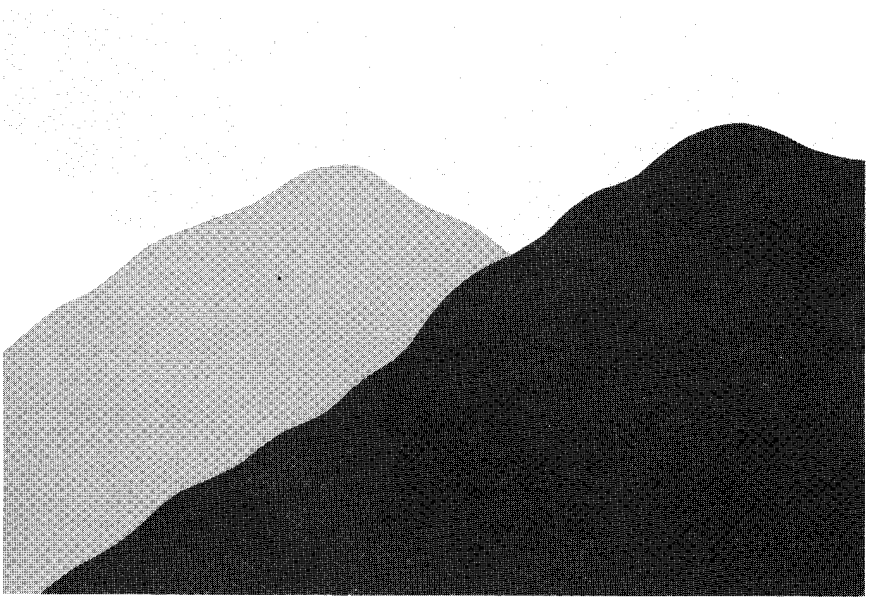


Processor Reference Manual

RIDGE

9008

Ridge Processor
Reference Manual

Fourth Edition: 9008-C (SEP 85)

PUBLICATION HISTORY

Manual Title: Ridge Processor Reference Manual

First Edition: 9008 (MAR 83)
Second Edition: 9008-A (FEB 84)
Third Edition: 9008-B (JUN 84)
Fourth Edition: 9008-C (SEP 85)

NOTICE

CONTENTS

Chapter 3: REGISTER FORMAT INSTRUCTIONS

Chapter 4: PROGRAM CONTROL INSTRUCTIONS

ILLUSTRATIONS

Chapter 1
OVERVIEW

This manual provides a general description of the processors available on Ridge 32 computers.

The Ridge 32 computer is an engineering workstation with a 32-bit, high performance processor implemented in MSI and LSI bipolar logic. Ridge processors have a simple, general purpose, microcoded architecture that incorporates paged virtual memory. The processing power of a Ridge 32 computer is equal to medium performance mainframes and high performance minicomputer systems.

KEY FEATURES

- Reduced Instruction Set Computer (RISC) Architecture
- 125-nanosecond Processor Cycle Time
- 375-nanosecond Memory Cycle Time
- 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

V1 AND V2 PROCESSORS

Two processors are available for the Ridge 32:

V1 processor The original, Version 1 Ridge processor.

V2 processor An upgraded version of the V1 with special hardware that increases the speed of floating point calculations by 20 - 100 percent.

The V1 and V2 processors utilize a register-oriented design incorporating 16 general registers. Virtual addressing is accomplished using 4096-byte pages within a four-gigabyte address space. Simple instructions can be completed in one 125-nanosecond machine cycle, resulting in a maximum instruction rate of eight-million instructions per second (8 MIPS).

RISC ARCHITECTURE

Both the V1 and V2 processors are based on RISC (*Reduced Instruction Set Computer*) architecture. 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 results in a very fast and low-cost computer.

RISC architecture is characterized by the following:

Simple addressing modes. The Ridge 32 uses only three modes which reduces the amount of logic needed to perform memory references.

Simple instruction formats. The Ridge 32 uses three instruction formats that can be decoded with a minimum of logic.

Separated code and data. The Ridge 32 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. These languages tend to generate short sequences of the required functions. Complex instructions and instructions not used by a compiler are eliminated. Thus, the Ridge 32 instruction set offers the "primitives" which will be assembled by a compiler.

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. To do otherwise complicates the compiler and may slow the overall operation of the machine.

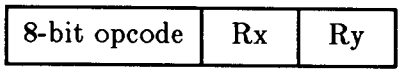
Linear address space. Code and data space are each linear with a byte-addressable area that is four-gigabytes long. Segmentation schemes appear to save logic to support the full 32-bit address widths, but instead they complicate the hardware and compilers, and slow the processor's performance.

General registers. All registers are available for use as data, indexing, and addressing. If registers are specialized, they complicate compilers, reduce the available fast storage area, and increase code size when data must be moved to the appropriate register type.

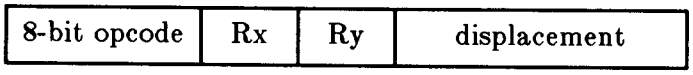
INSTRUCTION FORMATS

The V1 and V2 processors contain 16 32-bit registers. The two-operand instruction set uses three instruction formats. The instruction formats are register-to-register (16-bits long), short displacement memory address (32-bits long), and long displacement memory address (48-bits long). The instruction formats are shown in Figure 1-1.

Register-to-register



Short displacement memory address



Long displacement memory address

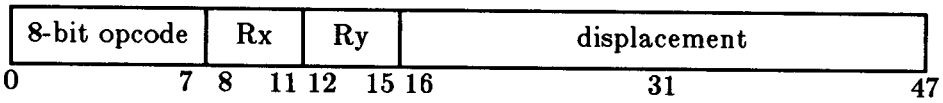


Figure 1-1. Instruction Formats

All instructions use an eight-bit opcode followed by two four-bit operands. The first operand always names a register or a register pair. The second operand names a register or is a four-bit constant. Instructions exist to operate on registers, load from memory, store to memory, and transfer program control.

The register-to-register format is used for instructions that operate on the contents of one or two registers and do not address memory. The short and long displacement memory address format instructions are used for memory-addressing instructions, such as storing and loading. The short displacement memory address format is used for referencing addresses that can be specified in 16 bits. The long displacement memory address format is used for referencing addresses that must be specified in 32 bits.

Any arithmetic or address operation can be performed on any register. Registers are not specialized for counting or indexing.

PROCESSOR ARCHITECTURE

A general model of the architecture used by the Ridge processors is shown in Figure 1-2. The user-visible features of the processor are instructions, general registers, and the program counter. Instructions operate on the general registers (register-to-register) or on a register and a memory location (load from memory or store to memory). The program counter is visible when using program control instructions such as subroutine call and branch.

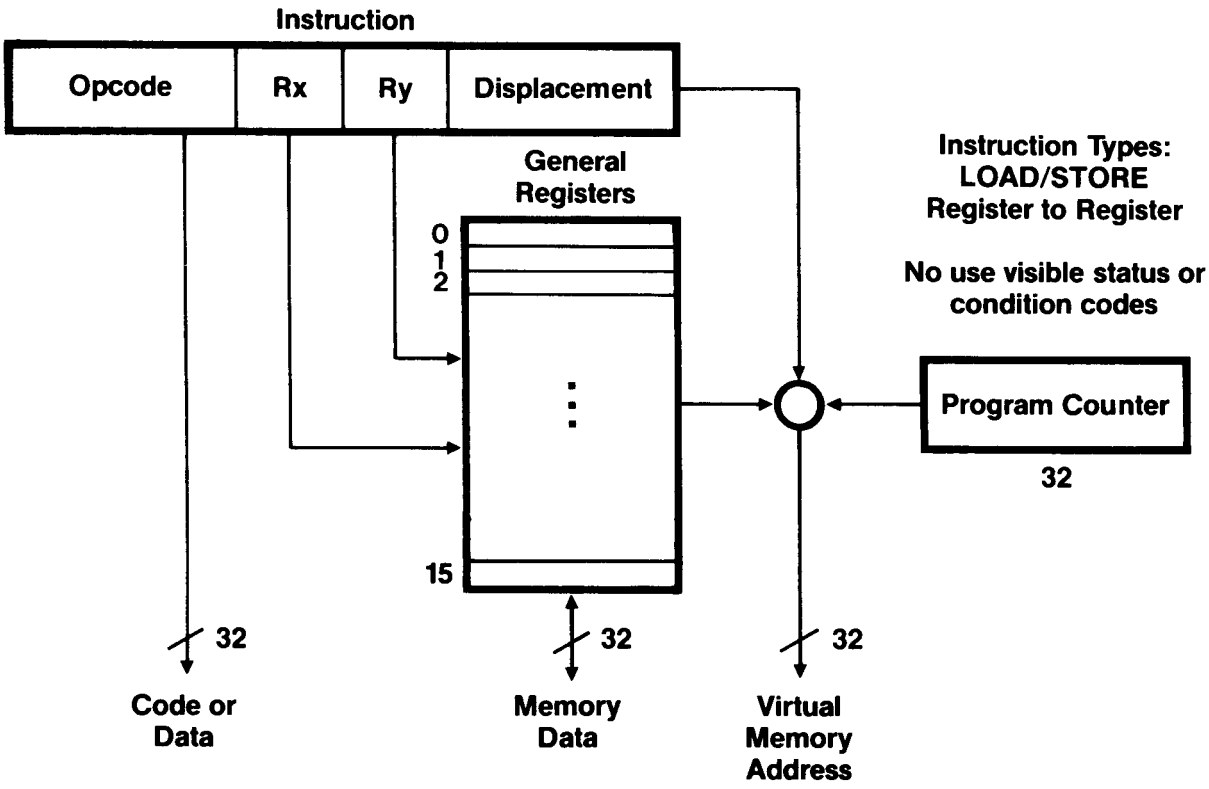


Figure 1-2. Model of Processor Architecture

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. An individual program may access a maximum of four gigabytes of code space and a maximum of four gigabytes of data space.

A status register containing condition codes is purposely missing from this architecture. Status registers complicate and tend to slow down high-speed processors. On high-speed machines several instructions are in various stages of execution at any given moment. Condition codes tend to be generated at various times during these stages and must be properly propagated from stage to stage. In virtual machines, there is the additional problem of preserving the condition codes throughout the stages when an instruction aborts due to a page fault.

The processor architecture includes the conditional branch instruction that obviates the need for condition codes. This instruction combines the compare function and the conditional branch instruction. The compare function generates the condition code and the conditional branch instruction changes program flow of control based upon condition code values.

INTERNAL STRUCTURE

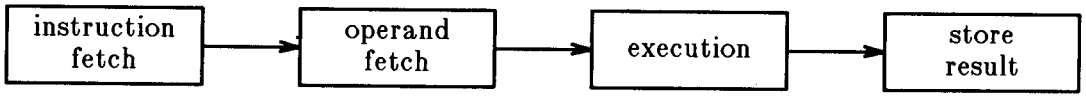
Both the V1 and V2 processors consist of two printed circuit boards. The first is the instruction fetch unit and the second is the execution unit. A private bus to the memory controller provides separate 32-bit address and data lines. The instruction fetch unit and execution unit may each independently access main memory. Memory cycle time is 375 nanoseconds, which includes virtual-to-real memory translation and error correction.

A block diagram of the V1 processor, memory, and I/O system are shown in Figure 1-4. A similar block diagram for the V2 processor is shown in Figure 1-5. In the following text, the items in **bold type** are illustrated in both figures.

PIPELINED ORGANIZATION

The V1 and V2 processors uses a pipelined organization. The pipeline is composed of four stages: instruction fetch, 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.

Pipeline Stages



The operations performed during each processor cycle are as follows:

Instruction Fetch. The instruction is fetched from the *prefetch buffer*. The *opcode* is used as an index into the *control store*, which controls instruction execution. The *Rx* and *Ry* operands in the instruction are used to enable the *register select logic*.

Operand Fetch. Rx and Ry are fetched from the *register files*.

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

Store Result. The data is moved from the result register into the Rx and Ry register files.

The purpose of the pipeline is to increase machine speed by using parallelism. Each stage of the pipe operates on a separate instruction. Instructions flow through each of the four stages of the pipe, one cycle at a time. Although complete execution of an instruction takes four machine cycles, one instruction completes each cycle, thus creating an effective processor speed that is four times the speed of a non-pipelined operation. The instruction pipeline includes all of the logic on the execution unit and part of the logic on the instruction fetch unit.

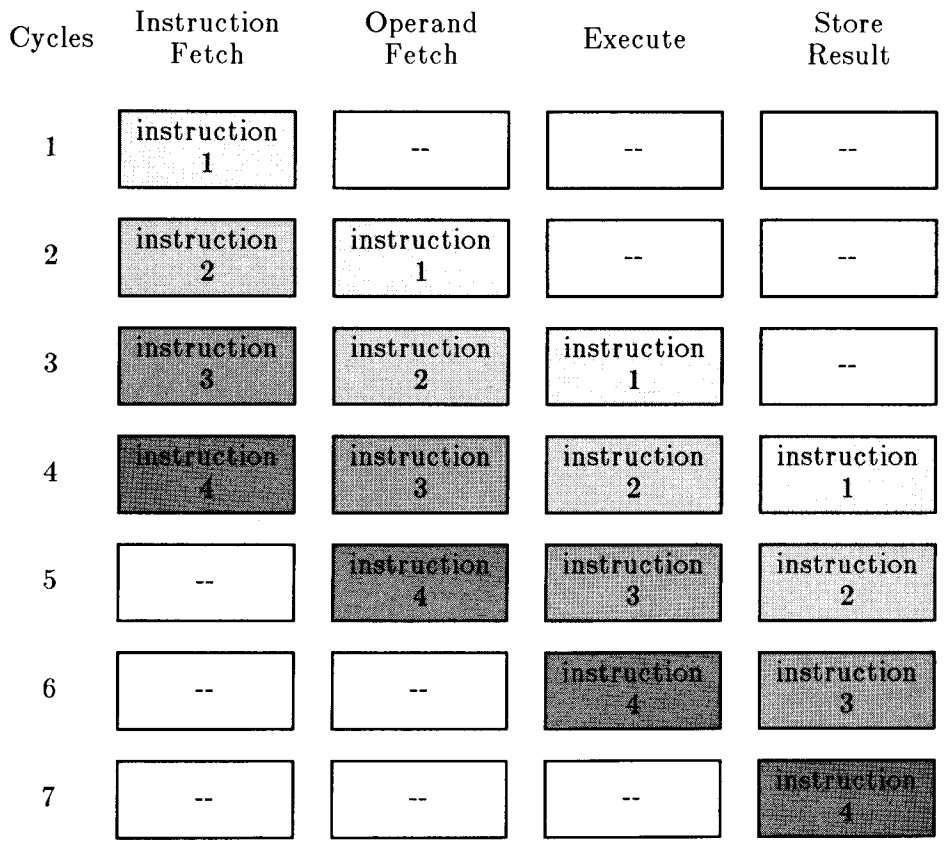


Figure 1-3. Instruction Flow Through Pipeline Stages

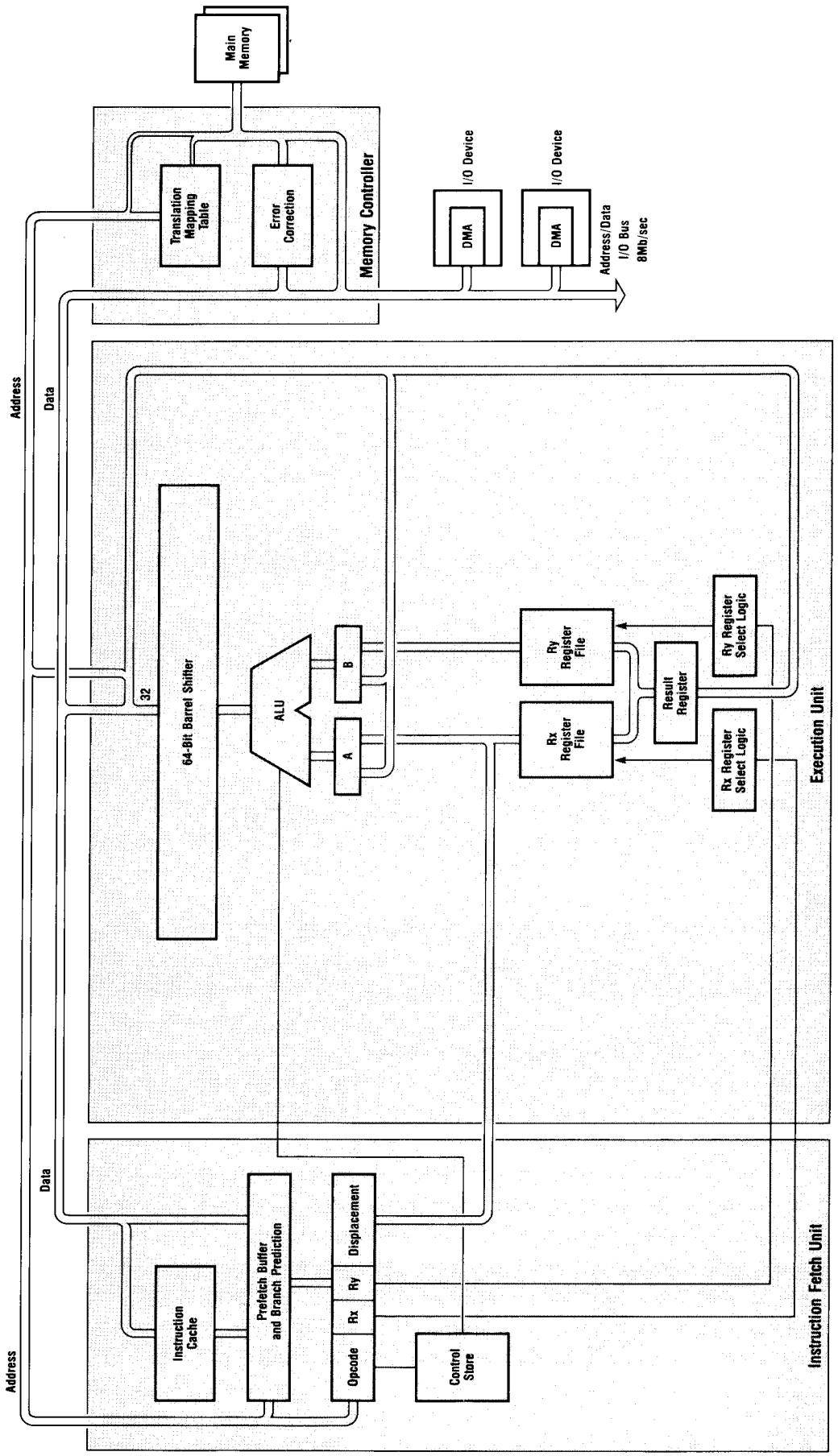


Figure 1-4. Internal Structure of V1 Processor

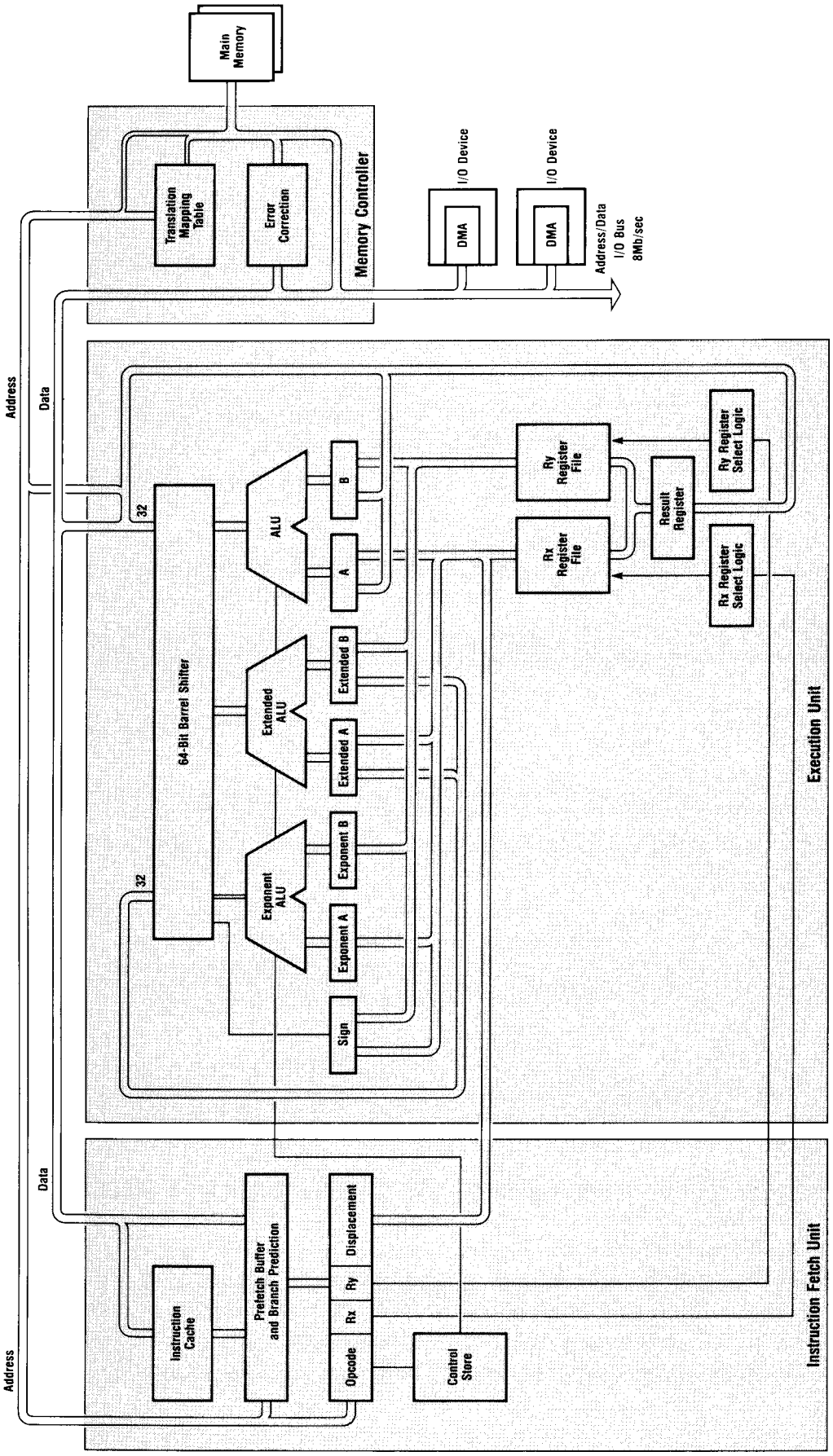


Figure 1-5. Internal Structure of V2 Processor

INSTRUCTION FETCH UNIT

The instruction fetch unit performs instruction prefetch and decoding. It contains a 256-byte *instruction cache* and a maximum of 4096 words of 48-bit wide *control store*. The instruction fetch unit fetches instructions from the instruction cache or main memory ahead of the execution unit and stores them in its eight-byte **prefetch buffer**.

Branch Prediction

The implementation of branch instructions is critical to the performance of pipelined machines. Without special handling, 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 uses a technique to load the instruction into the pipe, which is the most likely result of the branch, thus reducing the chance that the pipeline is 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 fetch 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 fetch unit prefetches 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 fetch 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 fetch 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 fetch unit must flush this instruction and fetch the next sequential instruction, which will then be executed.

This flushing of the instruction pipeline causes a two- to four-cycle delay for the incorrectly predicted conditional branch instruction. Measurements have shown this to be infrequent, and consequently program speed is 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 AS 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
        STORE  R0,I              ; Store value of R0 at I
```

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

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

Unconditional Branch Instructions

Unconditional branch instructions also make use of the branch prediction and prefetch logic in the instruction fetch unit. In unconditional branches, the instruction is decoded, the target location is fetched and placed in the instruction stream, and the unconditional branch is flushed from the prefetch buffer. This effectively removes the unconditional branches from the program entirely, and if the instruction fetch unit is ahead of the execution unit, unconditional branches can be performed with zero instruction time.

EXECUTION UNIT

The execution unit contains the general registers and is responsible for instruction execution. The arithmetic logic units *(ALU)* and *barrel shifter* work in close association with the execution unit. The barrel shifter is a hardware device that can shift any number of bits left, right, or circularly in a single clock cycle. The V1 processor has a 32-bit barrel shifter and the V2 processor has a 64-bit barrel shifter.

The general registers are found in the *Rx register file.* A duplicate copy of the registers is contained in the *Ry register file.* Duplicating the registers allows both Rx and Ry to be accessed in a single clock cycle.

The general data flow through the execution unit for numbers is as follows. Data is fetched from the Rx and Ry register files, operated on by the ALU, temporarily stored in the *result register* and then stored in the register files. Should data not yet stored in the register files be needed in a computation, the *register select logic* may bypass the register file and use the data on the bus as input to the ALU.

On the V2 processor illustrated in Figure 1-5, floating point numbers are unpacked and sent to the sign and exponent hardware and to the exponent ALU. This makes execution of floating point instructions more efficient. In addition to the exponent ALU and its hardware, double precision floating point numbers make use of the extended ALU for a 64-bit data path. The barrel shifter packs (reassembles) the results from the three ALU's and the sign hardware into floating point values.

The following is an example of a two-instruction sequence that utilizes the register bypass data path in the execution unit. This bypass avoids the "pipeline interlock" delay that results when an instruction's operand is dependent on an instruction still in the pipe. The example also illustrates the use of the instruction pipeline shown in Figure 1-3.

ADD R6, R7 (operation: R6 is added to R7 and the sum is put in R6.)

AND R5, R6 (operation: R5 logically ANDs with R6 and the result is put in R5.)

Instruction Pipeline Stage Operation

Cycle Clock	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 and R6 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 register file 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.

MEMORY CONTROLLER

The memory controller provides virtual-to-real address translation and *error correction* while handling all memory data for the processor and I/O devices. All memory accesses from the processor are virtual and go through the *translation mapping table* where they are converted to real addresses and presented to *main memory.* I/O devices on the *I/O bus* use real addresses and bypass the translation mapping table.

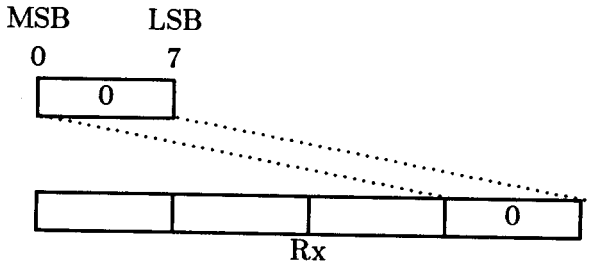
Main memory cycle time is 375 nanoseconds, and the memory controller processes four bytes per cycle. The *CPU memory bus* runs at full memory speed giving this bus a bandwidth of 10.7 megabytes per second. The I/O bus uses multiplexed *address* and *data* lines to minimize the use of connector pins on I/O boards. The I/O bus cycles in 500 nanoseconds and provides eight megabytes per second of direct memory access *(DMA)* bandwidth for I/O devices. Each board on the I/O bus contains its own DMA logic.

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

DATA TYPES

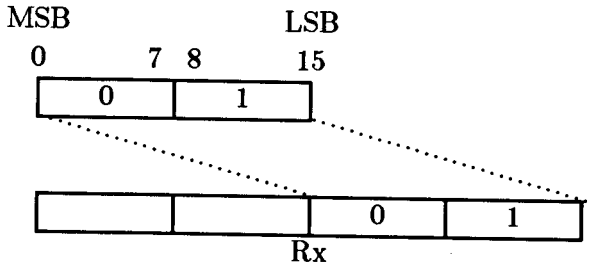
Both processors have instructions to load and store four different sizes of operands. The basic addressable unit is the 8-bit byte. The other operand sizes are the halfword (16-bits), the word (32-bits) and the double word (64-bits). The illustrations below give the notation and memory layout for each type of operand. Below each operand is an illustration showing how that operand is stored in registers.

Byte

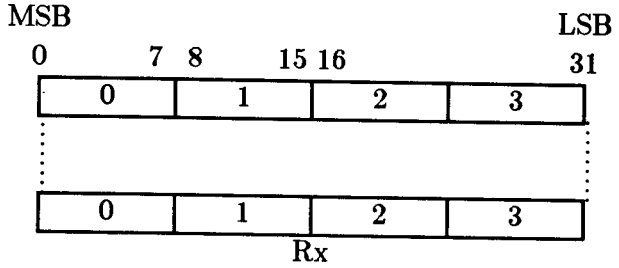


MSB = most significant bit
LSB = least significant bit

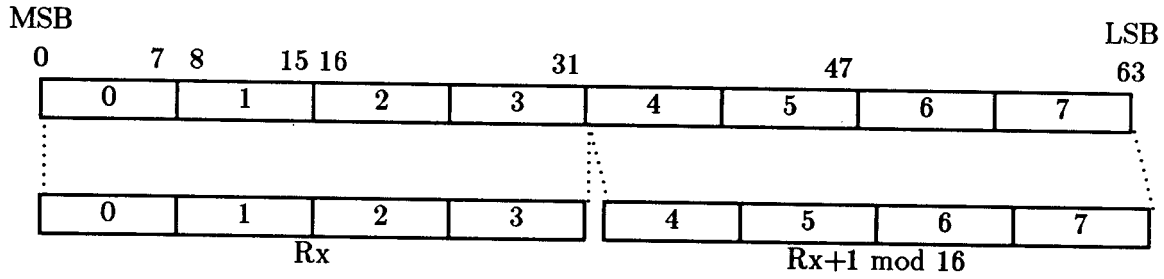
Half-Word



Word



Double-Word



There are instructions that manipulate registers as 32-bit and 64-bit data types. The three 32-bit data types are: two's complement signed integers, unsigned integers, and real numbers. The 64-bit data types consist of 64-bit unsigned integers, double precision real numbers, and 64-bit sets. Integer data types longer than 32 bits may be manipulated using extended precision integer arithmetic instructions.

INTEGERS

Integers are represented in two's-complement form and are in the range -2147,483,648 to 2,147,483,647, or unsigned in the range 0 to 4,294,967,295. The MSB of any data type is referred to as the sign bit, as shown below.

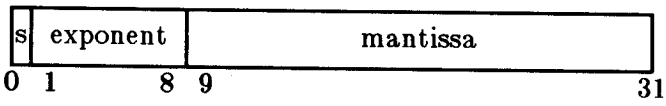
s = sign bit

REAL NUMBERS (SINGLE PRECISION)

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.



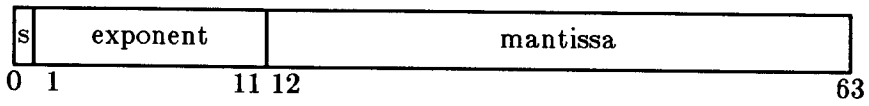
example: *1* = $0\ 01111111\ 00000000000000000000000_2 = 3F80\ 0000_{16}$

example: *-10* = $1\ 10000010\ 01000000000000000000000_2 = C120\ 0000_{16}$

REAL NUMBERS (DOUBLE PRECISION)

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* = 0 01111111111 00000000000000...000000000000$_2$ = *3FF*0 0000 0000 0000$_{16}$

example: *-10* = 1 10000000010 01000000000000...000000000000$_2$ = *C*024 0000 0000 0000$_{16}$

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.

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.

The program counter is referred to as PC. 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.

NAME OF INSTRUCTION OR INSTRUCTION CLASS

Instruction Summary:

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

Operation:

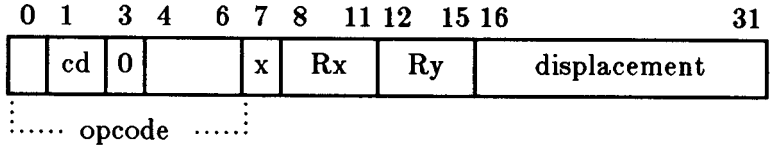
The TYP instruction has no operation; it is an example of syntax conventions.

Chapter 2
MEMORY REFERENCE INSTRUCTIONS

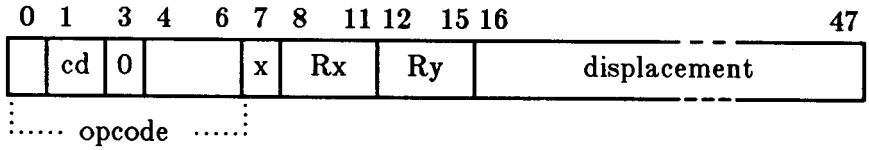
INSTRUCTION FORMATS

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

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 = indexed

The Ridge 32 processors have two addressing modes, direct and indexed. These modes may be used in accessing either code or data space with either short or long displacement memory address formats. One bit of the opcode is used to specify that the instruction is indexed, another bit is used to specify long displacement, and another two bits in combination indicate code or data space.

The 32-bit short displacement memory address format instructions have a 16-bit displacement field which is sign extended to a full 32 bits. The 48-bit long displacement memory address format instructions have a 32-bit displacement field.

The effective address for a memory reference instruction is calculated as follows.

Address Space	Indexed	Effective Address
Data	No	Displacement
Data	Yes	Ry + displacement
Code	No	PC + displacement
Code	Yes	PC + Ry + displacement

Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruction. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

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.

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 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 relocatable code.

All addressing formats have the same instruction execution time. Instructions referencing data space optionally add Ry to the displacement as the address is presented to memory. Instructions referencing code space optionally add Ry to the precomputed PC + displacement. The fetch unit contains logic that performs this function as part of the instruction prefetch.

INSTRUCTION DESCRIPTIONS

Descriptions of load, store, and load address memory instructions follow. 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 Double Word	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 Word	Double	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.*

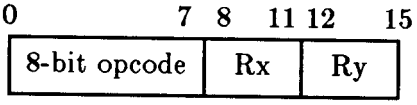
If the constant is code-relative, LADDR either loads register Rx with PC + *constant* or loads register Rx with the sum of the contents of Ry and PC + *constant.*

Chapter 3
REGISTER FORMAT INSTRUCTIONS

INSTRUCTION FORMAT

Register-to-register format instructions process data taken from a specified general register. These instructions use the register-to-register instruction format shown below. Generally, two registers are specified and the result usually replaces Rx.

Register-to-register



A few register-to-register format instructions also have an immediate mode. In immediate mode the 4-bit value of the Ry register field is used to specify an integer in the range from 0 to 15.

INSTRUCTION DESCRIPTIONS

The following pages describe the register-to-register format 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

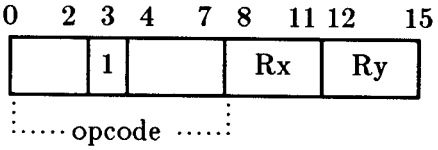
Instruction Summary:

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 register contents of Ry. The integer and logical immediate register-to-register instruction format is shown below.

Register-to-register



The Ry field is treated as a 4-bit integer constant.

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
		Ry	← the 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 and preserves the sign bit.

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.

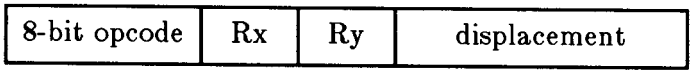
The SEH instruction makes bits 0-15 in register Rx the same as bit 16 in register Ry. Bits 16-31 in Ry are copied to Rx.

Chapter 4
PROGRAM CONTROL INSTRUCTIONS

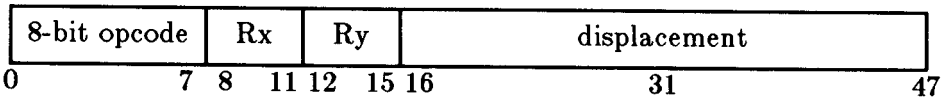
BRANCH INSTRUCTION FORMAT AND DESCRIPTION

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

Short displacement memory address



Long displacement memory address



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 DESCRIPTIONS

The following pages describe the branch instructions.

BRANCH INSTRUCTIONS

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

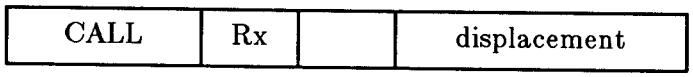
There are three subroutine call and return instructions: call subroutine, call subroutine register and return from subroutine.

CALL SUBROUTINE INSTRUCTION

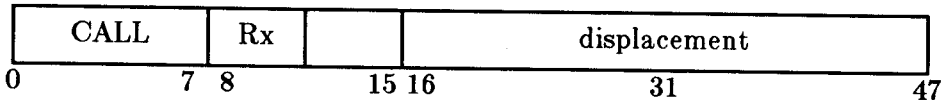
Instruction Format:

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



Long displacement memory address



Instruction Summary:

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

Operation:

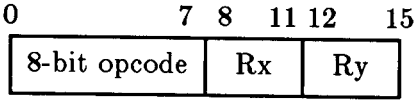
The call instruction places the address of the next instruction in Rx and transfers execution to the target location (PC + displacement). Short displacement memory addresses are sign extended. Like the branch instructions, the call instruction uses program counter (PC) relative addressing, which allows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

CALL SUBROUTINE REGISTER AND RETURN INSTRUCTIONS

Instruction Format:

The CALLR and RET instructions use the register-to-register instruction format shown below.

Register-to-register



Instruction Summary:

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

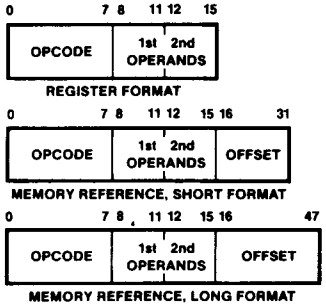
Operation:

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.

Appendix A: RIDGE OPCODE MAP

INSTRUCTION FORMATS:



Register Format

Memory Reference Format

	Segment Referenced	Format Length
	Code	Short
	Code	Long
	Data	Short
	Data	Long
	Data	Short
	Data	Long
	Code	Short
	Code	Long

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

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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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 <	=	CALLR	TEST Immediate >	<	=	RET	< =	TEST > =	<>	KCALL	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					
8	> BR >		= BR =	CALL	> BR Immediate >	<	=	LOOP	< = BR < =		<> BR <>	BR	< = BR Immediate < =	> =	<>	
9																
A	STOREB X		STOREH X				STORE X		STORED X							
B	X		X				X		X							
C	X		X				X		X							X
D	LOADB X		LOADH X				LOAD X		LOADD X							LADDR X
E	X		X				X		X							X
F	X		X				X		X							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.

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956