CRAY
RESEARCH, INC.

CRAY COMPUTER SYSTEMS

CRAY-2 COMPUTER SYSTEM FUNCTIONAL DESCRIPTION

HR-2000

Revision	Description
	May 1985 - Original printing.

PREFACE

This publication describes the functions of the CRAY-2 Computer System and the CRAY Assembly Language (CAL) Version 2 symbolic machine instructions specifically used with this machine. It is written to assist programmers and engineers and assumes a familiarity with digital computers and assemblers.

The manual describes the overall computer system including its configuration and characteristics. It also describes the operation of the Common Memory, Foreground Processor, and Background Processors. Both the machine code and the associated symbolic machine instructions are explained.

Site planning information for the CRAY-2 Computer System is available in the CRAY-2 Site Planning Reference Manual, publication HR-2001.

Additional information on the CRAY Assembly Language (CAL) Version 2 is available in the CAL Assembler Version 2 Reference Manual, publication SR-2003.

//

WARNING

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

//

CONTENTS

APPENDIX SECTION

FIGURES

1. INTRODUCTION

The CRAY-2 computer is a powerful, general-purpose computer system with extremely high processing rates. Scalar and vector capabilities in a multiprocessing environment combined with integrated foreground processing achieve these high rates.

1.1 CRAY-2 FEATURES

The CRAY-2 mainframe contains four independent Background Processors, each more powerful than a CRAY-1 processor. Featuring a clock-cycle time faster than any other computer system available, each of these processors offers exceptional scalar and vector processing capabilities. The four Background Processors can operate independently on separate jobs or concurrently on a single problem. The very high speed Local Memory integral to each Background Processor is available for temporary storage of vector and scalar data.

Common Memory is one of the most important features of the CRAY-2. It consists of 256 million 64-bit words randomly accessible from any of the four Background Processors and from any of the high-speed and common data channels. The memory is arranged in quadrants with 128 interleaved banks. All memory access is performed automatically by the hardware. Any user may use all or part of the memory not being used by the operating system.

Control of network access equipment and the high-speed disk drives is integral to the CRAY-2 mainframe hardware. A single Foreground Processor coordinates the data flow between the system's Common Memory and all the external devices across four high-speed I/O channels. The synchronous operation of the Foreground Processor with the four Background Processors and the external devices provides a significant increase in data throughput.

The most important CRAY-2 features are:

- Extremely large directly addressable Common Memory
- Fastest cycle time available in a computer system
- Scalar, vector, and multiprocessing combined in one system
- Integral Foreground Processor

- Elegant architecture
- Extremely high reliability
- High density memory chips and extremely fast silicon logic chips
- Liquid immersion cooling

1.1.1 PHYSICAL CHARACTERISTICS

The CRAY-2 mainframe is elegant in appearance as well as in architecture (see figure 1-1). The memory, computer logic, and DC power supplies are integrated into a compact mainframe composed of 14 vertical columns arranged in a 300° arc.

The upper part of each column contains a stack of modules and the lower part contains power supplies for the system. Total cabinet height, including the power supplies, is 45 inches; the diameter of the mainframe is 53 inches. Thus, the "footprint" of the mainframe is a mere 16 square feet of floor space.

An inert fluorocarbon liquid circulates in the mainframe cabinet in direct contact with the integrated circuit packages. This liquid immersion cooling technology allows for the small size of the CRAY-2 mainframe and is thus largely responsible for the high computation rates.

Significant CRAY-2 physical characteristics are:

- Occupies only 16 sq ft of floor space
- Stands 45 inches high (diameter is 53 inches)
- Contains 14 columns arranged in a 300° arc
- Contains 3-dimensional modules
- Contains liquid immersion cooling
- Contains chilled water heat exchange

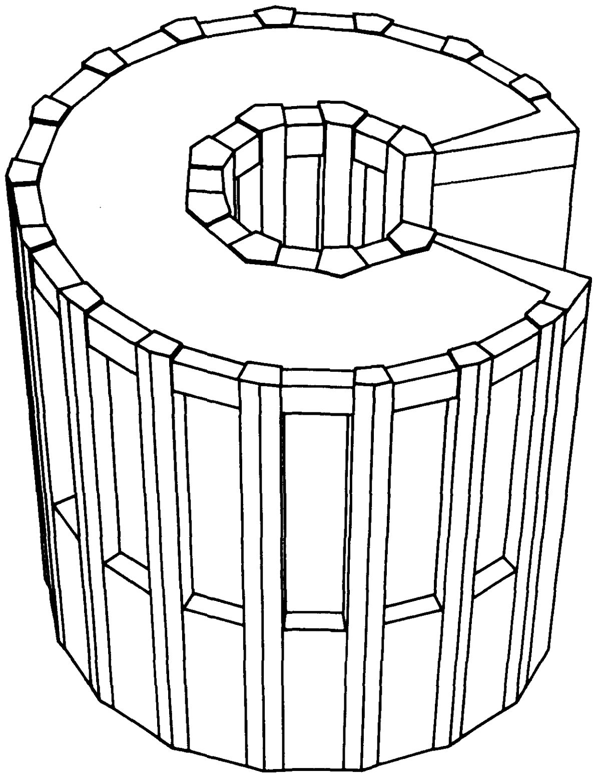


Figure 1-1. CRAY-2 Mainframe

1.1.2 ARCHITECTURE AND DESIGN

In addition to the cooling technology, the extremely high processing rates are achieved by a balanced integration of scalar and vector capabilities and a large Common Memory in a multiprocessing environment.

Significant architectural components of the CRAY-2 Computer System include the following:

- Four independent Background Processors capable of vector and scalar operation. Synchronization of the Background Processors is achieved through the Foreground Processor and semaphore flags in the Background Processors.
- 256 megawords of dynamic Common Memory
- A foreground system that controls and monitors system operation, including:
  - A Foreground Processor for system supervision
  - Four high-speed synchronous communication channels
  - Up to 40 I/O Devices
  - Disk controllers to control up to 36 disk storage units
  - Four Common Memory ports for data transfer
  - Four Background Processor ports to allow Foreground Processor control
  - Front-end Interfaces (from one to as many as four per channel)

The four identical Background Processors each contain registers and functional units to perform both vector and scalar operations. The single Foreground Processor supervises the four Background Processors. The large Common Memory complements the processors and provides architectural balance, thus assuring extremely high throughput rates (see figure 1-2).

On-site maintenance is possible via the maintenance control console.

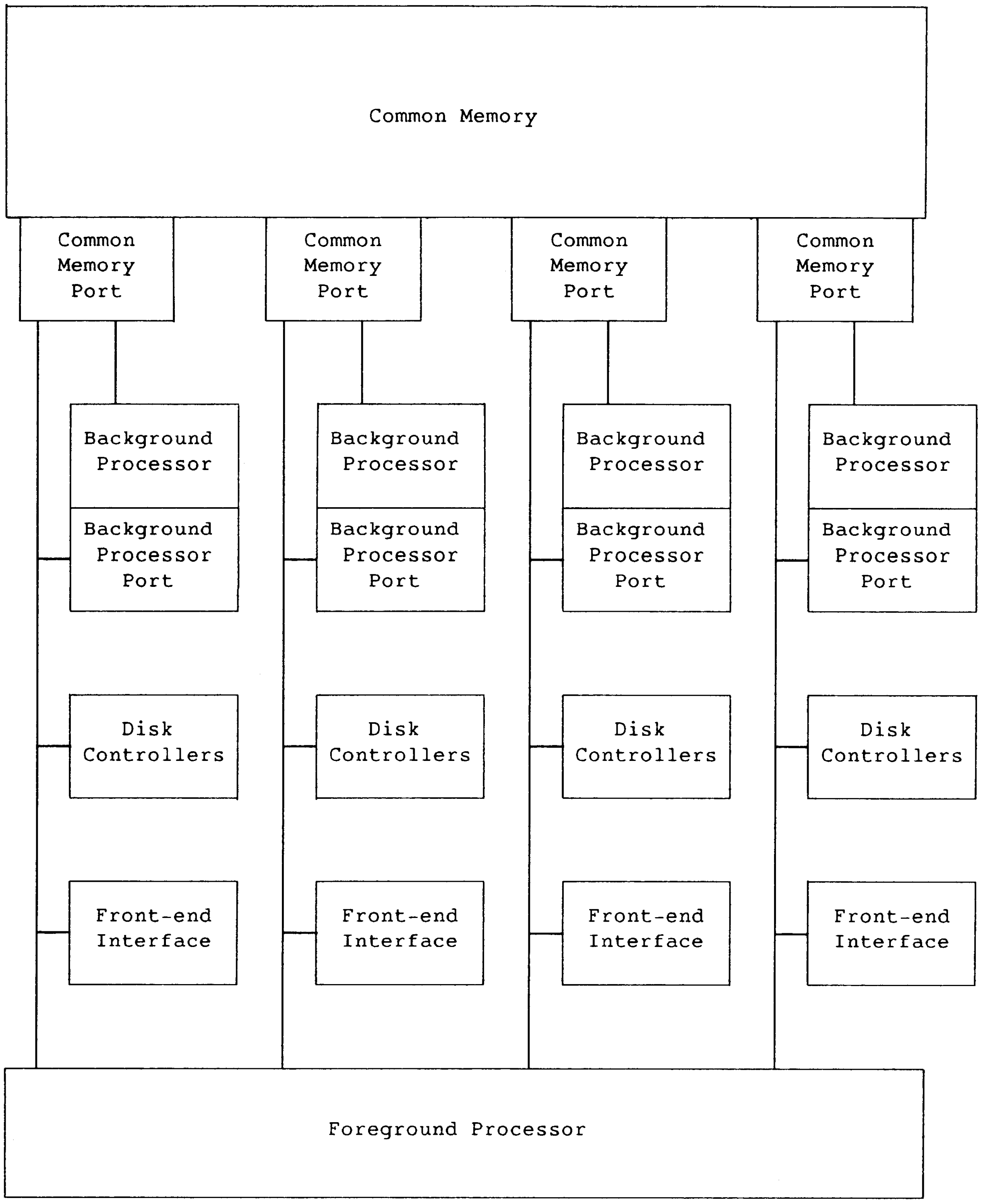


Figure 1-2. CRAY-2 Mainframe Configuration

1.2 CONVENTIONS

The following conventions are used in this manual.

Convention	Description
lowercase italics	Variable information
CP	Clock period
(S1),(S2), etc.	The contents of registers S1, S2, etc.
A, a, S, s, V, v register designators	For example, "Transmit (a_k) to s_i" means "Transmit the contents of the A register specified by the *k* designator to the S register specified by the *i* designator".
Register bit designators	Numbered right to left as powers of 2, starting with 2^0. Bit 2^{63} of an S or V register value represents the most significant bit. Bit 2^{31} of an A register value represents the most significant bit. The Vector Mask register has 64 bits, each corresponding to a word element in a Vector register. Bit 2^{63} corresponds to element 0, bit 2^0 corresponds to element 63.

Unless otherwise indicated, numbers in this manual are decimal numbers. Octal numbers are indicated with an 8 subscript. Exceptions are register numbers, channel numbers, instruction parcels in instruction buffers, and instruction forms which are given in octal without the subscript.

1.3 MANUAL DESCRIPTION

Section 1	Contains the introduction to this manual
Section 2	Describes the CRAY-2 Background Processor. The registers, functional units, and algorithms used are described.

Section 3	Provides detailed information on the CAL instructions that operate on the CRAY-2. Each machine instruction can be represented symbolically in CRAY Assembly Language (CAL) Version 2. The instructions are listed octally in a box format that provides the CRAY Assembly Language (CAL) Version 2 syntax format, an operand if required, a brief description of each instruction, and the machine instruction. Following the boxed information is a detailed description of the instruction and an example using the instruction.
Section 4	Describes the CRAY-2 Common Memory, phased memory access, and single error correction/double error detection (SECDED)
Section 5	Describes the CRAY-2 foreground system, which handles the I/O
Appendix A	Lists the symbolic machine instructions by function. The octal machine code may be used as an index to refer to section 3 for a detailed description of the instruction.

2. BACKGROUND PROCESSOR

The CRAY-2 computer contains four identical Background Processors. Each Background Processor contains operating and vector control registers and functional units to perform both vector and scalar operations. The Foreground Processor supervises the four Background Processors.

A Background Processor performs arithmetic and logical calculations. These operations, and the other functions of a Background Processor, are coordinated through the control section.

Control and data paths for one Background Processor are shown in figure 2-1.

2.1 CONTROL SECTION

Each Background Processor contains an identical, independent control section of registers and instruction buffers for instruction issue and control. The following control mechanisms are described in this section.

- Instruction issue and control
- Real-time clock
- Semaphore flags to provide interlocks for Common Memory access
- Common Memory field protection

2.1.1 INSTRUCTION ISSUE AND CONTROL

Each Background Processor contains a Program Address register, an instruction buffer with eight fields, and an instruction issue control mechanism to implement instruction issue and control.

Program Address register

Each Background Processor has a 32-bit Program Address (P) register indicating the address of the program instruction parcel currently in the issue position during normal operation. The Foreground Processor loads the P register with data at the beginning of a computation period. As each parcel issues from the instruction queue, the content of the P register advances by 1.

The P register content is reset to the branch destination address when a jump instruction is executed.

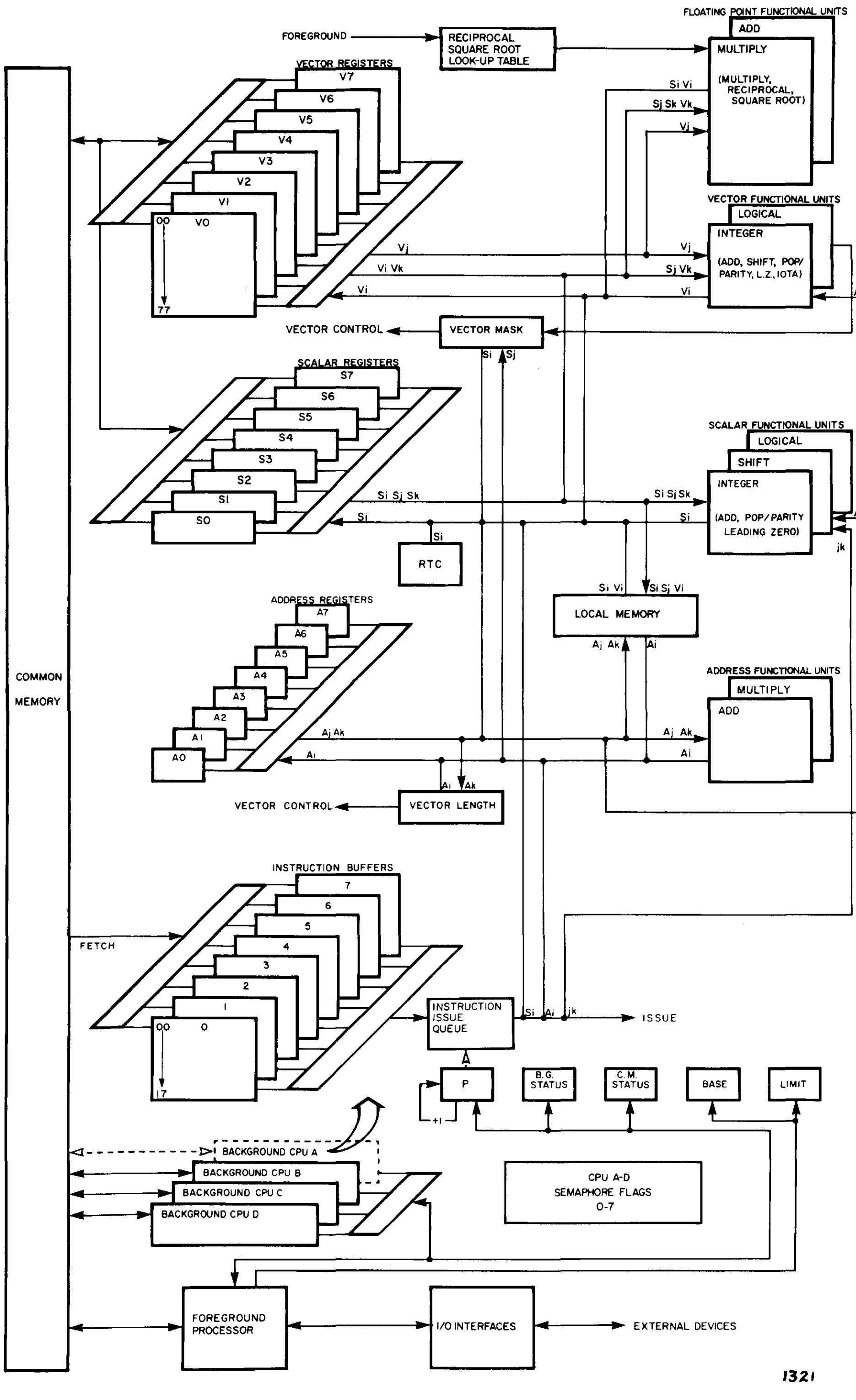


Figure 2-1. Control and Data Paths in a Background Processor

Instruction buffers

Each Background Processor has a buffer with eight independent fields to allow program loops to execute without additional Common Memory references. Programs can loop within the instruction buffer using any of the branch instructions.

Each independent field contains 16 words. The total instruction buffer size is 128 words.

The next sequential instruction out of the instruction buffer or a branch out of the instruction buffer discards the oldest data field and replaces it with 16 words of new data.

Instruction issue

Background instructions are translated in several steps and are allowed to issue sequentially by an instruction issue control mechanism. The words are disassembled into 16-bit parcels that are placed in a queue where the translation occurs. The instruction issue process involves checking the reservation flags for the registers and functional unit involved in the instruction sequence. The parcel waits in issue position in the instruction queue until all required resources are free.

Instruction parcels and 16-bit constants are intermixed in the instruction queue. The constant parcels are passed through the instruction queue without test.

2.1.2 REAL-TIME CLOCK

Each Background Processor has a 64-bit register that counts continuously at the clock period rate. This count value is used to determine the passage of real time to an accuracy of 1 clock period. The real-time clocks in the Background Processors are synchronized at deadstart. Instruction 115 reads the real-time clock.

2.1.3 SEMAPHORE FLAGS

To synchronize Common Memory references, eight semaphore flags in the background system interlock Common Memory references when multiple Background Processors are executing a single job. One semaphore flag is assigned to each currently active job in the background system. A Background Processor assigned to a job is assigned a semaphore flag at the same time.

The Background Processor uses four instructions in synchronizing its Common Memory references: 004, 005, 006, and 007. A 004 or 005 instruction requests the semaphore flag when the Background Processor program is accessing a Common Memory area that can interfere with other processors assigned to the job. The branch instruction results determine when the processor has exclusive access to this Common Memory area. The program must clear the semaphore flag to release the Common Memory area to another processor assigned to the same job.

2.1.4 COMMON MEMORY FIELD PROTECTION

At execution time each object program has a designated field of Common Memory holding instructions and data. Field limits are specified by the foreground functions when the object program is loaded and initiated. Field limits are contained in the Base Address (BA) register and the Limit Address (LA) register.

All memory addresses contained in the object program code are relative to the base address beginning the defined field. An object program cannot read or alter any Common Memory location with an absolute address lower than the base address. Each object program reference to Common Memory is checked against the limit and base addresses to determine if the address is within the assigned bounds.

Base Address register

Each Background Processor has a 32-bit BA register. The BA register defines the lower boundary of the Common Memory address field. The Foreground Processor enters data into this register while the Background Processor is in idle mode. The data remains in the register for the duration of the Background Processor computation period.

Each Common Memory reference from the Background Processor includes the addition of the BA register content to the other parts of the memory reference base address. All Background Processor references to Common Memory are relative to the base address boundary.

Limit Address register

Each Background Processor has a 32-bit LA register. The LA register defines the upper boundary of the Common Memory address field. The Foreground Processor enters data into this register while the Background Processor is in idle mode. The data remains in this register for the duration of the Background Processor computation period.

Memory range error

When a memory reference exceeds the range limits, a memory range error occurs. Each Common Memory reference from the Background Processor includes a test of the resulting absolute Common Memory address against the contents of the BA and LA registers. An error signal is sent to the status register if the resulting absolute Common Memory address is less than the base address or equal to, or greater than, the limit address. A read reference results in zero data for this case. A write reference is aborted.

2.2 OPERATING REGISTERS

Each Background Processor contains the following independent set of operating registers.

- Address
- Scalar
- Vector

Operating registers, a primary programmable resource of the Background Processor, enhance the speed of the system by satisfying heavy demands for data made by functional units. Different functional units can be used concurrently.

2.2.1 ADDRESS REGISTERS

Eight 32-bit Address (A) registers are used primarily to calculate memory locations for Local Memory and Common Memory references. A registers are used for 32-bit integer calculations and moving data directly from Local Memory. Data is also transferred between Address and Scalar registers.

2.2.2 SCALAR REGISTERS

Eight 64-bit Scalar (S) registers serve as source and destination for operands executing scalar arithmetic and logical instructions. S registers can furnish one operand in vector instructions.

The eight 64-bit S registers in a Background Processor support Vector registers in operations when one element of the computation is a constant value. The S registers function as computational way stations between Common Memory and the functional units where vector implementation of the work is not possible.

2.2.3 VECTOR REGISTERS

The major computational registers of the Background Processor are eight Vector (V) registers, each having 64 elements. Each V register element has 64 bits. When associated data is grouped into successive elements of a V register, the register quantity is treated as a vector. Examples of vector quantities are rows or columns of a matrix, and elements of a table.

Computational efficiency is achieved by identically processing each element of a vector. Vector instructions provide for the iterative processing of successive V register elements. A vector operation begins by obtaining operands from the first element of one or more V registers and delivering the result to the first element of a V register. Successive elements are provided during each clock period, and as each operation is performed the result is delivered to successive elements of the result V register. Vector operation continues until the number of operations performed by the instruction equals a count specified by the content of the Vector Length register (described later in this section).

Since many vectors exceed 64 elements, longer vectors are processed as one or more 64-element segments and a possible remainder of less than 64 elements.

The instruction issue control mechanism reserves the V registers that are involved in a functional unit operation. One, two, or three Vector registers can be involved, depending on the specific instruction. The functional unit is reserved at the same time as the V registers. The instruction sequence can then proceed to the next instruction and initiate concurrent activity as long as the resources reserved are not required.

The *i*, *j*, and *k* designators in a vector instruction can have the same value; it is advised, however, that the *i* designator always has a unique value. In the case of identical source operands, the data is streamed from the same V register to both data paths. In the case of a Destination register that is the same as a Source register, the V register writing function takes priority over reading. When this occurs, the reading vector delivers all zero words to the functional unit.

2.3 VECTOR CONTROL REGISTERS

The Vector Length register and the Vector Mask register provide control information needed in the performance of vector operations.

2.3.1 VECTOR LENGTH REGISTER

The Vector Length (VL) register is a 6-bit special purpose register explicitly referenced in the Background Processor instructions. The VL register holds the vector length during a portion of the background computation. All vector operations capture the vector length at the time of instruction issue from the VL register.

Vector registers always begin a read or write operation at the zero element position in the V register. Elements are read or written sequentially for the length of the current vector data. A short vector after a long vector leaves the old vector data in those positions not replaced with new data.

Values allowed in the VL register are 0 through 63. A zero value is interpreted as 64. Background instructions 025 and 036 communicate explicitly with the VL register.

2.3.2 VECTOR MASK REGISTER

The Vector Mask register (VM) is a 64-bit special purpose register explicitly referenced by the Background Processor instructions. The VM register merges vector data according to a set of precomputed Element flags. In effect, it provides a vehicle for implementing vector branch operations.

One bit of the VM register is associated with each element in the 64-element vector registers. The high-order bit (2^{63}) of the vector mask corresponds to element 0 of the vector data. The bits of the mask then proceed in order to represent the following vector elements.

The vector mask data can be formed by a vector operation in which each element is evaluated for a specific criterion. Instructions 030 through 033 perform these tests. The VM register is cleared at the beginning of these instruction sequences and then bits are entered one at a time as the vector stream passes the test station.

The vector mask data can be used to merge two vector streams into a single result stream. Instructions 146 and 147 are used for this purpose. Elements of the *j* operand are selected when the mask contains 1 bits. Elements of the *k* operand are selected when the mask contains 0 bits.

Instructions 034 and 114 move data between the VM register and an S register.

2.4 FUNCTIONAL UNITS

Each Background Processor has a set of functional units to implement algorithms for the instruction set. A number of functional units can operate simultaneously. Each functional unit produces one result per clock period. No information is retained in a functional unit for reference by subsequent instructions.

A functional unit receives operands from registers and delivers the result to a register when the function has been performed. Functional units operate essentially in three-address mode. Nonvector functional units can accept operands as fast as the instructions can issue.

A functional unit engaged in a vector operation remains busy for the duration and cannot participate in other operations. In this state, the functional unit is reserved. Other instructions requiring the same functional unit will not issue until the previous operation is completed. Only one functional unit of each type is available to the vector instruction hardware. When the vector operation completes, the reservation is dropped and the functional unit is then available for another operation.

Each Background Processor has the following set of functional units.

- Address Add
- Address Multiply
- Scalar Integer
- Scalar Shift
- Scalar Logical
- Vector Integer
- Vector Logical
- Floating-point Add
- Floating-point Multiply

In addition, a Background Processor contains a Local Memory which is a buffer for the A, S, and V register data.

2.4.1 ADDRESS ADD FUNCTIONAL UNIT

The Address Add unit performs 32-bit integer addition and subtraction of two A register operands. (Instruction 020 performs integer sums and 021 performs integer differences.) This unit can accept address operands as fast as the instructions can issue.

2.4.2 ADDRESS MULTIPLY FUNCTIONAL UNIT

The Address Multiply unit performs 32-bit integer multiplication of two A register operands. (Instructions 022 and 023 perform integer products.) This unit can accept address operands as fast as the instructions can issue.

2.4.3 SCALAR INTEGER FUNCTIONAL UNIT

The Scalar Integer unit performs 64-bit integer addition and subtraction of S register operands. (Instruction 104 performs integer sums and 105 performs integer differences.) It also performs population count (instruction 106*ij*0), population count parity (instruction 106*ij*1), and leading zero (instruction 107). This unit can accept scalar operands as fast as the instructions can issue.

2.4.4 SCALAR SHIFT FUNCTIONAL UNIT

The Scalar Shift unit shifts the entire 64-bit contents of an S register (instruction 110 left or 111 right) or the double 128-bit contents of two concatenated S registers (instruction 112 left or 113 right). This unit can accept scalar operands as fast as the instructions can issue.

2.4.5 SCALAR LOGICAL FUNCTIONAL UNIT

The Scalar Logical unit manipulates bit-by-bit the 64-bit quantities obtained from S registers. (Instruction 100 performs logical products, 101 performs logical products complemented, 102 performs logical differences, and 103 performs logical sums.) This unit can accept scalar operands as fast as the instructions can issue.

2.4.6 VECTOR INTEGER FUNCTIONAL UNIT

The Vector Integer unit performs vector shifts (150 for left single, 151 for right single, 152 for left double, and 153 for right double), vector integer arithmetic (160 and 161 for integer sums and 162 and 163 for integer differences), vector population count (164*ij*0 for population count and 164*ij*1 for population parity), vector leading zero count (165), and compressed iota (176). The unit can accept operand data each clock period, and after a transit time delay, can deliver a result each clock period.

2.4.7 VECTOR LOGICAL FUNCTIONAL UNIT

The Vector Logical unit manipulates bit-by-bit the 64-bit quantities from two V registers or from V registers and S registers (140 and 141 logical products, 142 and 143 for logical differences, and 144 and 145 for logical sums). The unit can accept operand data each clock period, and after a transit time delay, can deliver a result each clock period.

2.4.8 FLOATING-POINT ADD FUNCTIONAL UNIT

The Floating-Point Add unit performs addition or subtraction of 64-bit operands in floating-point format for both scalar and vector operations. It also performs the conversion between integer and floating-point. Refer to discussion of floating-point arithmetic for a description of the instructions that use this unit.

The unit is reserved for the time of a vector stream during execution of vector addition instructions. The unit can accept vector operand data each clock period, and after a transit time delay, can deliver a result each clock period. The unit can accept scalar references as fast as they issue if the unit is not processing vector data.

2.4.9 FLOATING-POINT MULTIPLY FUNCTIONAL UNIT

The Floating-Point Multiply unit performs full multiplication of 64-bit operands in floating-point format for both scalar and vector operations. It also performs reciprocal approximation, reciprocal square root approximation, reciprocal iteration, and reciprocal square root iteration. Refer to discussion of floating-point arithmetic for a description of the instructions that use this unit.

The unit is reserved for the time of a vector stream during execution of vector addition instructions. The unit can accept vector operand data each clock period, and after a transit time delay, can deliver a result each clock period. The unit can accept scalar references as fast as they issue if the unit is not processing vector data.

2.4.10 LOCAL MEMORY

Each Background Processor contains 16,384 64-bit words of Local Memory. This memory holds scalar operands during a computation period. The Local Memory can also be used for temporary storage of vector elements when these elements are used more than once in a computation in the V registers. Instructions that use Local Memory are:

- 044 and 046 read from Local Memory to A register
- 045 and 047 write to Local Memory from A register
- 054 and 056 read from Local Memory to S register
- 055 and 057 write to Local Memory from S register
- 074 read from Local Memory to V register
- 075 write to Local Memory from V register

2.5 ARITHMETIC OPERATIONS

Functional units in the Background Processor perform either twos complement integer arithmetic or floating-point arithmetic.

2.5.1 INTEGER ARITHMETIC

All integer arithmetic, whether 32 bits or 64 bits, is twos complement. The Address Add and Address Multiply units perform 32-bit arithmetic. The Scalar Integer unit performs scalar 64-bit arithmetic and the Vector Integer unit performs vector 64-bit arithmetic.

Integer representations of the integers 0, +1, and -1 in 32-bit and 64-bit format are illustrated using octal notation.

Integer	32-bit Format	64-bit Format
0	00000000000	0000000000000000000000
+1	00000000001	0000000000000000000001
-1	37777777777	1777777777777777777777

Multiplication of two scalar integer operands is accomplished by using the floating-point multiply instruction. Division is done by algorithm; the particular algorithm used depends on the number of bits in the quotient.

2.5.2 FLOATING-POINT ARITHMETIC

Floating-point numbers are represented in a standard format throughout the Background Processor. This format is a packed representation of a binary coefficient and an exponent. The coefficient is a 48-bit signed fraction. The sign of the coefficient is separated from the rest of the coefficient as shown in figure 2-2. Since the coefficient is signed magnitude, it is not complemented for negative values.

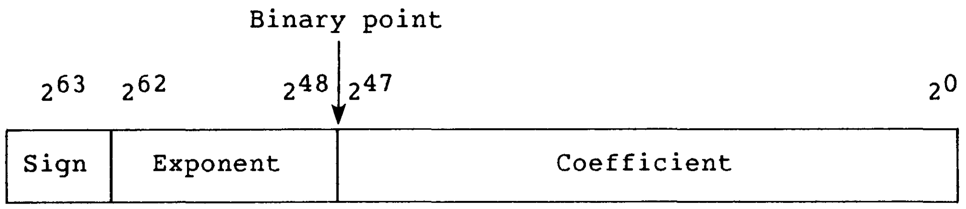


Figure 2-2. Floating-point Data Format

The exponent portion of the floating-point format is represented as a biased integer in bits 2^{62} through 2^{48}. The bias that is added to the exponents is 40000_8. The positive range of exponents is 40000_8 through 57777_8. The negative range of exponents is 37777_8 through 20000_8. Thus, the unbiased range of exponents is the following (note the negative range is one larger):

2^{-20000_8} through 2^{+17777_8}

In terms of decimal values, the floating-point format of the Background Processor allows the accurate expression of numbers to about 15 decimal digits in the approximate decimal range of 10^{-2466} through 10^{+2466}.

A floating-point representation of the integers 0, +1, and -1 in normalized form is illustrated using octal notation for each of the three fields.

Integer	Floating-point representation
0	0 00000 0000000000000000
+1	0 40001 4000000000000000
-1	1 40001 4000000000000000

Normalizing

A nonzero floating-point number is normalized if the most significant bit of the coefficient is nonzero. This condition implies the coefficient has been shifted as far left as possible and the exponent adjusted accordingly. Therefore, the floating-point number has no leading zeros in the coefficient. The exception is that a normalized floating-point zero is all zeros.

When a floating-point number is created by inserting an exponent of 40060_8 into a 48-bit integer word, the result should be normalized before being used in a floating-point operation. Normalization can be accomplished by adding the unnormalized floating-point operand to 0 (see integer to floating-point conversion in this section).

Range errors

Exponent values of 60000_8 and greater are considered to have overflowed the exponent range. Hardware tests are performed for these values to indicate floating-point range error. Exponent values less than 20000_8 are considered to have underflowed the floating-point range. Such values are treated as if they had a zero value. The hardware does not indicate when a computation underflows the floating-point range.

Whether or not range errors are enabled, when an overflow condition is detected by the hardware the result exponent is forced to an overflow value. Each floating-point operation forces a signature exponent as follows:

Floating-point add/subtract	60000_8
Floating-point multiply	60001_8
Floating-point reciprocal approximation	60002_8
Floating-point square root approximation	60004_8

Floating-point addition

The Floating-point Add unit forms the sum of two operands in floating-point format and delivers a result in floating-point format. The result is always normalized regardless of source operand status. Instructions 120, 170, and 171 use the Floating-point Add sequence.

In the process of adding two floating-point operands, one operand coefficient is shifted right for exponent matching. The coefficient from this shifting operation is rounded up.

A special test is made for all 0 bits in the result coefficient. When this occurs the exponent field in the result is also cleared. A word of all zeros is delivered to the destination register.

A special test is made for one or both operands with an overflow exponent. An error signal is sent to the Background Port Status register (refer to section 5) if range errors are enabled, and an overflow exponent (60000_8) is forced in the result delivered to the destination register.

Floating-point subtraction

The Floating-point Add unit forms the difference of two operands in floating-point format and delivers a result in floating-point format. Instructions 121, 172, and 173 use the floating-point subtraction sequence.

Floating-point to integer conversion

The Floating-point Add unit forms an integer representation of a floating-point operand. This process is accomplished by adding the operand to a constant integer. Instructions 122 and 174 use this form of the floating-point add sequence.

The maximum size of the resulting integer value is 48 bits. A positive or negative result is sign extended to form a 64-bit integer result.

An operand with a floating-point value greater than a 48-bit integer is an error condition. An error signal is sent to the Background Port Status register if floating-point range errors are enabled, and a zero result is delivered to the destination register.

Integer to floating-point conversion

The Floating-point Add unit forms a floating-point representation of an integer operand. This process is accomplished by adding the operand to a constant and using the floating-point normalize hardware to form the proper floating-point result. Instructions 123 and 175 use this form of the floating-point add sequence.

The maximum allowable size of the integer operand is 48 bits; if greater, no error is flagged. The bits above 48 bits are discarded during the operation.

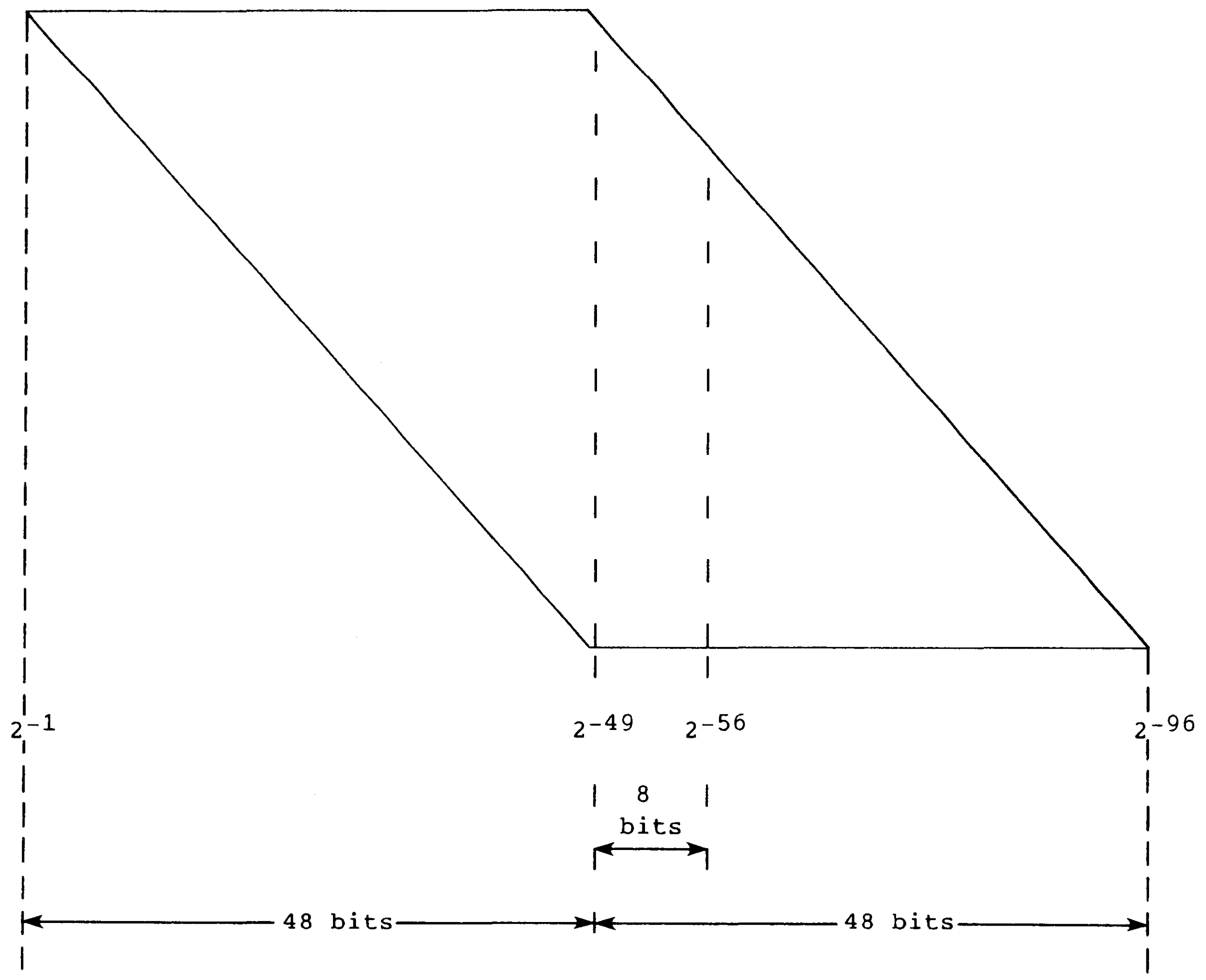
Floating-point product

The Floating-point Multiply unit forms the product of two operands in floating-point format and delivers a result in floating-point format. If both operands are normalized, the result is also normalized. Instructions 124, 154, and 155 use this sequence.

The 48-by-48 matrix of logical product bits is truncated 8 bit positions below the low-order result coefficient bit (see figure 2-3). Round bits are added to this lower field to give an equal population of high and low round errors for random operands. A round bias exists over narrow ranges of operands because of the 1-bit correction shift after the round operation.

The following special cases are treated in floating-point multiplication for operands out of range.

1. One or both operands have overflow exponent.
2. Sum of operand exponents is an overflow.
3. Sum of exponents is an underflow.
4. Both exponents are all zeros.



For instructions 124, 132, 133, 154, 166, and 167, bits 2^{-49} through 2^{-56} are used for rounding. Bits 2^{-50} and 2^{-51} are the round bits and bits 2^{-53} through 2^{-56} compensate for truncation.

```
2-1 through 2-48   2-49   2-50   2-51   2-52   2-53   2-54   2-55   2-56

  0--------0         0      1      1      0      1      0      0      1
```

Figure 2-3. 48-by-48 Bit Matrix Used for Floating-point Product

Cases 1 and 2 cause a Floating-point Error signal to be sent to the Background Port Status register if the floating-point range errors are enabled. The result delivered to the Destination register is forced to an overflow exponent value (60001_8). Case 3 results in an all-zero word sent to the Destination register. Case 4 computes the coefficients with no normalize correction. The resulting exponent for this case is 0, which aids multiple-precision and integer calculations.

Reciprocal approximation

The Floating-point Multiply unit forms an approximation to the reciprocal of a floating-point operand value. Instructions 132 and 166 use this sequence.

The values from the table are used in a linear interpolation computation. The form of this computation is illustrated in the following example.

Example:

In this example, A is a reciprocal approximation for the high-order 12 bits of operand coefficient; B is the operand coefficient; and R is the better reciprocal approximation.

Then the iteration step for interpolation is:

$$R = 2A - A*A*B$$

The two approximations read from the table are 2A and -A*A. The normal multiply mechanism is then used to form the product with the additional term included in the summing process.

Two special cases occur in the reciprocal approximation sequence.

- Operand exponent has overflow value.
- Operand exponent has underflow value.

Both cases cause an error signal to be sent to the Background Port Status register if the floating-point range error is enabled and cause the computational result exponent to be forced to an overflow value (60002_8).

Reciprocal iteration

****

CAUTION

The reciprocal iteration instructions (126 and 156) should be used only with the reciprocal approximation instructions (132 and 166) and should only be used for one additional iteration. Operands not generated by the reciprocal approximation instructions may not deliver the expected result.

****

The Floating-point Multiply unit forms a floating-point number that is used in a second iteration for the reciprocal of a full-precision operand. The first iteration is formed in the reciprocal approximation described above. The second iteration uses the same process to form a reciprocal approximation with 46 bits of coefficient accuracy. Instructions 126 and 156 use this sequence (see figure 2-4).

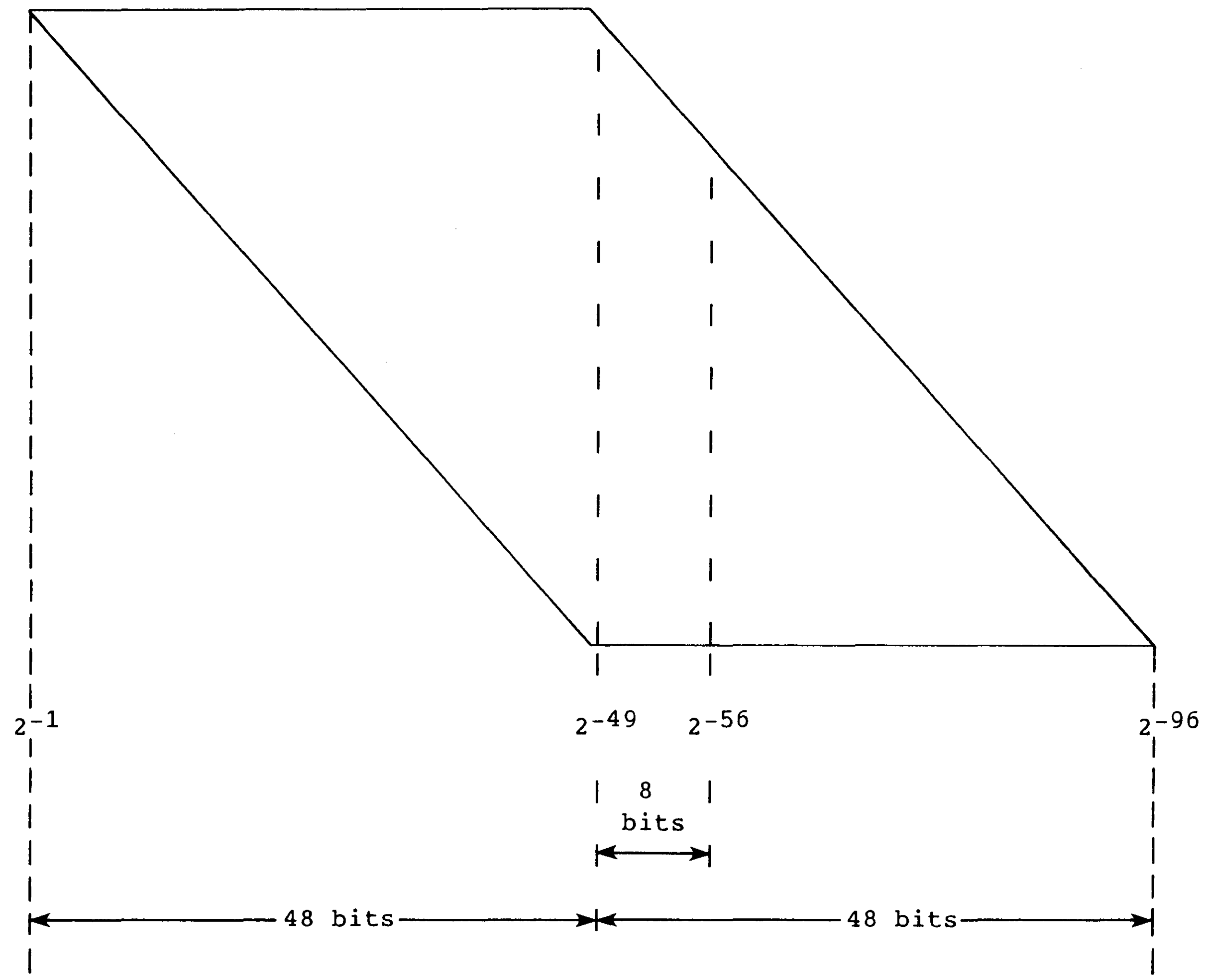
The division algorithm that computes S1/S2 to full precision requires four operations.

1.	S3 = 1/S2	Half-precision reciprocal
2.	S4 = 2 - S2 * S3	Correction factor
3.	S5 = S3 * S4	Reciprocal = Half-precision reciprocal * correction factor
4.	S6 = S1 * S5	Quotient = numerator * reciprocal

Reciprocal square root approximation

The Floating-point Multiply unit forms an approximation to the reciprocal square root of a floating-point operand value. Instructions 133 and 167 use this sequence.

The values from the table are used in a linear interpolation computation. The form of this computation is illustrated in the following example.



For instructions 126 and 156, bits 2^{-49} through 2^{-56} are used for rounding. Bits 2^{-50} and 2^{-51} are the round bits and bits 2^{-53} through 2^{-56} compensate for truncation.

2^{-1} through 2^{-48}	2^{-49}	2^{-50}	2^{-51}	2^{-52}	2^{-53}	2^{-54}	2^{-55}	2^{-56}
1--------1	1	0	1	1	0	0	1	0

Figure 2-4. 48-by-48 Bit Matrix Used for Reciprocal Iteration

Example:

In this example, A is a reciprocal square root approximation for the operand coefficient, B is the operand coefficient, and R is the better reciprocal square root approximation.

The iteration step for interpolation is:

R = (3A/2) - (A*A*A*B/2)

The two approximations read from the table are 3A/2 and -A*A*A/2. The normal multiply mechanism is then used to form the product with the additional term included in the summing process.

Three special cases occur in the reciprocal square root approximation sequence.

1. Operand exponent has overflow value.
2. Operand exponent has value of 0 through 3.
3. Operand is a negative value.

Cases 1 and 3 cause an error signal to be sent to the Background Port Status register. All three cases cause the computational result exponent to be forced to an overflow value (60004_8).

<u>Reciprocal square root iteration</u>

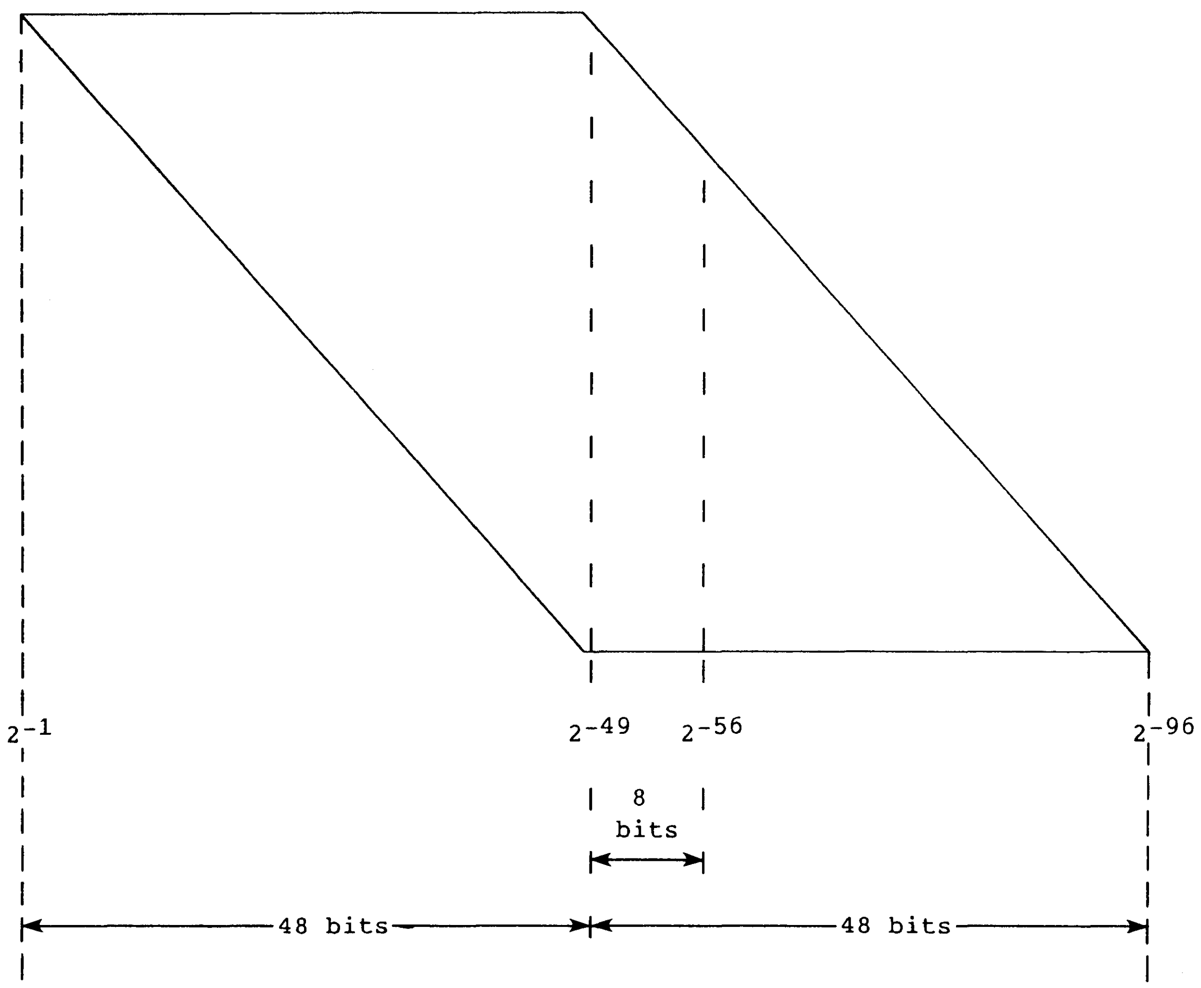
****

CAUTION

The square root iteration instructions (127 and 157) should be used only with the reciprocal square root approximation instructions (133 and 167) and should only be used for one additional iteration. Operands not generated by the reciprocal square root approximation instructions may not deliver the expected result.

****

The Floating-point Multiply unit forms a floating-point number which is used in a second iteration for the reciprocal square root of an operand. The first iteration is formed in the reciprocal square root approximation described above. The second iteration uses the same process to form a reciprocal square root with 46 bits of coefficient accuracy. Instructions 127 and 157 use this sequence (see figure 2-5).



For instructions 127 and 157, bits 2^{-49} through 2^{-56} are used for rounding. Bits 2^{-50} and 2^{-51} are the round bits and bits 2^{-53} through 2^{-56} compensate for truncation.

2^{-1} through 2^{-48}	2^{-49}	2^{-50}	2^{-51}	2^{-52}	2^{-53}	2^{-54}	2^{-55}	2^{-56}
1--------1	0	1	0	0	0	1	1	0

Figure 2-5. 48-by-48 Bit Matrix Used for Square Root Iteration

The square root algorithm that computes the square root of S1 requires four operations.

1.	S2 = 1/ S1	Half-precision reciprocal square root approximation
2.	S3 = S1 * S2	Half-precision square root
3.	S4 = (3 - S2 * S3)/2	Correction factor
4.	S5 = S3 * S4	Square root = half-precision square root * correction factor

3. BACKGROUND PROCESSOR SYMBOLIC MACHINE INSTRUCTIONS

This section contains detailed information about individual instructions or groups of related instructions. Each instruction begins with boxed information consisting of the CRAY-2 Assembly Language (CAL) Version 2 syntax format, an operand (if required), a brief description of each instruction, and the machine instruction (octal code sequence defined by the f field).

Following the boxed information is a more detailed decscription of the instruction and an example using the instruction.

3.1 SYMBOLIC INSTRUCTION FORMAT

The following special characters can appear in the operand field of symbolic machine instructions and are used by the assembler in determining the operation to be performed.

+	Integer sum of adjoining registers
+F,+f	Floating-point sum of adjoining registers
-	Integer difference of adjoining registers
-F,-f	Floating-point difference of adjoining registers
*	Integer product of adjoining registers
*F,*f	Floating-point product of adjoining registers
*I,*i	Reciprocal iteration of adjoining registers
*Q,*q	Floating-point square root approximation
*Q,*q	Square root iteration of adjoining registers
/H,/h	Floating-point reciprocal approximation
#	Use ones complement
>	Shift value or form mask from left to right
<	Shift value or form mask from right to left
&	Logical product of adjoining registers
!	Logical sum of adjoining registers
\	Logical difference of adjoining registers
CI,ci	Compressed iota
F,f	Full load (64-bits)
FIX,fix	Convert from floating-point to integer
FLT,flt	Convert from integer to floating-point
H,h	Half load (32-bits)
L,l	Left load (32-bits)
M,m	Negative
N,n	Nonzero

P,p	Parcel load (16-bits)
P,p	Population count
P,p	Positive
Q,q	Parity count
S,s	Short load (6-bits)
Z,z	Leading-zero count
Z,z	Zero

3.2 MACHINE INSTRUCTION FORMAT

The Background Processors translate instructions in 16-bit parcels of data. These parcels are packed four-per-word in the Common Memory. The parcels are addressed as if the Common Memory had four times as many locations and the data were 16 bits long.

Figure 3-1 illustrates the format of a 16-bit instruction parcel.

f	*i*	*j*	*k*
7	3	3	3

Figure 3-1. Instruction Parcel Format

As shown in figure 3-1, the *f* designator is the operation code. The *i, j,* and *k* designators generally refer to V, S, or A registers in a three-address format. Uppercase or lowercase designators for the registers are allowed in CAL; both will be used in the symbolic instruction descriptions. The mnemonics may be entered in all uppercase or all lowercase. The *i* designator generally specifies the Destination register for the functional computation. The *j* and *k* designators generally specify the source operands.

Some instructions include additional parcels of constant data. There can be the following parcels of constant data depending on the specific instruction:

- 1 (m_1)
- 2 (m_1 and m_2)
- 4 (m_1, m_2, m_3, and m_4)

Single parcel constants are generally used to address the Local Memory. Two parcel constants are generally used to address Common Memory. Four parcel constants are used to enter 64-bit values in the S registers.

When instructions read constants from the following parcels in the instruction stream, the Program address is advanced over these data parcels to point to the next instruction. The high-order data parcel is read first for those cases of multiparcel data.

3.3 INSTRUCTION DESCRIPTIONS

The instruction descriptions begin with the octal code for the high-order 7 bits of the parcel (*f* designator). The three octal register designators (*i, j,* and *k*) then follow. An *x* appears in the description where a register's designator is ignored. CAL will insert a zero for every *x*.

Result	Operand	Description	Machine Instruction
err		Error exit	000*x*00
exit		Normal exit	000*x*01
exit	*exp*	Normal exit	000*xjk*
		Executes as 000*xjk*	001*xjk*

Instructions 000 and 001 stop the current program sequence, place the Background Processor in idle mode, and set the Exit Mode and Idle Mode flags in the Background Port Status register. The 6-bit *jk* value is entered into the Background Port Status register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
000000		err		
000001		exit		

INSTRUCTION 002

Result	Operand	Description	Machine Instruction
r,a_i	a_k	Register jump to (a_k) with return address to a_i	002*ixk*
j	a_k	Register jump to (a_k), value in a_k erased	002*kxk*

Instruction 002 stops the current program sequence and begins a new sequence at a computed parcel address read from the A_k register. The parcel address for the next instruction in the current program sequence is entered into the A_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002*ixk*				
002*kxk*				

INSTRUCTION 003

Result	Operand	Description	Machine Instruction
j	*exp*	Unconditional jump	003*xxx* m_1 m_2

Instruction 003 stops the current program sequence and begins a new sequence at a specified constant parcel address read from the next 2 parcels in the instruction queue.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
003*xxx*				

Result	Operand	Description	Machine Instruction
jcs	*exp*	Jump to constant parcel if Semaphore clear; set Semaphore.	004*xxx* m_1 m_2
jss	*exp*	Jump to constant parcel if Semaphore set; set Semaphore.	005*xxx* m_1 m_2

Instructions 004 and 005 conditionally stop the current instruction sequence and begin a new sequence at a specified constant parcel address read from the next 2 parcels in the instruction queue.

The branch is conditional on the state of the Semaphore flag assigned to this Background Processor. The Background Port Status register points to the Semaphore flag. The Semaphore flag is set for either instruction if it was not previously set. The Semaphore flag bit in the Background Port Status register is set if either instruction alters the state of the flag from 0 to 1.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
004*xxx*				
005*xxx*				

INSTRUCTION 006

Result	Operand	Description	Machine Instruction
ssm		Set Semaphore	006*xxx*

Instruction 006 sets the Semaphore flag assigned to this Background Processor without regard to its previous state. The Semaphore flag bit in the Background Port Status register is set if the previous state of the Semaphore flag was a 0. The operating system program uses this instruction to restore Semaphore flag values at the time of job restart.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
006*xxx*				

INSTRUCTION 007

Result	Operand	Description	Machine Instruction
csm		Clear Semaphore	007*xxx*

Instruction 007 clears the Semaphore flag assigned to this Background Processor without regard to its previous value. When this instruction executes, the semaphore bit in the Background Port Status register is cleared. A Background Processor program may use this instruction to release access to a privileged area of Common Memory for other processors assigned to this job.

This instruction issues without delay. Execution of the function, however, may be delayed by activity in the Common Memory port. The following instruction does not issue until the Common Memory quadrant buffers are clear. The delay ensures that any Common Memory write operations have been completed before another processor is allowed access to the privileged area.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
007*xxx*				

Result	Operand	Description	Machine Instruction
jz	a_k,*exp*	Branch if (a_k) is zero	010*xxk* m_1 m_2
jn	a_k,*exp*	Branch if (a_k) is nonzero	011*xxk* m_1 m_2
jp	a_k,*exp*	Branch if (a_k) is positive	012*xxk* m_1 m_2
jm	a_k,*exp*	Branch if (a_k) is negative	013*xxk* m_1 m_2

Instructions 010 through 013 conditionally stop the current instruction sequence and begin a new sequence at a specified constant parcel address read from the next 2 parcels in the instruction queue.

The content of the A_k register determines the condition of the branch. The current program sequence is continued if the branch criterion is not met.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
010*xxk*				
011*xxk*				
012*xxk*				
013*xxk*				

Result	Operand	Description	Machine Instruction
jz	s_j,*exp*	Branch if (s_j) is zero	014*xjx* m_1 m_2
jn	s_j,*exp*	Branch if (s_j) is nonzero	015*xjx* m_1 m_2
jp	s_j,*exp*	Branch if (s_j) is positive	016*xjx* m_1 m_2
jm	s_j,*exp*	Branch if (s_j) is negative	017*xjx* m_1 m_2

Instructions 014 through 017 conditionally stop the current instruction sequence and begin a new sequence at a specified constant parcel address read from the next 2 parcels in the instruction queue.

The content of the S_j register determines the condition of the branch as indicated above. The current program sequence is continued if the branch criterion is not met.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
014*xjx*				
015*xjx*				
016*xjx*				
017*xjx*				

INSTRUCTIONS 020 - 021

Result	Operand	Description	Machine Instruction
a_i	a_j+a_k	Integer sum of (a_j) and (a_k) to a_i	020*ijk*
a_i	a_j-a_k	Integer difference of (a_j) and (a_k) to a_i	021*ijk*

Instructions 020 and 021 perform 32-bit integer arithmetic in the A registers. The operands are obtained from registers A_j and A_k, and the result is delivered to register A_i.

Instruction 020 forms the 32-bit integer sum.

Instruction 021 forms the 32-bit integer difference.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
020*ijk*				
021*ijk*				

Result	Operand	Description	Machine Instruction
a_i	a_j*a_k	Integer product of (a_j) and (a_k) to a_i	022*ijk*
		Executes the same as 022*ijk*	023*ijk*

Instruction 022 forms the integer product of two 32-bit integer operands. The operands are obtained from the A_j and A_k registers. The low-order 32-bits of the result data are delivered to the A_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
022*ijk*				
023*ijk*				

INSTRUCTION 024

Result	Operand	Description	Machine Instruction
a_i	s_j	Copy (s_j) to a_i	024*ijx*

Instruction 024 reads a 64-bit word from the S_j register and enters the low-order 32 bits into the A_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
024*ijx*				

INSTRUCTION 025

Result	Operand	Description	Machine Instruction
a_i	vl	Copy (vl) to a_i	025*ixx*

Instruction 025 forms a 32-bit word from the data in the VL register. The low-order 6 bits are copied from the VL data. The high-order 24 bits are 0. The result data is delivered to the A_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
025*ixx*				

Result	Operand	Description	Machine Instruction
a_i	*exp*	Load a_i with a value	026*ijk*
a_i	*exp*,s	Load a_i with a 6-bit value	026*ijk*
a_i	*exp*,s,p	Load a_i with a 6-bit positive value	026*ijk*
a_i	*exp*	Load a_i with a value	027*ijk*
a_i	*exp*,s	Load a_i with a 6-bit value	027*ijk*
a_i	*exp*,s,m	Load a_i with a 6-bit negative value	027*ijk*

Instructions 026 and 027 form a 32-bit word from the *jk* data in the instruction parcel. The low-order 6 bits are copied from the instruction parcel. For instruction 026, the high-order 26 bits are zeros. For instruction 027, the high-order 26 bits are ones. The result data is delivered to the A_i register.

The A_i *exp* instruction will map into either an 026, 027, 040, 041, or an 042 opcode. If all symbols within the expression have been previously defined within the currently enabled qualifier then CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 042 opcode.

CAL will map the A_i *exp*,S instruction into the 027 opcode if the expression is negative and has a relative attribute of absolute. Otherwise, this instruction will be mapped into the 026 opcode.

Instruction 026 loads the A_i register with positive *jk*.

Instruction 027 loads the A_i register with negative *jk*.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
026*ijk*				
026*ijk*				
026*ijk*				
027*ijk*				
027*ijk*				
027*ijk*				

Result	Operand	Description	Machine Instruction
vm	v_k,z	Set vm from zero elements of (v_k)	030*xxk*
vm	v_k,n	Set vm from nonzero elements of (v_k)	031*xxk*
vm	v_k,p	Set vm from positive elements of (v_k)	032*xxk*
vm	v_k,m	Set vm from negative elements of (v_k)	033*xxk*

Instructions 030 through 033 create a vector mask in the VM register based on the results of testing the contents of the elements of register V_k. The VM register is initially cleared, and a bit is entered in the VM register where elements of the vector stream meet the test criterion. The high-order bit position in the VM register corresponds to the first element of the vector. The bit positions are then assigned in order for the remainder of the vector stream.

These instructions are performed in the vector logical unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
030*xxk*				
031*xxk*				
032*xxk*				
033*xxk*				

INSTRUCTION 034

Result	Operand	Description	Machine Instruction
vm	s_j	Copy (s_j) to vm	034*xjx*

Instruction 034 enters the VM register with a 64-bit word from the S_j register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
034*xjx*				

INSTRUCTION 035

Result	Operand	Description	Machine Instruction
dri		Disable halt on memory field range error	035*xx*0
eri		Enable halt on memory field range error	035*xx*1
dfi		Disable halt on floating-point error	035*xx*2
efi		Enable halt on floating-point error	035*xx*3

Instruction 035 alters 2 status bits (bits 21 and 22) in the Background Port Status register depending on the value of the *k* designator in the instruction parcel.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
035*xx*0				
035*xx*1				
035*xx*2				
035*xx*3				

Result	Operand	Description	Machine Instruction
vl	a_k	Copy (a_k) to vl Executes the same as 036*xxk*	036*xxk* 037*xxk*

Instruction 036 enters the low-order 6 bits of data from the A_k register into the VL register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
036*xxk*				
037*xxk*				

Result	Operand	Description	Machine Instruction
a_i	*exp*	Load a_i with a value	040*ixx* m_I
a_i	*exp*,p	Load a_i with a 16-bit value	040*ixx* m_I
a_i	*exp*,p,p	Load a_i with a 16-bit positive value	040*ixx* m_I
a_i	*exp*	Load a_i with a value	041*ixx* m_I
a_i	*exp*,p	Load a_i with a 16-bit value	041*ixx* m_I
a_i	*exp*,p,m	Load a_i with a 16-bit negative value	041*ixx* m_I

Instructions 040 and 041 enter a 32-bit constant into the A_i register. The low-order 16 bits are read from the following parcel in the instruction queue.

The A_i *exp* instruction will map into either an 026, 027, 040, 041, or an 042 opcode. If all symbols within the expression have been previously defined within the currently enabled qualifier, CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 042 opcode.

CAL will map the A_i *exp*,P instruction into the 041 opcode if the expression is negative and has a relative attribute of absolute. Otherwise, this instruction will be mapped into the 040 opcode.

For instruction 040, the high-order 16 bits are zero-filled.

For instruction 041, the high-order 16 bits are set to ones.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
040*ixx*				
040*ixx*				
040*ixx*				
041*ixx*				
041*ixx*				
041*ixx*				

Result	Operand	Description	Machine Instruction
a$_i$	*exp*	Load a$_i$ with a value	042*ixx* m_1 m_2
a$_i$	*exp*,h	Load a$_i$ with a 32-bit value	042*ixx* m_1 m_2
		Executes the same as 042*ixx*	043*ixx* m_1 m_2

Instruction 042 loads the A$_i$ register with a 32-bit constant read from the next 2 parcels in the instruction queue.

The A$_i$ *exp* instruction will map into either an 026, 027, 040, 041, or 042 opcode. If all symbols within the expression have been previously defined within the currently enabled qualifier, CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 042 opcode.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
042*ixx*				
042*ixx*				
043*ixx*				

INSTRUCTION 044

Result	Operand	Description	Machine Instruction
a_i	[*exp*]	Read from location exp in Local Memory to a_i	044*ixx* m_l

Instruction 044 enters the A_i register with the low-order 32 bits of a data word in Local Memory. The Local Memory address is obtained from the following parcel in the instruction queue.

If the expression has a relative attribute of relocatable, it must be relative to a Local Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
044*ixx*				

INSTRUCTION 045

Result	Operand	Description	Machine Instruction
[*exp*]	a_k	Write (a_k) to location exp in Local Memory	045*xxk* m_l

Instruction 045 writes one 64-bit word in Local Memory. The Local Memory address is obtained from the following parcel in the instruction queue. The data word is obtained by sign extending the content of the A_k register through the high-order 32 bit positions of the 64-bit word.

If the expression has a relative attribute of relocatable, it must be relative to a Local Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
045*xxk*				

INSTRUCTION 046

Result	Operand	Description	Machine Instruction
a_i	$[a_k]$	Read from location a_k in Local Memory to a_i	046*ixk*

Instruction 046 enters the A_i register with the low-order 32 bits of a data word in Local Memory. The Local Memory address is obtained from the A_k register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
046*ixk*				

Result	Operand	Description	Machine Instruction
[a$_k$]	a$_j$	Write (a$_j$) to location a$_k$ in Local Memory	047*xjk*

Instruction 047 writes one 64-bit word in Local Memory. The Local Memory address is obtained from the A$_k$ register. The write data word is obtained by sign extending the content of the A$_j$ register through the high-order 32 bit positions of the 64-bit word.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
047*xjk*				

Result	Operand	Description	Machine Instruction
s_i	*exp*	Load s_i with a value	050*ixx* m_1 m_2
s_i	*exp*,h	Load s_i with a 32-bit value	050*ixx* m_1 m_2
s_i	*exp*,h,p	Load s_i with a 32-bit positive value	050*ixx* m_1 m_2
s_i	*exp*	Load s_i with a value	051*ixx* m_1 m_2
s_i	*exp*,h	Load s_i with a 32-bit value	051*ixx* m_1 m_2
s_i	*exp*,h,m	Load s_i with a 32-bit negative value	051*ixx* m_1 m_2
s_i	*exp*,l	Load s_i left side with a 32-bit value	052*ixx* m_1 m_2

The S_i *exp* instruction will map into either a 050, 051, 052, 053, 116, or 117 opcode. If all the symbols within the expression have been previously defined within the currently enabled qualifier, CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 053 opcode.

CAL will map the S_i *exp*,H instruction into the 051 opcode if the expression is negative and has a relative attribute of absolute. Otherwise, this instruction will be mapped into the 050 opcode.

Instructions 050 through 052 load a 64-bit value into the S_i register.

Instruction 050 reads the low-order 32 bits from the next 2 parcels in the instruction queue. The high-order 32 bits are zero-filled.

Instruction 051 reads the low-order 32 bits from the next 2 parcels in the instruction queue. The high-order 32 bits are filled with ones.

Instruction 052 reads the high-order 32 bits of a constant from the next 2 parcels in the instruction queue. The low-order 32 bits are zero-filled.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
050*ixx*				
050*ixx*				
050*ixx*				
051*ixx*				
051*ixx*				
051*ixx*				
052*ixx*				

INSTRUCTION 053

Result	Operand	Description	Machine Instruction
s_i	*exp*	Load s_i with a value	053*ixx* m_1 m_2 m_3 m_4
s_i	*exp*,f	Load s_i with a 64-bit value	053*ixx* m_1 m_2 m_3 m_4

The S_i *exp* instruction will map into either an 050, 051, 052, 053, 116, or a 117 opcode. If all the symbols within the expression have been previously defined within the currently enabled qualifier, CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 053 opcode.

Instruction 053 loads the S_i register with a 64-bit constant read from the following 4 parcels in the instruction queue.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
053*ixx*				
053*ixx*				

Result	Operand	Description	Machine Instruction
s_i	[*exp*]	Read from location *exp* in Local Memory	054*ixx* m_l

Instruction 054 enters the S_i register with a 64-bit data word from the Local Memory. The Local Memory address is obtained from the following parcel in the instruction queue.

If the expression has a relative attribute of relocatable, it must be relative to a Local Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
054*ixx*				

INSTRUCTION 055

Result	Operand	Description	Machine Instruction
[*exp*]	s_j	Write (s_j) to location exp in Local Memory	055*xjx* m_l

Instruction 055 writes one 64-bit word into the Local Memory. The Local Memory address is obtained from the following parcel in the instruction queue. The 64-bit word is obtained from the S_j register.

If the expression has a relative attribute of relocatable, it must be relative to a Local Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
055*xjx*				

INSTRUCTION 056

Result	Operand	Description	Machine Instruction
s_i	$[a_k]$	Read from location (a_k) in Local Memory	056*ixk*

Instruction 056 enters the S_i register with a 64-bit data word from Local Memory. The Local Memory address is obtained from the A_k register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
056*ixk*				

INSTRUCTION 057

Result	Operand	Description	Machine Instruction
[a_k]	s_i	Write (s_i) to location (a_k) in Local Memory	057*ixk*

Instruction 057 stores one 64-bit word in Local Memory. The Local Memory address is obtained from the A_k register. The 64-bit word is obtained from the S_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
057*ixk*				

Result	Operand	Description	Machine Instruction
s_i	(a_j,a_k)	Read from Common Memory location $(a_j)+(a_k)$ to s_i	060*ijk*

Instruction 060 reads one 64-bit word from Common Memory and enters it in the S_i register. The relative Common Memory location is determined by adding the content of register A_j to the content of register A_k.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
060*ijk*				

INSTRUCTION 061

Result	Operand	Description	Machine Instruction
(a_j, a_k)	s_i	Write (s_i) to Common Memory at location $(a_j)+(a_k)$	061*ijk*

Instruction 061 stores one 64-bit word into Common Memory from the S_i register. The relative Common Memory location is determined by adding the content of register A_j to the content of register A_k.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
061*ijk*				

INSTRUCTION 062

Result	Operand	Description	Machine Instruction
s_i	(a_k)	Read from Common Memory at location (a_k) to s_i	062*ixk*

Instruction 062 reads one 64-bit word from Common Memory and enters it in the S_i register. The relative Common Memory location is obtained from the A_k register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
062*ixk*				

INSTRUCTION 063

Result	Operand	Description	Machine Instruction
(a_k)	s_i	Write (s_i) to Common Memory at location (a_k)	063*ixk*

Instruction 063 writes one 64-bit word in the Common Memory. The relative Common Memory location is obtained from the A_k register. The 64-bit word is obtained from the S_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
063*ixk*				

INSTRUCTION 064

Result	Operand	Description	Machine Instruction
s_i	(a*k*,*exp*)	Read from Common Memory at location (a_k)+*exp* to s_i	064*ixk* m_1 m_2

Instruction 064 reads one 64-bit word from Common Memory and enters it in the S_i register. The relative Common Memory location is determined by adding the content of register A_k to a 32-bit constant from the next 2 parcels in the instruction queue.

If the expression has a relative attribute of relocatable, it must be relative to a Common Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
064*ixk*				

INSTRUCTION 065

Result	Operand	Description	Machine Instruction
(a*k*,*exp*)	s_i	Write (s_i) to Common Memory at location (a_k)+*exp*	065*ixk* m_1 m_2

Instruction 065 writes one 64-bit word into Common Memory. The relative Common Memory location is determined by adding the content of the A_k register to a 32-bit constant from the next 2 parcels in the instruction queue. The 64-bit word is obtained from the S_i register.

If the expression has a relative attribute of relocatable, it must be relative to a Common Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
065*ixk*				

INSTRUCTION 066

Result	Operand	Description	Machine Instruction
s_i	(*exp*)	Read from Common Memory location *exp* to s_i	066*ixx* m_1 m_2

Instruction 066 reads one 64-bit word from Common Memory and enters it in the S_i register. The relative memory location is obtained from the next 2 parcels in the instruction queue.

If the expression has a relative attribute of relocatable, it must be relative to a Common Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
066*ixx*				

INSTRUCTION 067

Result	Operand	Description	Machine Instruction
(*exp*)	s_i	Write (s_i) to Common Memory at location *exp*	067*ixx* m_1 m_2

Instruction 067 writes one 64-bit word in the Common Memory. The relative Common Memory location is obtained from the next 2 parcels in the instruction queue. The data word is obtained from the S_i register.

If the expression has a relative attribute of relocatable, it must be relative to a Common Memory section.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
067*ixx*				

Result	Operand	Description	Machine Instruction
v_i	(a_j,a_k)	Read from Common Memory location (a_j) incremented by (a_k) to v_i	070*ijk*

Instruction 070 reads a vector stream of 64-bit words from Common Memory and enters it into the V_i register. The content of the VL register determines the length of the stream.

The first address for the Common Memory reference is formed by adding the content of the A_j register to the Background Processor base address. The following addresses for the Common Memory reference are separated by constant increments or decrements (strides). The stride is read from register A_k. A_k may contain positive, zero, or negative values.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
070*ijk*				

INSTRUCTION 071

Result	Operand	Description	Machine Instruction
(a_j,a_k)	v_i	Write (v_i) to Common Memory location (a_j) incremented by (a_k)	071*ijk*

Instruction 071 writes a vector stream of 64-bit words from the V_i register into Common Memory. The content of the VL register determines the length of the stream.

The first address for the Common Memory reference is formed by adding the content of the A_j register to the Background Processor base address. The following addresses for the Common Memory reference are separated by constant increments. The increment is read from register A_k.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
071*ijk*				

Result	Operand	Description	Machine Instruction
v_i	(a_k, v_j)	Gather from Common Memory locations $(a_k)+(v_j)$ to v_i	072*ijk*

Instruction 072 reads a vector stream of 64-bit words from Common Memory into the V_i register. The content of the VL register determines the length of the stream.

The relative Common Memory location is computed separately for each element of the vector. The content of the A_k register is read at the beginning of instruction execution and held in the Common Memory port. The content of the V_j register is then streamed to the Common Memory port. The high-order 32 bits of this data are discarded. The low-order 32 bits are used as components in the address calculation.

The first address for the Common Memory reference is formed by adding the first element of V_j data to A_k data and the Background Processor base address. The following addresses for the Common Memory reference are formed by adding the following elements of V_j data to the A_k data and the Background Processor base address.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
072*ijk*				

INSTRUCTION 073

Result	Operand	Description	Machine Instruction
(a_k, v_j)	v_i	Scatter (v_i) to Common Memory locations $(a_k)+(v_j)$	073*ijk*

Instruction 073 stores a vector stream of 64-bit words into Common Memory from the V_i register. The content of the VL register determines the length of the stream.

The relative Common Memory location is computed separately for each element of this vector stream. The content of the A_k register is read at the beginning of instruction execution and held in the Common Memory port. The content of the V_j register is then streamed to the Common Memory port. The high-order 32 bits of this data stream are discarded. The low-order 32 bits are used as components in the address calculation.

The first address for the Common Memory reference is formed by adding the first element of V_j data to A_k data and the Background Processor base address. The following addresses for the Common Memory reference are formed by adding the following elements of V_j data to the A_k data and the Background Processor base address.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
073*ijk*				

INSTRUCTION 074

Result	Operand	Description	Machine Instruction
v_i	$[a_k]$	Read from Local Memory location (a_k) to v_i	074*ixk*

Instruction 074 reads a stream of 64-bit words from Local Memory at consecutive locations. The initial Local Memory address is obtained from the A_k register. The data stream is entered into the V_i register. The content of the VL register determines the length of the stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
074*ixk*				

INSTRUCTION 075

Result	Operand	Description	Machine Instruction
$[a_k]$	v_i	Write (v_i) to Local Memory location (a_k)	075*ixk*

Instruction 075 stores a vector stream of 64-bit words into Local Memory at consecutive locations. The initial Local Memory address is obtained from the A_k register. The V_i register contains the data stream, and the content of the VL register determines the length of the stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
075*ixk*				

Result	Operand	Description	Machine Instruction
pass		Pass	076*xxx*
pass	*exp*	Pass	076*ijk*
		Executes same as 076*xxx*	077*xxx*

Instructions 076 and 077 issue without functional activity.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
076*xxx*				
076*ijk*				
077*xxx*				

Result	Operand	Description	Machine Instruction
s_i	s_j&s_k	Logical product of (s_j) and (s_k) to s_i	100*ijk*
s_i	#s_k&s_j	Logical product of (s_j) and complement (s_k) to s_i	101*ijk*
s_i	s_j\\s_k	Logical difference of (s_j) and (s_k) to s_i	102*ijk*
s_i	s_j!s_k	Logical sum of (s_j) and (s_k) to s_i	103*ijk*
s_i	s_j	S register copy (*j*=*k*)	103*ijj*

Instructions 100 through 103 perform scalar logical operations. The operands are obtained from registers S_j and S_k, and the result is returned to register S_i.

Instructions 100 and 101 read two 64-bit scalar operands and form the bit-by-bit logical product. Instruction 101 complements the S_k data before the logical product is formed.

Instruction 102 reads two 64-bit scalar operands and forms the bit-by-bit logical difference.

Instruction 103 reads two 64-bit scalar operands and forms the bit-by-bit logical sum.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
100*ijk*				
101*ijk*				
102*ijk*				
103*ijk*				
103*ijj*				

Result	Operand	Description	Machine Instruction
s_i	s_j+s_k	Integer sum of $(s_j)+(s_k)$ to s_i	104*ijk*
s_i	s_j-s_k	Integer difference of $(s_j)-(s_k)$ to s_i	105*ijk*

Instructions 104 and 105 perform integer arithmetic. The operands are obtained from registers S_j and S_k, and the result is returned to register S_i.

Instruction 104 reads two 64-bit scalar operands and forms the integer sum.

Instruction 105 reads two 64-bit scalar operands and forms the integer difference.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
104*ijk*				
105*ijk*				

Result	Operand	Description	Machine Instruction
s_i	ps_j	Population count of (s_j) to s_i	106*ij*0
s_i	qs_j	Population count parity of (s_j) to s_i	106*ij*1
s_i	zs_j	Leading zero count of (s_j) to s_i	107*ijx*

Instruction 106*ij*0 reads a 64-bit operand from the S_j register and forms a count of the number of 1 bits in the operand. This count is delivered as a positive integer to the S_i register.

Instruction 106*ij*1 counts the number of bits set to 1 in the S_j register. Then the low-order bit, showing the odd/even state of the result, is transferred to the low-order bit position of the S_i register. The high-order 63 bits are cleared. The actual population count is not transferred.

Instruction 107 reads a 64-bit operand from the S_j register and forms a count of the number of leading zeros in the operand. The operand is considered a field of 64 individual bits in this operation. The resulting count can have the values 0 through 64. The result is delivered to the S_i register as a positive integer.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
106*ij*0				
106*ij*1				
107*ijx*				

Result	Operand	Description	Machine Instruction
s_i	s_i<*exp*	Shift (s_i) left *exp*=64-*jk* places to s_i	110*ijk*
s_i	s_i>*exp*	Shift (s_i) right *exp*=*jk* places to s_i	111*ijk*

Instructions 110 and 111 shift 64-bit values in an S register by an amount specified by *jk*.

Instruction 110 reads a 64-bit operand from the S_i register, shifts the data to the left, and returns it to the S_i register. The number of bit positions in the shift count is a constant from the instruction parcel. This constant has a value 64 minus the low-order 6 bits in the parcel. The range of this constant is 1 through 64.

The data is shifted left in an open-ended manner. That is, zero bits are inserted from the right as bits shift off to the left. A shift count of 64 results in a word of all zeros.

Instruction 111 reads a 64-bit operand from the S_i register, shifts the data to the right, and returns it to the S_i register. The number of bit positions in the shift count is a constant from the instruction parcel. This constant has a value equal to the low-order 6 bits in the parcel. The range of this constant is 0 through 63.

The data is shifted right in an open-ended manner. That is, zero bits are inserted from the left as bits shift off to the right.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
110*ijk*				
111*ijk*				

Result	Operand	Description	Machine Instruction
s_i	$s_i,s_j<a_k$	Shift (s_i and s_j) left (a_k) places to S_i	112*ijk*
s_i	$s_j,s_i>a_k$	Shift (s_i and s_j) right (a_k) places to s_i	113*ijk*

Instructions 112 and 113 shift 128-bit values formed from two S registers. The data is shifted in an open-ended manner. That is, as bits shift off one end of the register, zeros are inserted in the other end.

Instruction 112 reads two 64-bit operands from registers S_i and S_j. The data is concatenated in a 128-bit field with the low-order bit of S_i next to the high-order bit of S_j data.

Instruction 113 reads two 64-bit operands from registers S_i and S_j. The data is concatenated in a 128-bit field with the low-order bit of S_j next to the high-order bit of S_i data.

The result field is taken from the 64-bit window corresponding to the original S_i data. The shift count is read from the A_k register. The A register content is treated as a 32-bit positive integer. Shift counts greater than or equal to 128 result in a zero data field; a shift count of 64 results in the S_j data; and a shift count of 0 results in the original S_i data.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
112*ijk*				
113*ijk*				

INSTRUCTION 114

Result	Operand	Description	Machine Instruction
s_i	vm	Transmit (vm) to s_i	114*ixx*

Instruction 114 reads the 64-bit mask from the VM register and enters it into the S_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
114*ixx*				

INSTRUCTION 115

Result	Operand	Description	Machine Instruction
s_i	rt	Transmit real-time count to s_i	115*ixx*

Instruction 115 reads the 64-bit real-time clock and enters the count into the S_i register.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
115*ixx*				

Result	Operand	Description	Machine Instruction
s_i	*exp*	Load s_i with a value	116*ijk*
s_i	*exp*,s	Load s_i with a 6-bit value	116*ijk*
s_i	*exp*,s,p	Load s_i with a 6-bit positive value	116*ijk*
s_i	*exp*	Load s_i with a value	117*ijk*
s_i	*exp*,s	Load s_i with a 6-bit value	117*ijk*
s_i	*exp*,s,m	Load s_i with a 6-bit positive negative value	117*ijk*

The S_i *exp* instruction will map into either a 050, 051, 052, 053, 116, or 117 opcode. If all the symbols within the expression have been previously defined within the currently enabled qualifier, CAL will map this instruction into the proper opcode with the fewest number of parcels into which the expression will fit. Otherwise, this instruction will be mapped into the 053 opcode.

CAL will map the S_i *exp*,S instruction into the 117 opcode if the expression is negative and has a relative attribute of absolute. Otherwise, this instruction will be mapped into the 116 opcode.

Instructions 116 and 117 form a 64-bit word from the *jk* data in the instruction parcel. The low-order 6 bits are copied from the instruction parcel. The result is delivered to the S_i register.

For instruction 116, the high-order bits are zeros.

For instruction 117, the high-order bits are ones.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
116*ijk*				
116*ijk*				
116*ijk*				
117*ijk*				
117*ijk*				
117*ijk*				

Result	Operand	Description	Machine Instruction
s_i	s_j+fs_k	Floating-point sum of (s_j) and (s_k) to s_i	120*ijk*
s_i	s_j-fs_k	Floating-point difference of (s_j) and (s_k) to s_i	121*ijk*

Instructions 120 and 121 perform floating-point arithmetic operations.

Instruction 120 forms the 64-bit floating-point sum of two 64-bit floating-point operands read from registers S_j and S_k. The result is delivered to the S_i register.

Instruction 121 forms the 64-bit floating-point difference of two 64-bit floating-point operands. The minuend is read from the S_j register and the subtrahend from the S_k register. The result is delivered to the S_i register.

Special case treatment of instructions 120 and 121 is described under Floating-point Add unit in the Background Processor section of this manual.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
120*ijk*				
121*ijk*				

Result	Operand	Description	Machine Instruction
s_i	fix,s_k	Convert (s_k) from floating-point to integer and enter into s_i	122*ixk*
s_i	flt,s_k	Convert (s_k) from integer to floating-point and enter into s_i	123*ixk*

Instructions 122 and 123 perform conversions between floating-point and integer (fixed-point) formats.

Instruction 122 reads a floating-point operand from the S_k register and delivers an integer result to the S_i register. The conversion from floating-point to integer is accomplished by adding the operand to a constant in the Floating-point Add unit. The result is then sign extended to form a 64-bit integer.

Instruction 123 reads an integer operand from the S_k register and delivers a floating-point result to the S_i register. The conversion from integer to floating-point is accomplished by adding the operand to a constant in the Floating-point Add unit.

Special case treatment of instructions 122 and 123 is described under Floating-point Add unit in the Background Processor section of this manual.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
122*ixk*				
123*ixk*				

Result	Operand	Description	Machine Instruction
s_i	s_j*fs_k	Floating-point product of (s_j) and (s_k) to s_i	124*ijk*
		Executes same as 124*ijk*	125*ijk*

Instruction 124 forms the 64-bit floating-point product of two 64-bit floating-point operands. The operands are read from registers S_j and S_k. The result is delivered to the S_i register.

Special case treatment of instruction 124 is described under Floating-point Multiply unit in the Background Processor section of this manual.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
124*ijk*				
125*ijk*				

Result	Operand	Description	Machine Instruction
s_i	s_j*is_k	Reciprocal iteration of 2-(s_j)*(s_k) to s_i	126*ijk*
s_i	s_j*qs_k	Reciprocal square root iteration of [3-(s_j)*(s_k)]/2 to s_i	127*ijk*

Instruction 126 forms the 64-bit floating-point quantity used in the reciprocal iteration algorithm. The operands are read from registers S_j and S_k. The result is delivered to the S_i register.

Instruction 127 forms a floating-point quantity used in the reciprocal square root iteration algorithm. The operands are read from registers S_j and S_k. The result is delivered to the S_i register.

See the description of Floating-point Multiply unit in the Background Processor section of this manual for details of this sequence.

****

CAUTION

Instruction 126 should be used only with the reciprocal approximation instruction (132), and instruction 127 should be used only with the reciprocal square root approximation instruction (133).

****

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
126*ijk*				
127*ijk*				

Result	Operand	Description	Machine Instruction
s_i	a_k	Transmit (a_k) to s_i with no sign extension	130*ixk*
s_i	$+a_k$	Transmit (a_k) to s_i with sign extension	131*ixk*

Instructions 130 and 131 read a 32-bit operand from the A_k register and transmit it to the S_i register.

Instruction 130 zero fills the high-order 32 bits, creating a 64-bit result.

Instruction 131 fills the high-order 32 bits with copies of bit 2^{31}, creating a 64-bit result.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
130*ixk*				
131*ixk*				

Result	Operand	Description	Machine Instruction
s_i	$/hs_j$	Floating-point reciprocal approximation of (s_j) to s_i	132*ijx*
s_i	$*qs_j$	Floating-point reciprocal square root approximation of (s_j) to s_i	133*ijx*

Instruction 132 forms a floating-point first approximation to the reciprocal of a floating-point operand. The operand is read from the S_j register, and the result is delivered to the S_i register.

Instruction 133 forms a floating-point first approximation to the reciprocal square root of a floating-point operand. The operand is read from the S_j register, and the result is delivered to the S_i register.

See the description of Floating-point Multiply unit in the Background Processor section of this manual for details of the sequence.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
132*ijx*				
133*ijx*				

INSTRUCTIONS 134 - 137

Result	Operand	Description	Machine Instruction
		Pass	134*xxx*
		Pass	135*xxx*
		Pass	136*xxx*
		Pass	137*xxx*

Instructions 134 through 137 issue without functional activity.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
134*xxx*				
135*xxx*				
136*xxx*				
137*xxx*				

INSTRUCTIONS 140 and 141

Result	Operand	Description	Machine Instruction
v_i	s_j&v_k	Logical products of (s_j) and (v_k) to v_i	140*ijk*
v_i	v_j&v_k	Logical products of (v_j) and (v_k) to v_i	141*ijk*

Instruction 140 reads a stream of vector elements from the V_k register, processes the data in the vector logical unit, and delivers a stream of result elements to register V_i. Data is read from the S_j register and is held in the vector logical unit during the streaming operation.

Instruction 141 reads two sets of vector elements, processes them in the vector logical unit and delivers result elements to register V_i. The source streams are from the V_j and V_k registers.

For both instructions, the VL register determines the number of operations performed. Each element of the vector is processed independent of the other elements in the stream. A bit-by-bit logical product is formed between the two source operands. The resulting 64 logical products are then delivered as one element to the destination stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
140*ijk*				
141*ijk*				

Result	Operand	Description	Machine Instruction
v_i	$s_j \backslash v_k$	Logical differences of (s_j) and (v_k) to v_i	142*ijk*
v_i	$v_j \backslash v_k$	Logical differences of (v_j) and (v_k) to v_i	143*ijk*

Instruction 142 reads a stream of vector elements from register V_k, processes the data in the vector logical unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the vector logical unit during the streaming operation.

Instruction 143 reads two streams of vector elements, processes them in the vector logical unit, and delivers a stream of result elements to register V_i. The source streams are from registers V_j and V_k.

For both instructions, the VL register determines the length of the operation. Each element of the vector stream is processed independent of the other elements in the stream. A bit-by-bit logical difference is formed between the two source operands. The resulting 64 logical differences are delivered as one element to the destination stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
142*ijk*				
143*ijk*				

Result	Operand	Description	Machine Instruction
v_i	$s_j!v_k$	Logical sums of (s_j) and (v_k) to v_i	144*ijk*
v_i	$v_j!v_k$	Logical sums of (v_j) and (v_k) to v_i	145*ijk*
v_i	v_j	v register copy $(j=k)$	145*ijj*

Instruction 144 reads a stream of vector elements from register V_k, processes the data in the Vector Logical unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Vector Logical unit during the streaming operation.

Instruction 145 reads two streams of vector elements, processes them in the Vector Logical unit, and delivers a stream of result elements to register V_i. The source streams are from registers V_j and V_k.

For both instructions, the VL register determines the length of the operation. Each element of the vector stream is processed independent of the other elements in the stream. A bit-by-bit logical sum is formed between the two source operands. The resulting 64 logical sums are delivered as one element to the destination stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
144*ijk*				
145*ijk*				
145*ijj*				

Result	Operand	Description	Machine Instruction
v_i	$s_j!v_k\&vm$	Transmit (s_j) if vm bit=1; (v_k) if vm bit=0 to v_i	146*ijk*

Instruction 146 reads a stream of vector elements in sequence from the V_k register, processes the data in the Vector Logical unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Vector Logical unit during the streaming operation. The content of the VL register determines the length of the vector stream.

The VM register works as a control mechanism to select either the S register data or the vector element data as each element arrives at the Vector Logical functional unit. A bit of VM register data is associated with each element. The high-order bit of VM data is associated with the first vector element. The following bits of VM register data correspond with the following vector elements. The S register data is selected as a result element if the VM register contains a 1 in the designated element position. The V_k register element is selected as a result element if the VM register contains a 0 in the designated element position.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
146*ijk*				

INSTRUCTION 147

Result	Operand	Description	Machine Instruction
v_i	v_j!v_k&vm	Transmit (v_j) if vm bit=1; (v_k) if vm bit=0 to v_i	147*ijk*

Instruction 147 reads two streams of vector elements, processes them in the Vector Logical unit, and delivers a stream of result elements to the V_i register. The source streams are from registers V_j and V_k. The content of the VL register determines the length of each vector stream.

The VM register works as a control mechanism to select either the V_j data or the V_k data as each element pair arrive at the Vector Logical unit. A bit of VM register data is associated with each element. The high-order bit of VM data is associated with the first vector element. The following bits of VM register data correspond with the following vector elements. The V_j data is selected as a result element if the VM register contains a 1 in the designated element position. The V_k register element is selected as a result element if the VM register contains a 0 in the designated element position.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
147*ijk*				

INSTRUCTIONS 150 and 151

Result	Operand	Description	Machine Instruction
v_i	$v_j<a_k$	Shift (v_j) left (a_k) bits with zero fill, results to v_i	150*ijk*
v_i	$v_j>a_k$	Shift (v_j) right (a_k) bits with zero fill, results to v_i	151*ijk*

Instructions 150 and 151 read a stream of vector elements in sequence from the V_j register, process the data in the Vector Integer unit, and deliver a stream of result elements to the V_i register. Data is read from the A_k register and is held in the Vector Integer unit during the streaming operation. The content of the VL register determines the length of the vector stream.

Instruction 150 shifts data to the left and instruction 151 shifts data to the right. Each element of the vector stream is processed independent of the other elements in the stream. Each element is shifted by the number of bit positions indicated by the A_k register value. Zero bits are inserted as bits shift off.

The content of the A_k register is treated as a 32-bit positive integer. Shift counts equal to or greater than 64 cause a zero data field.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
150*ijk*				
151*ijk*				

INSTRUCTIONS 152 and 153

Result	Operand	Description	Machine Instruction
v_i	$v_j, v_j < a_k$	Double shift (v_j) left (a_k) places to V_i	152*ijk*
v_i	$v_j, v_j > a_k$	Double shift (v_j) right (a_k) places to v_i	153*ijk*

Instructions 152 and 153 process the elements of data from the V_j register in pairs for this sequence. Each element is concatenated with the following element and the resulting 128-bit field is shifted by the number of bit positions in the A_k register data. A 64-bit field from the original element window is then delivered to the destination vector stream.

Instruction 152 shifts data to the left. The first element of V_j data is positioned in the high-order 64 bits of the 128-bit shift field. The second element of V_j data is positioned in the low-order 64 bits of the 128-bit shift field. The 128-bit field then shifts left by the amount of the shift count. A first result element is read from that portion of the 128-bit field originally occupied by the first element of data.

The second element of V_j data is then positioned in the higher portion of the 128-bit shift field. The third element of V_j data is entered in the low-order 64 bits of the field. This 128-bit field is then shifted left by the amount of the shift count. A second result element is read from the high-order 64 bits of the 128-bit field originally occupied by the second element of data.

This process continues until the last element of data is entered in the high-order 64 bits of the 128-bit shift field. A zero field is entered in the low-order 64 bits. This 128-bit field is then shifted left by the amount of the shift count. The last result element is read from the upper portion of the shift field.

The A_k register content is treated as a 32-bit positive integer. Shift counts greater than 128 result in a zero data field. Zero bits are inserted at the right end of the 128-bit shift field as bits are shifted off to the left.

Instruction 153 shifts data to the right. The first element of V_j data is positioned in the low-order 64 bits of the 128-bit shift field. The high-order 64 bits of the 128-bit shift field is cleared. The 128-bit field then shifts to the right by the amount of the shift count. A first result element is read from the low-order 64 bits of the 128-bit field originally occupied by the first element of data.

The second element of V_j data is then positioned in the lower portion of the 128-bit shift field. The first element of V_j data is entered in the high-order 64 bits of the field. This 128-bit field is then shifted right by the amount of the shift count. A second result element is read from the low-order 64 bits of the 128-bit field originally occupied by the second element of data.

This process continues until the last element of data is entered in the low-order 64 bits of the 128-bit shift field. The preceding element is entered in the high-order 64 bits. This 128-bit field is then shifted right by the amount of the shift count. The last result element is read from the low-order 64 bits of the field.

The A_k register content is treated as a 32-bit positive integer. Shift counts greater than 128 result in a zero data field. 0 bits are inserted at the left end of the 128-bit shift field as bits are shifted off to the right.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
152*ijk*				
153*ijk*				

INSTRUCTION 154

Result	Operand	Description	Machine Instruction
v_i	s_j*fv_k	Floating-point product of (s_j) and (v_k) to v_i	154*ijk*

Instruction 154 reads a stream of vector elements in sequence from the V_k register, processes the data in the Floating-point Multiply unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Floating-point Multiply unit during the streaming operation. The content of the VL register determines the length of the vector stream.

Each element of the vector stream is processed independent of the other elements in the stream. The Floating-point Multiply unit forms the 64-bit floating-point product of the arriving vector element and the scalar operand held in the unit. The result element is delivered to the V_i register. See the description of Floating-point Multiply unit for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
154*ijk*				

Result	Operand	Description	Machine Instruction
v_i	v_j*fv_k	Floating-point product of (v_j) and (v_k) to v_i	155*ijk*

Instruction 155 reads two streams of vector elements, processes them in the Floating-point Multiply unit, and delivers a result stream to the V_i register. The source streams are from registers V_j and V_k. The VL register determines the length of each vector stream.

Each element of the vector stream is processed independent of the other elements in the stream. The Floating-point Multiply unit forms the 64-bit floating-point product of the arriving vector elements. The result element is delivered to the V_i register. See the description of Floating-point Multiply unit in the Background Processor section of this manual for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
155*ijk*				

Result	Operand	Description	Machine Instruction
v_i	v_j*iv_k	Reciprocal iteration of 2-(v_j)*(v_k) to v_i	156*ijk*
v_i	v_j*qv_k	Reciprocal square root iteration of [3-(v_j)*(v_k)]/2 to v_i	157*ijk*

Instructions 156 and 157 read two streams of vector elements, process them in the Floating-point Multiply unit, and deliver a result stream to the V_i register. The source streams are from registers V_j and V_k. The content of the VL register determines the length of each vector stream.

For instruction 156, the Floating-point Multiply unit forms a 64-bit floating-point quantity used in the reciprocal iteration algorithm from each pair of arriving vector elements.

For instruction 157, the Floating-point Multiply unit forms a 64-bit floating-point quantity used in the reciprocal square root iteration algorithm from each pair of arriving elements.

See the description of Floating-point Multiply unit in section 2 for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
156*ijk*				
157*ijk*				

Result	Operand	Description	Machine Instruction
v_i	s_j+v_k	Integer sums of (s_j) and (v_k) to v_i	160*ijk*
v_i	v_j+v_k	Integer sums of (v_j) and (v_k) to v_i	161*ijk*

Instruction 160 reads a stream of vector elements from the V_k register, processes the data in the Vector Integer unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Vector Integer unit during the streaming operation.

Instruction 161 reads two streams of vector elements, processes them in the Vector Integer unit, and delivers a stream of result elements to the V_i register. The source streams are from registers V_j and V_k.

For both instructions, the VL register determines the length of the vector stream. Each element of the vector stream is processed independent of the other elements in the stream. The Vector Integer unit forms the integer sum of the two operands. The result is delivered as one element of the destination stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
160*ijk*				
161*ijk*				

Result	Operand	Description	Machine Instruction
v_i	$s_j - v_k$	Integer differences of (s_j) and (v_k) to v_i	162*ijk*
v_i	$v_j - v_k$	Integer differences of (v_j) and (v_k) to v_i	163*ijk*

Instruction 162 reads a stream of vector elements from V_k register, processes the data in the Vector Integer unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Vector Integer unit during the streaming operation.

Instruction 163 reads two streams of vector elements, processes them in the Vector Integer unit, and delivers a stream of result elements to the V_i register. The source streams are from registers V_j and V_k.

For both instructions, the VL register determines the length of the vector stream. Each element of the vector stream is processed independent of the other elements in the stream. The Vector Integer unit forms the integer difference of the two operands. The result is delivered as one element of the destination stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
162*ijk*				
163*ijk*				

Result	Operand	Description	Machine Instruction
v_i	pv_j	Population counts of (v_j) to v_i	164*ij*0
v_i	qv_j	Population count parity of (v_j) to v_i	164*ij*1
v_i	zv_j	Leading zero count of (v_j) to v_i	165*ijx*

Instruction 164 reads a stream of vector elements in sequence from the V_j register, processes the data in the Vector Integer unit, and delivers a stream of result elements to the V_i register. The content of the VL register determines the length of the vector stream.

Each element of the vector stream is processed independent of the other elements in the stream. The Vector Integer unit counts the number of one bits in each vector element and delivers the count as a positive integer to the result stream.

Instruction 164*ij*0 counts the number of bits set to 1 in each element of V_j and enters the results into corresponding elements of V_i. The results are entered into the low-order 7 bits of each V_i element; the remaining high-order bits of each V_i element are zeroed.

Instruction 164*ij*1 counts the number of bits set to 1 in each element of V_j. The least significant bit of each result shows whether the result is an odd or even number. Only the least significant bit of each result is transferred to the least significant bit position of the corresponding element of register V_i. The remainder of the result is set to zeroes. The actual population count results are not transferred.

Instruction 165*ijx* reads a stream of vector elements in sequence from the V_j register, processes the data in the Vector Integer unit, and delivers a stream of result elements to the V_i register. The content of the VL register determines the length of the vector stream.

Each element of the vector stream is processed independent of the other elements in the stream. The Vector Integer unit counts the number of leading zeros in each element. The element is considered as a field of 64 individual bits in this operation. This count is delivered as a positive integer to the result stream.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
164*ij*0				
164*ij*1				
165*ijx*				

Result	Operand	Description	Machine Instruction
v_i	$/hv_k$	Floating-point reciprocal approximations of (v_k) to v_i	166*ixk*
v_i	$*qv_k$	Floating-point reciprocal square root approximations of (v_k) to v_i	167*ixk*

Instruction 166 and 167 read a stream of vector elements in sequence from the V_k register, process the data in the Floating-point Multiply unit, and deliver a stream of result elements to the V_i register. The content of the VL register determines the length of the vector stream. See the description of the Floating-point Multiply unit in section 2 for details of this sequence.

For instruction 166, the Floating-point Multiply unit forms a floating-point quantity which is a first approximation to the reciprocal of the arriving vector element.

For instruction 167, the Floating-point Multiply unit forms a floating-point quantity which is a first approximation to the reciprocal square root of the arriving vector element.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
166*ixk*				
167*ixk*				

Result	Operand	Description	Machine Instruction
v_i	s_j+fv_k	Floating-point sum of (s_j) and (v_k) to v_i	170*ijk*
v_i	v_j+fv_k	Floating-point sum of (v_j) and (v_k) to v_i	171*ijk*

Instruction 170 reads a stream of vector elements in sequence from the V_k register, processes the data in the Floating-point Add unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Floating-point Add unit during the streaming operation.

Instruction 171 reads two streams of vector elements, processes them in the Floating-point Add unit, and delivers a result stream to the V_i register. The source streams are from registers V_j and V_k.

For both instructions, the content of the VL register determines the length of the vector stream. Each element of the vector stream is processed independent of the other elements in the stream. The Floating-point Add unit forms the 64-bit floating-point sum of the two operands. The result is delivered to register V_i. See the description of Floating-point Add unit for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
170*ijk*				
171*ijk*				

Result	Operand	Description	Machine Instruction
v_i	s_j-fv_k	Floating-point difference of (s_j) and (v_k) to v_i	172*ijk*
v_i	v_j-fv_k	Floating-point difference of (v_j) and (v_k) to v_i	173*ijk*

Instruction 172 reads a stream of vector elements in sequence from the V_k register, processes the data in the Floating-point Add unit, and delivers a stream of result elements to the V_i register. Data is read from the S_j register and is held in the Floating-point Add unit during the streaming operation.

Instruction 173 reads two streams of vector elements, processes them in the Floating-point Add unit, and delivers a result stream to the V_i register. The source streams are from registers V_j and V_k.

For both instructions, the content of the VL register determines the length of the vector stream. Each element of the vector stream is processed independent of the other elements in the stream. The Floating-point Add functional unit forms the 64-bit floating-point difference of the two operands. The result is delivered to register V_i. See the description of Floating-point Add unit for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
172*ijk*				
173*ijk*				

Result	Operand	Description	Machine Instruction
v_i	fix,v_k	Integer form of floating-point (v_k) to v_i	174*ixk*
v_i	flt,v_k	Floating-point form of integer (v_k) to v_i	175*ixk*

Instructions 174 and 175 read a stream of vector elements in sequence from the V_k register, process the data in the Floating-point Add unit, and deliver a stream of result elements to the V_i register. The content of the VL register determines the length of the vector stream.

Instruction 174 performs the conversion from floating-point to integer format by adding the operand to a constant in the Floating-point Add unit. The result is sign extended to form a 64-bit integer.

Instruction 175 performs the conversion from integer to floating-point format by adding the operand to a constant in the Floating-point Add unit. The result is delivered to the V_i register.

See the description of Floating-point Add unit for details and special case treatment.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
174*ixk*				
175*ixk*				

Result	Operand	Description	Machine Instruction
v_i	ci,s_j&s_k	Enter v_i with compressed iota s_j and s_k	176*ijk*
		Executes same as 176*ijk*	177*xxx*

Instruction 176 forms a vector from two scalar operands. The first scalar operand is a 64-bit mask from the S_j register. The second scalar operand is a 32-bit vector stride from the S_k register. The stride is taken from the low-order 32 bits of the S_k register data.

The Vector Integer unit forms a 64-element iota vector from the stride. This is a vector whose first element has a zero value, and whose subsequent elements are spaced by the stride increment. The sequence of element values is then as follows.

$0*S_k$, $1*S_k$, $2*S_k$, $3*S_k$, $4*S_k$, $5*S_k$, etc.

The two scalar operands are captured and held in the Vector Integer unit. The S_k value is repeatedly added to the accumulated sum to form the iota vector. The 64-bit mask is shifted to the left 1 bit position per clock period. The Vector Integer unit then compresses the iota vector, using the mask data, and delivers the resulting vector to register V_i.

An element of the iota vector is delivered to the result vector where there is a 1 bit in the mask. An element of the iota vector is skipped, and the position compressed, where there is a 0 bit in the mask. The resulting vector has the same number of elements as there were one bits in the mask.

The first mask bit tested is the high-order bit. Bits are then tested in order to the low-order bit. A zero test is made on the remaining mask bits to stop the sequence. Execution time is then variable depending on the mask content.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
176*ijk*				
177*xxx*				

4. COMMON MEMORY

Common Memory contains 256 million words of dynamic memory. The dynamic memory consists of 128 banks with 2 million words in each bank. Each 72-bit word consists of 64-data bits and 8 error correction bits.

Common Memory is organized into quadrants with 32 banks in each quadrant. Each memory quadrant has a data path to each of four Common Memory ports. A Background Processor and a foreground communication channel are connected to each Common Memory port. Total memory bandwidth is 64 gigabits per second. Total memory capacity is 17 gigabits.

The Foreground Processor, Background Processors, and disk controllers share Common Memory. Common Memory contains program code for the Background Processors, data for problem solution, and Foreground Processor system tables.

4.1 MEMORY ADDRESSING

A word in memory is addressed by 32 bits. The low-order 2 bits select the quadrants and the next 5 bits select the bank. Figure 4-1 illustrates the format of the memory address for Common Memory.

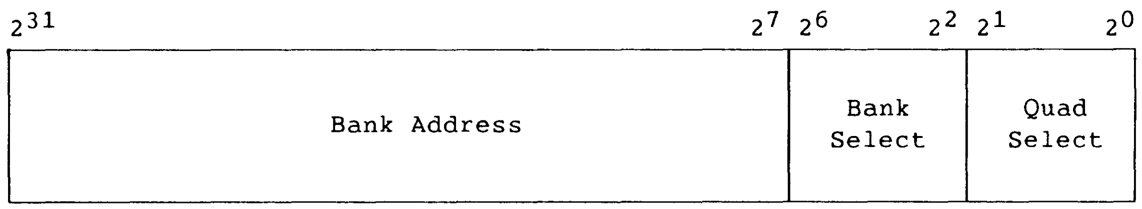


Figure 4-1. Memory Address for Common Memory

4.2 MEMORY ACCESS

The Background Processors are locked into a phased access time scheme with the memory quadrants through the Common Memory ports. Through its Common Memory port, a Background Processor can access any given quadrant but only in the processor's own phase time, that is, every fourth clock period (CP). If a Background Processor requests a quadrant out of its phase time, the request is delayed until the correct time.

For example, assume the Background Processors are A through D, and the quadrants are 0 through 3. Also assume processor A is locked into quadrant 0 at phase time 0. If processor A references quadrant 0 at phase time 1, it must wait until the next phase time 0 (CP 4) to have access to memory in that quadrant.

Memory banks in a quadrant share a data path to each Common Memory port. Because of the phased access time between the quadrants and the Common Memory ports, however, only one bank accesses the path in a given 4-CP time slot. Because two banks never compete for the same data path in the same time slot, each bank functionally has an independent path to each of the four Common Memory ports.

4.3 MEMORY CONFLICTS

To prevent memory conflicts, each memory bank has a Bank Busy flag. If the bank is busy, the quadrant sends a rejected signal to the requesting memory port. The requesting port retries the data.

4.4 MEMORY BACKUP

Memory backup occurs when too many memory references arrive at a single memory quadrant. Each Common Memory port has four quadrant buffers, one for each quadrant, each buffer can hold two memory references for its memory quadrant. Therefore, references can continue to the memory port when the reference is not in the proper phase time. When a quadrant buffer in a memory port is filled, and another reference to that quadrant is made, the memory port begins a backup procedure.

The memory port backup procedure stops instruction issue for the associated Background Processor if that processor is making a memory reference. Vector streams initiated in the Background Processor and associated with a Common Memory reference are held.

After all references have been submitted for retry, a stop issue is released allowing additional references to issue. A conflict during the retry process causes the backup procedure to begin again at the point the conflict occurred; which could be the original backup references or additional new references filling buffer positions that became empty during retry.

NOTE

A special timing problem exists for execution of Background Processor instruction 072 (the gather instruction). This instruction allows addresses in any sequence with respect to the low-order 2 bits, quadrant select. Without special treatment of this instruction, the data could arrive at the Vector Destination register out of order. Therefore, the hardware forces a maximum memory reference pattern of four references and 12 null references which averages to one reference every 4 clock periods.

4.5 MEMORY ERROR CORRECTION

A single error correction/double error detection (SECDED) network is used between the Background Processors and memory. SECDED assures that data written into memory is returned to the Background Processors with consistent precision.

Using SECDED, the single error alteration is automatically corrected if a single bit of a data word is altered before the data word is passed to the computer. If 2 bits of the same data word are altered, the double error is detected but not corrected. In either case, the Background Processors can be interrupted, depending on interrupt options selected, to allow processing of the error. For 3 or more bits in error, results are ambiguous.

The 8 check bits and the data word are stored in memory at the same location. When read from memory, the 64-bit matrix, illustrated in figure 4-2, is used to generate a new set of check bits, which are compared with the old check bits that were stored in memory. The resulting 8 comparison bits are called syndrome bits (S bits). The states of these S bits are symptomatic of any error that occurred (1 = no compare). If all syndrome bits are 0, no memory error is assumed.

The matrix is designed so that:

- If all syndrome bits are 0, no error is assumed.
- If only 1 syndrome bit is 1, the associated check bit is in error.
- If more than 1 syndrome bit is 1 and the parity of all syndrome bits is odd, then a single correctable error is assumed to have occurred. The syndrome bits can be decoded to identify the bit in error.
- If 3 or more memory bits are in error, the parity of all syndrome bits is odd and results are ambiguous.
- If more than 1 syndrome bit is 1 and the parity of all syndrome bits S0 through S7 is even, then a double error (or an even number of bit errors) occurred within the data bits or check bits.

	CHECK BYTE							
	2^{71}	2^{70}	2^{69}	2^{68}	2^{67}	2^{66}	2^{65}	2^{64}
check bit 0								x
check bit 1							x	
check bit 2						x		
check bit 3					x			
check bit 4				x				
check bit 5			x					
check bit 6		x						
check bit 7	x							

	2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}	2^{57}	2^{56}
check bit 0								
check bit 1	x	x	x	x	x	x	x	x
check bit 2	x	x	x	x	x	x	x	x
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}	2^{48}
check bit 0	x	x	x	x	x	x	x	x
check bit 1								
check bit 2	x	x	x	x	x	x	x	x
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{47}	2^{46}	2^{45}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}
check bit 0	x	x	x	x	x	x	x	x
check bit 1	x	x	x	x	x	x	x	x
check bit 2								
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{39}	2^{38}	2^{37}	2^{36}	2^{35}	2^{34}	2^{33}	2^{32}
check bit 0	x	x	x	x	x	x	x	x
check bit 1	x	x	x	x	x	x	x	x
check bit 2	x	x	x	x	x	x	x	x
check bit 3								
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4								
check bit 5	x	x	x	x	x	x	x	x
check bit 6	x	x	x	x	x	x	x	x
check bit 7	x	x	x	x	x	x	x	x

	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5								
check bit 6	x	x	x	x	x	x	x	x
check bit 7	x	x	x	x	x	x	x	x

	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5	x	x	x	x	x	x	x	x
check bit 6								
check bit 7	x	x	x	x	x	x	x	x

	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5	x	x	x	x	x	x	x	x
check bit 6	x	x	x	x	x	x	x	x
check bit 7								

1270

Figure 4-2. Error Correction Matrix

5. FOREGROUND SYSTEM

The CRAY-2 computer contains a foreground system to control and monitor system operations. The Foreground Processor contains the following:

- Four high-speed synchronous communication channels to interconnect the Background Processors, Foreground Processor, disk controllers, and Front-end Interfaces (FEIs)
- Foreground channel ports
  - Four Common Memory ports to control data transfer between Common Memory and the Foreground Processor, disk storage units, and the FEI modules
  - Four Background Processor ports to allow the Foreground Processor to monitor and control the Background Processors
- Up to 40 I/O devices can be attached
  - Disk controllers to control up to 36 disk storage units
  - Interfaces to connect the CRAY-2 mainframe to the 6 Mbyte per second channels or Network Systems Corporation (NSC) HYPERchannels
- A Foreground Processor to supervise overall system activity and respond to requests for interaction among the system members
- A maintenance control console to deadstart the CRAY-2 mainframe and monitor system operation

5.1 FOREGROUND COMMUNICATION CHANNELS

Four high-speed communication channels in the foreground system link the Common Memory, Background Processors, Foreground Processor, disk controllers, and FEIs. The Foreground Processor supervises the four channels. Data blocks are generally 512 Common Memory words.

Each channel accesses one Common Memory port and one Background Processor port. Each channel in the system can have up to four Front-end Interfaces. Disk controllers are generally divided equally among the channels. The disk controller configuration, however, can be adjusted for special system requirements.

A channel interconnects the Foreground Processor, disk controllers, FEI modules, a Background Processor port, and a Common Memory port in a continuous channel loop. A configuration of a single channel loop is shown in figure 5-1.

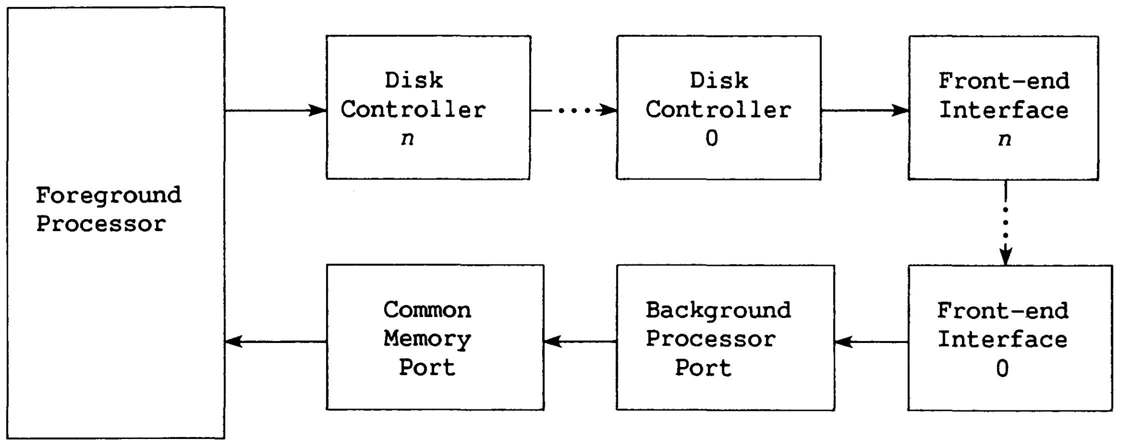


Figure 5-1. Channel Loop

Each member of the loop is called a channel node. Each channel node receives data on the path during each clock period and transmits that data to the next node in the following clock period. Data can then move about the loop from any transmitting node to any receiving node.

5.2 FOREGROUND CHANNEL PORTS

Two independent sets of channel ports exist in the Foreground Processor: Common Memory ports and Background Processor ports. The Common Memory ports contain controls and status information for transfer of data to and from Common Memory. The Background Processor ports contain controls and status information used by the Foreground Processor to control the Background Processors.

5.2.1 COMMON MEMORY PORTS

The foreground system contains four Common Memory ports. One Common Memory port is associated with each of the four Background Processors. A foreground channel is associated with each of the Common Memory ports. The Foreground Processor makes Common Memory requests through the Common Memory port for those foreground devices on the same channel. Background

Processor Common Memory requests have priority over foreground system requests. There is one exception; the refresh has priority over the background operand references. The Common Memory port accepts requests according to the following priority scheme, from highest to lowest priority:

1. Background Processor operand references
2. Background Processor instruction references
3. Foreground channel transfer references

5.2.2 BACKGROUND PROCESSOR PORTS

Each Background Processor has a Background Processor port connecting it to one of the four channels in the foreground system. This port allows the Foreground Processor to control the operation of the Background Processor.

5.3 DISK STORAGE UNITS

The Foreground Processor spends considerable time transferring data between the disk storage units and Common Memory. The system has provision for 36 disk storage units. Control for these units is on an individual disk unit basis so that all 36 units can operate concurrently.

5.3.1 DISK SYSTEM ORGANIZATION

The disk storage units can be addressed as individual storage units, but problems arise with this approach: the data transfer rate for individual files, the rotational latency of the disk units, and the reliability of mechanical devices.

The disk storage system on the CRAY-2 computer has the option of operating in a synchronous mode with all disk units running in parallel in a lockstep mode. For this approach to be practical, the buffer size for individual disk references must be about 100,000 words.

A system configuration with 16 disk storage units can illustrate the synchronous mode of operation. The Foreground Processor is given a Disk address consisting of a pseudo-track number. This number is the cylinder and head group for a disk file with no flaws. A table look-up converts this pseudo-track into a physical track for each disk unit. All disk storage units are positioned in parallel.

The Foreground Processor reads angular position for each disk surface to determine the sector currently under the recording head. It then begins a data stream from Common Memory to disk surfaces, choosing the portion of the Common Memory buffer appropriate for the current angular position of each disk storage unit. Data to 15 of the disk storage units is directly from the Common Memory buffer. Data for the 16th disk storage unit is a logical difference data stream using the word-by-word data from the desired file. All 16 disk storage units write one track of data as the basic reservation unit.

On data readback, the 16th disk is read concurrently with the other 15 disks. If the fire code detectors indicate no data errors, the 16th disk data is discarded. If an error has occurred, it can be corrected without time loss in the data stream.

The overhead introduced by this arrangement is one disk storage unit for every 15 disks required. The following three benefits occur:

- The data rate is 525 megabits per second instead of 35 megabits per second.
- The disk storage unit rotational latency has gone to 1/2 of a sector time for Foreground Processor single disk I/O.
- A disk storage unit can fail completely due to a head crash or motor failure with no loss of data or time.

A disk failure in this system can be corrected during system operation by removing the defective file and replacing it with another unit. The new unit can then be brought on line by running a background job that takes 2.5 minutes of disk system time to record the faulty unit data from the data on the other 15 files.

5.4 FRONT-END INTERFACE

The CRAY-2 mainframe is connected to a front-end computer system through an interface in the foreground system. The FEI can support a 6 Mbyte per second channel or an NSC HYPERchannel. Each channel loop can hold up to four interfaces.

Each interface contains a 512 64-bit word buffer. The data block can be of arbitrary word length up to this limit.

5.5 FOREGROUND PROCESSOR

The Foreground Processor supervises system operation by responding to Background Processor requests and sequencing Channel Communication signals. The user programs reside in the Common Memory in a protected area and are executed in Background Processors.

The Foreground Processor code is loaded at deadstart from a diskette at the maintenance control console. (The maintenance control console is described later in this section.) The code is firmware and is not altered during the operation of the system.

****

CAUTION

A Foreground Processor program code error is as fatal to system operation as a hardware failure.

****

The primary functions of the Foreground Processor program are real-time response to various signals from a variety of sources in the foreground system. As many as 50 simultaneous real-time sequences can be operating in an interleaved manner in the Foreground Processor. Many of these responses must be of the order of a microsecond or less.

The Foreground Processor contains the following sections:

- Instruction Memory
- Local Data Memory
- Arithmetic functions
- Real-time clock
- Error checking
- Instruction issue mechanism
- Instruction set

The Foreground Processor performs arithmetic functions on 32-bit integers. The following functions are performed.

- Add
- Subtract
- Shift left, open ended
- Shift right, open ended
- Logical product
- Logical difference
- Logical sum

A detailed description of the Foreground Processor and its functional units is beyond the scope of this manual. The Foreground Processor is transparent to the user of the CRAY-2 Computer System.

5.6 MAINTENANCE CONTROL CONSOLE

The maintenance control console is used to deadstart the system and to exchange data with the Foreground Processor. Instructions for execution in the Foreground Processor are loaded into the Foreground Instruction Memory at deadstart from a diskette at the maintenance control console. This memory is a Read-only Memory during system operation. Data for supervision of the system is maintained in Common Memory and is moved to the Foreground Processor Local Memory as required.

APPENDIX SECTION

A. SYMBOLIC MACHINE INSTRUCTIONS LISTED BY FUNCTIONALITY

A.1 SYMBOLIC NOTATION

This appendix lists the symbolic machine instructions by functionality. Instructions are described in the following functional categories:

- Branch instructions
- Pass instructions
- Semaphore instructions
- Register entry instructions
- Inter-register transfer instructions
- Memory transfer instructions
- Integer arithmetic operation instructions
- Floating-point arithmetic operation instructions
- Logical operation instructions
- Bit count instructions
- Shift operation instructions

Instructions are listed in numerical order and explained in section 3 of this manual. The octal machine code may be used to cross-reference instructions in this appendix to their descriptions in section 3. For descriptions of functional units, refer to section 2 of this manual.

Register Entry Instructions

a_i	*exp*	s_i	*exp*
a_i	*exp*,s	s_i	*exp*,s
a_i	*exp*,s,p	s_i	*exp*,s,p
a_i	*exp*,s,m	s_i	*exp*,s,m
a_i	*exp*,p	s_i	*exp*,h
a_i	*exp*,p,p	s_i	*exp*,h,p
a_i	*exp*,p,m	s_i	*exp*,h,m
a_i	*exp*,h	s_i	*exp*,l
		s_i	*exp*,f

Integer Arithmetic Operations

a_i	a_j+a_k	a_i	a_j-a_k	a_i	a_j*a_k
s_i	s_j+s_k	s_i	s_j-s_k		
v_i	s_j+v_k	v_i	s_j-v_k		
v_i	v_j+v_k	v_i	v_j-v_k	v_i	ci,s_j&s_k

Inter Register Transfers

a_i	s_j	s_i	a_k
		s_i	+a_k
s_i	s_j	v_i	v_j
a_i	vl	vl	a_k
s_i	vm	vm	s_j
s_i	rt		

Floating Point Operations

s_i	s_j+fs_k	s_i	s_j-fs_k	s_i	s_j*fs_k
v_i	s_j+fv_k	v_i	s_j-fv_k	v_i	s_j*fv_k
v_i	v_j+fv_k	v_i	v_j-fv_k	v_i	v_j*fv_k
s_i	s_j*is_k	s_i	fix,s_k	s_i	s_j*qs_k
v_i	v_j*iv_k	v_i	fix,v_k	v_i	v_j*qv_k
s_i	/hs_j	s_i	flt,s_k	s_i	*qs_j
v_i	/hv_k	v_i	flt,v_k	v_i	*qv_k

dfi efi

Bit Count Instructions

s_i	ps_j	v_i	pv_j
s_i	qs_j	v_i	qv_j
s_i	zs_j	v_i	zv_j

Logical Operations

s_i	s_j&s_k	s_i	s_j!s_k	s_i	$s_j \backslash s_k$
v_i	s_j&v_k	v_i	s_j!v_k	v_i	$s_j \backslash v_k$
v_i	v_j&v_k	v_i	v_j!v_k	v_i	$v_j \backslash v_k$
s_i	#s_k&s_j			vm	v_k,z
				vm	v_k,n
v_i	s_j!v_k&vm			vm	v_k,p
v_i	v_j!v_k&vm			vm	v_k,m

Shift Instructions

s_i	s_i<*exp*	s_i	s_i>*exp*
v_i	v_j<a_k	v_i	v_j>a_k
s_i	s_i,s_j<a_k	s_i	s_j,s_i>a_k
v_i	v_j,v_j<a_k	v_i	v_j,v_j>a_k

Pass Instructions

pass	pass	*exp*

Semaphore Instructions

csm	ssm

Memory Transfers

a_i	[*exp*]	[*exp*]	a_k
a_i	[a_k]	[a_k]	a_j
s_i	[*exp*]	[*exp*]	s_j
s_i	[a_k]	[a_k]	s_i
v_i	[a_k]	[a_k]	v_i
s_i	(*exp*)	(*exp*)	s_i
s_i	(a_k)	(a_k)	s_i
s_i	(a_k,*exp*)	(a_k,*exp*)	s_i
s_i	(a_j,a_k)	(a_j,a_k)	s_i
v_i	(a_j,a_k)	(a_j,a_k)	v_i
v_i	(a_k,v_j)	(a_k,v_j)	v_i

dri eri

Branch Instructions

jz	a_k,*exp*	jz	s_j,*exp*
jn	a_k,*exp*	jn	s_j,*exp*
jp	a_k,*exp*	jp	s_j,*exp*
jm	a_k,*exp*	jm	s_j,*exp*
jcs	*exp*	j	a_k
jss	*exp*	r,a_i	a_k
j	*exp*		
err		exit	
		exit	*exp*

1

1342

A.2 BRANCH INSTRUCTIONS

A.2.1 CONDITIONAL BRANCHES

Result	Operand	Description	Machine Instruction
jz	a_k,*exp*	Branch if (a_k) is zero	010*xxk*
jn	a_k,*exp*	Branch if (a_k) is nonzero	011*xxk*
jp	a_k,*exp*	Branch if (a_k) is positive	012*xxk*
jm	a_k,*exp*	Branch if (a_k) is negative	013*xxk*
jz	s_j,*exp*	Branch if (s_j) is zero	014*xjx*
jn	s_j,*exp*	Branch if (s_j) is nonzero	015*xjx*
jp	s_j,*exp*	Branch if (s_j) is positive	016*xjx*
jm	s_j,*exp*	Branch if (s_j) is negative	017*xjx*
jcs	*exp*	Jump to constant parcel if Semaphore clear; set Semaphore	004*xxx*
jss	*exp*	Jump to constant parcel if Semaphore is set; set Semaphore	005*xxx*

A.2.2 UNCONDITIONAL JUMPS

Result	Operand	Description	Machine Instruction
j	*exp*	Unconditional jump	003*xxx*
r,a_i	a_k	Register jump to (a_k) with return address to a_i	002*ixk*
j	a_k	Register jump to (a_k), value is a_k erased	002*kxk*

A.2.3 EXITS

Result	Operand	Description	Machine Instruction
err		Error exit	000*x*00
exit		Normal exit	000*x*01
exit	*exp*	Normal exit	000*xjk*

A.3 PASS INSTRUCTIONS

Result	Operand	Description	Machine Instruction
pass		Pass	076*xxx*
pass	*exp*	Pass	076*ijk*

A.4 SEMAPHORE INSTRUCTIONS

Result	Operand	Description	Machine Instruction
ssm		Set Semaphore	006*xxx*
csm		Clear Semaphore	007*xxx*

A.5 REGISTER ENTRY INSTRUCTIONS

A.5.1 ENTRIES INTO A REGISTERS

Result	Operand	Description	Machine Instruction
a_i	*exp*	Load a_i with a value	026*ijk* or 027*ijk* or 040*ijk* or 041*ijk* or 042*ijk*†††
a_i	*exp*,s	Load a_i with a 6-bit value	026*ijk*† or 027*ijk*†
a_i	*exp*,s,p	Load a_i with a 6-bit positive value	026*ijk*††
a_i	*exp*,s,m	Load a_i with a 6-bit negative value	027*ijk*††
a_i	*exp*,p	Load a_i with a 16-bit value	040*ixx*† or 041*ixx*†
a_i	*exp*,p,p	Load a_i with a 16-bit positive value	040*ixx*††
a_i	*exp*,p,m	Load a_i with a 16-bit negative value	041*ixx*††
a_i	*exp*	Load a_i with a value	042*ixx* or
a_i	*exp*,h	Load a_i with a 32-bit value	042*ixx*† or

† Forces one of two opcodes
†† Forces a single opcode
††† Forces one of five opcodes

A.5.2 ENTRIES INTO S REGISTERS

Result	Operand	Description	Machine Instruction
s_i	*exp*	Load s_i with a value	050*ixx* or 051*ixx* or 052*ixx* or 053*ixx* or 116*ijk* or 117*ijk*†††
s_i	*exp*,s	Load s_i with a 6-bit value	116*ijk*† or 117*ijk*†
s_i	*exp*,s,p	Load s_i with a 6-bit positive value	116*ijk*††
s_i	*exp*,s,m	Load s_i with a 6-bit negative value	117*ijk*††
s_i	*exp*,h	Load s_i with a 32-bit value	050*ixx*† 051*ixx*†
s_i	*exp*,h,p	Load s_i with a 32-bit positive value	050*ixx*††
s_i	*exp*,h,m	Load s_i with a 32-bit negative value	051*ixx*††
s_i	*exp*,l	Load s_i left side with a 32-bit value	052*ixx*†
s_i	*exp*,f	Load s_i with a 64-bit value	053*ixx*††

† Forces one of two opcodes
†† Forces a single opcode
††† Forces one of six opcodes

A.6 INTER-REGISTER TRANSFER INSTRUCTIONS

Instructions in this group provide for transferring the contents of one register to another register. In some cases, the register contents can be complemented, converted to floating-point format, or sign extended as a function of the transfer.

A.6.1 TRANSFERS TO A REGISTERS

Result	Operand	Description	Machine Instruction
a_i	s_j	Copy (s_j) to a_i	024*ijx*
a_i	vl	Copy (vl) to a_i	025*ixx*

A.6.2 TRANSFERS TO S REGISTERS

Result	Operand	Description	Machine Instruction
s_i	s_j	Copy (s_j) to s_i ($j=k$)	103*ijj*
s_i	a_k	Copy (a_k) to s_i with no sign extension	130*ixk*
s_i	$+a_k$	Copy (a_k) to s_i with sign extension	131*ixk*
s_i	vm	Copy (vm) to s_i	114*ixx*
s_i	rt	Copy real-time count to s_i	115*ixx*

A.6.3 TRANSFERS TO V REGISTERS

Result	Operand	Description	Machine Instruction
v_i	v_j	Copy (v_j) to v_i (j=k)	145*ijj*

A.6.4 TRANSFER TO VECTOR MASK REGISTER

The following syntax and its special form transmit the contents of register S_j to the VM register. The VM register is zeroed if the *j* designator is 0; the special form accommodates this case.

This instruction may be used in conjunction with the vector merge instructions where an operation is performed depending on the contents of the VM register.

Result	Operand	Description	Machine Instruction
vm	s_j	Copy (s_j) to vm	034*xjx*

A.6.5 TRANSFER TO VECTOR LENGTH REGISTER

The following syntax and its special form enters the low-order 7 bits of the contents of register A_k into the VL register.

The contents of the VL register determines the number of operations performed by a vector instruction. Since a Vector register has 64 elements, from 1 to 64 operations can be performed. The number of operations is (VL) modulo 64. A special case exists such that when (VL) modulo 64 is 0, then the number of operations performed is 64.

In this publication, a reference to register V_i implies operations involving the first *n* elements where *n* is the vector length unless a single element is explicitly noted as in the instructions S_i V_j, A_k and V_i, A_k S_j.

Result	Operand	Description	Machine Instruction
vl	a_k	Copy (a_k) to vl	036*xxk*

Vector operations controlled by the contents of VL begin with element 0 of the Vector registers.

A.7 MEMORY TRANSFER INSTRUCTIONS

This category includes instructions that transfer data between registers and memory.

A.7.1 STORES

Several instructions store data from registers into memory.

Local Memory writes

Result	Operand	Description	Machine Instruction
[*exp*]	a_k	Write (a_k) to location *exp* in Local Memory	045*xxk*
[a_k]	a_j	Write (a_j) to location a_k in Local Memory	047*xjk*
[*exp*]	s_j	Write (s_j) to location *exp* in Local Memory	055*xjx*
[a_k]	s_i	Write (s_i) to location a_k in Local Memory	057*ixk*
[a_k]	v_i	Write (v_i) to Local Memory location (a_k)	075*ixk*

Common Memory writes

Result	Operand	Description	Machine Instruction
(*exp*)	s_i	Write (s_i) to Common Memory at location *exp*	067*ixx*
(a_k)	s_i	Write (s_i) to Common Memory at location (a_k)	063*ixk*
(a_k,*exp*)	s_i	Write (s_i) to Common Memory at location (a_k)+*exp*	065*ixk*
(a_j,a_k)	s_i	Write (s_i) to Common Memory at location (a_j)+(a_k)	061*ijk*
(a_j,a_k)	v_i	Write (v_i) to Common Memory location (a_j) incremented by (a_k)	071*ijk*
(a_k,v_j)	v_i	Scatter (v_i) to Common Memory locations (a_k)+(v_j)	073*ijk*

A.7.2 LOADS

Several instructions can be used to load data from memory into registers.

Local Memory reads

Result	Operand	Description	Machine Instruction
a_i	[*exp*]	Read from location *exp* in Local Memory to a_i	044*ixx*
a_i	[a_k]	Read from location to a_k in Local Memory to a_i	046*ixk*
s_i	[*exp*]	Read from location *exp* in Local Memory to s_i	054*ixx*
s_i	[a_k]	Read from location to a_k in Local Memory to s_i	056*ixk*
v_i	[a_k]	Read from Local Memory location (a_k) to v_i	074*ixk*

Common Memory reads

Result	Operand	Description	Machine Instruction
s_i	(*exp*)	Read from Common Memory location *exp* to s_i	066*ixx*
s_i	(a_k)	Read from Common Memory at location (a_k) to s_i	062*ixk*
s_i	(a_k,*exp*)	Read from Common Memory at location (a_k)+*exp* to s_i	064*ixk*
s_i	(a_j,a_k)	Read from Common Memory location (a_j)+(a_k) to s_i	060*ijk*
v_i	(a_j,a_k)	Read from Common Memory location (a_j) incremented by a_k	070*ijk*
v_i	(a_k,v_j)	Gather from Common Memory locations (a_k)+(v_j) to v_i	072*ijk*

Memory Range Error flags

Result	Operand	Description	Machine Instruction
dri		Disable halt on memory field range error	035*xx*0
eri		Enable halt on memory field range error	035*xx*1

A.8 INTEGER ARITHMETIC OPERATION INSTRUCTIONS

Integer arithmetic operations obtain operands from registers and return results to registers. No direct memory references are allowed.

A.8.1 INTEGER SUMS

Result	Operand	Description	Machine Instruction
a_i	a_j+a_k	Integer sum of (a_j) and (a_k) to a_i	020*ijk*
s_i	s_j+s_k	Integer sum of (s_j) and (s_k) to s_i	104*ijk*
v_i	s_j+v_k	Integer sums of (s_j) and (v_k) to v_i	160*ijk*
v_i	v_j+v_k	Integer sums of (v_j) and (v_k) to v_i	161*ijk*

A.8.2 INTEGER DIFFERENCES

Result	Operand	Description	Machine Instruction
a_i	a_j-a_k	Integer difference of (a_j) and (a_k) to a_i	021*ijk*
s_i	s_j-s_k	Integer difference of (s_j) and (s_k) to s_i	105*ijk*
v_i	s_j-v_k	Integer differences of (s_j) and (v_k) to v_i	162*ijk*
v_i	v_j-v_k	Integer differences of (v_j) and (v_k) to v_i	163*ijk*

A.8.3 INTEGER PRODUCTS

Result	Operand	Description	Machine Instruction
a_i	a_j*a_k	Integer product of (a_j) and (a_k) to a_i	022*ijk*

A.9 FLOATING-POINT ARITHMETIC INSTRUCTIONS

All floating-point arithmetic operations use registers as the source of operands and return results to registers.

A.9.1 FLOATING-POINT SUMS

Result	Operand	Description	Machine Instruction
s_i	s_j+fs_k	Floating-point sum of (s_j) and (s_k) to s_i	120*ijk*
v_i	s_j+fv_k	Floating-point sums of (s_j) and (v_k) to v_i	170*ijk*
v_i	v_j+fv_k	Floating-point sums of (v_j) and (v_k) to v_i	171*ijk*

A.9.2 RECIPROCAL ITERATIONS

Result	Operand	Description	Machine Instruction
s_i	s_j*is$_k$	Reciprocal iteration step, 2-(s_j)*(s_k) to s_i	126*ijk*
v_i	v_j*iv$_k$	Reciprocal iteration step, 2-(v_j)*(v_k) to s_i	156*ijk*

A.9.3 RECIPROCAL APPROXIMATIONS

Result	Operand	Description	Machine Instruction
s_i	/hs$_j$	Floating-point reciprocal approximation of (s_j) to s_i	132*ijx*
v_i	/hv$_j$	Floating-point reciprocal approximation of (v_k) to v_i	166*ixk*

A.9.4 FLOATING-POINT DIFFERENCES

Result	Operand	Description	Machine Instruction
s_i	s_j-fs$_k$	Floating-point difference of (s_j) and (s_k) to s_i	121*ijk*
v_i	s_j-fv$_k$	Floating-point difference of (s_j) and (v_k) to v_i	172*ijk*
v_i	v_j-fv$_k$	Floating-point difference of (v_j) and (v_k) to v_i	173*ijk*

A.9.5 INTEGER TO FLOATING-POINT CONVERSIONS

Result	Operand	Description	Machine Instruction
s_i	fix,s_k	Convert (s_k) from floating-point to integer and enter into s_i	122*ixk*
v_i	fix,v_k	Integer form of floating-point (v_k) to v_i	174*ixk*

A.9.6 FLOATING-POINT TO INTEGER CONVERSIONS

Result	Operand	Description	Machine Instruction
s_i	flt,s_k	Convert (s_k) from integer to floating-point and enter into s_i	123*ixk*
v_i	flt,v_k	Floating-point form of integer (v_k) to v_i	175*ixk*

A.9.7 FLOATING-POINT PRODUCTS

Result	Operand	Description	Machine Instruction
s_i	s_j*fs_k	Floating-point product of (s_j) and (s_k) to s_i	124*ijk*
v_i	s_j*fv_k	Floating-point products of (s_j) and (v_k) to v_i	154*ijk*
v_i	v_j*fv_k	Floating-point products of (v_j) and (v_k) to v_i	155*ijk*

A.9.8 SQUARE ROOT ITERATIONS

Result	Operand	Description	Machine Instruction
s_i	s_j*qs_k	Square root iteration of [3-(s_j)*(s_k)]/2 to s_i	127*ijk*
v_i	v_j*qv_k	Square root iteration of [3-(v_j)*(v_k)]/2 to v_i	157*ijk*

A.9.9 SQUARE ROOT APPROXIMATIONS

Result	Operand	Description	Machine Instruction
s_i	*qs_j	Square root approximation of (s_j) to s_i	133*ijx*
v_i	*qv_k	Square root approximation of (v_k) to v_i	167*ixk*

A.9.10 FLOATING-POINT ERRORS

Result	Operand	Description	Machine Instruction
dfi		Disable halt on floating-point error	035*xx*2
efi		Enable halt on floating-point error	035*xx*3

A.10 LOGICAL OPERATION INSTRUCTIONS

A.10.1 LOGICAL PRODUCTS

Result	Operand	Description	Machine Instruction
s_i	s_j&s_k	Logical product of (s_j) and (s_k) to s_i	100*ijk*
s_i	#s_k&s_j	Logical product of (s_j) and complement of (s_k) to s_i	101*ijk*
v_i	s_j&v_k	Logical product of (s_j) and (v_k) to v_i	140*ijk*
v_i	v_j&v_k	Logical product of (v_j) and (v_k) to v_i	41*ijk*

A.10.2 LOGICAL SUMS

Result	Operand	Description	Machine Instruction
s_i	s_j!s_k	Logical sum of (s_j) and (s_k) to s_i	103*ijk*
v_i	s_j!v_k	Logical sums of (s_j) and (v_k) to v_i	144*ijk*
v_i	v_j!v_k	Logical sums of (v_j) and (v_k) to v_i	145*ijk*

A.10.3 VECTOR STREAMING

Result	Operand	Description	Machine Instruction
v_i	s_j!v_k&vm	Transmit (s_j) if vm bit=1; (v_k) if vm bit=0 to v_i	146*ijk*
v_i	v_j!v_k&vm	Transmit (v_j) if vm bit=1; (v_k) if vm bit=0 to v_i	147*ijk*

A.10.4 LOGICAL DIFFERENCES

Result	Operand	Description	Machine Instruction
s_i	s_j\\s_k	Logical difference of (s_j) and (s_k) to s_i	102*ijk*
v_i	s_j\\v_k	Logical difference of (s_j) and (v_k) to v_i	142*ijk*
v_i	v_j\\v_k	Logical difference of (v_j) and (v_k) to v_i	143*ijk*

A.10.5 VECTOR MASK

Result	Operand	Description	Machine Instruction
vm	v_k,z	Set vm from zero elements of (v_k)	030*xxk*
vm	v_k,n	Set vm from nonzero elements of (v_k)	031*xxk*
vm	v_k,p	Set vm from positive elements of (v_k)	032*xxk*
vm	v_k,m	Set vm from negative elements of (v_k)	033*xxk*

A.10.6 COMPRESSED IOTA

Result	Operand	Description	Machine Instruction
v_i	ci,s_j&s_k	Enter v_i with compressed iota (s_j) and (s_k)	176*ijk*

A.11 BIT COUNT INSTRUCTIONS

Result	Operand	Description	Machine Instruction
s_i	ps_j	Population count of (s_j) to s_i	106*ij*0
v_i	pv_j	Population count of (v_j) to v_i	164*ij*0
s_i	qs_j	Population count of parity of (s_j) to s_i	106*ij*1
v_i	qv_j	Population count of parity of (v_j) to v_i	164*ij*1
s_i	zs_j	Leading zero count of (s_j) to s_i	107*ijx*
v_i	zv_j	Leading zero count of (v_j) to v_i	165*ijx*

A.12 SHIFT INSTRUCTIONS

A.12.1 LEFT SHIFTS

Result	Operand	Description	Machine Instruction
s_i	s_i<*exp*	Shift (s_j) left *exp*=64-*jk* places to s_i	110*ijk*
v_i	v_j<a_k	Shift (v_j) left (a_k) bits with zero fill. Results to v_i	150*ijk*
s_i	s_i,s_j<a_k	Shift (s_i and s_j) left a_k places to s_i	112*ijk*
v_i	v_j,v_j<a_k	Double shift (v_j) left a_k places to v_i	152*ijk*

A.12.2 RIGHT SHIFTS

Result	Operand	Description	Machine Instruction
s_i	s_i>*exp*	Shift (s_i) right *exp*=*jk* places to s_i	111*ijk*
v_i	v_j>a_k	Shift (v_j) right (a_k) bits with zero fill. Results to v_i	151*ijk*
s_i	s_j,s_i>a_k	Shift (s_j and s_i) right a_k places to s_i	113*ijk*
v_i	v_j,v_j>a_k	Double shift (v_j) right a_k places to v_i	153*ijk*

READER COMMENT FORM

CRAY-2 Computer System Functional Description HR-2000

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME ____________________

JOB TITLE ____________________

FIRM ____________________

ADDRESS ____________________

CITY ____________ STATE ________ ZIP ________

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

FOLD

STAPLE

READER COMMENT FORM

CRAY-2 Computer System Functional Description HR-2000

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME ____________________

JOB TITLE ____________________

FIRM ____________________

ADDRESS ____________________

CITY __________ STATE ______ ZIP ______

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

FOLD

STAPLE