

A comprehensive reference for the original
R2000/R2010 and the new, higher-performance
R3000/R3010 Reduced Instruction Set Computers

mips RISC ARCHITECTURE



GERRY KANE

MIPS RISC Architecture

*This book describes the MIPS RISC Architecture.
The R2000 Processor and R2010 Floating Point Accelerator
are the original implementation of the architecture and
the R2000/R2010 numbering notation is used throughout the text.*

*Higher-performance versions of the two machines,
the R3000 and R3010, are the newest implementation of
the MIPS RISC architecture. All references in this book
to the R2000/R2010 Processors apply fully
to the new R3000/R3010 Processors.*

Gerry Kane

Prentice Hall, Englewood Cliffs, NJ 07632

Library of Congress Catalog Card Number 88-060290

© 1987, 1988, 1989. MIPS Computer Systems, Inc. All Rights Reserved. No part of this document may be reproduced in any form or by any means without prior express written consent of MIPS Computer Systems, Inc., and the publisher.

The information contained in this book is for reference only and is subject to change without notice.

RISComputer, RISC/os, and RISCompilers are Trademarks of MIPS Computer Systems.
UNIX is a Trademark of AT&T.

Ethernet is a Trademark of XEROX.

VAX is a Trademark of Digital Equipment Corp.

IBM is a Trademark of International Business Machines.

68000 is a Trademark of Motorola Semiconductor Corp.

Book Design: Suzanne Hayes-Kane.

The publisher offers discounts on this book when ordered
in bulk quantities. For more information, write:

Special Sales/College Marketing

Prentice-Hall, Inc.

College Technical and Reference Division

Englewood Cliffs, NJ 07632

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7

ISBN 0-13-584293-X

ISBN 0-13-584749-4 {PH}

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

About This Book

This book is a comprehensive reference manual for the MIPS RISC architecture. It describes the functional characteristics and capabilities of the R2000/R3000 Processors and the R2010/R3010 Floating Point Accelerators (FPA). The new, higher-performance R3000/R3010 machines are architecturally identical to the original R2000/R2010 versions and the descriptions in this book apply to both series of machines — only the electrical and performance characteristics differ. The processors are available from the following manufacturers:

- Integrated Device Technology, Inc.
3226 Scott Boulevard
P.O. Box 58015
Santa Clara, CA 95054-3029
(408) 727-6116
Telex 887766
FAX (408) 988-3029
- LSI Logic Corporation
1551 McCarthy Blvd.
MS D-105
Milpitas, CA 95035
(408) 433-8000
Telex 171.641
FAX (408) 433-3816
Attn: Standard Products Marketing
- MIPS Computer Systems, Inc.
930 Arques Avenue
Sunnyvale, CA 94086-3650
(408) 720-1700
Telex 510 601 5346
FAX (408) 720-9809
- Performance Semiconductor Corporation
610 E. Weddell Drive
Sunnyvale, CA 94089
(408) 734-9000
FAX (408) 734-0258
Attn: Microprocessor Marketing
- NEC Electronics, Inc.
401 Ellis Street
P.O. Box 7241
Mountain View, CA
(415) 960-6000
Attn: Microprocessor Marketing
- Siemens Components, Inc.
2191 Laurelwood Road
Santa Clara, CA 95054
(408) 980-4500
Attn: Integrated Circuits Division

Organization

This book is organized into two major sections: Chapters 1 through 5 describe the characteristics of the R2000 Processor and Chapters 6, 7, and 8 describe the R2010 FPA. The contents of each chapter are summarized in the list that follows:

- **Chapter 1, RISC Architecture: An Overview**, describes the general characteristics and concepts of reduced instruction set computers.
- **Chapter 2, R2000 Processor Overview**, describes the general characteristics and capabilities of the processor. It also provides a programming model which describes how data is represented in the R2000 registers and in memory and also provides a summary of the R2000 CPU registers.
- **Chapter 3, Instruction Set Summary**, provides a summary description of the R2000 instruction set.
- **Chapter 4, Memory Management System**, describes the virtual memory system supported by the R2000's System Control Coprocessor.
- **Chapter 5, Exception Processing**, describes the events that cause R2000 exceptions and the sequences that occur during processing of the exceptions.
- **Chapter 6, R2010 FPA Overview**, describes the general characteristics and capabilities of the FPA. This chapter also provides a summary of the R2010 FPA registers and describes how data is represented in the R2010 registers.
- **Chapter 7, FPA Instruction Summary & Instruction Pipeline**, provides a summary description of the R2010 instruction set and a discussion of instruction overlapping.
- **Chapter 8, Floating Point Exceptions**, describes how the R2010 FPA supports the IEEE standard floating point exceptions.
- **Appendix A** provides a detailed description of the format and operation of each R2000 instruction.
- **Appendix B** provides a detailed description of the format and operation of each R2010 FPA instruction.
- **Appendix C** describes machine language programming tips that can simplify implementation of commonly required tasks.
- **Appendix D** describes assembly language programming techniques and provides guidelines for writing programs for use with the MIPS assembler.
- **Appendix E** describes how the R2010 FPA supports the IEEE floating point standard and provides programming tips that can simplify implementation of standard operations not implemented by the FPA.

Contents

1

RISC Architecture: An Overview

Scope of this Overview	1-1
What Is RISC?	1-1
Defining Performance	1-2
Time/Instruction	1-3
Cycles/Instruction (C)	1-4
Instruction Pipelines	1-4
Load/Store Architecture	1-7
Delayed Load Instructions	1-7
Delayed Branch Instructions	1-9
Time/Cycle (T)	1-11
Instruction Decode Time	1-11
Instruction Operation Time	1-11
Instruction Access Time	1-12
Overall Architectural Simplicity	1-13
Instructions/Task (I)	1-14
Optimizing Compilers	1-14
Operating System Support	1-17
The RISC Design Process	1-18
Hidden Benefits of RISC Design	1-19
Shorter Design Cycle	1-19
Smaller Chip Size	1-19
User (Programmer) Benefits	1-19
Most Aggressive Technologies	1-19

2

R2000 Processor Overview

R2000 Processor Features	2-2
R2000 CPU Registers	2-2
Instruction Set Overview	2-3
R2000 Processor Programming Model	2-6
Data Formats and Addressing	2-6
R2000 CPU General Registers	2-8
R2000 System Control Coprocessor	2-9
System Control Coprocessor Registers	2-9
Memory Management System	2-10
The TLB (Translation Lookaside Buffer)	2-10
R2000 Operating Modes	2-10
R2000 Pipeline Architecture	2-12
Memory System Hierarchy	2-13

3

R2000 Instruction Set Summary

Instruction Formats	3-1
Instruction Notational Conventions	3-2
Load and Store Instructions	3-2
Computational Instructions	3-5
ALU Immediate Instructions	3-5
3-Operand Register Type Instructions	3-6
Shift Instructions	3-7
Multiply/Divide Instructions	3-8
Jump and Branch Instructions	3-9
Special Instructions	3-10
Coprocessor Instructions	3-11
System Control Coprocessor Instructions	3-12
R2000 Instruction Pipeline	3-13
The Delayed Instruction Slot	3-14

Delayed Loads	3-15
Delayed Jumps and Branches	3-16

4

Memory Management System

Memory System Architecture	4-1
Privilege States	4-2
User-Mode Virtual Addressing	4-3
Kernel-Mode Virtual Addressing	4-3
Virtual Memory and the TLB	4-4
TLB Entries	4-5
EntryHi & EntryLo Registers	4-6
Virtual Address Translation	4-7
The Index Register	4-9
The Random Register	4-10
TLB Instructions	4-10

5

Exception Processing

The Exception Handling Registers	5-3
The Cause Register	5-4
The EPC Register	5-5
The Status Register	5-6
Status Register Mode Bits and Exception Processing	5-9
BadVAddr Register	5-10
Context Register	5-10
Processor Revision ID Register	5-11
Exception Description Details	5-12
Exception Vector Locations	5-12
Address Error Exception	5-13

Breakpoint Exception	5-14
Bus Error Exception	5-15
Coprocessor Unusable Exception	5-16
Interrupt Exception	5-17
Overflow Exception	5-18
Reserved Instruction Exception	5-19
Reset Exception	5-20
System Call Exception	5-21
TLB Miss Exceptions	5-22
TLB Miss Exception	5-23
TLB Mod Exception	5-25
UTLB Miss Exception	5-26

6

R2010 FPA Overview

R2010 FPA Features	6-1
R2010 FPA Programming Model	6-2
Floating-Point General Registers (FGR)	6-3
Floating-Point Registers (FPR)	6-4
Floating-Point Control Registers (FGR)	6-4
Control/Status Register	6-5
Implementation and Revision Register	6-8
Floating-Point Formats	6-8
Number Definitions	6-10
Normalized Numbers	6-10
Denormalized Numbers	6-11
Infinity	6-11
Zero	6-11
Coprocessor Operation	6-11
Load, Store, and Move Operations	6-11
Floating-Point Operations	6-12
Exceptions	6-12
Instruction Set Overview	6-13
R2010 Pipeline Architecture	6-14

7

FPA Instruction Set Summary & Instruction Pipeline

Instruction Set Summary	7-1
Load, Store, and Move Instructions	7-1
Floating Point Computational Instructions	7-3
Floating Point Relational Operations	7-5
Branch on FPA Condition Instructions	7-7
The Instruction Pipeline	7-8
Instruction Execution Times	7-10
Overlapping FPA Instructions	7-12

8

Floating Point Exceptions

Exception Trap Processing	8-2
Inexact Exception (I)	8-4
Invalid Operation Exception (V)	8-5
Division-by-Zero Exception (Z)	8-6
Overflow Exception (O)	8-7
Underflow Exception (U)	8-8
Unimplemented Operation Exception (E)	8-9
Saving and Restoring State	8-10

A

R2000 Instruction Set Details

B

R2010 FPA Instruction Set Details

C

Machine Language Programming Tips

32-bit Addresses or Constants	C-2
Indexed Addressing	C-3
Subroutine Return using JR Instruction	C-4
Jumping to 32-bit Addresses	C-4
Branching On Arithmetic Comparisons	C-4
Filling the Branch Delay Slot	C-5
Testing for Carry	C-6
Testing for Overflow	C-8
Multi-precision Math	C-10
Double-Word Shifts	C-12

D

Assembly Language Programming

Register Use and Linkage	D-1
General Registers	D-1
Special Registers	D-3
Floating Point Registers	D-3
Assembly Instruction Summaries	D-4
Addressing	D-9
Address Formats	D-10
Address Descriptions	D-11
Pseudo Op-Codes	D-12
Linkage Conventions	D-18
Examples	D-24
Memory Allocation	D-27
Basic Machine Definition	D-28
Load and Store Instructions	D-28
Computational Instructions	D-28
Branch Instructions	D-29
Coprocessor Instructions	D-29
Special Instructions	D-29

E

IEEE Floating Point Standard Compatibility Issues

Interpretation of the Standard	E-1
Underflow	E-1
Exceptions	E-1
Inexact	E-2
NaNs	E-2
Software Assistance for IEEE Compatibility	E-2
IEEE Exception Trapping	E-3
IEEE Format Compatibility	E-3
Implementing IEEE Standard	
Operations in Software	E-4
Remainder	E-4
Round to Integer	E-4
Convert between Binary and Decimal	E-5
Copy Sign	E-5
Scale Binary	E-5
Log Binary	E-5
Next After	E-5
Finite	E-5
Is NaN	E-6
Arithmetic Inequality	E-6
Class	E-6

Index

Overflow Exception

Cause. This exception occurs when an **ADD ADDI, SUB, or SUBI** instruction results in two's complement overflow. This exception is not maskable.

Handling. The R2000 branches to the General Exception vector (0x80000080) for this exception. The R2000 sets the *OV* code of the *Cause* register.

The *EPC* register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the *EPC* register points at the **branch** instruction that preceded the exception-causing instruction and sets the *BD* bit of the *Cause* register.

The R2000 saves the *KUp*, *IEp*, *KUc*, and *IEc* bits of the *Status* register in the *KUo*, *IEo*, *KUp*, and *IEp* bits, respectively, and clears the *KUc* and *IEc* bits.

Servicing. A kernel should hand the executing process a floating point exception or integer overflow error when this exception occurs. Such an error is usually fatal.

A

R2000 Processor Instruction Set Details

This appendix provides a detailed description of the operation of each R2000 instruction. The instructions are listed in alphabetical order.

Refer to **Appendix B** for a detailed description of the R2010 FPA instructions.

The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and manner of handling exceptions is omitted from the instruction descriptions in this chapter. Refer to **Chapter 5** for detailed descriptions of exceptions and handling.

A chart on the last page of this appendix lists the bit encoding for the constant fields of each instruction.

Instruction Classes

R2000 instructions are divided into the following classes:

- **Load/Store** instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is *base register + 16-bit immediate offset*.
- **Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.
- **Jump and Branch** instructions change the control flow of a program. Jumps are always to absolute 26-bit word addresses (J-type format), or 32-bit register addresses (R-type). Branches have 16-bit offsets relative to the program counter (I-type). Jump and Link instructions save a return address in Register 31.
- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor Loads and Stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPA instructions). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.
- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, syscall, and breakpoint. They are always R-type.

Instruction Formats

Every R2000 instruction consists of a single word (32 bits) aligned on a word boundary and there are only three instruction formats as shown in Figure A.1.

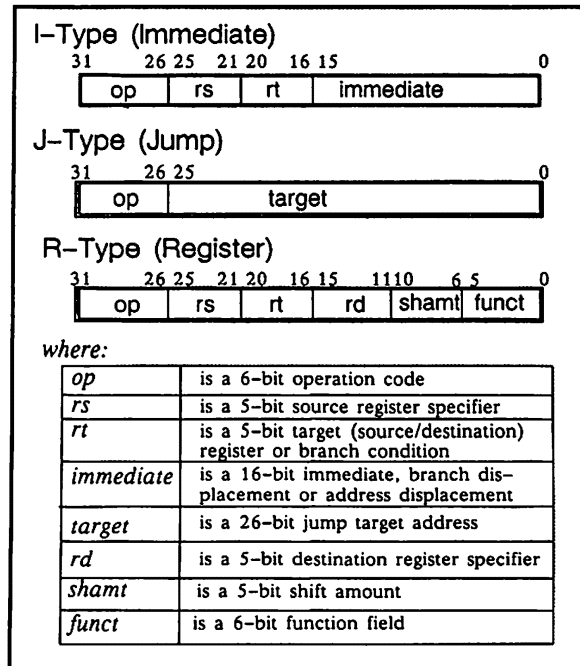


Figure A.1 R2000 Instruction Formats

Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *rs*, *rt*, *immediate*, etc.) are shown in lower-case names.

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

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. Special symbols used in the notation are described in Table A.2.

Symbol	Meaning
$\leftarrow$	Assignment
$\parallel$	Bit string concatenation
x^y	Replication of bit value x into a y -bit string. Note that x is always a single-bit value.
$x_{y..z}$	Selection of bits y through z of bit string x . Little-endian bit notation is always used. If y is less than z , this expression is an empty (zero length) bit string.
$+$	Two's complement addition
$-$	Two's complement subtraction
$*$	Two's complement multiplication
<i>div</i>	Two's complement integer division
<i>mod</i>	Two's complement modulo
$<$	Two's complement less than comparison
<i>and</i>	Bitwise logic AND
<i>or</i>	Bitwise logic OR
<i>xor</i>	Bitwise logic XOR
<i>nor</i>	Bitwise logic NOR
GPR[x]	General Register x . The contents of GPR[0] is always zero. Attempts to alter the contents of GPR[0] have no effect.
CPR[z,x]	Coprocessor unit z , general register x
CCR[z,x]	Coprocessor unit z , control register x
$T+i$	Indicates the time steps (CPU cycles) between operations. Thus, operations identified as occurring at $T+1$ are performed during the cycle following the one where the instruction was initiated. This type of operation occurs with loads, stores, jumps, branches and coprocessor instructions.
vAddress	Virtual address
pAddress	Physical address

Table A.2 R2000 Instruction Operation Notations

Instruction Notation Examples

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

Example #1:

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

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

Example #2:

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

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

Load and Store Instructions

All load operations have a latency of one instruction. That is, the instruction immediately following a load cannot use the contents of the register which will be loaded with the data being fetched from storage. An exception is that the target register for the load word left and load word right instructions may be specified as the same register used as the destination of a load instruction that immediately precedes it.

In the load/store operation descriptions, the functions listed in Table A.2a are used to summarize the handling of virtual addresses and physical memory.

Function	Description
Addr Translation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the entry for the page containing the virtual address is not present in the TLB (Translation Lookaside Buffer).
Load Memory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
Store Memory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data into the word containing the specified physical address. The low-order two bits of the address and the access type field indicate which of the four bytes within the data word should be stored.

Table A.2a Load/Store Common Functions

The access type field indicates the size of the data item to be loaded or stored as shown in Table A.2b. Regardless of access type or byte-numbering order (endian-ness), the address specifies the byte which has the smallest byte address of the bytes in the addressed field. For a big-endian machine, this is the leftmost byte and contains the sign for a two's complement number; for a little-endian machine, this is the rightmost byte and contains the lowest precision byte.

Access Type		Meaning
Mnemonic	Value	
WORD	3	word (32 bits)
TRIPLE-BYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

Table A.2b Access Type Specifications for Loads/Stores

The bytes within the addressed word which are used can be determined directly from the access type and the two low-order bits of the address, as shown in Table A.2c. Note that certain combinations of access type and low-order address bits can never occur (WORD/01/10/11, TRIPLE-BYTE/10/11, HALFWORD/01/11).

Access Type	Low-Order Address Bits	Bytes Accessed	
		Big-Endian	Little-Endian
1 0	1 0	31 _____ 0	31 _____ 0
1 1 (word)	0 0	0 1 2 3	3 2 1 0
1 0 (triple-byte)	0 0	0 1 2	2 1 0
	0 1	1 2 3	3 2 1
0 1 (halfword)	0 0	0 1	1 0
	1 0	2 3	3 2
0 0 (byte)	0 0	0	0
	0 1	1	1
	1 0	2	2
	1 1	3	3

Table A.2c Byte Specifications for Loads/Stores

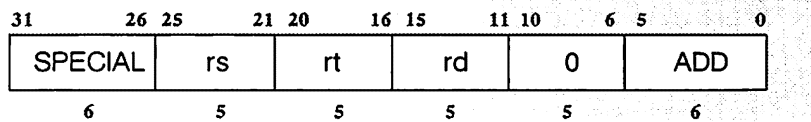
Jump and Branch Instructions

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

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the R2000 sets the EPC register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are re-executed.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 may not be used as a source register.

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

ADD**ADD****Format:**

ADD rd,rs,rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed into general register *rd*.

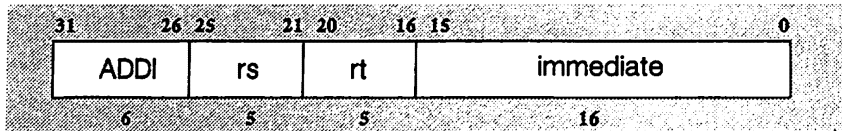
An overflow exception occurs if the two highest order carry-out bits differ (two's complement overflow).

Operation:

T:	$\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] + \text{GPR}[\text{rt}]$
----	--

Exceptions:

Overflow exception

ADDI**Add Immediate****Format:**ADDI *rt*,*rs*,*immediate***Description:**

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form a 32-bit result. The result is placed into general register *rt*.

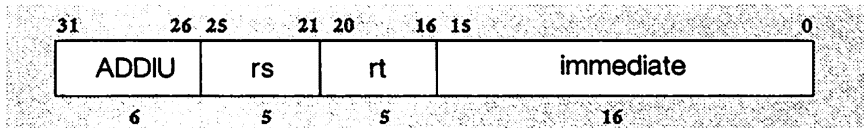
An overflow exception occurs if the two highest order carry-out bits differ (two's complement overflow).

Operation:

$$T: \quad \text{GPR}[rt] \leftarrow \text{GPR}[rs] + (\text{immediate}_{16})^{18} \parallel \text{immediate}_{15..0}$$

Exceptions:

Overflow exception

Add Immediate Unsigned**ADDIU****Format:**

ADDIU rt,rs,immediate

Description:

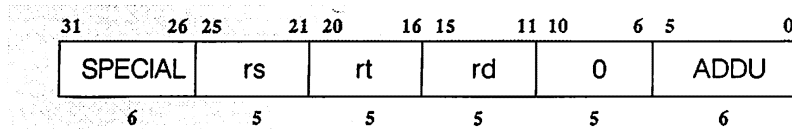
The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form a 32-bit result. The result is placed into general register *rt*. No overflow exception occurs under any circumstances.

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

Operation:

$$T: \quad \text{GPR}[rt] \leftarrow \text{GPR}[rs] + (\text{immediate}_{15})^{16} \parallel \text{immediate}_{15..0}$$
Exceptions:

None.

ADDU**ADD Unsigned****Format:**

ADDU rd,rs,rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed into general register *rd*.

No overflow exception occurs under any circumstances.

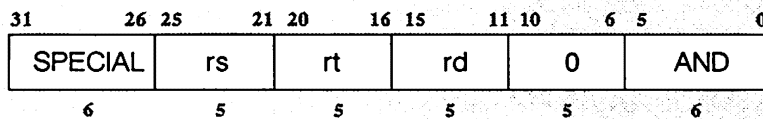
Note that the only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

T: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] + \text{GPR}[\text{rt}]$
--

Exceptions:

None.

And**AND****Format:**

AND rd,rs,rt

Description:

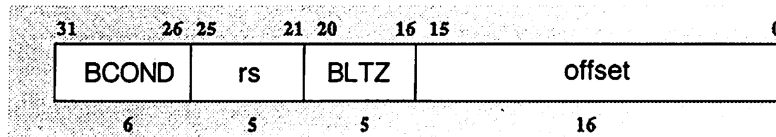
The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

T: $GPR[rd] \leftarrow GPR[rs] \text{ and } GPR[rt]$

Exceptions:

None.

BLTZ**Branch On Less Than Zero****Format:**

BLTZ rs,offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended to 32 bits. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

Operation:

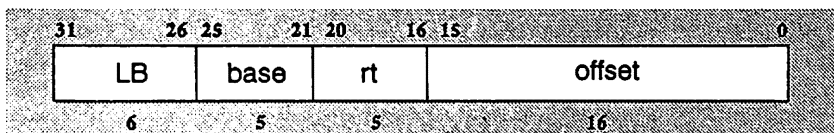
```

T:  target ← (offset15)14 || offset || 02
    condition ← (GPR[rs]31 = 1)
T + 1: if condition then
        PC ← PC + target
    endif

```

Exceptions:

None.

Load Byte**LB****Format:**LB *rt*,*offset*(*base*)**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

Operation:

```

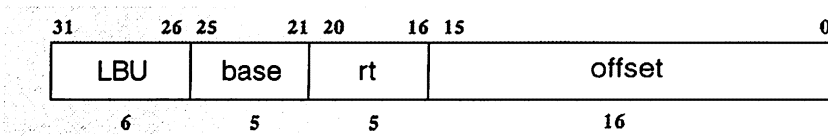
T:   vAddress ← (offset15)16 || offset15..0 + GPR[base]
      (pAddress, nonCacheable) ← AddrTranslation (vAddress)
      mem ← LoadMemory (nonCacheable, BYTE, pAddress)
      byte ← vAddress1..0

T + 1: if BigEndian then
      GPR[rt] ← (mem31-8*byte)24 || mem31-8*byte...24-8*byte
    else
      GPR[rt] ← (mem7+8*byte)24 || mem7+8*byte...8*byte
    endif

```

Exceptions:

UTLB miss fault
 TLB miss fault
 Bus error exception
 Address error exception

LBU**Load Byte Unsigned****Format:**LBU *rt*,*offset*(*base*)**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

Operation:

```

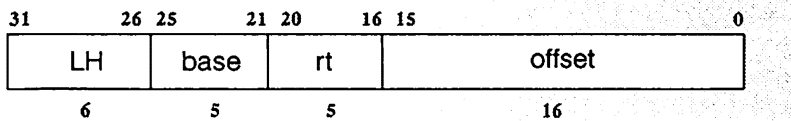
T:   vAddress ← (offset15)16 || offset15..0 + GPR[base]
      (pAddress, nonCacheable) ← AddrTranslation (vAddress)
      mem ← LoadMemory (nonCacheable, BYTE, pAddress)
      byte ← vAddress1..0

T + 1: if BigEndian then
      GPR[rt] ← 024 || mem31-8*byte..24-8*byte
    else
      GPR[rt] ← 024 || mem7+8*byte..8*byte
    endif

```

Exceptions:

UTLB miss fault
 TLB miss fault
 Bus error exception
 Address error exception

Load Halfword**LH****Format:**LH *rt*,*offset*(*base*)**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

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

Operation:

```

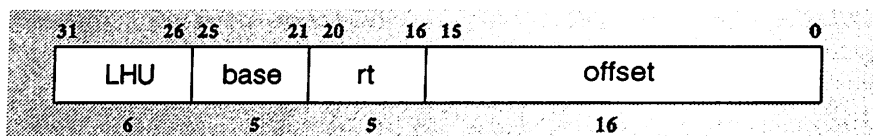
T:   vAddress ← (offset15)16 || offset15..0 + GPR[base]
      (pAddress, nonCacheable) ← AddrTranslation (vAddress)
      mem ← LoadMemory (nonCacheable, HALFWORD, pAddress)
      byte ← vAddress1..0

T + 1: if BigEndian then
      GPR[rt] ← (mem31-8*byte)16 || mem31-8*byte..16-8*byte
    else
      GPR[rt] ← (mem15+8*byte)16 || mem15+8*byte..8*byte
    endif

```

Exceptions:

UTLB miss fault
 TLB miss fault
 Bus error exception
 Address error exception

LHU**Load Halfword Unsigned****Format:**LHU *rt*,*offset*(*base*)**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

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

Operation:

```

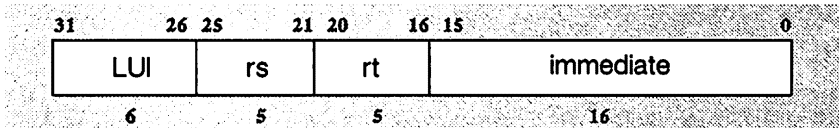
T:   vAddress ← (offset15)16 || offset15..0 + GPR[base]
      (pAddress, nonCacheable) ← AddrTranslation (vAddress)
      mem ← LoadMemory (nonCacheable, HALFWORD, pAddress)
      byte ← vAddress1..0

T + 1: if BigEndian then
      GPR[rt] ← 016 || mem31-8*byte..16-8*byte
    else
      GPR[rt] ← 016 || mem15+8*byte..8*byte
    endif

```

Exceptions:

UTLB miss fault
 TLB miss fault
 Bus error exception
 Address error exception

Load Upper Immediate**LUI****Format:**

LUI rt,immediate

Description:

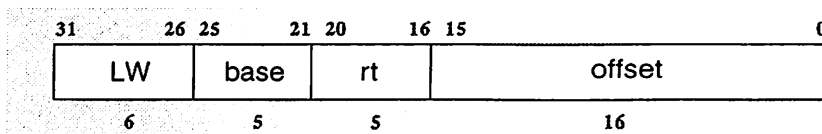
The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeroes. The result is placed into general register *rt*.

Operation:

$T: \quad \text{GPR}[rt] \leftarrow \text{immediate} \ll 0^{16}$
--

Exceptions:

None.

LW**Load Word****Format:**LW *rt*,*offset*(*base*)**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*.

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

Operation:

```

T:   vAddress ← (offset15)16 || offset15..0 + GPR[base]
      (pAddress, nonCacheable) ← AddrTranslation (vAddress)
      mem ← LoadMemory (nonCacheable, WORD, pAddress)
      byte ← vAddress1..0

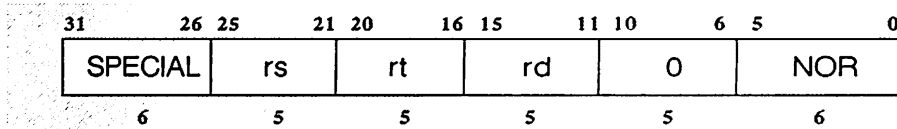
T+1: GPR[rt] ← mem;
```

Exceptions:

UTLB miss fault
 TLB miss fault
 Bus error exception
 Address error exception

NOR

Nor

**Format:**

NOR rd,rs,rt

Description:

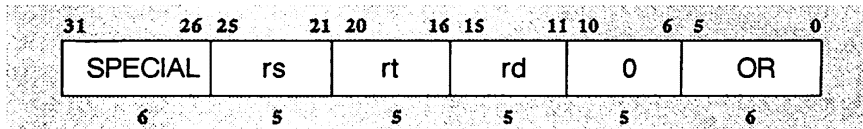
The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into general register *rd*.

Operation:

T:	$GPR[rd] \leftarrow GPR[rs] \text{ nor } GPR[rt]$
----	---

Exceptions:

None.

Or**OR****Format:**

OR rd,rs,rt

Description:

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

Operation:

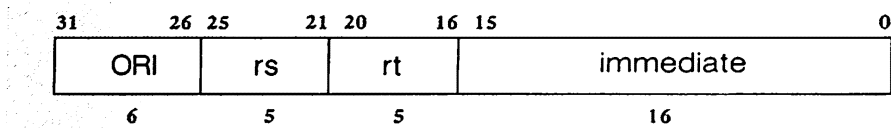
T:	$\text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ or } \text{GPR}[rt]$
----	---

Exceptions:

None.

ORI

Or Immediate

**Format:**

ORI rt,rs,immediate

Description:

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

Operation:

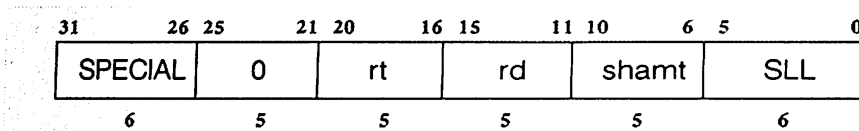
T: $\text{GPR}[rt] \leftarrow \text{GPR}[rs]_{31:16} \parallel (\text{immediate or } \text{GPR}[rs]_{15:0})$
--

Exceptions:

None.

SLL

Shift Left Logical

**Format:**

SLL rd,rt,shamt

Description:

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

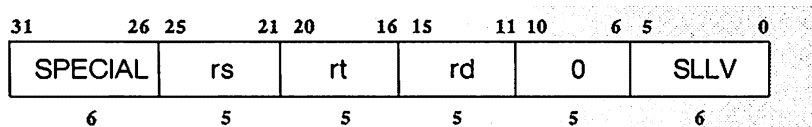
Operation:

T: $\text{GPR}[rd] \leftarrow \text{GPR}[rt]_{31-shamt:0} \parallel 0_{shamt}$

Exceptions:

None.

Shift Left Logical Variable

SLLV**Format:**

SLLV rd,rt,rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order 5 bits of the contents of general register *rs*, inserting zeroes into the low order bits. The 32-bit result is placed in register *rd*.

Operation:

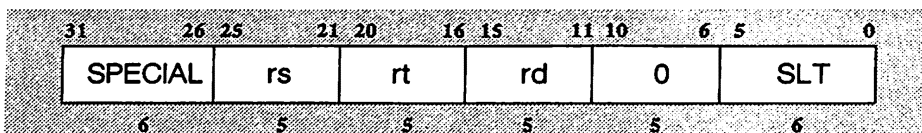
$$T: \quad \text{GPR}[rd] \leftarrow \text{GPR}[rt]_{(31-\text{GPR}[rs]_4..0)} \parallel 0^{\text{GPR}[rs]_4..0}$$

Exceptions:

None.

SLT

Set On Less Than

**Format:**

SLT rd,rs,rt

Description:

The contents of general register *rt* are compared with the contents of general register *rs*. Considering both quantities as signed 32-bit integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rd*.

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

Operation:

```

T:      if GPR[rs] < GPR[rt] then
          GPR[rd] ← 031 || 1
        else
          GPR[rd] ← 032
        endif

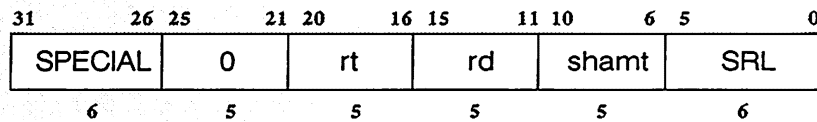
```

Exceptions:

None.

SRL

Shift Right Logical

**Format:**

SRL rd,rt,shamt

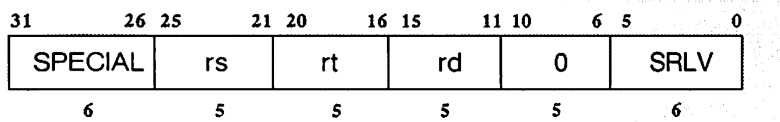
Description:

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

Operation:

$$T: \quad \text{GPR}[rd] \leftarrow 0^{shamt} \parallel \text{GPR}[rt]_{31..shamt}$$
Exceptions:

None.

Shift Right Logical Variable**SRLV****Format:**

SRLV rd,rt,rs

Description:

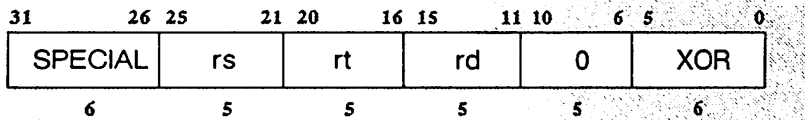
The contents of general register *rt* are shifted right by the number of bits specified by the low-order 5 bits of the contents of general register *rs*, inserting zeroes into the high order bits. The 32-bit result is placed in register *rd*.

Operation:

$$T: \quad \text{GPR}[rd] \leftarrow 0 \text{ GPR}[rs]_{4..0} \parallel \text{GPR}[rt]_{31..(\text{GPR}[rs]_{4..0})}$$

Exceptions:

None.

Exclusive Or**XOR****Format:**

XOR rd,rs,rt

Description:

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

Operation:

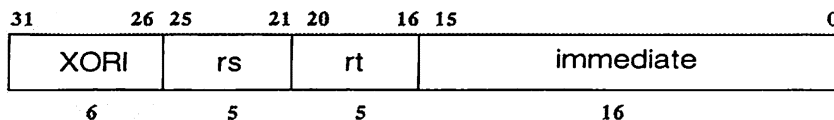
T:	$\text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ xor } \text{GPR}[rt]$
----	--

Exceptions:

None.

XORI

Exclusive Or Immediate

**Format:**

XORI rt,rs,immediate

Description:

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

Operation:

$T: \quad \text{GPR}[rt] \leftarrow \text{GPR}[rs]_{31..16} \parallel (\text{immediate} \text{ xor } \text{GPR}[rs]_{15..0})$

Exceptions:

None.



C

Machine Language Programming Tips

The RISC architecture implemented by the R2000 Processor provides an efficient, uniform, and streamlined instruction set to obtain maximum efficiency for the most commonly performed operations. As a result, some operations that would require single, multi-cycle instructions in more traditional architectures require multiple, single-cycle R2000 instructions.

The R2000 architecture also provides no condition code register containing status bits such as *Carry* and *Overflow*. Instead, the conditions generated by the Set instructions (*slt*, *sltu*) are loaded directly into a general-purpose register. This approach greatly simplifies handling of the instruction pipeline and eases many compiler tasks but does require that programs explicitly check for conditions such as overflow and carry.

This appendix describes techniques that can be used to implement the following operations:

- Handling 32-bit addresses or constants
- Implementing indexed addressing
- Using the Jump Register (*jr*) instruction for subroutine returns
- Jumping to 32-bit addresses
- Branching on arithmetic comparisons
- Filling the branch delay slot
- Testing for overflow
- Testing for carry
- Performing multi-precision math

Note that the MIPS assembly language supports many of these functions (such as loading 32-bit addresses and branching on arithmetic comparisons) that are not

directly implemented by the R2000 machine language. It does this using techniques similar to those described in this appendix. Refer to Appendix D for an overview of the MIPS assembly language.

In many of the following examples, a temporary register is used to hold intermediate results. In the following description, `$at` is used to represent that temporary register, and it is assumed that the register is reserved for just this purpose and therefore doesn't conflict with anything else.

32-bit Addresses or Constants

The R2000 does not provide specific "load address" or "load immediate" instructions. (Note: the MIPS assembly language does provide load address (`la`) and load immediate (`li`) instructions that are implemented using multiple machine language instructions. To load an address with relocatable code, you *must* use the assembler's `la` instruction.) Use the following two-instruction sequence to load any 32-bit pattern into a register:

```
lui  $destination, <upper 16 bits>
ori  $destination, <lower 16 bits>
```

There are three special cases which require only one instruction to obtain the desired 32-bit pattern:

- **Constants with the upper 16 bits set to zero.** Use the `ori` instruction, which zero-extends the immediate field:

```
ori  $destination, $0, <lower 16 bits>
```

- **Constants with the upper 17 bits set to one.** Use the low-order 16 bits of the constant in the `addi` instruction, which sign-extends the immediate field:

```
addi $destination, $0, <lower 16 bits>
```

- **Constants with the lower 16 bits set to zero.** Use the high-order 16 bits of the constant in the `lui` instruction, which shifts its immediate field left by 16 bits, bringing in zeroes on the right:

```
lui  $destination, <upper 16 bits>
```

Testing for Overflow

The R2000 does not provide a status bit to indicate whether an arithmetic operation resulted in an overflow. The signed addition and subtraction instructions (**ADD** and **SUB**) trap if an overflow occurs and thus implement overflow detection at no cost. However, if it is necessary to detect signed overflow without using traps or to detect overflow for unsigned operations, the techniques described in this section can be used.

Figure C.1 provides examples of code that checks for overflow for various arithmetic operations. These examples are based on the following simple rules for signed overflow:

- During addition, overflow occurs if the signs of the addends are the same and the sign of the sum is different.
- During subtraction, overflow occurs if the signs of the operands are different and the sign of the resultant difference is not the same as the sign of the minuend.

Signed Addition

```
/* compute t0 = t1 + t2, branch to L on signed overflow */
addu  t0, t1, t2    /* compute sum */
xor   t3, t1, t2    /* if operands have different signs */
bltz  t3, 1f        /* then overflow not possible */
/* t1 and t2 have same sign */
xor   t3, t0, t1    /* if sum does not also have the same sign */
bltz  t3, L         /* then addition overflowed */
/* nop */
1:
```

Signed Subtraction

```
/* compute t0 = t1 - t2, branch to L on signed overflow */
subu  t0, t1, t2    /* compute difference */
xor   t3, t1, t2    /* if operands have same signs */
bgez  t3, 1f        /* then overflow not possible */
/* t1 and t2 have different signs */
xor   t3, t0, t1    /* if difference does not also have */
/* the same sign as the minuend */
bltz  t3, L         /* then subtraction overflowed */
/* nop */
1:
```

Unsigned Addition

```
/* compute t0 = t1 + t2 , branch to L on unsigned overflow */
addu  t0, t1, t2
not   t3, t1
sltu  t3, t3, t2
bne   t3, 0, L
```

Unsigned Subtraction

```
/* compute t0 = t1 - t2, branch to L on unsigned overflow */
subu  t0, t1, t2
sltu  t3, t1, t2
bne   t3, 0, L
```

Signed Multiplication

```
/* compute t0 = t1 * t2, branch to L on signed overflow */
mult  t1, t2
mflo  t0
mfhi  t3
sra   t4, t0, 31
bne   t4, t3, L
```

Unsigned Multiplication

```
/* compute t0 = t1 * t2, branch to L on unsigned overflow */
multu t1, t2
mflo  t0
mfhi  t3
bne   t3, 0, L    /* if HI  $\neq$  0 ,then multiplication overflowed */
```

Figure C.1 Calculating Overflow for Arithmetic Operations

Assembly Language Programming

This appendix provides an overview of the assembly language supported by the MIPS compiler system. The assembler converts assembly language statements into machine code. In most assembly languages, each instruction corresponds to a single machine instruction; however, some MIPS assembly language instructions can generate several R2000 machine instructions. This approach provides a more regular assembler that generates optimized code for certain short sequences and also results in assembly programs that can run without modification on future machines, which might have extended machine instructions. See *Basic Machine Definition* at the end of this appendix for more information about assembler instructions that generate multiple machine instructions.

Register Use and Linkage

This section describes the naming and usage conventions that the assembler applies to the R2000 and R2010 registers.

General Registers

The R2000 Processor has thirty-two 32-bit integer registers. Table D.1 summarizes the assembler's usage conventions and restrictions for these registers. The assembler reserves all register names, and you must use lowercase for the names. All register names start with a dollar sign (\$).

The general registers have the names `$0..$31`. By including the file `regdef.h` (use `#include <regdef.h>`) in your program, you can use software names for some general registers. The operating system and the assembler use the general registers `$1`, `$26`, `$27`, `$28`, and `$29` for specific purposes. (NOTE: Attempts to use these general registers in other ways can produce unexpected results.) If a program uses the names `$1`, `$26`, `$27`, `$28`, `$29` rather than the names `$at`, `$kt0`, `$kt1`, `$gp`, `$sp` respectively, the assembler issues warning messages.

register name	software name (from regdef.h)	use and linkage
\$0		always has the value 0
\$at or \$1		reserved for the assembler
\$2..\$3	v0-v1	used for expression evaluations and to hold integer function results Also used to pass the static link when calling nested procedures
\$4..\$7	a0-a3	used to pass the first 4 words of integer type actual arguments; their values are not preserved across procedure calls
\$8..\$15	t0-t7	temporary registers; used for expression evaluations; their values are not preserved across procedure calls.
\$16..\$23	s0-s7	saved registers; their values must be preserved across procedure calls.
\$24..\$25	t8-t9	temporary registers; used for expression evaluations; their values are not pre-served across procedure calls.
\$26..\$27 or \$kt0..\$kt1	k0-k1	reserved for the operating system kernel
\$28 or \$gp	gp	contains the global pointer
\$29 or \$sp	sp	contains the stack pointer
\$30	s8	a saved register (like s0-s7)
\$31	ra	contains the return address; used for expression evaluation.

Table D.1 General (Integer) Registers

General register \$0 always contains the value 0. All other general registers are equivalent, except that general register \$31 also serves as the implicit link register for jump and link instructions.

mips RISC ARCHITECTURE

GERRY KANE

The evolution of computer architectures was dominated, until recently, by increasingly complex processors. Reduced Instruction Set Computers (RISC) are now challenging that domination. The MIPS R2000 RISC architecture described in this book delivers dramatic cost/performance advantages over computers based on traditional architectures.

The book is a comprehensive reference manual for the MIPS RISC architecture. It describes the functional characteristics and capabilities of the R2000/R3000 Processors and the R2010/R3010 Floating Point Accelerators (FPA). The new, higher-performance R3000/R3010 machines are architecturally identical to the original R2000/R2010 versions and the descriptions in this book apply to both series of machines — only the electrical and performance characteristics differ. The processors are available from the following manufacturers:



Integrated Device Technology, Inc.

LSI LOGIC

mips



- Integrated Device Technology, Inc.
3226 Scott Boulevard
P.O. Box 58015
Santa Clara, CA 95054-3029
(408) 727-6116
Telex 887766
FAX (408) 988-3029
- LSI Logic Corporation
1551 McCarthy Blvd.
MS D-105
Milpitas, CA 95035
(408) 433-8000
Telex 171.641
FAX (408) 433-3816
Attn: Standard Products Marketing
- MIPS Computer Systems, Inc.
930 Arques Avenue
Sunnyvale, CA 94086-3650
(408) 720-1700
Telex 510 601 5346
FAX (408) 720-9809
- Performance Semiconductor Corporation
610 E. Weddell Drive
Sunnyvale, CA 94089
(408) 734-9000
FAX (408) 734-0258
Attn: Microprocessor Marketing

ISBN 0-13-584749-4

Prentice Hall
Englewood Cliffs, N.J. 07632