Ridge 3200 Processor Reference Manual

(Preliminary)

December, 1986

PUBLICATION HISTORY

Manual Title: Ridge 3200 Processor Reference Manual

Preliminary Edition: 9091 (DEC 86)

NOTICE

No part of this document may be translated, reproduced, or copied in any form or by any means without the written permission of Ridge Computers.

The information contained in this document is subject to change without notice. Ridge Computers shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

© Copyright 1983, 1984, 1985, 1986 Ridge Computers.
All rights reserved.
Printed in the U.S.A.

PREFACE

This manual provides an overview of the Ridge 3200 processor.

Chapter 1 lists the characteristics and features of the 3200 processor.

Chapter 2 describes the logical operation of the individual boards that make up the 3200 processor.

Chapter 3 describes the Ridge I/O system.

Chapter 4 describes the Ridge instruction formats and data types.

Chapter 5 describes the hardware components used by the 3200 to manage processes.

Chapter 6 describes the operation of traps and interrupts.

Chapter 7 describes how the 3200 translates virtual addresses to physical addresses.

Chapter 8 lists the instructions executable in user mode.

Chapter 9 lists the instructions executable in kernel mode.

CONVENTIONS USED IN THIS MANUAL

Italics are used to reference items appearing illustrations.

The 16 general registers are referred to as Rx or Ry. A register pair is referenced as RPx and consists of Rx and Rx+1 mod 16. Rx holds the most significant bits and Rx+1 holds the least significant bits.

TABLE OF CONTENTS

Chapter 1: OVERVIEW

INTRODUCTION 1-1
PROCESSOR CHARACTERISTICS 1-1
KEY FEATURES 1-2

Chapter 2: 3200 PROCESSOR ARCHITECTURE

INTRODUCTION 2-1
INSTRUCTION FETCH UNIT 2-3
INSTRUCTION PIPELINE 2-3
BRANCH PREDICTION LOGIC 2-4
Conditional Branch Instructions 2-5
Branch Prediction Example 2-5
Unconditional Branch Instructions 2-7
REGISTER FILE / MULTIPLIER UNIT 2-7
REGISTER FILE LOGIC 2-7
MULTIPLIER LOGIC 2-7
EXECUTION UNIT 2-8
REGISTER BYPASS FUNCTION 2-9
CACHE / MEMORY CONTROLLER 2-10
CODE/DATA CACHE 2-10

Chapter 3: RIDGE I/O SYSTEM

INTRODUCTION 3-1
I/O OPERATIONS 3-2
I/O READ / WRITE 3-2
I/O Read Operation 3-3
I/O Write Operation 3-4
I/O INTERRUPTS 3-6
Interrupt Operation 3-7
DMA READ / WRITE 3-8
DMA Read Operation 3-9
DMA Write Operation 3-10
BUS CONTENTION 3-11
DUAL DAISY-CHAINED SIGNALS 3-11
EXAMPLE OF BUS CONTENTION 3-12

Chapter 4: INSTRUCTION FORMATS AND ADDRESSING

INTRODUCTION 4-1
GENERAL REGISTERS 4-2
INSTRUCTION FORMATS 4-2

REGISTER INSTRUCTIONS....4-2
Register Instruction Format....4-2
MEMORY REFERENCE INSTRUCTIONS....4-3
Memory Reference Instruction Formats....4-4
DATA REPRESENTATION....4-5
INTEGER REPRESENTATION....4-5
REAL NUMBER REPRESENTATION....4-5
Single Precision Real Numbers....4-5
Double Precision Real Numbers....4-6
REAL NUMBERS WITH SPECIAL MEANING....4-6
Special Numbers as Operands....4-7
Special Numbers as Results....4-7
REAL NUMBER ROUNDING RULES....4-9
DATA STORAGE IN REGISTERS....4-9

Chapter 5: PROCESS MANAGEMENT

INTRODUCTION....5-1
CODE AND DATA SEGMENTS....5-1
PROCESSOR MODES....5-1
KERNEL MODE....5-1
USER MODE....5-2
Privileged Process Bit....5-2
PROCESSOR CONTROL....5-3
SPECIAL REGISTERS....5-3
PROCESS CONTROL BLOCK....5-5

Chapter 6: TRAPS AND INTERRUPTS

INTRODUCTION....6-1
EVENT HANDLING....6-1
CPU CONTROL BLOCK....6-2
CCB DATA AREA....6-4
PAGE FAULT....6-4
INTERRUPTS....6-4
DOUBLE-BIT PARITY ERROR....6-5
Double-Bit Parity Error on Instruction Fetch....6-5
Double-Bit Parity Error on Instruction Execute....6-5
EXTERNAL INTERRUPTS....6-5
SWITCH 0....6-5
POWER FAIL WARNING....6-6
TIMER INTERRUPTS....6-6
RESET....6-6
TRAPS....6-7
KERNEL CALLS....6-7
DATA ALIGNMENT TRAP....6-7
ILLEGAL INSTRUCTION TRAP....6-7
KERNEL VIOLATION TRAP....6-8
CHECK TRAP....6-8
TRAP INSTRUCTION TRAP....6-8
ARITHMETIC TRAPS....6-9

Traps Word 6-9
Integer Overflow Trap 6-10
Integer Divide By Zero Trap 6-11
Before Trap 6-11
Real Divide by Zero Trap 6-12
Real Underflow Trap 6-12
Real Overflow Trap 6-13
Inexact Result Trap 6-14
RECOVERY FROM REAL OVERFLOW, UNDERFLOW, AND INEXACT RESULT TRAPS 6-14

Chapter 7: VIRTUAL MEMORY MANAGEMENT

INTRODUCTION 7-1
VIRTUAL ADDRESS 7-1
VIRTUAL TO REAL TRANSLATION HARDWARE 7-2
TRANSLATION TABLES 7-2
TRANSLATION SEQUENCE 7-3
VRT TABLE ORGANIZATION 7-4
VRT Entry Format 7-4
VRT Translation Process 7-5
TMT TABLE ORGANIZATION 7-7
TMT Entry Format 7-7
TMT Translation Process 7-7

Chapter 8: USER MODE INSTRUCTIONS

INTRODUCTION 8-1
SYNTAX CONVENTIONS 8-1
MEMORY REFERENCE INSTRUCTIONS 8-2
LOAD INSTRUCTIONS 8-2
STORE INSTRUCTIONS 8-3
LOAD ADDRESS INSTRUCTIONS 8-3
REGISTER INSTRUCTIONS 8-4
INTEGER ARITHMETIC INSTRUCTIONS 8-4
LOGICAL OPERATOR INSTRUCTIONS 8-5
INTEGER AND LOGICAL IMMEDIATE INSTRUCTIONS 8-6
EXTENDED PRECISION INTEGER INSTRUCTIONS 8-7
REAL INSTRUCTIONS 8-8
DOUBLE REAL INSTRUCTIONS 8-9
BIT-ORIENTED INSTRUCTIONS 8-10
TEST INSTRUCTION 8-10
COMPARE INSTRUCTIONS 8-11
SHIFT INSTRUCTIONS 8-12
SIGN EXTEND INSTRUCTIONS 8-13
PROGRAM CONTROL INSTRUCTIONS 8-14
BRANCH INSTRUCTIONS 8-14
LOOP CONTROL INSTRUCTION 8-15
SUBROUTINE CALL AND RETURN INSTRUCTIONS 8-16
Call Subroutine Instruction 8-16
Call Subroutine Register and Return Instructions 8-17

TRAP INSTRUCTIONS....8-18

Chapter 9: KERNEL MODE INSTRUCTIONS

INTRODUCTION....9-1
STATE SWITCHING INSTRUCTIONS....9-1
MAINTENANCE INSTRUCTIONS....9-3
VIRTUAL MEMORY SUPPORT INSTRUCTIONS....9-7
INPUT/OUTPUT INSTRUCTIONS....9-7

APPENDIX A: INSTRUCTION INDEX....A-1

APPENDIX B: INSTRUCTION EXECUTION TIMES....B-1

APPENDIX C: RIDGE OPCODE MAP....C-1

Chapter 1
OVERVIEW

INTRODUCTION

The Ridge 3200 is a 32-bit high performance RISC (Reduced Instruction Set Computer) processor implemented in MSI and LSI bipolar logic. The The 3200 has a simple, general purpose, microcoded architecture that incorporates paged virtual memory.

The main objective of RISC architecture is to simplify the functions of the machine, thereby reducing the amount of hardware necessary to implement the processor. The reduction in logic allows a faster cycle time and permits instructions to complete in one machine cycle.

This functional simplification is made possible by reducing the size and complexity of the processor's instruction set. With fewer and simpler instructions, the RISC-based computer is able to execute programs faster and with greater reliability.

PROCESSOR CHARACTERISTICS

The Ridge 3200 is characterized by the following:

Simple addressing modes. The Ridge 3200 uses only three modes which reduces the amount of logic needed to perform memory references.

Simple instruction formats. The Ridge 3200 uses three instruction formats that can be decoded with a minimum of logic.

Separated code and data. The Ridge 3200 uses separated code and data eliminating the need for logic that detects and resolves self-modifying code.

High-level language support. The instructions provided are designed to match the code generation capabilities of such languages as FORTRAN, C, and Pascal. The Ridge 3200 instruction set provides simple, quickly-executable instructions which are assembled into their optimum function sequences by the Ridge compilers.

Regularity. Data types and addressing modes are examples of regularity. For memory reference instructions there are four operand sizes and three addressing modes. Each of the addressing modes is available for all operands.

Linear address space. Code and data space are each linear with a byte-addressable area that is four-gigabytes long.

General registers. All registers are available for use as data, indexing, and addressing.

KEY FEATURES

- Reduced Instruction Set Computer (RISC) Architecture
- 83-nanosecond Processor Cycle Time
- 83-nanosecond Cache Cycle Time and 250-nanosecond access time to Main Memory
- One-clock Cycle Minimum Instruction Time
- 4096-byte paged virtual memory
- Four-gigabytes Linear Address Space
- Separated Code and Data
- Branch Prediction Logic
- Single and Double Real Floating Point Instructions
- 16 General Registers

Chapter 2
3200 PROCESSOR ARCHITECTURE

INTRODUCTION

The Ridge 3200 processor consists of four printed circuit boards. These are:

- Instruction Unit
- Register File / Multiplier
- Execution Unit
- Cache / Memory Controller

A private bus to the memory controller provides separate 32-bit address and data lines. The instruction unit and execution unit can each access main memory through the cache. Memory access time for a LOAD instruction is 250 nanoseconds, which includes virtual-to-real memory translation and error correction.

The general organization of the 3200 processor is illustrated in Figure 2-1. A more detailed block diagram of the processor is illustrated in Figure 2-2. References to the processor components illustrated in Figure 2-2 appear as *italics* in the course of this chapter.

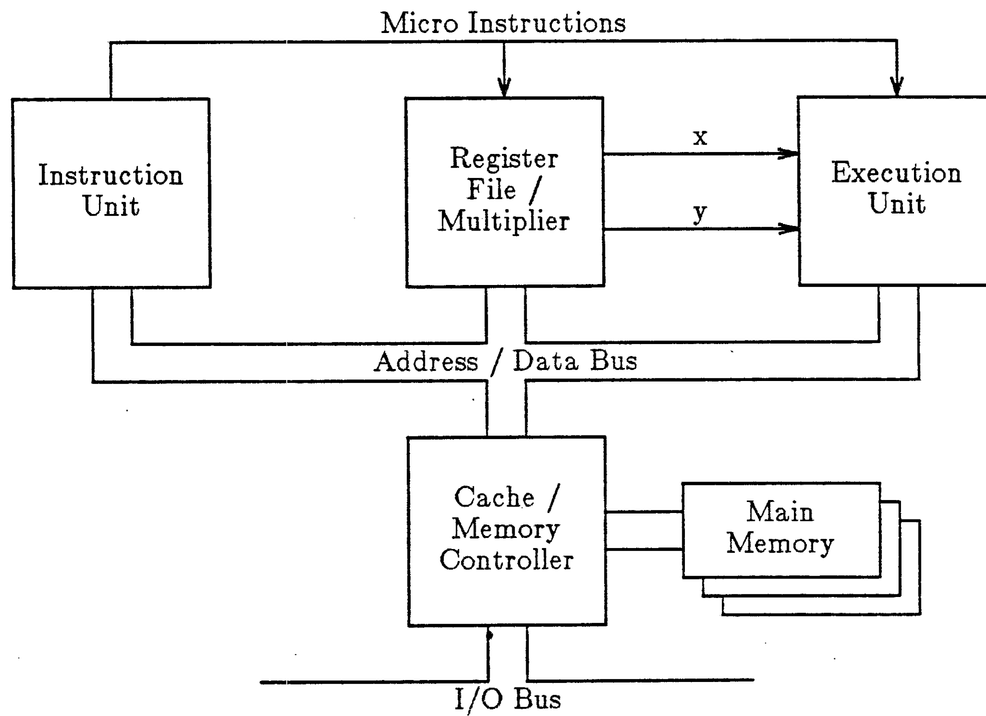


Figure 2-1. General Organization of 3200 Processor

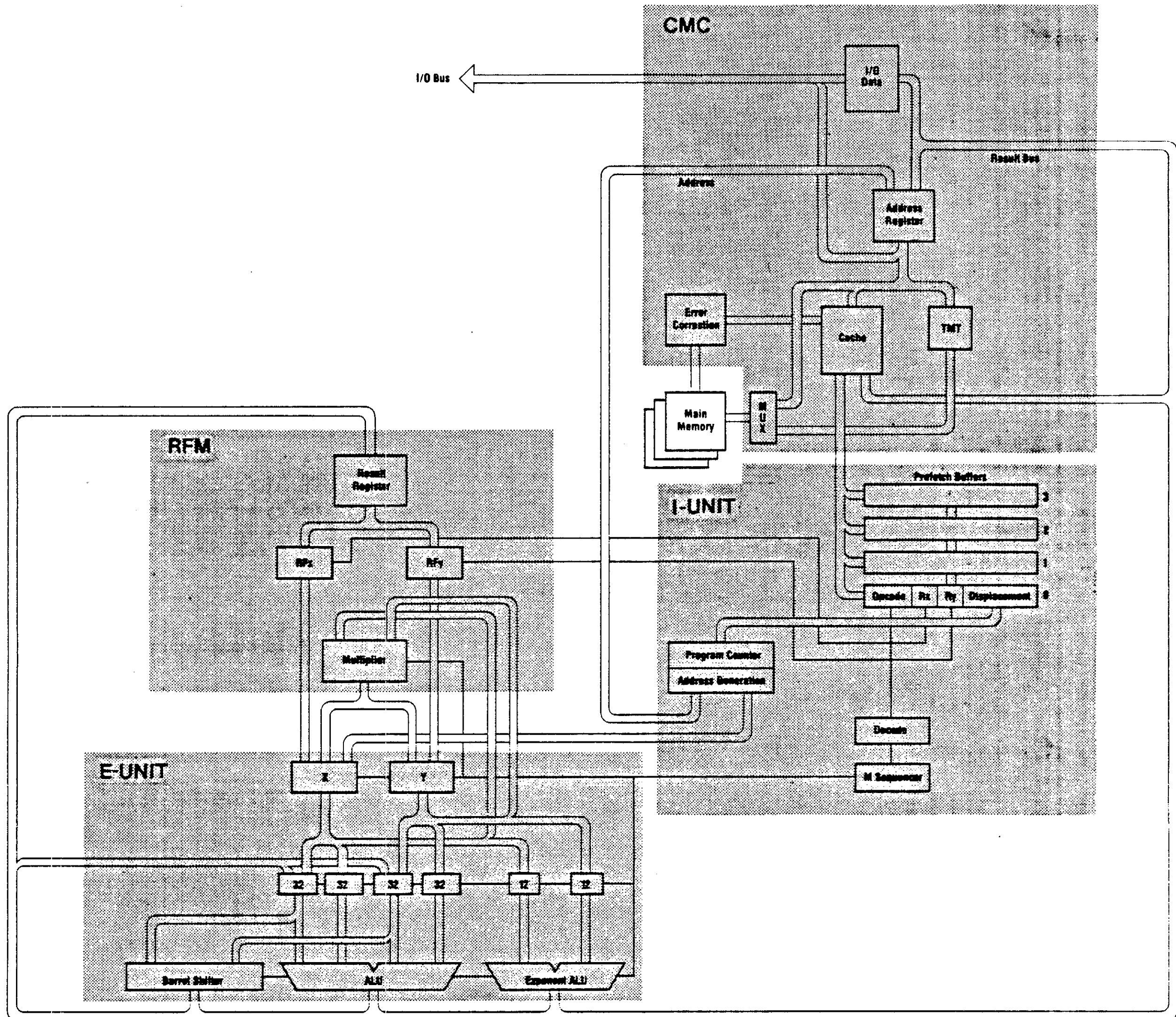


Figure 2-2. Internal Structure of the 3200 Processor

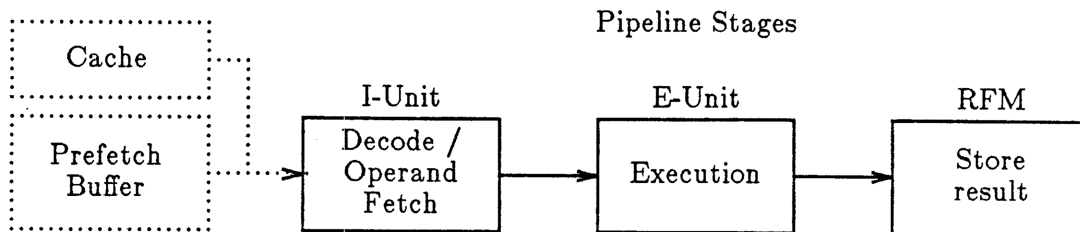
INSTRUCTION UNIT

The instruction unit (*I-UNIT*) performs instruction prefetch and decoding. It contains 4 *prefetch buffers*, *decode*, *address generation*, branch prediction, and interrupt logic. The I-unit also contains a *microsequencer* with 4 banks of control store to provide four-way branching.

The I-unit fetches instructions from the *cache* ahead of the execution unit and stores them in four word-wide *prefetch buffers*. Typically, instructions are held in the *prefetch buffers* before entering a 3-stage instruction pipeline. However, if the prefetch buffers are empty, the I-unit can fetch instructions directly from the *cache*.

INSTRUCTION PIPELINE

The instruction pipeline is composed of three stages: decode/operand fetch, execution, and store result. Each pipeline stage performs its function in one processor cycle. The stages of the processor instruction pipeline are illustrated below.



When the I-unit places an instruction in the pipeline, the following sequence begins:

Decode & Operand Fetch. This first pipeline stage is performed in the I-unit. The *Rx* and *Ry* operands in the instruction are used to enable the register select logic. The decoder interprets the instruction to determine what function is to take place. And the *microsequencer* begins executing microcode.

Execution. The *Execution Unit* operates on Rx and Ry, the result either passes through the *ALU* or the *barrel shifter* and is stored in the *result register*.

Store Result. The data is moved from the result register into the *RFx* and *RFy* register files in the RFM Unit.

The purpose of the pipeline is to increase machine throughput by using parallelism. In a single clock cycle, an instruction can be fetched from memory, another instruction can be decoded, while another instruction is executed and the results produced for another instruction are stored. By pipelining instructions in this way, results can be produced as frequently as every clock cycle.

Cycles	Cache or Prefetch Buffer	Decode & Operand Fetch	Execute	Store Result
1	instruction 1	--	--	--
2	instruction 2	instruction 1	--	--
3	instruction 3	instruction 2	instruction 1	--
4	instruction 4	instruction 3	instruction 2	instruction 1
5	--	instruction 4	instruction 3	instruction 2
6	--	--	instruction 4	instruction 3
7	--	--	--	instruction 4

Figure 2-3. Instruction Flow Through Pipeline Stages.

BRANCH PREDICTION LOGIC

The implementation of branch instructions is critical to the performance of pipelined machines. Without branch prediction, a conditional branch instruction would empty the pipeline, preventing the processor from prefetching the next instruction until the outcome of the branch has been determined.

For this reason, branches can be among the slowest instructions on high performance machines. The Ridge processor incorporates branch prediction logic which loads the instruction that is the most likely result of the branch. This keeps the pipeline full and reduces the chance that the pipeline will be loaded with instructions on the wrong path.

Conditional Branch Instructions

Conditional branch instructions contain a static prediction bit in the instruction displacement field that can be set by a compiler. The branch prediction logic in the instruction unit then fetches along the predicted path. This keeps the pipeline full and makes conditional branch instructions fast.

Branch Prediction Example

Consider Pascal REPEAT ... UNTIL loops. The loop is constructed by the compiler as a linear section of code ended with a conditional branch. This branch is part of the UNTIL expression. Usually these loops are executed more than once, so the compiler marks the conditional branch at the bottom of the loop to be "predicted."

When the program is executed, the processor fetches and executes all the instructions in the linear portion of the loop. As the instruction unit decodes the conditional branch at the end of the loop, the prediction bit is detected. Instead of fetching the next sequential instruction as it normally would, the instruction unit fetches the instruction at the top of the loop, which is the branch target. This prefetching the location of the branch target allows loops to execute at the same speed as linear sections of code.

As the loop is executed for its last time, the instruction unit incorrectly fetches the instruction at the top of the loop. This time the UNTIL condition has been reached, and the loop has ended. Now the instruction unit must flush this instruction, then fetch the next sequential instruction to be executed.

An incorrectly predicted conditional branch instruction causes a 1 cycle delay. Measurements have shown this to be infrequent and overall program speed to be increased by the use of the branch prediction logic.

For example, the following PASCAL program:

```
I := 0;
REPEAT;
    J := I;
    I := I+1;
UNTIL I=100;
```

can be represented by the following instructions:

```
       MOVE   R0,0            ; I := 0
       LADDR  R2,100          ; Load 100 into R2 (Loop Terminator)
LOOP:  MOVE   R1,R0           ; Identify loop start, J := I
       ADD    R0,1            ; I := I+1
       BR     R0 < R2, LOOP!  ; Loop until I = 100
                              ; "!" sets branch prediction bit
       STORE  R1,J            ; Store value of R1 at J
```

The following illustrates the path of each instruction through each stage of the pipeline:

Proc. Cycles	Cache or Prefetch Buffer	Decode & Operand Fetch	Execute	Store Result	Comments
1	MOVE	--	--	--	
2	ADD	MOVE	--	--	
3	BR	ADD	MOVE	--	1st MOVE executed
4	--	BR	ADD	MOVE	Prediction bit detected 1st ADD executed
5	MOVE	--	BR	ADD	Check Branch Condition BR target (MOVE) fetched
6	ADD	MOVE	--	--	
7	BR	ADD	MOVE	--	2nd time through loop - second MOVE executed
8	--	BR	ADD	MOVE	2nd ADD executed
9	MOVE	--	BR	ADD	
.	.	.	.	.	
.	.	.	.	.	
n	--	BR	ADD	MOVE	Branch Prediction
n+1	MOVE	--	BR	ADD	I = 100, loop complete Incorrectly Predicted Branch
n+2	STORE	--	--	--	Flush pipeline. Fetch STORE instruction
n+3	--	STORE	--	--	STORE instruction decoded
n+4	--	--	STORE	--	
n+5	--	--	STORE	--	
n+6	--	--	--	STORE	

Figure 2-4. Branch Prediction Example

Unconditional Branch Instructions

Unconditional branch instructions also make use of the branch prediction and decode logic in the instruction unit. In unconditional branches, the instruction is decoded, the target location is fetched and placed in the instruction stream.

REGISTER FILE / MULTIPLIER UNIT

The Register File / Multiplier (*RFM*) unit contains the register file and multiplication logic. Both the Register File and Multiplier portions are double clocked. In all other respects, these portions are functionally independent of one another.

REGISTER FILE

The register file portion of the RFM consists of two register files: *RFx* and *RFy*. The *RFx* register file is made up of 16 general registers, 16 special purpose registers, and a scratch pad area for storing intermediate results and constants. The contents of the *RFx* register file are duplicated in the *RFy* register file. Duplicating the registers allows both Rx and Ry to be accessed in a single clock cycle.

During an instruction execution, the register file is accessed in the operand fetch cycle for reading the data from a given register(s) and in store cycle for writing the result back into a given register(s). Since instructions overlap, both read and write capability in a single cycle is required. This is accomplished by dividing the clock cycle into two halves. The first half is used for write access and the second half is for read access.

MULTIPLIER

The multiplier operates on integers and the mantissa portion of floating point numbers.

The multiplier is based on a modified version of Booth's recoded algorithm with overlapped scanning to process 4 bits during each half-clock cycle. This allows 2 4-bit multiplies to be done in parallel during each clock cycle.

The RFM unit supports five different modes of multiplication:

- Integer Multiply; 32 x 32-bit with 32-bit product.
- Integer (Immediate) Multiply; 32 x 4-bit with 32-bit product.
- Extended Integer Multiply; 32 x 32-bit with 64-bit product.
- Single Precision Real Multiply; 24 x 24-bit with 25-bit mantissa product.
- Double Precision Real Multiply; 53 x 53-bit with 54-bit mantissa product.

During floating point multiplication, instructions are unpacked and the exponent portion of the operands set to the *E-unit* and the mantissa portion loaded into the *RFM* operand registers. The product of the mantissas is then sent to the *E-unit*, where post normalization and rounding are performed. The product is then merged with the exponent.

EXECUTION UNIT

The arithmetic logic units *(ALU)*, *barrel shifter*, and register bypass hardware are all contained in the Execution Unit (*E-UNIT*). The 64-bit *barrel shifter* can shift up to 64 bits left, right, or circularly in a single clock cycle.

The general data flow for numbers through the execution unit is as follows: Data is fetched from the *RFx* and *RFy* register files in the *Register File / Multiplier* (RFM) unit; put into the *x* and *y* registers; operated on by the *ALU*, then temporarily held in the *result register* before being stored in the register files. Should data not yet stored in the *RFM* unit be needed in a computation, the register select logic may bypass the *RFM* unit and use the data on the bus as input to the *ALU*.

The execution of floating point instructions is made more efficient by unpacking and sending the exponent to the *exponent ALU* and the mantissa to the standard *ALU*. The 64-bit *barrel shifter* packs (reassembles) the results from the registers into floating point values.

REGISTER BYPASS FUNCTION

The execution unit incorporates register bypass hardware that avoids the "pipeline interlock" delay that results when an instruction's operand is dependent on an instruction still in the pipe.

The register bypass function can be illustrated by the following two-instruction sequence that utilizes the register bypass data path in the execution unit. This example also illustrates the use of the instruction pipeline discussed earlier.

The following instruction sequence executes as shown in the table below:

```
ADD   R6, R7   ;R6 is added to R7 and the sum is put in R6.
AND   R5, R6   ;R5 logically ANDs with R6 and the result is put in R5.
```

Instruction Pipeline Stage Operation		
Clock Cycle	**ADD**	**AND**
1	The ADD instruction is fetched.	
2	R6 and R7 are fetched from the register files.	The AND instruction is fetched.
3	The ALU ADDs R6 and R7, and puts the new R6 value on the bus.	R5 and R6 are to be fetched, but the new R6 value is on the bus, not in the register file. R5 is fetched from the register file, while the Ry register select logic bypasses the register file and uses the R6 value from the bus.
4	The new R6 value is stored in the register file	The ALU ANDs R5 with R6 and puts the new R5 value on the bus.
5		The new R5 value is stored in the register file.

During clock cycle 3, the AND instruction must fetch its operand R6. However, the value of R6 in the RFM unit is outdated due to the ADD instruction computing a new R6 value. Consequently, the register bypass is used. This moves instructions through each pipeline stage in one clock cycle, and allows the pipeline to complete one instruction each clock cycle.

CACHE / MEMORY CONTROLLER

The Cache / Memory Controller (*CMC*) contains the virtual-to-real translation hardware, *error correction* logic, a 16k-byte code/data *cache*, and handles all memory data for the processor and I/O devices. All virtual memory accesses from the processor go through the translation mapping table (*TMT*) where they are converted to real addresses and presented to the code/data *cache* and/or *main memory*. I/O devices on the *I/O bus* use real addresses and bypass the translation mapping table (*TMT*).

The memory controller processes four bytes (1 word) per cycle. Cycle time from the *cache* is 83 nanoseconds and access time for a LOAD from *main memory* is 250 nanoseconds. The CPU memory bus runs at 83 nanosecond clock cycles, giving it a bandwidth of 27.4 Mbytes per second on read and 19 Mbytes per second on write. The *I/O bus* uses multiplexed address and data lines to minimize the use of connector pins on I/O boards. The I/O bus runs at 125 nanosecond clock cycles. The bandwidth for direct memory access (DMA) is approximately 14 Mbytes per second on both read and write. Each board on the I/O bus contains its own DMA logic.

The memory controller can access from 4 to 128 megabytes of *main memory*. All memory accesses are single-bit error corrected and double-bit error detected.

See Chapter 3 for details on the Ridge I/O system and Chapter 7 for a complete discussion of the virtual-to-physical memory translation process.

CODE/DATA CACHE

The memory controller contains a two-way set associative, 16k-byte cache which services both code and data. This cache allows the CPU to access a 32-bit word in a single 83 ns clock cycle.

The cache is divided into two sets. Each set contains a separate *tag table* and *data table*. Each data table contains 512 lines consisting of 4 32-bit words, or "quadwords." Lines 0 - 255 of the data table contain code; lines 256 - 511 contain data. The tag table contains the physical address of each code or data value located in the data table.

If the data required by the processor is not in the cache, the processor will load the data from main memory into the cache data table. Whenever the processor reads a word from memory, the entire quadword in which that word is located is loaded as a single line into the cache data table.

All writes are write-through, which means data is written to both the cache and main memory. LOAD instructions require 2 CPU clock cycles and STORE instructions require 3 CPU clock cycles.

Figure 2-5 illustrates how the virtual address is used to locate data in the cache. Bits 20..27 of the virtual address determine which line to access in the tag and data tables. Since the cache data table lines are four words across, bits 28..29 of the virtual address select the word in the line.

When locating data in the cache, bits 0..10 of the virtual address are compared with a *tag* field in the TMT. Bits 11..19 of the virtual address are used to index into the TMT, which translates the virtual address to a physical address (as described in Chapter 7). This value is compared to both physical address values located in the tag tables. If the *tag* field of the virtual address matches that of the TMT and the physical address produced by the TMT matches either of the two physical addresses located in the tag tables, then a "cache hit" occurs. In this event, the code or data value (as identified by the *code/data bit*) is read from the data table that corresponds to the tag table containing the matching physical address.

The main features of the Ridge Code/Data Cache can be illustrated as follows:

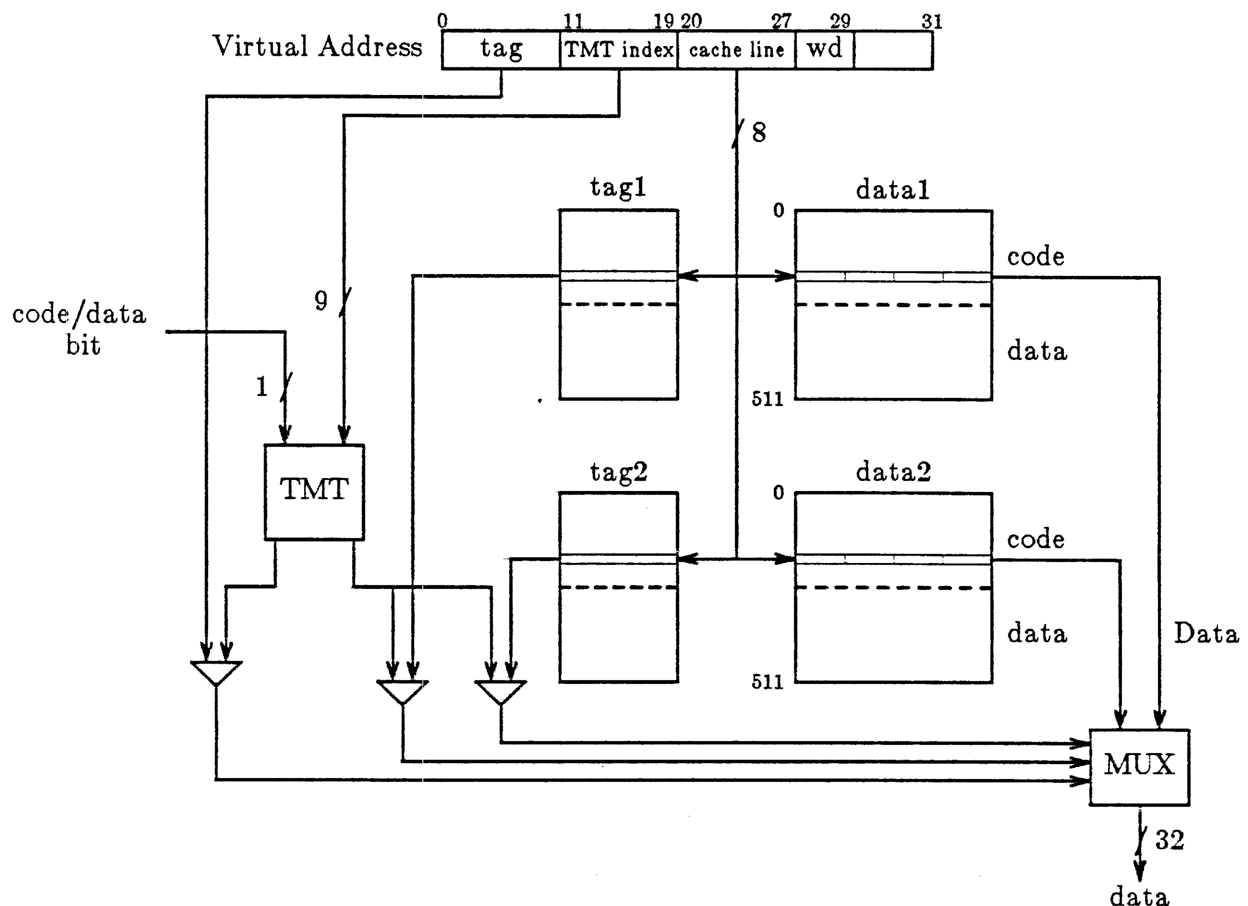


Figure 2-5. Ridge Code/Data Cache Organization

See Chapter 7 for details on the TMT and the virtual-to-real address translation process.

Chapter 3
RIDGE I/O SYSTEM

INTRODUCTION

This chapter describes the basic functions of the Ridge I/O system.

Data and addresses are transferred between the memory controller and the I/O boards by means of a synchronous bus with multiplexed 32-bit address and data. Communication between the memory controller and the I/O boards consists of *I/O Read/Write*, *I/O Interrupts*, and *Direct Memory Access (DMA) Read/Write* operations. I/O board priorities are resolved by daisy-chained priority signals.

DMA hardware contains a *burst mode* feature that allows four 32-bit words to be transferred in a single read or write operation. The number of 125 ns clocks and the resulting bandwidths on burst mode and non-burst mode operations are as follows:

	Non-Burst		**Burst**	
Operation	clocks 125 ns	Mbytes/sec	clocks 125 ns	Mbytes/sec
DMA Write	5	6.4	9	14
DMA Read	6	5.3	9	14

Figure 3-1 illustrates the basic structure of the Ridge I/O bus.

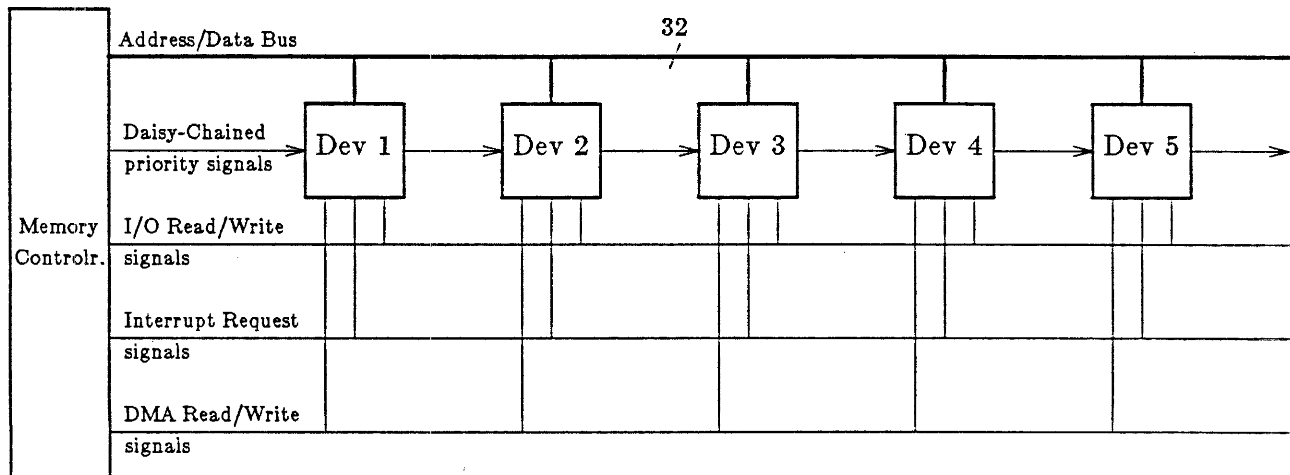


Figure 3-1. Ridge I/O Bus

I/O OPERATIONS

Each type of I/O operation and its control signals are described below. A minus sign preceding the signal indicates that signal is asserted when it goes low; a plus sign indicates that signal is asserted when it goes high.

I/O READ / WRITE

The I/O read/write operations are used to transfer 32-bit words between the processor and an individual I/O board. The I/O Read (IOR) operation moves a 32-bit word from an I/O board to the processor. The I/O Write (IOW) operation moves a 32-bit word from the processor to an I/O board. Both I/O Read and I/O Write are initiated by the memory controller upon execution of a READ or WRITE instruction in the processor.

The I/O read/write operations use the following signals:

-MCIOREQ *Memory Controller I/O Request.* The memory controller asserts this signal to start an I/O operation. MCIOREQ enables the device number and operation code of the *IO Address Word* from the WRITE or READ instruction on the IODATA bus.

-ACKMCIO *Acknowledge memory controller I/O.* This signal is the I/O board's response to the MCIOREQ signal. ACKMCIO is an open collector signal.

-IODACK *I/O Data Acknowledge.* This signal indicates that the memory controller is driving data onto the IODATA bus. The memory controller asserts IODACK along with the I/O Address Word and the Write Data. This signal is also asserted during memory read operations, as described in the *DMA READ / WRITE* section.

-MCIOW *Memory Controller I/O Write.* The memory controller asserts this signal for IOW operations. MCIOW is valid while the I/O Address Word is sent.

-IODNVM *I/O Data Not Valid Memory.* If the data on the bus is invalid, this signal will be asserted after the I/O board has recognized the MCIOREQ signal and has responded with ACKMCIO. This signal can be sent with both IOR and IOW operations. If this signal is asserted during an IOR operation, the data is read, but indicated as being invalid. During an IOW operation, this signal can be used by the I/O board if it is not ready to accept the data. IODNVM is an open collector signal.

-IODATA *I/O data.* IODATA is a 32-bit bidirectional, tri-state bus on which data and the I/O Address Word are multiplexed. Bit 0 is the most significant and bit 31 is the least significant.

I/O Read Operation

An I/O Read (IOR) operation moves one 32-bit word from an I/O board, through the memory controller, to the processor. When the processor executes a READ instruction, it issues a request to the memory controller for an I/O read data (IORD) word from the I/O board specified in the I/O read address (IORA) word.

The I/O read operation can be illustrated by the following:

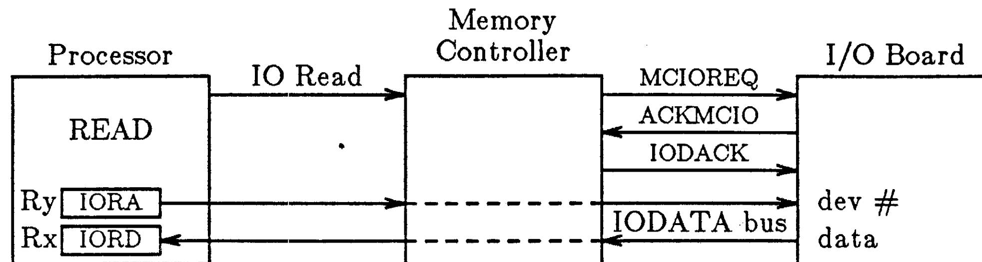


Figure 3-2. I/O Read Operation

The timing diagram in Figure 3-3 illustrates the discussion in this section.

The memory controller requires 3 states (P1, P2, P3) of an internal state machine to complete an IOR. During the P1 state, the memory controller asserts an MCIOREQ signal to request an I/O operation. Bits 0..7 of the I/O Address Word on the IODATA bus contain the address of the I/O board to be accessed and bits 8..31 contain information unique to the selected device.

If the value in bits 0..7 of the I/O Address Word match the device number switch on an I/O board, that I/O board will assert an ACKMCIO. The I/O board is now ready to transfer data. ACKMCIO must be asserted within 15 clock cycles or the memory controller will timeout. In this event, bit 30 in Rx is set and the timeout error is reported to software.

The I/O board transfers the read data on the IODATA bus the clock cycle after the ACKMCIO signal was asserted. If the I/O board does not want to perform the transfer, IODNVM should be asserted to avoid a timeout error. The P3 state is used internally by the memory controller.

Below is the timing diagram for an I/O read. Note that P1 will be at least 2 clock periods (up to 15). IODNVM is only asserted to indicate that the data is not valid or to avoid an I/O timeout.

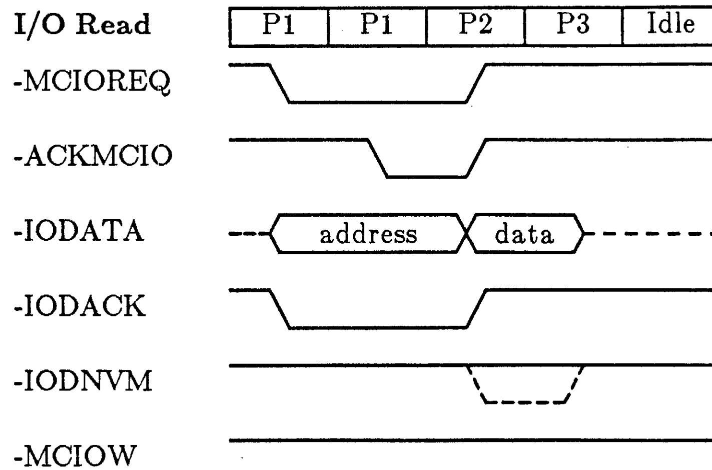


Figure 3-3. I/O Read Timing Diagram

I/O Write Operation

An I/O Write (IOW) operation moves one 32-bit word from the processor, through the memory controller, to an I/O board. When the processor executes a WRITE instruction, it issues a request to the memory controller to send an I/O write data (IOWD) word to the I/O board specified in the I/O write address (IOWA) word.

The I/O write operation can be illustrated by the following:

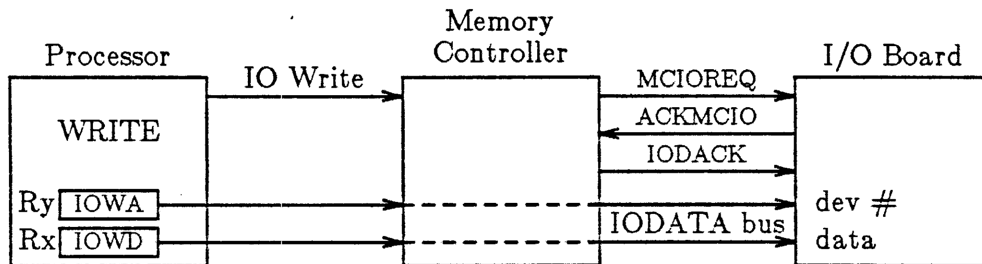


Figure 3-4. I/O Write Operation

The timing diagram in Figure 3-5 illustrates the discussion in this section.

The memory controller requires 3 states (P1, P2, P3) of an internal state machine to complete an IOW. During the P1 state, the memory controller asserts an MCIOREQ signal to request an I/O operation. Bits 0..7 of the I/O Address Word on the IODATA bus contain the address of the I/O board to be accessed and bits 8..31 contain information unique to the selected device.

If the value in bits 0..7 of the I/O Address Word match the device number switch on the I/O board, the I/O board asserts an ACKMCIO. The I/O board is now ready to receive data. ACKMCIO must be asserted within 15 clock cycles or the memory controller will timeout. In this event, bit 30 in Rx is set and the timeout error is reported to the software.

During P3, the memory controller asserts IODACK to indicate that the memory controller has put the IOW data word is on the IODATA bus.

If the I/O board is not ready to accept the data, it can assert an IODNVM signal on the clock cycle after ACKMCIO was asserted.

Below is the timing diagram for an I/O write.

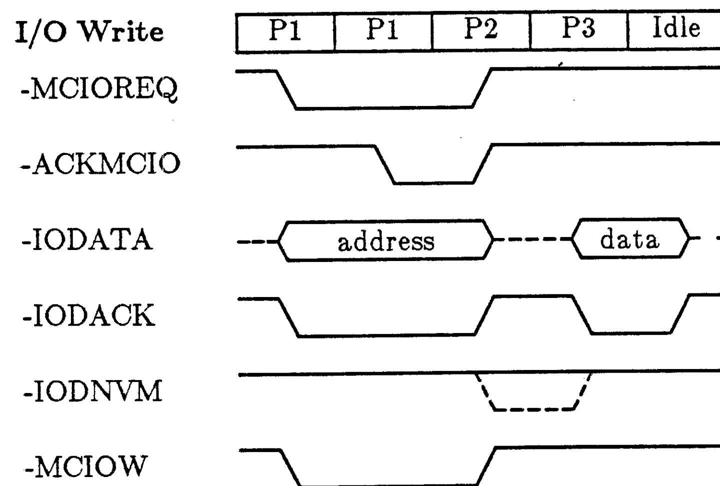


Figure 3-5. I/O Write Timing Diagram

I/O INTERRUPTS

An I/O interrupt signals the processor that an I/O board requires service. When an I/O interrupt occurs, the interrupt handler returns an I/O Interrupt Read (IOIR) word from the I/O board, through the memory controller, to the processor.

The signals needed for an I/O interrupt read (IOIR) are:

-IOIREQ — *I/O Interrupt Request.* This is an open collector signal used by the I/O boards to request interrupts. When an IOIREQ signal is active, there are one or more devices requesting an interrupt. This signal will remain active until all the requesting devices receive an Acknowledge I/O Interrupt (ACKIOI, see below).

+ACKIOI — *Acknowledge I/O Interrupt.* This signal is used by the memory controller to acknowledge an I/O board's interrupt request. ACKIOI is daisy-chained from I/O board to I/O board. This daisy chain exists as an ACKIOI*in* and an ACKIOI*out* signal on each I/O board. When an I/O board is not making an interrupt request, it will pass ACKIOI*in* through to ACKIOI*out*. If an I/O board is requesting an interrupt, it is waiting for the ACKIOI*in* and will block the ACKIOI*out*. (see the *BUS CONTENTION section for details.)*

-IODATA — *I/O Data.* The I/O board sends the device number (bits 0..7) and other device-specific information (IOIR word) on the IODATA bus.

Interrupt Operation

An interrupt operation requires 3 states (P1, P2, P3). See Figure 3-6, below.

The I/O board asserts an IOIREQ, requesting an interrupt. At the end of the P1 state, the memory controller acknowledges the interrupt request by generating an ACKIOI. (The P1 state may be longer than one clock cycle.)

The I/O board transfers an IO Interrupt Read word (IOIR) on the IODATA bus in response to the ACKIOI on the previous clock cycle. Bits 0..7 of the IOIR word contain the address of the requesting I/O board and bits 8..31 contain device-specific information.

See the *BUS CONTENTION* section for details on how multiple requests for the I/O bus are handled.

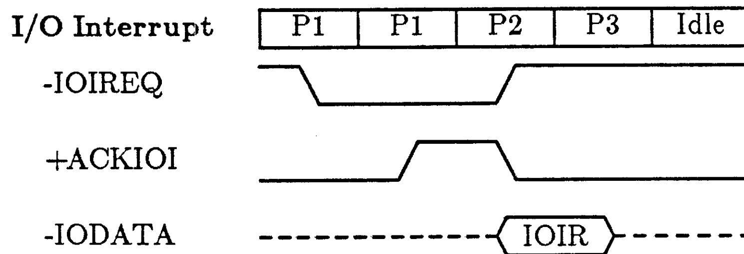


Figure 3-6. Interrupt Timing Diagram

DMA READ / WRITE

A Direct Memory Access (DMA) operation transfers one or four words of data directly between the I/O board and the memory controller, with no intervention from the processor. DMA Read can transfer up to 16 Mbytes of data per second. DMA Write can transfer up to 18.3 Mbytes of data per second.

The signals needed for an DMA Read or Write are:

-IOMREQ — *I/O Memory Request.* IOMREQ is an open collector signal used by I/O boards to request a DMA Read or DMA Write operation.

-IODACK — *I/O Data Acknowledge.* IODACK is asserted by the memory controller to transfer data onto the IODATA bus. IODACK is not asserted during DMA Writes.

-MDNVIO — *Memory Data Not Valid I/O.* MDNVIO is asserted if a double-bit parity error has occurred. MDNVIO is not asserted during DMA Writes.

+ACKIOM — *Acknowledge I/O Memory Request.* This signal is used by the memory controller to respond to an I/O board's IOMREQ signal. ACKIOM is daisy-chained from I/O board to I/O board. Like the ACKIOI signal described in the *INTERRUPTS* section, the ACKIOM signal will not be propagated to lower priority I/O boards while a higher priority I/O board is asserting an IOMREQ signal. (See the *BUS CONTENTION* section for details.)

-IODATA — *I/O Data.* The I/O board sends the DMA address and data on the IODATA bus. Bits 30 and 31 of the address have special meaning. Bit 30 is not used and bit 31 indicates whether the DMA operation is a read or a write. On DMA Write, address bit 31 is asserted and not asserted on DMA Read.

-MULTIWD — *Multi-Word Data Transfer.* The MULTIWD signal requests four 32-bit words of data to be transferred over the IODATA bus. On multiple word data transfers, this signal will be asserted by the I/O board at the same time the memory address is asserted on the IODATA bus.

DMA Read Operation

A DMA Read operation transfers data directly from memory to the I/O board. See Figure 3-7.

An I/O board requests a DMA operation by asserting an IOMREQ signal. When the memory controller recognizes the IOMREQ it asserts an ACKIOM. (The ACKIOM signal will not be propagated to lower priority I/O boards if a higher priority I/O board has asserted an IOMREQ.)

The I/O board then sends the DMA address to the memory controller on the IODATA bus. Bit 31 of the address is not asserted, indicating a DMA Read operation.

The memory controller latches the DMA address and reads the specified address in memory.

The memory controller asserts IODACK for one or four clock cycles to transfer one or four words of data onto the IODATA bus. If a double-bit parity error is detected by the memory controller, then MDNVIO is asserted.

If the MULTIWD signal was asserted on the same cycle the address was on the IODATA bus, there will be 4 successive data words on the bus.

The timing diagram in Figure 3-7 shows a DMA Read operation. The last four clock cycles assume that the MULTIWD signal was asserted during the third clock cycle.

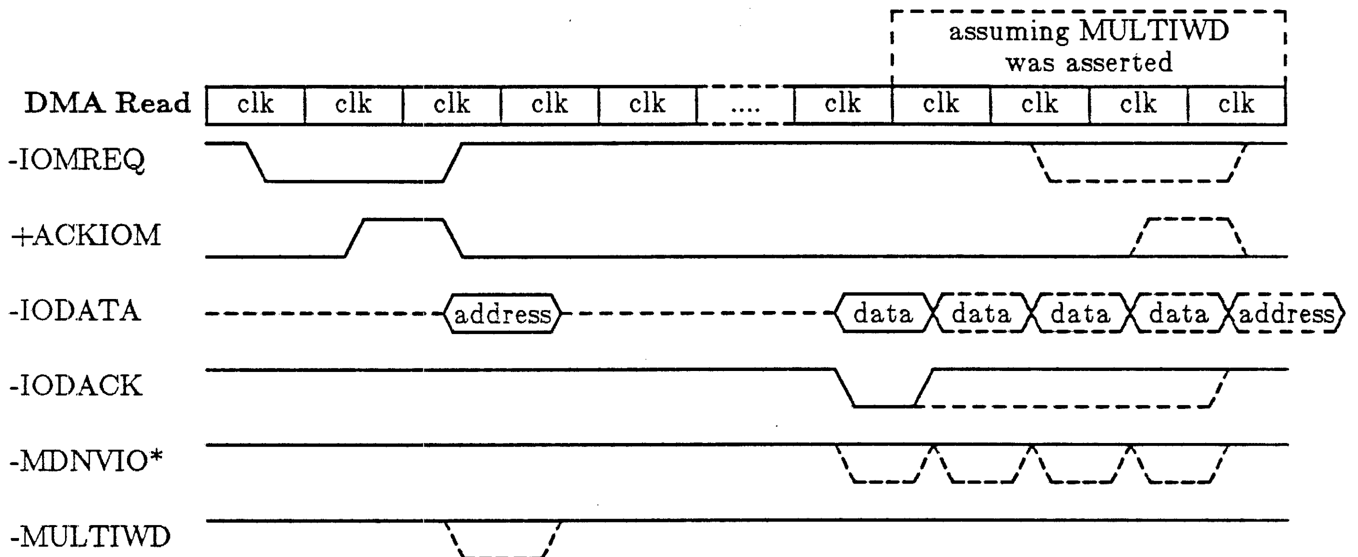


Figure 3-7. DMA Read Timing Diagram

* Active on double-bit errors.

DMA Write Operation

A DMA Write operation transfers data directly from the I/O board to memory. See Figure 3-8.

An I/O board requests a DMA operation by asserting an IOMREQ signal. When the memory controller recognizes the IOMREQ it asserts an ACKIOM. (The ACKIOM signal will not be propagated to lower priority I/O boards if a higher priority I/O board has asserted an IOMREQ.)

The I/O board then sends the DMA address to the memory controller on the IODATA bus. Bit 31 of the address is asserted, indicating a DMA Write operation.

If the MULTIWD signal was asserted on the same cycle the address was on the IODATA bus, there will be 4 successive data words on the bus.

The memory controller latches the DMA address and writes the data.

The timing diagram in Figure 3-8 shows a DMA Write operation. The last six clock cycles assume that the MULTIWD signal was asserted during the third clock cycle.

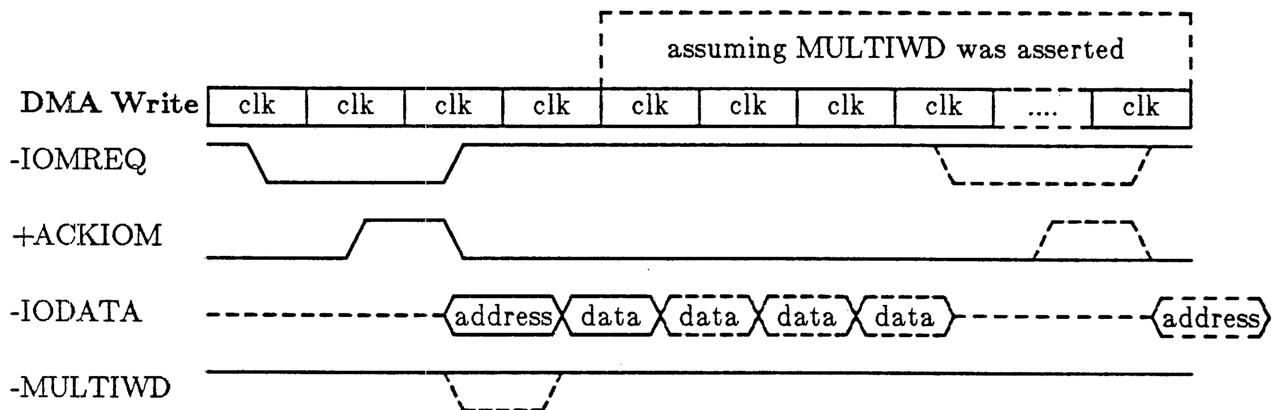


Figure 3-8. DMA Write Timing Diagram

BUS CONTENTION

The Ridge I/O system's two acknowledge signals, ACKIOI and ACKIOM, consist of two wires linked from I/O board to I/O board in a configuration referred to as a *daisy chain.*

When more than one I/O board issues a request to use the I/O bus, the I/O board acknowledged by the memory controller is determined by its position in the daisy chain.

Bus contention is resolved by the *ACKin* and *ACKout* actions on each I/O board (see Figure 3-9). When an acknowledge (ACK) signal is asserted by the memory controller, it is first received by the highest priority I/O board on the daisy chain. If a request (REQ) signal is not active for that I/O board, the ACK signal will be propagated on to the next highest priority board. The ACK signal will be propagated on down the daisy-chained I/O boards in this manner until it reaches the highest priority board asserting a REQ.

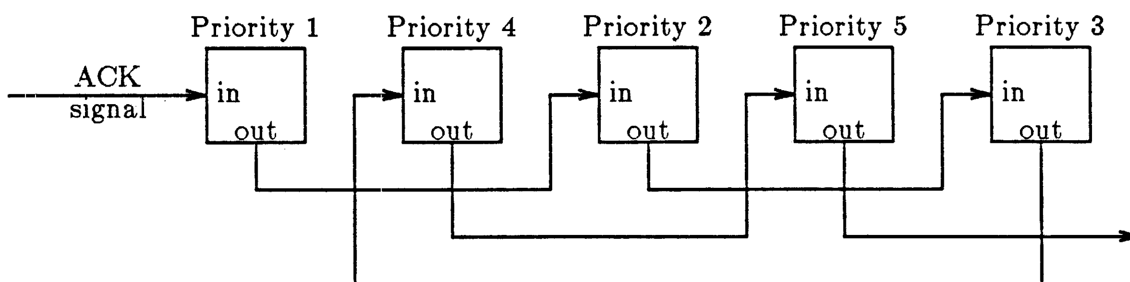


Figure 3-9. Possible Daisy Chain Configuration

If the memory controller asserts an ACK signal in response to a REQ from a lower priority I/O board during the same clock cycle a higher priority board asserts a REQ signal, the ACK signal will be blocked by the higher priority board. Only after the higher priority I/O board has completed its operation, will the memory controller reassert the ACK signal for the lower priority board.

DUAL DAISY-CHAINED SIGNALS

So far, the daisy-chained acknowledge ACKIOI and ACKIOM signals for I/O interrupts and DMA read/write operations have been described as single lines servicing all of the I/O boards.

However, each signal actually consists of two separate daisy chains, each of which services a maximum of 8 I/O boards. For I/O interrupts, the ACKIOI signal consists of ACKIOI1 and ACKIOI2; the ACKIOI2 signal having a higher priority than ACKIOI1. The ACKIOM signal consists of ACKIOM1 and ACKIOM2, which have the same priority scheme as the ACKIOI signals.

Logically, the dual acknowledge signals are the same as a single acknowledge signal.

EXAMPLE OF BUS CONTENTION

All of the signals described thus far are signals that are either asserted or received by the memory controller. However, when considering bus contention, it is important to understand the *internal signals* operating within the individual I/O boards.

The IOIREQ signal, for example, represents the state of the interrupt request signal received by the memory controller. The state of this signal is the logical OR of the *internal request* signals within the individual I/O boards. This means that, when the IOIREQ signal is asserted, the memory controller knows an interrupt request has been asserted by *at least* one I/O board. If more than one board has asserted a request, the board that is acknowledged by the memory controller will depend on its position in the daisy chain.

The ACKIOI signal, on the other hand, is the acknowledge signal asserted by the memory controller in response to an IOIREQ. This signal is received by the I/O board as an ACKIOI*in* signal. If the I/O board is asserting a request, the ACKIOI*in* signal is interpreted as an *internal acknowledge.* If no request is asserted, then the I/O board asserts an ACKIOI*out.*

The following example of bus contention involves two I/O boards competing for service. The principles illustrated here can be applied to most situations of bus contention.

The timing diagram in Figure 3-10 illustrates a situation in which two I/O boards assert an IOIREQ signal to request an interrupt. The IOIREQ signal was initially asserted by the lower priority I/O board. However, the ACKIOI signal is intercepted by a higher priority board, which asserts its IOIREQ during the same clock cycle the ACKIOI signal is asserted by the memory controller.

Note the state of the individual boards' request and acknowledge signals and their relationship to the corresponding signals at the memory controller.

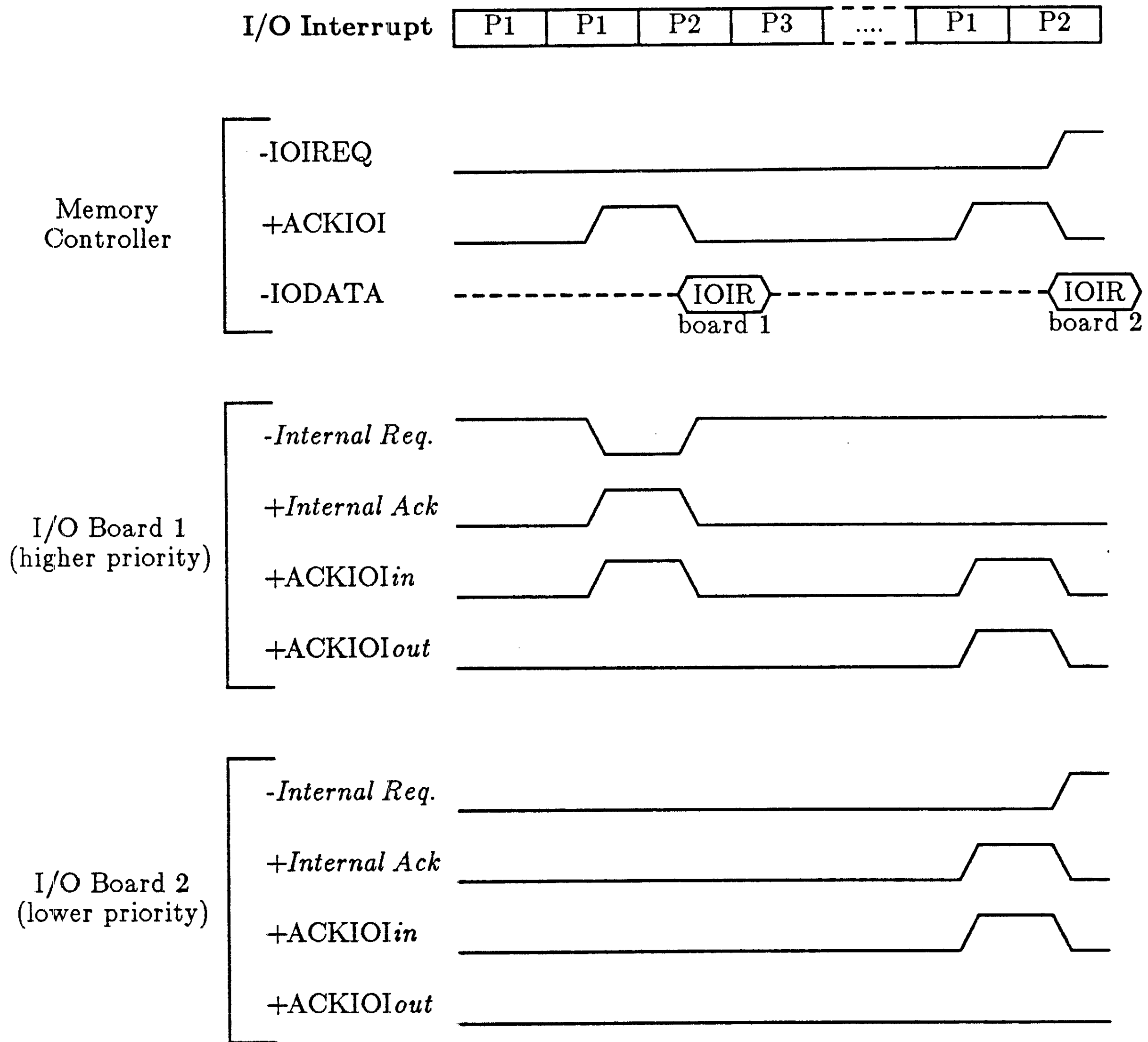


Figure 3-10. Contention for bus during I/O interrupt

Chapter 4
INSTRUCTION FORMATS AND DATA TYPES

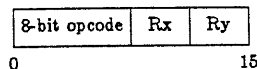
INTRODUCTION

Instructions operate on the general registers or on a register and a memory location (load from memory or store to memory). The program counter is manipulated by program control instructions, such as subroutine call and branch.

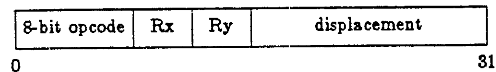
All addresses generated by the processor are 32-bit virtual addresses. Memory reference instructions indicate code or data space by utilizing a bit in the instruction opcode. (Code and data segments are discussed in Chapter 5.)

The Ridge instruction set uses three instruction formats:

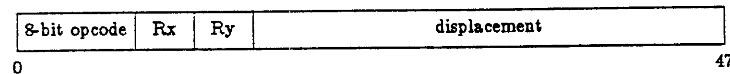
Register-to-register



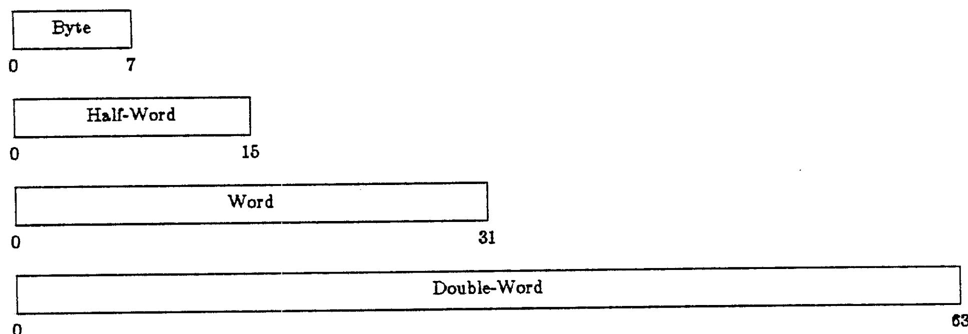
Short displacement memory address



Long displacement memory address



Four sizes of operands are loaded and stored by the Ridge processor. The basic addressable unit is the 8-bit *byte*. The other operand sizes are the *halfword* (16-bits), the *word* (32-bits) and the *double-word* (64-bits).



The *DATA STORAGE IN REGISTERS* section at the end of this chapter illustrates how each type of operand is stored in registers.

GENERAL REGISTERS

There are 16 32-bit general-purpose registers available for use as data, indexing, and addressing. Any arithmetic or address operation can be performed on any register. Registers are not specialized for counting or indexing.

INSTRUCTION FORMATS

All Ridge instructions use an eight-bit opcode followed by two four-bit operand specifiers. The first operand specifier (Rx) names the register, or register pair containing the operand. The second operand specifier (Ry) names the register(s) or is a four-bit constant.

The type of operand specified depends on which opcode is used.

REGISTER INSTRUCTIONS

Register instructions use the contents of the general registers or a constant as their operands and, therefore, do not require access to memory. In most register instruction operations, the contents of Rx are replaced with the result.

There are two types of register instructions:

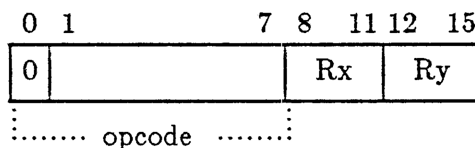
- Register-to-Register
- Register-Immediate

Register-to-register instructions operate on the contents of Rx and Ry. Most register instructions are of this type.

Register-Immediate instructions interpret the 4-bit value of the Ry register field as the operand, rather than as the address of the general register containing the operand. The constant stored in Ry can be an integer in the range from 0 to 15.

Register Instruction Format

The register instruction format is as follows:



A register instruction is specified by bit 0 of the instruction being off.

MEMORY REFERENCE INSTRUCTIONS

Memory reference instructions transfer code and data between the general registers and locations in memory.

There are two types of memory reference instructions:

- Direct Address
- Indexed Address

Direct and indexed instructions exist for accessing either code or data space with either short or long displacement memory address formats discussed below.

The following table describes how the effective addresses for memory reference instructions are calculated.

Memory Instruction Types and Effective Addresses

Instr. Type	Address Space	Effective Address	Description
Direct	Data	Displacement	The memory address is the displacement field from the instruction. All memory references are 32-bit addresses. This form references data space.
	Code	PC + displacement	Instructions that reference code space do so relative to the program counter (PC). PC is added to the displacement field and memory is read from this location. Code space is never written.
Indexed	Data	Ry + displacement	The contents of register Ry are added to the displacement field. Memory is then read or written at this location.
	Code	PC + Ry + displacement	PC is added to the displacement field, the result is added to the contents of Ry. Memory is then read at this location.

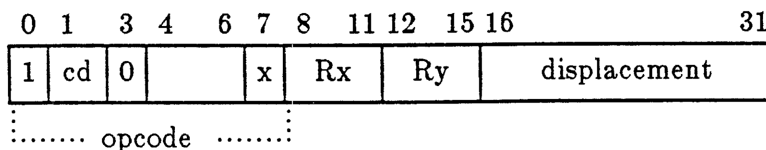
Indexing takes place with full 32-bit signed integers in two's-complement notation. Displacements are also treated as 32-bit signed integers in two's complement notation. Short displacement memory addresses are sign extended to 32 bits by replicating the MSB into the upper 16 bits. The resulting effective address is an absolute displacement from location zero in the data space. Negative addresses (MSB set) are virtual addresses in the range of two to four billion.

These memory address computations allow indexes to be positive or negative relative to the displacement, or allow the displacement to be positive or negative relative to the index. Code space addresses are program counter (PC) relative and thus make code relocatable.

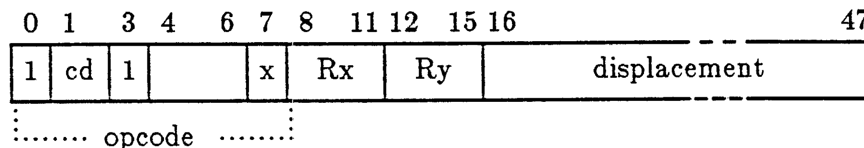
Memory Reference Instruction Formats

Memory reference instructions use either the short displacement or long displacement formats shown below. The short displacement memory address format is used for referencing addresses within +/- 32K bytes of 0 (if direct) or Ry (if indexed). The long displacement memory address format is used for referencing addresses that must be specified in 32 bits.

Short displacement memory address



Long displacement memory address



cd = code or data space reference.
code is specified as 00, 11
data is specified as 01, 10

x = 1 is indexed address
0 is direct address

When bit 0 of the instruction opcode is set, the instruction is a memory reference. Bits 1 and 2 in combination indicate code or data space. Bit 3 specifies displacement size (0 = short, 1 = long). Bit 7 is used to specify that the instruction is indexed.

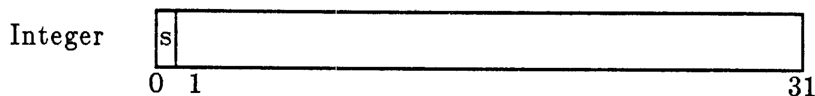
DATA REPRESENTATION

Instructions interpret data in 32-bit and 64-bit quantities. 32-bit data quantities are two's complement signed integers, unsigned integers, and real numbers. 64-bit data quantities consist of 64-bit unsigned integers, double precision real numbers, and 64-bit sets. Integers longer than 32 bits may be manipulated using extended precision integer arithmetic instructions.

The most significant bit (MSB) of any type of signed data is used as the sign bit.

INTEGER REPRESENTATION

Signed integers can range between -2147,483,648 to 2,147,483,647. Unsigned integers can range between 0 to 4,294,967,295.



s = sign bit

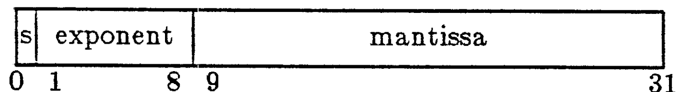
REAL NUMBER REPRESENTATION

Single Precision Real Numbers

Real numbers (represented in floating-point form) consist of three parts: a sign, a power-of-two exponent, and a mantissa. The value of a real number is:

(-1)***s* x 2**(*exponent*-127) x 1.*mantissa*

For positive numbers, the sign bit (bit 0) is 0. For negative numbers, the sign bit is 1. The exponent of a real number is 8 bits long, and is biased by +127. The eight bits of the exponent give a range of 0 through 255. Subtracting the bias yields an exponent range of -127 through +128. The mantissa has an implicit leading one, and is 23 bits long. Zero is represented by all zeros.



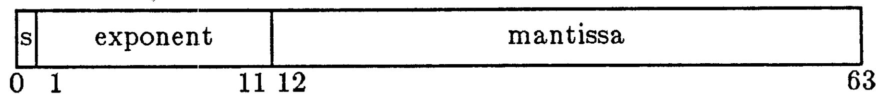
example: *1.2* = 0 01111111 $00110011001100110011010_2$ = $3F99\ 999A_{16}$

example: *-10.6* = 1 10000010 $01010011001100110011100_2$ = $C129\ 999C_{16}$

Double Precision Real Numbers

Double reals are similar to reals, except that the mantissa is 52 bits, and the exponent is 11 bits. The exponent is biased by +1023. The eleven exponent bits give a range of 0 through 2047.

Subtracting the bias yields an exponent range of -1023 through +1024.



example: *1.2* $= 0\ 01111111111\ 00110011001100...0011001100011_2 = 3FF3\ 3333\ 3333\ 3333_{16}$

example: *-10.6* $= 1\ 10000000010\ 01010011001100...0011001100011_2 = C025\ 3333\ 3333\ 3333_{16}$

REAL NUMBERS WITH SPECIAL MEANING

Certain bit patterns in both single and double precision floating point numbers have special meanings to the processor:

Infinity

Exponent value of all 1's, mantissa = 0. May be + or -. Infinity is represented by the following hexadecimal values:

Single Precision:
Positive = 7F80 0000
Negative = FF80 0000

Double Precision:
Positive = 7FF0 0000 0000 0000
Negative = FFF0 0000 0000 0000

Not a Number (NaN)

Exponent value of all 1's, mantissa <> 0. May be + or -.

Denormalized Number (DN)

Numbers smaller than exponent = 1. Exponent value of all 0's, mantissa <> 0. May be + or -

Zero

Exponent and mantissa all 0's. May be + or -

Special Numbers as Operands

If any of the special numbers described above are used in a floating point instruction, a special trap, called a before† trap, is taken (if enabled) and the processor returns a fixed result.

The combinations of special numbers that will invoke a before trap on input to an instruction and the resulting outputs are summarized in Table 4-1. Note that Not-a-Number (NaN) is treated as Infinity (INF). The results with INF as an operand are the same as those with NaN as an operand. Also note that a denormalized number (DN) is treated as zero. The results with a DN as an operand are the same as with zero.

Special Numbers as Results

Special numbers can also be the result of an overflow, underflow, or divide-by-zero operation when the respective trap for that operation is disabled.

- An overflow with the overflow trap disabled will produce infinity.
- An underflow with the underflow trap disabled will produce zero.
- Dividing a floating point number by zero will produce infinity, with the divide-by-zero trap disabled.

See Chapter 6, *TRAPS AND INTERRUPTS* for a detailed discussion on traps.

---

† "Before" in the sense that the trap occurs before the instruction is executed. See Chapter 6 for more information.

Table 4-1. Floating Point Results with Special Numbers

		(D) RADD		(D) RSUB		(D) RMPY		(D) RDIV	
Op1	Op2	Sgn	Rslt	Sgn	Rslt	Sgn	Rslt	Sgn	Rslt
N	Zero	S1	Zero	S1	Zero	XOR	Zero	Div-by-0	
N	DN	S1	N	S1	N	XOR	Zero	Div-by-0	
N	INF	S2	INF	-S2	INF	XOR	INF	XOR	Zero
N	NaN	S2	INF	-S2	INF	XOR	INF	XOR	Zero
Zero	N	S2	N	-S2	N	XOR	Zero	XOR	Zero
Zero	Zero	(4.)	Zero	(4.)	Zero	XOR	Zero	XOR	Zero
Zero	DN	(4.)	Zero	(4.)	Zero	XOR	Zero	Div-by-0	
Zero	INF	S2	INF	-S2	INF	XOR	Zero	XOR	Zero
Zero	NaN	S2	INF	-S2	INF	XOR	Zero	XOR	Zero
DN	N	S2	N	-S2	N	XOR	Zero	XOR	Zero
DN	Zero	(4.)	Zero	(4.)	Zero	XOR	Zero	Div-by-0	
DN	DN	(4.)	Zero	(4.)	Zero	XOR	Zero	Div-by-0	
DN	INF	S2	INF	-S2	INF	XOR	Zero	XOR	Zero
DN	NaN	S2	INF	-S2	INF	XOR	Zero	XOR	Zero
#INF	N	S1	INF	S1	INF	XOR	INF	XOR	INF
#INF	Zero	S1	INF	S1	INF	XOR	Zero	Div-by-0	
#INF	DN	S1	INF	S1	INF	XOR	Zero	Div-by-0	
#INF	INF	+	INF	+	INF	XOR	INF	XOR	Zero
#INF	NaN	+	INF	+	INF	XOR	INF	XOR	Zero

- NaN here is treated as infinity

	(D) FIXT (D) FIXR		MAKERD MAKEDR	
Op2	Sgn	Result	Sgn	Rslt
DN	S2	Zero	S2	Zero
INF	S2	largest integer	S2	INF
NaN	S2	largest integer	S2	INF

Abbreviations:

Op1 = Operand 1
Op2 = Operand 2
Sgn = Sign of result
Rslt = Result

N = Number
DN = Denormalized No.
INF = Infinity
NaN = Not-a-Number

S1 = Sign Operand 1
S2 = Sign Operand 2
XOR = S1 XOR S2

Notes:

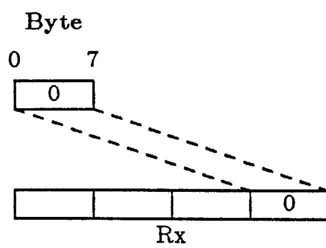
- Div-by-zero is treated as a real divide by zero trap.
- The (D)RNEG and (D)FLOAT instructions have no "before" traps.

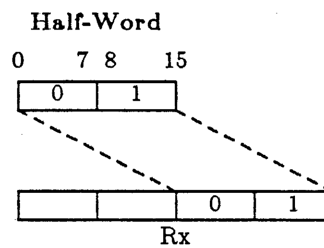
REAL NUMBER ROUNDING RULES

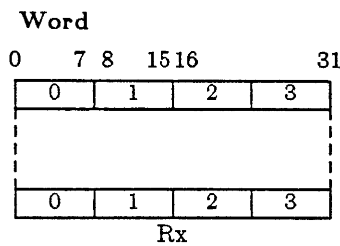
The processor will round up when a floating point value has guard bits‡ greater than .5 and down when less than .5. However, when the guard bits are *exactly* will round up if this bit is "1" and down if "0".

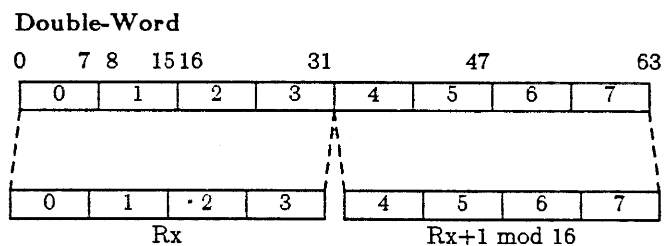
DATA STORAGE IN REGISTERS

The illustrations below give the notation and memory layout for each type of operand, as well as how that operand is stored in the general registers.









‡ Guard bits are intermediate values to the right of the mantissa that are held internally in the processor during computation.

Chapter 5
PROCESS MANAGEMENT

INTRODUCTION

Processes are created and controlled by a coordination of the Ridge hardware and operating system. This section discusses processes management from the aspect of the Ridge hardware. For information on process management from the perspective of the Ridge Operating System, refer to the *Ridge Internals Manual.*

CODE AND DATA SEGMENTS

Each process in the system is assigned its own unique address space for its code and another address space for its data. These address spaces are referred to as the process's Code Segment and Data Segment.

Segments are addressed in 4096-byte pages and can extend up to 4G-bytes of virtual memory. The system can address up to 65536 segments.

The Data Segment for a process is not created until execution time.

PROCESSOR MODES

The processor can operate in one of two modes:

- Kernel Mode
- User Mode

KERNEL MODE

In kernel mode, all memory references go straight to real memory, bypassing the virtual-to-real address translation hardware. External interrupts are disabled in kernel mode.

The kernel mode is entered at one of 256 entry points by means of the KCALL instruction, or by means of an interrupt or trap.

With the exception of the KCALL instruction, all instructions in the Ridge instruction set can be executed while in kernel mode.

USER MODE

In user mode, all memory addresses are virtual, and memory references are made to code space or data space. All memory addresses first pass through the processor's virtual to real translation hardware before going to real memory.

Only non-kernel mode instructions can be executed in user mode.

Privileged Process Bit

It is sometimes convenient to execute certain kernel maintenance instructions, such as READ or WRITE, while in user mode and without losing the benefit of virtual addressing and interrupts. This is possible when the *privileged process bit,* which is bit 31 in a special register called *SR10,* is set. (This bit is actually *set* in a data structure called the *Process Control Block,* which is described later in this chapter.)

Kernel mode instructions, as a rule, can only be executed when the processor is in kernel mode. Some kernel mode instructions, however, can be executed in user mode when the privileged process bit is set. When a process attempts to execute a kernel instruction, the microcode checks to see if the processor is executing a user mode or kernel mode process. If not in kernel mode, bit 31 in SR10 is checked to see if the process has privileged permission. If the process fails both the kernel mode and privileged process checks, the kernel violation trap is taken. (Special Registers are described in the next section. Traps are explained in Chapter 6.)

PROCESSOR CONTROL

The internal functions of the processor are controlled by a set of 16 special registers and the *Process Control Block* (PCB).

SPECIAL REGISTERS

The processor maintains 16 special purpose registers to control the processor and to pass information between the kernel and the hardware. Special registers can only be accessed by kernel mode instructions.

Register	Contents
SR0	Kernel flag: user mode = 1, kernel mode = PC or IOIR word
SR1	Opcode / KCALL number / fault type
SR2	segment number / Rx literal field
SR3	virtual address / Ry literal field
SR4	Real addr of link block during send type KCALLs (ROS only)
SR5	Unused
SR6	Unused
SR7	Unused
SR8	Code segment number
SR9	Data segment number
SR10	Traps word
SR11	Address of CPU Control Block (CCB)
SR12	Address of Virtual to Real Translation table (VRT)
SR13	VRMASK Used to perform modulo for VRT main table lookup
SR14	Address of current Process Control Block
SR15	User PC

SR0 Kernel flag register. This register is set on entry to a trap or interrupt. When SR0 = 1, the trap occurred in user mode. If the trap or interrupt occurred in kernel mode, SR0 will contain the value of the kernel program counter (PC) at the time of the event. The only exception to this rule occurs when an external interrupt takes place and SR0 is loaded with the *I/O Interrupt Read* (IOIR) word generated by the interrupt routine. The most significant byte of the IOIR word contains the device number of the device that generated the interrupt.

SR1 Opcode. When an illegal instruction is encountered or a kernel violation has occurred, the opcode of the current instruction is placed in SR1. On a page fault, -1 is placed in SR1 if the page fault occurred because the page was absent from memory, or SR1 = -2 if the fault occurred because the page was write protected.

SR2 Rx field/segment number. When traps occur, this register will either contain the current instruction's segment number or the value in its Rx field. See Table 6-1 in Chapter 6 for details.

SR3 Ry field/effective address. When certain traps occur, the Ry field from the instruction is placed in SR3. On some interrupts or other traps, SR3 contains the effective address where the event occurred.

SR4 Contains the real address of the link block during send-type KCALLs. This special register is maintained by ROS.

SR5 - SR7 unused.

SR8 Contains the code segment number of the current process. If no process is active, it contains the code segment number of the last process run.

SR9 Contains the data segment number of the current process. If no process is active, it contains the data segment number of the last process run.

SR10 Traps word for the current process. The traps word consists of 32 bits, which can be set to enable or disable various traps (see *Traps Word* in the *TRAPS* Section in Chapter 6). The traps word is set in the Process Control Block and loaded into SR10 by means of the LUS instruction. Note that traps can only be enabled or disabled by loading the traps word into SR10 via the LUS instruction; it is insufficient to use the MOVE instruction to change SR10.

SR11 CPU Control Block (CCB) address. This is the real memory address of the CCB, which is a data structure that points to the trap vectors, interrupt vectors, and maintains the processor time-keeping facilities (see Chapter 6). If the value in this special register is odd, no CCB is present.

SR12 Virtual to Real Translation (VRT) table address. This points to the base of the Virtual to Real Translation tables, which is used by the microcode to find the mapping records for any virtual page which is resident in main memory.

SR13 VRT mask. This is used by the microcode to compute the hash function for VRT lookups.

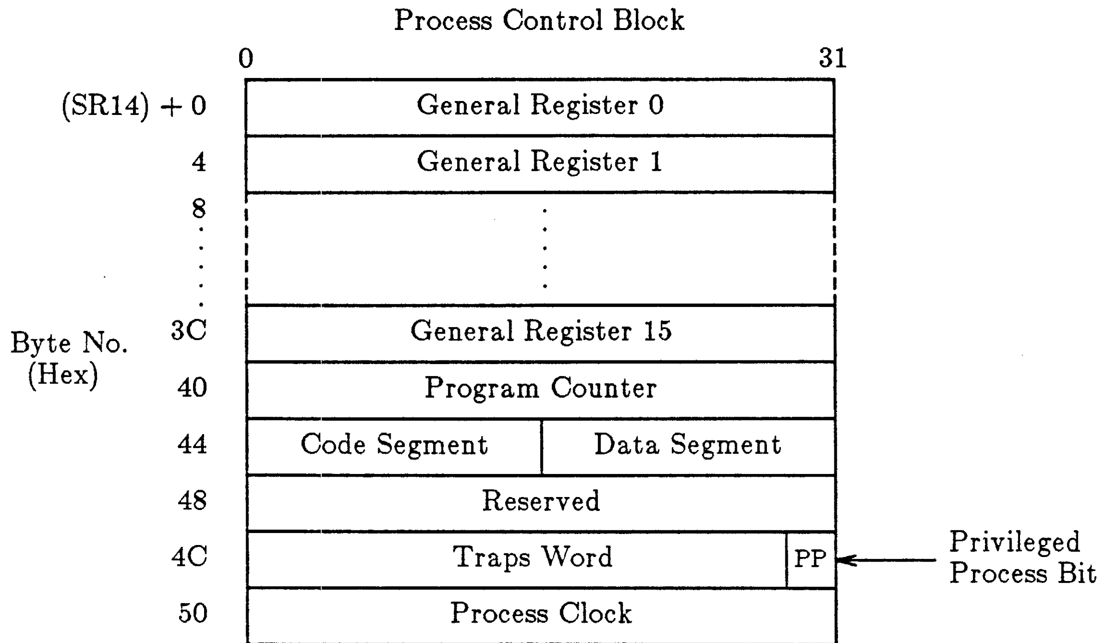
SR14 Current Process Control Block (PCB) address. Points to the current PCB. It is used extensively by both microcode and the kernel. The microcode uses this pointer in the **LUS**, **SUS**, and **LDREGS** instructions. SR14 is also used to do current process time accounting. If SR14 contains a 1, then there is no current process.

SR15 Program Counter (PC). This register is loaded with the current PC value when a KCALL instruction is executed, or upon a trap or interrupt in user mode.

PROCESS CONTROL BLOCK

The kernel maintains the state of running processes by means of the Process Control Block (PCB). The current PCB address is located in SR14.

This section only describes those portions of the Process Control Block maintained by microcode. See the *Ridge Internals Guide* for a description of the PCB sections maintained by the ROS kernel.



General Registers 0 through 15, and the Program Counter register are saved / loaded at context switch time. The *Code/Data* Segment register consists of two 16-bit segment numbers. The *Traps Word* register contains the bits that are used to enable or disable specific traps. Bit 31 of the Traps Word register is the privileged process bit which, when enabled, allows some kernel mode instructions to be executed while the processor is in user mode. The *Process Clock* register is incremented if the timer ticks while the process is running.

The PCB registers must be loaded into the special registers before they are used by the processor. When a Load User State (LUS) instruction is executed, the Processor Control Block registers are loaded into the special registers as follows:

Program Counter → SR15
Code Segment → SR8
Data Segment → SR9
Traps Word → SR10

The process clock is maintained in SR14 + 50 (Hex).

Chapter 6
TRAPS AND INTERRUPTS

INTRODUCTION

The normal flow of a process is sometimes altered as a result of some external event. Events are defined as follows:

- A *page fault* occurs when the page specified by the virtual address has no corresponding page in memory. Page faults are discussed in Chapter 7.

- If an event originates in a source external to the running process, the condition is termed an *interrupt.* In this situation, the current instruction in the user process completes and the PC value for the next instruction is placed in SR15.

- If the event originates in the current instruction, the condition is termed a *trap.* When a trap occurs, the current instruction is aborted and the current PC value is placed in SR15.

EVENT HANDLING

When an event occurs, the following sequence takes place:

1. If the event is an interrupt, it will not be recognized until the current instruction has completed. If the event is a trap, the current instruction is aborted.

2. If the event occurs while in user mode, the contents of the program counter, which points to the next user mode instruction (in the case of an interrupt) or to the aborted instruction (in the case of an trap), are placed in SR15 and SR0 is set to 1. On external interrupts, the contents of the I/O Interrupt Word (IOIR) are placed in SR0.

   If the event occurs while in kernel mode, SR15 is left unchanged and the contents of the program counter are placed in SR0. (This allows traps such as double bit parity error to occur in both user and kernel mode.)

3. Special registers SR1, SR2, and SR3 are set as prescribed by the microcode. The processor then switches to kernel mode and, by means of the *CPU Control Block,* executes the *kernel function* associated with the event. (See the *CPU CONTROL BLOCK* section below for details on how the kernel function is selected.)

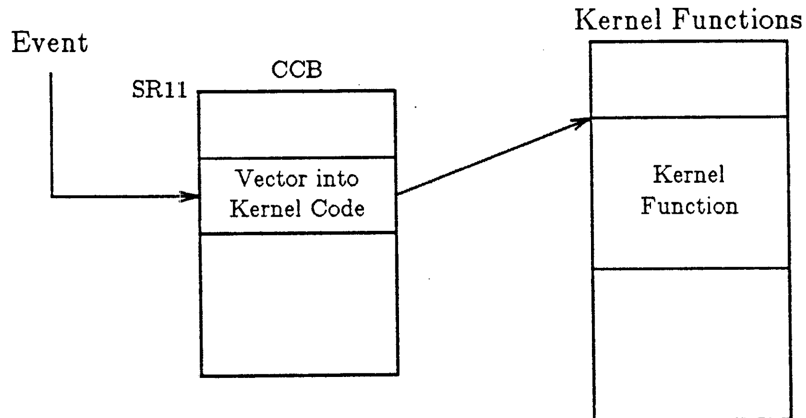
In the course of a kernel function, the LUS (Load User State) and SUS (Save User State) instructions might be used to execute another user mode process, or to save the state information of the process halted by the event. (LUS and SUS use the value in SR14 as the pointer to the Process Control Block.)

After handling the event, the kernel function may return to executing the process that was halted by the event. If the event occurred while executing in user mode, the kernel function will execute the RUM (Resume User Mode). This instruction causes the processor to load the PC with the value found in SR15 and switch to user mode. If the event occurred while executing in kernel mode, the kernel function will execute the TRAPEXIT instruction, which will load the PC with the value found in SR0 and resume program execution in kernel mode.

CPU CONTROL BLOCK

The CPU Control Block (CCB) is located at the real memory address specified by SR11. The CCB contains a vector to each possible kernel function. When an event occurs, the microcode locates the vector to the appropriate kernel function in the CCB and begins executing the kernel function at that address in real memory.

This process can be illustrated as follows:



The CCB entry for an event is located by adding SR11 to the event's offset. Table 6-1 below lists the CCB and the offset accessed for each event. The states of Special Registers SR0-SR3 and SR15 at entry to the kernel are also given.

Table 6-1. CPU Control Block and Special Register Values

Offset	Event	SR0	SR1	SR2	SR3	SR15
0	KCALL 0	nc	nc	nc	nc	PC+2
4	KCALL 1	nc	nc	nc	nc	PC+2
8	KCALL 2	nc	nc	nc	nc	PC+2
.	. .	.	.	.	.	.
.	. .	.	.	.	.	.
3FC	KCALL 255	nc	nc	nc	nc	PC+2
400	Data Alignment	see †	nc	Segment #	Virtual or Real Address	see †
404	Illegal Instructions:					
	Memory Instruction	see †	Opcode	Segment #	Virtual Address	see †
	Reference Instruction	see †	Opcode	Rx Field	Ry Field	see †
408	Double Bit Parity Error - Fetch	see ‡	nc	nc	Virtual or Real Address	see ‡
40C	Double Bit Parity Error - Execute	see ‡	nc	nc	Virtual or Real Address	see ‡
410	Page Fault	1	see *	Segment #	Virtual Address	PC
414	Kernel Violation	see †	Opcode	Rx Field	Ry Field	see †
414	Check Trap	see †	Opcode	Rx Field	Ry Field	see †
41C	Arithmetic Traps:					
	Trap Instruction	see †	Opcode	Rx/Ry Fields	Ry Field	see †
	Integer Overflow	1	Opcode	Rx/Ry Fields	16	PC
	Integer Divide by 0	1	Opcode	Rx/Ry Fields	17	PC
	Real Overflow	1	Opcode	Rx/Ry Fields	18	PC
	Real Underflow	1	Opcode	Rx/Ry Fields	19	PC
	Real Divide by 0	1	Opcode	Rx/Ry Fields	20	PC
	Inexact Result	1	Opcode	Rx/Ry Fields	24	PC
	Before	1	Opcode	Rx/Ry Fields	25	PC
420	External Interrupt	Device # word	nc	nc	nc	PC+*il*
424	Switch 0 Interrupt	see ‡	nc	nc	nc	see ‡
428	Power Fail Warning	see ‡	nc	nc	nc	see ‡
430	Timer 1 Interrupt	1	nc	nc	nc	PC+*il*
434	Timer 2 Interrupt	1	nc	nc	nc	PC+*il*
438	Reserved					
	Data Area:					
43C	Idle Count					
440	Timer 1 Count					
444	Timer 2 Count					
448	Time of Day in ns					
.	. .					
.	. .					
44C						

Notes: PC+*il* The value of PC incremented by the length of the current instruction.

nc The event has not changed the value in the register.

† If executing instruction in user mode, SR0 = 1 and SR15 = PC.
If executing in kernel mode, SR0 = PC; SR15 is unchanged.

‡ If executing instruction in user mode, SR0 = 1 and SR15 = PC + length of current instruction.
If executing in kernel mode, SR0 = PC + length of current instruction; SR15 is unchanged.

* If the page fault occurs because the page is absent from memory, then SR1 = -1.
If the page fault occurs because the page was write protected, then SR1 = -2.

CCB DATA AREA

The portion of the CCB pointed to by offsets 43C through 44C contains the timekeeping facilities used by the processor. These are:

- Idle Count
- Timer 1 Count
- Timer 2 Count
- Time of Day in Nanoseconds

Process time is incremented once each millisecond. When SR1 <> 1, a user process is running, and the *process clock* word in the PCB is incremented (see Chapter 5). If no process is running (SR14 = 1), the *idle count* in the CCB data area is incremented.

The *timer 1 count* timer is used by the ROS kernel to keep track of the time-slice intervals of the running process. The *timer 2 count* counter is typically used to keep track of sleeping processes and for other non-slicing purposes. These timers are decremented once each millisecond. If a timer counter goes negative while executing a user process, a timer interrupt will occur. Timer interrupts that occur during kernel mode are ignored. (Timer interrupts will be discussed later in this chapter.)

The *time of day in nanoseconds* double word keeps track of the time of day. Each millisecond, this counter is incremented by one million nanoseconds.

PAGE FAULT

A page fault occurs when the page specified in the virtual address has no corresponding physical page in memory. This fault can never occur in kernel mode, so SR0 will always be set to 1 and SR15 will be set to the current PC.

See the *TRANSLATION SEQUENCE* Section in Chapter 7 for details.

INTERRUPTS

The following events are termed interrupts:

- Double-Bit Parity Error
- External Interrupt
- Switch 0
- Power Fail Warning
- Timer Interrupts
- Reset

DOUBLE-BIT PARITY ERROR

A double-bit parity error can occur in either the instruction fetch or instruction execute stage. The interrupts for both events are discussed below.

Double-Bit Parity Error on Instruction Fetch

A code fetch error may occur when the Instruction Fetch unit reads a word of code from memory. The address of the parity error is determined as follows:

User Mode: SR8 contains the code segment number, and SR3 contains the virtual address.

Kernel Mode: SR3 contains the real memory address.

Double-Bit Parity Error on Instruction Execute

An execute error can occur as the CPU executes a memory reference instruction. The address of the parity error is determined as follows:

User Mode: SR8 contains the segment number. SR15 contains the PC value when the memory reference instruction failed. The opcode of the instruction determines whether it is a reference to a code or data segment. SR8 contains the code segment number, and SR9 contains the data segment number. The virtual address is in SR3.

Kernel Mode: SR0 contains value of the PC when the memory reference instruction failed. The real address of the parity error is in SR3.

EXTERNAL INTERRUPTS

An external interrupt is an interrupt caused by a peripheral device. When this interrupt occurs, the kernel places the *I/O Interrupt Read* word (IOIR) into SR0. The most significant byte of the IOIR contains the device number of the device that generated the interrupt; the remainder of this word varies, depending on the device. Interrupts that occur in kernel mode are suspended until the processor returns to user mode.

SWITCH 0

This interrupt occurs when switch 0 on the clock board is pressed and released. The switch 0 interrupt stops all currently running processes and traps through CCB offset 424, which typically enters the RBUG debugger.

POWER FAIL WARNING

The power fail warning interrupt is caused when the power supply detects that AC power is being removed. When this occurs, the clock board sets bit 23 in the status word. The kernel then loops while waiting for status bit 23 to go to zero. If the power fails completely, the processor dies while looping. However, if the power only glitches, the clock board will eventually drop status bit 23 and the kernel will resume from the point of interruption.

The status bits are described with the *ELOGR* instruction in the *MAINTENANCE INSTRUCTIONS* section in Chapter 9.

TIMER INTERRUPTS

There are two types of timer interrupts:

- Timer 1
- Timer 2

Every millisecond the timer 1 count and then the timer 2 count in the CCB data area are decremented. The timer 1 or timer 2 interrupt will occur when its respective interval timer contains a value of less than zero.

If both the timer 1 and timer 2 counter become less than zero at the same time, the timer 1 interrupt will occur first. The timer 2 interrupt will be handled when the timer 1 interrupt has completed and the processor returns to user mode.

RESET

Reset is a hard-wired interrupt that occurs when the *reset* button on the clock board is pressed. Pressing reset causes the processor to enter kernel mode without going through the CCB. Once in kernel mode the PC is set to 3E000 (which contains the first page of data from the boot media), the internal state of the CPU is cleared, and the special registers are set to the following values:

SR2 =Memory size
SR11 =1 (no CCB)
SR14 =1 (no PCB)

TRAPS

The following events are termed traps:

- Kernel Calls
- Data Alignment Violation
- Illegal Instruction Execution
- Kernel Violation
- Check Instruction Trap
- Trap Instruction
- Arithmetic Traps

KERNEL CALLS

The kernel call trap is invoked by a KCALL instruction from the user process to perform some kernel operation during the normal course of execution.

This trap first sets the processor to kernel mode. The 8 bits following the KCALL opcode are multiplied by 4. This value is added to SR11 to locate the entry in the CCB containing the address of the kernel entry point. Execution then begins at this address.

An attempt to execute a KCALL instruction while in kernel mode results in a kernel violation trap.

DATA ALIGNMENT TRAP

LOAD and STORE instructions operating on half-word, word, and double-word operands require that their addresses be evenly divisible by their lengths in bytes. Otherwise, the data alignment trap is taken.

The following instructions can cause a data alignment trap:

LOAD	LOADP	STORE
LOADD	LOADDP	STORED
LOADH	LOADHP	STOREH

ILLEGAL INSTRUCTION TRAP

An illegal instruction trap occurs when an attempt is made to execute an undefined instruction. Upon an illegal instruction trap, the illegal opcode will be placed in SR1. The contents of SR2 are dependent upon the format of the illegal instruction (determined by its opcode group). If the instruction is a register format instruction (opcode is less than 80 hex), then SR2 will contain the value (i.e., four bits) of the Rx field and SR3 will contain the value of the Ry field. If the instruction is a memory format instruction (opcode is equal to or greater than 80 hex), SR2 will contain the segment number and SR3 will contain the virtual address.

KERNEL VIOLATION TRAP

If a user attempts to execute a kernel instruction, a kernel violation trap is taken. Some kernel instructions, such as READ and WRITE, can be executed by a process when the *privileged process* bit is set in the traps word (see *TRAPS WORD* later in this chapter). When an instruction causes a kernel violation trap, Rx remains unmodified.

The following instructions can cause a kernel violation trap:

CREGIN	ITEST	MOVESR	TRAPEXIT
CREGOUT	KCALL (in Kernel Mode)	READ	TWRITED
DIRT	LDREGS	RUM	VERSION
ELOGR	LUS	SUS	WRITE
ELOGW	MACHINEID	TRANS	
FLUSH	MOVERS	TRAP	

CHECK TRAP

The check trap is taken when a CHK instruction is executed under the following conditions:

Instruction	Trap Condition	Registers Modified (trap on or off)
CHK	Rx > Ry	None
CHKI	Rx < 0 or Rx > Ry field	None

TRAP INSTRUCTION TRAP

There are 16 bits in the *traps word* which can be individually enabled or disabled in order to set breakpoints while using the rbug, debug, or dbx debuggers. The specific bits to be examined in the traps word are specified by the value in the Ry field of the TRAP instruction (described in Chapter 8). A trap will occur if the value specified by the Ry field of the TRAP instruction indicates a bit among the most significant 16 bits (0..15) in the traps word. If the bit specified by the TRAP instruction is not set in the traps word, no trap will be taken. (The traps word is discussed in the *Traps Word* section.)

The TRAP instruction always traps in kernel mode.

ARITHMETIC TRAPS

Arithmetic traps result when an instruction attempts an illegal arithmetic operation.

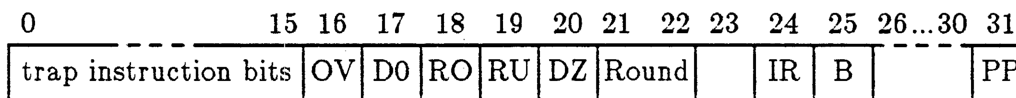
The following are arithmetic traps:

- Integer Overflow
- Integer Divide by zero
- Real Overflow
- Real Underflow
- Real Divide by zero
- IEEE Inexact Result
- IEEE "Before" Trap

Arithmetic traps can be enabled or disabled by setting or clearing bits in the *traps word.*

Traps Word

The traps word for the current process is contained in SR10. The traps word also exists as an entry in the Processor Control Block, where its bits can be enabled or disabled by means of kernel services. This modified traps word can then be loaded into SR10 by means of a LUS instruction. The format of the traps word is as follows:



The 16 most significant bits of the traps word are used by the TRAP instruction to selectively trap while executing a program. (See the *Trap Instruction Trap* section.)

- OV enables integer overflow trap
- D0 enables integer divide-by-zero trap
- RO enables real overflow trap
- RU enables real underflow trap
- DZ enables real divide-by-zero trap
- Round sets the IEEE rounding modes:
  - 00 - round to nearest
  - 01 - round to plus infinity
  - 10 - round to minus infinity
  - 11 - round to zero
- IR enables the inexact result trap
- B enables "before" traps (must be 0 under ROS)
- PP enables privileged access. See *User Mode* in Chapter 5.

Integer Overflow Trap

The overflow trap can generate a variety of results. In all situations, the process will be suspended if the integer overflow trap is enabled, or will continue executing if the trap is disabled.

If, when executing an ADD, MPY or SUB instruction, an integer is too large for the Rx, the least significant 31 bits of the result will be delivered.

If an overflow occurs during a DIV or EDIV instruction, the dividend will not be modified and no result will be delivered.

If an overflow occurs during a REM instruction, a result of 0 is delivered.

If an overflow occurs during a ASL instruction, the result is placed in Rx.

Executing the NEG instruction with a value of -2**31 will cause an overflow. If this occurs, -2**31 is delivered.

The table below illustrates the instructions and conditions that can cause an integer overflow trap:

Instruction	Trap Condition	Registers Modified
ADD	Result > 31 bits	Rx ← least sig. 31 bits
ASL	any shifted bits ≠ Old Rx[0]	Rx ← result
DIV	Rx=8000 0000 Ry=FFFF FFFF	None
EDIV	Result > 31 bits	None
MPY	Result > 31 bits	Rx ← least sig. 31 bits
NEG	-2**31	Rx ← -2**31
REM	Rx=8000 0000 Ry=FFFF FFFF	Rx ← 0
SUB	Result > 31 bits	Rx ← least sig. 31 bits

Integer Divide By Zero Trap

An attempt to divide an integer by zero will leave Rx unmodified and the divide-by-zero trap will be taken, if enabled.

The table below illustrates the instructions and conditions that can cause an integer divide-by-0 trap:

Instruction	Trap Condition	Registers Modified
DIV	Ry = 0	None
EDIV	Ry = 0	None
REM	Ry = 0	None

Before Trap

Two *types* of traps can occur when executing real numbers. These trap types are referred to as *before* and *after* traps. A *before* trap can occur as a result of some condition before the instruction is executed; *after* traps can occur after the result has been generated and put in the Rx register. The *before* trap is described in this section. *After* traps consist of the real number traps described in the following sections.

Before a floating point instruction is executed, its operands are checked for special numbers (special numbers are described in the *REAL NUMBERS WITH SPECIAL MEANING* section in Chapter 4). If either operand is a special number, the *before* is taken (if enabled) and the value in Rx is not modified.

If the *before* trap is disabled, the result is determined as shown in Table 4-1 in Chapter 4 and processing continues. Note that with the *before* trap disabled, a denormalized number is treated as zero and not-a-number is treated as infinity. This trap is disabled by the Ridge Operation System (ROS).

The following instructions can cause a *before* trap:

DFIXR	DRDIV	FIXT	RCOMP
DFIXT	DRMPY	MAKEDR	RDIV
DRADD	DRSUB	MAKERD	RMPY
DRCOMP	FIXR	RADD	RSUB

Real Divide by Zero Trap

An attempt to divide a real number by zero or a denormalized number causes a real divide by zero trap (if enabled), which will suspend the process and leave Rx or RPx unmodified. If the divide by zero trap is disabled, infinity is returned as a result and the process will continue executing.

This trap can occur when executing the DRDIV or RDIV instructions.

Real Underflow Trap

When a floating point operation underflows, the generated result will be delivered and the underflow trap (if enabled) will suspend the process. If the underflow trap is disabled, zero will be returned as the result and the process will continue executing.

The following instructions and conditions can cause a real underflow trap:

Instruction	Trap Condition	Registers Modified (trap bit on)	Registers Modified (trap bit off)
DRADD	Result < min real	RPx ← result	RPx ← 0
DRDIV	Result < min real	RPx ← result	RPx ← 0
DRMPY	Result < min real	RPx ← result	RPx ← 0
DRSUB	Result < min real	RPx ← result	RPx ← 0
MAKEDR	Result < min real	Rx ← +/- 0	Rx ← +/- 0
RADD	Result < min real	Rx ← result	Rx ← 0
RDIV	Result < min real	Rx ← result	Rx ← 0
RMPY	Result < min real	Rx ← result	Rx ← 0
RSUB	Result < min real	Rx ← result	Rx ← 0

Real Overflow Trap

When a floating point number is too large for the destination register(s), the generated result will be delivered and the overflow trap (if enabled) will suspend the process. If the overflow trap is disabled, infinity will be returned as the result and the process will continue executing.

The only instructions not subject to this rule are the MAKEDR, (D)FIXR and (D)FIXT instructions, which will return a fixed result with the overflow trap on or off.

The following instructions and conditions can cause a real overflow trap:

Instruction	Trap Condition	Registers Modified (trap bit on)	Registers Modified (trap bit off)
DFIXR	Result > integer	Rx ← 7FFF FFFF (if positive) Rx ← 8000 0000 (if negative)	same
DFIXT	Result > integer	Rx ← 7FFF FFFF (if positive) Rx ← 8000 0000 (if negative)	same
DRADD	Result > max real	RPx ← result	RPx ← +/- infinity
DRDIV	Result > max real	RPx ← result	RPx ← +/- infinity
DRMPY	Result > max real	RPx ← result	RPx ← +/- infinity
DRSUB	Result > max real	RPx ← result	RPx ← +/- infinity
FIXR	Result > integer	Rx ← 7FFF FFFF (if positive) Rx ← 8000 0000 (if negative)	same
FIXT	Result > integer	Rx ← 7FFF FFFF (if positive) Rx ← 8000 0000 (if negative)	same
MAKEDR	Result > single precision	Rx ← +/- infinity	same
RADD	Result > max real	Rx ← result	Rx ← +/- infinity
RDIV	Result > max real	Rx ← result	Rx ← +/- infinity
RMPY	Result > max real	Rx ← result	Rx ← +/- infinity
RSUB	Result > max real	Rx ← result	Rx ← +/- infinity

Inexact Real Results Trap

If a floating point result has been rounded, the inexact trap is taken if enabled. (See the *REAL NUMBER ROUNDING RULES* section in Chapter 4.)

The following instructions can cause an inexact result trap:

FIXR	DRADD	DRSUB	RDIV
FLOAT	DRDIV	MAKEDR	RMPY
DFIXR	DRMPY	RADD	RSUB

RECOVERY FROM REAL OVERFLOW, UNDERFLOW, AND INEXACT RESULT TRAPS

When a floating point number overflows or underflows, the value delivered in the result register has been truncated by 1 bit. Since the largest result from any 31-bit computation is a 32-bit value (sign bit excluded), you can determine the correct result by looking at the value in the result register and logically inserting a bit value between the sign bit and the most significant bit of the exponent. On overflow, insert a "1" if the most significant bit (MSB) in the exponent is "0", or a "0" if the MSB is "1". On underflow, insert a "1" if the MSB for the exponent is "1", or a "0" if the MSB is "0".

The following examples illustrate these operations:

If you multiply 7F000000 by 40800000 an overflow will occur and the delivered result will be 00000000. Since the MSB of the exponent is "0", adding a "1" between the sign bit and the most significant bit of the exponent will produce 80000000 (positive), which is the correct value.

If you divide 00800000 by 41000000 an underflow will occur and the delivered result will be 7F000000. Since the MSB of the exponent is "1", adding a "1" between the sign bit and the most significant bit of the exponent will produce the correct value, FF000000 (positive).

Chapter 7
VIRTUAL MEMORY MANAGEMENT

INTRODUCTION

User mode processes operate on virtual memory addresses. The Ridge processor has a unique virtual memory design made up of instructions, microcode and hardware. This section describes the virtual memory system used by the processor.

VIRTUAL ADDRESS

Virtual addresses are generated when the processor is in user mode. Virtual addressing is accomplished by means of a 16-bit segment number and a 32-bit virtual address.

The format of the Ridge segment number and virtual address appears below:

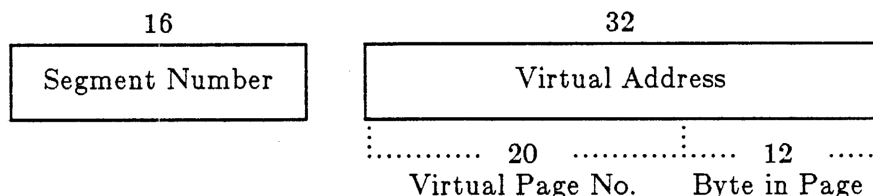


Figure 7-1. Segment Number and Virtual Address Formats

The Segment Number identifies the code or data segment for the process. The maximum number of addressable segments is 65536, and each segment is addressable up to 4G-bytes.

The 32-bit Virtual Address consists of a 20-bit virtual page number (VPN) and a 12-bit byte offset that identifies the address's location within the page.

The 32-bit virtual address is generated in the instruction stream (or, "contained" within the instruction), while the segment number is located in a special register. SR8 contains the code segment number, and SR9 contains the data segment number.

VIRTUAL TO REAL TRANSLATION HARDWARE

The virtual address is sent to the memory controller, where the 20-bit virtual page number and a code/data reference bit are translated to a 15-bit physical page number by a 1024-entry cache of virtual address/real address pairs called the Translation Mapping Table (TMT). The physical address is completed by joining the physical page generated by the TMT with the virtual address's 12-bit page offset.

The Ridge 3200 virtual memory hardware can be illustrated as follows:

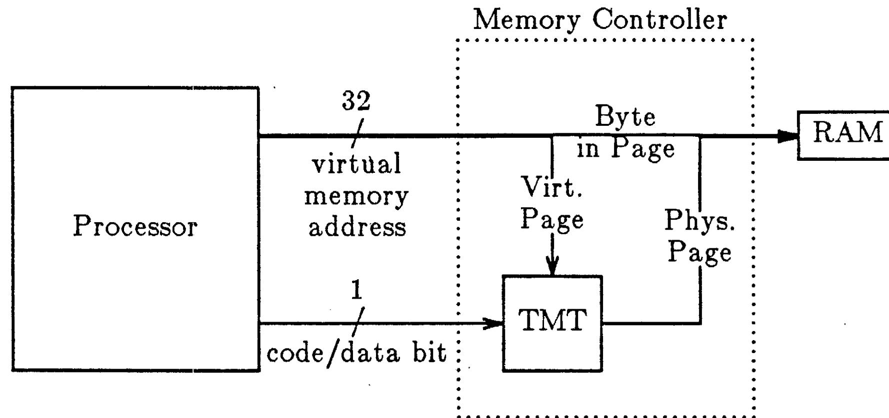


Figure 7-2. Ridge Virtual Memory Hardware

The TMT is controlled by microcode and does not require software intervention. This translation method results in virtual memory overhead that is roughly equivalent to the cpu cycles lost to memory refresh for the MOS main memory parts, which is approximately one percent.

TRANSLATION TABLES

The Ridge processor uses two mechanisms to translate virtual addresses to real addresses:

- Virtual to Real Translation (VRT) Table
- Translation Mapping Table (TMT)

Ridge's virtual memory translation tables are the inverse of traditional page tables. Instead of creating a page table whose number of entries is based on the virtual memory size, Ridge creates a table whose number of entries is based on the size of main memory. This table is called the Virtual to Real Translation (VRT) table.

The VRT is a microcode-interpreted table containing 48-bit virtual addresses that are associated with specific pages of main memory. A VRT entry exists for each page currently in main memory.

The TMT is a hardware cache for the most active VRT memory references, allowing virtual (user mode) addressing to be accomplished in the same cycle time as real (kernel mode) addressing. The TMT is bypassed for real, or non-virtual memory operations and is invalidated by the LUS instruction.

TRANSLATION SEQUENCE

The VRT table is accessed when the address cannot be found in the TMT. When a virtual address is sent from the processor to the memory controller, the TMT is searched to determine the real address. If no entry is found in the TMT, a page fault occurs. The microcode then searches the VRT. If the VRT contains an entry for the page, it is loaded into the TMT and the processor begins the operation over again. If the VRT does not contain an entry for the page, the processor aborts the current instruction and a page fault trap is taken.

See the *ROS Internals Manual* for more information on VRT page faults.

Ridge virtual to real translation can be illustrated as follows:

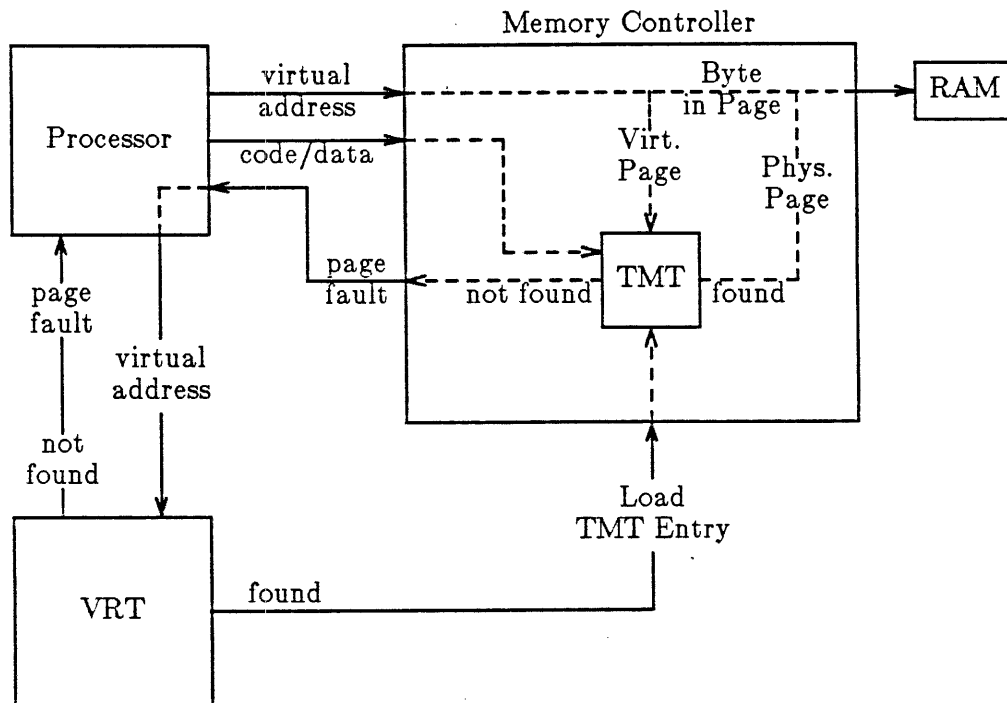


Figure 7-3. Virtual-to-Real Translation Sequence

VRT TABLE ORGANIZATION

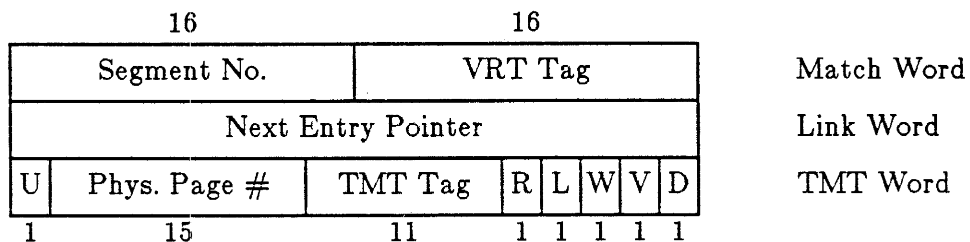
The VRT is organized in main memory as a hash table of pointers which point to VRT entries that identify the pages currently in memory. The processor uses microcode to efficiently search the VRT without software intervention.

The VRT hash table's address is stored in SR12 (see the *SPECIAL REGISTERS* section in Chapter 5). The size of the VRT hash table is stored in special register SR13. Under ROS, the hash table size is determined by the size of main memory. The formula for determining this is:

Hash Table Size = # pages of memory^2

VRT Entry Format

The format of a VRT entry is as follows:



Segment No.	Code or data segment number.
VRT Tag	High order 16-bits (0:15) of the virtual address.
Next Entry Pointer	Pointer to next VRT entry.
U	Unused. Must be 0.
Phys. Page	Real page in memory associated with the virtual address.
TMT Tag	High order 11-bits (0:10) of the virtual address.
R	Referenced Bit.
L	Locked Bit (Used only by Kernel).
W	Write Allowed Bit (1 = write allowed).
V	Valid Bit (Must = 1).
D	Dirty Bit (1 = dirty).

VRT Translation Process

When the processor needs to search the VRT, it proceeds as follows:

The virtual address is run through a hash function which adds the current segment number (contained in SR8 or SR9) to the virtual page number (bits 0:19 of the virtual page offset). This sum is ANDed with the contents of VRMASK (located in SR13). The result is shifted left 2 bits and added to the VRT table base address (located in SR12).

The 32-bit value produced by the hash function points to an address in the *VRT hash table* which, in turn, points to a VRT entry in memory. (If a zero is located at the VRT hash table address, a page fault is generated.)

Several segment number / virtual address pairs may generate the same hash value. Because of this, VRT entries sharing the same hash function value are chained together into a linked list. If there is no match between the segment number / virtual address value and the first VRT entry's *Match Word*, the *Link Word* in that entry is used to locate the address of the next VRT entry sharing the same hash value.

The chain of VRT entries will be followed until either a matching VRT entry is found, or the end of the chain is encountered. If a match is made between a VRT entry and the virtual address, the VRT entry's *TMT Word* is loaded into the TMT and the VRT entry's referenced bit is set. If the end of the linked list is reached without locating a matching VRT entry, a page fault is generated.

A successful search through the VRT can be illustrated as follows:

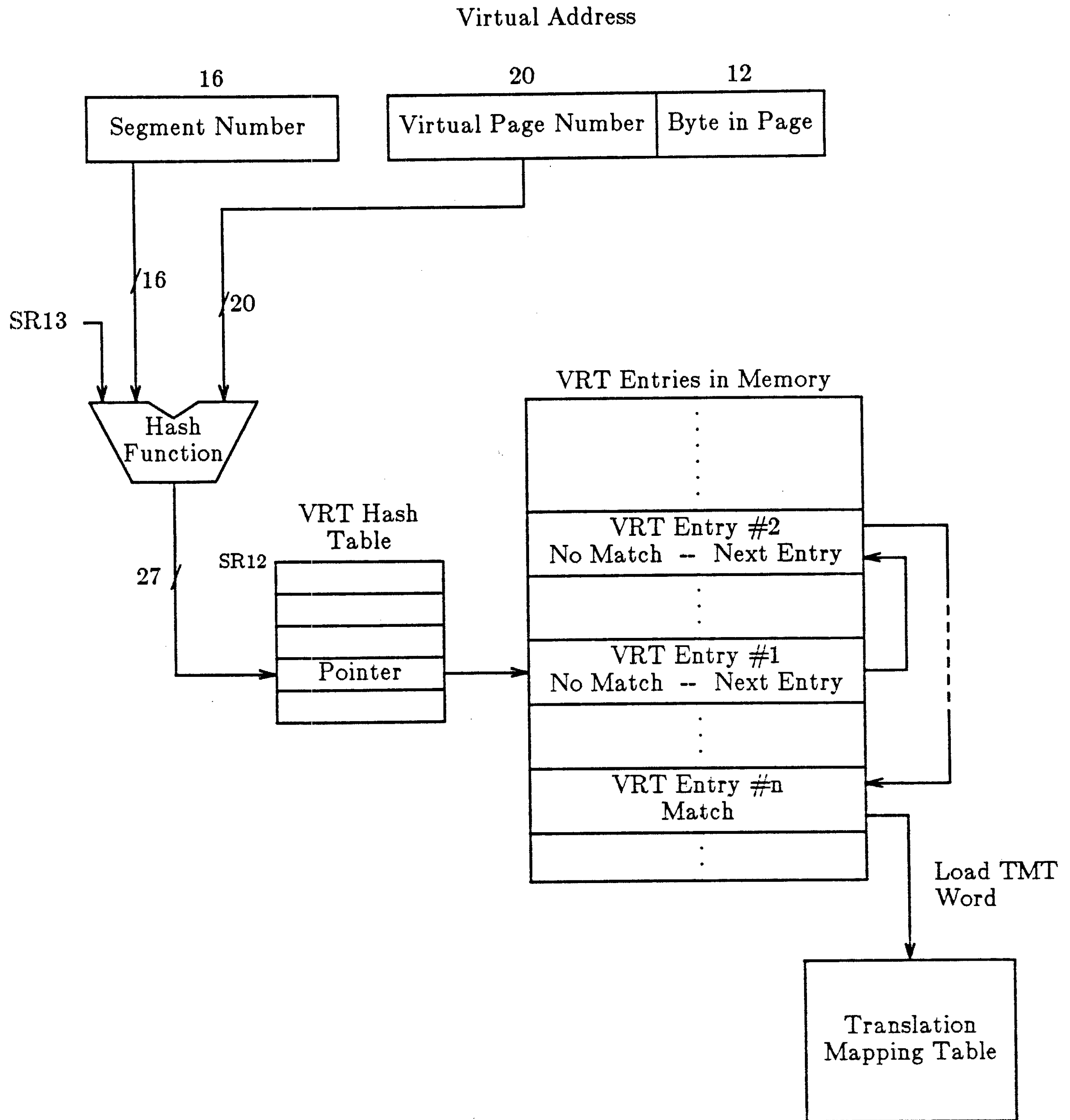


Figure 7-4. VRT Translation Sequence

TMT TABLE ORGANIZATION

The Translation Mapping Table (TMT) consists of a cache of 1024 registers. Each register (or entry) logically contains a virtual page number and its corresponding real page address. 512 of the TMT's registers are reserved for code and 512 are reserved for data.

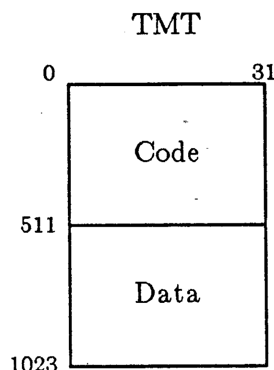
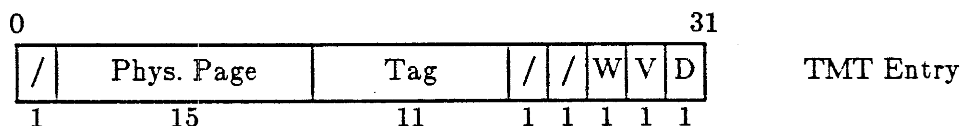


Figure 7-5. TMT Table

The addresses stored in the TMT are for the current active process. When a new process is dispatched by means of a LUS instruction, the entries for the previous process are flushed from the TMT. The TMT flush operation is executed in a single clock cycle.

TMT Entry Format

The format of a TMT entry is identical to that of the TMT Word in the corresponding VRT entry. Bits 0, 27, and 28 are ignored.



The TMT entry fields are described under *TMT Word* in the *VRT Table* Section.

TMT Translation Process

The TMT table translates a virtual address into a physical address in the following manner:

Bits 20:31 of the virtual address offset determine the byte offset within the physical page and do not require translation. Bits 11:19 are used as an index for a specific TMT entry. If the entry is valid (determined by the V bit being set), bits 0:10 of the virtual address offset are then compared with the **TMT Tag** field. If these values match, the value contained in the **Phys. Page** field specifies the physical page to complete the physical address.

The TMT translation sequence can be illustrated as follows:

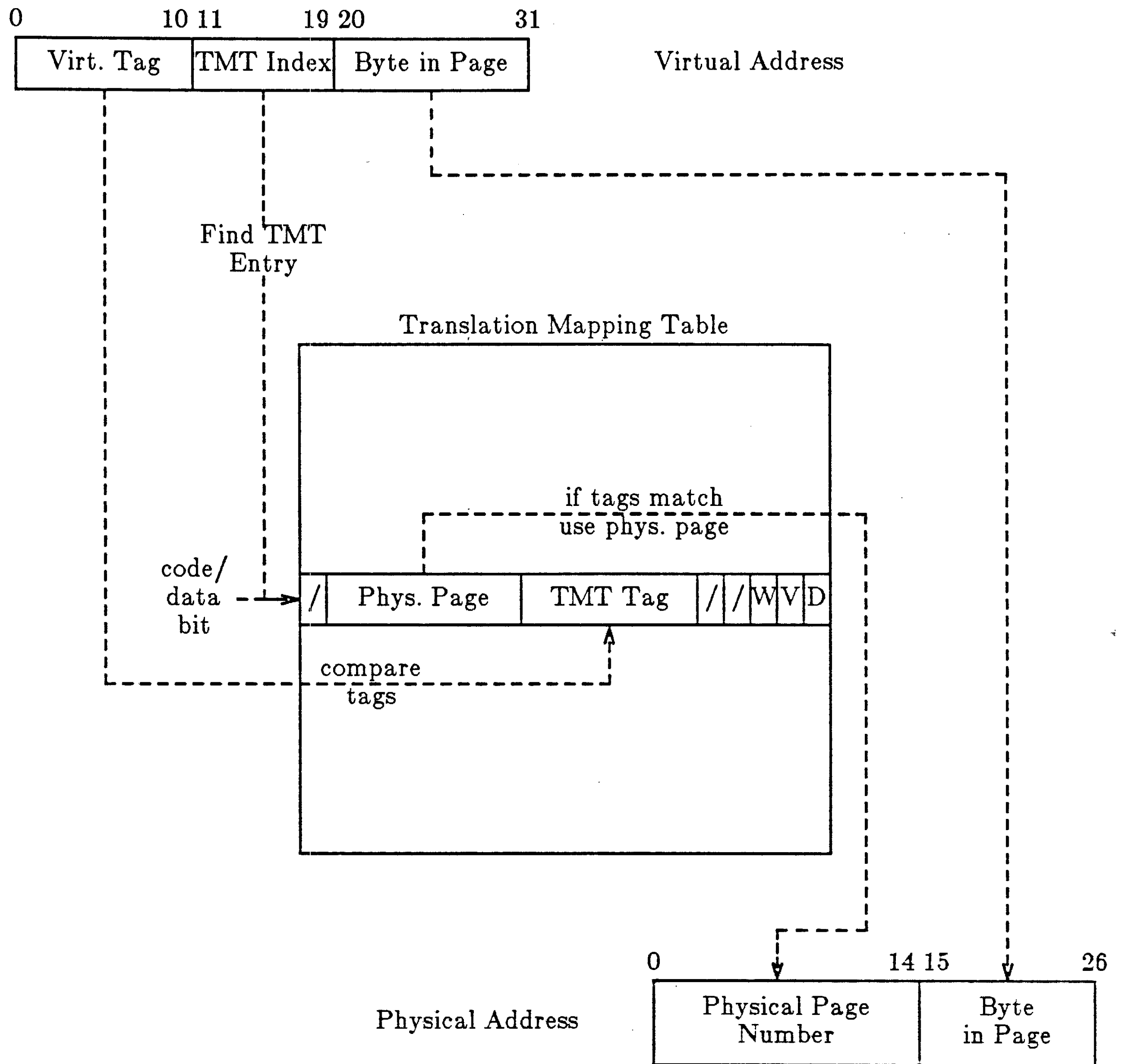


Figure 7-6. TMT Translation Sequence

Chapter 8
USER MODE INSTRUCTIONS

INTRODUCTION

The Ridge instruction set is divided into user mode instructions and kernel mode instructions. This chapter describes the user mode instructions. Kernel mode instructions are described in Chapter 9.

SYNTAX CONVENTIONS

In the descriptions of instructions, the 16 general registers are referred to as Rx or Ry. Registers 0 through 15 are referred to as R0 through R15. And the program counter is referred to as PC.

Double words occupy register pairs. A register pair, RPx, consists or Rx and Rx+1 mod 16. Rx holds the most significant bits of RPx, and Rx+1 holds the least significant bits. Example: RP5 refers to R5 and R6, with the most significant bits of the pair in R5, and the least significant bits in R6. RP15 refers to R15 and R0.

Bit 0 is the most significant bit of a data type. For 32-bit data types, bit 31 is the least significant bit. For 64-bit data types, bit 63 is the least significant bit.

Specific bits of a register or word are enclosed in brackets. For example, bit 3 of a register is referred to as Rx[3], or Ry[3]. The symbol ".." denotes a range of bits. For example, consecutive bits 6 through 9 of a register are referred to as Rx[6..9], or Ry[6..9].

Some instructions can optionally specify the 4-bit value in the Ry register field instead of the contents of Ry. This is indicated by using *Ry-field* instead of "Ry".

The instructions in the following sections are documented in the format shown below.

Instruction Summary:

Instruction Mnemonic	Instruction Function	Syntactical Description
TYP	Typical	This is a typical instruction

Operation:

The TYP instruction has no operation; it is an example of syntax conventions.

MEMORY REFERENCE INSTRUCTIONS

Memory reference instructions use either the short displacement or long displacement formats described in Chapter 4. These instructions either load data from memory to a register or store data in a register to memory.

In the following descriptions of memory instructions, optional items are surrounded by parentheses.

LOAD INSTRUCTIONS

Instruction Summary:

LOADB	Load Byte	Rx[24..31] Rx[0..23]	← contents of (Ry +) displacement ← 0
LOADH	Load Halfword	Rx[16..31] Rx[0..15]	← contents of (Ry +) displacement ← 0
LOAD	Load Word	Rx	← contents of (Ry +) displacement
LOADD	Load Doubleword	RPx	← contents of (Ry +) displacement

Operation:

The register Rx is loaded with the data stored in memory at the effective address. Ry may optionally be used as an index register. The data element must be aligned on a boundary that is a multiple of the length of the data element.

The LOADB instruction loads the byte into bits 24-31 of the specified register and sets bits 0-23 to zero.

The LOADH instruction loads the halfword into bits 16-31 of the specified register and sets bits 0-15 to zero.

The LOAD instruction loads the word into the specified register.

The LOADD instruction loads two words into RPx.

The instructions shown above are for loading data from data space. A load-from-code-space form for each of the above instructions adds PC to the effective address. The Ridge assembler, AS, distinguishes between the instruction forms by noting that the displacement is in code or data space. See the AS section in the *ROS Programmer's Guide* for details.

STORE INSTRUCTIONS

Instruction Summary:

STOREB	Store	Byte	Rx[24..31]	→ (Ry +) displacement
STOREH	Store	Halfword	Rx[16..31]	→ (Ry +) displacement
STORE	Store	Word	Rx	→ contents of (Ry +) displacement
STORED	Store	Doubleword	RPx	→ contents of (Ry +) displacement

Operation:

The store instructions move data from the registers into memory. The effective address must be a multiple of the length of the data element.

The STOREB instruction places bits 24-31 of the specified register into memory at the effective address. Other bits (0-23) are ignored.

The STOREH instruction places bits 16-31 of the specified register into memory at the effective address. Other bits (0-15) are ignored.

The STORE instruction places the word into memory at the effective address.

The STORED instruction places the double words into memory at the effective address.

LOAD ADDRESS INSTRUCTIONS

Instruction Summary:

LADDR	Load Address	Rx ← (contents of Ry) + constant
LADDR	Load Code Address	Rx ← PC (+ contents of Ry) + constant

Operation:

The load address instructions store the effective address into Rx. These instructions do not perform memory references, but instead load a constant from the instruction stream into a code- or data-relative register.

The LADDR instruction can be used to load two- or four-byte immediate values and, in indexed mode, can be used to add a constant to a register.

The operation of LADDR is varied by specifying Ry or a code-relative constant. If *constant* is data-relative, LADDR either loads register Rx with *constant* or loads register Rx with the sum of the contents of Ry and *constant.*

If the constant is code-relative, LADDR either loads register Rx with PC + *constant* or loads register Rx with the sum of the contents of Ry and PC + *constant.*

REGISTER INSTRUCTIONS

Register instructions process data taken from a specified general register. These instructions use the register instruction format described in Chapter 4. Most general register instructions specify two registers and the result usually replaces Rx.

The following pages describe the register instructions.

INTEGER ARITHMETIC INSTRUCTIONS

Instruction Summary:

ADD	Integer add	Rx ← Rx + Ry
DIV	Integer divide	Rx ← Rx/Ry
MPY	Integer multiply	Rx ← Rx*Ry
NEG	Integer negate	Rx ← 2's complement of Ry
REM	Integer remainder	Rx ← Rx - ((Rx/Ry)*Ry)
SUB	Integer subtract	Rx ← Rx - Ry

Operation:

The integer arithmetic instructions operate on 32-bit two's complement integers.

The ADD instruction adds Rx and Ry and puts the sum in Rx.

The DIV instruction divides Rx by Ry and puts the quotient in Rx.

The MPY instruction multiplies Rx and Ry and replaces the contents of Rx with the low order 32 bits of the product.

The NEG instruction puts the 2's complement of Ry in Rx.

The REM instruction divides Rx by Ry and puts the signed remainder in Rx. The sign of the remainder will be the sign of the divisor.

The SUB instruction subtracts Rx from Ry and puts the difference in Rx.

LOGICAL OPERATOR INSTRUCTIONS

Instruction Summary:

AND	Logical And	Rx ← Rx AND Ry
MOVE	Move Register	Rx ← Ry
NOT	Logical Not	Rx ← 1's complement of Ry
OR	Logical Or	Rx ← Rx OR Ry
XOR	Logical Xor	Rx ← Rx XOR Ry
NOP	No operation	Rx ← Rx

Operation:

The logical operator instructions operate on 32-bit unsigned integers in registers. The result replaces the contents of Rx.

The AND instruction performs logical AND on the contents of Rx and Ry and puts the result in Rx.

The MOVE instruction copies the contents of Ry into Rx.

The NOT instruction 1's complements the contents of Ry and puts the result in Rx.

The OR instruction performs logical OR on the contents of Rx and Ry and puts the result in Rx.

The XOR instruction performs logical XOR on the contents of Rx and Ry and puts the result in Rx.

The NOP instruction performs no operation and is sometimes used to fill instruction space. It supplies padding between modules to allow for proper alignment.

INTEGER AND LOGICAL IMMEDIATE INSTRUCTIONS

Instruction Summary:

MOVE	Move immediate	Rx ← Ry field
NOT	Not immediate	Rx ← 1's complement of Ry field
ADD	Add immediate	Rx ← Rx + Ry field
SUB	Subtract immediate	Rx ← Rx - Ry field
AND	And immediate	Rx ← Rx AND Ry field
MPY	Multiply immediate	Rx ← Rx*Ry field

Operation:

The integer and logical immediate instructions share the same format and perform the same operations as the integer arithmetic and logical operator instructions previously described. The immediate instructions differ in that the four-bit value of the Ry field is used instead of the *contents* of the Ry register. The value in Ry can be any integer between 0 and 15.

EXTENDED PRECISION INTEGER INSTRUCTIONS

Instruction Summary:

EADD	Extended Integer Add	Rx	← Rx + Ry + R0[31]
		R0[31]	← carry
		R0[30]	← overflow
EDIV	Extended Integer Divide	Rx	← RPx/Ry
		Rx+1	← unsigned remainder
EMPY	Extended Integer Multiply	RPx	← Rx*Ry
ESUB	Extended Integer Subtract	Rx	← Rx 1's complement + Ry + R0[31]
		R0[31]	← carry
		R0[30]	← overflow

Operation:

The extended precision integer instructions can be used to implement multiple-word arithmetic.

The EADD instruction adds the two's-complement integers in Rx and Ry, and at the same time adds the carry-in from R0[31], and puts the least significant 32 bits of the sum in Rx. The carry-out (most significant) bit is put in R0[31]. Overflow is indicated in R0[30]. The upper 30 bits of R0 are set to zero.

The typical use of the EADD instruction to implement multiple-word arithmetic is used as follows: R0[31] is set to zero. The least significant words are EADDed, the next-most significant words are EADDed, and so on to the most significant words. Overflow can then be checked after the last EADD.

The EDIV instruction divides the 64-bit unsigned contents of RPx by the unsigned 32-bit contents of Ry, and places the unsigned quotient in Rx and the unsigned remainder in Ry.

The EMPY instruction takes two unsigned 32-bit integers and produces an unsigned 64-bit product and places it in RPx.

The ESUB instruction one's complement subtracts the two's-complement integers in Rx and Ry, and at the same time adds the carry-in from R0[31], then puts the least significant 32-bit two's complement difference in Rx. The carry-out (most significant) bit is put in R0[31]. Overflow is indicated in R0[30]. The upper 30 bits of R0 are set to zero.

The typical use of the ESUB instruction to implement multiple-word arithmetic is used as follows: R0[31] is set to one. The least significant words are ESUBed, the next-most significant words are ESUBed, and so on to the most significant words. Overflow can then be checked after the last ESUB.

REAL INSTRUCTIONS

Instruction Summary:

FIXR	Round Real to Integer	Rx ← ROUND Ry
FIXT	Truncate Real to Integer	Rx ← TRUNC Ry
FLOAT	Convert Integer to Real	Rx ← FLOAT Ry
MAKERD	Convert Real to Double Real	RPx ← DOUBLE Ry
RADD	Real Add	Rx ← Rx + Ry
RDIV	Real Divide	Rx ← Rx/Ry
RMPY	Real Multiply	Rx ← Rx*Ry
RNEG	Real Negate	Rx ← -Ry
RSUB	Real Subtract	Rx ← Rx - Ry

Operation:

These instructions operate on 32-bit real numbers.

The FIXR instruction converts the single-precision real contents of Ry into a two's-complement integer in Rx. Values are rounded as described in the AS section of the *ROS Programmer's Guide.*

The FIXT instruction converts the single-precision real number in Ry into a 32-bit integer in Rx. All bits to the right of the decimal point are lost.

The FLOAT instruction converts the integer in Ry into a real number in Rx and rounds if necessary.

The MAKERD instruction converts the real number in Ry into a double precision real number in RPx.

The RADD instruction adds the 32-bit real numbers in Rx and Ry and puts the sum in Rx.

The RDIV instruction divides the 32-bit real number in Rx by the 32-bit real number in Ry and puts the result in Rx.

The RMPY instruction multiplies the 32-bit real numbers in Rx and Ry and puts the product in Rx.

The RNEG instruction negates the real number in Ry and puts the result in Rx.

The RSUB instruction subtracts the real number in Ry from the real number in Rx and puts the difference in Rx.

DOUBLE REAL INSTRUCTIONS

Instruction Summary:

DFIXR	Round Double Real to Integer	Rx	← ROUND RPy
DFIXT	Truncate Double Real to Integer	Rx	← TRUNC RPy
DFLOAT	Convert Integer to Double Real	RPx	← DOUBLE FLOAT Ry
DRADD	Double Real Add	RPx	← RPx + RPy
DRDIV	Double Real Divide	RPx	← RPx/RPy
DRMPY	Double Real Multiply	RPx	← RPx*RPy
DRNEG	Double Real Negate	RPx	← -RPy
DRSUB	Double Real Subtract	RPx	← RPx - RPy
MAKEDR	Round Double Real to Real	Rx	← REAL RPy

Operation:

The double real instructions perform the same operations as the real instructions previously described, except the double real instructions operate on double real format data, working on register pairs.

BIT-ORIENTED INSTRUCTIONS

Instruction Summary:

CBIT	Clear Bit	RPx[Ry mod 64] ← 0
SBIT	Set Bit	RPx[Ry mod 64] ← 1
TBIT	Test Bit	Rx[31] ← 1 if RPx [Ry mod 64] = 1 0 if RPx [Ry mod 64] = 0 Rx[0..30] ← 0

Operation:

The CBIT instruction specifies a bit number from 0-63 in Ry and the specified bit of RPx is set to zero.

The SBIT instruction specifies a bit number from 0-63 in Ry and the specified bit of RPx is set to 1.

In the TBIT instruction Ry specifies a bit number from 0-63, which is tested in RPx. The tested bit is duplicated in bit 31 of Rx, and bits 0-30 of Rx are set to zero.

TEST INSTRUCTION

Instruction Summary:

TEST	Test Values	Rx ←	1 if Rx relop Ry is true 0 if Rx relop Ry is false
			or
		Rx ←	1 if Rx relop Ry-field is true 0 if Rx relop Ry-field is false

Operation:

The TEST instruction uses a relational operator (relop) to compare two values and sets Rx to either 0 or 1, depending on the result of the test. The second operand is either the contents of the register Ry, or the 4-bit value of the Ry register field. The comparison is done using signed two's complement arithmetic. The comparison relop may be one of the following: equal to (=), less than (<), greater than (>), not equal to (<>), less than or equal to (<=), or greater than or equal to (>=).

COMPARE INSTRUCTIONS

Instruction Summary:

LCOMP	Logical Compare	Rx ← -1, if Rx < Ry Rx ← 0, if Rx = Ry Rx ← 1, if Rx > Ry
DCOMP	Double Integer Compare	Rx ← -1, if RPx < RPy Rx ← 0, if RPx = RPy Rx ← 1, if RPx > RPy
RCOMP	Real Compare	Rx ← -1, if Rx < Ry Rx ← 0, if Rx = Ry Rx ← 1, if Rx > Ry
DRCOMP	Double Real Compare	Rx ← -1, if RPx < RPy Rx ← 0, if RPx = RPy Rx ← 1, if RPx > RPy

Operation:

The LCOMP instruction compares registers Rx and Ry using unsigned arithmetic. Register Rx is set to -1, 0, or +1, depending on whether Rx is less than, equal to, or greater than Ry, respectively.

The DCOMP instruction compares register pairs RPx and RPy using two's complement arithmetic. Register Rx is set to -1, 0, or +1, depending on whether RPx is less than, equal to, or greater than RPy, respectively.

The RCOMP instruction compares real numbers in registers Rx and Ry using sign magnitude form. Register Rx is set to -1, 0, or +1, depending on whether Rx is less than, equal to, or greater than Ry, respectively.

The DRCOMP instruction compares double real numbers in register pairs RPx and RPy using sign magnitude form. Register Rx is set to -1, 0, or +1, depending on whether RPx is less than, equal to, or greater than RPy, respectively.

SHIFT INSTRUCTIONS

The shift instructions take the shift count from the contents of register Ry or from the 4-bit value of the Ry field. All shift execution times are independent of the number of bits shifted due to the use of the barrel shifter.

Single register shifts shift the value in Rx from 0 to 31 bits. Double register shifts shift the value in RPx from 0 to 63 bits. Only the low order 5 bits (6 bits for double shifts) of Ry are used as the shift count. The immediate shift forms allow shifts from 0 to 15 bits using the four bits of Ry field as the shift count.

Instruction Summary:

CSL	Circular Shift Left	Rx circularly shifted left by Ry or Ry-field
LSL	Logical Shift Left	Rx shifted left by Ry or Ry-field
LSR	Logical Shift Right	Rx shifted right by Ry or Ry-field
ASL	Arithmetic Shift Left	Rx shifted left by Ry or Ry-field
ASR	Arithmetic Shift Right	Rx shifted right by Ry or Ry-field, filling with sign bit
DLSL	Double Logical Shift Left	RPx shifted left by Ry or Ry-field
DLSR	Double Logical Shift Right	RPx shifted right by Ry or Ry-field

Operation:

The CSL instruction circularly shifts bits left in Rx. Bits shifted out of bit 0 are shifted into bit 31.

The LSL instruction shifts bits left in Rx and fills emptied positions with zeros.

The LSR instruction shifts bits right in Rx and fills emptied positions with zeros.

The ASL instruction shifts left.

The ASR instruction shifts right and fills the left bits with duplicates of the sign bit.

The DLSL and DLSR instructions correspond to LSL and LSR, except that RPx is treated as a single 64-bit register.

SIGN EXTEND INSTRUCTIONS

Instruction Summary:

SEB	Sign Extend Byte	Rx[0..23] ← Ry[24], Rx[24..31] ← Ry[24..31]
SEH	Sign Extend Halfword	Rx[0..15] ← Ry[16], Rx[16..31] ← Ry[16..31]

Operation:

The sign extend instructions change 8- or 16-bit integers into full word integers.

The SEB instruction makes bits 0-23 in register Rx the same as bit 24 in register Ry. Bits 24-31 in Ry are copied to Rx.

The SEH instruction makes bits 0-15 in register Rx the same as bit 16 in register Ry. Bits 16-31 in Ry are copied to Rx.

PROGRAM CONTROL INSTRUCTIONS

Program control instructions consists of branch, loop, and subroutine call/return instructions.

BRANCH INSTRUCTIONS

Branch instructions use either the short or long displacement memory address instruction formats described in Chapter 4. When the least significant bit of the displacement is set, the branch is predicted to be taken.

Branch instructions either switch execution to the instruction at the branch target address, or have no effect. If the branch instructions have no effect then the next sequential instruction following the branch is executed. Branch instructions affect the value of the program counter (PC) as shown below.

Next PC ← PC + branch instruction length (next sequential instruction)
or
Next PC ← PC + displacement (branch target address)

The branch instructions use program counter (PC) relative addressing, which allows self-relocating code. The target address of the branch instruction is computed by adding the 32-bit signed displacement (sign extended to 32 bits in the short form case) to the PC at the beginning of the branch instruction.

The least significant bit of the displacement field is used by the processor to predict whether or not the branch will be taken. If the bit is one, the processor will prefetch the instruction at the target address. If the bit is zero, the processor will prefetch the next sequential instruction. If the bit is incorrect, the program will execute correctly, but the next instruction after the branch will be delayed by two to four cycles to fill the pipeline.

Instruction Summary:

BR	Unconditional Branch	PC ← PC + displacement
BR	Conditional Branch	PC ← PC + displacement, if Rx relop Ry or Rx relop Ry-field

Operation:

The unconditional branch instruction changes PC to the target address, PC + displacement. The branch prediction bit is ignored and the target instruction is always prefetched.

The conditional branch instruction compares Rx to the contents of Ry or to the 4-bit value of the Ry-field, then may conditionally branch to the target location. The conditional branch instruction comparisons are made using two's complement arithmetic. The comparison uses the relational operator (relop), which may be: equal to (=), less than (<), greater than (>), not equal to (<>), less than or equal to (<=), or greater than or equal to (>=).

LOOP CONTROL INSTRUCTION

Instruction Summary:

LOOP Increment and Branch	Rx ← Rx + Ry-field PC ← PC + displacement, if Rx $<$ 0,

Operation:

The LOOP instruction is similar to the conditional branch described above. The LOOP instruction adds the 4-bit value of the Ry field to the contents of Rx and branches to the target location if the result is less than zero. If Rx is equal to or greater than zero, the next sequential instruction is executed.

SUBROUTINE CALL AND RETURN INSTRUCTIONS

There are three subroutine call and return instructions: call subroutine, call subroutine register and return from subroutine.

Call Subroutine Instruction

Instruction Format:

The call subroutine instruction uses the short and long displacement memory address instruction format shown below. The second operand field, Ry, is not used in this instruction.

Short displacement memory address

CALL	Rx	not used	displacement

Long displacement memory address

CALL	Rx	not used	displacement
0 7	8 11	12 15	16 31 47

Instruction Summary:

CALL Call Subroutine Rx ← PC + instruction length,
PC ← PC + displacement

Operation:

The call instruction places the address of the next instruction in Rx and transfers execution to the target location (PC + displacement). Short displacement memory addresses are sign extended. Like the branch instructions, the call instruction uses program counter (PC) relative addressing, which allows self-relocating code.

Call Subroutine Register and Return Instructions

The CALLR and RET instructions use the register instruction format described in Chapter 4.

Instruction Summary:

CALLR	Call Subroutine Register	Rx ← PC + 2 PC ← PC + Ry
RET	Return from Subroutine	Rx ← PC + 2 PC ← Ry

Operation:

The CALLR instruction stores the address of the next sequential instruction, PC + 2 in Rx, and branches to the location PC + Ry.

The RET instruction stores the address of the next sequential instruction, PC + 2 in Rx and branches to the absolute address in Ry. The main use of RET is in returning from subroutines, but it can also be used as a call to a subroutine when the absolute rather than the relative address is known. Care must be taken in using the RET instruction for this purpose so that the code remains self-relocating.

TRAP INSTRUCTIONS

Trap instructions can be used by a user process to trap to the kernel and execute specific event handling routines in real memory.

Instruction Summary:

CHK	Check Rx range	Trap if Rx > Ry
CHKI	Check Immediate Rx range	Trap if NOT (0 <= (Rx) <= Ry)
TRAP	Trap	Trap if Ry enabled in traps word
KCALL	Kernel call	Trap to specified location

Operation:

The CHK and CHKI instructions check whether Rx is larger than Ry. If so, the instruction performs no operation, otherwise the instruction traps to the kernel (see the *CHECK TRAP* Section in Chapter 6).

The TRAP instruction is used to set breakpoints in debuggers, and other purposes that require user-specified traps to be selectively enabled or disabled.

There are 16 instruction traps which can be individually enabled or disabled in order to set breakpoints while using the rbug, debug, or dbx debuggers. The specific trap to be taken is specified by the Ry field of the TRAP instruction. When the TRAP instruction is executed, its Ry field is loaded into SR3. A trap will occur if the 4-bit value in SR3 indicates a set bit among the most significant 16 bits (0..15) in the *traps word* (SR10). If the bit specified by the TRAP instruction is not set in the traps word, no trap will be taken. (See the *TRAP INSTRUCTION TRAP* in Chapter 6.)

The KCALL instruction is used to trap to specific memory locations in order to perform some kernel operation, such as process creation or modification of a particular data structure. The CPU Control Block (see Chapter 6) contains KCALL vectors to 255?? memory locations. The kernel operation to be performed is specified by the KCALL instruction followed by a number that identifies that operation. (See the *KERNEL CALLS* section in Chapter 6 and the *Ridge Internals Manual* for more detail on the KCALL instruction.)

Chapter 9
KERNEL MODE INSTRUCTIONS

INTRODUCTION

Kernel mode instructions handle process communication, process creation and deletion, interrupts, traps, and faults. Kernel mode is used to provide all privileged activity that involves data sharing or synchronization.

As described in Chapter 5, kernel mode is distinguished from user mode in that real, rather than virtual, memory addresses are used. When in kernel mode, external interrupts are disabled and a broader range of instructions are available.

Some of the kernel mode instructions discussed in this chapter can be executed in user mode if the privileged process bit is set (see Chapter 5). However, when executed in user mode, kernel instructions will use virtual addresses and all interrupts will be enabled. As a consequence, it is impractical to execute some of the kernel instructions in user mode.

The same syntax conventions described in the *SYNTAX CONVENTIONS* section in Chapter 8 are used for the instructions discussed in this chapter.

STATE SWITCHING INSTRUCTIONS

These instructions are executable in kernel mode only and cannot be executed in user mode by the privileged process.

Instruction Summary:

SUS	Save User State	PCB ← PC, process clock, (Ri)
LUS	Load User State	(SRi) ← PC, code & data seg #'s, traps word
LDREGS	Load Registers	(Rx) ← (Ry)
RUM	Resume User Mode	user PC ← (SR15)
MOVE	Move General to Special Register	Rx ← Ry; (SR1) ← Ry; Rx ← (SR2)

Operation:

The SUS instruction stores the user program counter and the process clock into the PCB pointed to by special register SR14. In addition, the general-purpose registers are stored beginning with Rx and ending with Ry. The instruction can store from 1 to 16 of the general-purpose registers.

If the Rx specification is greater than Ry then only Rx is stored (i.e., register numbers do not wrap around). If SR14 = 1, no registers are stored and the instruction performs no operation.

The LUS instruction is the inverse of the SUS instruction. It loads the user program counter, the code and data segment numbers, and the traps word from the PCB pointed to by SR14 into SR15, SR8, SR9, and SR10, respectively. From 1 to 16 general registers can also be loaded.

If Rx is greater than Ry, only Rx is loaded (i.e., register numbers do not wrap around). The instruction cache and translation mapping table (see the section of virtual memory which follows)) are flushed. If SR14 = 1, no registers are loaded and the instruction performs no operation.

The LDREGS instruction loads from 1 to 16 registers from the PCB pointed to by SR14. It provides a faster method for loading registers than the LUS instruction and is useful in restoring the user state after a kernel operation that does not cause a process context switch.

The LDREGS instruction differs from LUS in that the program counter, code and data segments, and traps word are not read from the PCB, and the instruction cache and translation mapping table are left unchanged.

If Rx is greater than Ry, then only Rx is loaded (i.e., there is no register wrap around). If SR14 = 1, no registers are loaded and the instruction performs no operation.

The RUM instruction switches the processor from kernel to user mode, loading the program counter from SR15. The user program begins executing at the location indicated in SR15.

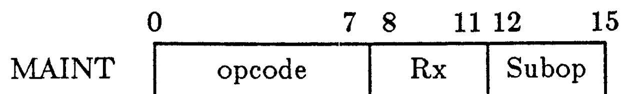
If SR14 = 1, the processor pauses until an interrupt occurs. At this point the kernel is entered, and SR0 is set to the kernel's program counter. The kernel interrupt handler may then LUS and RUM to a new user program, or if SR14 remains set to one, the LUS has no effect and the RUM again causes the processor to pause.

The MOVE instruction moves a general-purpose register to a special-purpose register, or vice-versa.

MAINTENANCE INSTRUCTIONS

Maintenance instructions are register format instructions that use the Ry field as part of the opcode ("subop"). In these instructions, Rx or RPx is used for both input and output. These instructions are kernel mode instructions that can also be executed in user mode with the privileged process set.

Instruction Format:



Where:

Subop (decimal)	Maintenance Instruction
0	ELOGR
1	ELOGW
5	TWRITED
6	FLUSH
7	TRAPEXIT
8	ITEST
10	MACHINEID
11	VERSION
12	CREG
13	RDLOG

Instruction Summary:

ELOGR	Read Processor Status	Rx ← processor status
ELOGW	Write Memory Error Logging Data	Rx → error logging data
TWRITED	External Interrupt Enable/Disable	Rx → interrupt mask
FLUSH	Flush Translation Buffer	
TRAPEXIT	Exit From Trap Instruction	
ITEST	Interrupt Test	RPx ← interrupt, IOIR data
MACHINEID	Machine Identification Word	Rx ← machine ID
VERSION	Microcode Version Number	Rx ← microcode version #
CREG	Clock Register In/Out	Rx[30,31] → clock board output reg. Rx+1 ← clock board input reg.
RDLOG	Read Memory Error Logging Data	Rx ← error logging data

Operation:

ELOGR

The ELOGR instruction reads the processor status, regardless of the input Rx value.

Bits 16..31 of Rx are the status bits used by the processor. Only bit 23 (the power fail warning flag) and bit 31 (loaf enable) can be interpreted, the rest are manipulated by microcode and will always be zero when the ELOGR instruction is executed.

ELOGW

The ELOGW instruction writes up to 16 single-bit errors into a logging area in hardware. Before execution of ELOGW, the low order bit in the Rx register will contain a 1 if the entire logging area is to be reset. A zero in bit 31 indicates no reset.

TWRITED

The TWRITED instruction enables or disables external interrupts from I/O devices. Two masks, Mask 1 and Mask 2, control the two daisy-chained signals discussed in Chapter 3.

Bits 24-25 control Mask 1. 01 sets and 00 resets the mask.
Bits 26-27 control Mask 2. 01 sets and 00 resets the mask.
Bits 28-29 must be 1's to set or reset either mask.

Mask 2 controls IOIREQ2, which is used by real-time devices. Mask 1 controls IOIREQ1, which is used by all other non-real-time devices.

When Mask 1 is set, all interrupts over IOIREQ1 are ignored. When Mask 2 is set, all interrupts over IOIREQ1 and IOIREQ2 are ignored. When neither mask is set and a real-time I/O board asserts an IOIREQ2 signal, all interrupts requested over IOIREQ1 are ignored, as are IOIREQ2 signals from any device of a lower priority than that of current interrupting device. Devices asserting an IOIREQ2 with a higher priority than that of the current interrupting device are acknowledged.

FLUSH

The FLUSH instruction causes the entire cache (both code and data) to be flushed. This instruction may be executed in user mode with the privileged process bit set as well as in kernel mode.

TRAPEXIT

The TRAPEXIT instruction sets PC to the value contained in SR0 and begins executing at that address.

The TRAPEXIT instruction is used by the kernel to resume execution in the kernel after hitting a TRAP instruction (see Chapter 8). The TRAP instruction is used to set breakpoints, TRAPEXIT is used to exit kernel breakpoints, and resume user mode (RUM) is used to exit user breakpoints. (The RUM instruction clears kernel mode, so it can't be used to resume executing in the kernel.)

The TRAPEXIT instruction flushes the cache and the TMT. An attempt to execute this instruction when not in kernel mode results in a kernel violation trap.

ITEST

The ITEST instruction is used in kernel mode to test for the presence of an interrupt. ITEST returns the I/O Interrupt Read (IOIR) word if an interrupt is present in Rx+1 and Rx is set to zero. If there is no interrupt, Rx is set to one and Rx+1 is left unchanged.

Bits 0..7 of the I/O word contain the interrupting device number and the remaining bits contain device dependent data.

MACHINEID

The MACHINEID instruction is used to read the serial number assigned to the machine.

The number returned in Rx contains the encoded serial number, the machine model number, and the maximum user configuration.

The two 8-bit values: Rx[8..15] and Rx[24..31] can be joined together to determine a 16-bit model number. Adding this value to a serial number base produces the serial number of the machine. The maximum user configuration is determined by bits Rx[20..23]. If Rx[20..23] are all 1's, there is no maximum user configuration.

Specific hardware options can be determined by bits Rx[28..31]. These are:

31 = Not used. (Formerly copy-on-write memory controller present. Obsolete, but still returned by some versions of microcode.)

30 = Enhanced floating point present. (Set to 1 for 3200 processor.)

29 = old/new VRT layout. Old VRT was limited to 8 Mbytes of memory. New VRT is limited to 128 Mbytes.

28 = 3200 processor present.

VERSION

The VERSION instruction returns the current microcode version number in Rx.

CREG

The CREG instruction is used to both write data to the clock board output register and to read data from the clock board input register.

The processor initiates requests by setting bit 31 in the Rx register. Bit 30 in the Rx register is used for data-out.

The clock board acknowledges requests by setting bit 31 in register Rx+1. Bit 30 in the Rx+1 register is used for data-in.

Acknowledge and data-in bits are latched before the request and data-out are given to the clock board.

RDLOG

The RDLOG instruction either reads the contents of any of the 16 logout data locations or returns the number of valid logout entries.

To return the number of valid logout entries, bit 0 must be turned on. All other bits in Rx are ingored.

To return the contents of a logout area location, the low order 4 bits of Rx must contain the desired logout area address and bit 0 must be turned off.

After execution, Rx will contain the logout data from the specified location, or the number of valid logout entries.

VIRTUAL MEMORY SUPPORT INSTRUCTIONS

These instructions are executable in kernel mode only and cannot be executed in user mode by the privileged process.

Instruction Summary:

TRANS	Translate Virtual Address	(Rx) ← Real RPy
DIRT	Translate Virtual Address and Mark Page Dirty	(Rx) ← Real RPy

Operation:

The TRANS instruction takes the segment number in Ry and the virtual address in Ry' and replaces Rx with the corresponding real address. If the address is not translatable with the current Virtual to Real Translation (VRT) table, then Rx is set to -1. If the address is translatable, then the reference bit is set in the VRT for this address. (The VRT is discussed in Chapter 7.)

The DIRT instruction is the same as the TRANS instruction except that the modified bit in the VRT for the page containing the virtual address is also set.

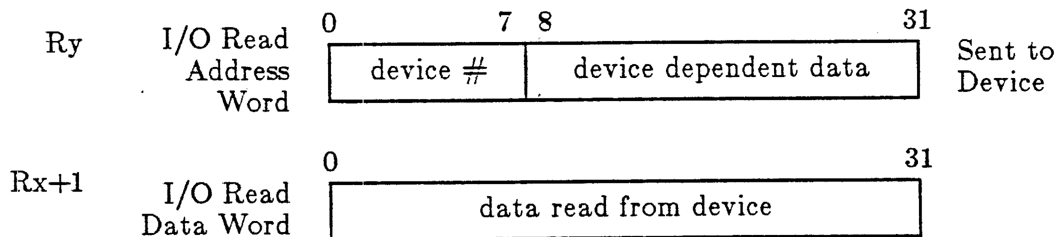
INPUT/OUTPUT INSTRUCTIONS

I/O is accomplished by the READ and WRITE instructions. These instructions can be executed in user mode when the privileged process bit is set.

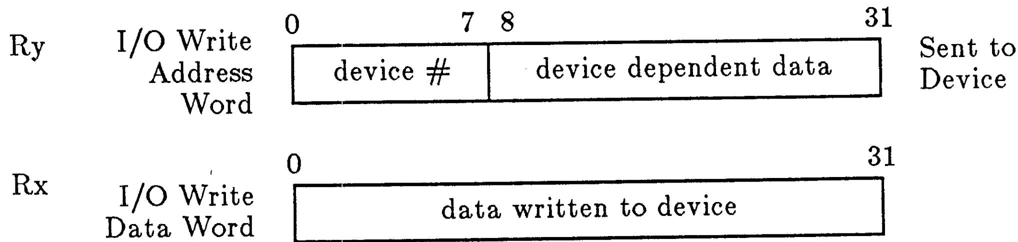
Register formats for read, write, and I/O status words are as follows:

Instruction Formats:

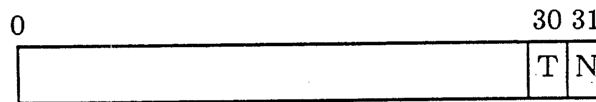
READ Instruction



WRITE Instruction



I/O STATUS returned in Rx



T = "0" is ok, "1" is device timed out and did not respond.
N = "0" is ok, "1" is I/O device not ready to accept command.

Instruction Summary:

READ	Read Data from Device	device	← IORA from (Ry),
		(Rx)	← status from device,
		(Rx+1)	← IORD from device
WRITE	Write Data to Device	device	← IOWA from (Ry) and IOWD from (Rx),
		(Rx)	← status from device

Operation:

The READ instruction sends the contents of Ry as an I/O read address (IORA) word to the device number specified in the most significant byte of Ry. (The least significant bytes of Ry are device dependent data.) Rx is set to the I/O status and Rx+1 (mod 16) contains the I/O read data (IORD) word from the device.

The WRITE instruction sends Ry as an I/O write address (IOWA) word and Rx as an I/O write data (IOWD) word to the device specified in the most significant byte of Ry. Rx is then set to the I/O status.

Appendix A
INSTRUCTION INDEX

In the following instruction definitions, *disp* refers to a 16- or 32-bit displacement, *val* refers to a 4-bit value, and *num* refers to an 8-bit value.

Instruction	Description	Page
ADD Rx, Ry	Integer Add	8-4
ADD Rx, *val*	Add immediate	8-6
AND Rx, Ry	Logical And	8-5
AND Rx, *val*	And immediate	8-6
ASL Rx, Ry	Arithmetic Shift Left	8-12
ASR Rx, Ry	Arithmetic Shift Right	8-12
BR Rx *relop* Ry, *disp*	Conditional Branch	8-14
BR *disp*	Unconditional Branch	8-14
CALL Rx, *disp*	Call Subroutine	8-16
CALLR Rx, Ry	Call Subroutine Register	8-17
CBIT Rx, Ry	Clear Bit	8-10
CHK Rx, Ry	Check Rx range	8-18
CHKI Rx, *val*	Check Immediate Rx range	8-18
CREG Rx	Clock Register In/Out	9-6
CSL Rx, Ry	Circular Shift Left	8-12
DCOMP Rx, Ry	Double Integer Compare	8-11
DFIXR Rx, Ry	Round Double Real to Integer	8-9
DFIXT Rx, Ry	Truncate Double Real to Integer	8-9
DFLOAT Rx, Ry	Convert Integer to Double Real	8-9
DIRT Rx, Ry	Translate Virtual Address and Mark Page Dirty	9-7
DIV Rx, Ry	Integer divide	8-4
DLSL Rx, Ry	Double Logical Shift Left	8-12
DLSR Rx, Ry	Double Logical Shift Right	8-12
DRADD Rx, Ry	Double Real Add	8-9
DRCOMP Rx, Ry	Double Real Compare	8-11
DRDIV Rx, Ry	Double Real Divide	8-9
DRMPY Rx, Ry	Double Real Multiply	8-9
DRNEG Rx, Ry	Double Real Negate	8-9
DRSUB Rx, Ry	Double Real Subtract	8-9
EADD Rx, Ry	Extended Integer Add	8-7
EDIV Rx, Ry	Extended Integer Divide	8-7
ELOGR Rx	Read Processor Status	9-4
ELOGW Rx	Write Memory Error Logging Data	9-4
EMPY Rx, Ry	Extended Integer Multiply	8-7
ESUB Rx, Ry	Extended Integer Subtract	8-7
FIXR Rx, Ry	Round Real to Integer	8-8
FIXT Rx, Ry	Truncate Real to Integer	8-8
FLOAT Rx, Ry	Convert Integer to Real	8-8
FLUSH	Flush Translation Buffer	9-4
ITEST Rx	Interrupt Test	9-5
KCALL *num*	Kernel call	8-18
LADDR Rx [,Ry] ,*disp*	Load Data Address	8-3
LCOMP Rx, Ry	Logical Compare	8-11
LDREGS Rx, Ry	Load Registers	9-1
LOAD Rx [,Ry] ,*disp*	Load Word	8-2

Instruction	Description	Page
LOADB Rx [,Ry] ,*disp*	Load Byte	8-2
LOADD Rx [,Ry] ,*disp*	Load Doubleword	8-2
LOADH Rx [,Ry] ,*disp*	Load Halfword	8-2
LOOP Rx, *val, disp*	Increment and Branch	8-15
LSL Rx, Ry	Logical Shift Left	8-12
LSR Rx, Ry	Logical Shift Right	8-12
LUS Rx, Ry	Load User State	9-1
MACHINEID	Machine Identification Word	9-5
MAKEDR Rx, Ry	Round Double Real to Real	8-9
MAKERD Rx, Ry	Convert Real to Double Real	8-8
MOVE Rx, Ry	Move Register	8-5
MOVE Rx, *val*	Move immediate	8-6
MOVE Rx, SRy	Move General Register to Special Register	9-1
MOVE SRx, Ry	Move Special Register to General Register	9-1
MPY Rx, Ry	Integer multiply	8-4
MPY Rx, *val*	Multiply immediate	8-6
NEG Rx, Ry	Integer negate	8-4
NOP	No operation	8-5
NOT Rx, Ry	Logical Not	8-5
NOT Rx, *val*	Not immediate	8-6
OR Rx, Ry	Logical Or	8-5
RADD Rx, Ry	Real Add	8-8
RCOMP Rx, Ry	Real Compare	8-11
RDIV Rx, Ry	Real Divide	8-8
RDLOG Rx	Read Memory Error Logging Data	9-6
READ Rx, Ry	Read Data from Device	9-7
REM Rx, Ry	Integer remainder	8-4
RET Rx, Ry	Return from Subroutine	8-17
RMPY Rx, Ry	Real Multiply	8-8
RNEG Rx, Ry	Real Negate	8-8
RSUB Rx, Ry	Real Subtract	8-8
RUM	Resume User Mode	9-1
SBIT Rx, Ry	Set Bit	8-10
SEB Rx, Ry	Sign Extend Byte	8-13
SEH Rx, Ry	Sign Extend	8-13
STORE Rx, *disp*	Store Word	8-3
STOREB Rx, *disp*	Store Byte	8-3
STORED Rx, *disp*	Store Doubleword	8-3
STOREH Rx, *disp*	Store Halfword	8-3
SUB Rx, Ry	Integer subtract	8-4
SUB Rx, *val*	Subtract immediate	8-6
SUS Rx, Ry	Save User State	9-1
TBIT Rx, Ry	Test Bit	8-10
TEST Rx *relop* Ry	Test Values	8-10
TRANS Rx, Ry	Translate Virtual Address	9-7
TRAP *val*	Trap	8-18
TRAPEXIT	Exit From Trap Instruction	9-5
TWRITED Rx	External Interrupt Enable/Disable	9-4
VERSION Rx	Microcode Version Number	9-6
WRITE Rx, Ry	Write Data to Device	9-8
XOR Rx, Ry	Logical Xor	8-5

Appendix B
INSTRUCTION EXECUTION TIMES

In the following instruction definitions, *disp* refers to a 16- or 32-bit displacement, *val* refers to a 4-bit value, and *num*

Opcode	Assembler Mnemonic	Execution time (# of cycles)
Simple Instructions		
add	ADD Rx, Ry	1
addi	ADD Rx, Imm	1
and	AND Rx, Ry	1
cndi	AND Rx, Imm	1
lcomp	LCOMP Rx, Ry	2 - Rx < Ry 3 - Rx >= Ry
moverr	MOVE Rx, Ry	1
moveri	MOVE Rx, Imm	1
movesr	MOVE SRx, Ry	2
movers	MOVE Rx, SRy	2
neg	NEG Rx, Ry	2
nop	NOP 0,0	1
not	NOT Rx, Ry	1
noti	NOT Rx, Imm	1
or	OR Rx, Ry	1
seb	SEB Rx, Ry	2
seh	SEH Rx, Ry	2
sub	SUB Rx, Ry	1
subi	SUB Rx, Imm	1
xor	XOR Rx, Ry	1
Shift Instructions		
asl	ASL Rx, Ry	2 - Kernel 4 - TNK
asli	ASL Rx, Imm	Same as ASL
asr	ASR Rx, Ry	2
asri	ASR Rx, Imm	2
lsl	LSL Rx, Ry	1
lsli	LSL Rx, Imm	1
lsr	LSR Rx, Ry	1
lsri	LSR Rx, Imm	1
csl	CSL Rx, Ry	2
csli	CSL Rx, Imm	2
dlsl	DLSL Rx, Ry	3
dlsli	DLSL Rx, Imm	3
dlsr	DLSR Rx, Ry	3
dlsri	DLSR Rx, Imm	3

Branch/Call/Return

br	BR unc, short	1
brl	BR unc, long	1
br>	BR Rx > Ry, short	2 - correct prediction
		4 - incorrect prediction
br>l	BR Rx > Ry, long	Same as BR >
br=	BR Rx = Ry, short	Same as BR >
br=l	BR Rx = Ry, long	Same as BR >
br<=	BR Rx <= Ry, short	Same as BR >
br<=l	BR Rx <= Ry, long	Same as BR >
br<>	BR Rx <> Ry, short	Same as BR >
br<>l	BR Rx <> Ry, long	Same as BR >
bri>	BR Rx > imm, short	Same as BR >
bri>l	BR Rx > imm, long	Same as BR >
bri<	BR Rx < imm, short	Same as BR >
bri<l	BR Rx < imm, long	Same as BR >
bri=	BR Rx = imm, short	Same as BR >
bri=l	BR Rx = imm, long	Same as BR >
bri<=	BR Rx <= imm, short	Same as BR >
bri<=l	BR Rx <= imm, long	Same as BR >
bri>=	BR Rx >= imm, short	Same as BR >
bri>=l	BR Rx >= imm, long	Same as BR >
bri<>	BR Rx <> imm, short	Same as BR >
bri<>l	BR Rx <> imm, long	Same as BR >
call	CALL Rx, label (short)	2
calll	CALL Rx, label, L	2
callr	CALLR Rx, Ry	4
loop	LOOP Rx, imm, label	2 - sign xor ovfl
		4 - otherwise
loopl	LOOP Rx, imm, label, L	Same as LOOP
ret	RET Rx, Ry	4

Bit Instructions

cbit	CBIT Rx, Ry	3
sbit	SBIT Rx, Ry	3
tbit	TBIT Rx, Ry	3

Loads

load	LOAD Rx, <Addr>	2
loadl	LOAD Rx, <Addr>, L	2
loadx	LOAD Rx, Ry + <Addr>	2
loadlx	LOAD Rx, Ry+<Addr>, L	2
loadp	LOADP Rx, <Addr>	2
loadpl	LOADP Rx, <Addr>, L	2
loadpx	LOADP Rx, Ry + <Addr>	2
loadplx	LOADP Rx, Ry+<Addr>, L	2
loadh	LOADH Rx, <Addr>	2
loadhl	LOADH Rx, <Addr>, L	2
loadhx	LOADH Rx, Ry + <Addr>	2
loadhlx	LOADH Rx, Ry+<Addr>, L	2

loadhp	LOADHP Rx, <Addr>	2
loadhpl	LOADHP Rx, <Addr>, L	2
loadhpx	LOADHP Rx, Ry + <Addr>	2
loadhplx	LOADHP Rx, Ry+<Addr>, L	2
loadb	LOADB Rx, <Addr>	2
loadbl	LOADB Rx, <Addr>, L	2
loadbx	LOADB Rx, Ry + <Addr>	2
loadblx	LOADB Rx, Ry+<Addr>, L	2
loadbp	LOADBP Rx, <Addr>	2
loadbpl	LOADBP Rx, <Addr>, L	2
loadbpx	LOADBP Rx, Ry + <Addr>	2
loadbplx	LOADBP Rx, Ry+<Addr>, L	2
loadd	LOADD Rx, <Addr>	3
loaddl	LOADD Rx, <Addr>, L	3
loaddx	LOADD Rx, Ry + <Addr>	3
loaddlx	LOADD Rx, Ry+<Addr>, L	3
loaddp	LOADDP Rx, <Addr>	3
loaddpl	LOADDP Rx, <Addr>, L	3
loaddpx	LOADDP Rx, Ry + <Addr>	3
loaddplx	LOADDP Rx, Ry+<Addr>, L	3

Stores

store	STORE Rx, <Addr>	3
storel	STORE Rx, <Addr>, L	3
storex	STORE Rx, Ry + <Addr>	3
storelx	STORE Rx, Ry+<Addr>, L	3
storeh	STOREH Rx, <Addr>	3
storehl	STOREH Rx, <Addr>, L	3
storehx	STOREH Rx, Ry + <Addr>	3
storehlx	STOREH Rx, Ry+<Addr>, L	3
storeb	STOREB Rx, <Addr>	3
storebl	STOREB Rx, <Addr>, L	3
storebx	STOREB Rx, Ry + <Addr>	3
storeblx	STOREB Rx, Ry+<Addr>, L	3
stored	STORED Rx, <Addr>	7
storedl	STORED Rx, <Addr>, L	7
storedx	STORED Rx, Ry + <Addr>	7
storedlx	STORED Rx, Ry+<Addr>, L	7

Load Address

laddr	LADDR Rx, <Addr>	1
laddrx	LADDR Rx, Ry, <Addr>	1
laddrl	LADDR Rx, <Addr>, L	1
laddrlx	LADDR Rx, Ry, <Addr>, L	1
laddrp	LADDR Rx, <Addr>	1
laddrpx	LADDR Rx, Ry, <Addr>	1
laddrpl	LADDR Rx, <Addr>, L	1
laddrplx	LADDR Rx, Ry, <Addr>, L	1

Load/Store User State

lus	LUS Rx, Ry	17 + 2(#regs - 1)
sus	SUS Rx, Ry	10 + 3(#regs - 1)
rum	RUM 0,0	5 (resume case)
ldregs	LDREGS Rx, Ry	5 + 2(#regs - 1)

Check and Trap

trap	TRAP 0,<4 bit>	4 - No Trap 4 + Trap routine - Trap
chk	CHK Rx, Ry	2 - No Trap 2 + Trap routine - Trap
chki	CHK Rx,<4 bit>	2 - No Trap 2 + Trap routine - Trap

Dirt/Trans

trans	TRANS Rx, RPy	24 - typical
dirt	DIRT Rx, Ry	25 - typical

Read/Write

read	READ Rx, Ry	15
write	WRITE Rx, Ry	14

Kcall

kcall	KCALL <8 bit>	8

Maint

flush	FLUSH 0,6	13
trapexit	TRAPEXIT 0,7	12

Extended

dcomp	DCOMP RPx, RPy	2 - Rx(h.o.) < Ry(h.o.) 3 - Rx(h.o.) > Ry(h.o.) 3 - 4 others
eadd	EADD Rx, Ry	5
esub	ESUB Rx, Ry	5

SP Converts

fixt	FIXT Rx, Ry	4 - Ry > 0 8 - Ry = max neg.
fixr	FIXR Rx, Ry	4 - Ry = 0 5 - Ry <> 0 8 - Ry = max neg.
float	FLOAT Rx, Ry	3 - Ry = 0 4 - Ry > 0 7 - Ry < 0
makedr	MAKEDR Rx, RPy	3

DP Converts

dfixt	DFIXT Rx, Ry	4 - typical
dfixr	DFIXR RX, Ry	5 - typical
dfloat	DFLOAT Rx, Ry	4 - Ry = 0 5 - Ry > 0 7 - Ry < 0
makerd	MAKERD RPx, Ry	4

SP Simple

radd	RADD Rx,Ry	5 - typical
rsub	RSUB Rx,Ry	5 - typical
rneg	RNEG Rx,Ry	3
rcomp	RCOMP Rx,Ry	4 - X>=Y & sign(X)=sign(Y) (no spec-in) 3 - others (no spec-in)

DP Simple

dradd	DRADD Rx,Ry	7 - typical
drsub	DRSUB Rx,Ry	7 - typical
drneg	DRNEG Rx,Ry	4
drcomp	DRCOMP Rx,Ry	5 - X>=Y and sign(X)=sign(Y) (no spec-in) 4 - others (no spec-in)

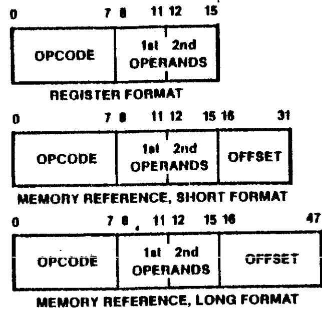
Multiply's

mpyr	MPYR Rx,Ry	10
mpyi	MPYI Rx,<imm>	6 - no overflow
		7 - overflow, trap off
empy	EMPY RPx,Ry	11 - no overflow
		12 - overflow, trap off
rmpy	RMPY Rx,Ry	10 - normal
		13 - ovf/unf, trap off
		14 - inexact, trap off
drmpy	DRMPY RPx,RPy	16 - normal
		20 - ovf/unf, trap off
		19 - inexact, trap off

NOTE: "no spec-in" means no special case floating point numbers (e.g. NaN, denorm's, etc.) used as operands in the instruction.

RIDGE OPCODE MAP

INSTRUCTION FORMATS:



Least Significant Nibble (Hex), Opcode (4:7)

Most Significant Nibble (Hex), Opcode (0:3)

Rows 0–7: Register Format. Rows 8–F: Memory Reference Format.

Format	Segment Referenced	Format Length	MSN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Register Format			0		MOVE	NEG	ADD	SUB	MPY	DIV	REM	NOT	OR	XOR	AND	CBIT	SBIT	TBIT	CHK
			1	NOP	MOVE Immed		ADD Immed	SUB Immed	MPY Immed			NOT Immed			AND Immed				CHK Immed
			2	FIXT	FIXR	RNEG	RADD	RSUB	RMPY	RDIV	MAKERD	LCOMP	FLOAT	RCOMP		EADD	ESUB	EMPY	EDIV
			3	DFIXT	DFIXR	DRNEG	DRADD	DRSUB	DRMPY	DRDIV	MAKEDR	DCOMP	DFLOAT	DRCOMP	TRAP				
			4	SUS	LUS	RUM	LDREGS	TRANS	DIRT	MOVE SR←R	MOVE R←SR					MAINT		READ	WRITE
			5	TEST >	TEST <	TEST =	CALLR	TEST Immediate >	TEST Immediate <	TEST Immediate =	RET	TEST <=	TEST >=	TEST <>	KCALL	TEST Immediate <=	TEST Immediate >=	TEST Immediate <>	
			6	LSL	LSR	ASL	ASR	DLSL	DLSR			CSL		SEB					
			7	LSL Immed	LSR Immed	ASL Immed	ASR Immed	DLSL Immed	DLSR Immed			CSL Immed		SEH					
Memory Reference Format	Code	Short	8	BR >		BR =	CALL	BR Immediate >	BR Immediate <	BR Immediate =	LOOP	BR <=		BR <>	BR	BR Immediate <=	BR Immediate >=	BR Immediate <>	
	Code	Long	9	BR >		BR =	CALL	BR Immediate >	BR Immediate <	BR Immediate =	LOOP	BR <=		BR <>	BR	BR Immediate <=	BR Immediate >=	BR Immediate <>	
	Data	Short	A	STOREB	STOREB X	STOREH	STOREH X			STORE	STORE X	STORED	STORED X						
	Data	Long	B	STOREB	STOREB X	STOREH	STOREH X			STORE	STORE X	STORED	STORED X						
	Data	Short	C	LOADB	LOADB X	LOADH	LOADH X			LOAD	LOAD X	LOADD	LOADD X					LADDR	LADDR X
	Data	Long	D	LOADB	LOADB X	LOADH	LOADH X			LOAD	LOAD X	LOADD	LOADD X					LADDR	LADDR X
	Code	Short	E	LOADB	LOADB X	LOADH	LOADH X			LOAD	LOAD X	LOADD	LOADD X					LADDR	LADDR X
	Code	Long	F	LOADB	LOADB X	LOADH	LOADH X			LOAD	LOAD X	LOADD	LOADD X					LADDR	LADDR X

X = Indexed (i.e., target address is further offset by a register named in the second operand field).
Immediate (Immed) = the second operand field contains a value.

Index

A

ACKIOI signal, 3-6
ACKIOM signal, 3-8
ACKMCIO signal, 3-2
ADD instruction, 8-4, 8-6
ALU, 2-8
AND instruction, 8-5, 8-6
Arithmetic traps, 6-7, 6-9
  before trap, 6-9, 6-11
  inexact result, 6-9, 6-14
  integer divide by zero, 6-9, 6-11
  integer overflow, 6-9, 6-10
  real divide by zero, 6-9, 6-12
  real overflow, 6-9, 6-13, 6-14
  real underflow, 6-9, 6-12, 6-14
ASL instruction, 8-12
ASR instruction, 8-12

B

Bandwidth, 2-10
Barrel shifter, 2-3, 2-8
Before trap, 4-7, 6-9, 6-11
Bit-oriented instructions, 8-10
Booth's recoded algorithm, 2-7
BR instruction, 8-14
Branch, conditional, 2-5
Branch instructions, 8-14
Branch prediction, 2-4, 8-14
  prediction bit, 2-5
  example, 2-5
  incorrect, 2-5
  logic, 2-3
Branch, unconditional, 2-7
Burst mode, 3-1
Bus contention, 3-11
Bus contention example, 3-12
Bus, I/O, 2-10, 3-1
Bus, processor, 2-1
Bypassing the RFM unit, 2-8
Byte operand, 4-1, 4-9

C

Cache, 2-3, 2-10
  code/data, 2-10
  flushing, 9-4
Cache cycle time, 2-10
Cache data table, 2-10
Cache hit, 2-11
Cache tag table, 2-10
CALL instruction, 8-16
CALLR instruction, 8-17
CBIT instructions, 8-10
CCB, 6-2
CCB address register, 5-4
CCB contents and SR states, 6-3
CCB table, 6-3
  data area, 6-3, 6-4
  interrupt offsets, 6-3
  trap offsets, 6-3
Check trap, 6-7, 6-8
CHK instruction, 8-18
CHKI instruction, 8-18
Clock board registers, 9-6
Code segment, maximum number addressable, 7-1
Code segment register, 5-4
Code segments, 5-1
Code/data cache, 2-10, 2-10
Compare instructions, 8-11
Conditional branch, 2-5, 8-14
Control store, 2-3, 2-3
Controlling processes, 5-1
CPU control block, 6-2
CPU memory bus, 2-10
Creating processes, 5-1
CREG instruction, 9-3, 9-6
CSL instruction, 8-12
Cycle time, 2-10
  cache, 2-10
  I/O bus, 2-10
  main memory, 2-10

D

Daisy-chain, 3-1
Daisy-chained I/O signals, 3-11
  dual, 3-11
Data alignment, 6-7
Data alignment violation, 6-7
Data in registers, 4-9
Data representation, 4-5
Data segment, maximum number addressable, 7-1
Data segment register, 5-4
Data segments, 5-1
Data sharing, 9-1
Data storage, 4-9
Data table, cache, 2-10

Data types, 8-1
DCOMP instruction, 8-11
Denormalized number, 4-6
DFIXR instruction, 8-9
DFIXT instruction, 8-9
DFLOAT instruction, 8-9
Direct address instructions, 4-3
Direct memory access, 2-10
DIRT instruction, 9-7
Dirty page, 9-7
Disabled interrupts, 5-1, 9-1
Disabling traps, 5-5, 6-9
DIV instruction, 8-4
Divide-by-zero trap, 4-7
DLSL instruction, 8-12
DLSR instruction, 8-12
DMA bandwidth, 2-10
DMA Logic, 2-10
DMA read operation, 3-9
DMA read/write, 3-8
DMA signals, 3-8
DMA transfer rate, 2-10, 3-1, 3-8
DMA write operation, 3-10
DN, 4-6
Double clocking, 2-7
Double precision real numbers, 4-6
Double real instructions, 8-9
Double words, 8-1
Double-bit error detection, 2-10
Double-bit parity error, 6-4, 6-5
    on instruction execute, 6-5
    on instruction fetch, 6-5
Double-word operand, 4-1, 4-9
DRADD instruction, 8-9
DRCOMP instruction, 8-11
DRDIV instruction, 8-9
DRMPY instruction, 8-9
DRNEG instruction, 8-9
DRSUB instruction, 8-9
Dual daisy-chained signals, 3-11

E

EADD instruction, 8-7
EDIV instruction, 8-7
ELOGR instruction, 9-3, 9-4
ELOGW instruction, 9-3, 9-4
EMPY instruction, 8-7
Enabling traps, 5-5, 6-9
Error correction, 2-1
Error correction logic, 2-10
ESUB instruction, 8-7
Event handling, 6-1
Example of bus contention, 3-12
Exceptions, 6-1
Executing floating point numbers, 2-8
Execution unit, 2-1, 2-8
Exiting trap, 9-5
Exponent ALU, 2-8
Extended integer instructions, 8-7
Extended precision arithmetic, 4-5
External interrupt, 6-4, 6-5
    disabling/enabling, 9-4

F

FIXR instruction, 8-8
FIXT instruction, 8-8
FLOAT instruction, 8-8
Floating point execution, 2-8
FLUSH instruction, 9-3, 9-4
Flushing, cache, 9-4
    TMT, 9-4
Flushing pipeline, 2-5
Flushing TMT table, 7-7
Format, TMT entry, 7-7
    VRT entry, 7-4
Four-way branching, 2-3

G

General registers, 2-7, 4-1, 4-2, 8-1
Guard bits, 4-9

H

Halfword operand, 4-1, 4-9
Hash table, 7-4

I

Idle count, 6-4
IEEE traps, 6-9
Illegal instruction, 6-7, 6-7
Immediate instructions, 8-6
Incorrect branch prediction, 2-5
Index register, 8-2
Indexed address instructions, 4-3
Inexact result trap, 6-9, 6-14, 6-14
INF, 4-6
Infinity, 4-6
Instruction, conditional branch, 8-14
    loop control, 8-15
    syntax, 8-1
    unconditional branch, 8-14
Instruction unit, 2-1, 2-3
Instruction addressing, 4-1
Instruction decoding, 2-3
Instruction formats, 4-1
Instruction pipeline, 2-3
Instruction prefetch, 2-3

Instruction types, 4-2
Instructions, bit-oriented, 8-10
  branch, 8-14
  compare, 8-11
  double real, 8-9
  extended integer, 8-7
  immediate, 8-6
  integer, 8-4
  I/O, 9-7
  kernel mode, 9-1
  load, 8-2
  load address, 8-3
  logical operator, 8-5
  maintenance, 9-3
  memory reference, 8-2
  program control, 8-14
  read, 9-7
  real, 8-8
  register, 8-4
  shift, 8-12
  sign extended, 8-13
  state switching, 9-1
  store, 8-3
  subroutine, 8-16
  test, 8-10
  virtual memory, 9-7
  write, 9-7
  Integer divide by zero trap, 6-9, 6-11
Integer instructions, 8-4
Integer overflow trap, 6-9, 6-10
Integer representation, 4-5
Interrupt, testing for, 9-5
Interrupts, 6-1
  defined, 6-1
  described, 6-4
  disabled, 5-1, 9-1
  double-bit parity error, 6-4
  external, 6-4
  instruction fetch page fault, 6-4
  power fail warning, 6-4
  reset, 6-4
  switch 0, 6-4
  timer, 6-4
I/O acknowledge signals, 3-11
I/O board priorities, 3-1, 3-11
I/O bus, 2-10, 3-1
I/O bus cycle time, 2-10
I/O interrupt read signals, 3-6
I/O interrupt read word, 3-6
I/O interrupts, 3-6
I/O operations, 3-2
I/O read address word, 3-3
I/O read data word, 3-3
I/O read operation, 3-3
I/O read/write, 3-2
I/O read/write signals, 3-2
I/O status, 9-8
I/O system, 3-1
I/O write address word, 3-4
I/O write data word, 3-4
I/O write operation, 3-4
IODACK signal, 3-2, 3-8
IODATA signal, 3-2, 3-6, 3-8
IODNVM signal, 3-2
IOIR word, 3-6
IOIREQ signal, 3-6
IOMREQ signal, 3-8
IOR operation, 3-2, 3-3
IORA word, 3-3
IORD word, 3-3
IOW operation, 3-2, 3-4
IOWA word, 3-4
IOWD word, 3-4
ITEST instruction, 9-3, 9-5

K

KCALL instruction, 5-1, 8-18
Kernel calls, 6-7
Kernel function, 6-1
Kernel mode, 5-1
Kernel mode instructions, 9-1
Kernel violation, 6-7, 6-8

L

LADDR instruction, 8-3
LCOMP instruction, 8-11
LDREGS instruction, 9-1
Link block address register, 5-4
Link word, 7-5
LOAD instructions, 8-2
Logical operator instructions, 8-5
Long displacement, 4-3
Long displacement format, 4-4
Loop control instruction, 8-15
LOOP instruction, 8-15
LSL instruction, 8-12
LSR instruction, 8-12
LUS instruction, 5-5, 6-1, 9-1

M

MACHINEID instruction, 9-3, 9-5
Maintenance instructions, 9-3
MAKEDR instruction, 8-9
MAKERD instruction, 8-8
Mantissa, 2-8, 4-5
Mapping hardware, 7-2
Maximum user configuration, 9-5
MCIOREQ signal, 3-2
MCIOW signal, 3-2
MDNVIO signal, 3-8

Memory controller, 2-10
Memory controller unit, 2-1
Memory cycle time, 2-1, 2-10
Memory instruction formats, 4-4
Memory mapping hardware, 7-2
Memory reference instructions, 4-3, 8-2
Microcode version, 9-6
Model number, 9-5
MOVE instruction, 8-5, 8-6, 9-1
MPY instruction, 8-4, 8-6
Multiplexed address/data, 2-10, 3-1
Multiplication logic, 2-7
Multiplier, 2-7
MULTIWD signal, 3-8

N

NaN, 4-6
NEG instruction, 8-4
Negative indexes, 4-4
NOP instruction, 8-5
NOT instruction, 8-5, 8-6
Not a number, 4-6
Number of addressable segments, 7-1

O

Opcode register, 5-3
Operand fetch cycle, 2-7
Operand memory layout, 4-9
Operand specifiers, 4-2
Operand types, 4-9
OR instruction, 8-5
Overflow trap, 4-7
Overlapped scanning, 2-7

P

Page fault, 6-4
Page fault, defined, 6-1
Parallel multiplication, 2-7
Parallelism, 2-4
PC register, 5-4
PCB, 5-5
PCB address register, 5-4
Physical page field, TMT, 7-7
Pipeline, 2-3
Pipeline flushing, 2-5
Pipeline interlock, 2-9
Pipeline stages, 2-3
Positive indexes, 4-4
Post normalization, 2-8
Power fail warning interrupt, 6-4, 6-6
Power-of-two exponent, 4-5
Prediction bit, 2-5
Prefetch buffer, 2-3
Prefetch buffers, 2-3
Priority signals, 9-4
Privileged process bit, 5-2, 5-5, 9-1
Process clock, 5-5, 6-4
Process control, 5-1
Process control block, 5-5
Process management, 5-1
Process time, incrementing, 6-4
Processor boards, 2-1
Processor bus, 2-1
Processor components, 2-1
Processor modes, 5-1
Program control instructions, 8-14
Program counter, 4-1, 8-1
Program counter register, 5-4

Q

Quadword, 2-10

R

RADD instruction, 8-8
RCOMP instruction, 8-11
RDIV instruction, 8-8
RDLOG instruction, 9-3, 9-6
READ instruction, 9-7
Real addressing, 9-1
Real divide by zero trap, 6-9, 6-12
Real instructions, 8-8
    double, 8-9
Real memory references, 5-1
Real number representation, 4-5
Real numbers, 4-5
    rounding, 4-9
    special, 4-6
Real overflow trap, 6-9, 6-13, 6-14
Real underflow trap, 6-9, 6-12, 6-14
Real-time devices, 9-4
Register bypass function, 2-9
Register bypass hardware, 2-8
Register file logic, 2-7
Register file/multiplier, 2-1
Register files, 2-3, 2-7
Register instruction format, 4-2
Register instructions, 4-2, 8-4
Register select logic, 2-3, 2-8
Register-Immediate instructions, 4-2
Register-to-register instructions, 4-2
Relocatable code, 4-4
REM instruction, 8-4
Reset interrupt, 6-4, 6-6
Result register, 2-3
RET instruction, 8-17
RFM unit, 2-7

RFx and RFy, 2-3
RFx and RFy files, 2-7
RISC defined, 1-1
RMPY instruction, 8-8
RNEG instruction, 8-8
Rounding, 2-8
Rounding real numbers, 4-9
RPx and RPy, 8-1
RSUB instruction, 8-8
RUM instruction, 6-2, 9-1
Rx field/segment register, 5-3
Ry field/effective address register, 5-4

S

SBIT instructions, 8-10
Scratch pad, 2-7
SEB instruction, 8-13
Segments, code and data, 5-1
SEH instruction, 8-13
Self-relocating code, 8-14
Serial number, 9-5
Shift instructions, 8-12
Short displacement, 4-3
Short displacement format, 4-4
Sign and exponent hardware, 2-8
Sign bit, 4-5
Sign extended instructions, 8-13
Single-bit error correction, 2-10
Special numbers, as operands, 4-7
Special numbers, as results, 4-7
Special real numbers, 4-6
Special registers, 2-7, 5-3
Special registers, states of on events, 6-3
State switching instructions, 9-1
Static prediction bit, 2-5
Store cycle, 2-7
STORE instructions, 8-3
SUB instruction, 8-4, 8-6
Subroutine instructions, 8-16
SUS instruction, 6-1, 9-1
Switch 0 interrupt, 6-4, 6-5
Syntax, instruction, 8-1

T

Tables, virtual-to-real translation, 7-2
Tag field, TMT, 7-7
Tag table, cache, 2-10
TBIT instructions, 8-10
TEST instructions, 8-10
Time of day counter, 6-4
Timeout error, 3-3, 3-5
Timer 1 count, 6-4
Timer 2 count, 6-4
Timer interrupts, 6-4, 6-6
TMT, 7-2
    flushing, 9-4
TMT table, 7-7
    entry format, 7-7
    flushing, 7-7
    physical page field, 7-7
    tag field, 7-7
    translation process, 7-7
TMT word, 7-5
TRANS instruction, 9-7
Transfer rate, DMA, 3-8
Transferring data, 4-3
Translation mapping table, 2-10, 7-2, 7-7
Translation process, TMT, 7-7
Trap instruction, 6-7, 6-8
TRAP instruction, 8-18, 8-18
TRAPEXIT instruction, 9-3, 9-5
Traps, 6-1, 6-7
    arithmetic, 6-7
    before, 4-7
    check, 6-7, 6-8
    data alignment, 6-7, 6-7
    defined, 6-1
    divide-by-zero, 4-7
    illegal instruction, 6-7, 6-7
    instruction execute page fault, 6-7
    kernel calls, 6-7, 6-7
    kernel violation, 6-7, 6-8
    overflow, 4-7
    trap instruction, 6-7, 6-8
    underflow, 4-7
Traps word, 5-5, 6-9
Traps word register, 5-4
Two's complement signed integers, 4-5
Two's-complement notation, 4-4
Two-way set associative, 2-10
TWRITED instruction, 9-3, 9-4

U

Unconditional branch, 2-7, 8-14
Underflow trap, 4-7
Unsigned integers, 4-5
User configuration, maximum, 9-5
User mode, 5-1, 5-2
    instructions, 8-1

V

VERSION instruction, 9-3, 9-6
Virtual address, bits, 2-11
    format, 7-1
    offset, 7-1
    page number, 7-1
    translation, 9-7
Virtual addresses, 4-1

Virtual memory instructions, 9-7
Virtual memory management, 7-1
Virtual memory references, 5-2
Virtual-to-real translation, 2-1, 2-10, 5-2, 7-1
    bypassing, 5-1
    hardware, 7-2
    sequence, 7-3
    table, 7-2, 7-4
    tables, 7-2
VPN, 7-1
VRT hash table, 7-4
VRT hash table address, 7-5
VRT mask register, 5-4
VRT, searching, 9-7
VRT table, 7-2, 7-4
VRT table, address register, 5-4
    entry format, 7-4
    hash table address, 7-5
    link word, 7-5
    location of, 7-4
    referenced bit, 7-5
    searching, 7-4, 7-5
    TMT word, 7-5
VRT translation process, 7-5

W

Word operand, 4-1, 4-9
WRITE instruction, 9-7
Write-through, 2-10

X

XOR instruction, 8-5

Z

Zero, as special number, 4-6