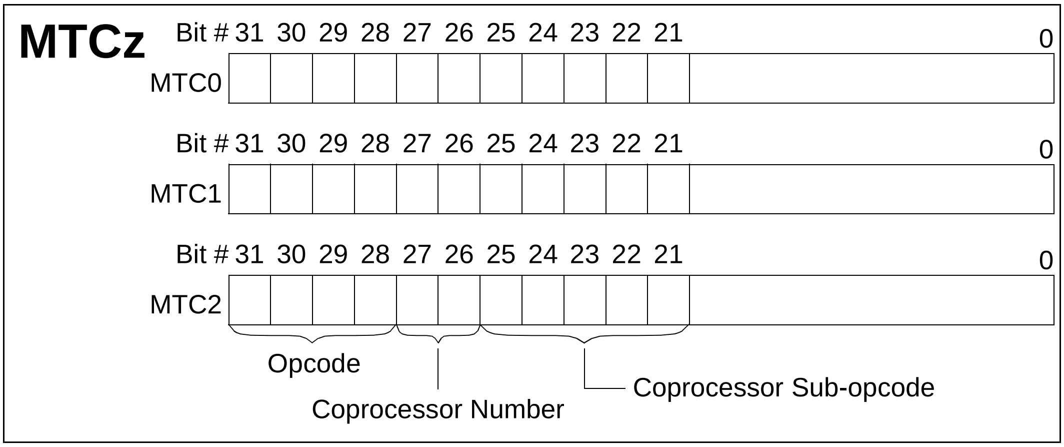
The contents of general-purpose register *rt* are loaded into general-purpose register *rd* of CPz.

Operation:

```
32    T:     data ← GPR[rt]
      T+1:   CPR[z, rd] ← data

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

Opcode Bit Encoding:


MTCz
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MTC0
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MTC1
Bit # 31 30 29 28 27 26 25 24 23 22 21 0
MTC2
Opcode
Coprocessor Number
Coprocessor Sub-opcode

Exceptions:

Coprocessor Unusable exception

MTDR — Move to Debug Register — MTDR

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL2 0 1 1 1 0 0	MTDR 0 0 1 0 0	rt	dr	0 0 0 0 0 0	Debug Move 1 1 1 1 0 1
6	5	5	5	5	6

Format:

MTDR rt, dr (VR5432 format)

Description:

The contents of general-purpose register *rt* are loaded into debug register *dr*.

Operation:

```
32, 64 T:    DEBUG[dr] ← GPR[rt]
```

Exceptions:

None

MTHI **Move to HI** MTHI

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTHI 0 1 0 0 0 1
6	5	15	6

Format:

MTHI rs (MIPS I format)

Description:

The contents of general-purpose register *rs* are loaded into special register HI.

If the MTHI instruction is executed following the MULT, MULTU, DIV, or DIVU instruction, the operation is performed normally. However, if the MFLO, MFHI, MTLO, or MTHI instruction is executed following the MTHI instruction, the contents of special register LO are undefined.

Operation:

```
32,64      T–2:  HI ← undefined
           T–1:  HI ← undefined
           T:    HI ← GPR[rs]
```

Exceptions:

None

MTLO — Move to LO — MTLO

31 — 26	25 — 21	20 — 6	5 — 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTLO 0 1 0 0 1 1
6	5	15	6

Format:

MTLO rs (MIPS I format)

Description:

The contents of general-purpose register *rs* are loaded into special register LO.

If the MTLO instruction is executed following the MULT, MULTU, DIV, or DIVU instruction, the operation is performed normally. However, if the MFLO, MFHI, MTLO, or MTHI instruction is executed following the MTLO instruction, the contents of special register HI are undefined.

Operation:

```
32,64     T–2:  LO ← undefined
          T–1:  LO ← undefined
          T:    LO ← GPR[rs]
```

Exceptions:

None

MTPC — Move to Performance Counter — MTPC

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 0 1 0 0 0 0	MTPC 0 0 1 0 0	rt	CP0 Move 1 1 0 0 1	0 0 0 0 0 0	reg	1
6	5	5	5	5	5	1

Format:

MTPC rt, reg (VR5432 format)

Description:

The contents of general-purpose register *rt* are loaded into Performance Counter *reg*.

Operation:

```
32,64  T:     CPR[0,reg] ← GPR[rt]
```

Exceptions:

Coprocessor Unusable exception

MTPS — Move to Performance Event Specifier — MTPS

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 0 1 0 0 0 0	MTPS 0 0 1 0 0	rt	CP0 Move 1 1 0 0 1	0 0 0 0 0 0	reg	0
6	5	5	5	5	5	1

Format:

MTPS rt, reg (VR5432 format)

Description:

The contents of general-purpose register *rt* are loaded into performance event specifier *reg.*

Operation:

```
32,64  T:     CPR[0,reg] ← GPR[rt]
```

Exceptions:

Coprocessor Unusable exception

MUL **Multiply and Move LO** MUL

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

Format:

MUL rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

An Integer Overflow exception never occurs.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MUL instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow GPR[rs] * GPR[rt]$
                $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{31..0}$
```

Exceptions:

None

MULHI **Multiply and Move HI** MULHI

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

Format:

MULHI rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

An Integer Overflow exception never occurs.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULHI instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow GPR[rs] * GPR[rt]$
                $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{63..32}$
```

Exceptions:

None

MULHIU — Unsigned Multiply and Move HI — MULHIU

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

Format:

MULHIU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULHIU instruction.

Operation:

```
32, 64    T:    $HI_{31..0}$ || $LO_{31...0}$ $\leftarrow$ GPR[rs] * GPR[rt]
                $GPR[rd]_{31..0}$ $\leftarrow$ $(GPR[rs] * GPR[rt])_{63..32}$
```

Exceptions:

None

MULS — Multiply, Negate, and Move LO — MULS

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

Format:

MULS rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied, and the product is negated and stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

An Integer Overflow exception never occurs.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULS instruction.

Operation:

32, 64 T: $HI_{31..0} \| LO_{31...0} \leftarrow 0 - (GPR[rs] * GPR[rt])$

$GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{31..0}$

Exceptions:

None

MULSHI — Multiply, Negate, and Move HI — MULSHI

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

Format:

MULSHI rd, rs, rt (VR5432 format)

Description:

The signed 32-bit operands in the *rs* and *rt* registers are multiplied and the product is negated and stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

An Integer Overflow exception never occurs.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULSHI instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MULSHIU — Unsigned Multiply, Negate, and Move HI — MULSHIU

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

Format:

MULSHIU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied and the product is negated and stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the most-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULSHIU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MULSU — Unsigned Multiply, Negate, and Move LO — MULSU

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

Format:

MULSU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied and the product is stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULSU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MULT Multiply MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	MULT 0 1 1 0 0 0
6	5	5	10	6

Format:

MULT rs, rt (MIPS I format)

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. An Integer Overflow exception never occurs.

In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the doubleword result is loaded into special register LO, and the high-order word of the doubleword result is loaded into special register HI. In the 64-bit mode, the respective results are sign extended and stored.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULT instruction.

MULT Multiply (continued) MULT

Operation:

```
32    T–2:  LO      ← undefined
            HI      ← undefined
      T–1:  LO      ← undefined
            HI      ← undefined
      T:    t       ← GPR[rs] * GPR[rt]
            LO      ← $t_{31...0}$
            H I     ← $t_{63...32}$
64    T–2:  LO      ← undefined
            HI      ← undefined
      T–1:  LO      ← undefined
            HI      ← undefined
      T:    t       ← $GPR[rs]_{31...0}$ * $GPR[rt]_{31...0}$
            LO      ← $(t_{31})^{32}$ || $t_{31...0}$
            HI      ← $(t_{63})^{32}$ || $t_{63...32}$
```

Exceptions:

None

MULTU Unsigned Multiply MULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	MULTU 0 1 1 0 0 1
6	5	5	10	6

Format:

MULTU rs, rt (MIPS I format)

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned values. An Integer Overflow exception never occurs.

In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the doubleword result is loaded into special register LO, and the high-order word of the doubleword result is loaded into special register HI. In 64-bit mode, these results are sign extended and loaded.

If either of the two preceding instructions is MFHI or MFLO, the execution results of these transfer instructions are undefined. To obtain the correct result, insert two or more additional instructions between MFHI or MFLO and the MULTU instruction.

MULTU — Unsigned Multiply (continued) — MULTU

Operation:

```
32    T–2:  LO    ← undefined
            HI    ← undefined
      T–1:  LO    ← undefined
            HI    ← undefined
      T:    t     ← (0 || GPR[rs]) * (0 || GPR[rt])
            LO    ← $t_{31...0}$
            HI    ← $t_{63...32}$
64    T–2:  LO    ← undefined
            HI    ← undefined
      T–1:  LO    ← undefined
            HI    ← undefined
      T:    t     ← (0 || $GPR[rs]_{31...0}$) * (0 || $GPR[rt]_{31...0}$)
            LO    ← $(t_{31})^{32}$ || $t_{31...0}$
            HI    ← $(t_{63})^{32}$ || $t_{63...32}$
```

Exceptions:

None

MULU Unsigned Multiply and Move LO MULU

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

Format:

MULU rd, rs, rt (VR5432 format)

Description:

The unsigned 32-bit operands in the *rs* and *rt* registers are multiplied and the product is stored in the 64-bit register formed by the least-significant 32 bits of the HI and LO registers. A copy of the least-significant 32 bits of the result is stored in general-purpose register *rd*.

If either of the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain correct results, insert two or more other instructions between MFHI or MFLO and the MULU instruction.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31...0} \leftarrow GPR[rs] * GPR[rt]$
                $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{31..0}$
```

Exceptions:

None

NOR **NOR** NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	NOR 1 0 0 1 1 1
6	5	5	5	5	6

Format:

NOR rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are bitwise NORed with the contents of general-purpose register *rt*. The result is stored in general-purpose register *rd*.

Operation:

```
32, 64     T:          GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

OR

OR

OR

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

Format:

OR rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are bitwise ORed with the contents of general-purpose register *rt*. The result is stored in general-purpose register *rd*.

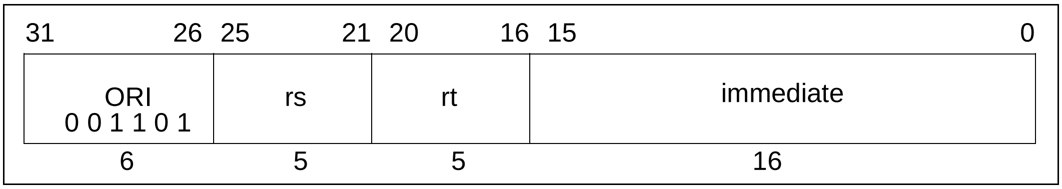
Operation:

```
32, 64     T:          GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

ORI — OR Immediate — ORI



Format:

ORI rt, rs, immediate (MIPS I format)

Description:

The 16-bit *immediate* is zero extended and bitwise ORed with the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt*.

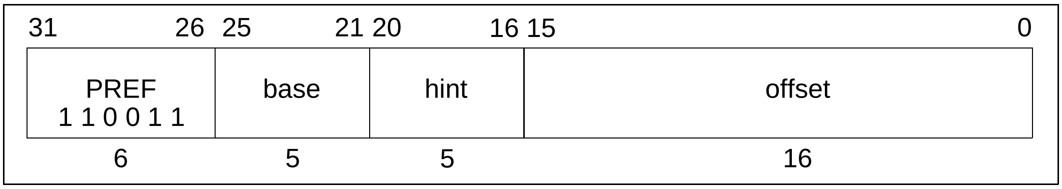
Operation:

```
32    T:    $GPR[rt] \leftarrow GPR[rs]_{31...16}$ || (immediate or $GPR[rs]_{15...0}$)
64    T:    $GPR[rt] \leftarrow GPR[rs]_{63...16}$ || (immediate or $GPR[rs]_{15...0}$)
```

Exceptions:

None

PREF Prefetch PREF



Format:

PREF hint, offset (base) (MIPS IV format)

Description:

PREF adds the 16-bit signed *offset* to the contents of general-purpose register *base* to form an effective byte address. It advises that data at the effective address may be used in the near future. The *hint* field supplies information about the way the data is expected to be used.

Unlike the VR5000, in which the PREF instruction is executed as an NOP, the VR5432 data may be prefetched into the data cache as a result of executing this instruction.

PREF is an advisory instruction that may change the performance of the program. However, for all *hint* values and all effective addresses, it neither changes the architecturally visible state nor alters the meaning of the program.

If MIPS IV instructions are supported and enabled, PREF does not cause addressing-related exceptions. If it does happen to raise an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data is prefetched. However, even if no data is prefetched, some action that is not architecturally visible—such as write-back of a dirty cache line—can take place.

If PREF results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

The *hint* field supplies information about the way the data is expected to be used. A *hint* value cannot cause an action to modify an architecturally visible state. A processor may use a *hint* value to improve the effectiveness of the prefetch action. The defined *hint* values are shown in Table 17-17.

PREF Prefetch (continued) PREF

Table 17-17 Values of Hint Field for PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1	store	Data is expected to be stored or modified. Fetch data as if for a store.
2–3		Reserved
4	load_streamed	Data is expected to be loaded (not modified) but not reused extensively; it “streams” through the cache. Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Data is expected to be stored or modified but not reused extensively; it “streams” through the cache. Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”
8–24		Reserved
25	writeback_invalidate	
26–31		Reserved

PREF — Prefetch (continued) — PREF

PREF never generates a memory operation for a location with an *uncached* memory access type.

Prefetch cannot prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Reserved Instruction exception

ROR Rotate Right ROR

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

Format:

ROR rd, rt, sa (VR5432 format)

Description:

The contents of general-purpose register *rt* are rotated right by *sa* bits. The result is stored in general-purpose register *rd.*

Operation:

32, 64 T: $GPR[rd] \leftarrow GPR[rt]_{sa-1...0} \;||\; GPR[rt]_{31...sa}$

Exceptions:

None

RORV **Rotate Right Variable** RORV

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

Format:

RORV rd, rt, rs (VR5432 format)

Description:

The contents of general-purpose register *rt* are rotated right by the number of bits specified by the low-order five bits of general-purpose register *rs*. The result is stored in general-purpose register *rd*.

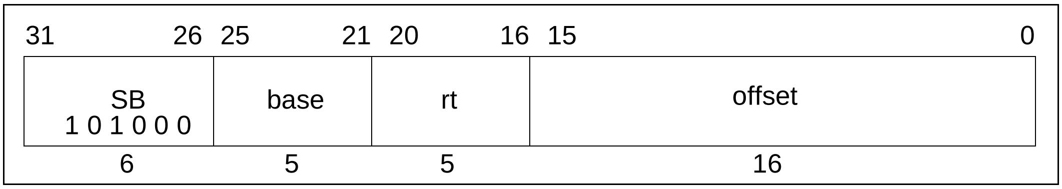
Operation:

```
32, 64 T:   $s \leftarrow GPR[rs]_{4...0}$
            $GPR[rd] \leftarrow GPR[rt]_{s-1...0} \| GPR[rt]_{31...s}$
```

Exceptions:

None

SB Store Byte SB



Format:

SB rt, offset (base) (MIPS I format)

Description:

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

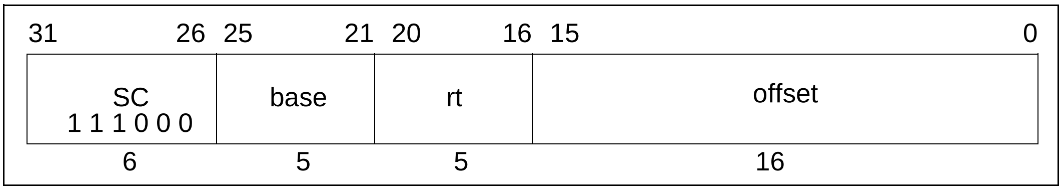
Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ 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)
64  T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $ReverseEndian^3)$
        $byte \leftarrow vAddr_{2...0}$ xor $BigEndianCPU^3$
        $data \leftarrow GPR[rt]_{63-8*byte...0} \| 0^{8*byte}$
        StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception

SC — Store Conditional — SC



Format:

SC rt, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of general-purpose register *rt* are stored at the memory location specified by the address only when the *LL* bit is set. If another processor or device changes the physical address after the previous LL instruction has been executed, or if the ERET instruction exists between the LL and SC instructions, the register contents are not stored to memory, and storing fails.

This instruction is provided for compatibility with MIPS implementations that implement multiprocessing facilities. The VR5432 does not implement these facilities.

The success or failure of the SC operation is indicated by the contents of general-purpose register *rt* after execution of the instruction. A successful SC instruction sets the contents of general-purpose register *rt* to 1; an unsuccessful SC instruction sets them to 0.

The operation of SC is undefined when the address is different from the address used in the last LL instruction.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the low-order two bits of the address is not zero, an Address Error exception takes place.

If this instruction both fails and causes an exception, the exception takes precedence.

SC Store Conditional (continued) SC

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
          $data \leftarrow GPR[rt]_{31...0}$
          if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{31} \|$ LLbit
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
          $data \leftarrow GPR[rt]_{31...0}$
          if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{63} \|$ LLbit
```

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception

SCD Store Conditional Doubleword SCD

31 … 26	25 … 21	20 … 16	15 … 0
SCD 1 1 1 1 0 0	base	rt	offset
6	5	5	16

Format:

SCD rt, offset (base) (MIPS III format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of general-purpose register *rt* are stored at the memory location specified by the address only when the *LL* bit is set. If another processor or device changes the target address after the previous LLD instruction has been executed, or if the ERET instruction exists between the LLD and SCD instructions, the register contents are not stored to memory, and storing fails.

This instruction is provided for compatibility with MIPS implementations that implement multiprocessing facilities. The VR5432 does not implement these facilities.

The success or failure of the SCD operation is indicated by the contents of general-purpose register *rt* after execution of the instruction. A successful SCD instruction sets the contents of general-purpose register *rt* to 1; an unsuccessful SCD instruction sets them to 0.

The operation of SCD is undefined when the address is different from the address used in the last LLD.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If any of the low-order three bits of the address is not zero, an Address Error exception takes place. If this instruction both fails and causes an exception, the exception takes precedence.

This instruction is defined in 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the Reserved Instruction exception occurs.

SCD Store Conditional Doubleword (continued) SCD

Operation:

```
64    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
```

Note: In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception
Reserved Instruction exception

SD **Store Doubleword** SD

31 26	25 21	20 16	15 0
SD 1 1 1 1 1 1	base	rt	offset
6	5	5	16

Format:

SD rt, offset (base) (MIPS III format)

Description:

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

If any of the low-order three bits of the address are not zero, an Address Error exception occurs.

This operation is defined in 64-bit mode and 32-bit Kernel mode. Execution of this instruction in 32-bit User or Supervisor mode causes a Reserved Instruction exception.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $data \leftarrow GPR[rt]$
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $data \leftarrow GPR[rt]$
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

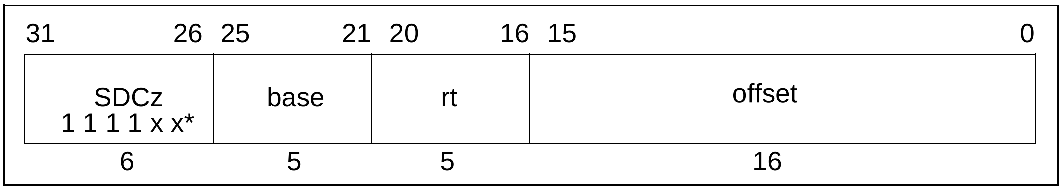
Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SD Store Doubleword (continued) SD

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception
Reserved Instruction exception

SDCz — Store Doubleword from Coprocessor z — SDCz



Format:

SDCz rt, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. Register *rt* of coprocessor unit *z* sources a doubleword, which the processor writes to the addressed memory location. The stored data is defined by individual coprocessor specifications.

If any of the low-order three bits of the address is not zero, an Address Error exception takes place.

This instruction is not valid for use with CP0.

When CP1 is specified, the *FR* bit of the Status register equals 0 and the least-significant bit in the *rt* field is not 0, the operation of this instruction is undefined. If the *FR* bit equals 1, both odd and even registers can be specified by *rt*.

SDCz — Store Doubleword from Coprocessor z (continued) — SDCz

Operation:

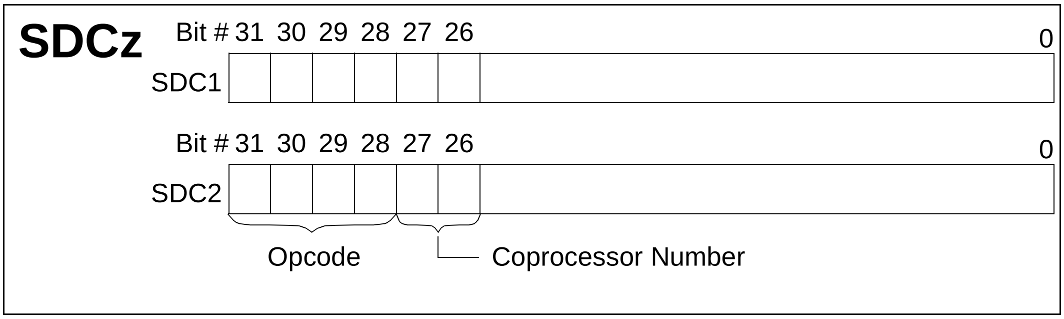
```
32   T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
         $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
         $data \leftarrow GPR(rt),$
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

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

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception
Coprocessor Unusable exception

Opcode Bit Encoding:



SDL **Store Doubleword Left** SDL

31 — 26	25 — 21	20 — 16	15 — 0
SDL 1 0 1 1 0 0	base	rt	offset
6	5	5	16

Format:

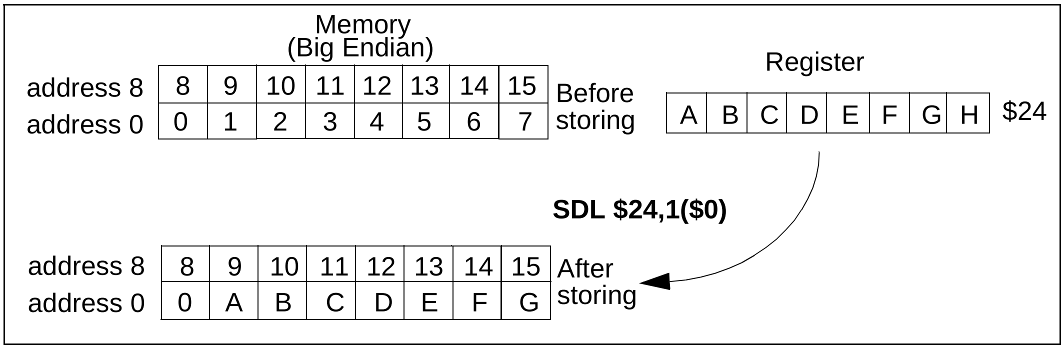
SDL rt, offset (base) (MIPS III format)

Description:

This instruction is used in combination with the SDR instruction to store the doubleword data in the register to the doubleword in the memory that is not at the doubleword boundary. The SDL instruction stores the higher portion of the data to the memory, while the SDR instruction stores the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address. Of the doubleword data in the memory where the most-significant byte is specified by the generated address, only the lower portion of general-purpose register *rt* is stored to memory at the same doubleword boundary as the target address. Depending on the address specified, the number of bytes to be loaded changes from one to eight.

In other words, first the most-significant byte position of general-purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the low-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of the memory is repeated.



SDL Store Doubleword Left (continued) SDL

The Address Error exception does not occur, even if the specified address is not located at the doubleword boundary. This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the Reserved Instruction exception occurs.

Operation:

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

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SDL — Store Doubleword Left (continued) — SDL

The relationships between the addresses given to the SDL instruction and the result (bytes for doublewords in the memory) are shown below:

SDL

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2...0}$	Destination	Type	LEM	BEM	Destination	Type	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

Note:	*Type*:	Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a doubleword)
	Offset:	$pAddr_{2...0}$ Output to memory
	LEM	Little-endian memory (BigEndianMem = 0)
	BEM	Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Reserved Instruction exception

SDR Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 1 0 1 1 0 1	base	rt	offset
6	5	5	16

Format:

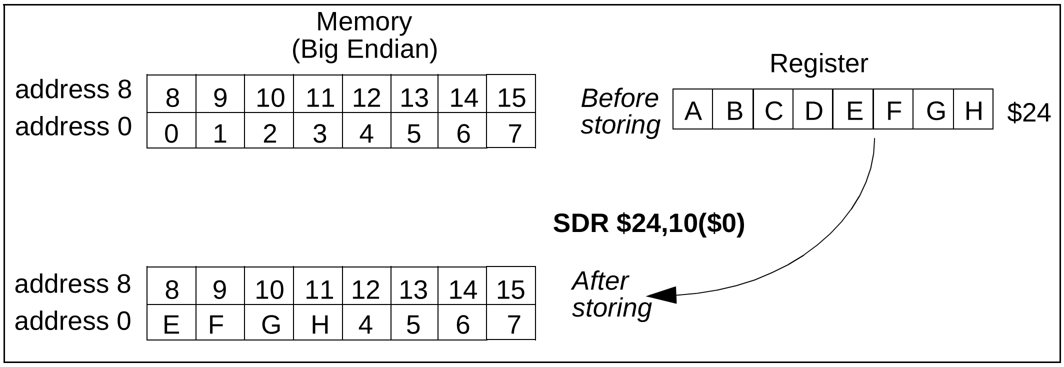
SDR rt, offset (base) (MIPS III format)

Description:

This instruction is used in combination with the SDL instruction to store the doubleword data in the register to the word data in the memory that is not at the doubleword boundary. The SDL instruction stores the higher portion of the data to the memory, while the SDR instruction stores the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address. Of the doubleword data in the memory where the least-significant byte is specified by the generated address, only the lower portion of general-purpose register *rt* is stored to memory at the same doubleword boundary as the target address. Depending on the address specified, the number of bytes to be loaded changes from 1 to 8.

In other words, first the least-significant byte position of general-purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the high-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of the memory is repeated.



SDR Store Doubleword Right (continued) SDR

The Address Error exception does not occur, even if the specified address is not located at the doubleword boundary. This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the Reserved Instruction exception occurs.

Operation:

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

Note: In 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SDR Store Doubleword Right (continued) SDR

The relationships between the addresses given to the SDR instruction and the result (bytes for doublewords in the memory) are shown below:

SDR

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2...0}$	Destination	Type	LEM	BEM	Destination	Type	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

Note: *Type*: Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a doubleword)

Offset: $pAddr_{2...0}$ Output to memory
LEM Little-endian memory (BigEndianMem = 0)
BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Reserved Instruction exception

SH Store Halfword SH

31 26	25 21	20 16	15 0
SH 1 0 1 0 0 1	base	rt	offset
6	5	5	16

Format:

SH rt, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The least-significant halfword of register *rt* is stored in the memory specified by the address.

If the least-significant bit of the address is not zero, an Address Error exception occurs.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...}) + GPR[base]$
              $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $(ReverseEndian^2 \| 0))$
          $byte \leftarrow vAddr_{2...0}$ xor $(BigEndianCPU^2 \| 0)$
          $data \leftarrow GPR[rt]_{63-8*byte...0} \| 0^{8*byte}$
          StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...}) + GPR[base]$
              $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ 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)
```

SH — Store Halfword (continued) — SH

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception

SLL Shift Left Logical SLL

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

Format:

SLL rd, rt, sa (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is stored in general-purpose register *rd*. In the 64-bit mode, the value resulting from sign-extending the shifted 32-bit value is stored as a result. If the shift value is 0, the low-order 32 bits of the 64-bit value are sign extended. This instruction can generate a 64-bit value that sign-extends a 32-bit value.

Operation:

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

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

Exceptions:

None

Caution: If the shift value of this instruction is 0, the assembler may treat this instruction as an NOP. When using this instruction for sign extension, check the specifications of the assembler.

SLLV **Shift Left Logical Variable** SLLV

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

Format:

SLLV rd, rt, rs (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted left the number of bits specified by the low-order five bits of general-purpose register *rs*, inserting zeros into the low-order bits. The result is stored in general-purpose register *rd*. In the 64-bit mode, the value resulting from sign-extending the shifted 32-bit value is stored as a result. If the shift value is 0, the low-order 32 bits of the 64-bit value are sign extended. This instruction can generate a 64-bit value that sign-extends a 32-bit value.

Operation:

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

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

Exceptions:

None

Caution: If the shift value of this instruction is 0, the assembler may treat this instruction as an NOP. When using this instruction for sign extension, check the specifications of the assembler.

SLT Set On Less Than SLT

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

Format:

SLT rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*. Interpreting these values as signed integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, one is stored in the general-purpose register *rd*; otherwise, zero is stored in general-purpose register *rd*.

An Integer Overflow exception never occurs. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < GPR[rt] then
                    $GPR[rd] \leftarrow 0^{31} || 1$
            else
                    $GPR[rd] \leftarrow 0^{32}$
            endif
64    T:    if GPR[rs] < GPR[rt] then
                    $GPR[rd] \leftarrow 0^{63} || 1$
            else
                    $GPR[rd] \leftarrow 0^{64}$
            endif
```

Exceptions:

None

SLTI — Set On Less Than Immediate — SLTI

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

Format:

SLTI rt, rs, immediate (MIPS I format)

Description:

The 16-bit *immediate* is sign extended and subtracted from the contents of general-purpose register *rs*. Interpreting these values as signed integers, if *rs* contents are less than the sign-extended *immediate*, one is stored in general-purpose register *rt*; otherwise, zero is stored in the general-purpose register *rt*.

An Integer Overflow exception never occurs. The comparison is valid even if the subtraction overflows.

Operation:

```
32    T:    if GPR[rs] < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$ then
                    GPR[rt] $\leftarrow 0^{31}$ || 1
            else
                    GPR[rt] $\leftarrow 0^{32}$
            endif
64    T:    if GPR[rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
                    GPR[rt] $\leftarrow 0^{63}$ || 1
            else
                    GPR[rt] $\leftarrow 0^{64}$
            endif
```

Exceptions:

None

SLTIU — Set On Less Than Immediate Unsigned — SLTIU

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

Format:

SLTIU rt, rs, immediate (MIPS I format)

Description:

The 16-bit *immediate* is sign extended and subtracted from the contents of general-purpose register *rs*. Interpreting these values as unsigned integers, if *rs* contents are less than the sign-extended *immediate*, one is stored in the general-purpose register *rt*; otherwise zero is stored in the general-purpose register *rt*.

An Integer Overflow exception never occurs. The comparison is valid even if the subtraction overflows.

Operation:

```
32    T:    if (0 || GPR[rs]) < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$ then
                        GPR[rt] ← $0^{31}$ || 1
            else
                        GPR[rt] ← $0^{32}$
            endif
64    T:    if (0 || GPR[rs]) < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
                        GPR[rt] ← $0^{63}$ || 1
            else
                        GPR[rt] ← $0^{64}$
            endif
```

Exceptions:

None

SLTU **Set On Less Than Unsigned** SLTU

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

Format:

SLTU rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*. Interpreting these values as unsigned integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, one is stored in general-purpose register *rd*; otherwise, zero is stored in the general-purpose register *rd*.

An Integer Overflow exception never occurs. The comparison is valid even if the subtraction overflows.

Operation:

```
32    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                        GPR[rd] ← $0^{31}$ || 1
            else
                        GPR[rd] ← $0^{32}$
            endif
64    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                        GPR[rd] ← $0^{63}$ || 1
            else
                        GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SRA — Shift Right Arithmetic — SRA

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRA 0 0 0 0 1 1
6	5	5	5	5	6

Format:

SRA rd, rt, sa (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted right by *sa* bits, inserting signed bits into the high-order bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

Operation:

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

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

Exceptions:

None

SRAV — Shift Right Arithmetic Variable — SRAV

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

Format:

SRAV rd, rt, rs (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the low-order five bits of general-purpose register *rs*, sign-extending the high-order bits. The result is stored in the general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

Operation:

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

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

Exceptions:

None

SRL — Shift Right Logical — SRL

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRL 0 0 0 0 1 0
6	5	5	5	5	6

Format:

SRL rd, rt, sa (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

Operation:

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

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

Exceptions:

None

SRLV **Shift Right Logical Variable** SRLV

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

Format:

SRLV rd, rt, rs (MIPS I format)

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the low-order five bits of general-purpose register *rs,* inserting zeros into the high-order bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

Operation:

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

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

Exceptions:

None

SUB **Subtract** SUB

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

Format:

SUB rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*. The result is stored into general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

An Integer Overflow exception occurs if the carries-out of bits 30 and 31 differ (a two’s-complement overflow). The destination register *rd* is not modified when an Integer Overflow exception occurs.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

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

Exceptions:

Integer Overflow exception

SUBU Subtract Unsigned SUBU

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

Format:

SUBU rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*. The result is stored in general-purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

The only difference between this instruction and the SUB instruction is that SUBU never causes an Integer Overflow Exception.

Operation:

```
32    T:    GPR[rd] ← GPR[rs] – GPR[rt]

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

Exceptions:

None

SW Store Word SW

31 26	25 21	20 16	15 0
SW 1 0 1 0 1 1	base	rt	offset
6	5	5	16

Format:

SW rt, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of general-purpose register *rt* are stored in the memory location specified by the address. If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $data \leftarrow GPR[rt]_{31...0}$
        StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64  T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $data \leftarrow GPR[rt]_{31...0}$
        StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception

SWCz Store Word from Coprocessor z SWCz

31 … 26	25 … 21	20 … 16	15 … 0
SWCz 1 1 1 0 x x*	base	rt	offset
6	5	5	16

Format:

SWCz rt, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. Coprocessor register *rt* of the CPz is stored in the addressed memory. The data to be stored is defined by individual coprocessor specifications. This instruction is not valid for use with CP0.

If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

Operation:

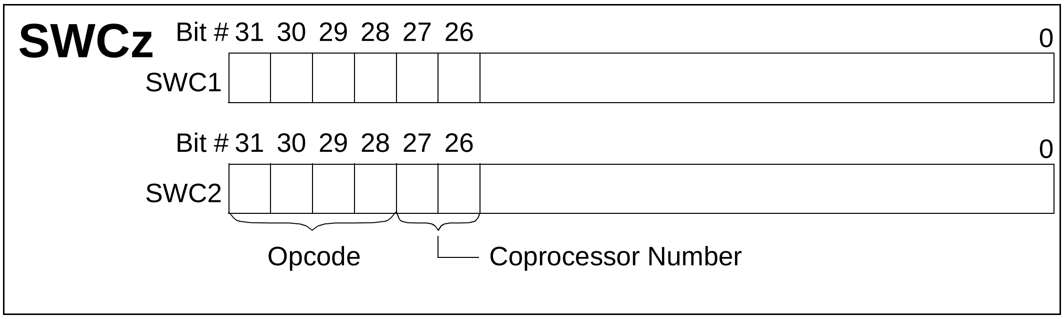
```
32    T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $(ReverseEndian \| 0^2))$
          $byte \leftarrow vAddr_{2...0}$ xor $(BigEndianCPU \| 0^2)$
          $data \leftarrow COPzSW (byte, rt)$
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64    T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}$ xor $(ReverseEndian \| 0^2))$
          $byte \leftarrow vAddr_{2...0}$ xor $(BigEndianCPU \| 0^2)$
          $data \leftarrow COPzSW (byte,rt)$
          StoreMemory (uncached, WORD, data, pAddr, vAddr DATA)
```

SWCz Store Word from Coprocessor z (continued) SWCz

Exceptions:

TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception
Coprocessor Unusable exception

Opcode Bit Encoding:



SWL Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 1 0 1 0 1 0	base	rt	offset
6	5	5	16

Format:

SWL rt, offset (base) (MIPS I format)

Description:

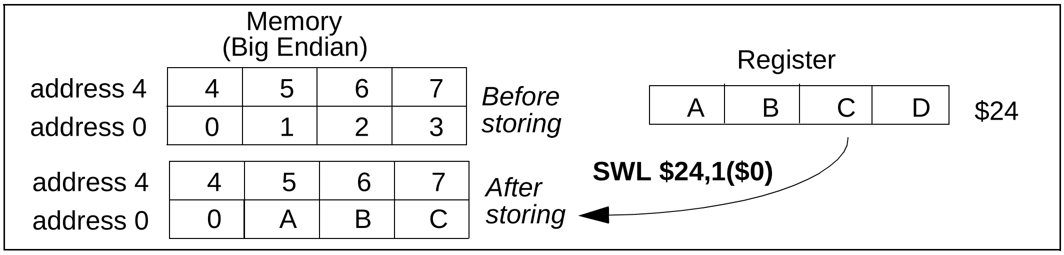
This instruction is used in combination with the SWR instruction to store a word in a register to a word in memory that is not at the word boundary. The SWL instruction stores the higher portion of the data to memory, while the SWR instruction stores the lower portion.

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address. Of the word data in the memory where the most-significant byte is specified by the generated address, only the higher portion of general-purpose register *rt* is stored to memory at the same word boundary as the target address.

Depending on the address specified, the number of bytes to be stored changes from 1 to 4.

In other words, first the most-significant byte position of general-purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the low-order byte that follows the same word boundary, the operation to store this data to the next byte of the memory is repeated.

No Address Error exceptions occur when the specified address is not located at the word boundary.



SWL Store Word Left (continued) SWL

Operation:

```
32    T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0} \text{ xor } ReverseEndian^3)$
          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)
64    T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{31...3} \| (pAddr_{2...0} \text{ xor } ReverseEndian^3)$
          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)
```

SWL — Store Word Left (continued) — SWL

The relationships between the contents given to the SWL instruction and the result (bytes for words in the memory) are shown below:

SWL

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			Offset				Offset	
$vAddr_{2...0}$	Destination	Type	LEM	BEM	Destination	Type	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

Note: *Type*: Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a double-word)

Offset: $pAddr_{2...0}$ Output to memory

LEM: Little-endian memory (BigEndianMem = 0)

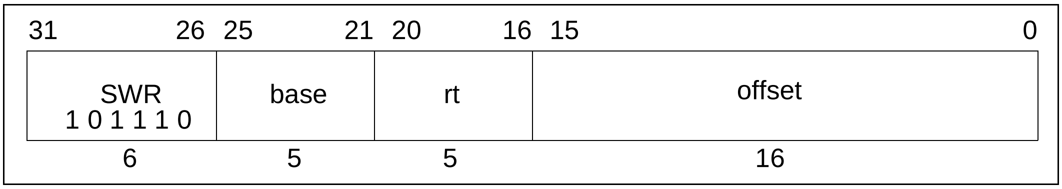
BEM: Big-endian memory (BigEndianMem = 1)

SWL — Store Word Left (continued) — SWL

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception
Reserved Instruction exception

SWR Store Word Right SWR



Format:

SWR rt, offset (base) (MIPS I format)

Description:

This instruction is used in combination with the SWL instruction to store word data in a register to a word in memory that is not at the word boundary. The SWL instruction stores the higher portion of the data to memory, while the SWR instruction stores the lower portion.

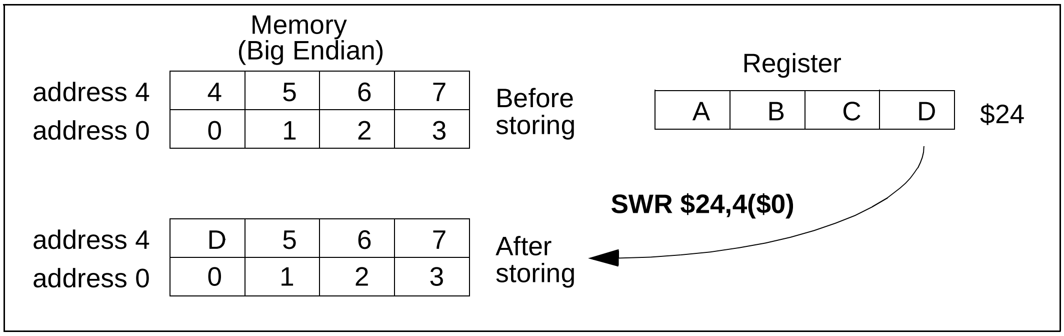
The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to generate a virtual address. Of the word data in the memory where the least-significant byte is specified by the generated address, only the lower portion of general-purpose register *rt* is stored to memory at the same word boundary as the target address. Depending on the address specified, the number of bytes to be stored changes from 1 to 4.

In other words, first the least-significant byte position of general-purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the high-order byte that follows the same word boundary, the operation to store this data to the next byte of the memory is repeated.

SWR Store Word Right SWR

(Continued)

No Address Error exceptions occur when the specified address is not located at the word boundary.



SWR Store Word Right (continued) SWR

Operation:

```
32    T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ ReverseEndian^3)$
          BigEndianMem = 0 then
              $pAddr \leftarrow pAddr_{31...2} \| 0^2$
          endif
          $byte \leftarrow vAddr_{1...0}\ xor\ BigEndianCPU^2$
          if $(vAddr_2\ xor\ BigEndianCPU) = 0$ then
              $data \leftarrow 0^{32} \| GPR[rt]_{31-8*byte...0} \| 0^{8*byte}$
          else
              $data \leftarrow GPR[rt]_{31-8*byte...0} \| 0^{8*byte} \| 0^{32}$
          endif
          Storememory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
64    T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15...0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1...3} \| (pAddr_{2...0}\ xor\ ReverseEndian^3)$
          If BigEndianMem = 0 then
              $pAddr \leftarrow pAddr_{31...2} \| 0^2$
          endif
          $byte \leftarrow vAddr_{1...0}\ xor\ BigEndianCPU^2$
          if $(vAddr_2\ xor\ BigEndianCPU) = 0$ then
              $data \leftarrow 0^{32} \| GPR[rt]_{31-8*byte...0} \| 0^{8*byte}$
          else
              $data \leftarrow GPR[rt]_{31-8*byte...0} \| 0^{8*byte} \| 0^{32}$
          endif
          StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

SWR — Store Word Right (continued) — SWR

The relationships between the register contents given to the SWR instruction and the result (bytes for words in the memory) are shown below:

SWR

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

$vAddr_{2...0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offse		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

Note:	*Type*:	Access type output to memory (refer to Table 16-3 on page 324 for information on byte access within a double-word)
	Offset:	$pAddr_{2...0}$ Output to memory
	LEM:	Little-endian memory (BigEndianMem = 0)
	BEM:	Big-endian memory (BigEndianMem = 1)

SWR — Store Word Right (continued) — SWR

Exceptions:

TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

SYNC **Synchronize** SYNC

31 26	25 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	stype	SYNC 0 0 1 1 1 1
6	15	5	6

Format:

SYNC (MIPS II format)

Description:

This instruction is provided for compatibility with MIPS implementations that implement multiprocessing facilities. The VR5432 does not implement these facilities. This instruction executes as an NOP on the VR5432.

Operation:

```
32, 64   T:   SyncOperation ()
```

Exceptions:

None

SYSCALL System Call SYSCALL

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	code	SYSCALL 0 0 1 1 00
6	20	6

Format:

SYSCALL (MIPS I format)

Description:

A System Call exception occurs after this instruction is executed, unconditionally transferring control to the exception handler.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64 T:    SystemCallException
```

Exceptions:

System Call exception

TEQ **Trap If Equal** TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TEQ 1 1 0 1 0 0
6	5	5	10	6

Format:

TEQ rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with those of general-purpose register *rs*. If the contents of general-purpose register *rs* are equal to the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64      T:   if GPR[rs] = GPR[rt] then
                     TrapException
                 endif
```

Exceptions:

Trap exception

TEQI **Trap If Equal Immediate** TEQI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TEQI 0 1 1 0 0	immediate
6	5	5	16

Format:

TEQI rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. If the contents of general-purpose register *rs* are equal to the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:    if GPR[rs] = $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$ then
                TrapException
            endif

64    T:    if GPR[rs] = $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
                TrapException
            endif
```

Exceptions:

Trap exception

TGE — Trap If Greater Than or Equal — TGE

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGE 1 1 0 0 0 0
6	5	5	10	6

Format:

TGE rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with the contents of general-purpose register *rs*. Interpreting both register contents as signed integers, if the contents of general-purpose register *rs* are greater than or equal to the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64  T:  if GPR[rs] ≥ GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TGEI — Trap If Greater Than or Equal Immediate — TGEI

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	TGEI 0 1 0 0 0	immediate
6	5	5	16

Format:

TGEI rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. Interpreting both values as signed integers, if the contents of general-purpose register *rs* are greater than or equal to the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:  if $GPR[rs] \geq (immediate_{15})^{16} || immediate_{15...0}$ then
              TrapException
          endif
64    T:  if $GPR[rs] \geq (immediate_{15})^{48} || immediate_{15...0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TGEIU — Trap If Greater Than or Equal Immediate Unsigned — TGEIU

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 0 0 0 0 0 1	rs	TGEIU 0 1 0 0 1	immediate
6	5	5	16

Format:

TGEIU rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. Interpreting both values as unsigned integers, if the contents of general-purpose register *rs* are greater than or equal to the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:  if (0 || GPR[rs]) $\geq$ (0 || $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$) then
              TrapException
          endif
64    T:  if (0 || GPR[rs]) $\geq$ (0 || $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$) then
              TrapException
          endif
```

Exceptions:

Trap exception

TGEU — Trap If Greater Than or Equal Unsigned — TGEU

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGEU 1 1 0 0 0 1
6	5	5	10	6

Format:

TGEU rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with the contents of general-purpose register *rs*. Interpreting both values as unsigned integers, if the contents of general-purpose register *rs* are greater than or equal to the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:    if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
                     TrapException
                endif
```

Exceptions:

Trap exception

TLBP Probe TLB for Matching Entry TLBP

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBP 0 0 1 0 0 0
6	1	19	6

Format:

TLBP (MIPS I format)

Description:

Searches for a TLB entry that matches with the contents of the EntryHI register and stores an index for that TLB entry in the Index register. If a TLB entry that matches is not found, sets the most-significant bit of the Index register.

Memory references by the instruction immediately after a TLBP instruction, or in cases in which more than one TLB entry is a hit, are undefined.

Operation:

```
32   T:   $Index \leftarrow 1 \| 0^{25} \| Undefined^{6}$
          for i in 0...TLBEntries – 1
               if $(TLB[i]_{95...77} = EntryHi_{31...13})$ and $(TLB[i]_{76}$ or
               $(TLB[i]_{71...64} = EntryHi_{7...0}))$ then
                    $Index \leftarrow 0^{26} \| i_{5...0}$
               endif
          endfor
64   T:   $Index \leftarrow 1 \| 0^{25} \| Undefined^{6}$
          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

TLBR Read Indexed TLB Entry TLBR

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBR 0 0 0 0 0 1
6	1	19	6

Format:

TLBR (MIPS I format)

Description:

The EntryHi and EntryLo registers are loaded with the contents of the TLB entry selected by the contents of the Index register. The *G* bit (which controls ASID matching) read from the TLB is written into both of the EntryLo0 and EntryLo1 registers.

The operation is invalid (and the results are undefined) if the contents of the Index register are greater than the number of TLB entries in the processor.

Operation:

```
32    T:  $PageMask \leftarrow TLB[Index_{5...0}]_{127...96}$
          $EntryHi \leftarrow TLB[Index_{5...0}]_{95...64}$ and not $TLB[Index_{5...0}]_{127...96}$
          $EntryLo1 \leftarrow TLB[Index_{5...0}]_{63...33} \| TLB[Index_{5...0}]_{76}$
          $EntryLo0 \leftarrow TLB[Index_{5...0}]_{31...1} \| TLB[Index_{5...0}]_{76}$
64    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

TLBWI — Write Indexed TLB Entry — TLBWI

31 … 26	25	24 … 6	5 … 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBWI 0 0 0 0 1 0
6	1	19	6

Format:

TLBWI (MIPS I format)

Description:

The TLB entry selected by the Index register is loaded with the contents of the EntryHi and EntryLo registers. The *G* bit of the TLB is written with the logical AND of the *G* bits in the EntryLo0 and EntryLo1 registers.

The operation is invalid (and the results are undefined) if the value in the Index register is greater than the number of TLB entries in the processor.

Operation:

```
32, 64   T:   TLB[Index$_{5...0}$] ←
                   PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor Unusable exception

TLBWR — Write Random TLB Entry — TLBWR

31 … 26	25	24 … 6	5 … 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBWR 0 0 0 1 1 0
6	1	19	6

Format:

TLBWR (MIPS I format)

Description:

The TLB entry selected by the Random register is loaded with the contents of the EntryHi and EntryLo registers. The *G* bit of the TLB is written with the logical AND of the *G* bits in the EntryLo0 and EntryLo1 registers.

Operation:

```
32, 64   T:   TLB[Random$_{5...0}$] $\leftarrow$
                 PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor Unusable exception

TLT — Trap If Less Than — TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLT 1 1 0 0 1 0
6	5	5	10	6

Format:

TLT rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with general-purpose register *rs*. Interpreting both values as signed integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64  T:  if GPR[rs] < GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TLTI **Trap If Less Than Immediate** TLTI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TLTI 0 1 0 1 0	immediate
6	5	5	16

Format:

TLTI rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. Interpreting both values as signed integers, if the contents of general-purpose register *rs* are less than the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:  if GPR[rs] < $(\text{immediate}_{15})^{16} \| \text{immediate}_{15...0}$ then
              TrapException
          endif
64    T:  if GPR[rs] < $(\text{immediate}_{15})^{48} \| \text{immediate}_{15...0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TLTIU Trap If Less Than Immediate Unsigned TLTIU

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TLTIU 0 1 0 1 1	immediate
6	5	5	16

Format:

TLTIU rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. Interpreting both values as unsigned integers, if the contents of general-purpose register *rs* are less than the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:   if $(0 || GPR[rs]) < (0 || (immediate_{15})^{16} || immediate_{15...0})$ then
                TrapException
           endif
64    T:   if $(0 || GPR[rs]) < (0 || (immediate_{15})^{48} || immediate_{15...0})$ then
                TrapException
           endif
```

Exceptions:

Trap exception

TLTU — Trap If Less Than Unsigned — TLTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLTU 1 1 0 0 1 1
6	5	5	10	6

Format:

TLTU rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with general-purpose register *rs*. Interpreting both values as unsigned integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64 T:   if (0 || GPR[rs]) < (0 || GPR[rt]) then
                TrapException
            endif
```

Exceptions:

Trap exception

TNE **Trap If Not Equal** TNE

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TNE 1 1 0 1 1 0
6	5	5	10	6

Format:

TNE rs, rt (MIPS II format)

Description:

The contents of general-purpose register *rt* are compared with those of general-purpose register *rs*. If the contents of general-purpose register *rs* are not equal to the contents of general-purpose register *rt*, a Trap exception occurs.

The *code* field is available for transferring parameters to the exception handler. The parameter is retrieved by the exception handler only by loading as data the contents of the memory word containing the instruction.

Operation:

```
32, 64 T:   if GPR[rs] ≠ GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TNEI Trap If Not Equal Immediate TNEI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TNEI 0 1 1 1 0	immediate
6	5	5	16

Format:

TNEI rs, immediate (MIPS II format)

Description:

The 16-bit *immediate* is sign extended and compared with the contents of general-purpose register *rs*. If the contents of general-purpose register *rs* are not equal to the sign-extended *immediate*, a Trap exception occurs.

Operation:

```
32    T:  if GPR[rs] $\neq$ $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$ then
              TrapException
          endif
64    T:  if GPR[rs] $\neq$ $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

XOR **Exclusive OR** XOR

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

Format:

XOR rd, rs, rt (MIPS I format)

Description:

The contents of general-purpose register *rs* are bitwise ORed with the contents of general-purpose register *rt*. The result is stored into general-purpose register *rd.*

Operation:

```
32, 64 T:    GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

XORI Exclusive OR Immediate XORI

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

Format:

XORI rt, rs, immediate (MIPS I format)

Description:

The 16-bit zero-extended *immediate* is bitwise ORed with the contents of general-purpose register *rs*. The result is stored in general-purpose register *rt*.

Operation:

```
32    T:    GPR[rt] ← GPR[rs] xor ($0^{16}$ || immediate)
64    T:    GPR[rt] ← GPR[rs] xor ($0^{48}$ || immediate)
```

Exceptions:

None

17.5 CPU Instruction Opcode Bit Encoding

Figure 17-1 and Figure 17-2 list the VR5432 opcode bit encoding.

Opcode

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIe	DADDIUe	LDLe	LDRe	DEBUG	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUe
5	SB	SH	SWL	SW	SDLe	SDRe	SWR	CACHE d
6	LL	LWC1	LWC2	*	LLDe	LDC1	LDC2	LDe
7	SC	SWC1	SWC2	*	SCDe	SDC1	SDC2	SDe

SPECIAL function

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRLp	SRA	SLLV	*	SRLVp	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVe	*	DSRLVep	DSRAVe
3	MULTp	MULTUp	DIV	DIVU	DMULTe	DMULTUe	DDIVe	DDIVUe
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDe	DADDUe	DSUBe	DSUBUe
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLe	*	DSRLep	DSRAe	DSLL32e	*	DSRL32ep	DSRA32e

REGIMM rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

COPz rs

25, 24 \ 23...21	0	1	2	3	4	5	6	7
0	MF	DMFe	CF	g	MT	DMTe	CT	g
1	BC	g	g	g	g	g	g	g
2	CO							
3								

Figure 17-1 VR5432 Opcode Bit Encoding (1 of 2)

COPz rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

CP0 Function

5 ... 3 \ 2 ... 0	0	1	2	3	4	5	6	7
0	ϕ	TLBR	TLBWI	ϕ	ϕ	ϕ	TLBWR	ϕ
1	TLBP	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
2	ξ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
3	ERET χ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
4	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
5	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

DEBUG Function

5 ... 3 \ 2 ... 0	0	1	2	3	4	5	6	7
0	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
1	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
2	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
3	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
4	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
5	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	ϕ	MF/TDR	DRET	DBREAK

Figure 17-2 VR5432 Opcode Bit Encoding (2 of 2)

Key:

*	If the operation code marked with an asterisk is executed, the Reserved Instruction exception occurs. These codes are reserved for future expansion.
γ	Operation codes marked with a gamma cause a Reserved Instruction exception. They are reserved for future expansion

δ	Operation codes marked with a delta are valid only with CP enabled and cause a Reserved Instruction exception on other processors.
ϕ	Operation codes marked with a phi are invalid but do not caus Reserved Instruction exceptions
ξ	Operation codes marked with a xi cause a Reserved Instructio exception.
χ	Operation codes marked with a chi are valid only on VR4000 and VR5000 processors.
ε	Operation codes marked with an epsilon are valid in the 64-bit mode and 32-bit Kernel mode. In the 32-bit User or Supervisor mode, these codes generate the Reserved Instruction exception.
π	Operation codes marked with a pi have been used for the implementation-specific instruction set extensions on th VR5432, specifically the Multiply-Accumulate instructions and the Rotate instructions

Floating-Point Unit Instruction Set

18

This chapter provides a detailed description of each Floating-Point Unit (FPU) instruction. (For a general overview of VR5432 instructions, see Chapter 16.)

18.1 Instruction Formats

The instruction description subsections that follow show how the three basic instruction formats (I-, R-, and J-type) are used by:

- Load and Store instructions
- Transfer instructions
- Floating-point arithmetic instructions
- Floating-point Branch instruction

Floating-Point instructions are mapped onto the MIPS coprocessor instructions, defining Coprocessor one (CP1) as the floating-point unit.

Each operation is valid only for certain formats. Implementations may support some of these formats and operations through emulation, but they only need to support combinations that are valid (marked V in Table 18-1). Combinations marked R (reserved) in Table 18-1 are not currently specified by this architecture, and cause an Unimplemented Instruction exception. They are reserved for future extensions of the architecture.

Table 18-1 Valid FPU Instruction Forma t

Operation	**Source Format**			
	Single	**Double**	**Word**	**Longword**
ADD	V	V	R	R
SUB	V	V	R	R
MUL	V	V	R	R
DIV	V	V	R	R
SQRT	V	V	R	R
ABS	V	V	R	R
MOV	V	V		
NEG	V	V	R	R
TRUNC.L	V	V		
ROUND.L	V	V		
CEIL.L	V	V		
FLOOR.L	V	V		
TRUNC.W	V	V		
RECIP	V	V		
ROUND.W	V	V		
RSQRT	V	V		
CEIL.W	V	V		
FLOOR.W	V	V		
CVT.S		V	V	V
CVT.D	V		V	V
CVT.W	V	V		
CVT.L	V	V		
C	V	V	R	R

The FPU Branch instruction can be used with the logic of the condition reversed, so it is only necessary to provide half of the 32 comparison predicates and relations required by the IEEE-754 standard. A four-bit field in the C instruction (comparison) specifies one of the 16 conditions shown in the “True” column of Table 18-2. Inverting the sense of the condition in the Branch instruction provides the 16 conditions shown in the “False” column. Unordered conditions result when one of the operands is a NaN (i.e., a “Not a Number” encoding), which has no numerical order when compared to a number or another NaN.

Table 18-2 Logical Reverse of Predicates by Condition True/False

Condition			**Relations**				**Invalid Operation Exception if Unordered**
Mnemonic		**Code**	**Greater Than**	**Less Than**	**Equal**	**Unordered**	
True	**False**						
F	T	0	F	F	F	F	No
UN	OR	1	F	F	F	T	No
EQ	NEQ	2	F	F	T	F	No
UEQ	OGL	3	F	F	T	T	No
OLT	UGE	4	F	T	F	F	No
ULT	OGE	5	F	T	F	T	No
OLE	UGT	6	F	T	T	F	No
ULE	OGT	7	F	T	T	T	No
SF	ST	8	F	F	F	F	Yes
NGLE	GLE	9	F	F	F	T	Yes
SEQ	SNE	10	F	F	T	F	Yes
NGL	GL	11	F	F	T	T	Yes
LT	NLT	12	F	T	F	F	Yes
NGE	GE	13	F	T	F	T	Yes
LE	NLE	14	F	T	T	F	Yes
NGT	GT	15	F	T	T	T	Yes

F: False
T: True

18.1.1 Floating-Point Loads, Stores, and Transfers

All movement of data between the floating-point unit (FPU) and memory is accomplished by load and store operations, which reference the Floating-Point General-Purpose registers (FGRs). These operations are unformatted; no format conversions are performed and, therefore, no floating-point exceptions can be generated by these operations.

Data may also be directly moved between the floating-point unit and the processor by Move to Coprocessor (MTC) and Move from Coprocessor (MFC) instructions. Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

In addition, two Floating-Point Control registers (FCRs) are provided for FPU control bits, status bits, implementation level, and revision level. These registers can only be accessed by the CTC1 and CFC1 instructions.

18.1.2 Floating-Point Operations

The floating-point unit instruction set includes:

- Floating-point Add instructions
- Floating-point Subtract instruction
- Floating-point Multiply instruction
- Floating-point Divide instruction
- Floating-point Square Root instructions
- Floating-point Reciprocal instruction
- Floating-point Reciprocal Square Root instructions
- Conversion between fixed-point and floating-point format
- Conversion between floating-point format
- Floating-point Compare instructions

These operations satisfy the requirements of the IEEE-754 standard for accuracy. Specifically, these operations obtain a result identical to an infinite-precision result rounded to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for conversion functions, mixed-format operations cannot be performed.

In the VR5432 implementation, the instruction immediately following a load may use the contents of the register being loaded. In such cases, the hardware interlocks by the number of cycles required for reading. Scheduling load delay slots is not required for functional code; however, it still may be desirable for highest performance or compatibility with the VR3000 device (which lacks interlocks).

Load and Store instruction behavior depends on FGR width.

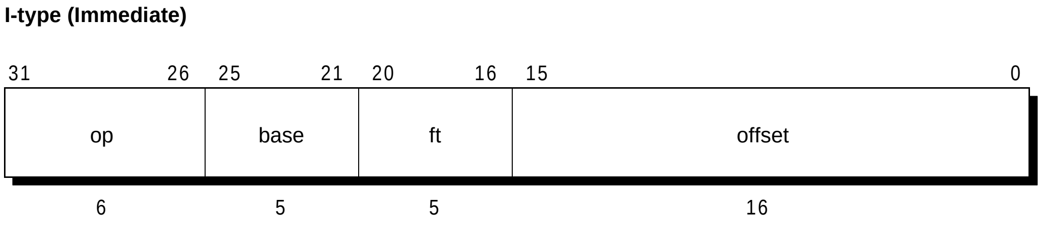
- When the *FR* bit in the Status register is clear, the Floating-Point General-Purpose registers (FGRs) are 32 bits wide.
- To hold single-precision floating-point format data, sixteen even-numbered registers out of 32 FGRs are available.
- To hold double-precision floating-point format data, the 32-bit registers are used in pairs as 16 64-bit registers
- When the *FR* bit in the Status register is set, the FGRs are 64 bits wide.
- To hold single-precision floating-point format data, the low half of 32 FGRs are used.
- To hold double-precision floating-point format data, 32 FGRs are used.

In the load and store operation descriptions, the functions listed in Table 18-3 are used to represent the handling of virtual addresses and physical memory.

Table 18-3 FPU Load/Store Instructions Using Registe r +Register Addressing

Mnemonic	**Description**	**Defined in MIPS...**
LWXC1	Load Word Indexed to FPU	IV
SWXC1	Store Word Indexed from FPU	IV
LDXC1	Load Doubleword Indexed to FPU	IV
SDXC1	Store Doubleword Indexed from FPU	IV

Figure 18-1 shows the I-type instruction format used by Load and Store instructions.



op:	6-bit opcod
base:	5-bit base register specifier
ft:	5-bit source (for stores) or destination (for loads) FPU register specifie
offset:	16-bit signed immediate offset

Figure 18-1 Load and Store Instruction Format

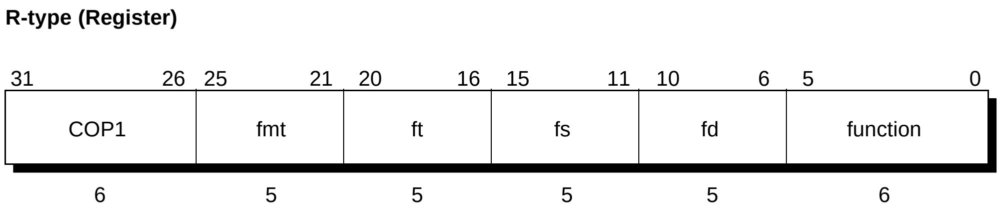
All coprocessor loads and stores reference data that is located at word boundaries. For word loads and stores, the access type field is always *word*, and the low-order two bits of the address must always be zero. For doubleword loads and stores, the access type field is always *doubleword*, and the low-order three bits of the address must always be zero.

Regardless of byte-numbering order, the address specifies the byte that has the smallest byte address in the accessed field. For a big-endian system, this is the left-most byte; for a little-endian system, this is the right-most byte.

18.2 Floating-Point Computational Instructions

Computational instructions include all of the floating-point arithmetic operations performed by the FPU.

Figure 18-2 shows the R-type instruction format used for computational instructions.



COP1:	6-bit opcode
fmt:	5-bit format specifie
fs:	5-bit source 1 register
ft:	5-bit source 2 register
fd:	5-bit destination register
function:	6-bit function field

Figure 18-2 Computational Instruction Format

The *function* field indicates the floating-point operation to be performed.

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by the 5-bit format field (*fmt*); decoding for this field is shown in Table 18-4.

Table 18-4 Format Field Decoding

Code	Mnemonic	Size	Format
0–15	Reserved		
16	S	Single (32 bits)	Binary floating-point
17	D	Double (64 bits)	Binary floating-point
18	Reserved		
19	Reserved		
20	W	32 bits	Binary fixed-point
21	L	64 bits	Binary fixed-point
22–31	Reserved		

Table 18-5 lists all floating-point computational instructions.

Table 18-5   Floating-Point Computational Instructions and Operations

Code (5:0)	**Mnemonic**	**Operation**
0	ADD	Add
1	SUB	Subtract
2	MUL	Multiply
3	DIV	Divide
4	SQRT	Square root
5	ABS	Absolute value
6	MOV	Transfer
7	NEG	Sign reverse
8	ROUND.L	Convert to 64-bit fixed-point, rounded to nearest even number
9	TRUNC.L	Convert to 64-bit fixed-point, rounded toward zero
10	CEIL.L	Convert to 64-bit fixed-point, rounded to $+\infty$
11	FLOOR.L	Convert to 64-bit fixed-point, rounded to $-\infty$
12	ROUND.W	Convert to 32-bit fixed-point, rounded to nearest even number
13	TRUNC.W	Convert to 32-bit fixed-point, rounded toward zero
14	CEIL.W	Convert to 32-bit fixed-point, rounded to $+\infty$
15	FLOOR.W	Convert to 32-bit fixed-point, rounded to $-\infty$
16–31	—	Reserved
32	CVT.S	Convert to single floating-point
33	CVT.D	Convert to double floating-point
34	—	Reserved
35	—	Reserved
36	CVT.W	Convert to 32-bit fixed-point
37	CVT.L	Convert to 64-bit fixed-point
38–47	—	Reserved
48–63	C	Floating-Point Compare

The following routines are used in the description of the floating-point operations to retrieve the value of an FPR or to change the value of an FGR:

32-Bit Mode

```
value <-- ValueFPR(fpr, fmt)
    /* undefined for odd fpr */
    case fmt of
        S, W:
            value <-- FGR[fpr+0]
        D:
            value <-- FGR[fpr+1] || FGR[fpr+0]
    end

StoreFPR(fpr, fmt, value):
    /* undefined for odd fpr */
    case fmt of
        S, W:
            FGR[fpr+1] <-- undefined
            FGR[fpr+0] <-- value
        D:
            FGR[fpr+1] <-- value63...32
            FGR[fpr+0] <-- value31...0
    end
```

64-Bit Mode

```
value <-- ValueFPR(fpr, fmt)
    case fmt of
        S, W:
            value <-- FGR[fpr]31...0
        D, L:
            value <-- FGR[fpr]
    end

StoreFPR(fpr, fmt, value):
    case fmt of
        S, W:
            FGR[fpr] <-- undefined32 || value
        D, L:
            FGR[fpr] <-- value
    end
```

18.3 FPU Instructions

This section describes in detail the FPU instructions.

Exceptions that may occur are listed at the end of each instruction's description. For details regarding FPU exceptions and exception processing, refer to Chapter 8.

ABS.fmt **Floating-Point Absolute Value** ABS.fm

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ABS 0 0 0 1 0 1
6	5	5	5	5	6

Format:

ABS.fmt fd, fs (MIPS I format)

Description:

The absolute value of the contents of floating-point register *fs* is taken and stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

The absolute value operation is arithmetically performed. If the operand is NaN, therefore, the Invalid Operation exception occurs.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined.

If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64 T:     StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt) ) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception

ADD.fmt Floating-Point Add ADD.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	ADD 0 0 0 0 0 0
6	5	5	5	5	6

Format:

ADD.fmt fd, fs, ft (MIPS I format)

Description:

The contents of floating-point registers *fs* and *ft* are added and the result is stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*. The operation is executed as if the accuracy were infinite, and the result is rounded according to the current rounding mode.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) + ValueFPR (ft, fmt) )
```

ADD.fmt **Floating-Point Add (continued)** ADD.fmt

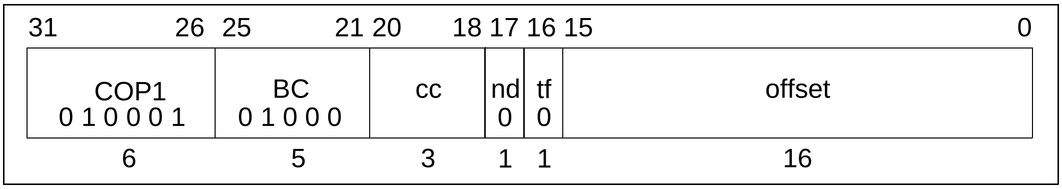
Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Inexact Operation exception
Overflow exception
Underflow exception

BC1F **Branch on FPU False (Coprocessor 1)** BC1F



Format:

BC1F offset	(MIPS I format, *cc* = 0 is implied)
BC1F cc, offset	(MIPS IV format)

Description:

An 18-bit signed *offset* (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the floating-point condition code bit *cc* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

A floating-point condition code is set by the floating-point Compare instruction, C.cond.fmt.

The MIPS I architecture defines a single floating-point condition code, implemented as the Coprocess or1 condition signal (*Cp1Cond*) and the *C* bit in the FCR31 register. MIP SI, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the “Format” section above. Both assembler formats are valid for MIPS IV.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. Floating-Point Compare and Conditional Branch instructions specify the condition code bit to set or test. The condition code bit specified by the *cc* field is modified by the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

In the MIPS I, II, and III implementations, there must be at least one instruction between the Compare instruction that sets the condition code and the Branch instruction that tests it. Hardware does not detect a violation of this restriction. In the MIPS IV instruction set, this restriction has been removed.

BC1F Branch on FPU False (Coprocessor 1) (continued) BC1F

Operation:

```
MIPS I, II, and III:
    T-1:  condition $\leftarrow$ FPConditionCode(0) = 0
    T:    target_offset $\leftarrow$ $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC $\leftarrow$ PC + target_offset
          endif
MIPS IV:
    T:    condition $\leftarrow$ FPConditionCode(cc) = 0
          target_offset $\leftarrow$ $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC $\leftarrow$ PC + target_offset
          endif
```

Note: With the 18-bit signed instruction offset, the conditional branch range is ±128K. Use the Jump (J) or Jump Register (JR) instructions to branch to addresses outside this range.

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

BC1FL **Branch on FPU False Likely (Coprocessor 1)** BC1FL

31 26	25 21	20 18	17	16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	cc	nd 1	tf 0	offset
6	5	3	1	1	16

Format:

BC1FL offset	(MIPS I format, *cc* = 0 is implied)
BC1FL cc, offset	(MIPS IV format)

Description:

An 18-bit signed *offset* (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the floating-point condition code bit *cc* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

A floating-point condition code is set by the Floating-Point Compare instruction, C.cond.fmt.

The MIPS I architecture defines a single floating-point condition code, implemented as the Coprocess or1 condition signal (*Cp1Cond*) and the *C* bit in the FCR31 register. MIP SI, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the “Format” section above. Both assembler formats are valid for MIPS IV.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. Floating-Point Compare and Conditional Branch instructions specify the condition code bit to set or test. The condition code bit specified by the *cc* field is modified by the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

In the MIPS I, II, and III implementations, there must be at least one instruction between the Compare instruction that sets the condition code and the Branch instruction that tests it. Hardware does not detect a violation of this restriction. In the MIPS IV instruction set, this restriction has been removed.

BC1FL — Branch on FPU False Likely (Coprocessor 1) (continued) — BC1FL

Operation:

```
MIPS I, II, and III:
    T-1:  condition ← FPConditionCode(0) = 0
    T:    target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC ← PC + target_offset
          else
                  NullifyCurrentInstruction()
          endif
MIPS IV:
    T:    condition ← FPConditionCode(cc) = 0
          target_offset ← $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC ← PC + target_offset
          else
                  NullifyCurrentInstruction()
          endif
```

Note: Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, users are encouraged to use the BC1F instruction instead.

BC1FL Branch on FPU False Likely (Coprocessor 1) (continued) BC1FL

Note: With the 18-bit signed instruction offset, the conditional branch range is ±128K. Use Jump (J) or Jump Register (JR) instructions to branch to addresses outside this range.

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

BC1T — Branch on FPU True (Coprocessor 1) — BC1T

31 26	25 21	20 18	17	16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	cc	nd 0	tf 1	offset
6	5	3	1	1	16

Format:

BC1T offset	(MIPS I format, *cc* = 0 is implied)
BC1T cc, offset	(MIPS IV format)

Description:

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the floating-point condition code bit *cc* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

A floating-point condition code is set by the floating-point Compare instruction, C.cond.fmt.

The MIPS I architecture defines a single floating-point condition code, implemented as the Coprocess or1 condition signal (*Cp1Cond*) and the *C* bit in the FCR31 register. MIP SI, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the “Format” section above. Both assembler formats are valid for MIPS IV.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. Floating-Point Compare and Conditional Branch instructions specify the condition code bit to set or test. The condition code bit specified by the *cc* field is modified by the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

In the MIPS I, II, and III implementations, there must be at least one instruction between the Compare instruction that sets the condition code and the Branch instruction that tests it. Hardware does not detect a violation of this restriction. In the MIPS IV instruction set, this restriction has been removed.

BC1T Branch on FPU True (Coprocessor 1) (continued) BC1T

Operation:

```
MIPS I, II, and III:
        T-1:  condition ← FPConditionCode(0) = 1
        T:    target_offset ← $(\text{offset}_{15})^{\text{GPRLEN}-(16+2)}$ || offset || $0^2$
        T+1:  if condition then
                        PC ← PC + target_offset
              endif
MIPS IV:
        T:    condition ← FPConditionCode(cc) = 1
              target_offset ← $(\text{offset}_{15})^{\text{GPRLEN}-(16+2)}$ || offset || $0^2$
        T+1:  if condition then
                        PC ← PC + target_offset
              endif
```

Note: With the 18-bit signed instruction offset, the conditional branch range is ± 128K. Use Jump (J) or Jump Register (JR) instructions to branch to addresses outside this range.

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

BC1TL Branch on FPU True Likely (Coprocessor 1) BC1TL

31 26	25 21	20 18	17	16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	cc	nd 1	tf 1	offset
6	5	3	1	1	16

Format:

BC1TL offset	(MIPS I format, *cc* = 0 is implied)
BC1TL cc, offset	(MIPS IV format)

Description:

An 18-bit signed *offset* (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the floating-point condition code bit *cc* is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

A floating-point condition code is set by the floating-point Compare instruction, C.cond.fmt.

The MIPS I architecture defines a single floating-point condition code, implemented as the Coprocess or1 condition signal (*Cp1Cond*) and the *C* bit in the FCR31 register. MIP SI, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the “Format” section above. Both assembler formats are valid for MIPS IV.

The MIPS IV architecture adds seven more condition code bits to the original condition code 0. Floating-Point Compare and Conditional Branch instructions specify the condition code bit to set or test. The condition code bit specified by the *cc* field is modified by the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for *tf* and *nd*.

In the MIPS I, II, and III implementations, there must be at least one instruction between the Compare instruction that sets the condition code and the Branch instruction that tests it. Hardware does not detect a violation of this restriction. In the MIPS IV instruction set, this restriction has been removed.

BC1TL Branch on FPU True Likely (Coprocessor 1) (continued) BC1TL

Operation:

```
MIPS I, II, and III:
    T-1:  condition $\leftarrow$ FPConditionCode(0) = 1
    T:    target_offset $\leftarrow$ $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC $\leftarrow$ PC + target_offset
          else
                  NullifyCurrentInstruction()
          endif
MIPS IV:
    T:    condition $\leftarrow$ FPConditionCode(cc) = 1
          target_offset $\leftarrow$ $(\text{offset}_{15})^{\text{GPRLEN-(16+2)}}$ || offset || $0^2$
    T+1:  if condition then
                  PC $\leftarrow$ PC + target_offset
          else
                  NullifyCurrentInstruction()
          endif
```

Note: Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, users are encouraged to use the BC1T instruction instead.

BC1TL Branch on FPU True Likely (Coprocessor 1) (continued) BC1TL

Note: With the 18-bit signed instruction offset, the conditional branch range is ± 128K. Use Jump (J) or Jump Register (JR) instructions to branch to addresses outside this range.

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

C.cond.fmt — Floating-Point Compare — C.cond.fmt

31 26	25 21	20 16	15 11	10 8	7 6	5 4	3 0
COP1 0 1 0 0 0 1	fmt	ft	fs	cc	0 0 0	FC 1 1	cond
6	5	5	5	3	2	2	4

Format:

C.cond.fmt fs, ft	(MIPS I format, *cc* = 0 is implied)
C.cond.fmt cc, fs, ft	(MIPS IV format)

Description:

The value in floating-point register *fs* is compared to the value in floating-point register *ft*; the values are in format *fmt*. The comparison is exact and neither overflows nor underflows.

If the comparison specified by $cond_{2..1}$ is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code *cc*; true is 1 and false is 0.

If $cond_3$ is set and at least one of the values is a NaN, an Invalid Operation condition is raised; the result depends on the Floating-Point exception model currently active:

Precise exception model: The Invalid Operation flag is set in the FCR31 register. If the *Invalid Operation Enable* bit is set in the FCR31 register, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code *cc*.

Imprecise exception model (R8000® normal mode): The Boolean result is written into condition code *cc*. No FCR31 register flag is set. If th *Invalid Operation Enabl* bit is set in the FCR31 register, an Invalid Operation exception is taken, imprecisely, at some future time

There are four mutually exclusive ordering relations for comparing floating-point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than,* and *equal*. In addition, the IEEE floating-point standard defines the relation *unordered,* which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals –0.

C.cond.fmt — Floating-Point Compare (continued) — C.cond.fmt

The comparison condition is a logical predicate, or equation, of the ordering relations such as *less than or equal*, *equal*, *not less than*, or *unordered or equal*. Compare distinguishes among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two floating-point values in the equation. If the *equal* relation is true, for example, then all four example predicates above yield a true result. If the *unordered* relation is true, then only the final predicate, *unordered or equal*, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates, as shown in Table 18-6. Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, *compare* tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

The truth of the second predicate is the logical negation of the instruction result. After a Compare instruction, a test for the truth of the first predicate can be made with the Branch on FPU True (BC1T) instruction and the truth of the second can be made with the Branch on FPU False (BC1F) instruction.

C.cond.fmt **Floating-Point Compare (continued)** C.cond.fmt

Table 18-7 shows another set of eight compare operations, distinguished by a $cond_3$ value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including a Quiet NaN (QNaN), then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the *FCSR*, an Invalid Operation exception occurs.

C.cond.fmt **Floating-Point Compare (continued)** C.cond.fmt

Table 18-6 FPU Comparisons Without Special Operand Exceptions

Instruction	Comparison Predicate					Comparison CC Result		Instruction Condition Field	
Condition Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation	Relation Values				If Predicate Is True	Inv. Op. Excp. if QNaN?		
		>	<	=	?			3	2..0
F	False [this predicate is always False]	F	F	F	F	F	No	0	0
	True (T)	T	T	T	T				
UN	Unordered	F	F	F	T	T			1
	Ordered (OR)	T	T	T	F	F			
EQ	Equal	F	F	T	F	T			2
	Not Equal (NEQ)	T	T	F	T	F			
UEQ	Unordered or Equal	F	F	T	T	T			3
	Ordered or Greater Than or Less Than (OGL)	T	T	F	F	F			
OLT	Ordered or Less Than	F	T	F	F	T			4
	Unordered or Greater Than or Equal (UGE)	T	F	T	T	F			
ULT	Unordered or Less Than	F	T	F	T	T			5
	Ordered or Greater Than or Equal (OGE)	T	F	T	F	F			
OLE	Ordered or Less Than or Equal	F	T	T	F	T			6
	Unordered or Greater Than (UGT)	T	F	F	T	F			
ULE	Unordered or Less Than or Equal	F	T	T	T	T			7
	Ordered or Greater Than (OGT)	T	F	F	F	F			

Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = *False*

C.cond.fmt Floating-Point Compare (continued) C.cond.fmt

Table 18-7 FPU Comparisons With Special Operand Exceptions for QNaN

Instruction	Comparison Predicate					Comparison CC Result		Instruction Condition Field	
Condition Mnemonic	Name of Predicate and Logically Negated Predicate (Abbreviation	Relation Values >	<	=	?	If Predicate Is True	Inv. Op. Excp. if QNaN?	3	2..0
SF	Signaling False [this predicate is always False]	F	F	F	F	F	Yes	1	0
	Signaling True (ST)	T	T	T	T				
NGLE	Not Greater Than or Less Than or Equal	F	F	F	T	T			1
	Greater Than or Less Than or Equal (GLE)	T	T	T	F	F			
SEQ	Signaling Equal	F	F	T	F	T			2
	Signaling Not Equal (SNE)	T	T	F	T	F			
NGL	Not Greater Than or Less Than	F	F	T	T	T			3
	Greater Than or Less Than (GL)	T	T	F	F	F			
LT	Less Than	F	T	F	F	T			4
	Not Less Than (NLT)	T	F	T	T	F			
NGE	Not Greater Than or Equal	F	T	F	T	T			5
	Greater Than or Equal (GE)	T	F	T	F	F			
LE	Less Than or Equal	F	T	T	F	T			6
	Not Less Than or Equal (NLE)	T	F	F	T	F			
NGT	Not Greater Than	F	T	T	T	T			7
	Greater Than (GT)	T	F	F	F	F			

Key: ? = *unordered*, > = *greater than*, < = *less than*, = is *equal*, T = *True*, F = *False*

C.cond.fmt **Floating-Point Compare (continued)** C.cond.fmt

The instruction encoding is an extension made in the MIP SIV architecture. In previous architecture levels, the *cc* field for this instruction must equal 0.

The MIPS I architecture defines a single floating-point condition code, implemented as the Coprocess or1 condition signal (*Cp1Cond*) and the *C* bit in the FCR31 register. MIP SI, II, and III architectures must have the *cc* field set to 0, which is implied by the first format in the "Format" section. Both assembler formats are valid for MIPS IV.

The MIPS IV architecture adds seven more condition code bits to t he original condition code 0. Floating-Point Compare and Conditional Branch instructions specify the condition code bit to set or test.

The fields *fs* and *ft* must specify FPRs valid for operands of type *fmt*; if they are not valid, the result is undefined.

The operands must be values in format *fmt*; if they are not, the result is undefined and the value of the operand FPRs becomes undefined.

In the MIPS I, II, and III implementations, there must be at least one instruction between the Compare instruction that sets the condition code and the Branch instruction that tests it. Hardware does not detect a violation of this restriction. In the MIPS IV instruction set, this restriction has been removed.

C.cond.fmt Floating-Point Compare (continued) C.cond.fmt

Operation:

```
32, 64    T:    if NaN (ValueFPR (fs, fmt) ) or NaN (ValueFPR (ft, fmt) ) then
                          less ← false
                          equal ← false
                          unordered ← true
                          if 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)
                SetFPConditionCode (cc, condition)
```

C.cond.fmt **Floating-Point Compare (continued)** C.cond.fmt

Note: Floating-point computational instructions, including compare, that receive an operand value of Signaling NaN (SNan) raise the Invalid Operation condition. Comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the unordered relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs For example, consider a comparison in which we want to know i two numbers are equal, but for which unordered would be an error.

```
# comparisons using explicit tests for QNaN
   c.eq.d       $f2,$f4      # check for equal
   nop
   bc1t         L2           # it is equal
   c.un.d       $f2,$f4      # it is not equal,
                             # but might be unordered
   bc1t         ERROR        # unordered goes off to an error handler
# not-equal-case code here
   ...
# equal-case code here
L2:
# -------------------------------------------
# comparison using comparisons that signal QNaN
   c.seq.d      $f2,$f4      # check for equal
   nop
   bc1t         L2           # it is equal
   nop
# it is not unordered here
   ...
# not-equal-case code here
   ...
#equal-case code here
L2:
```

C.cond.fmt Floating-Point Compare (continued) C.cond.fmt

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception

CEIL.L.fmt **Floating-Point Ceiling to Long Fixed-Point Format** CEIL.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CEIL.L 0 0 1 0 1 0
6	5	5	5	5	6

Format:

CEIL.L.fmt fd, fs (MIPS III format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 64-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the + ∞ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of -2^{52} to $2^{52} - 1$, the Unimplemented Operation exception occurs. If the Unimplemented Operation exception is not enabled, the exception does not occur, and $2^{52}- 1$ is returned.

This operation is defined in 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User or Supervisor mode, a Reserved Instruction exception occurs.

CEIL.L.fmt — Floating-Point Ceiling to Long Fixed-Point Format (continued) — CEIL.L.fmt

Operation:

```
32, 64 T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Inexact Operation exception
Overflow exception

CEIL.W.fmt — Floating-Point Ceiling to Single Fixed-Point Format — CEIL.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CEIL.W 0 0 1 1 1 0
6	5	5	5	5	6

Format:

CEIL.W.fmt fd, fs (MIPS II format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 32-bit fixed-point format, and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the + ∞ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31} – 1$ to $–2^{31}$, the Invalid Operation exception occurs. If the Invalid Operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

CEIL.W.fmt — Floating-Point Ceiling to Single Fixed-Point Format (continued) — CEIL.W.fmt

Operation:

```
32, 64 T:     StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception

CFC1 — Move Control Word from FPU (Coprocessor 1) — CFC1

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 0
COP1 0 1 0 0 0 1	CF 0 0 0 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CFC1 rt, fs (MIPS I format)

Description:

The contents of floating-point control register *fs* are loaded into general-purpose register *rt.*

This instruction is only defined when *fs* equals 0 or 31.

The contents of general-purpose register *rt* are undefined while the instruction immediately following this Load instruction is being executed.

Operation:

```
32    T:    temp ← FCR[fs]
      T+1: GPR[rt] ← temp

64    T:    temp ← FCR[fs]
      T+1: GPR[rt] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

Coprocessor Unusable exception

CTC1 — Move Control Word to FPU (Coprocessor 1) — CTC1

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COP1 0 1 0 0 0 1	CT 0 0 1 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CTC1 rt, fs (MIPS I format)

Description:

The contents of general-purpose register *rt* are stored in floating-point control register *fs*. This instruction is defined only if *fs* is 0 or 31.

If the cause bit of the floating-point Control/Status register (FCR31) and the corresponding enable bit are set by writing data to FCR31, the Floating-Point exception occurs. Write the data to the register before the exception occurs.

The contents of floating-point control register *fs* are undefined while the instruction immediately following this instruction is executed.

Operation:

```
32    T:      temp ← GPR[rt]
      T+1:    FCR[fs] ← temp
              COC[1] ← $FCR[31]_{23}$
64    T:      temp ← $GPR[rt]_{31...0}$
      T+1:    FCR[fs] ← temp
              COC[1] ← $FCR[31]_{23}$
```

CTC1 — Move Control Word to FPU (Coprocessor 1) (continued) — CTC1

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Division by Zero exception
Inexact Operation exception
Overflow exception
Underflow exception

CVT.D.fmt — Floating-Point Convert to Double Floating-Point Format — CVT.D.fmt

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.D 1 0 0 0 0 1
6	5	5	5	5	6

Format:

CVT.D.S fd, fs	(MIPS I format, fmt = S)
CVT.D.W fd, fs	(MIPS I format, fmt = W)
CVT.D.L fd, fs	(MIPS III format, fmt = L)

Description:

The contents of floating-point register *fs* are arithmetically converted to a double-precision floating-point format; the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single-precision floating-point format and the 32-bit or 64-bit fixed-point formats.

In the single-precision floating-point format or 32-bit fixed-point format, this conversion operation is executed correctly without losing any accuracy.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64 T:    StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt), fmt, D) )
```

CVT.D.fmt — Floating-Point Convert to Double Floating-Point Format (continued) — CVT.D.fmt

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception

CVT.L.fmt Floating-Point Convert to Long Fixed-Point Format CVT.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.L 1 0 0 1 0 1
6	5	5	5	5	6

Format:

CVT.L.fmt fd, fs (MIPS III format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 64-bit fixed-point format; the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of -2^{52} to $2^{52} - 1$, the Unimplemented Operation exception occurs. If the Unimplemented Operation exception is not enabled, the exception does not occur, and $2^{52} - 1$ is returned.

This operation is defined in 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User or Supervisor mode, a Reserved Instruction exception occurs.

CVT.L.fmt — Floating-Point Convert to Long Fixed-Point Format (continued) — CVT.L.fmt

Operation:

```
64      T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L) )
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Inexact Operation exception
Overflow exception

CVT.S.fmt

Floating-Point Convert to Single Floating-Point Format

CVT.S.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.S 1 0 0 0 0 0
6	5	5	5	5	6

Format:

CVT.S.D fd, fs	(MIPS I format, fmt = D)
CVT.S.W fd, fs	(MIPS I format, fmt = W)
CVT.S.L fd, fs	(MIPS III format, fmt = L)

Description:

The contents of floating-point register *fs* are arithmetically converted into a single-precision floating-point format; the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*. The result of the conversion is rounded according to the current rounding mode.

This instruction is valid only for conversion from the double-precision floating-point format, and 32-bit or 64-bit fixed-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64 T:    StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S) )
```

CVT.S.fmt

Floating-Point Convert to Single Floating-Point Format (continued)

CVT.S.fmt

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception
Underflow exception

CVT.W.fmt — Floating-Point Convert to Fixed-Point Format — CVT.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.W 1 0 0 1 0 0
6	5	5	5	5	6

Format:

CVT.W.fmt fd, fs (MIPS I format)

Description:

The contents of floating-point register *fs* are arithmetically converted to a 32-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the Invalid Operation exception occurs. If the Invalid Operation exception is not enabled, the exception does not occur and $2^{31} - 1$ is returned.

CVT.W.fmt — Floating-Point Convert to Fixed-Point Format (continued) — CVT.W.fmt

Operation:

```
32, 64T:      StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception

DIV.fmt Floating-Point Divide DIV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	DIV 0 0 0 0 1 1
6	5	5	5	5	6

Format:

DIV.fmt fd, fs, ft (MIPS I format)

Description:

The contents of floating-point register *fs* are divided by those of floating-point register *ft*, and the result is stored in floating-point register *rd*. The operand is processed in the floating-point format *fmt*. The operation is executed as if the accuracy were infinite, and the result is rounded according to the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt)/ValueFPR (ft, fmt) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception	Invalid Operation exception
Division by Zero exception	Inexact Operation exception
Overflow exception	Underflow exception

DMFC1 — Doubleword Move from FPU (Coprocessor 1) — DMFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	DMF 0 0 0 0 1	rt	fs	0 0 0 0 0 0 0 0 0 0 00
6	5	5	5	11

Format:

DMFC1 rt, fs (MIPS III format)

Description:

The contents of FPU general-purpose register *fs* are stored in CPU general-purpose register *rt*.

The contents of general-purpose register *rt* are undefined while the instruction immediately following this instruction is being executed.

The *FR* bit of the Status register indicates whether all 32 registers of the FPU can be specified. If the *FR* bit is 0 and the least-significant bit of *fs* is 1, this instruction is undefined.

The operation is undefined if an odd number is specified when the *FR* bit of the Status register is 0. If the *FR* bit is 1, both odd-numbered and even-numbered registers are valid.

This operation is defined in 64-bit mode or 32-bit Kernel mode.

DMFC1 **Doubleword Move from FPU (Coprocessor 1) (continued)** DMFC1

Operation:

```
64    T:     if   $SR_{26}$ = 1 then
                  data ← FGR [fs]
             else

             if $fs_0$ = 0 then
                  data ← FGR [fs + 1] || FGR [fs]
             else
                  data ← $undefined^{64}$
             endif
      T+1:   GPR[rt] ← data
```

Note: Same operation in 32-bit Kernel mode.

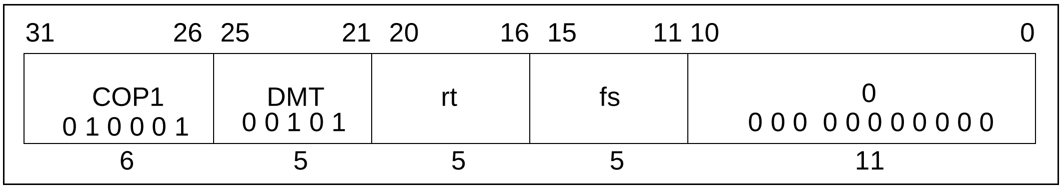
Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

DMTC1 — Doubleword Move to FPU (Coprocessor 1) — DMTC1



Format:

DMTC1 rt, fs (MIPS III format)

Description:

The contents of CPU general-purpose register *rt* are stored in FPU general-purpose register *fs*.

The contents of *fs* are undefined while the instruction immediately following this instruction is being executed.

The *FR* bit of the Status register indicates whether all the 32 registers of the FPU can be specified. If the *FR* bit is 0 and the least-significant bit of *fs* is 1, this instruction is undefined.

The operation is undefined if an odd number is specified when the *FR* bit of the Status register is 0. If the *FR* bit is 1, both odd-numbered and even-numbered registers are valid.

This operation is defined in 64-bit mode or 32-bit Kernel mode.

DMTC1 — Doubleword Move to FPU (Coprocessor 1) (continued) — DMTC1

Operation:

```
64      T:      data ← GPR[rt]

        T+1:    if  SR_26 = 1 then
                    FGR [fs] ← data
                else

                if fs_0 = 0 then
                    FGR [fs+1] ← data_63..32
                    FGR [fs] ← data_31..0
                else
                    undefined_result
                endif
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

FLOOR.L.fmt Floating-Point Floor to Long Fixed-Point Format FLOOR.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	FLOOR.L 0 0 1 0 1 1
6	5	5	5	5	6

Format:

FLOOR.L.fmt fd, fs (MIPS III format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 64-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the $-\infty$ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of -2^{52} to $2^{52} - 1$, the Unimplemented Operation exception occurs. If the Unimplemented Operation exception is not enabled, the exception does not occur and $2^{52} - 1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a Reserved Instruction exception occurs.

FLOOR.L.fmt — Floating-Point Floor to Long Fixed-Point Format (continued) — FLOOR.L.fmt

Operation:

```
64    T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Inexact Operation exception
Overflow exception

FLOOR.W.fmt — Floating-Point Floor to Single Fixed-Point Format — FLOOR.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	FLOOR.W 0 0 1 1 1 1
6	5	5	5	5	6

Format:

FLOOR.W.fmt fd, fs (MIPS II format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 32-bit fixed-point format; the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the $-\infty$ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the Invalid Operation exception occurs. If the Invalid Operation exception is not enabled, the exception does not occur and $2^{31} - 1$ is returned.

FLOOR.W.fmt **Floating-Point Floor to Single Fixed-Point Format (continued)** FLOOR.W.fmt

Operation:

```
32, 64T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

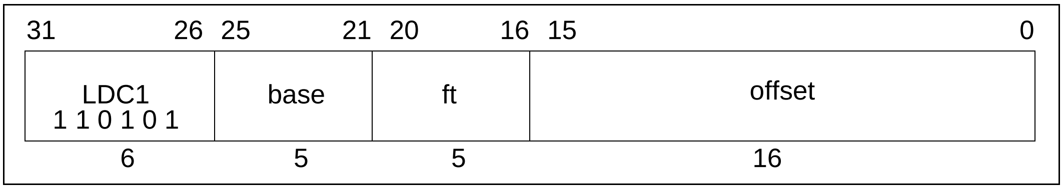
Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception

LDC1 — Load Doubleword to FPU (Coprocessor 1) — LDC1



Format:

LDC1 ft, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address.

If the *FR* bit of the Status register is 0, the contents of the doubleword at the memory location specified by the virtual address are stored in floating-point registers *ft* and *ft + 1*. At this time, the high-order 32 bits of the doubleword are stored in an odd-numbered register specified by *ft + 1* and the low-order 32 bits are stored in an even-numbered register specified by *ft*. The operation is undefined if the least-significant bit in the *ft* field is not 0.

If the *FR* bit is 1, the contents of the doubleword at the memory location specified by the virtual address are stored in floating-point register *ft*.

If any of the low-order three bits of the address is not zero, an Address Error exception occurs.

LDC1 Load Doubleword to FPU (Coprocessor 1) (continued) LDC1

Operation:

```
32   T:   vAddr ← ( (offset$_{15}$)$^{16}$ || offset$_{15...0}$) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          if SR$_{26}$ = 1 then
              FGR [ft] ← data
          elseif ft$_0$ = 0 then
              FGR [ft+1] ← data$_{63...32}$
              FGR [ft] ← data$_{31...0}$
          else
              undefined_result
          endif
64   T:   vAddr ← ( (offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR[base]
          (pAddr, uncached) ← Address Translation (vAddr, DATA)
          data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          if SR$_{26}$ = 1 then
              FGR [ft] ← data
          elseif ft$_0$ = 0 then
              FGR [ft+1] ← data$_{63...32}$
              FGR [ft] ← data$_{31...0}$
          else
              undefined_result
          endif
```

Exceptions:

Coprocessor Unusable Exception
TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LDXC1 **Load Doubleword Indexed to FPU (Coprocessor 1)** LDXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 0 1 0 0 1 1	base	index	0 0 0 0 0 0	fd	LDXC1 0 0 0 0 0 1
6	5	5	5	5	6

Format:

LDXC1 fd, index (base)   (MIPS IV format)

Description:

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in floating-point regi s t e*fd*. The contents of general-purpose registers *index* and *base* are added to form the effective address.

The *Region* bits of the effective address must be supplied by the contents of *base*. If EffectiveAddress$_{63..6}$ $\neq$ *base*$_{63..62}$, the result is undefined.

An Address Error exception occurs if EffectiveAddress$_{2..0} \neq 0$ (not doubleword aligned), and the result of the instruction is undefined.

LDXC1 Load Doubleword Indexed to FP (Coprocessor 1) (continued) LDXC1

Operation:

```
vAddr $\leftarrow$ GPR[base] + GPR[index]
if vAddr$_{2..0}$ $\neq$ $0^3$ then SignalException(AddressError) endif
(pAddr, CCA) $\leftarrow$ AddressTranslation (vAddr, DATA, LOAD)
mem $\leftarrow$ LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
if FP32RegistersMode then
    FPR[fd] $\leftarrow$ data
else
    if fd$_0$ = 0 then
        FPR[fd$_{4..1}$ || 0] $\leftarrow$ data
    else
        FPR[fd$_{4..1}$ || 0] $\leftarrow$ undefined$^{64}$
        FPR[fd$_{4..1}$ || 1] $\leftarrow$ undefined$^{64}$
    endif
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception
Address Error exception
TLB Refill exception
TLB Invalid exception

LWC1 **Load Word to FPU (Coprocessor 1)** LWC1

31 … 26	25 … 21	20 … 16	15 … 0
LWC1 1 1 0 0 0 1	base	ft	offset
6	5	5	16

Format:

LWC1 ft, offset (base) (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the virtual address are loaded to floating-point register *ft*.

If the *FR* bit of the Status register is 0 and if the least-significant bit in the *ft* field is 0, the contents of the word are stored in the low-order 32 bits of floating-point register *ft*. If the least-significant bit in the *ft* field is 1, the contents of the word are stored in the high-order 32 bits of floating-point register *ft – 1*.

If the *FR* bit is 1, all the 64-bit floating-point registers can be accessed; therefore, the contents of the word are stored in floating-point register *ft*. The value of the high-order 32 bits is undefined.

If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

LWC1 Load Word to FPU (Coprocessor 1) (continued) LWC1

Operation:

```
32    T:    vAddr ← ( (offset$_{15}$)$^{16}$ || offset$_{15...0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            if SR$_{26}$ = 1 then
                FGR [ft] ← undefined$^{32}$ || data
            else
                FGR [ft] ← data
            endif
64    T:    vAddr ← ( (offset$_{15}$)$^{48}$ || offset$_{15...}$ ) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            if SR$_{26}$ = 1 then
                FGR [ft] ← undefined$^{32}$ || data
            else
                FGR [ft] ← data
            endif
```

Exceptions:

Coprocessor Unusable exception
TLB Miss exception
TLB Invalid exception
Bus Error exception
Address Error exception

LWXC1 — Load Word Indexed to FPU (Coprocessor 1) — LWXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 0 1 0 0 1 1	base	index	0 0 0 0 0 0	fd	LWXC1 0 0 0 0 0 0
6	5	5	5	5	6

Format:

LWXC1 fd, index (base)   (MIPS IV format)

Description:

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed in the low word of floating-point register *fd*. The contents of general-purpose registers *index* and *base* are added to form the effective address.

The *Region* bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..6} \neq \textit{base}_{63..62}$, the result is undefined.

An Address Error exception occurs if $\text{EffectiveAddress}_{1..0} \neq 0$ (not word aligned), and the result of the instruction is undefined.

LWXC1 Load Word Indexed to FPU (Coprocessor 1) (continued) LWXC1

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{1..0} \neq 0^2$ then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
/* mem is aligned 64-bits from memory. Pick out correct bytes. */
mem ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
bytesel ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
if FP32RegistersMode then
        FPR[fd] ← $undefined^{32}$ || data
else
        if $fd_0$ = 0 then
                FPR[$fd_{4..1}$ || 0] ← FPR[$fd_{4..1}$ || 0]$_{63..32}$ || data
        else
                FPR[$fd_{4..1}$ || 0] ← data || FPR[$fd_{4..1}$ || 0]$_{31..0}$
        endif
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception
Address Error exception
TLB Refill exception
TLB Invalid exception

MADD.fmt **Floating-Point Multiply-Add** MADD.fmt

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	MADD 1 0 0	fmt
6	5	5	5	5	3	3

Format:

MADD.fmt fd, fr, fs, ft (MIPS IV format)

Description:

The value in floating-point register *fs* is multiplied by the value in floating-point register *ft* to produce a product. The value in floating-point register *fr* is added to the product. The resulting sum is calculated to infinite precision, rounded according to the current rounding mode in the FCR31 register, and placed into floating-point register *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

The fields *fr, fs, ft*, and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The operands must be values in format *fmt*; if they are not, the result is undefined and the value of the operand floating-point registers becomes undefined.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, vfr $+_{fmt}$ (vfs $\times_{fmt}$ vft))
```

MADD.fmt Floating-Point Multiply-Add (continued) MADD.fmt

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception

MFC1 — Move Word from FPU (Coprocessor 1) — MFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	MF 0 0 0 0 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFC1 rt, fs (MIPS I format)

Description:

The contents of floating-point general-purpose register *fs* are stored in general-purpose register *rt* of the CPU register *rt*.

The contents of general-purpose register *rt* are undefined while the instruction immediately following this instruction is being executed.

If the *FR* bit of the Status register is 0 and if the least-significant bit in the *ft* field is 0, the low-order 32 bits of floating-point register *ft* are stored in CPU general-purpose register *rt*. If the least-significant bit in the *ft* area is 1, the high-order 32 bits of floating-point register *ft – 1* are stored in general-purpose register *rt*.

If the *FR* bit is 1, all 64-bit floating-point registers can be accessed; therefore, the high-order 32 bits of floating-point register *ft* are stored in CPU general-purpose register *rt*.

Operation:

```
32    T:      data ← FGR [fs]$_{31...0}$
      T+1:    GPR [rt] ← data

64    T:      data ← FGR [fs]$_{31...0}$
      T+1:    GPR[rt] ← $(\text{data}_{31})^{32}$ || data
```

Exceptions:

Coprocessor Unusable exception

MOV.fmt **Floating-Point Move** MOV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	MOV 0 0 0 1 1 0
6	5	5	5	5	6

Format:

MOV.fmt fd, fs (MIPS I format)

Description:

The contents of floating-point register *fs* are stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

This instruction is not executed arithmetically, and no IEEE-754 exception is generated.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64     T:     StoreFPR (fd, fmt, ValueFPR (fs, fmt) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception

MOVF Move Conditional on FPU False MOVF

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	cc	0 0	tf 0	rd	0 0 0 0 0 0	MOVCI 0 0 0 0 0 1
6	5	3	1	1	5	5	6

Format:

MOVF rd, rs, cc (MIPS IV format)

Description:

If the floating-point condition code specified by *cc* is zero, then the contents of general-purpose register *rs* are placed into general-purpose register *rd*.

Operation:

```
if FPConditionCode(cc) = 0 then
        GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MOVF.fmt — Floating-Point Move Conditional on FPU False — MOVF.fmt

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	cc	0 0	tf 0	fs	fd	MOVCF 0 1 0 0 0 1
6	5	3	1	1	5	5	6

Format:

MOVF.fmt fd, fs, cc (MIPS IV format)

Description:

If the floating-point condition code specified by *cc* is zero, then the value in floating-point register *fs* is placed into floating-point register *fd*. The source and destination are values in format *fmt*.

If the condition code is not zero, then floating-point register *fs* is not copied and floating-point register *fd* retains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined. The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The move is nonarithmetic; it causes no IEEE-754 exceptions.

Operation:

```
if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

MOVN.fmt — Floating-Point Move Conditional on Not Zero — MOVN.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	rt	fs	fd	MOVN 0 1 0 0 1 1
6	5	5	5	5	6

Format:

MOVN.fmt fd, fs, rt (MIPS IV format)

Description:

If the value in general-purpose register *rt* is not equal to zero, then the value in floating-point register *fs* is placed in floating-point register *fd*. The source and destination are values in format *fmt*.

If general-purpose register *rt* contains zero, then floating-point register *fs* is not copied and floating-point register *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined. The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The move is nonarithmetic; it causes no IEEE-754 exceptions.

Operation:

```
if GPR[rt] ≠ 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

MOVT **Move Conditional on FPU True** MOVT

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	cc	0 0	tf 1	rd	0 0 0 0 0 0	MOVT 0 0 0 0 0 1
6	5	3	1	1	5	5	6

Format:

MOVT rd, rs, cc (MIPS IV format)

Description:

If the floating-point condition code specified by *cc* is one, then the contents of general-purpose register *rs* are placed into general-purpose register *rd*.

Operation:

```
if FPConditionCode(cc) = 1 then
        GPR[rd] ← GPR[rs]
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MOVT.fmt **Floating-Point Move Conditional on FPU True** MOVT.fmt

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	cc	0 0	tf 1	fs	fd	MOVT 0 1 0 0 0 1
6	5	3	1	1	5	5	6

Format:

MOVT.fmt fd, fs, cc (MIPS IV format)

Description:

If the floating-point condition code specified by *cc* is one, then the value in floating-point register *fs* is placed into floating-point register *fd*. The source and destination are values in format *fmt*.

If the condition code is not one, then floating-point register *fs* is not copied and floating-point register *fd* retains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined. The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The move is nonarithmetic; it causes no IEEE-754 exceptions.

Operation:

```
if FPConditionCode(cc) = 1 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

MOVZ.fmt **Floating-Point Move Conditional on Zero** MOVZ.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	rt	fs	fd	MOVZ 0 1 0 0 1 0
6	5	5	5	5	6

Format:

MOVZ.fmt fd, fs, rt (MIPS IV format)

Description:

If the value in general-purpose register *rt* is equal to zero, then the value in floating-point register *fs* is placed in floating-point register *fd*. The source and destination are values in format *fmt*.

If general-purpose register *rt* does not contain zero, then floating-point register *fs* is not copied and floating-point register *fd* contains its previous value in format *fmt*. If *fd* did not contain a value either in format *fmt* or previously unused data from a load or move-to operation that could be interpreted in format *fmt*, then the value of *fd* becomes undefined. The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The move is nonarithmetic; it causes no IEEE-754 exceptions.

Operation:

```
if GPR[rt] = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception

MSUB.fmt **Floating-Point Multiply-Subtract** MSUB.fmt

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	MSUB 1 0 1	fmt
6	5	5	5	5	3	3

Format:

MSUB.fmt fd, fr, fs, ft (MIPS IV format)

Description:

The value in floating-point register *fs* is multiplied by the value in floating-point register *ft* to produce a product. The value in floating-point register *fr* is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in the FCR31 register, and placed into floating-point register *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

The fields *fr, fs*, *ft*, and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The operands must be values in format *fmt*; if they are not, the result is undefined and the value of the operand floating-point registers becomes undefined.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, $(vfs \times_{fmt} vft) -_{fmt} vfr$)
```

MSUB.fmt Floating-Point Multiply-Subtract (continued) MSUB.fmt

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception

MTC1 — Move to FPU (Coprocessor 1) — MTC1

31 … 26	25 … 21	20 … 16	15 … 11	10 … 0
COP1 0 1 0 0 0 1	MT 0 0 1 0 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MTC1 rt, fs (MIPS I format)

Description:

The contents of CPU general-purpose register *rt* are stored in the floating-point general-purpose register *fs*.

The contents of floating-point register *fs* are undefined while the instruction immediately following this instruction is being executed.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the *FR* bit is 1, all of the 32 floating-point general-purpose registers can be accessed, but only the low-order 32 bits are affected by this instruction.

Operation:

```
32, 64  T:      data ← GPR [rt]2$_{31..0}$
        T+1:    if SR$_{26}$= 1 then
                     FGR [fs] ← undefined$^{32}$ || data
                else
                     FGR [fs] ← data
                endif
```

Exceptions:

Coprocessor Unusable exception

MUL.fmt Floating-Point Multiply MUL.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	MUL 0 0 0 0 1 0
6	5	5	5	5	6

Format:

MUL.fmt fd, fs, ft (MIPS I format)

Description:

The contents of floating-point register *fs* are multiplied by those of floating-point register *ft*, and the result is stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64T:     StoreFPR (fd, fmt, ValueFPR (fs, fmt) * ValueFPR (ft, fmt) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Inexact Operation exception
Overflow exception
Underflow exception

NEG.fmt Floating-Point Negate NEG.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	NEG 0 0 0 1 1 1
6	5	5	5	5	6

Format:

NEG.fmt fd, fs (MIPS I format)

Description:

The sign of the contents of floating-point register *fs* is inverted and the result is stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

The sign is inverted arithmetically. Therefore, the instruction is invalid if the operand is NaN.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64 T:    StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt) ) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception

NMADD.fmt — Floating-Point Negative Multiply-Add — NMADD.fmt

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 3	2 … 0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	NMADD 1 1 0	fmt
6	5	5	5	5	3	3

Format:

NMADD.fmt fd, fr, fs, ft (MIPS IV format)

Description:

The value in floating-point register *fs* is multiplied by the value in floating-point register *ft* to produce a product. The value in floating-point register *fr* is added to the product. The resulting sum is calculated to infinite precision, rounded according to the current rounding mode in the FCR31 register, negated by changing the sign bit, and placed into floating-point register *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

The fields *fr*, *fs*, *ft*, and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The operands must be values in format *fmt*; if they are not, the result is undefined and the value of the operand floating-point registers becomes undefined.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, $-(vfr +_{fmt} (vfs \times_{fmt} vft))$)
```

NMADD.fmt **Floating-Point Negative Multiply-Add** NMADD.fmt

(continued)

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception

NMSUB.fmt — Floating-Point Negative Multiply-Subtract — NMSUB.fmt

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 0 1 0 0 1 1	fr	ft	fs	fd	NMSUB 1 1 1	fmt
6	5	5	5	5	3	3

Format:

NMSUB.fmt fd, fr, fs, ft (MIPS IV format)

Description:

The value in floating-point register *fs* is multiplied by the value in floating-point register *ft* to produce a product. The value in floating-point register *fr* is subtracted from the product. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in the FCR31 register, negated by changing the sign bit, and placed into floating-point register *fd*. The operands and result are values in format *fmt*.

Cause bits are ORed into the *Flag* bits if no exception is taken.

The fields *fr*, *fs*, *ft*, and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

The operands must be values in format *fmt*; if they are not, the result is undefined and the value of the operand floating-point registers becomes undefined.

Operation:

```
vfr ← ValueFPR(fr, fmt)
vfs ← ValueFPR(fs, fmt)
vft ← ValueFPR(ft, fmt)
StoreFPR(fd, fmt, $-((vfs \times_{fmt} vft) -_{fmt} vfr)$)
```

NMSUB.fmt **Floating-Point Negative Multiply-Subtract (continued)** NMSUB.fmt

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception

PREFX Prefetch Indexed PREFX

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 0 1 0 0 1 1	base	index	hint	0 0 0 0 0 0	PREFX 0 0 1 1 1 1
6	5	5	5	5	6

Format:

PREFX hint, index (base)   (MIPS IV format)

Description:

PREFX adds the contents of general-purpose register *index* to the contents of general-purpose register *base* to form an effective byte address. It presents advice that data at the effective address may be used in the near future. The *hint* field supplies information about the way the data is expected to be used.

Unlike the VR5000, in which the PREFX instruction is executed as an NOP, in the VR5432 data may be prefetched into the data cache as a result of executing this instruction.

PREFX is an advisory instruction that may change the performance of the program. For all *hint* values, it neither changes architecturally visible state nor alters the meaning of the program. The supported hint values are shown in Table 18-8.

PREFX **Prefetch Indexed (continued)** PREFX

Table 18-8 Hint Field Values Used in PREFX Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1	store	Data is expected to be stored or modified. Fetch data as if for a store.
2–3		Reserved
4	load_streamed	Data is expected to be loaded (not modified) but not reused extensively; it “streams” through the cache. Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Data is expected to be stored or modified but not reused extensively; it “streams” through the cache. Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Data is expected to be loaded (not modified) and reused extensively; it should be “retained” in the cache. Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”
8–24		Reserved
25	writeback_invalidate	
26–31		Reserved

PREFX Prefetch Indexed (continued) PREFX

If MIPS IV instructions are supported and enabled and Coprocessor 1 is enabled (allowing access to CP1X), PREFX does not cause any addressing-related exceptions. If it does raise a nonaddressing-related exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data is prefetched. In such a case, even if no data is prefetched, some action that is not architecturally visible—such as write-back of a dirty cache line—can take place.

PREFX never generates a memory operation for a location with an *uncached* memory access type. However, it can result in a memory operation.

The *Region* bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..6} \neq \mathit{base}_{63..62}$, the result of the instruction is undefined.

Prefetch cannot prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Reserved Instruction exception
Coprocessor Unusable exception

RECIP.fmt **Reciprocal** RECIP.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	RECIP 010101
6	5	5	5	5	6

Format:

RECIP.fmt fd, fs (MIPS IV format)

Description:

The reciprocal of the value in floating-point register *fs* is placed into floating-point register *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation meets the full accuracy specified by the IEEE-754 floating-point standard for this operation.

The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

Operation:

```
StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception
Division by Zero exception

ROUND.L.fmt — Floating-Point Round to Long Fixed-Point Format — ROUND.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ROUND.L 0 0 1 0 0 0
6	5	5	5	5	6

Format:

ROUND.L.fmt fd, fs (MIPS III format)

Description:

The contents of floating-point register *fs* are converted into the 64-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded to the closest value or even number, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of -2^{52} to $2^{52} - 1$, the Unimplemented Operation exception occurs. If the Unimplemented Operation exception is not enabled, the exception does not occur and $2^{52} - 1$ is returned.

This operation is defined in 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User or Supervisor mode, a Reserved Instruction exception occurs.

ROUND.L.fmt — Floating-Point Round to Long Fixed-Point Format (continued) — ROUND.L.fmt

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Inexact Operation exception
Overflow exception

ROUND.W.fmt Floating-Point Round to Single Fixed-Point Format ROUND.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	ROUND.W 0 0 1 1 0 0
6	5	5	5	5	6

Format:

ROUND.W.fmt fd, fs (MIPS II format)

Description:

The contents of floating-point register *fs* are converted into the 32-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded to the closest value or even number, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the Invalid Operation exception occurs. If the Invalid Operation exception is not enabled, the exception does not occur and $2^{31} - 1$ is returned.

ROUND.W.fmt **Floating-Point Round to Single Fixed-Point Format (continued)** ROUND.W.fmt

Operation:

```
32, 64 T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception

RSQRT.fmt Reciprocal Square Root RSQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	RSQRT 0 1 0 1 1 0
6	5	5	5	5	6

Format:

RSQRT.fmt fd, fs (MIPS IV format)

Description:

The reciprocal of the positive square root of the value in floating-point register *fs* is placed into floating-point register *fd*. The operand and result are values in format *fmt*.

The numeric accuracy of this operation meets the full accuracy specified by the IEEE-754 floating-point standard for this operation.

The fields *fs* and *fd* must specify floating-point registers valid for operands of type *fmt*; if they are not valid, the result is undefined.

Operation:

```
StoreFPR(fd, fmt, 1.0 / SquareRoot(valueFPR(fs, fmt)))
```

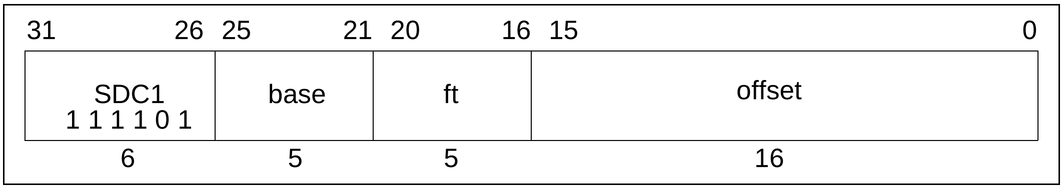
Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Overflow exception
Underflow exception
Inexact Operation exception
Division by Zero exception

SDC1 **Store Doubleword from FPU (Coprocessor 1)** SDC1



Format:

SDC1 ft, offset (base) (MIPS II format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address.

The contents of floating-point registers *ft* and *ft + 1* are stored in the memory position specified by the virtual address as a doubleword if the *FR* bit of the Status register is 0. At this time, the contents of the odd-numbered register specified by *ft + 1* correspond to the high-order 32 bits of the doubleword and the contents of the even-numbered register specified by *ft* correspond to the low-order 32 bits.

If the least-significant bit in the *ft* field is not 0, this instruction is not defined.

If the *FR* bit is 1, the contents of floating-point register *ft* are stored in the memory location specified by the virtual address as a doubleword.

If any of the low-order three bits of the address is not zero, an Address Error exception occurs.

SDC1 **Store Doubleword from FPU (Coprocessor 1) (continued)** SDC1

Operation:

```
32   T:  vAddr ← ( (offset$_{15}$)$^{16}$ || offset$_{15...0}$) + GPR [base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         if SR$_{26}$ = 1
             data ← FGR [ft]$_{63...0}$
         elseif ft$_0$ = 0 then
             data ← FGR [ft+1]$_{31...0}$ || FGR [ft]$_{31...0}$
         else
             data ← undefined$^{64}$
         endif
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64   T:  vAddr ← ( (offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         if SR$_{26}$ = 1
             data ← FGR [ft]$_{63...0}$
         elseif ft$_0$ = 0 then
             data ← FGR [ft+1]$_{31...0}$ || FGR [ft]$_{31...0}$
         else
             data ← undefined$^{64}$
         endif
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable exception
TLB Miss exception
TLB Invalid exception
TLB Modification exception
Bus Error exception
Address Error exception

SDXC1 Store Doubleword Indexed from FPU (Coprocessor 1) SDXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 0 1 0 0 1 1	base	index	fs	0 0 0 0 0 0	SDXC1 0 0 0 0 0 1
6	5	5	5	5	6

Format:

SDXC1 fs, index (base) (MIPS IV format)

Description:

The 64-bit doubleword in floating-point register *fs* is stored in memory at the location specified by the aligned effective address. The contents of general-purpose registers *index* and *base* are added to form the effective address.

The *Region* bits of the effective address must be supplied by the contents of *base*. If $\text{EffectiveAddress}_{63..6} \neq base_{63..62}$, the result is undefined.

An Address Error exception occurs if $\text{EffectiveAddress}_{2..0} \neq 0$ (not doubleword-aligned). If they are not, the result of the instruction is undefined.

Operation:

```
vAddr ← GPR[base] + GPR[index]
if $vAddr_{2..0} \neq 0^3$ then SignalException(AddressError) endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
if FP32RegistersMode then
    data ← FPR[fs]
else
    if $fs_0 = 0$ then
        data ← FPR[$fs_{4..1}$ || 0]
    else
        data ← $undefined^{64}$
    endif
endif
StoreMemory(CCA, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

SDXC1 Store Doubleword Indexed from FPU (Coprocessor 1) (continued) SDXC1

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception
Address Error exception
TLB Refill exception
TLB Modified exception
TLB Invalid exception

SQRT.fmt **Floating-Point Square Root** SQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	SQRT 0 0 0 1 0 0
6	5	5	5	5	6

Format:

SQRT.fmt fd, fs (MIPS II format)

Description:

The positive arithmetic square root of the contents of floating-point register *fs* is calculated and the result is stored in floating-point register *fd*. The operand is processed in the floating-point format *fmt*. The result is rounded as if calculated to infinite precision and then rounded according to the current rounding mode. If the value of the source operand is –0, the result will be –0. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64 T:     StoreFPR (fd, fmt, SquareRoot (ValueFPR (fs, fmt) ) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Inexact Operation exception

SUB.fmt **Floating-Point Subtract** SUB.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	SUB 0 0 0 0 0 1
6	5	5	5	5	6

Format:

SUB.fmt fd, fs, ft (MIPS I format)

Description:

The contents of floating-point register *ft* are subtracted from those of floating-point register *fs*, and the result is stored in floating-point register *fd*. The result is rounded as if calculated to infinite precision and then rounded according to the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

Operation:

```
32, 64T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) – ValueFPR (ft, fmt) )
```

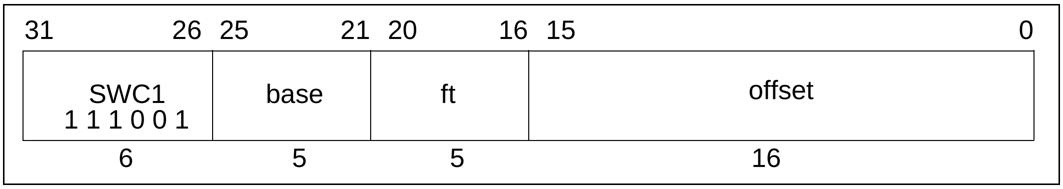
Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Unimplemented Operation exception
Invalid Operation exception
Inexact Operation exception
Overflow exception
Underflow exception

SWC1 — Store Word from FPU (Coprocessor 1) — SWC1



Format:

SWC1 ft, offset (base)   (MIPS I format)

Description:

The 16-bit *offset* is sign extended and added to the contents of general-purpose register *base* to form a virtual address. The contents of the floating-point general-purpose register *ft* are stored in the memory location at the specified address.

If the *FR* bit of the Status register is 0 and the least-significant bit in the *ft* field is 0, the contents of the low-order 32 bits of floating-point register *ft* are stored in memory. If the least-significant bit in the *ft* field is 1, the contents of the high-order 32 bits of floating-point register *ft – 1* are stored.

If the *FR* bit is 1, all of the 64-bit floating-point registers can be accessed. The contents of the low-order 32 bits of the register in the *ft* field are stored in memory.

If either of the low-order two bits of the address is not zero, an Address Error exception occurs.

SWC1 Store Word from FPU (Coprocessor 1) (continued) SWC1

Operation:

```
32    T:    vAddr ← ( (offset$_{15}$)$^{16}$ || offset$_{15...}$ ) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← FGR [ft]$_{31...0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64    T:    vAddr ← ( (offset$_{15}$)$^{48}$ || offset$_{15...}$ ) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← FGR [ft]$_{31...0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable exception
TLB Miss exception
TLB Invalid exception
TLB Modified exception
Bus Error exception
Address Error exception

SWXC1 — Store Word Indexed from FPU (Coprocessor 1) — SWXC1

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 0 1 0 0 1 1	base	index	fs	0 0 0 0 0 0	SWXC1 0 0 1 0 0 0
6	5	5	5	5	6

Format:

SWXC1 fs, index (base) (MIPS IV format)

Description:

The low 32-bit word from floating-point regis t e*fs* is stored in memory at the location specified by the aligned effective address. The contents of general-purpose registers *index* and *base* are added to form the effective address.

The *Region* bits of the effective address must be supplied by the contents of *base*. If EffectiveAddress$_{63..6}$ $\neq$ *base*$_{63..62}$, the result is undefined.

An Address Error exception occurs if EffectiveAddress$_{1..0} \neq 0$ (not word aligned). If they are not, the result of the instruction is undefined.

SWXC1 Store Word Indexed from FPU (Coprocessor 1) (continued) SWXC1

Operation:

```
vAddr $\leftarrow$ GPR[base] + GPR[index]
if vAddr$_{1..0} \neq 0^2$ then SignalException(AddressError) endif
(pAddr, CCA) $\leftarrow$ AddressTranslation(vAddr, DATA, STORE)
pAddr $\leftarrow$ pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || $0^2$))
bytesel $\leftarrow$ vAddr$_{2..0}$ xor (BigEndianCPU || $0^2$)
/* the bytes of the word are moved into the correct byte lanes */
if FP32RegistersMode then
    data $\leftarrow$ FPR[fs]$_{31..0}$
else
    if fs$_0$ = 0 then
        data $\leftarrow$ FPR[fs$_{4..1}$ || 0]$_{31..0}$
    else
        data $\leftarrow$ FPR[fs$_{4..1}$ || 0]$_{63..32}$
    endif
endif
StoreMemory (CCA, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception
Address Error exception
TLB Refill exception
TLB Modified exception
TLB Invalid exception

TRUNC.L.fmt — Floating-Point Truncate to Long Fixed-Point Format — TRUNC.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.L 0 0 1 0 01
6	5	5	5	5	6

Format:

TRUNC.L.fmt fd, fs (MIPS III format)

Description:

The contents of floating-point register *fs* are converted into the 64-bit fixed-point format and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward 0, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number, because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of -2^{52} to $2^{52} - 1$, the Unimplemented Operation exception occurs. If the Unimplemented Operation exception is not enabled, the exception does not occur and $2^{52} - 1$ is returned.

This operation is defined in 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User or Supervisor mode, a Reserved Instruction exception occurs.

TRUNC.L.fmt — Floating-Point Truncate to Long Fixed-Point Format (continued) — TRUNC.L.fmt

Operation:

```
64    T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Note: Same operation in 32-bit Kernel mode.

Exceptions:

Coprocessor Unusable exception
Floating-Point exception
Reserved Instruction exception

Floating-Point Exceptions:

Unimplemented Operation exception
Inexact Operation exception
Overflow exception

TRUNC.W.fmt Floating-Point Truncate to Single Fixed-Point Format TRUNC.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.W 0 0 1 1 0 1
6	5	5	5	5	6

Format:

TRUNC.W.fmt fd, fs (MIPS II format)

Description:

The contents of floating-point register *fs* are arithmetically converted into a 32-bit fixed-point single format, and the result is stored in floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward 0, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the Status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit is 1, both odd and even register numbers are valid.

If the source operand is infinite or NaN and if the rounded result is outside the range of $2^{31} – 1$ to $–2^{31}$, the Invalid Operation exception occurs. If the Invalid Operation exception is not enabled, the exception does not occur and $2^{31} – 1$ is returned.

TRUNC.W.fmt Floating-Point Truncate to Single Fixed-Point Format (continued) TRUNC.W.fmt

Operation:

```
32, 64T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor Unusable exception
Floating-Point exception

Floating-Point Exceptions:

Invalid Operation exception
Unimplemented Operation exception
Inexact Operation exception
Overflow exception

18.4 FPU Instruction Opcode Bit Encoding

Figure 18-3 lists the bit encoding for FPU instructions.

Opcode

28...26

31...29	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1		
7		SWC1				SDC1		

sub

23...21

25...24	0	1	2	3	4	5	6	7
0	MF	DMFh	CF	g	MT	DMTh	CT	g
1	BC	g	g	g	g	g	g	g
2	S	D	g	g	W	Lh	g	g
3	g	g	g	g	g	g	g	g

br

18...16

20...19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*

Figure 18-3 Bit Encoding for FPU Instructions (1 of 2)

	function							
5...3	2...0 0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	ROUND.Lη	TRUNC.Lη	CEIL.Lη	FLOOR.Lη	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W/RECIP
2	γ	γ	γ	γ	γ	γ	RSQRT	γ
3	γ	γ	γ	γ	γ	γ	γ	γ
4	CVT.S	CVT.D	γ	γ	CVT.W	CVT.Lη	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Figure 18-4 Bit Encoding for FPU Instructions (2 of 2)

Key:

*	When an opcode marked with an asterisk is executed, the Reserved Instruction exception occurs. These codes are reserved for future expansion.
γ	Opcodes marked with a gamma cause an Unimplemente Operation exception in all current implementations and ar reserved for future expansion
η	Opcodes marked with an eta are only defined when use of the MIPS III instruction set is enabled. If the opcode is executed when use of the instruction set is disabled (i.e., in 32-bit User or Supervisor mode), the Unimplemented Operation exception occurs.

Multimedia Instruction Set

19

This chapter provides a detailed description of the multimedia instructions. (For an general overview of VR5432 instructions, see Chapter 16.)

19.1 Multimedia Extensions

The VR5432 implements instructions and architectural extensions to support high-performance multimedia applications. These instructions interpret the 64-bit floating-point registers as packed vectors of eight unsigned 8-bit integers, called the octal byte or *OB* format. Considerable efficiency can be gained by operating in parallel on data, such as image data, in its original format rather than promoting it to larger integer formats. All of these instructions have a two-cycle latency and a one-cycle repeat rate.

Three types of vector operations are supported:

- **Vector-Vector.** Each element of source vector *vs* is operated against the corresponding elements of source vector *vt* to produce destination vector *vd*, as shown in Figure 19-1.
- **Vector-Scalar.** Each element of source vector *vs* is operated against a selected element of source vector *vt* to produce destination vecto *vd*, as shown in Figure 19-2
- **Vector-Immediate:** Each element of source vector *vs* is operated against an immediate value to produce destination vecto *vd*, as shown in Figure 19-3.

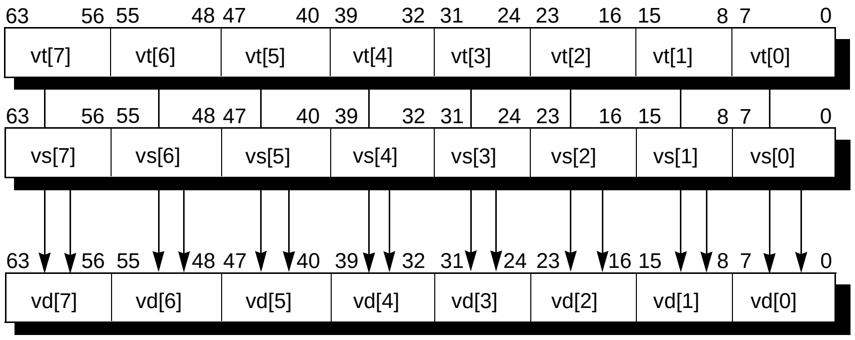

63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0
vt[7] vt[6] vt[5] vt[4] vt[3] vt[2] vt[1] vt[0]
vs[7] vs[6] vs[5] vs[4] vs[3] vs[2] vs[1] vs[0]
vd[7] vd[6] vd[5] vd[4] vd[3] vd[2] vd[1] vd[0]

Figure 19-1 Vector-Vector Operation

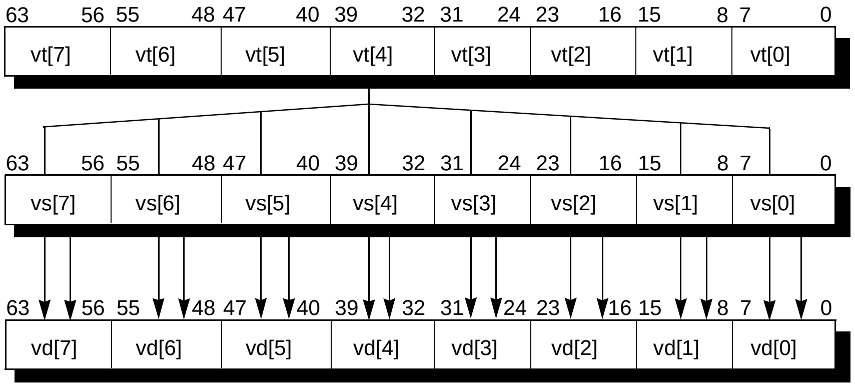

63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0
vt[7] vt[6] vt[5] vt[4] vt[3] vt[2] vt[1] vt[0]
vs[7] vs[6] vs[5] vs[4] vs[3] vs[2] vs[1] vs[0]
vd[7] vd[6] vd[5] vd[4] vd[3] vd[2] vd[1] vd[0]

Figure 19-2 Vector-Scalar Operation

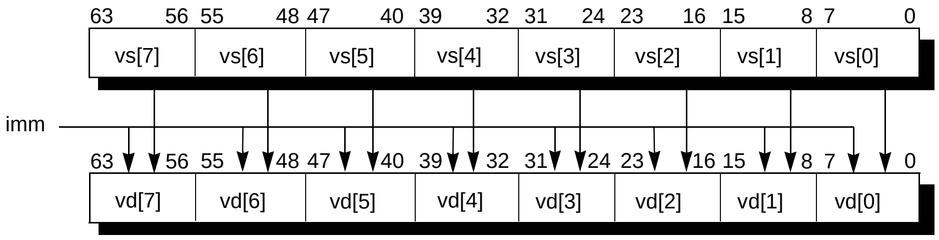


Figure 19-3 Vector-Immediate Operation

The type of vector operation is selected by a field in the instruction. The four-bit *sel* field selects the treatment of the *vt* operand field, as described in Table 19-1. When a vector-immediate operation is selected, the value of the immediate is taken from the *vt* operand field.

Table 19-1 sel Field Encoding

Bit Encoding	Description
0000	Vector-scalar operation; vt[0] is the source operand.
0001	Vector-scalar operation; vt[1] is the source operand.
0010	Vector-scalar operation; vt[2] is the source operand.
0011	Vector-scalar operation; vt[3] is the source operand.
0100	Vector-scalar operation; vt[4] is the source operand.
0101	Vector-scalar operation; vt[5] is the source operand.
0110	Vector-scalar operation; vt[6] is the source operand.
0111	Vector-scalar operation; vt[7] is the source operand.
1011	Vector-vector operation
1111	Vector-immediate operation

Vector arithmetic operations (except for multiply-accumulate and shift) are saturating; i.e., results that overflow or underflow are clamped to the largest or smallest representable values (255 and 0, respectively). No exceptions occur as a result of overflow or underflow.

Vector operations can also be performed using the 192-bit Vector Accumulator as the destination. This register is interpreted as eight 24-bit accumulators, which is sometimes referred to as the *OB* format because it is only operated upon by data in the octal byte format. As with many DSP architectures, having an accumulator wider than the operand data, shown in Figure 19-4, allows a series of operations to be performed without concern about overflow or accumulation of round-off error.

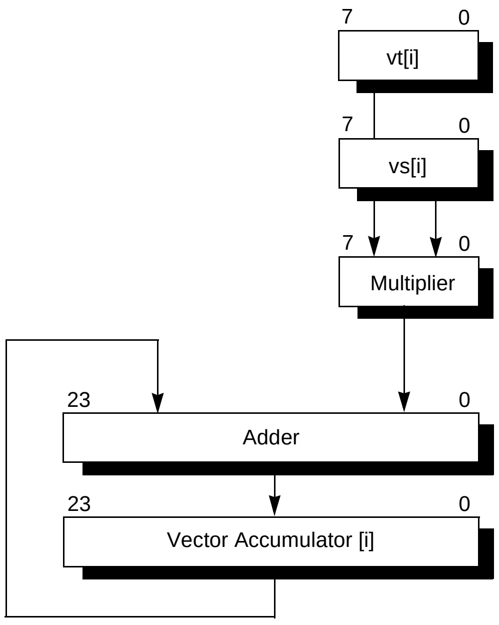
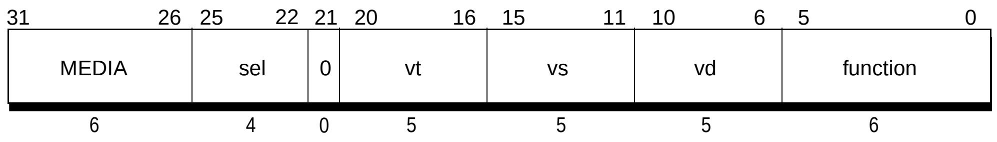


Figure 19-4 24-Bit Accumulator

19.2 Multimedia Instruction Format

A basic set of instructions to perform arithmetic and logical operations between registers is provided. In addition, instructions exist for handling unaligned data, permutations, comparisons, and conditional selection. For data movement, the standard FPU instruction set is used. The R-type format used by the multimedia instructions is shown in Figure 19-5. Some instructions do not require all fields, in which case they are sometimes used to provide additional function selection bits. The ALNI instruction has a unique interpretation of bits 21 through 25, not described by this figure.

R-type (Register)



MEDIA:	6-bit opcode
sel:	4-bit vector operation specifier or immediate value
vs:	5-bit source 1 register
vt:	5-bit source 2 register
vd:	5-bit destination register
function:	6-bit function field

Figure 19-5 Multimedia Instruction Format

19.3 Multimedia Instructions

Table 19-2 lists the multimedia instructions sorted by function field.

Table 19-2 Multimedia Instructions and Operations

Code (5:0)	**Mnemonic**	**Operation**
1	C.EQ.OB	Vector Compare Equal
2	PICKF.OB	Vector Pick False
3	PICKT.OB	Vector Pick True
4	C.LT.OB	Vector Compare Less Than
5	C.LE.OB	Vector Compare Less Than or Equal
6	MIN.OB	Vector Minimum
7	MAX.OB	Vector Maximum
10	SUB.OB	Vector Subtract
11	ADD.OB	Vector ADD
12	AND.OB	Vector AND
13	XOR.OB	Vector XOR
14	OR.OB	Vector OR
15	NOR.OB	Vector NOR
16	SLL.OB	Vector Shift Left Logical
18	SRL.OB	Vector Shift Right Logical
24	ALNI.OB	Vector Align
31, sel = 4	SHFL.PACH.OB	Vector Element Shuffle
31, sel = 5	SHFL.PACL.OB	Vector Element Shuffle
31, sel = 6	SHFL.MIXH.OB	Vector Element Shuffle
31, sel = 7	SHFL.MIXL.OB	Vector Element Shuffle
32	RZU.OB	Vector Scale, Round, and Clamp Accumulator
48	MUL.OB	Vector Multiply
50, vd = 0	MULS.OB	Vector Multiply and Subtract Accumulator
50, vd = 16	MULSL.OB	Vector Multiply, Subtract, and Load Accumulator
51, vd = 0	MULA.OB	Vector Multiply-Accumulate
51, vd = 16	MULL.OB	Vector Multiply and Load Accumulator
62, sel = 0	WACL.OB	Vector Write Accumulator Low
62, sel = 8	WACH.OB	Vector Write Accumulator High

Table 19-2 Multimedia Instructions and Operations (continued)

Code (5:0)	**Mnemonic**	**Operation**
63, sel = 0	RACL.OB	Vector Read Accumulator Low
63, sel = 4	RACM.OB	Vector Read Accumulator Middle
63, sel = 8	RACH.OB	Vector Read Accumulator High

ADD.OB　　Vector Add　　ADD.OB

31　26	25　22	21	20　16	15　11	10　6	5　0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	ADD 0 0 1 0 1 1
6	4	1	5	5	5	6

Format:

ADD.OB vd, vs, vt

Description:

The values in vector *vt* are added to the values in vector *vs*. Saturated arithmetic is performed: overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

ADD.OB Vector Add (continued) ADD.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← AddOB($ts_{63..56}$, $tt_{63..56}$)
        || AddOB($ts_{55..48}$, $tt_{55..48}$)
        || AddOB($ts_{47..40}$, $tt_{47..40}$)
        || AddOB($ts_{39..32}$, $tt_{39..32}$)
        || AddOB($ts_{31..24}$, $tt_{31..24}$)
        || AddOB($ts_{23..16}$, $tt_{23..16}$)
        || AddOB($ts_{15..8}$, $tt_{15..8}$)
        || AddOB($ts_{7..0}$, $tt_{7..0}$)
function AddOB(ts, tt)
        t ← (0 || ts) + (0 || tt)
        if $t_8$ = 1 then
                AddOB ← $1^8$
        else
                AddOB ← $t_{7..0}$
        endif
end AddOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

ALNI.OB Vector Align, Constant Alignment ALNI.OB

31 … 26	25 24	23 … 21	20 … 16	15 … 11	10 … 6	5 … 0
MEDIA 0 1 0 0 1 0	0 0 0	Imm	vt	vs	vd	ALNI 0 1 1 0 0 0
6	2	3	5	5	5	6

Format:

ALNI.OB vd, vs, vt, imm

Description:

The align amount is computed by masking the *immediate*, then using that value to control a funnel shift of vector *vs* concatenated with vector *vt*. No immediate or scalar mode is available.

No data-dependent exceptions are possible. The operands must be values in *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. This operation does not interpret the format of the registers specified. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

```
$s \leftarrow imm_{2..0} || 0^3$
if BigEndianCPU then
    $FPR[vd] \leftarrow (FPR[vs] || FPR[vt])_{127-s..64-s}$
else
    $FPR[vd] \leftarrow (FPR[vs] || FPR[vt])_{63+s..}$
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

AND.OB **Vector AND** AND.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	AND 0 0 1 1 0 0
6	4	1	5	5	5	6

Format:

AND.OB vd, vs, vt

Description:

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical AND operation. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

AND.OB Vector AND (continued) AND.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← AndOB($ts_{63..56}$, $tt_{63..56}$)
          || AndOB($ts_{55..48}$, $tt_{55..48}$)
          || AndOB($ts_{47..40}$, $tt_{47..40}$)
          || AndOB($ts_{39..32}$, $tt_{39..32}$)
          || AndOB($ts_{31..24}$, $tt_{31..24}$)
          || AndOB($ts_{23..16}$, $tt_{23..16}$)
          || AndOB($ts_{15..8}$, $tt_{15..8}$)
          || AndOB($ts_{7..0}$, $tt_{7..0}$)
function AndOB(ts, tt)
          AndOB ← (0 || ts) and (0 || tt)
end AndOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

C.EQ.OB Vector Compare (Equal) C.EQ.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	0 0 0 0 0 0	C.EQ 0 0 0 0 0 1
6	4	1	5	5	5	6

Format:

C.EQ.OB vs, vt

Description:

The values in vector *vt* are compared to the values in vector *vs* and the result is written to the condition codes. All 8 *CC* bits are written with comparison results. The comparison made is equal (EQ). The inverse comparison (NE) is not necessary; the instructions that use condition codes (BC1F, BC1T, MOVF, MOVT, PICKF, PICKT) allow both *CC* = 0 and *CC* = 1 tests. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

C.EQ.OB — Vector Compare (Equal) (continued) — C.EQ.OB

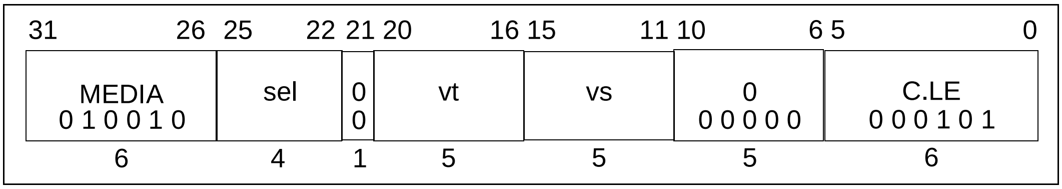
Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
SetFPConditionCode(7, ($ts_{63..56}$ = $tt_{63..56}$))
SetFPConditionCode(6, ($ts_{55..48}$ = $tt_{55..48}$))
SetFPConditionCode(5, ($ts_{47..40}$ = $tt_{47..40}$))
SetFPConditionCode(4, ($ts_{39..32}$ = $tt_{39..32}$))
SetFPConditionCode(3, ($ts_{31..24}$ = $tt_{31..24}$))
SetFPConditionCode(2, ($ts_{23..16}$ = $tt_{23..16}$))
SetFPConditionCode(1, ($ts_{15..8}$ = $tt_{15..8}$))
SetFPConditionCode(0, ($ts_{7..0}$ = $tt_{7..0}$))
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

C.LE.OB Vector Compare (Less Than or Equal) C.LE.OB



Format:

C.LE.OB vs, vt

Description:

The values in vector *vt* are compared to the values in vector *vs* and the result is written to the condition codes. All 8 *CC* bits are written with comparison results. The comparison made is less than or equal (LE). The inverse comparison (GT) is not necessary; the instructions that use condition codes (BC1F, BC1T, MOVF, MOVT, PICKF, PICKT) allow both *CC* = 0 and *CC* = 1 tests. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

C.LE.OB — Vector Compare (Less Than or Equal) (continued) — C.LE.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
SetFPConditionCode(7, $(ts_{63..56} \le tt_{63..56})$)
SetFPConditionCode(6, $(ts_{55..48} \le tt_{55..48})$)
SetFPConditionCode(5, $(ts_{47..40} \le tt_{47..40})$)
SetFPConditionCode(4, $(ts_{39..32} \le tt_{39..32})$)
SetFPConditionCode(3, $(ts_{31..24} \le tt_{31..24})$)
SetFPConditionCode(2, $(ts_{23..16} \le tt_{23..16})$)
SetFPConditionCode(1, $(ts_{15..8} \le tt_{15..8})$)
SetFPConditionCode(0, $(ts_{7..0} \le tt_{7..0})$)
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

C.LT.OB **Vector Compare (Less Than)** C.LT.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	0 0 0 0 0 0	C.LT 0 0 0 1 0 0
6	4	1	5	5	5	6

Format:

C.LT.OB vs, vt

Description:

The values in vector *vt* are compared to the values in vector *vs* and the result is written to the condition codes. All 8 *CC* bits are written with comparison results. The comparison made is less than(LT). The inverse comparison (GE) is not necessary; the instructions that use condition codes (BC1F, BC1T, MOVF, MOVT, PICKF, PICKT) allow both *CC* = 0 and *CC* = 1 tests. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

C.LT.OB — Vector Compare (Less Than) (continued) — C.LT.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
SetFPConditionCode(7, $(ts_{63..56} < tt_{63..56})$)
SetFPConditionCode(6, $(ts_{55..48} < tt_{55..48})$)
SetFPConditionCode(5, $(ts_{47..40} < tt_{47..40})$)
SetFPConditionCode(4, $(ts_{39..32} < tt_{39..32})$)
SetFPConditionCode(3, $(ts_{31..24} < tt_{31..24})$)
SetFPConditionCode(2, $(ts_{23..16} < tt_{23..16})$)
SetFPConditionCode(1, $(ts_{15..8} < tt_{15..8})$)
SetFPConditionCode(0, $(ts_{7..0} < tt_{7..0})$)
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MAX.OB Vector Maximum MAX.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	MAX 0 0 0 1 1 1
6	4	1	5	5	5	6

Format:

MAX.OB vd, vs, vt

Description:

The values in vector *vt* are compared to the values in vector *vs* and the larger is written to each element of vector *vd*. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MAX.OB Vector Maximum (continued) MAX.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← MaxOB($ts_{63..56}$, $tt_{63..56}$)
        || MaxOB($ts_{55..48}$, $tt_{55..48}$)
        || MaxOB($ts_{47..40}$, $tt_{47..40}$)
        || MaxOB($ts_{39..32}$, $tt_{39..32}$)
        || MaxOB($ts_{31..24}$, $tt_{31..24}$)
        || MaxOB($ts_{23..16}$, $tt_{23..16}$)
        || MaxOB($ts_{15..8}$, $tt_{15..8}$)
        || MaxOB($ts_{7..0}$, $tt_{7..0}$)
function MaxOB(ts, tt)
        if (0 || ts) > (0 || tt) then
                MaxOB ← ts
        else
                MaxOB ← tt
        endif
end MaxOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MIN.OB **Vector Minimum** MIN.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	MIN 0 0 0 1 1 0
6	4	1	5	5	5	6

Format:

MIN.OB vd, vs, vt

Description:

The values in vector *vt* are compared to the values in vector *vs* and the smaller is written to each element of vector *vd*. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MIN.OB Vector Minimum (continued) MIN.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← MinOB($ts_{63..56}$, $tt_{63..56}$)
        || MinOB($ts_{55..48}$, $tt_{55..48}$)
        || MinOB($ts_{47..40}$, $tt_{47..40}$)
        || MinOB($ts_{39..32}$, $tt_{39..32}$)
        || MinOB($ts_{31..24}$, $tt_{31..24}$)
        || MinOB($ts_{23..16}$, $tt_{23..16}$)
        || MinOB($ts_{15..8}$, $tt_{15..8}$)
        || MinOB($ts_{7..0}$, $tt_{7..0}$)
function MinOB(ts, tt)
        if (0 || ts) < (0 || tt) then
                MinOB ← ts
        else
                MinOB ← tt
        endif
end MinOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MUL.OB **Vector Multiply** MUL.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	MUL 1 1 0 0 0 0
6	4	1	5	5	5	6

Format:

MUL.OB vd, vs, vt

Description:

The values in vector *vt* are multiplied by the values in vector *vs* and the product is written into vector *vd*. Saturated arithmetic is performed: overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MUL.OB Vector Multiply (continued) MUL.OB

Operation:

```
ts $\leftarrow$ FPR[vs]
tt $\leftarrow$ select(sel, vt)
FPR[vd] $\leftarrow$ MulOB(ts$_{63..56}$, tt$_{63..56}$)
        || MulOB(ts$_{55..48}$, tt$_{55..48}$)
        || MulOB(ts$_{47..40}$, tt$_{47..40}$)
        || MulOB(ts$_{39..32}$, tt$_{39..32}$)
        || MulOB(ts$_{31..24}$, tt$_{31..24}$)
        || MulOB(ts$_{23..16}$, tt$_{23..16}$)
        || MulOB(ts$_{15..8}$, tt$_{15..8}$)
        || MulOB(ts$_{7..0}$, tt$_{7..0}$)
function MulOB(ts, tt)
        t $\leftarrow$ (0$^8$ || ts) $\times$ (0$^8$ || tt)
        if t$_{15..8}$ $\neq$ 0$^8$ then
                MulOB $\leftarrow$ 1$^8$
        else
                MulOB $\leftarrow$ t$_{7..0}$
        endif
end MulOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MULA.OB — Vector Multiply-Accumulate — MULA.OB

31 … 26	25 … 22	21	20 … 16	15 … 11	10 … 6	5 … 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	A 0 0 0 0 0	MULA 1 1 0 0 1 1
6	4	1	5	5	5	6

Format:

MULA.OB vs, vt

Description:

The values in vector *vt* are multiplied by the values in vector *vs* and the product is added to the Accumulator. Wrapped arithmetic is performed: overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator. The Accumulator is in the *OB* format. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MULA.OB Vector Multiply-Accumulate (continued) MULA.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
ACC ← AccMulOB(ACC$_{191..168}$, ts$_{63..56}$, tt$_{63..56}$)
        || AccMulOB(ACC$_{167..144}$, ts$_{55..48}$, tt$_{55..48}$)
        || AccMulOB(ACC$_{143..120}$, ts$_{47..40}$, tt$_{47..40}$)
        || AccMulOB(ACC$_{119..96}$, ts$_{39..3}$ , tt$_{39..32}$)
        || AccMulOB(ACC$_{95..72}$, ts$_{31..24}$, tt$_{31..24}$)
        || AccMulOB(ACC$_{71..48}$, ts$_{23..16}$, tt$_{23..16}$)
        || AccMulOB(ACC$_{47..24}$, ts$_{15..8}$, tt$_{15..8}$)
        || AccMulOB(ACC$_{23..0}$, ts$_{7..0}$, tt$_{7..0}$)
function AccMulOB(a, ts, tt)
        AccMulOB ← a + (0$^{16}$ || ts) × (0$^{16}$ || tt)
end AccMulOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MULL.OB — Vector Multiply and Load Accumulator — MULL.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	L 1 0 0 0 0	MULL 1 1 0 0 1 1
6	4	1	5	5	5	6

Format:

MULL.OB vs, vt

Description:

The values in vector *vt* are multiplied by the values in vector *vs* and the product is stored in the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator. The Accumulator result is in the *OB* format. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MULL.OB Vector Multiply and Load Accumulator (continued) MULL.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
ACC ← AccMulOB($0^{24}$, $ts_{63..56}$, $tt_{63..56}$)
        || AccMulOB($0^{24}$, $ts_{55..48}$, $tt_{55..48}$)
        || AccMulOB($0^{24}$, $ts_{47..40}$, $tt_{47..40}$)
        || AccMulOB($0^{24}$, $ts_{39..32}$, $tt_{39..32}$)
        || AccMulOB($0^{24}$, $ts_{31..24}$, $tt_{31..24}$)
        || AccMulOB($0^{24}$, $ts_{23..16}$, $tt_{23..16}$)
        || AccMulOB($0^{24}$, $ts_{15..8}$, $tt_{15..8}$)
        || AccMulOB($0^{24}$, $ts_{7..0}$, $tt_{7..0}$)
function AccMulOB(a, ts, tt)
        AccMulOB ← a + ($0^{16}$ || ts) × ($0^{16}$ || tt)
end AccMulOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MULS.OB **Vector Multiply and Subtract Accumulator** MULS.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	S 0 0 0 0 0	MULS 1 1 0 0 1 0
6	4	1	5	5	5	6

Format:

MULS.OB vs, vt

Description:

The values in vector *vt* are multiplied by the values in vector *vs* and the product is subtracted from the Accumulator. Wrapped arithmetic is performed: overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator. The Accumulator is in the *OB* format. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MULS.OB Vector Multiply and Subtract Accumulator (continued) MULS.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
ACC ← SubMulOB(ACC$_{191..168}$, ts$_{63..56}$, tt$_{63..5}$ )
        || SubMulOB(ACC$_{167..144}$, ts$_{55..48}$, tt$_{55..48}$)
        || SubMulOB(ACC$_{143..120}$, ts$_{47..40}$, tt$_{47..40}$)
        || SubMulOB(ACC$_{119..96}$, ts$_{39..32}$, tt$_{39..32}$)
        || SubMulOB(ACC$_{95..72}$, ts$_{31..24}$, tt$_{31..24}$)
        || SubMulOB(ACC$_{71..48}$, ts$_{23..16}$, tt$_{23..16}$)
        || SubMulOB(ACC$_{47..24}$, ts$_{15..8}$, tt$_{15..8}$)
        || SubMulOB(ACC$_{23..0}$, ts$_{7..0}$, tt$_{7..0}$)
function SubMulOB(a, ts, tt)
        SubMulOB ← a - $(0^{16}$ || ts$) \times (0^{16}$ || tt$)$
end SubMulOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

MULSL.OB Vector Multiply Subtract and Load MULSL.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	L 1 0 0 0 0	MULSL 1 1 0 0 1 0
6	4	1	5	5	5	6

Format:

MULSL.OB vs, vt

Description:

The values in vector *vt* are multiplied by the values in vector *vs* and negated. The vector result is stored to the Accumulator. The Accumulator result is in the *OB* format. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

MULSL.OB — Vector Multiply Subtract and Load (continued) — MULSL.OB

Operation:

```
ts $\leftarrow$ FPR[vs]
tt $\leftarrow$ select(sel, vt)
ACC $\leftarrow$ SubMulOB($0^{24}$, $ts_{63..56}$, $tt_{63..56}$)
        || SubMulOB($0^{24}$, $ts_{55..48}$, $tt_{55..48}$)
        || SubMulOB($0^{24}$, $ts_{47..40}$, $tt_{47..40}$)
        || SubMulOB($0^{24}$, $ts_{39..32}$, $tt_{39..32}$)
        || SubMulOB($0^{24}$, $ts_{31..24}$, $tt_{31..24}$)
        || SubMulOB($0^{24}$, $ts_{23..16}$, $tt_{23..16}$)
        || SubMulOB($0^{24}$, $ts_{15..8}$, $tt_{15..8}$)
        || SubMulOB($0^{24}$, $ts_{7..0}$, $tt_{7..0}$)
function SubMulOB(a, ts, tt)
        SubMulOB $\leftarrow$ a - ($0^{16}$ || ts) $\times$ ($0^{16}$ || tt)
end SubMulOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

NOR.OB Vector NOR NOR.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	NOR 0 0 1 1 1 1
6	4	1	5	5	5	6

Format:

NOR.OB vd, vs, vt

Description:

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical NOR operation. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

NOR.OB — Vector NOR (continued) — NOR.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← NorOB(ts$_{63..56}$, tt$_{63..56}$)
        || NorOB(ts$_{55..48}$, tt$_{55..48}$)
        || NorOB(ts$_{47..40}$, tt$_{47..40}$)
        || NorOB(ts$_{39..32}$, tt$_{39..32}$)
        || NorOB(ts$_{31..24}$, tt$_{31..24}$)
        || NorOB(ts$_{23..16}$, tt$_{23..16}$)
        || NorOB(ts$_{15..8}$, tt$_{15..8}$)
        || NorOB(ts$_{7..0}$, tt$_{7..0}$)
function NorOB(ts, tt)
        NorOB ← (0 || ts) nor (0 || tt)
end NorOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

OR.OB Vector OR OR.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	OR 0 0 1 1 1 0
6	4	1	5	5	5	6

Format:

OR.OB vd, vs, vt

Description:

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical OR operation. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

OR.OB Vector OR (continued) OR.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← OrOB($ts_{63..56}$, $tt_{63..56}$)
        || OrOB($ts_{55..4}$ , $tt_{55..48}$)
        || OrOB($ts_{47..4}$ , $tt_{47..40}$)
        || OrOB($ts_{39..3}$ , $tt_{39..32}$)
        || OrOB($ts_{31..2}$ , $tt_{31..24}$)
        || OrOB($ts_{23..1}$ , $tt_{23..16}$)
        || OrOB($ts_{15..8}$, $tt_{15..8}$)
        || OrOB($ts_{7..0}$, $tt_{7..0}$)
function OrOB(ts, tt)
        OrOB ← (0 || ts) or (0 || tt)
end OrOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

PICKF.OB Vector Pick False PICKF.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	PICKF 0 0 0 0 1 0
6	4	1	5	5	5	6

Format:

PICKF.OB vd, vs, vt

Description:

The vector *vd* is written with either the corresponding element of vector *vs* or the corresponding element of vector *vt*, depending on the state of the *CC* bits. All 8 *CC* bits are used. The *sel* field selects the values of *vt*[] used for each *i*.

Both PICKF and PICKT are necessary since the operands are not symmetrical; every element of vector *vs* is used, whereas the *sel* field selects values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be a value in *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

PICKF.OB Vector Pick False (continued) PICKF.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← PickOB(FPConditionCode(7) = 0, $ts_{63..56}$, $tt_{63..56}$)
        || PickOB(FPConditionCode(6) = 0, $ts_{55..48}$, $tt_{55..48}$)
        || PickOB(FPConditionCode(5) = 0, $ts_{47..40}$, $tt_{47..40}$)
        || PickOB(FPConditionCode(4) = 0, $ts_{39..32}$, $tt_{39..32}$)
        || PickOB(FPConditionCode(3) = 0, $ts_{31..24}$, $tt_{31..24}$)
        || PickOB(FPConditionCode(2) = 0, $ts_{23..16}$, $tt_{23..16}$)
        || PickOB(FPConditionCode(1) = 0, $ts_{15..8}$, $tt_{15..8}$)
        || PickOB(FPConditionCode(0) = 0, $ts_{7..0}$, $tt_{7..0}$)
function PickOB(c, ts, tt)
        if c then
                PickOB ← ts
        else
                PickOB ← tt
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

PICKT.OB — Vector Pick True — PICKT.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	PICKT 0 0 0 0 1 1
6	4	1	5	5	5	6

Format:

PICKT.OB vd, vs, vt

Description:

The vector *vd* is written with either the corresponding element of vector *vs* or the corresponding element of vector *vt*, depending on the state of the *CC* bits. All 8 *CC* bits are used. The *sel* field selects the values of *vt*[] used for each *i*.

Both PICKF and PICKT are necessary since the operands are not symmetrical; every element of vector *vs* is used, whereas the *sel* field selects values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be a value in *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

PICKT.OB **Vector Pick True (continued)** PICKT.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← PickOB(FPConditionCode(7) = 1, $ts_{63..56}$, $tt_{63..56}$)
        || PickOB(FPConditionCode(6) = 1, $ts_{55..48}$, $tt_{55..48}$)
        || PickOB(FPConditionCode(5) = 1, $ts_{47..40}$, $tt_{47..40}$)
        || PickOB(FPConditionCode(4) = 1, $ts_{39..32}$, $tt_{39..32}$)
        || PickOB(FPConditionCode(3) = 1, $ts_{31..24}$, $tt_{31..24}$)
        || PickOB(FPConditionCode(2) = 1, $ts_{23..16}$, $tt_{23..16}$)
        || PickOB(FPConditionCode(1) = 1, $ts_{15..8}$, $tt_{15..8}$)
        || PickOB(FPConditionCode(0) = 1, $ts_{7..0}$, $tt_{7..0}$)
function PickOB(c, ts, tt)
        if c then
               PickOB ← ts
        else
               PickOB ← tt
endif
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

RACH.OB — Vector Read Accumulator High — RACH.OB

31 … 26	25 … 22	21	20 … 16	15 … 11	10 … 6	5 … 0
MEDIA 0 1 0 0 1 0	H 1 0 0 0	0 0	0 0 0 0 0 0	0 0 0 0 0 0	vd	RACH 1 1 1 1 1 1
6	4	1	5	5	5	6

Format:

RACH.OB vd

Description:

Read the most-significant third of the bits of the Accumulator elements. No clamping of the values extracted is performed; the bits are simply copied into elements of *vd*[].

RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator.

No data-dependent exceptions are possible. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

```
FPR[vd] ← $ACC_{191..184}$
        || $ACC_{167..160}$
        || $ACC_{143..136}$
        || $ACC_{119..112}$
        || $ACC_{95..88}$
        || $ACC_{71..64}$
        || $ACC_{47..40}$
        || $ACC_{23..16}$
```

Exceptions:

Coprocessor Unusable exception

RACL.OB Vector Read Accumulator Low RACL.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	L 0 0 0 0	0 0	0 0 0 0 0 0	0 0 0 0 0 0	vd	RACL 1 1 1 1 1 1
6	4	1	5	5	5	6

Format:

RACL.OB vd

Description:

Read the least-significant third of the bits of the Accumulator elements. No clamping of the values extracted is performed; the bits are simply copied into elements of *vd*[].

RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator.

No data-dependent exceptions are possible. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

```
FPR[vd] ← $ACC_{175..168}$
        || $ACC_{151..144}$
        || $ACC_{127..120}$
        || $ACC_{103..96}$
        || $ACC_{79..72}$
        || $ACC_{55..48}$
        || $ACC_{31..24}$
        || $ACC_{7..0}$
```

Exceptions:

Coprocessor Unusable exception

RACM.OB — Vector Read Accumulator Middle — RACM.OB

31 … 26	25 … 22	21	20 … 16	15 … 11	10 … 6	5 … 0
MEDIA 0 1 0 0 1 0	M 0 1 0 0	0 0	0 0 0 0 0 0	0 0 0 0 0 0	vd	RACM 1 1 1 1 1 1
6	4	1	5	5	5	6

Format:

RACM.OB vd

Description:

Read the middle third of the bits of the Accumulator elements. No clamping of the values extracted is performed; the bits are simply copied into elements of *vd*[].

RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator.

No data-dependent exceptions are possible. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

```
FPR[vd] ← $ACC_{183..176}$
        || $ACC_{159..152}$
        || $ACC_{135..128}$
        || $ACC_{111..104}$
        || $ACC_{87..80}$
        || $ACC_{63..56}$
        || $ACC_{39..32}$
        || $ACC_{15..8}$
```

Exceptions:

Coprocessor Unusable exception

RZU.OB Vector Scale, Round, and Clamp Accumulator RZU.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	0 0 0 0 0 0	vd	RZU 1 0 0 0 0 0
6	4	1	5	5	5	6

Format:

RZU.OB vd, vt

Description:

The values in the Accumulator are logically shifted right by the values in vector *vt*, rounded to the nearest value with exactly halfway results rounded toward zero, and clamped to an unsigned subset of the range of *vd*[]. The Accumulator is in the *OB* format. The *sel* field selects the values of *vt*[] used for each *i*. The shift amount must be an immediate and the value must be 0, 8, or 16. The clamping range is 0..255.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

RZU.OB **Vector Scale, Round, and Clamp Accumulator (continued)** RZU.OB

Operation:

```
tt ← select(sel, vt)
FPR[vd] ← RZUOB($ACC_{191..168}$, $tt_{63..56}$)
        || RZUOB($ACC_{167..144}$, $tt_{55..48}$)
        || RZUOB($ACC_{143..120}$, $tt_{47..40}$)
        || RZUOB($ACC_{119..96}$, $tt_{39..32}$)
        || RZUOB($ACC_{95..72}$, $tt_{31..24}$)
        || RZUOB($ACC_{71..48}$, $tt_{23..16}$)
        || RZUOB($ACC_{47..24}$, $tt_{15..8}$)
        || RZUOB($ACC_{23..0}$, $tt_{7..0}$)
function RZUOB(a, s)
        if 0 || s > 23 then
                RZUOB ← $0^8$
        else
                t ← $0^s$ || $a_{23..s}$
                if 0 || t < $0^{17}$ || $1^8$ then
                                RZUOB ← $t_{7..0}$
                else
                                RZUOB ← $1^8$
                endif
        endif
end RZUOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

SHFL.op.OB — Vector Shuffle — SHFL.op.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	SHFL 0 1 1 1 1 1
6	4	1	5	5	5	6

Format:

SHFL.op.OB vd, vs, vt

Description:

Elements of vectors *vs* and *vt* are merged into a new vector. Not all combinations of values are available; the operations of the variants of this instruction are tailored to the data movement patterns of specific calculations. The shuffles available are given in Table 19-3.

The *sel* field selects the values of *vt*[] used for each *i*. The *sel* field must specify a vector, not an immediate or a scalar. The remaining bits in the field are not used for a *vt*[] select, but rather are used to encode the shuffle operation.

Table 19-3 Operation Encoding for Shuffles

sel	Operation	vd[7]	vd[6]	vd[5]	vd[4]	vd[3]	vd[2]	vd[1]	vd[0]
0100	PACH	vs[7]	vs[5]	vs[3]	vs[1]	vt[7]	vt[5]	vt[3]	vt[1]
0101	PACL	vs[6]	vs[4]	vs[2]	vs[0]	vt[6]	vt[4]	vt[2]	vt[0]
0110	MIXH	vs[7]	vt[7]	vs[6]	vt[6]	vs[5]	vt[5]	vs[4]	vt[4]
0111	MIXL	vs[3]	vt[3]	vs[2]	vt[2]	vs[1]	vt[1]	vs[0]	vt[0]

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

SHFL.op.OB **Vector Shuffle (continued)** SHFL.op.OB

Operation:

```
PACH.OB
    ts ← FPR[vs]
    tt ← select(sel, vt)
    FPR[vd] ← $ts_{63..56}$ || $ts_{47..40}$
        || $ts_{31..24}$ || $ts_{15..8}$
        || $tt_{63..56}$ || $tt_{47..4}$
        || $tt_{31..24}$ || $tt_{15..8}$
PACL.OB
    ts ← FPR[vs]
    tt ← select(sel, vt)
    FPR[vd] ← $ts_{55..48}$ || $ts_{39..32}$
        || $ts_{23..16}$ || $ts_{7..0}$
        || $tt_{55..48}$ || $tt_{39..3}$
        || $tt_{23..16}$ || $tt_{7..0}$
MIXH.OB
    ts ← FPR[vs]
    tt ← select(sel, vt)
    FPR[vd] ← $ts_{63..56}$ || $tt_{63..56}$
        || $ts_{55..48}$ || $tt_{55..48}$
        || $ts_{47..40}$ || $tt_{47..40}$
        || $ts_{39..32}$ || $tt_{39..32}$
```

SHFL.op.OB **Vector Shuffle (continued)** SHFL.op.OB

Operation (continued):

```
MIXL.OB
    ts ← FPR[vs]
    tt ← select(sel, vt)
    FPR[vd] ← $ts_{31..24}$ || $tt_{31..24}$
        || $ts_{23..16}$ || $tt_{23..16}$
        || $ts_{15..8}$ || $tt_{15..8}$
        || $ts_{7..0}$ || $tt_{7..0}$
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

SLL.OB **Vector Shift Left Logical** SLL.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	SLL 0 1 0 0 0 0
6	4	1	5	5	5	6

Format:

SLL.OB vd, vs, vt

Description:

Each element of vector *vs* is shifted left by an amount specified by an immediate or an element of vector *vt*, and zeros are shifted into the low-order bits. The results are written into vector *vd*. All but the lower 3 bits of the shift amount are masked to 0, so the largest possible shift is 7 places. The *sel* field selects the values of *vt*[] used for each *i*, which must be a scalar or an immediate.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

SLL.OB **Vector Shift Left Logical (continued)** SLL.OB

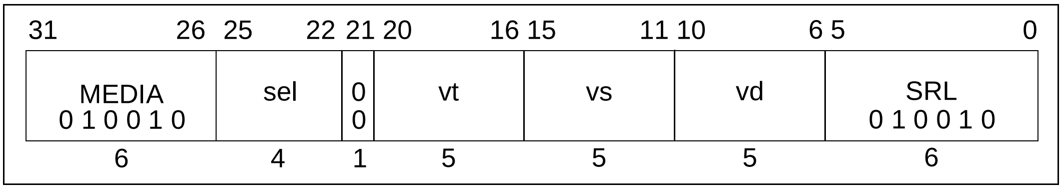
Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← SLLOB($ts_{63..56}$, $tt_{63..56}$)
        || SLLOB($ts_{55..48}$, $tt_{55..48}$)
        || SLLOB($ts_{47..40}$, $tt_{47..40}$)
        || SLLOB($ts_{39..32}$, $tt_{39..32}$)
        || SLLOB($ts_{31..24}$, $tt_{31..24}$)
        || SLLOB($ts_{23..16}$, $tt_{23..16}$)
        || SLLOB($ts_{15..8}$, $tt_{15..8}$)
        || SLLOB($ts_{7..0}$, $tt_{7..0}$)
function SLLOB(ts, tt)
        s ← $tt_{2..0}$
        SLLOB ← $ts_{7-s..0}$ || $0^s$
end SLLOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

SRL.OB Vector Shift Right Logical SRL.OB



Format:

SRL.OB vd, vs, vt

Description:

Each element of vector *vs* is shifted right by an amount specified by an immediate or an element of vector *vt*, and zeros are shifted into the low-order bits. The results are written into vector *vd*. All but the lower 3 bits of the shift amount are masked to 0, so the largest possible shift is 7 places. The *sel* field selects the values of *vt*[] used for each *i*, which must be a scalar or an immediate.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

SRL.OB Vector Shift Right Logical (continued) SRL.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← SRLOB($ts_{63..56}$, $tt_{63..56}$)
        || SRLOB($ts_{55..48}$, $tt_{55..48}$)
        || SRLOB($ts_{47..40}$, $tt_{47..40}$)
        || SRLOB($ts_{39..32}$, $tt_{39..32}$)
        || SRLOB($ts_{31..24}$, $tt_{31..24}$)
        || SRLOB($ts_{23..16}$, $tt_{23..16}$)
        || SRLOB($ts_{15..8}$, $tt_{15..8}$)
        || SRLOB($ts_{7..0}$, $tt_{7..0}$)
function SRLOB(ts, tt)
        s ← $tt_{2..0}$
        SRLOB ← $0^s$ || $ts_{7..s}$
end SRLOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

SUB.OB Vector Subtract SUB.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	SUB 0 0 1 0 1 0
6	4	1	5	5	5	6

Format:

SUB.OB vd, vs, vt

Description:

The difference of the values in vector *vt* and vector *vs* is written into vector *vd*. Saturated arithmetic is performed: overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the specified format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

SUB.OB Vector Subtract (continued) SUB.OB

Operation:

```
ts ← FPR[vs]
tt ← select(fmtsel, vt)
FPR[vd] ← SubOB($ts_{63..56}$, $tt_{63..56}$)
        || SubOB($ts_{55..48}$, $tt_{55..48}$)
        || SubOB($ts_{47..40}$, $tt_{47..40}$)
        || SubOB($ts_{39..32}$, $tt_{39..32}$)
        || SubOB($ts_{31..24}$, $tt_{31..24}$)
        || SubOB($ts_{23..16}$, $tt_{23..16}$)
        || SubOB($ts_{15..8}$, $tt_{15..8}$)
        || SubOB($ts_{7..0}$, $tt_{7..0}$)
function SubOB(ts, tt)
        t ← (0 || ts) - (0 || tt)
        if $t_8$ = 1 then
                SubOB ← $0^8$
        else
                SubOB ← $t_{7..0}$
        endif
end SubOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

WACH.OB Vector Write Accumulator High WACH.OB

31–26	25–22	21	20–16	15–11	10–6	5–0
MEDIA 0 1 0 0 1 0	H 1 0 0 0	0 0	0 0 0 0 0 0	vs	0 0 0 0 0 0	WACH 1 1 1 1 1 0
6	4	1	5	5	5	6

Format:

WACH.OB vs

Description:

This instruction writes the most-significant third of the bits of the Accumulator elements. The least-significant two-thirds of the bits of the Accumulator elements are unaffected.

RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator.

This instruction is the only instruction that writes a portion of the Accumulator. WACL writes all bits in the accumulator, so it must precede WACH when restoring the Accumulator.

No data-dependent exceptions are possible. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

$$\begin{aligned}
\mathrm{ACC} \leftarrow\ & \mathrm{FPR[vs]}_{63..56} \,||\, \mathrm{ACC}_{183..168} \\
& ||\, \mathrm{FPR[vs]}_{55..48} \,||\, \mathrm{AC}_{159..144} \\
& ||\, \mathrm{FPR[vs]}_{47..40} \,||\, \mathrm{AC}_{135..120} \\
& ||\, \mathrm{FPR[vs]}_{39..32} \,||\, \mathrm{AC}_{111..96} \\
& ||\, \mathrm{FPR[vs]}_{31..24} \,||\, \mathrm{AC}_{87..72} \\
& ||\, \mathrm{FPR[vs]}_{23..16} \,||\, \mathrm{AC}_{63..48} \\
& ||\, \mathrm{FPR[vs]}_{15..8} \,||\, \mathrm{AC}_{39..24} \\
& ||\, \mathrm{FPR[vs]}_{7..0} \,||\, \mathrm{ACC}_{15..0}
\end{aligned}$$

WACH.OB Vector Write Accumulator High (continued) WACH.OB

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

WACL.OB Vector Write Accumulator Low WACL.OB

31 … 26	25 … 22	21	20 … 16	15 … 11	10 … 6	5 … 0
MEDIA 0 1 0 0 1 0	L 0 0 0 0	0 0	vt	vs	0 0 0 0 0 0	WACL 1 1 1 1 1 0
6	4	1	5	5	5	6

Format:

WACL.OB vs, vt

Description:

This instruction writes the least-significant two-thirds of the bits of the Accumulator elements. The upper one-third of the bits of the Accumulator elements are written by the sign bits of the corresponding elements of vector *vs*[] and replicated by 8, depending on the format.

RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator.

No data-dependent exceptions are possible. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

Operation:

```
$ACC \leftarrow 0^8 \,||\, FPR[vs]_{63..56} \,||\, FPR[vt]_{63..56}$
    $||\, 0^8 \,||\, FPR[vs]_{55..48} \,||\, FPR[vt]_{55..48}$
    $||\, 0^8 \,||\, FPR[vs]_{47..40} \,||\, FPR[vt]_{47..40}$
    $||\, 0^8 \,||\, FPR[vs]_{39..32} \,||\, FPR[vt]_{39..32}$
    $||\, 0^8 \,||\, FPR[vs]_{31..24} \,||\, FPR[vt]_{31..24}$
    $||\, 0^8 \,||\, FPR[vs]_{23..16} \,||\, FPR[vt]_{23..16}$
    $||\, 0^8 \,||\, FPR[vs]_{15..8} \,||\, FPR[vt]_{15..8}$
    $||\, 0^8 \,||\, FPR[vs]_{7..0} \,||\, FPR[vt]_{7..0}$
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

XOR.OB Vector XOR XOR.OB

31 26	25 22	21	20 16	15 11	10 6	5 0
MEDIA 0 1 0 0 1 0	sel	0 0	vt	vs	vd	XOR 0 0 1 1 0 1
6	4	1	5	5	5	6

Format:

XOR.OB vd, vs, vt

Description:

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical XOR operation. The *sel* field selects the values of *vt*[] used for each *i*.

No data-dependent exceptions are possible. The operands must be values in the *OB* format. If they are not, the results are undefined and the values of the operand vectors become undefined. The result of this instruction is undefined if the processor is executing in 16 FP register mode.

XOR.OB Vector XOR (continued) XOR.OB

Operation:

```
ts ← FPR[vs]
tt ← select(sel, vt)
FPR[vd] ← XorOB(ts$_{63..56}$, tt$_{63..56}$)
        || XorOB(ts$_{55..48}$, tt$_{55..48}$)
        || XorOB(ts$_{47..40}$, tt$_{47..40}$)
        || XorOB(ts$_{39..32}$, tt$_{39..32}$)
        || XorOB(ts$_{31..24}$, tt$_{31..24}$)
        || XorOB(ts$_{23..16}$, tt$_{23..16}$)
        || XorOB(ts$_{15..}$ , tt$_{15..8}$)
        || XorOB(ts$_{7..0}$, tt$_{7..0}$)
function XorOB(ts, tt)
        XorOB ← (0 || ts) xor (0 || tt)
end XorOB
```

Exceptions:

Coprocessor Unusable exception
Reserved Instruction exception

19.4 Multimedia Instruction Opcode Bit Encoding

Figure 19-6 lists the bit encoding for multimedia instructions.

Function (for Opcode = COP2)

bits 5...3 \ 2...0	0	1	2	3	4	5	6	7
0		C.EQ	PICKF	PICKT	C.LT	C.LE	MIN	MAX
1			SUB	ADD	AND	XOR	OR	NOR
2	SLL		SRL					
3	ALNI							SHFL
4	RZU							
5								
6	MUL		MULS{,L}	MUL{A,L}				
7							WAC{H,L}	RAC{H,L}

Figure 19-6 Bit Encoding for Multimedia Instructions

Debug and Test Features

20

This chapter describes the VR5432 processor's debug and test functions, which are intended for the exclusive use of debug software and hardware tools. These functions do not involve the WatchLo and WatchHi registers; instead, they replace and greatly improve on the debug functions implemented by the WatchLo and WatchHi registers.

The debug and other JTAG-accessible registers described here are not architecturally visible parts of the processor. Programs running in Normal mode (User, Supervisor, or Kernel mode) cannot access the debug resources directly. However, a special Normal mode instruction, DBREAK, is provided for accessing Debug mode, and a debug tool attached to the JTAG port can access Debug mode directly.

20.1 Overview

The processor implements both internally and externally accessible debug resources. The externally accessible resources are accessed via the JTAG interface, which complies with IEEE Standards 1149.1 and 1149.1a and implements N-Wire and N-Trace debug enhancements.

The processor's Debug mode is entered when a debug break occurs. The debug functions can be controlled internally or externally, as follows:

- **Internal Access.** A processor-resident debugger program, invoked via the Debug Exception vector, uses *debug instructions* to access the processor's *debug registers*. The DBREAK instruction is provided fo this purpose; when executed, it causes the processor to enter Debug mode.
- **External Access.** An external debug tool, attached to the JTAG test access port (TAP), can access the processor's internal *debug module*, which includes *JTAG-accessible registers* that support JTAG, N-Wire, and N-Trace test interfaces.

Figure 20-1 shows the processor resources accessible to debug programs and external debug tools using the internal- and external-access methods.

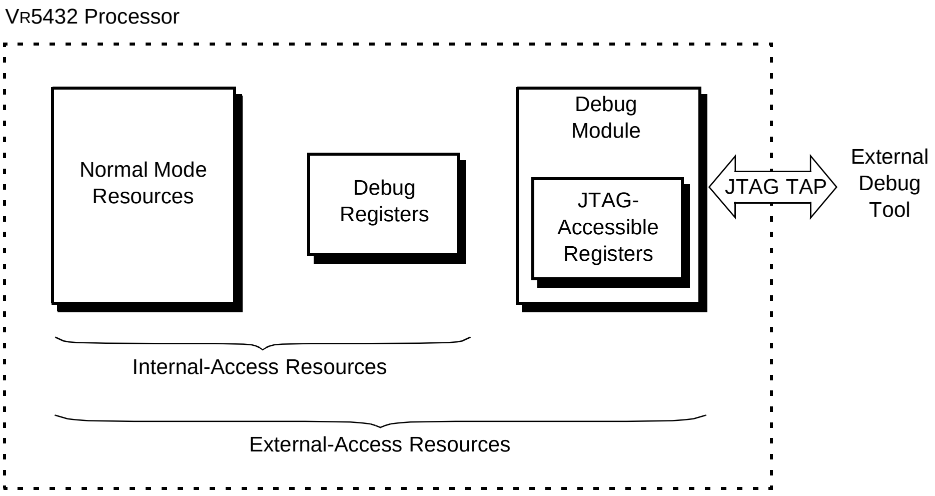


Figure 20-1 Access to Processor Resources in Debug Mode

The *debug registers*—DR0 through DR15—can be accessed by the *debug instructions*—DBREAK, MTDR, MFDR, and DRET—in either the internal- or external-access Debug mode. These registers and instructions give software or hardware the ability to break the processor, modify its state or set breakpoints, and resume running, or to break the processor, single-step, and resume running.

The N-Wire and N-Trace interfaces, available in external-access Debug mode, support comprehensive hardware and software breakpoints and trace functions. They use a *monitor* mechanism that gives debug tools access to all system resources, including the processor's user and debug registers, program counter, register file, caches, external memory, and I/O. For example, an external-access debug tool can download data via the JTAG port into external memory, return to normal operation mode, and monitor the result of execution using this data. The N-Wire functions provide run-time control and access to the processor's internal state. The N-Trace functions support instruction-execution tracing via trace packets on the trace signals. Both functions share a set of *JTAG-accessible registers*. In the VR5432 implementation, "N" equals 4, as represented by the four TrcData [3:0] signals.

Because an external-access debug tool can access both the debug registers and the JTAG-accessible registers, the external-access Debug mode provides more control of the processor than does internal-access Debug mode.

20.2 Definition of Terms

Debug Break. An event that causes the processor to asynchronously leave Normal mode (User, Supervisor, or Kernel mode) execution and enter Debug mode. The terms "break" and "debug break" are used interchangeably. Section 20.3 defines all possible debug break events.

Debug Exception Vector. Address 0xFFFF FFFF BFC0 1000. The DBREAK instruction is designed for accessing this vector.

Debug Instructions. DBREAK, MTDR, MFDR, and DRET, as described in Section 20.4.1.

Debug Mode. The processor enters Debug mode as the result of a debug break. If the debug module is in reset at the debug break, the processor begins executing internal-access resident debugger instructions, starting at the Debug Exception vector address. If the debug module is not in reset at the debug break, the processor begins executing external-access instructions from the JTAG-accessible N-Wire Monitor Instruction (MON_INST) register. Although Debug mode can be entered and controlled via internal or external access, external access supports maximum control of the processor. See also *Normal Mode*, *Debug Module,* and Section 20.3.

Debug Mode Registers. The internally accessible debug registers and the JTAG-accessible registers.

Debug Module. A module inside the processor that supports external access to the debug features via the JTAG port. The debug module contains the JTAG, N-Wire, and N-Trace interfaces. See Figure 20-1.

Debug Module Reset. The processor state in which external access to the processor's debug module is disabled. This reset is unrelated to the processor reset (Reset*). The debug module is enabled and disabled with the *DINIT* bit in the JTAG-accessible N-Wire Debug Module System (DM_SYSTEM) register. In the internal-access method, the debug module is disabled (in reset), and the processor can enter Debug mode by executing the DBREAK instruction. In the external-access method, the debug module is enabled (not in reset). Compare to *Debug Reset*.

Debug Registers. The registers accessible with the *debug instructions*. These registers include DR0 through DR15 (DRCNTL, DEPC, DDATA0, DDATA1, IBC, DBC, IBA, IBAM, DBA, DBAM, DBD, and DBDM). All of the debug registers are accessible directly in the internal-access Debug mode, and they are accessible either directly or indirectly in the external-access Debug mode. The debug registers are described in Section 20.4.2. These registers overlap (share registers or copy register bits) with the JTAG-accessible registers described in Section 20.5.2.

Debug Reset. A reset to the processor from the debug module, accomplished by externally setting the *RESET* bit in the N-Wire Debug Module Control (DM_CONTROL) register. The effect of a debug reset on the processor is the same as asserting Reset*. Compare to *Debug Module Reset*.

External Access. Debug access to processor resources and operations by an external debug tool through the JTAG port and the on-chip debug module, which supports JTAG, N-Wire, and N-Trace debug functions (see Figure 20-1). In the external-access method, a debug tool can access all of the debug registers and all of the JTAG-accessible registers. External access thus provides more control of processor resources than does internal access.

Hardware Breakpoint. An instruction address, data address, or data-data breakpoint specified in the debug registers or the JTAG-accessible registers. The hardware breakpoint registers are shared between the debug and JTAG-accessible register sets.

Internal Access. Debug access to processor resources and operations via a resident debugger program invoked at the processor's Debug Exception vector address (the DBREAK instruction is provided for this purpose). The resident debugger program can use the debug instructions MTDR and MFDR to access the processor state, set breakpoints, and single-step.

JTAG-Accessible Registers. The registers accessible in external-access Debug mode. They include the three required JTAG registers (Instruction, Bypass, and Boundary Scan), plus registers to support the N-Wire and N-Trace debug functions (DM_SYSTEM, DM_CONTROL, MON_INST, MON_DATA, TRCSYS, and most of the internal-access debug registers). These registe rsare described in Section 20.5.2.

Monitor. A JTAG-accessible mechanism for accessing all system resources, including the processor's Normal mode and Debug mode registers, cache, external memory, and I/O.

Normal Mode. User, Supervisor, or Kernel mode. The processor is also in Normal mode when it is in Reset or is being reset by the debug module. See also *Debug Mode*.

Resident Debugger. An optional program that can be accessed internally via the Debug Exception vector (the DBREAK instruction is designed for this purpose). This program provides system access to most (but not all) of the processor's debug features when there is no attached debug tool or the debug tool is in Reset.

Trigger. The BkTgIO* output signal.

Trigger Event. An event that causes assertion of the BkTgIO* output signal. Such events can include:

- An enabled hardware breakpoint
- An enabled debug break

20.3 Debug Mode

The processor enters Debug mode as a result of one of the following possible debug break events:

- Internal-access debug break events
  - Execution of the DBREAK instructio
  - Setting th *STEP* bit in the DRCNTL debug register (DR0)
  - Reaching an instruction-address, data-address, or data-data breakpoint specified in the debug registers
- External-access debug break events
  - Assertion o the BkTgIO* signal, when it is configured for inpu
  - Setting of th *BREAK* bit in the JTAG-accessible N-Wire Debug Module Control (DM_CONTROL) register
  - Setting th *STEP* bit in either the DRCNTL register or th DM_CONTROL register
  - Reaching an instruction-address, data-address, or data-data breakpoint specified in the debug registers

Debug mode is entered regardless of the state of the debug module.

- If the *debug modul* is *in Reset* (*DINIT* bit set to 1 in the N-Wire DM_SYSTEM register), the processor begins executing internal-access resident debugger instructions starting at the Debug Exception vector address. In this case, the DRCNTL register controls Debug mode operations.
- If the *debug modul* is *not in Reset* (*DINIT* bit cleared to 0 in the N-Wire DM_SYSTEM register), the processor begins executing external-access instructions from the N-Wire Monitor Instruction (MON_INST) register, if execution is enabled. In this case, the JTAG port controls Debug mode operations.

When Debug mode is entered, all incomplete instructions are flushed from the pipeline, all outstanding external bus transactions are completed, execution transitions to Debug mode at an instruction boundary, the program counter (PC) is saved in the DEPC debug register, and execution is redirected to the 64-bit Debug Exception vector (location 0xFFFF FFFF BFC0 1000). There may be a delay entering Debug mode to allow the pipeline flush and to allow all outstanding external transactions to complete; if so, the processor stalls during this time.

The processor will not enter Debug mode at a branch delay slot instruction boundary. Instead it stops at the branch instruction or the target of the branch. If a software or hardware breakpoint occurs for the branch delay slot instruction, the breakpoint occurs at the corresponding Branch instruction. If a single-step break is executed on a Branch instruction, both the branch and its delay slot are executed.

Instructions that redirect the PC (e.g., branches) are not allowed to be executed in the MON_INST register when the debug module is in reset. Any attempt to do so results in undefined behavior. Instructions that redirect the PC are allowed if the debug module is not in reset.

While in Debug mode, the processor behaves as if it is in Kernel mode (CP0 Status *EXL* = 1), although entering Debug mode does not set the *EXL* bit. All interrupts are disabled, including NMI*, and any debug break events are ignored. If a Load or Store instruction causes an exception in Debug mode, the exception is processed as if the processor is in Kernel mode. The *DM_EXCEPT* bit in the relevant Debug Control register (DRCNTL for internal access, or DM_CONTROL for external access) indicates whether an exception occurred. If any instruction other than Load or Store causes an exception, the results and processor state are undefined.

The processor returns to Normal mode from Debug mode by executing a DRET instruction. The processor vectors the PC to the address in the Debug Exception PC (DEPC) register.

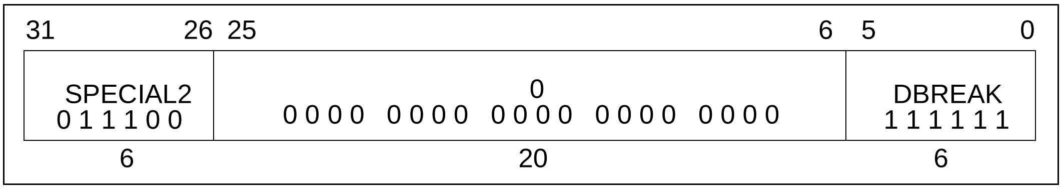
20.4 Internal Access

In the processor’s internal-access Debug mode, a resident debugger program can use the debug instructions to access all of the debug registers. These instructions and registers are (with a few exceptions) also available to external-access Debug mode, as described in Section 20.5.

20.4.1 Debug Instructions

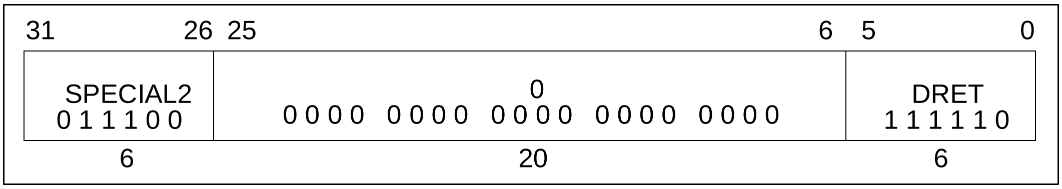
The DBREAK, DRET, MTDR and MFDR instructions are unique to the processor’s debug features. Except for DBREAK, these instructions are accessible only when the processor is in Debug mode; executing them in Normal mode causes a Reserved Instruction trap.

20.4.1.1 DBREAK: Debug Break



The DBREAK instruction forces entry into Debug mode by causing a trap to the Debug Exception vector address (0xFFFF FFFF BFC0 1000). This instruction may only be executed in Normal (User, Supervisor, or Kernel) mode. Execution in Debug mode results in undefined behavior.

20.4.1.2 DRET: Debug Return



The DRET instruction returns from Debug mode to the mode in effect (User, Supervisor, or Kernel mode) when the last debug break occurred. Control is passed to the instruction pointed to by the Debug Exception PC (DEPC) register. Unlike most jumps and branches, the execution of which also executes the next instruction (the one in the delay slot), DRET does not execute a delay slot instruction. The DRET instruction must not be placed in a branch delay slot.

20.4.1.3 MTDR: Move to Debug Register

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 0 1 1 1 0 0	MTDR 0 0 1 0 0	rt	dr	0 0 0 0 0 0	Debug Move 1 1 1 1 0 1
6	5	5	5	5	6

This instruction moves the contents of general register *rt* into debug register *dr*.

20.4.1.4 MFDR: Move from Debug Register

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 0 1 1 1 0 0	MFDR 0 0 0 0 0	rt	dr	0 0 0 0 0 0	Debug Move 1 1 1 1 0 1
6	5	5	5	5	6

This instruction moves the contents of debug register *dr* into general register *rt*.

20.4.2 **Debug Registers**

Table 20-1 lists the debug registers. The *DME* bit in the CP0 Status register is only accessible in Normal mode via Normal mode instructions. All of the debug registers except the *DME* bit are accessible in both the internal-access and external-access Debug modes via the MFDR and MTDR instructions. Unless otherwise specified, the contents of the debug registers are undefined after a processor cold reset.

Table 20-1 Debug Registers

Register Mnemonic	Register Name	Register Number	Register Width (Bits)	Register-Set Membership[1]
DME	*DME* bit in the CP0 Status register	—	1	Internal
DRCNTL	Debug Register Control register	DR0	32	Internal
DEPC	Debug Exception PC register	DR1	64	Internal
DDATA0	Debug Data Monitor 0 and Monitor Data register	DR2	64	Internal and external
DDATA1	Debug Data Monitor 1 register	DR3	64	Internal
IBC	Instruction Breakpoint Control/Status register	DR4	32	Internal and external
DBC	Data Breakpoint Control/Status register	DR5	32	Internal and external
—	*Reserved*	DR6		—
—	*Reserved*	DR7		—
IBA	Instruction Breakpoint Address register	DR8	64[2]	Internal and external
IBAM	Instruction Breakpoint Address Mask register	DR9	64[2]	Internal and external
—	*Reserved*	DR10		—
—	*Reserved*	DR11		—
DBA	Data Breakpoint Address register	DR12	64	Internal and external
DBAM	Data Breakpoint Address Mask register	DR13	64	Internal and external
DBD	Data Breakpoint Data register	DR14	64	Internal and external
DBDM	Data Breakpoint Data Mask register	DR15	64	Internal and external

Notes:

1. All debug registers except DME are accessible in both the internal-access and external-access Debug modes via the MFDR and MTDR instructions. However, the registers marked “internal and external” are actually shared by the internal-access and external-access register sets.
2. Only 40 bits of the virtual address, plus the region bits (63:62), are compared. The unused address bits must be sign extended to bit 61 for all address spaces, except *xkphys*. For *xkphys* address space, bits 61:59 must also indicate the correct cacheability attribute, because these bits are compared.

20.4.2.1 Debug Mode Enable (DME) bit in the CP0 Status register

Bit 24 is the *Debug Mode Enable* (*DME*) bit in the *Diagnostic Status* (*DS*) field of the CP0 Exception Processing Status register (see Section 6.2.5 on page 97). It indicates to the processor that there is a resident debugger program at the Debug Exception vector. The bit is only accessible in Normal mode via Normal mode instructions, and it is only meaningful when the debug module is in reset.

- *DME* = 0: A debug break event does not cause the processor to enter Debug mode. The DBREAK instruction causes a Reserved Instruction exception instead of a debug break
- *DME* = 1: A debug break event causes the processor to enter Debug mode.

20.4.2.2 DRCNTL: Debug Register Control register (DR0)

The DRCNTL register is accessible only to internal-access resident debugger programs. It duplicates a subset of bits from two external-access registers that constitute part of the N-Wire interface—the Debug Module System (DM_SYSTEM) register and the Debug Module Control (DM_CONTROL) register, described in Section 20.5.2.5 and Section 20.5.2.6.

Although the *DRCNTL* bits duplicate some of the DM_SYSTEM and DM_CONTROL bits, the *DRCNTL* bits are a separate set of bits; they are not shared by the DM_SYSTEM and DM_CONTROL registers. Either the DRCNTL register is active or the two external-access registers are active; use of these registers is mutually exclusive. DRCNTL is used when the debug module is in reset (i.e., for internal access). The two external-access registers are used when the debug module is not in reset. The *DINIT* bit in the DM_SYSTEM register determines whether the debug module is in reset.

Figure 20-2 shows the register format. Table 20-2 describes the register fields.

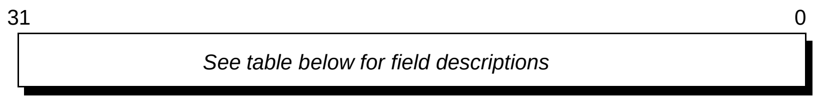


Figure 20-2 Debug Register Control (DRCNTL) Register Format

Table 20-2 Debug Register Control Register (DRCNTL) Fields

Bits	Field	Description
1:0	*Reserved*	—
2	MRST	Mask User Reset in Debug mode 1 → Ignores Reset* input while in Debug mode 0 → Accepts Reset* input while in Debug mode Defaulted to 1 at the debug module initialization.
3	MNMI	Mask User NMI* 1 → Ignores NMI* 0 → Accepts NMI* Defaulted to 0 at the debug module initialization.
4	MINT	Mask User Interrupts 1 → Ignores user interrupt input 0 → Accepts user interrupt input Defaulted to 0 at the debug module initialization. MINT effects interrupts via the Int signals or via an external write. Software interrupts are not masked.
5	STEP	Single-Step Break Single-step allows the user to execute one Normal mode instruction. Single-step occurs after a DRET instruction. The processor returns to Normal mode, executes a single instruction, and breaks back into Debug mode. 1 → Enables single-step break (single-step mode) 0 → Disables single-step break Defaulted to 0 at the debug module initialization.
13:6	BRK_CAUSE	Break Cause This consists of multiple bits. One bit is assigned for each break cause and a corresponding bit is set when the break occurred. Multiple bits are set if the break occurred by multiple break causes. The bit assignments are defined as follows: Bit 6 → External break Bit 7 → Single-step Bit 8 → Software breakpoint Bit 9 → *Reserved* Bit 10 → *Reserved* Bit 11 → Instruction-address breakpoint Bit 12 → Data access (address or data) breakpoint Bit 13 → *Reserved*

Table 20-2 Debug Register Control Register (DRCNTL) Fields (continued)

Bits	Field	Description
14	DBM	Debug Mode Indicates Debug mode or Normal mode (read only). 1 → Normal mode (User, Supervisor, or Kernel mode) 0 → Debug mode
17:15	CPU_STAT	Processor Status 0 0 0 → Reset (highest) 0 0 1 → *Reserved* 0 1 0 → *Reserved* 0 1 1 → *Reserved* 1 0 0 → *Reserved* 1 0 1 → *Reserved* 1 1 0 → *Reserved* 1 1 1 → Normal (lowest)
20:18	*Reserved*	—
21	DM_EXCEPT	Debug Mode Exception Indicates that an exception occurred while in Debug mode (read/write). Read 1 → Instruction executed in Debug mode caused exception Read 0 → No exception in Debug mode since flag was cleared Write 1 → No operation Write 0 → Clear exception flag If any instruction other than Load or Store causes an exception, the results and processor state are undefined.
22	BKIODIR	BkTgIO* direction Indicates the direction of the BkTgIO* signal. 1 → Input 0 → Output Defaulted to 1 at the debug module initialization.
23	BKIOEN	BkTgIO* Break Enable 1 → Enable driving of BkTgIO* trigger output at a debug break event, or to break the processor at a BkTgIO* break input 0 → Disable Defaulted to 0 at the debug module initialization.
24	BKIOTEN	BkTgIO* Trigger Enable (read/write) 1 → Enable detected internal trigger events to the BkTgIO* signal when it is configured in the output direction 0 → Disable Defaulted to 0 at the debug module initialization.
31:25	*Reserved*	—

20.4.2.3 DEPC: Debug Exception PC (DR1)

When entering Debug mode, the DEPC register contains the virtual address of the instruction where the debug break occurred. This is the address at which Normal mode instruction processing may resume after exiting Debug mode. Figure 20-3 shows the register format.

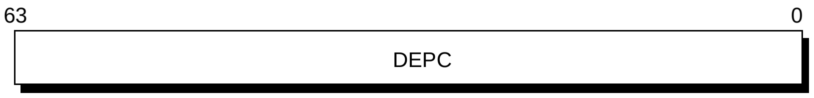


Figure 20-3 Debug Exception PC (DEPC) Register Format

20.4.2.4 DDATA0: Debug Data Monitor 0 and Monitor Data (DR2)

The DDATA0 register and the JTAG-accessible Monitor Data (MON_DATA) register (see Section 20.5.2.8) are the same register. The register is used for external access when the debug module is active, and therefore is scannable. It can also be used as a scratch register in Debug mode. The user is responsible for ensuring that the types of use for the register do not overlap. Figure 20-4 shows the register format.

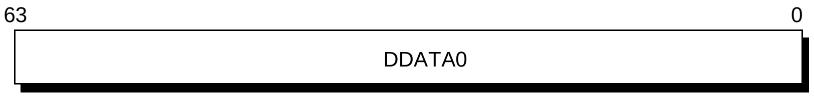


Figure 20-4 Debug Data Monitor 0 (DDATA0) Register Format

20.4.2.5 DDATA1: Debug Data Monitor 1 (DR3)

The DDATA1 register can be used as a stack pointer or scratch register. Figure 20-5 shows the register format.

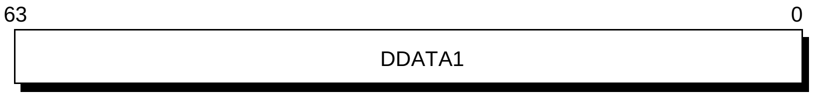


Figure 20-5 Debug Data Monitor 1 (DDATA1) Register Format

20.4.2.6 Instruction Address Breakpoint

Instruction address hardware breakpoints are supported by three registers: IBC, IBA, and IBAM. These three registers are used in both internal-access and external-access Debug mode.

To determine an instruction address match, the program counter is compared with the Breakpoint instruction address before TLB translation. If the breakpoint condition is met and the break is enabled in the IBC register, the processor enters Debug mode. The instruction that caused the breakpoint is not executed.

The VR5432 implementation of instruction address breakpoints has the following limitations:

- Only doubleword addresses can be compared (IBAM[2:0] must be 111_2) for instruction address breakpoints.
- Triggers (BkTgIO* trigger output) are not supported for instructio address breakpoints
- Only 40-bit virtual addresses are supported for instruction address breakpoints.

The following registers are used to set an instruction address breakpoint.

IBC: Instruction Breakpoint Control/Status register (DR4)

The IBC register is the control and status register for the instruction-address breakpoint. Figure 20-6 shows the register format. Table 20-3 describes the register fields.

Figure 20-6 Instruction Breakpoint Control/Status (IBC) Register Format

Table 20-3 Instruction Breakpoint Control/Status (IBC) Register Fields

Bits	Name	Description
0	BS	Breakpoint Status 1 → Breakpoint match occurred. 0 → Breakpoint match did not occur. Cleared to 0 on cold reset.
1	BE	Break Enable. Causes a debug break when a breakpoint match occurs. 1 → Enabled. 0 → Disabled. Cleared to 0 on cold reset.
2	*Reserved*	—
3	INV	Invert address match condition 1 → Address matches when conditions don't match. 0 → Address matches when conditions match.
4	ASIDM	ASID compare mask 1 → Address match is not qualified with ASID matching. 0 → Address match is qualified with ASID matching the current processor ASID.
12:5	ASID	Address Space ID to compare.
31:13	*Reserved*	—

IBA: Instruction Breakpoint Address register (DR8)

The IBA register contains the address of the instruction breakpoint. When the instruction stored at the specified address is being executed, the condition is met. Figure 20-7 shows the register format.

Even though a 64-bit IBA register is specified, only 40 bits of the virtual address, plus the region bits (63:62) are compared. The unused address bits must be sign extended to bit 61 for all address spaces except *xkphys*. For *xkphys* address space, bits 61:59 must also indicate the correct cacheability attribute, because these bits are compared. Please refer to the memory mapping and address space discussions in Chapter 4.

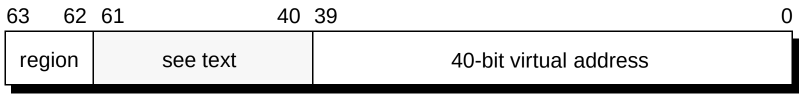


Figure 20-7 Instruction Breakpoint Address (IBA) Register Format

IBAM: Instruction Breakpoint Address Mask register (DR9)

The IBAM register contains the mask for IBA. If a bit of this register is 1, the corresponding bit of IBA is not compared. Figure 20-8 shows the register format.

As with the IBA register, even though a 64-bit IBAM register is specified, only 40 bits of the virtual address, plus the region bits (63:62), are compared. The unused address bits must be sign extended to bit 61 for all address spaces except for *xkphys*. For *xkphys* address space, bits 61:59 must also indicate the correct cacheability attribute, because these bits are compared.

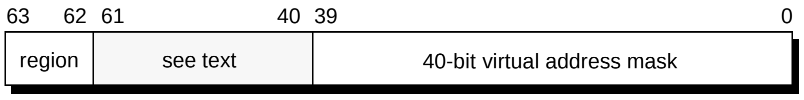


Figure 20-8 Instruction Breakpoint Address Mask (IBAM) Register Format

20.4.2.7 Data Access Breakpoint

Data access hardware breakpoints (break on address, break on data, or break on both) are supported by five registers: DBC, DBA, DBAM, DBD, and DBDM. These five registers are used in both internal-access and external-access Debug mode.

To determine a data instruction address match, the program counter is compared with the breakpoint instruction address before TLB translation. If the breakpoint condition is met and the break is enabled, the processor enters Debug mode. If only a data address condition is specified, the instruction that caused the breakpoint is not executed. If a data access condition (load or store) is specified in the DBC register, the break occurs sometime after the instruction that caused the breakpoint. If the breakpoint condition is met and the trigger is enabled in the DBC register, the processor asserts a trigger on BkTgIO* output.

The VR5432 implementation of data access breakpoints has the following limitations and features:

- For data access store breakpoints, only doubleword addresses can be compared (IBAM[2:0] must be 111_2).
- For data access load breakpoints, data access sizes other than 64 bits are supported.
- Only 40-bit virtual addresses are supported for data access breakpoints.

The processor supports data access sizes other than 64 bits. For loads, the DBDM register must mask all bits that are not part of the data access size, or the DBD register must specify the proper sign-extended 64-bit value. For stores, only data access for doublewords is supported.

The following registers are used to set a data access breakpoint.

DBC: Data Breakpoint Control/Status register (DR5)

The DBC register provides control and status for the data address and data access breakpoints. Figure 20-9 shows the register format. Table 20-4 describes the register fields.

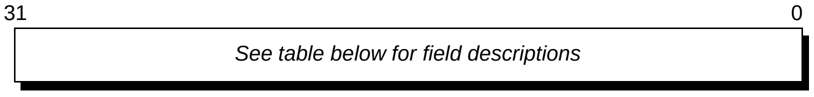


Figure 20-9 Data Breakpoint Control/Status (DBC) Register Format

Table 20-4 Data Breakpoint Control/Status (DBC) Register Fields

Bits	Name	Description
0	BS	Breakpoint Status 1 → Breakpoint match occurred 0 → Breakpoint match did not occur Cleared to 0 on cold reset.
1	BEA	Break Enable at Address Match Causes a debug break when the address match condition is met. 1 → Enabled 0 → Disabled
2	TEA	Trigger Enable at Address Match Outputs a trigger on BkTgIO* when the address match condition is met. 1 → Enabled 0 → Disabled
3	AINV	Invert Address-Match Condition 1 → Address matches when conditions don’t match 0 → Address matches when conditions match
4	ASIDM	ASID Compare Mask 1 → Address match is not qualified with ASID matching 0 → Address match is qualified with ASID matching the current processor ASID
12:5	ASID	Address Space ID to compare
15:13	*Reserved*	—

Table 20-4 Data Breakpoint Control/Status (DBC) Register Fields (continued)

Bits	Name	Description
16	TS	Trigger Status 1 → Trigger occurred 0 → Trigger has not occurred
17	BED	Break Enable at Data Match Causes a debug break when the data condition is met. 1 → Enabled 0 → Disabled BEA and BED are in effect if either BERD or BEWR are set. In this case, if both BEA and BED are set, a break occurs only when both conditions are met. If both are cleared, a break occurs regardless of the compare results.
18	TED	Trigger Enable at Data Match Outputs a trigger on BkTgIO* when the data match condition is met. 1 → Enabled 0 → Disabled TEA and TED are in effect if either TERD or TEWR is set. In this case, if both TEA and TED are set, a trigger is output when both conditions are met. If both are cleared, a trigger is output regardless of the compare results.
19	DINV	Invert Data Match Condition 1 → A data match occurs when conditions don't match 0 → A data match occurs when conditions match
20	BERD	Break Enable for Read Access (i.e., Loads) 1 → Break enabled for loads 0 → No break enabled for loads Cleared to 0 on cold reset.
21	BEWR	Break Enable for Write Access (i.e., Stores) 1 → Break enabled for stores 0 → No break enabled for stores Cleared to 0 on cold reset. If neither BERD nor BEWR are set, no data access debug break occurs. These are the primary break enable bits.

Table 20-4 Data Breakpoint Control/Status (DBC) Register Fields (continued)

Bits	Name	Description
22	TER	Trigger Enable for Read Access (i.e., Loads) 1 → Trigger enabled for loads 0 → No trigger enabled for loads Cleared to 0 on cold reset.
23	TEWR	Trigger Enable for Write Access (i.e., Stores) 1 → Trigger enabled for stores 0 → No trigger enabled for stores Cleared to 0 on cold reset. If neither TERD nor TEWR is set, no data access trigger occurs. These are the primary trigger-enable bits.
31:24	*Reserved*	—

DBA: Data Breakpoint Address register (DR12)

The DBA register contains the address of the data breakpoint. When the instruction stored in the specified address is being executed, the condition is met. Figure 20-10 shows the register format.

Even though a 64-bit DBA register is specified, only 40 bits of the virtual address, plus the region bits (63:62) are compared. The unused address bits must be sign-extended to bit 61 for all address spaces except for *xkphys*. For *xkphys* address space, bits 61:59 must also indicate the correct cacheability attribute, because these bits are compared. Please refer to the memory mapping and address space discussions in Chapter 4.

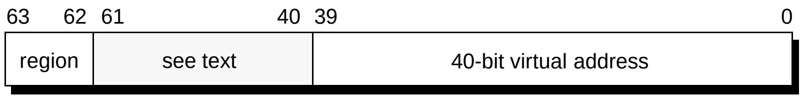


Figure 20-10 Data Breakpoint Address (DBA) Register Format

DBAM: Data Breakpoint Address Mask register (DR13)

The DBAM register contains the bit mask for DBA. If a bit of this register is 1, the corresponding bit of DBA is not compared. Figure 20-11 shows the register format.

As with the DBA register, even though a 64-bit DBAM register is specified, only 40 bits of the virtual address, plus the region bits (63:62), are compared. The unused address bits must be sign extended to bit 61 for all address spaces except for *xkphys*. For *xkphys* address space, bits 61:59 must also indicate the correct cacheability attribute, because these bits are compared.

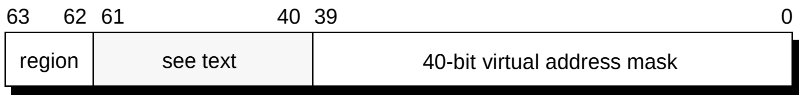


Figure 20-11 Data Breakpoint Address Mask (DBAM) Register Format

DBD: Data Breakpoint Data register (DR14)

The DBD register contains the data of the data breakpoint. The break condition is met when this data is read or written. Figure 20-12 shows the register format.

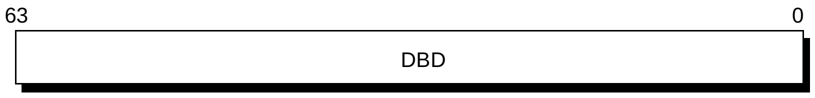


Figure 20-12 Data Breakpoint Data (DBD) Register Format

DBDM: Data Breakpoint Data Mask register (DR15)

The DBDM register contains the bit mask for DBD. If a bit of this register is 1, the corresponding bit of DBD is not compared. For partial word or partial doubleword operations, the unused bits must be masked. Figure 20-13 shows the register format.

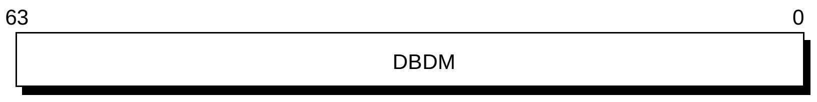


Figure 20-13 Data Breakpoint Data Mask (DBDM) Register Format

20.5 External Access

In the processor's external-access Debug mode, an external debug tool controls processor operations through the JTAG test access port (TAP). The JTAG port supports not only JTAG testing but also N-Wire and N-Trace testing. In external access, the debug tool can access all Debug mode registers, including the debug registers (Section 20.4.2) available to an internal-access resident debugger program and the JTAG, N-Wire, and N-Trace registers described below. This access to debug resources gives external-access Debug mode more control of the processor than does internal-access Debug mode.

20.5.1 JTAG Port Signals

20.5.1.1 Signal summary

Table 20-5 JTAG Test Access Port Signal

Name	Definition	Direction	Description
JTCK	JTAG Test Clock input	Input	The processor accepts a serial clock on the JTCK input. At the rising edge of JTCK, both JTDI and JTMS are sampled. The maximum frequency of JTCK is 33 MHz, and it runs asynchronously to the processor clock, SysClock. The ratio of SysClock to JTCK must be at least 4:1 for proper N-Wire and N-Trace synchronization.
JTMS	JTAG Test Mode Select	Input	The JTAG command signal. It is decoded by the TAP controller to control test operations. The signal has an internal pull-up so that its level is High when the debug tool is not connected.
JTDI	JTAG Test Data In	Input	Data is serially scanned in through this signal. The signal has an internal pull-up so that its level is High when the debug tool is not connected.
JTDO	JTAG Test Data Out	Output	Data is serially scanned out through this signal on the falling edge of JTCK. Per the IEEE-1149.1 standard, the JTDO output is tristated unless data is actively being scanned.
TrcData (3:0)	N-Trace Data Port	Output	This bus is used for output of all trace packets generated as a result of processor execution. Trace packets can consist of one or more clock cycles of data on this bus.
TrcEnd	N-Trace End	Output	Assertion of this signal indicates the end of a trace packet on the TrcData (3:0) bus.

Table 20-5 JTAG Test Access Port Signals (continued)

Name	Definition	Direction	Description
TrcClk	N-Trace Clock	Output	This clock is generated for the benefit of test equipment that requires a clock reference for trace information. It runs at the same frequency as SysClock.
RMode*, BkTgIO*	N-Wire Reset Mode, or N-Wire Break or Trigger I/O	Input/ output	This pin supports two N-Wire signals: debug reset (RMode*), and debug break or trigger (BkTgIO*). During assertion of ColdReset*, the pin carries the RMode* input signal. In all other states the pin carries the BkTgIO* debug break input or debug trigger output signal, depending on its setup in various debug registers (Section 20.4.2) and JTAG-accessible registers (Section 20.5.2). The pin operates at SysClock frequency and must be driven synchronously with SysClock. The pin has an internal pull-up so that its level is High when the debug tool is not connected. See Section 20.5.1.2 and Section 20.5.1.3 for details.
Tristate	Tristate Outputs	Input	This signal floats all processor outputs to allow isolation for board-level tests.

20.5.1.2 Reset mode (RMode*) signal

When ColdReset* is deasserted, the RMode* input is sampled to set the value of the *RESET* bit in the Debug Module Control (DM_CONTROL) register, which is the N-Wire debug reset variable. The RMode* value initializes the *RESET* bit as follows:

- **Low.** Enables debug reset (when RMode* is sampled low, the *RESET* bit is set to 1). The debug module asserts debug reset to th processor. The effect on the processor of asserting RMode* is the same as asserting Reset*.
- **High.** Disables debug reset. The debug module does not assert debu reset to the processor.

20.5.1.3 Break or Trigger I/O (BkTgIO*) signal

After ColdReset* is deasserted, BkTgIO* acts as a debug break input or a debug trigger output. The direction of the BkTgIO* signal defaults to input at debug module initialization, but its direction can thereafter be configured in the JTAG-accessible DM_SYSTEM register (see Section 20.5.2.5).

If the signal is configured for *output*, it can be enabled to act as a trigger to an external debug tool, or it indicates whether the processor is currently in Debug mode:

- **Low (1 cycle pulse)** The debug module has detected one or more processor internal trigger events
- **Low (> 1 cycle).** The processor is in Debug mode
- **High.** The processor is operating in Normal mode (User, Supervisor, or Kernel mode).

Since the processor is a superscalar core running at a higher frequency than the system interface, trigger events can occur much faster than BkTgIO* can report them. Trigger events can be reported at the maximum rate of one every two SysClock cycles (1 cycle pulse). All trigger events that have occurred since the last BkTgIO* trigger output are reported in one trigger. If the processor enters Debug mode, any trigger events that have not been reported will not be reported.

If the signal is configured for *input*, it acts as a debug break from an external debug tool that can force the processor from Normal mode (User, Supervisor, or Kernel mode) to Debug mode:

- **Low.** Break request, forces processor into Debug mode
- **High.** Maintain current processor mod

The debug break request needs to be asserted for only one cycle. The processor enters Debug mode as soon as it is conveniently possible. If the processor is already in Debug mode, or if there is already an outstanding debug break request, a subsequent debug break request has no effect.

20.5.1.4 Board connector for debug tool

System designers are encouraged to incorporate into their board design a 26-pin high-density connector that provides 13 signals and 13 grounds. This assures maximum performance and eliminates noise problems. The target connector is a 0.05”-pitch 26-pin header connector, Samtec part number FTSH-113-01-L-D (through-hole) or FTSH-113-01-L-DV (surface mount), or equivalent. The 26 pins are allocated to 12 signals (and one spare) and 13 grounds. The connector spacing is a convenient 0.05” x 0.05” and provides easy cabling to external equipment.

Alternatively, there is a 10-pin connector option. This smaller connector contains only the basic JTAG boundary scan TAP signals and excludes the real-time trace-related signals.

Figure 20-14 shows the two connectors. Table 20-6 and Table 20-7 list their pinouts.

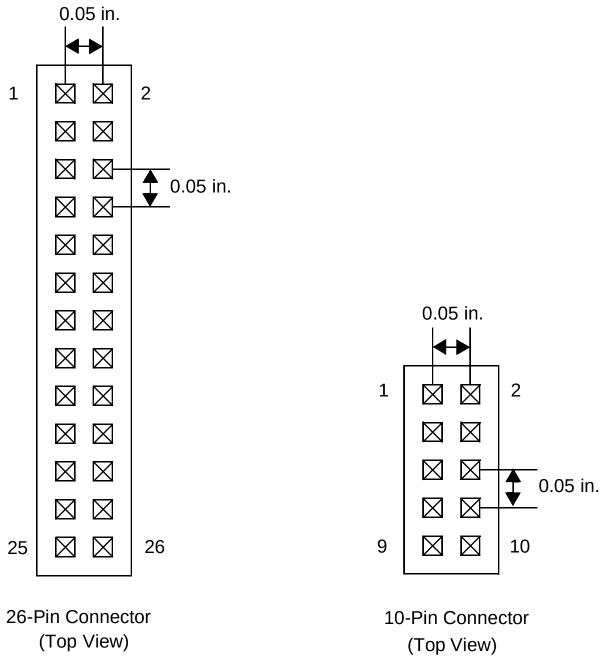


Figure 20-14 JTAG Connector Types

Table 20-6 26-Pin JTAG Connector Signals

Pin	Signal	I/O	Target Termination
1	*Reserved*	—	—
3	JTDI	Input	1-KOhm pull-up resistor
5	JTDO	Output	33-ohm series resistor
7	JTMS	Input	1-KOhm pull-up resistor
9	JTCK	Input	1-KOhm pull-up resistor
11	Tristate	Input	1-KOhm pull-down resistor
13	RMode*/ BkTgIO*	Input/Output	1-KOhm pull-up resistor
15	TrcData 0	Output	33-ohm series resistor
17	TrcData 1	Output	33-ohm series resistor
19	TrcData 2	Output	33-ohm series resistor
21	TrcData 3	Output	33-ohm series resistor
23	TrcEnd	Output	33-ohm series resistor
25	TrcClk	Output	33-ohm series resistor

Table 20-7 10-Pin JTAG Connector Sign a l

Pin	Signal	I/O	Target Termination
1	*Reserved*	—	—
3	JTDI	Input	1-KOhm pull-up resistor
5	JTDO	Output	33-ohm series resistor
7	JTMS	Input	1-KOhm pull-up resistor
9	JTCK	Input	1-KOhm pull-up resistor

In addition to the above debug port connector, system designers may also want to include a 208-pin PQFP test socket. The socket or connector should have the exact pinout, shape, and layout of the actual 208-pin PQFP processor chip and should be placed as close as possible to the processor chip. This extra socket or connector enables connection to a logic analyzer preprocessor between the target board and the processor without having to remove the processor from the board. The preprocessor can then support full visibility to all external processor signals, as well as real-time trace and inverse assembly.

20.5.2 JTAG-Accessible Registers

Table 20-8 lists the registers accessible by a debug tool through the JTAG port. These registers support JTAG, N-Wire, and N-Trace functions.

Table 20-8 JTAG-Accessible Regist e r

Mnemonic	Register Name	Width (Bits)
—	JTAG Instruction register	5
—	JTAG Bypass register	1
—	JTAG Boundary Scan register	109
—	Processor Type register	25
DM_SYSTEM	N-Wire Debug Module System register	7
DM_CONTROL	N-Wire Debug Module Control register	22
MON_INST	N-Wire Monitor Instruction register	64
MON_DATA	N-Wire Monitor Data register *This is the same as Debug register DR2 (DDATA0). See Section 20.4.2.4.*	64
TRCSYS	N-Trace System register	11
IBC IBA IBAM DBC DBA DBAM DBD DBDM	N-Wire and N-Trace Hardware Breakpoint registers *These are the same as Debug Registers DR4–DR15. See Section 20.4.2.6 and Section 20.4.2.7.*	Various

20.5.2.1 JTAG Instruction register

The JTAG Instruction register holds the opcodes for JTAG, N-Wire, and N-Trace operations. Instructions are entered into the test logic during an instruction register scan sequence in the TAP controller. Figure 20-15 shows the register format. Table 20-9 describes the JTAG, N-Wire, and N-Trace instructions.

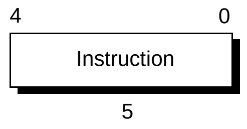


Figure 20-15 JTAG Instruction Register Format

Table 20-9 JTAG Instructions

Instruction	Opcode	Data Register	Function
EXTEST	00000	JTAG Boundary Scan register	Tests circuitry external to the chip
SAMPLE/ PRELOAD	00001	JTAG Boundary Scan register	Allows a snapshot of the normal operation of the chip to be taken and examined. Also allows data to be preloaded into the parallel outputs of the Boundary Scan register prior to another instruction such as EXTEST.
DM_SYSTEM	00010	N-Wire Debug Mode System register	Accesses the Debug Module System register
DM_CONTROL	00011	N-Wire Debug Mode Control register	Accesses the Debug Module Control register
PROCTYPE	00100	Processor Type register	Accesses the Processor Type register
NTRACE_SYS	00101	N-Trace System register	Accesses the Trace System register
MON_INST	01000	N-Wire Monitor Instruction register	Accesses the Monitor Instruction register
MON_DATA	01001	N-Wire Monitor Data register	Accesses the Monitor Data register

Table 20-9 JTAG Instructions (continued)

Instruction	Opcode	Data Register	Function
CACHE_TEST	01100	Cache Test register	Enables Cache Test mode
HIGHZ	01110	JTAG Bypass register	Tristates all outputs of the chip
BYPASS	11111	JTAG Bypass register	Connects JTDI to JTDO through the 1-bit Bypass register

20.5.2.2 JTAG Bypass register

The JTAG Bypass register is 1 bit wide. When the TAP controller is in the Shift-DR (Bypass) state, the data on the JTDI signal is shifted into the Bypass register, and the data on Bypass register output shifts to the JTDO output signal.
Figure 20-16 shows the register format.

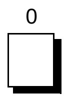


Figure 20-16 JTAG Bypass Register Format

The Bypass register is like a short-circuit. It allows bypassing of board-level devices in the boundary scan chain that do not require a specific test. Use of the register speeds up access to Boundary Scan registers in those ICs that remain active in the board-level test data path.

20.5.2.3 JTAG Boundary Scan register

The JTAG Boundary Scan register is a single bus comprising a 74-bit Shift register, each bit of which is connected to a processor signal. The Boundary Scan register retains states for all of the processor's input and output signals, except for some clock and phase-locked loop signals. The external signals can be configured to drive any arbitrary pattern, depending on the data scanned into the Boundary Scan register while in the JTAG Shift-DR state. Data driven into the signals from other devices can be examined while in the Capture-DR state.

Figure 20-17 shows the register format. Table 20-10 describes the register bits in their scan order.

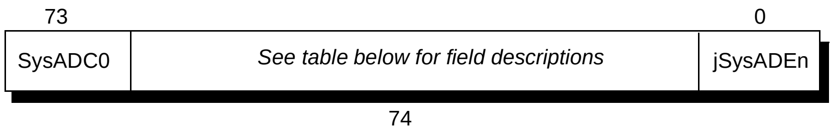


Figure 20-17 JTAG Boundary Scan Register Format

The least-significant bit, jSysADEn, is the JTAG output enable bit for all processor outputs. Output is enabled when this bit is set to 1. The remaining 73 bits correspond to the processor's 73 signal pads, as shown in Table 20-10. The scan starts by shifting the least-significant bit out of the Boundary Scan register, so the first scan-out bit is the jSysADEn signal.

Table 20-10 JTAG Boundary Scan Register Order

No.	Signal	No.	Signal	No.	Signal	No.	Signal
1	jSysADEn	20	NMI*	39	SysCmd7	58	RdRdy*
2	Tristate	21	SysAD8	40	SysCmd6	59	SysAD30
3	ColdReset*	22	SysAD9	41	SysCmd5	60	ValidOut*
4	BigEndian	23	SysAD10	42	SysCmd4	61	SysAD31
5	DivMode0	24	SysAD11	43	SysCmd3	62	PReq*
6	DivMode1	25	SysAD12	44	SysCmd2	63	SysAD0
7	ByPassPLL	26	SysAD13	45	ValidIn*	64	SysAD1
8	TrcEnd	27	SysAD14	46	OptionR43k*	65	SysAD2
9	TrcData3	28	SysAD15	47	Reset*	66	SysAD3
10	TrcData2	29	SysAD16	48	SysCmd1	67	SysAD4

Table 20-10 JTAG Boundary Scan Register Order (continued)

No.	Signal	No.	Signal	No.	Signal	No.	Signal
11	TrcClk	30	SysAD17	49	SysCmd0	68	SysAD5
12	TrcData1	31	SysAD18	50	ExtRqst*	69	SysAD6
13	TrcData0	32	SysAD19	51	SysAd25	70	SysAD7
14	BkTgIO*	33	SysAD20	52	Release*	71	SysADC3
15	Int4	34	SysAD21	53	SysAD26	72	SysADC2
16	Int3	35	SysAD22	54	SysAD27	73	SysADC1
17	Int2	36	SysAD23	55	SysAD28	74	SysADC0
18	Int1	37	SysAD24	56	SysAD29		
19	Int0	38	SysCmd8	57	WrRdy*		

20.5.2.4 Processor Type register

This register contains the CPU type and the debug module version. Figure 20-18 shows the register format. Table 20-11 describes the register fields.

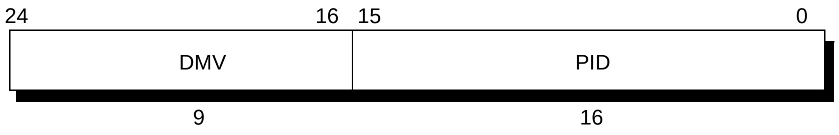


Figure 20-18 Processor Type Register Format

Table 20-11 Processor Type Register Format Register Fields

Bits	Field	Description
15:0	DMV	Debug module version. Set to 10H.
24:16	PID	Processor ID. Set to 5400H.

20.5.2.5 N-Wire Debug Module System register (DM_SYSTEM)

The DM_SYSTEM register contains the basic configuration fields for debug module initialization, N-Wire RMode*/BkTgIO* signal functions, and N-Trace functions. Figure 20-19 shows the register format. Table 20-12 describes the register fields. Certain fields of this register are copied into the DRCNTL debug register, described in Section 20.4.2.

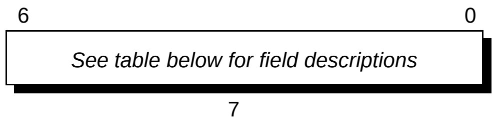


Figure 20-19 N-Wire Debug Module System (DM_SYSTEM) Register Format

Table 20-12 N-Wire Debug Module System (DM_SYSTEM) Register Fields

Bits	Name	Description
0	DINIT	Initialize the Debug Module (read/write) 1 → Resets (initializes) the debug module 0 → Releases reset of debug module (enable debug module) Defaulted to 1 at processor ColdReset* or JTAG in reset. When *DINIT* = 1, all the N-Wire register bits are at their reset value. The N-Wire function cannot be loaded unless *DINIT* is enabled (*DINIT* = 0).
1	BKTGIO	RMode* /BkTgIO* Signal Implementation (read only) 1 → Implemented 0 → Not implemented
2	BKTGIODIR	BkTgIO* Direction (read/write) 1 → Input 0 → Output Defaulted to 1 at the debug module initialization.
3	BKIOBEN	BkTgIO* Break Enable (read/write) 1 → Enable driving of trigger output on BkTgIO* at a processor break, or to break the processor at a BkTgIO* input 0 → Disable Defaulted to 0 at the debug module initialization.

Table 20-12 N-Wire Debug Module System (DM_SYSTEM) Register Fields (continued)

Bits	Name	Description
4	BKIOTEN	BkTgIO* Trigger Enable (read/write) 1 → Enable detected internal trigger events to the BkTgIO* signal when it is configured in the output direction 0 → Disable Defaulted to 0 at the debug module initialization.
5	NTRACE	N-Trace Implementation (read only) 1 → Implemented 0 → Not implemented
6	NTRACEN	N-Trace Port Enable (read/write) 1 → Enable 0 → Disable Defaulted to 0 at the debug module initialization.

20.5.2.6 N-Wire Debug Module Control register (DM_CONTROL)

The DM_CONTROL register contains enabling and status fields for debug reset, processor breaking, interrupt and exception handling, single-stepping, and execution of N-Wire Monitor instructions. Figure 20-20 shows the register format. Table 20-13 describes the register fields. Certain fields of this register are copied into the DRCNTL debug register, as described in Section 20.4.2.

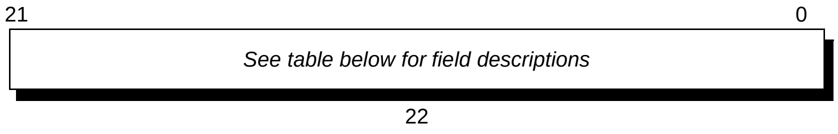


Figure 20-20 N-Wire Debug Module Control (DM_CONTROL) Register Format

Table 20-13 N-Wire Debug Module Control (DM_CONTROL) Register Fields

Bits	Name	Description
0	RESET	Debug Reset (read/write) 1 → Requests debug reset 0 → Releases debug reset Defaulted according to the level of the RMode* input at processor ColdReset*. When RMode* is active low, this register bit is active high.
1	BREAK	Break Request (read/write) Write 1 → Requests break Write 0 → No operation Read 1 → Command not completed (still requesting break) Read 0 → Break is completed This bit is cleared when the break is completed. Defaulted to 0 at the debug module initialization.
2	MRST	Mask Reset* (read/write) 1 → Ignores (masks) Reset* input while in Debug mode 0 → Accepts Reset* input while in Debug mode Defaulted to 1 at the debug module initialization. ColdReset* is not masked by this bit.
3	MNMI	Mask NMI* (read/write) 1 → Suppress the occurrence of NMI* 0 → Do not suppress the occurrence of NMI* Defaulted to 0 at the debug module initialization.
4	MINT	Mask Interrupts (read/write) 1 → Ignores user interrupt input 0 → Accepts user interrupt input This mask affects interrupts via the Int* signals or via an external write. Software interrupts are not masked. Defaulted to 0 at the debug module initialization.
5	STEP	Single-Step (read/write) This bit allows the user to execute one Normal mode instruction followed by a break. Single-step occurs after a DRET instruction. The processor returns to Normal mode, executes a single instruction, and breaks back into Debug mode. Enabling a single-step break while the processor is in Normal mode results in undefined behavior. 1 → Enable single-step break (single-step mode) 0 → Disable single-step break Defaulted to 0 at the debug module initialization.

Table 20-13 N-Wire Debug Module Control (DM_CONTROL) Register Fields (continued)

Bits	Name	Description
13:6	BRKCAUSE	Break Cause (read only) This field consists of multiple bits. One bit is assigned for each break cause and the corresponding bit is set when the break occurred. Multiple bits are set if the break occurred by multiple break causes. Break Cause is cleared by DRET or by processor reset. The bit assignments are defined as follows: Bit 6 → External break Bit 7 → Single-step Bit 8 → Software breakpoint Bit 9 → *Reserved* Bit 10 → *Reserved* Bit 11 → Instruction address breakpoint Bit 12 → Data access (address or data) breakpoint Bit 13 → *Reserved*
14	DBM	Debug or Normal Mode (read only) 1 → Debug mode is active 0 → Normal mode is active
17:15	CPUSTAT	CPU Status (read only) The processor status is encoded as follows: 0 0 0 → Reset (highest) 0 0 1 → *Reserved* 0 1 0 → *Reserved* 0 1 1 → *Reserved* 1 0 0 → *Reserved* 1 0 1 → *Reserved* 1 1 0 → *Reserved* 1 1 1 → Normal mode (lowest)
18	ACTFLG_CLK	Active Flag for Processor Clock (read/write) This bit indicates clock activity. The debug tool can use this bit to detect whether a clock is supplied into the processor from the target system board. Write 1 → No operation Write 0 → Clear active flag Read 1 → Clock is active Read 0 → Clock has not been active since flag was cleared This flag is set when there's any activity on the clock. This flag is cleared when the debug tool writes 0 into this bit.

Table 20-13 N-Wire Debug Module Control (DM_CONTROL) Register Fields (continued)

Bits	Name	Description
19	ACTFLG_BUS	Active Flag for Bus (read/write) This bit indicates bus activity. Write 1 → No operation Write 0 → Clear active flag Read 1 → Bus cycle is active Read 0 → Bus cycle has not occurred since flag was cleared This flag is set when there is any activity on the bus. It is cleared when the debug tool writes 0 into this bit.
20	MON_INSTEXEC	Monitor Instruction Execution (read/write) Setting this bit causes the processor to fetch and execute the instruction in the MON_INST register. Write 1 → Fetches and executes MON_INST instruction Write 0 → No operation Read 1 → A monitor instruction is executing Read 0 → No monitor instruction is executing
21	DM_EXCEPT	Debug Mode Exception (read/write) Read 1 → Instruction executed in Debug mode has caused an exception Read 0 → No exception in Debug mode since flag was cleared Write 1 → No operation Write 0 → Clear exception flag If any instruction other than a Load or Store causes an exception, the results and processor state are undefined.

20.5.2.7 N-Wire Monitor Instruction register (MON_INST)

All JTAG accesses to system resources, such as the processor's Normal mode and Debug mode registers, cache, external memory, and I/O are accessed via a monitor mechanism. The MON_INST and MON_DATA registers are used to insert instructions and data, respectively, into the processor.

When the debug module is active (*DINIT* bit cleared to 0 in the DM_SYSTEM register) and a debug break occurs, processor instructions can be loaded and executed. The MON_INST instruction causes a processor instruction to be scanned into the write-only MON_INST register through the JTAG port. The *MON_INSTEXEC* bit in the DM_CONTROL register can then be set to cause the processor to execute the instruction.

When executing Monitor instructions, the processor PC does not give meaning to the instruction. Therefore, all processor instructions and events that redirect the PC are not defined and produce unpredictable behavior. The DRET instruction is the only instruction that can be used for redirecting the PC.

Figure 20-21 shows the register format. A monitor instruction can only be executed while the processor is in Debug mode and the debug module is not reset. If the *MON_INSTEXEC* bit is written to while in Normal mode, the results are undefined. Attempts to modify the MON_INST or MON_DATA registers while executing a Monitor instruction will result in undefined behavior.

An example of loading processor instructions and data with the Monitor instruction is given in Section 20.5.3.

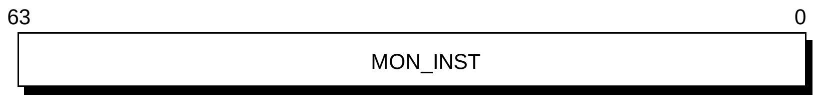


Figure 20-21 N-Wire Monitor Instruction (MON_INST) Register Format

20.5.2.8 N-Wire Monitor Data register (MON_DATA)

The MON_DATA register is identical to the DDATA0 debug register (DR2), described in Section 20.4.2.4. The MON_DATA instruction and register are used in conjunction with the MON_INST instruction and register to insert data and instructions into the processor. The MON_DATA instruction causes data to be scanned into the MON_DATA (DR2) register through the JTAG port. The MON_INST instruction is then used to scan the MFDR instruction into the MON_INST register. The *MON_INSTEXEC* bit in the DM_CONTROL register can then be set to cause the instruction currently loaded in the MON_INST register (i.e., the MFDR instruction) to move this data into a general-purpose register.

Figure 20-22 shows the register format. An example of loading processor instructions and data with the Monitor instruction is given in Section 20.5.3.

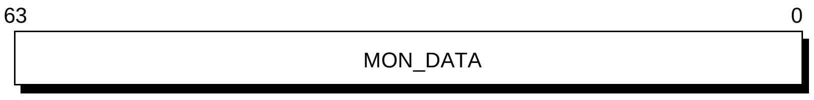


Figure 20-22 N-Wire Monitor Data (MON_DATA) Register Format

20.5.2.9 N-Trace System register (TRCSYS)

The TRCSYS register is used to control N-Trace reset and to give read-only information that indicates the processor's N-Trace implementation parameters. Figure 20-23 shows the register format. Table 20-14 describes the register fields.

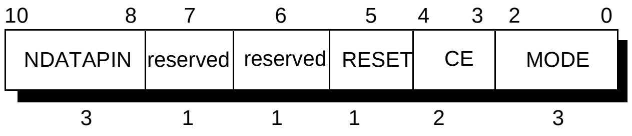


Figure 20-23 N-Trace System (TRCSYS) Register Format

Table 20-14 N-Trace System (TRCSYS) Register Fields

Bits	Name	Description
2:0	MODE	Trace Mode (read only) The read value is 2H. Bits 2:1 are 01, indicating Target PC (TPC) packet tracing in the N-Trace Level 1 mode (TPC packets at exceptions and indirect jumps). Bit 0 is 0, indicating a non-real-time trace.
4:3	CLKDIV	Trace Clock (TrcClk) Divisor (read only) These bits are set by the *DivMode* (1:0) pins. The trace port runs at the system interface clock frequency.
5	RESET	Reset N-Trace (read/write) 1 → N-Trace is in reset. The NOP packet is output to the TrcData (3:0) port. The bit is initialized to 1 at ColdReset*. 0 → N-Trace is active. Trace information is output to the TrcData (3:0) port. If N-Trace is reset, no trace packets are generated (NOP packets are on the internal N-Trace port). When reset is released, the value of the current PC is output and trace information proceeds.
7:6	*Reserved*	The read value is 0H.
10:8	NDATAPIN	N-Trace Data Pins (read only) The read value is 4H, indicating 4 pins in the TrcData (3:0) port.

20.5.2.10 N-Wire and N-Trace Hardware Breakpoint registers

Debug registers DR4–DR15 serve as the hardware breakpoint registers for both internal-access and external-access Debug mode. The registers are described in Section 20.4.2.

20.5.3 **N-Wire Monitor Data Download Example**

The following example describes the steps for downloading data into external memory using the N-Wire Monitor instruction and data resources. To do this, use the following sequence:

1. Break into Debug mode with the debug module enabled (*DINIT* bit cleared).
2. Scan the download data into the MON_DATA register via JTAG.
3. Scan the MFDR instruction into the MON_INST register via JTAG.
4. Set the *MON_INSTEXEC* bit in the DM_CONTROL register via JTAG. This causes the processor to execute the instruction in the MON_INST register, thus moving the data from the MON_DATA debug register (same as debug register DR2) into a general-purpose register in preparation for a store operation.
5. Check for completion by checking the *MON_INSTEXEC* bit via JTAG.
6. Scan a Store instruction into the MON_INST register via JTAG.
7. Set the *MON_INSTEXEC* bit via JTAG. This causes the processor to execute the Store instruction in the MON_INST register, thus storing the data from the general-purpose register into memory.
8. Check for completion by checking the *MON_INSTEXEC* bit via JTAG.
9. Repeat steps 2–8 for each doubleword of data to be stored in memory.
10. Scan a DRET instruction into the MON_INST register via JTAG.
11. Set the *MON_INSTEXEC* bit via JTAG to execute the DRET.
12. Check for completion by checking the *MON_INSTEXEC* bit via JTAG. When the DRET is complete, the processor has returned to Normal mode.

20.5.4 N-Trace Packets

The processor can trace its internal instruction execution by using the N-Trace protocol. It uses the TrcData (3:0), TrcEnd, and TrcClk signals on the JTAG port (Section 20.5.1) to send N-Trace packets to an external debug tool. The processor supports the N-Trace packets shown in Table 20-15. All packets maintain a 4-bit code definition and output information that is a multiple of four bits.

Table 20-15 N-Trace Packet Types

Mnemonic	Code	Description
EXP	0,1,1,0,<exp_id>	Exception
LSEQ	0,1,1,1	Long Sequential Execution
NOP	0,0,0,0	No Operation
NSEQ	1,0,0,0,<seq #>	Non-Sequential Operation
TPC	0,1,0,0,<program_counter>	Target PC

The processor generates useful trace packets only in Normal mode (User, Supervisor, or Kernel mode). It does not generate trace packets (other than NOP) in Debug mode; instead, NOP packets are continuously output on the N-Trace interface. When the processor is at the instruction boundary before entering Debug mode, all packets that have been generated are output. The processor also finishes all pending system interface operations before entering Debug mode. When the processor leaves Debug mode, it generates a Target PC (TPC) trace packet to indicate the instruction address where the normal execution resumes.

The processor supports only one N-Trace mode (N-Trace Level 1 mode, with TPC packets at exceptions and indirect jumps). The mode is non-real time, which implies that the CPU pipeline stalls if the trace buffer fills. N-Trace is either on or off and does not have additional control options.

20.5.4.1 Exception (EXP)

- Mnemonic EXP <exp_id>
- Code 0,1,1,0,<exp_id>

The EXP packet is output when an exception occurs within the processor. The <exp_id> field contains the Exception vector address taken. Bit 3 is equal to the *BEV* bit of the Status register. Bits 2:0 are an ID indicating the exception type.

ID	Exception
0 0 0	NMI
0 0 1	Debug Break
0 1 0	*Reserved*
0 1 1	*Reserved*
1 0 0	TLB Refill
1 0 1	XTLB Refill
1 1 0	Cache Error
1 1 1	Others

20.5.4.2 Long Sequential Execution (LSEQ)

- Mnemonic LSEQ
- Code 0,1,1,1

The LSEQ packet indicates that 256 instructions have been executed sequentially. This is the limit of the sequential instruction counter.

20.5.4.3 No Operation (NOP)

- Mnemonic NOP
- Code 0,0,0,0

The NOP packet is output if there are no other packets while trace is enabled. It is also output if trace is disabled.

20.5.4.4 Non-Sequential Operation (NSEQ)

- Mnemonic NSEQ <seq #>
- Code 1,0,0,0, <seq #>

The NSEQ packet indicates the current value of the 8-bit Trace instruction counter (IC). It is output when a branch, jump, or exception occurs. The <seq #> field is the count of instructions since the last NSEQ or LSEQ occurred; the count starts at 0.

20.5.4.5 Target PC (TPC)

- Mnemonic TPC <program_counter>
- Code 0,1,0,0,<program_counter>

The TPC packet contains a 40-bit value representing the virtual address of:

- The target address of a Jump Register instruction after an NSEQ packet
- The new PC after an ERET instructi
- The starting trace location when trace reset is released
- The new PC when the processor leaves Debug mode whil N-Trace is enabled

20.5.4.6 N-Trace instruction summary

Table 20-16 summarizes the Trace instructions and the trace behavior that they create. The instructions are grouped according to the classifications that are defined as part of the N-Trace architecture. The instruction counter (IC) is a pointer that indicates the count of instructions after an NSEQ or TPC packet. This count starts at 0. The IC reported by a trace action is the IC of the instruction that caused the action.

Table 20-16 N-Trace Instruction Summary

Instruction Set	Instruction or Group	Trace Action[1]
CPU Instruction Set	J, JAL (Action occurs in the delay slot)	NSEQ <IC>; IC = 0;
CPU Instruction Set	JR, JALR (Action occurs in the delay slot)	NSEQ <IC>; IC = 0; TPC;

CPU Instruction Set	PC-Relative Conditional Branches (Action occurs in the delay slot for Branch Taken case)	If (Branch Taken)
		NSEQ <IC>; IC = 0;
		Else (Not Taken)
		IC <- IC + 1; If (IC = 256)
		LSEQ; IC = 0;

CPU Instruction Set	Exceptions and SYSCALL, BREAK instructions	NSEQ<IC>; EXP<cause>; IC = 0;

CPU Instruction Set	Conditional Traps	If (Trap Taken)
		NSEQ<IC>; EXP<cause>; IC = 0;
		Else (Not Taken)
		IC <- IC +; If (IC = 256) LSEQ; IC = 0;

CPU Instruction Set	All other instructions	IC <- IC + 1; If (IC = 256)
		LSEQ; IC = 0;

CP0 Instruction Set	ERET	NSEQ <IC>; IC = 0; TPC;

CP0 Instruction Set	All other instructions	IC <- IC + 1;
		If (IC = 256) LSEQ; IC = 0;

FPU Instruction Set	Conditional Branches
	(Action occurs in the delay slot for Branch Taken case)
		If (Branch Taken)
		NSEQ <IC>; IC = 0;
		Else (Not Taken)
		IC <- IC + 1; If (IC = 256)
		LSEQ; IC = 0;

FPU Instruction Set	All other instructions	IC <- IC + 1; If (IC = 256)
		LSEQ; IC = 0;

Debug Instructions	Debug Break or Break instruction	NSEQ <IC>; IC = 0;
Debug Instructions	DRET	IC = 0; TPC;
Debug Instructions	All other instructions	No action

Note:
1. IC = instruction counter, a pointer indicating the number of instructions after an NSEQ or TPC packet.

Table 20-17 shows an example of a Break instruction with an exception handler instruction indicated as the target.

Table 20-17 Trace Example #1

IC	Instruction	Trace Packet(s)
N	Break	NSEQ<N>; EXP<cause>
0	Target	
1	Target + 1	

For taken branches and Jump instructions, the PC is not redirected until the delay slot is executed. The NSEQ and IC reported is for the delay slot of the Branch instruction. Table 20-18 shows an example of a Branch instruction with a Target instruction target:

Table 20-18 Trace Example #2

IC	Instruction	Trace Packet(s)
N−1	Branch	
N	Delay Slot	NSEQ<N>
0	Target	
1	Target + 1	

Table 20-19 shows an example of a Jump Register instruction with a Target instruction and target address.

Table 20-19 Trace Example #3

IC	Instruction	Trace Packet(s)
N−1	Jump Register	
N	Delay Slot	NSEQ<N>; TPC<TargetAddress>
0	Target	
1	Target + 1	

For Branch instructions not taken, the delay slot is always part of the instruction flow. The NSEQ and IC reported include delay slots of all Branch instructions. delay slot is included even for branch-likely cases where the architecture does not include it. For branch-likely cases, the delay slot is treated as an NOP.

Subblock Data Retrieval Order

A

Data block elements (bytes, halfwords, words, or doublewords) can be retrieved from storage in either sequential or subblock order. This appendix describes these retrieval methods, with an emphasis on subblock retrieval order.

Note: The VR5432 processor requires external memory systems to retrieve data in subblock order.

Sequential retrieval fetches data block elements in serial, or sequential, order. Figure A-1 shows an example of sequential retrieval, in which word 0 is taken first and word 3 is taken last.

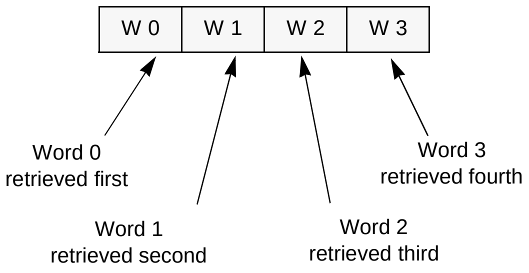


Figure A-1 Retrieving a Data Block in Sequential Order

Subblock retrieval allows the system to define the retrieval order. Figure A-2 shows retrieval of a four-word block; the critical word at the target address is retrieved first (W2), followed by the remaining words. (The smallest data element of a block transfer is a doubleword.)

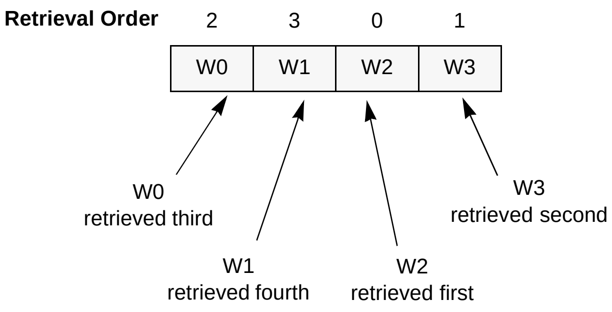


Figure A-2 Subblock Order Data Retrieval

The subblock ordering logic generates an address for each word as it is retrieved by executing a bitwise exclusive-OR (XOR) of the starting block address with the output of a binary counter that increments with each word, starting at word zero (00_2). Using this scheme, Table A-1 through Table A-3 list subblock word retrieval for a four-word block, based on three different starting-block addresses: 10_2, 11_2, and 01_2. The subblock order is generated by an XOR of the subblock address (10_2, 11_2, and 01_2) with the binary count of the word (00_2 through 11_2).

Table A-1 Subblock Sequence: Address 10_2

Cycle	**Starting Block Address**	**Binary Count**	**Word Retrieved**
1	10	00	10
2	10	01	11
3	10	10	00
4	10	11	01

Table A-2 Subblock Sequence: Address 11_2

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	11	00	11
2	11	01	10
3	11	10	01
4	11	11	00

Table A-3 Subblock Sequence: Address 01_2

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	01	00	01
2	01	01	00
3	01	10	11
4	01	11	10

Comparing the VR4300, VR5000, and VR5432 Processors

B

Table B-1 compares the VR4300, VR5000, and VR5432 processor features.

Table B-1 VR4300, VR5000, and VR5432 Feature Compariso

Feature	**VR4300**	**VR5000**	**VR5432**
Cache Algorithms	Cached (write-back) Uncached	Cached (write-back) Cached (write-through) Uncached	Cached (write-back) Cached (write-through) Uncached Accelerated uncached
Circuit Design Technique	Dynamic	Static	Static
Coprocessor 0 Hazards	Yes	Yes	No
Data Cache Array Size	8 KB	32 KB	32 KB
Data Cache Associativity	Direct mapped	2-way set associative	2-way set associative
Data Cache Line Locking	No	No	Yes (Lock bit/cache line)
Data Cache Line Size	16 bytes	32 bytes	32 bytes
Data Cache Parity Support	No	Yes	No

Table B-1 VR4300, VR5000, and VR5432 Feature Comparison (continued)

Feature	VR4300	VR5000	VR5432
Hardware Debug Features	JTAG Boundary Scan	No	JTAG Boundary Scan N-Wire debug support Hardware breakpoints Instruction jamming
Instruction Cache Array Size	16 KB	32 KB	32 KB
Instruction Cache Associativity	Direct-mapped	2-way set associative	2-way set associative
Instruction Cache Line Locking	No	No	Yes (Lock bit/cache line)
Instruction Cache Line Size	32 bytes	32 bytes	32 bytes
Instruction Cache Parity Support	No	Yes	No
Instruction Fetch Branch Prediction	No	No	4096 entries 2-bit saturating counter
Instruction Set Architecture	MIPS III	MIPS IV	MIPS IV + Rotate + DSP (Integer MAC, etc.) + Media
Load/Store Architecture	Blocking	Blocking	Nonblocking hits under misses Up to 4 outstanding D-cache misses
Performance Counters (Software/Code Tuning)	No	No	Two 32-bit counters Selectable any 2 of 16 different events
Physical Address Size	32 bits	36 bits	36 bits internal; 32 bits external
Power-On Configuration Modes	Dedicated pins	Scan-in boot ROM	Dedicated pins
Secondary Cache Support	No	Yes	No

Table B-1 VR4300, VR5000, and VR5432 Feature Comparison (continued)

Feature	**VR4300**	**VR5000**	**VR5432**
Superscalar (Execution Units)	Scalar (Single Issue)	Limited 2-way (1 Integer + 1 Floating Point)	Symmetrical 2-way (2 Integer + 2 Floating Point + 1 Load/Store + 1 MAC + 1 Media)
System Interface Clock Divisors	1, 1.5, 2, 3	2, 3, 4, 5, 6, 7, 8	2, 2.5, 3, 4
System Interface Parity Support	No	Yes	No
System Interface Protocol	R4000-like (Removed Unused Encodings)	R4000 + Additional Write Modes	R5000 + Split Transactions, or R4000-like (in VR4300 Emulation mode)
System Interface Width	32 bits address/data multiplexed	64 bits + parity address/data multiplexed	32 bits + parity address/data multiplexed
TLB Data Micro-TLB	No	2 entries (4 KB fixed page size)	4 entries (4 KB–16 MB variable page sizes)
TLB Instruction Micro-TLB	2 entries (4 KB fixed page size)	2 entries (4 KB fixed page size)	4 entries (4 KB – 16 MB variable page sizes)
TLB Joint (2nd Level)	32 double entries (4 KB–16 MB variable page sizes)	48 double entries (4 KB–16 MB variable page sizes)	48 double entries (4 KB–16 MB variable page sizes)
Virtual Address Size (largest segment)	40 bits	40 bits	40 bits

PLL Analog Power Filtering

C

For noisy module environments, a phase-locked loop (PLL) filter circuit, as shown in Figure C-1, is recommended. In addition, the configuration shown in Figure C-2 is required for PLLCap input.

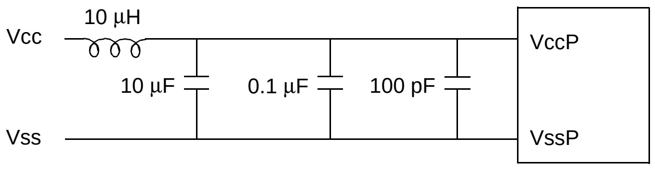


Figure C-1 PLL Filter Circuit

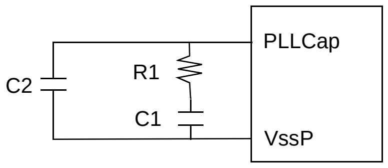


Figure C-2 PLLCap Circuit

R1 = 1 KOhm, C1 = 400 pF, and C2 = 40 pF. All values shown are nominal. Minimum and maximum values are TBD. All components should be placed as closely as possible to the indicated pins.

Instruction Hazards

D

This chapter identifies R5432 instruction hazards that occur with certain instruction and event combinations (such as pipeline delays, cache misses, interrupts, and exceptions). These hazards can cause unpredictable system behavior and malfunctions.

Most hazards result from instructions modifying and reading state in different pipeline stages. Such hazards are defined between instruction pairs, not on a single instruction in isolation. Other hazards are associated with instruction restartability in the presence of exceptions.

For the following code hazards, the behavior is undefined and unpredictable.

- Any instruction that would modify the PageMask, EntryHi, EntryLo0, EntryLo1, or Random CP0 registers should not be followed by TLBWR instruction. There should be at least two integer instruction between the register modification and the TLBWR instruction
- Any instruction that would modify the PageMask, EntryHi, EntryLo0, EntryLo1, or Index CP0 registers should not be followed by TLBWI instruction. There should be at least two integer instructions between the register modification and the TLBWI instruction.
- Any instruction that would modify the Index CP0 register or the contents of the JTLB should not be followed by a TLBR instruction. There should be at least two integer instructions between the register modification and the TLBR instruction
- Any instruction that would modify the PageMask or EntryHi CP0 registers or the contents of the JTLB should not be followed by a TLBP instruction. There should be at least two integer instructions between the register modification and the TLBP instruction.
- Any instruction that would modify the EPC, ErrorEPC, or Status CP0 registers should not be followed by an ERET instruction. There should be at least two integer instructions between the register modification and the ERET instruction.
- A Branch or Jump instruction is not allowed in the delay slot o another Branch/Jump instruction. This sequence is illegal in th MIPS architecture.
- The two instructions preceding a DIV, DIVU, DDIV, DDIVU, MULT, MULTU, DMULT, or DMULTU instruction should not read the HI o LO registers. There should be at least two integer instruction between the register read and the register modification

Index

NEC

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

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

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

NEC Electronics Inc. (U.S.)
Santa Clara
Tel: 800-366-9782
Fax: 800-729-9288

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

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

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

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

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

NEC Electronics (France) S.A.
Spain Office
Madrid, Spain
Tel: 01-504-2787
Fax: 01-504-2860

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

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

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

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore 1130
Tel: 253-8311
Fax: 250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-719-2377
Fax: 02-719-5951

NEC do Brasil S.A.
Sao Paulo-SP, Brasil
Tel: 011-889-1680
Fax: 011-889-1689

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Corporation. NEC Corporation assumes no responsibility for any errors which may appear in this document. NEC Corporation does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from use of a device described herein or any other liability arising from use of such device. No license, either express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Corporation or others. While NEC Corporation has been making continuous effort to enhance the reliability of its semiconductor devices, the possibility of defects cannot be eliminated entirely. To minimize risks of damage or injury to persons or property arising from a defect in an NEC semiconductor device, customers must incorporate sufficient safety measures in its design, such as redundancy, fire-containment, and anti-failure features. NEC devices are classified into the following three quality grades: "Standard," "Special," and "Specific." The Specific quality grade applies only to devices developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a device depend on its quality grade, as indicated below. Customers may check the quality grade of each device before using it in a particular application. Standard: Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots. Special: Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment (not specifically designed for life support). Specific: Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems or medical equipment for life support, etc. The quality grade of NEC devices is "Standard" unless otherwise specified in NEC's data sheets or data books. If customers intend to use NEC devices for applications other than those specified for Standard quality grade, they should contact an NEC sales representative in advance.

In North America: No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics Inc. (NECEL). The information in this document is subject to change without notice. All devices sold by NECEL are covered by the provisions appearing in NECEL Terms and Conditions of Sales only, including the limitation of liability, warranty, and patent provisions. NECEL makes no warranty, express, statutory, implied or by description, regarding information set forth herein or regarding the freedom of the described devices from patent infringement. NECEL assumes no responsibility for any errors that may appear in this document. NECEL makes no commitments to update or to keep current information contained in this document. The devices listed in this document are not suitable for use in applications such as, but not limited to, aircraft control systems, aerospace equipment, submarine cables, nuclear reactor control systems and life support systems. "Standard" quality grade devices are recommended for computers, office equipment, communication equipment, test and measurement equipment, machine tools, industrial robots, audio and visual equipment, and other consumer products. For automotive and transportation equipment, traffic control systems, anti-disaster and anti-crime systems, it is recommended that the customer contact the responsible NECEL salesperson to determine the reliability requirements for any such application and any cost adder. NECEL does not recommend or approve use of any of its products in life support devices or systems or in any application where failure could result in injury or death. If customers wish to use NECEL devices in applications not intended by NECEL, customers must contact the responsible NECEL salespeople to determine NECEL's willingness to support a given application.

©2000 NEC Electronics Inc./Printed in U.S.A.

U15397EU5V0UMJ1