System Programmer Reference
(Cray SV1™ Series)

108-0245-003

Cray Proprietary

Direct requests for copies of publications to:

Mail: Cray Inc. Logistics
PO Box 6000
Chippewa Falls, WI 54729-0080
USA

E-mail: spares@cray.com
Web: http://techinfo.cray.com/pubscat/data/pubscat.htm
Select the **Order Form** button that is located at the bottom of the Web page.
Fax: +1 715 726 4602

Direct comments about this publication to:

Mail: Cray Inc.
Technical Training and Documentation
P.O. Box 6000
Chippewa Falls, WI 54729-0080
USA

E-mail: fiona@cray.com
Fax: +1 715-726-4991

System Programmer Reference

108-0245-003
Cray SV1 Series
Last Modified: February 2002

Figures

Tables

Record of Revision

October 1999

Original printing.

June 2001

Added 5 CPU instructions and additional information for the Cray SV1ex series machines.

February 2002

Clarified the description of SV1-1, SV1-1A, SV1ex-1, and SV1ex-1A memory, corrected block diagrams, clarified the bit matrix multiply (BMM) register, added Appendix A containing a description of the block transfer engine (BLT) and translate look-aside buffer (TLB), and made several corrections to the text.

Cray SV1 Series System Overview

Cray SV1 series systems (hereinafter referred to as SV1) are available in four models: the Cray SV1-1A and SV1ex-1A, and the Cray SV1-1 and SV1ex-1. Each model includes a mainframe cabinet and a minimum of one scalable input/output (SIO) peripheral cabinet (PC-10). The mainframe cabinet houses the processor and memory modules along with the system clock. The processor modules contain the central processing unit (CPU) components, and the memory modules contain the memory components; both reside in a backplane or midplane card cage.

The SV1-1A or SV1ex-1A mainframe cabinet contains four memory modules and from two to four processor modules. The SV1-1 or SV1ex-1 mainframe cabinet includes eight memory modules and from two to eight processor modules. A memory module contains dynamic random-access memory (DRAM) chips on the SV1-1A or SV1-1 or static dynamic random-access memory (SDRAM) chips on the SV1ex-1A or SV1ex-1. Each processor module can support one GigaRing channel and up to four CPUs. Refer to the Cray SV1 series *System Overview*, publication number 108-0196, for more information on system configurations, including memory options and sizes.

Note: SV1 series publications are accessible on the *techinfo* web site: `http://techinfo.cray.com/`

The SV1 processor and memory modules use application-specific integrated circuit (ASIC) technology. Each module is composed of an array of ASICs, which are based on very large-scale integration (VLSI) complementary metal oxide semiconductor (CMOS) technology.

The SV1 is a redesign of the Cray J90 series systems with improved processor performance and enhanced cache capability at the processor level with system-level scalability up to 1,024 processors. The SV1 processor module includes a faster processor clock, improved vector performance with a dual-pipe vector processor, and a larger 256-Kbyte cache per processor. This cache is common for all scalar and vector data and all instruction fetch request data per processor. Each processor module includes four CPUs, each with its own cache. The processor module CPUs share a common interface to memory.

Each processor module includes an optional I/O connection. The SIO architecture provides scalable high-performance and high-resilience I/O support for SV1 systems. SIO data and control information transmits over GigaRing channels. The flexibility of the GigaRing channel architecture enables multiple system configurations with system functionality and performance that are appropriate to the needs of the customer.

Cray J90 series systems can be upgraded to Cray SV1 systems and Cray SV1 systems can be upgraded to Cray SV1ex systems.

Cray SV1e Processor

The Cray SV1e processor is an upgrade to SV1 processor modules.

The new processor design includes a newer and faster ASIC that combines the functions of the PV and CA ASICs that are used on the Cray SV1-1 and Cray SV1-1A in a single PVC ASIC. By combining functions in one ASIC, CPU-to-cache bandwidth is increased and CPU-to-cache latency is decreased. The processor upgrade also includes a faster (500-MHz) system clock.

Cray SV1ex-1 and Cray SV1ex-1A Systems

The Cray SV1ex-1 and SV1ex-1A systems include the processor module upgrade described above as well as an upgrade to memory modules.

The Cray SV1ex-1 and Cray SV1ex-1A systems utilize synchronous dynamic random-access memory (SDRAM) chips. The SDRAM chips have a 37.5-ns access time. The Cray SV1ex-1 and Cray SV1ex-1A memory also includes onboard secondary storage device (SSD) memory. SSD memory allows high-speed transfer of data between main memory and SSD memory.

Note: For more information on SV1 and SV1ex memory, refer to the SV1 Series Processor and Memory Components manual.

Mainframe Overview

The SV1 mainframe contains processor modules, an interprocessor communication section, a real-time clock, and central memory. Each CPU has a computation section that consists of operating registers, functional units, and a control section. The control section determines instruction issue and coordinates the three types of processing (vector, scalar, and address). The I/O section, interprocessor communication section, real-time clock, and central memory are shared by the CPUs and are called *shared resources*.

Figure 1 is a block diagram of the SV1 series mainframe. It shows the internal organization of the CPU, with paths to central memory and I/O, and registers that are distributed among all CPUs within a cluster.

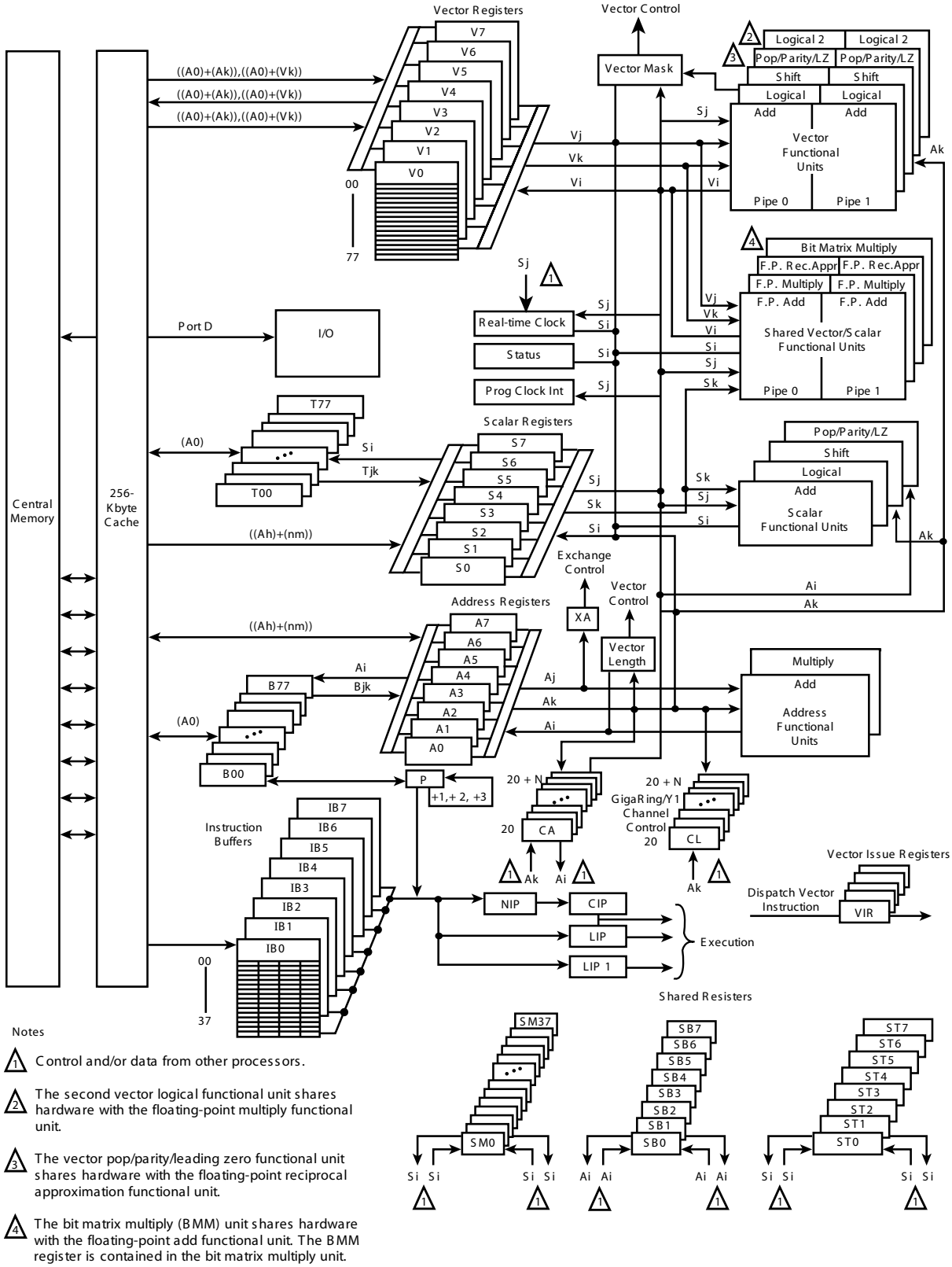
Central memory, which holds program code and data, is shared among all CPUs in the mainframe. It is available in various sizes and configurations. The I/O section provides high-speed data transfers to and from the I/O subsystem (IOS). The interprocessor communication section enables each CPU to synchronize operation and transfer data to and from other CPUs.

The CPU architecture, with the cache, enables efficient computation and memory access for both vector and scalar operations. Separate registers and functional units, with the exception of the floating-point units, exist for vector and scalar operations. All scalar floating-point operations execute in the vector floating-point units without delay (because the unit is busy). Vector processing uses a single instruction to perform a repeated operation on sets of ordered data. With the dual-pipe vector design, two operand results are produced per cycle per instruction. Scalar processing uses one instruction to perform one operation and produce one result.

Sequential vector instructions cause sequential portions of each operation to occur simultaneously. That, along with the dual-pipe design, generally causes the computational rate for vector processing to greatly exceed that of scalar processing. Scalar operations complement vector capability by providing solutions to problems that are not readily adaptable to vector techniques. Because the start-up time for vector operations is short, vector processing is more efficient than scalar processing for vectors that contain as few as two elements.

Multiple-processor systems enable multiprocessing and multitasking techniques. Multiprocessing allows several programs to run concurrently on multiple CPUs within the mainframe. Multitasking allows two or more parts of a program to run in parallel in separate CPUs and to share a common memory space.

Figure 1. Cray SV1 Mainframe Block Diagram



Notes

1. Control and/or data from other processors.
2. The second vector logical functional unit shares hardware with the floating-point multiply functional unit.
3. The vector pop/parity/leading zero functional unit shares hardware with the floating-point reciprocal approximation functional unit.
4. The bit matrix multiply (BMM) unit shares hardware with the floating-point add functional unit. The BMM register is contained in the bit matrix multiply unit.

SIO Overview

The SV1 system supports the scalable I/O subsystem. SIO is a single-cabinet or multicabinet subsystem that provides high-performance, high-resilience I/O support; it is a collection of I/O nodes in which each node is an independent unit that connects to a GigaRing channel.

GigaRing Overview

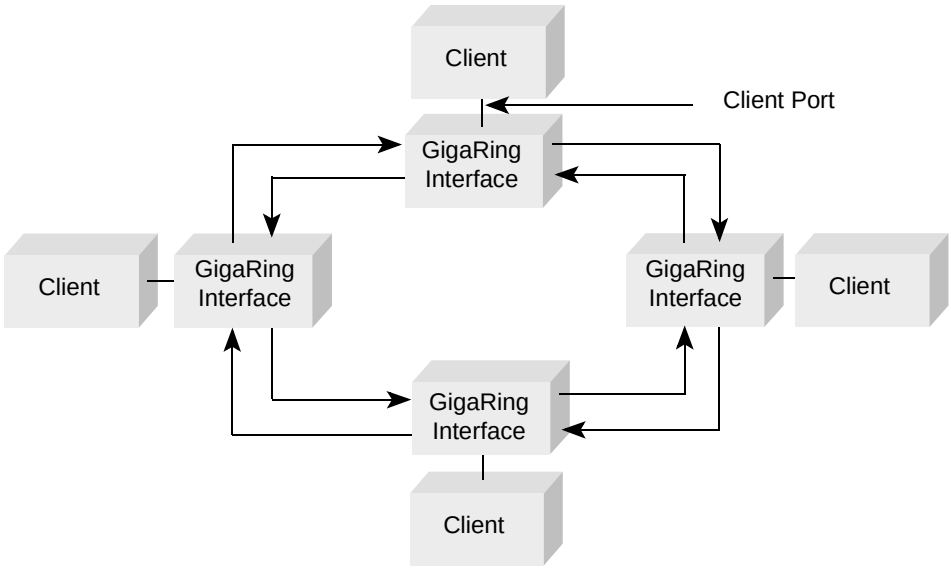
The GigaRing channel allows for high-speed communication among the current mainframes and peripherals, as well as direct interconnections between all Cray products.

The GigaRing channel incorporates a pair of unidirectional, counter-rotating rings to support multiple nodes. Each of the two rings has a maximum transfer rate of 500 Mbytes/s, which provides an effective total bandwidth of 800 Mbytes/s. The redundancy (two rings) and counter-rotation enable the GigaRing channel to operate during a link or node failure at a reduced data rate; the rings can be folded to map out faulty nodes or channel connections. The counter-rotating rings also enable shortest-path communication.

Each Cray SV1 processor module can be configured with one GigaRing channel adapter. Each GigaRing adapter is a single node with a full duplex bandwidth of 400 Mbytes/s (200 Mbytes/s input and 200 Mbytes/s Output). The I/O bandwidth requirements of the system determine the quantity of channel adapters needed in the system. Typically there will be processor modules in the system that are configured without channel adapters.

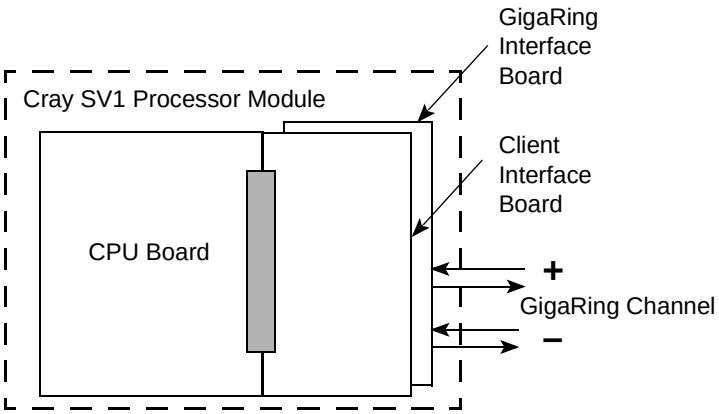
A GigaRing channel consists of two or more GigaRing node chips that are connected and that use GigaRing protocol. Based on the Scalable Coherent Interface (SCI) standards, GigaRing protocol supports direct memory access, peer-to-peer messaging, and remote memory data transfers. I/O data and control information messages pass among mainframes and nodes via the GigaRing channel. Figure 2 shows two possible GigaRing channel configurations.

Figure 2. Cray SV1 Four-node GigaRing Channel Configuration



The GigaRing node chip implements the logical layer of the GigaRing channel and supports the I/O protocol. A GigaRing node chip contains a client port interface, incoming and outgoing positive links, and incoming and outgoing negative links. Figure 3 shows an SV1 I/O node. The node chips use a packet-based protocol and balance the communication loads of the devices automatically.

Figure 3. Cray SV1 I/O Node



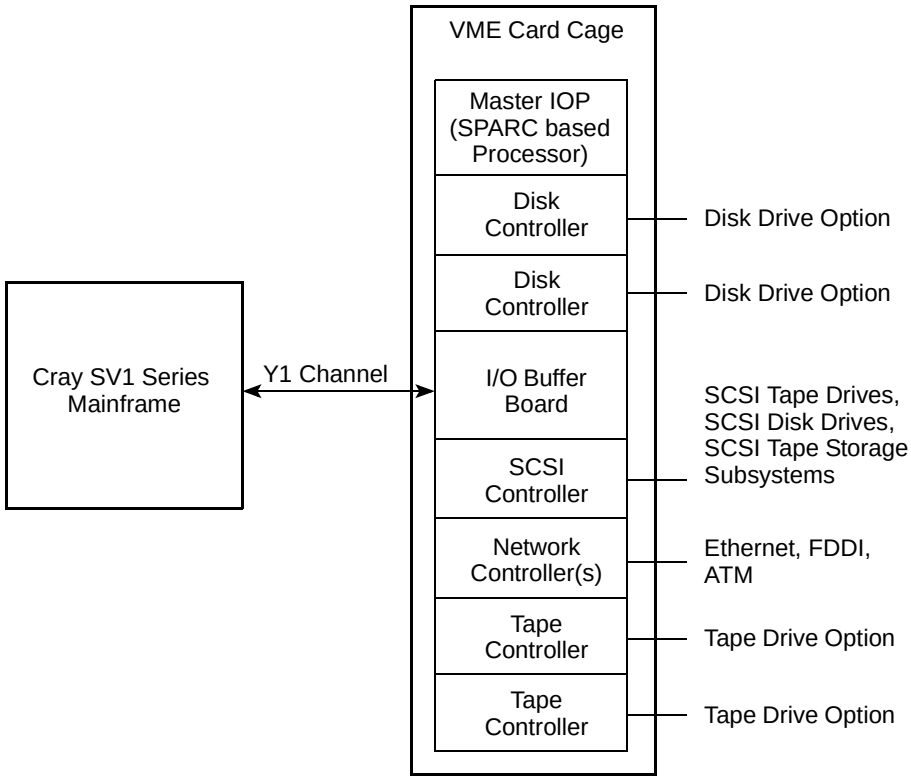
Note: Each processor module contains a new channel adapter board that provides one GigaRing node chip. This new channel adapter (client interface) and the GigaRing interface board provide the interface to the GigaRing node.

Communication between nodes occurs when the source node sends packets of information to a target node. When a client transmits a packet, the packet is placed in the send buffer of its local interface.This client and its local interface become the source node. The source node then transmits the packet around the ring until the packet reaches its target node. Each transfer is protected by a cyclic redundancy checksum (CRC).

VME-based I/O Subsystem Overview

SV1 systems may include an IOS VME 64-bit bus architecture for data transfers from central memory to peripherals and networks. The VME 64-bit bus is a high-performance industry standard backplane that can connect vendor-compatible I/O controllers to the IOS. Refer to Figure 4 for a block diagram of the IOS and examples of the peripheral devices that may be included in an SV1 system.

Figure 4. IOS Block Diagram



Data travels from a peripheral device across a data channel to the device controller, and then from the device controller to the input/output buffer board (IOBB) across the VMEbus. From the I/O buffer board, data travels to mainframe memory through the Y1 50-Mbyte/s data channel. There are four Y1 channels for each processor module.

The IOS input/output processor (IOP) is the CPU for the IOS. The IOP performs I/O functions for the I/O controllers, processes external interrupts and CPU I/O requests, and executes peripheral driver routines.

The VME controller boards enable the IOS to support the following operations and devices:

- System console operation
- Disk subsystems
- Tape subsystem
- Network subsystem

An SV1 system can contain up to 16 IOSs, and each processor module can handle up to 4 IOSs each. Each of the four possible peripheral cabinets may contain from 1 to 4 IOSs.

Each IOS can support either two or four I/O controllers, plus two required boards: the IOP and IOBB. The number of controllers that are supported depends on the type of VME backplane.

Note: A channel adapter board is connected to the processor module to provide for the IOS interface. A different channel adapter board is required for the GigaRing channel.

Network Interfaces

An SV1 series system is designed to communicate easily with front-end computer systems and computer networks and can function as a stand-alone system or can be networked into an existing computing environment. The system can be connected to a multiple-system network with an Ethernet connection or a fiber-distributed data interface (FDDI) local area network using Transmission Control Protocol/Internet Protocol (TCP/IP). SV1 systems also support asynchronous transfer mode (ATM) protocol UNICOS 10.0.0.4 (SWS-ION 4.0).

Maintenance Platform

The SV1 systems maintenance platform includes a Cisco router. The system console is a Sun Microsystems SPARCstation 5 Workstation or an Ultra 5 Workstation.

Central Memory

The Cray SV1-1 and Cray SV1-1A machines include central memory that consists of solid-state, dynamic random-access memory (DRAM) that is shared by all the CPUs and the I/O section in a mainframe. Each memory word consists of 72 bits: 64 data bits and 8 error-correction bits. These 8 bits perform single-error correction/double-error detection (SECDED). DRAM chips provide storage for data and correction bits. The DRAM chips have a 50-ns access time. In order to improve memory access speed, central memory has multiple banks that can be active simultaneously. Each central memory bank can be accessed once every 14 system clock periods (140 ns).

The Cray SV1ex-1 and Cray SV1ex-1A systems utilize synchronous dynamic random-access memory (SDRAM) chips. The SDRAM chips have a 37.5-ns access time. Cray SV1ex memory also includes on-board SSD memory, which enables high-speed transfer of data between main memory and SSD memory.

Cray SV1 central memory consists of 8 sections. Each Cray SV1-1 memory section contains 8 subsections, and each Cray SV1-1A memory section contains 4 subsections. Cray SV1ex-1A and Cray SV1ex-1 memory contain 8 sections with 16 subsections per section.

Note: For more information on SV1 and SV1ex memory, refer to the *Processor and Memory Components* manual, publication number 108-0197.

In each SV1 CPU, the operating registers, instruction buffers, and exchange package have access to central memory through two memory ports, port A and port B. At the CPU interface these two ports are expanded to four physical request ports with the CPU cache for up to two write requests and two read requests per cycle. The CPU generates all fetch requests instead of using port D for these requests. All I/O operations with memory use the separate port D.

Memory Instructions

Table 1 shows the CPU memory instructions that transfer data between CPU registers and central memory or cache. The contents of the database address (DBA) register are added to instruction-generated memory addresses to form absolute memory addresses.

Table 1. CPU Memory Instructions

Machine Instruction	CAL Syntax	Description	Types of Memory References
10*hi*00*mn*	A*i exp*,A*h*	Load A*i* from address ((A*h*) + *exp* + (DBA)) *exp* = *nm*	Scalar
11*hi*00*mn*	*exp*,A*h* A*i*	Store (A*i*) to address ((A*h*) + *exp* + (DBA)) *exp* = *nm*	
12*hi*00*mn*	S*i exp*,A*h*	Load S*i* from address ((A*h*) + *exp* + (DBA)) *exp* = *nm*	
13*hi*00*mn*	*exp*,A*h* S*i*	Store (S*i*) to address ((A*h*) + *exp* + (DBA)) *exp* = *nm*	
034*ijk*	B*jk*,A*i*, ,A0	Load (A*i*) words from B*jk to* address (A0 + (DBA)) to B*j*k	Block
035*ijk*	,A0 B*jk*,A*i*	Store (A*i*) words to (A0 + (DBA)) from B*jk*	
036*ijk*	T*jk*,A*i* ,A0	Load (A*i*) words from address (A0 + (DBA)) to T*jk*	
037*ijk*	,A0 T*jk*,A*i*	Store (A*i*) words from T*jk* to address ((A0 + (DBA))	
176*i*0*k*	V*i* ,A0,A*k*	Load (VL) words from ((A0 + (DBA)) incremented by (A*k*) to V*i* elements	Stride
1770*jk*	,A0,A*k* V*j*	Store (VL) words from (V*j* elements) to address (A0 + (DBA)) incremented by (A*k*)	
176*i*1*k*	V*i* ,A0,V*k*	Load (VL) words from ((A0) + (DBA) + (V*k*)) to V*i* elements	Gather
1771*jk*	,A0,V*k*, V*j*	Store (VL) words from V*j* elements to address ((A0) + (V*k* elements) + (DBA))	Scatter
002300	ERI	Enable interrupt on operand range error	None
002400	DRI	Disable interrupt on operand range error	
002500	DBM	Disable bidirectional memory transfers	
002600	EBM	Enable bidirectional memory transfers	
002700	CMR	Complete memory references	
002703	ETSI	Enable test and set invalidate	
002704	CPA	Hold if port A or B busy	
002705	CPR	Hold if port A or B busy	
002706	CPW	Hold ports if block store or scalar load/store busy	
002707	DTSI	Disable test and set invalidate	

Note: Instructions 002703 through 002707 are for the SV1ex series machines only.

Instructions 10*hi*00 through 13*hi*00 perform scalar references. The 11hi00 and 13hi00 instructions transfer 1 word from a specified register to cache and to memory. The 10*hi*00 and 12*hi*00*mn* instructions in the SV1 system transfer 8 words from memory to cache, and 1 word is then transferred into the CPU-specified resister. Instructions 034ijk through 037ijk perform block transfers. Each instruction transfers a block of 1 or more words to or from consecutive memory locations. Instructions 176*i*0*k* and 1770*jk* perform stride references. From 1 to 64 words are transferred to or from memory locations that are separated by a constant increment (stride). Instructions 176*i*1*k* and 1771*jk* perform gather and scatter references. These instructions transfer 1 to 64 words to or from randomly programmable locations in memory.

Instructions 002300 through 00270k affect memory operation. Instructions 002300 and 002400 set and clear the interrupt-on-operand range (IOR) error bit in the exchange package (Figure 6) mode register. Instructions 002500 and 002600 clear and set the bidirectional memory (BDM) bit in the mode register. Instruction 002700 performs no operation, but it holds issue until all previously issued instructions complete all memory references. Refer to "Port Utilization" for an explanation of the BDM bit and the 002700 instruction.

Instructions 002703 through 002707 are added for the SV1s with enhanced processors and the SV1ex machine. Instruction 002703 sets the ECI (Enable Cache Invalidate) bit in the cache-enable field (bit 34 of word 7) of the exchange package, which enables any subsequent test-and-set instructions to invalidate cache. Instructions 002704 and 002705 perform a CPA (complete ports all). Instruction 002706 performs a CPW (complete port writes). Instruction 002707 clears bit 2 in the cache-enable field (bit 34 of word 7), which prevents any subsequent test-and-set instructions from invalidating cache.

In addition to direct memory references that are generated by CPU machine instructions, there are three ways that memory references are generated indirectly. First, a no-coincidence condition in a CPU causes an instruction fetch sequence to begin, which causes 32 consecutive words to be read from central memory into an instruction buffer. Second, an exchange sequence in a CPU causes 16 words to be read from central memory and 16 words to be written into central memory. The third indirect memory reference method occurs when an I/O transfer to or from an external device causes a block of words to be read from or written into central memory. Refer to the "VME I/O Section" or "GigaRing I/O Section" for details on I/O transfers.

Logical Organization

Figure 5 shows a CPU's memory ports and paths to cache and to the mainframe central memory. Refer to this figure while you read the following paragraphs. The cache is located between the CPU and the interface to memory. All CPU requests to memory, including fetch requests, pass through the CPU cache. Ports A and B are functional, but these ports are internal to the CPU. Four paths, independent of the ports, are used for requests to cache and from the cache to the memory interface.

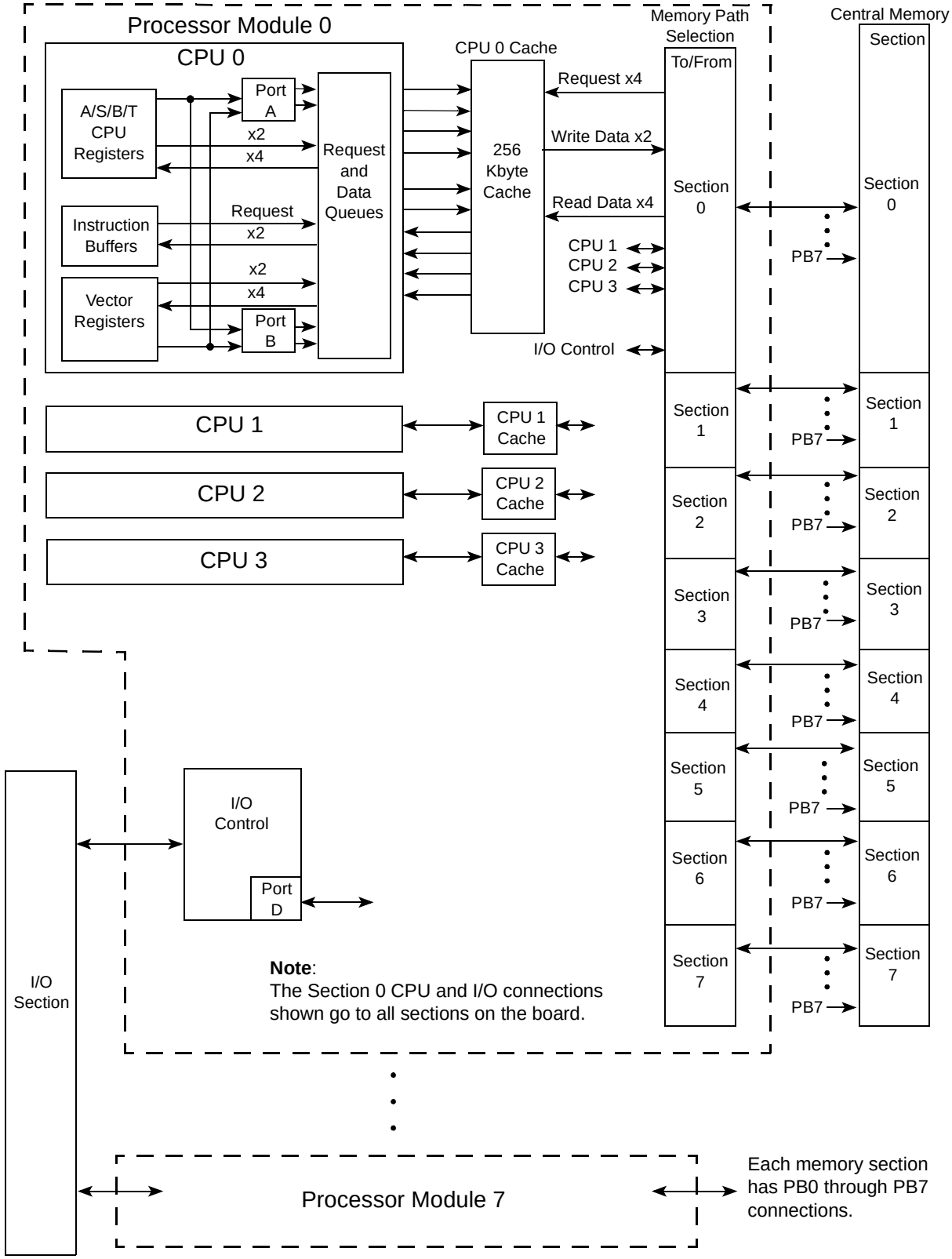
In an SV1-1 series system, central memory is divided into 8 sections; each contains 8 subsections. Each subsection contains 16 pseudobanks. Memory contains $1{,}024_{10}$ banks. In an SV1-1A series system, central memory is divided into 8 sections and each section contains 4 subsections. Each subsection contains 16 pseudobanks. Each central memory bank can be accessed once every 14 CPs.

In an SV1ex-1 series system, central memory is divided into 8 sections; each section contains 16 subsections. In an SV1ex-1A series system, central memory is divided into 8 sections; each section contains 8 subsections. Each subsection contains 8 memory banks.

Note: For more information on SV1 and SV1ex memory, refer to the *SV1 Series Processor and Memory Components* manual.

Each memory section processes the requests from all processor modules in the system. Each memory section buffers the requests as required by bank busy signals and requests activity from all CPUs in the system. A memory section guarantees order for the request from a single CPU but not between CPUs.

Figure 5. SV1 CPU Central Memory Architecture



Port Utilization

Each CPU has two ports. Ports A and B are both read and write ports that have only one write operation active at a time (refer to Table 2). This enables a read on port A and a write on port B or the opposite (a write on port A and a read on port B) if the BDM (bidirectional memory) mode bit is set in the exchange package. Both Ports A and B can generate two requests per clock period. Also, both ports can also be active with a read operation

Although the ports are listed as A, B, and D, only port D is an actual physical port. Ports A and B are used as identification codes because they do not physically exist external to the PV ASIC (SV1 series) or PVC ASIC (SV1ex series).

Table 2. Port Specifications

Port	Type of Reference	Port Usage
A	Read or Write Memory	A registers (10*h*, 11*h* instructions) S registers (12*h*, 13*h* instructions) B registers (034, 035 instructions) T registers (037 write instruction V registers (176, 177 instructions) Exchange (Read and Write)
B	Read or Write Memory	B registers (035 write instruction) T registers (036, 037 instructions) V registers (176, 177 instructions)
D	Read or Write Memory	I/O Write Memory and/or I/O Read Memory
	Read Memory	Fetch (really portless in the SV1)

Each processor module can generate a maximum of 17 read references (4 from each processor plus 1 from I/O) or a maximum of 9 write references (2 from each processor plus 1 from I/O) per clock period. These references must share 8 section paths to memory across the backplane. In other words, if a processor module generates 17 read references, a maximum of 8 references can be issued at a time (one per section). Simultaneous and overlapping memory references from different processor modules are permitted within a section on a memory module.

The exchange sequence uses port A only. Before an exchange can occur, all CPU and memory activity for that CPU must go inactive. The exchange package contains 16 words. The SV1 sends all new exchange package read requests to memory on port A followed by the 16 write requests for the old exchange package, also on port A. Fetch requests are sent to memory after the appropriate new exchange package data is available in the CPU. All SV1 CPU requests to memory must pass through that CPU's cache.

Refer to Figure 6 and to "Exchange Package" on page 70 for an illustration and description of the SV1 exchange package. Figure 6 also includes information on the SV1 read mode error reporting and processor type. Refer to Table 3 for cache register (CRPE) parity error CA ASIC identifiers.

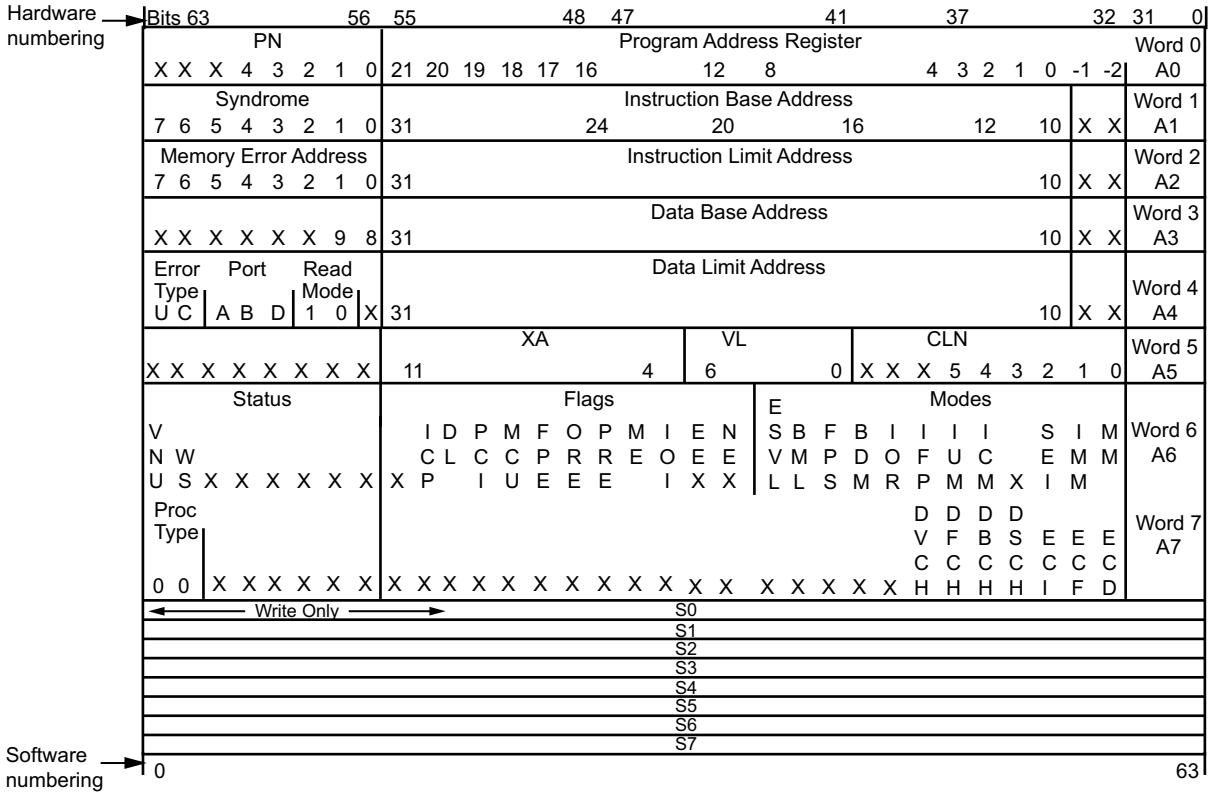
Table 3. CA ASIC Register Parity Error

Reported Section Number Error Address Bits (2:0)	CPU Number	CA ASIC
0, 2, 5, or 7	0	CA 0
1, 3, 4, or 6	0	CA 1
0, 2, 5, or 7	1	CA 2
1, 3, 4, or 6	1	CA 3
0, 2, 5, or 7	2	CA 4
1, 3, 4, or 6	2	CA 5
0, 2, 5, or 7	3	CA 6
1, 3, 4, or 6	3	CA 7

Memory error reporting occurs in the uppermost byte of exchange package words 0 through 4 after an uncorrectable or correctable error interrupt. A cache register parity error reports a syndrome code of all zeroes. The upper 2 bits of Word 7 define the processor type. The UNICOS operating system compensates for the difference among the Cray J90 and SV1 series processor modules. Some library routines have been altered to accommodate the difference in processor performance.

A fetch operation reads 32 words from memory and loads the data into one of eight instruction buffers. The fetch operation must complete as soon as possible to ensure that instructions continue to issue. The SV1 processor generates all 32 requests to cache and sends the requests through to memory. Read data is returned to the instruction buffer in mixed order. Cache supplies the fetch operands when they are available in cache or automatically requests them from memory if not in cache. Fetch requests are not assigned to any specific port.

Figure 6. Exchange Package



Processor Type

Bit 31	Bit 30	Type
0	0	Cray J90
0	1	Cray J90se
1	0	Cray SV1
1	1	Cray SV1e

Cray SV1 Read Mode Error Reporting

Bit 1	Bit 0	Port A	Port B	Port D Channel	Operation with Error
0	0	1	0	0	Exchange
0	0	0	1	0	Fetch
0	1	1	0	0	B
0	1	0	1	0	T
1	0	0	0	0	Vector
1	1	1	0	0	A or S
0	0	0	0	1	Channel n+1
0	1	0	0	1	Channel n+3
1	0	0	0	1	Channel n+5
1	1	0	0	1	Channel n+7

n = Processor board number X 10 + 20

I/O is controlled by the channel interface (CI) ASIC. When I/O requests on port D arrive at the memory interface ASICs on the processor module, they have the lowest priority when they conflict with the other processor ports for the same section of memory. A lockout counter keeps track of how often port D is denied access (per section). When a limit is reached, port D receives the highest priority for one cycle. A configuration file controls the lockout count value. This lockout count is set at the initial start-up of the system. A separate lockout counter exists for each section of memory.

Conflict Resolution

A memory conflict occurs whenever a memory port tries to access a part of memory that is in use, or whenever two or more ports try to access the same part of memory at the same time. Intra-CPU conflicts involve ports in the same CPU. Inter-CPU conflicts involve ports in different CPUs. The SV1 CPU intra-CPU conflicts cannot occur external to the CPU because no request leaving the CPU is associated with a port. Inter-CPU conflicts involve requests from different CPUs on the same processor module. Conflict resolution logic uses a predefined priority scheme to sequence the conflicting memory references and to maximize overall machine throughput.

There are four types of memory conflicts: section, bank busy, pseudobank busy, and subsection busy. The following paragraphs explain each type of conflict and how the conflict is resolved.

Each processor module contains four CPUs and an I/O channel. A section conflict can be caused by either an intra-CPU conflict or an inter-CPU conflict. An inter-CPU section conflict occurs when two or more CPUs or two or more I/O processors try to access the same section of memory. To resolve the inter-CPU section conflict, a four-slot priority is used in which access to the next higher-numbered section of memory receives one of four requesting processors during each cycle. The four slots have a rotating priority through the eight sections of memory with two-section spacing between the slots. Two sections are grouped together to accommodate the four slots. The two-section separation is required because requests are sent to memory in a read/write pair grouping that requires two cycles for transmission of write data to memory. In the absence of a write request, a second read request can be sent. The maximum request transmission rate across the backplane for a processor module is eight requests per cycle. This transmission rate can consist of eight reads or a combination of reads and writes, with a maximum of four write requests per cycle.

The four slots (each a two-section group) have a rotating priority through the eight sections of memory with two-section spacings between slots. Each slot has top priority to one section of memory during each cycle. This is the slot's natural priority. A natural slot priority that is not in use can be borrowed by another slot; the three non-natural slots use a priority scheme for borrowing. I/O is unslotted and uses any available slot. The user can configure I/O priority from lowest to highest priority, depending on system requirements. All read and write requests to memory per CPU share the same slot: CPU 0 requests share slot 0, CPU 1 requests share slot 1, CPU 2 requests share slot 2, and CPU 3 requests share slot 3. Table 4 shows this priority scheme.

Table 4. Memory Priority Scheme

	Slot Number Groups			
Priority	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
Natural slot	1 0 0 0	0 1 0 0	0 0 1 0	0 0 0 1
Borrowed first	0 0 1 0	0 0 0 1	1 0 0 0	0 1 0 0
Borrowed second	0 1 0 0	0 0 1 0	0 0 0 1	1 0 0 0
Borrowed third	0 0 0 1	1 0 0 0	0 1 0 0	0 0 1 0
Slot number	0	1	2	3

A processor module has an independent path into each memory section. Any conflicts between read and write requests to memory from a CPU are resolved before these requests leave that CPU. The general rule is that read requests have priority over write requests within a CPU. This priority is honored all the way through memory per CPU (per section of memory).

For requests from different CPUs on the same processor module as well as different processor modules, the order of access to a shared memory bank can be unrelated to the order in which these requests were sent to the section of memory.

The memory interface ASICs on the processor module process requests from six sources per section of memory: one request from each of the four CPUs and one I/O read request and one I/O write request. The logic within these ASICs determines which of the six requests has the highest priority for that particular cycle. The priority network rotates among the CPUs and is updated every other cycle. If the highest priority CPU has no pending request to be issued, the priority network checks the CPU that has the next highest priority for any pending requests. It continues traversing the priority tree until it finds a request to issue that cycle.

Fetch requests are given priority within that CPU and are not identified as fetch requests after leaving that CPU. Thus, they are not given higher priority at the processor module memory interface. I/O requests are generally given the lowest priority in this memory interface. To prevent an I/O port from being locked out during intense CPU memory request activity, an I/O lockout counter gives the I/O request highest priority until that one request is issued. This count is configurable so that systems with a high rate of I/O activity can obtain better memory bandwidth.

A CPU is given priority to two sections of memory during a 2-cycle rotating slot time. During this 2-cycle slot time, two read requests to the same section of memory, or a read and a write request, can be processed. A write request to memory is processed only during the second cycle of this slot time. Priority can be lent to another CPU to send a read request the first cycle while the original CPU sends the write request the second cycle.

A processor module sends a reference to one of eight sections of memory. This is true for all eight processor modules. Within a section, the reference is routed to one of eight subsections. In a subsection, the bank that is referenced is checked against the 16 banks within a subsection. The reference is held as long as the requesting bank remains busy. The memory arbiter (MAR) ASIC tracks two bank busy situations. One situation is called primary bank busy; this is the situation in which a bank that is referenced remains busy. The other bank busy situation is pseudobanking, in which a pair of banks shares an address and data bus. The time during which the pseudobank uses the bus is called the pseudobank busy time (8 CPs) and is considerably shorter than the primary bank busy time.

Memory conflicts are resolved on a subsection basis. In an SV1-1 series system, central memory is divided into 8 sections with each containing 8 subsections. Each subsection contains 16 pseudobanks. Memory contains $1{,}024_{10}$ banks. In an SV1-1A series system, central memory is divided into 8 sections, and each section contains 4 subsections. Each subsection contains 16 pseudobanks. Each central memory bank can be accessed once every 14 CPs. Eight paths from the MAR ASIC lead to the 8 memory subsections, one path to each subsection. When more than one reference attempts to use the same subsection within the same clock period, a subsection conflict occurs. A rotating priority scheme establishes priority across the processor modules. When all processor modules are active, each processor module receives access to a subsection once every four cycles.

Guaranteeing Memory Access Order

When the CPUs and memory must be synchronized, the complete memory reference (CMR) instruction must be issued. The CMR instruction holds issue until all references from that CPU are granted bank access. This is important in a multitasking environment in which a write operation must be complete before a write reservation is dropped. Other methods (for example, tracking port reservations or register reservations) ensure that the order of memory reservations for one CPU is correct, but they do not guarantee that the data reached memory.

Clearing the BDM bit in the exchange package mode register prevents out-of-sequence memory references. When the BDM bit is cleared, a memory read and write cannot occur simultaneously for that CPU. The memory read instruction holds issue until all write instruction requests are sent to the memory sections, and the memory write instruction holds issue until all read requests are sent to memory. The BDM bit has no effect on CPU operations.

A zero-length B- or T-register memory write instruction (pseudo-CMR) can be used to ensure the correct order of a write instruction and a subsequent read instruction issued by one CPU. When the zero-length B- or T-register memory write instruction is issued between the write and read instructions, subsequent reads of the same memory words do not occur ahead of the write. It does not guarantee that the write data is in memory.

A hardware limitation of the SV1 is that only one write to memory operation request can be active at a time.

Any active block transfer (B/T) or vector transfer prevents a scalar reference from issuing. The scalar reference issues when the internal requesting ports in the CPU are all quiet.

For vector references, the hardware guarantees memory ordering for a vector load followed by a vector store where both vector memory references have the same starting address and same stride.

For block transfers (B/T) a B read following a B write is **not** guaranteed to be a safe memory access if the B read is to memory addresses that the B write is writing to. Block T transfers of this type are also not guaranteed.

The hardware must ensure that all reads are complete before it grants write requests to memory with the same starting address and the same stride. A write operation before a read operation is not ensured.

Refer to Table 5 for the SV1 coding requirements for memory operations.

Table 5. SV1 Coding Requirements for Memory Operations

First Operation	Second Operation: vw	vr	sw	sr	bw	br	tw	tr	tas	fetch
vw	OK	ad	OK	OK	OK	a	OK	a	OK	f
vr	bde	OK	OK	OK	c	OK	c	OK	OK	OK
sw	OK	OK	OK	OK	OK	OK	OK	OK	OK	f
sr	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
bw	OK	a	OK	OK	OK	a	OK	a	OK	f
br	c	OK	OK	OK	OK	OK	c	OK	OK	OK
tw	OK	a	OK	OK	OK	a	OK	a	OK	f
tr	c	OK	OK	OK	c	OK	OK	OK	OK	OK
tas	OK	OK	OK	OK	OK	OK	OK	OK	OK	h
fetch	g	OK	g	OK	g	OK	g	OK	h	OK

v = vector, s = scalar, b = B register, t = T register, w = write, r = read, tas = test-and-set invalidate

All prior operations are guaranteed complete (with the exception of scalar prefetches) before an exchange can start.

OK = A safe sequence.

a = Must be protected by software by inserting one of the following instructions between them.
- any scalar reference
- any write reference
- pseudo-CMR (zero length B/T write)
- CMR

b = Must be protected by software by inserting one of the following instructions between them.
- any scalar reference
- 076 instruction using the V*i* of the vector read
- CMR

c = Must be protected by software by inserting one of the following instructions between them.
- any scalar reference
- CMR

d = The special case of gather/scatter or scatter/gather is protected by hardware. Gathers will hold issue until scatters have completed and vice versa.

e = The special case of same base, same stride is protected by hardware.

Table 5. SV1 Coding Requirements for Memory Operations (continued)

First Operation	Second Operation									
	vw	vr	sw	sr	bw	br	tw	tr	tas	fetch

f = There is no guarantee that an earlier write modification will reach memory before a later fetch to memory. This guarantee would have to be protected with one of the following instructions (which must be either in the same instruction buffer as the write modification or at least in an earlier buffer than the buffer that contains the modified data.)
- any scalar reference
- any write reference
- pseudo-CMR (zero length B/T write)
- CMR

g = There is no guarantee that a fetch will reach memory before a write reference that is inside the instruction buffer. Execution starts as soon as the word is returned from memory. If the first thing done is a write to a location being fetched, the write could reach memory before the fetch. A write to an address inside the current instruction buffer is not allowed.

h = There is no communication between a test-and-set invalidate and a fetch. There could be instructions in cache after a test-and-set invalidate if a fetch was in progress.

Calculating Absolute Memory Address

CPU memory reference instructions (listed in Table 1) calculate absolute memory addresses by adding combinations of the following values:

- A register contents
- V register contents
- DBA register contents
- 3-parcel instruction *nm* field contents

Each time an instruction makes a memory reference, the memory address that the instruction generates is added to the content of the DBA register to form the absolute memory address.

Only the following elements are used to calculate memory addresses: bits 0 through 31 of the A register, the V register, DBA register, and 3-parcel instruction *nm* field contents. Addressing memory where the upper address limit is greater than the amount of memory in the system will result in memory wraparound.

Address Range Checking

Four registers in the exchange package place a program's data and instruction areas in specific locations in memory and allocate specific amounts of memory to the areas. These registers allow all programs to be relocated. When a program is written, the programmer does not need to know the memory location of the instruction and data areas. These registers also enable the programmer to restrict certain parts of memory from any program. A program may halt if it tries to perform an instruction outside its allowed instruction area, or if it tries to read or write data outside its allowed data area. When more than one program occupies memory at the same time, programs may not be able to perform instructions or operate on data that belongs to other programs.

The DBA register determines where in memory a program's data area begins. Addresses generated by memory reference instructions are relative to the DBA register.

Each time an instruction makes a memory reference, the memory address that the instruction generates is added to the contents of the DBA register to form the absolute memory address. Refer again to Table 1 for a list of memory reference instructions.

The data limit address (DLA) register determines the highest absolute memory address that the program can use for reading or writing data. Each time an instruction makes a memory reference, the absolute memory address that the instruction generates is compared to the contents of the DLA and DBA registers. If the absolute memory address is less than the DLA register contents and equal to or greater than the DBA register contents, the reference proceeds. If the absolute memory address is equal to or greater than the DLA register contents or less than the DBA register contents, an out-of-range condition exists and the memory reference is aborted for a write to memory. For a read from memory, the read occurs but the data is forced to all zeroes in the CPU.

If the interrupt-on-operand range error (IOR) bit in the mode register of the exchange package is set, the out-of-range condition sets the operand range error (ORE) flag in the exchange package flag register and causes an exchange sequence to begin. If the IOR bit is clear, the program continues to run.

The instruction base address (IBA) register functions similarly to the DBA register, except that it operates on a program's instruction area. Each time an instruction fetch sequence takes place, absolute memory addresses are formed by adding the relative addresses that are generated by the fetch control logic to the contents of the IBA register.

The instruction limit address (ILA) register functions similarly to the DLA register, except that it operates on a program's instruction area and has no provision for continuing program execution when an out-of-range condition occurs. If an absolute memory address generated by an instruction fetch sequence is less than the ILA register contents and equal to or greater than the IBA register contents, the fetch sequence proceeds. If the absolute memory address is equal to or greater than the ILA register contents or less than the IBA register contents, an out-of-range condition exists. An out-of-range condition sets the program range error flag in the exchange package and causes an exchange sequence to begin.

The DBA, DLA, IBA, and ILA registers contain only address bits 10 and above. Bits 0 through 9 are always 0; therefore, the content of these registers is always a multiple of 2000 (octal) (1,024 decimal). The data and instruction areas must begin on a 2000 (octal) word boundary and must be a multiple of 2000 (octal) words.

Address range checking does not occur during exchange sequences and I/O transfers. Memory addresses that are generated by these operations are absolute memory addresses.

Error Detection and Correction

Single-error correction/double-error detection (SECDED) circuitry monitors central memory for data errors. Memory errors that involve only 1 bit in each data word (single-bit errors) can be detected and corrected by the hardware. Double-bit errors can be detected but cannot be corrected. Errors that involve more than 2 bits cannot be reliably detected.

When a 64-bit word (bits 0 through 63) is written to memory, an 8-bit checkbyte is generated and stored in memory with the data word. The check bits are numbered 0 through 7 and are stored as data bits 64 through 71. When the word is read from memory, a checkbyte is again generated and compared with the original checkbyte, using an exclusive OR (XOR) operation. The result of the comparison is called a syndrome code. If all the bits in the syndrome code are 0, the 2 checkbytes are identical and no memory error occurred.

If the syndrome code contains one or more 1 bits, some type of memory error occurred. The type of memory error (single-bit or double-bit) can be determined by interpreting the syndrome code. If a single-bit error occurs, the syndrome indicates the bit in error and the SECDED logic toggles the incorrect bit to its correct value. If a double-bit error occurs, the syndrome code indicates that there is an error, but it cannot determine the incorrect bits. Errors that involve more than 2 bits produce unpredictable results. In some cases,

errors produce unique syndrome codes that can be detected by the SECDED logic. In other cases, the syndrome code appears to be a no-error condition or a single- or double-bit error.

Table 6 shows the data bits that are used to generate each bit in the checkbyte. All data bits that are marked with an X contribute to the corresponding check bit. The parity of all data bits that are marked with an X determines the state of the check bit. If the parity is even, the check bit is set to 0. If it is odd, the check bit is set to 1. For example, the data bits that make up check bit 0 are bits 1 through 29 (odd) and 31 through 55 (all). If an even number of these bits is 1, check bit 0 is set to logic 0; otherwise, it is set to logic 1.

If a syndrome code other than all 0's is generated, memory error information is recorded to help isolate the hardware failure. A nonzero syndrome code may also initiate an exchange sequence, depending on the state of 2 bits in the exchange package mode register. If the interrupt-on-correctable memory error (ICM) bit is set, a single-bit (correctable) memory error sets the memory error flag in the exchange package flag register and starts an exchange sequence. If the interrupt-on-uncorrectable memory error (IUM) bit is set, a double-bit or detectable multiple-bit (uncorrectable) error sets the memory error flag and starts an exchange sequence. If either the ICM or the IUM bit is clear, the corresponding memory error does not start an exchange sequence and does not set the memory error flag.

In an SV1 system, all data that is written to memory must pass through the cache (CA) ASICs, in which the 8 checkbits are generated. For read data, the cache ASICs perform SECDED on the data. A read operand with a double-bit error is returned to the CPU as all-zero data. The CA ASICs report any single or double-bit errors to the CPU for recording.

Table 6. Check-bit Generation

	Data Bits								Data Bits							
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
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	x	x	x	x	x	x	x	x
Check Bit 3	x	x	x	x	x	x	x	x	x	x	x	x	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	x			x		x	x		x			x		x	x	
	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Check Bit 0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Check Bit 1	x	x	x	x	x	x	x	x	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		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	x			x		x	x		x			x		x	x	
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
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	x			x		x	x		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	x	x	x	x	x	x	x	x
Check Bit 7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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	x			x		x	x		x			x		x	x	
Check Bit 4	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Check Bit 5	x	x	x	x	x	x	x	x	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								

Central Memory Performance Summary

Access time is the time an instruction requires to transfer one or more operands from central memory to an operating register. Access time depends on the type of register that receives the operand(s) and the number of operands that are transferred. Timings for memory operations are listed in Table 7. If no memory conflicts occur, each register type has access times as listed in the table. Access times from cache are also given.

The SV1 uses a 100-MHz system clock and a 300-MHz CPU clock. The SV1ex uses a 100-MHz system clock and a 500-MHz CPU clock.

SV1ex memory uses 3 different clocks:

- Chip-to-chip is 200 MHz
- The internal chip clock is 100 MHz for some tasks and 200 MHz for other tasks
- SDRAM interface is 133 MHz

SV1ex I/O uses a 100-MHz clock for shared areas and the backplane.

The system clock is used in the memory area and for transmission of signals across the backplane to and from memory. It is also used in the I/O and shared areas. The CPU clock is used in the CPU and cache ASICs and in about half of the processor module memory interface.

Table 7. Timings for Memory Operations

Operation	Port Busy						Register Ready (data in cache)						Register Ready (data not in cache)					
	SV1		SV1e		SV1ex		SV1		SV1e		SV1ex		SV1		SV1e		SV1ex	
	C	N	C	N	C	N	C	N	C	N	C	N	C	N	C	N	C	N
Vector Load	10	20	10	20	10	20	25	83	22	44	22	44	109	363	173	346	159	318
Vector Store	13	43	15	30	15	30	2	7	2	4	2	4	2	7	2	4	2	4
Vector Gather	13	43	14	28	14	28	28	93	25	50	25	50	115	383	176	352	162	324
Vector Scatter	16	53	15	30	15	30	2	7	2	4	2	4	2	7	2	4	2	4
B/T Load	5	17	10	20	10	20	25	83	20	40	20	40	108	360	171	342	157	314
B/T Store	5	17	11	22	11	22	5	17	5	10	5	10	5	17	5	10	5	10
A/S Load			11	22	11	22	22	73	17	34	17	34	105	350	168	336	154	308
A/S Store			10	20	10	20			2	4	2	4			2	4	2	4

Notes:

1. C = clock times. Times for the SV1 are in 300-MHz CPU clocks. Times for the SV1e and SV1ex are in 500-MHz CPU clocks.
2. N = nanoseconds.
3. Times for block transfers (V, B, T registers) have an implicit L/2 added to the given time, where L is the number of words transferred.
4. For loads, the register ready time is when the data is available in the register. For stores, the register ready time is when the register is free for a subsequent use. The times (Cs) listed for these instructions does not include chaining or tailgating. Data times with cache and memory increase by 5 Cs when chaining or tailgating occurs. However, these instructions may begin execution many Cs earlier.
5. Times are the minimum for each case. Longer times may result from memory contention.

The maximum central memory data transfer rate per processor module is 8 requests per system CP, one request to each of the 8 sections of memory (8 read requests or 4 read requests and 4 write requests per CP). Each of the 4 CPUs is capable of generating a maximum of 4 read requests per CPU CP or 2 read requests and 2 write requests per CPU CP (12 CPU requests per system CP). Any one CPU has full access to memory via the backplane if the other 3 CPUs are not actively making such memory requests.

The maximum data transfer rates for a CPU per requesting register are as follows:

- 1 word (read or write) per 2 CPs for A and S registers
- 4 words (2 read and 2 read) per CP for B, and T registers together or,
- 4 words (2 read and 2 write) per CP for B, and T registers together
- 4 words (2 read and 2 read) per CP for V registers
- 4 words (2 read and 2 write) per CP for V registers
- 2+ words (read) per CP for an instruction fetch (32 requests in 14 CPs)
- 4 words (read) per CP for an exchange sequence (4 CPs) followed by
- 2 words (write) per CP for an exchange sequence (8 CPs for 12 CPs total)
- 2 words (read and write) per CP for an I/O transfer

If memory conflicts occur, access times increase and data transfer rates decrease. Addressing conflicts within the CPU also cause a data transfer rate decrease.

VME I/O Section

A wide selection of peripherals can interface with the system through the 64-bit architecture VME IOS, which communicates with the CPU through a Y1 channel. Each processor supports four Y1 channel pairs. The I/O section uses port D in each processor module to transfer data between central memory and I/O channels. Table 8 shows each CPU and its associated I/O channels.

The CC ASIC controls all channel activity. There are 2 CC ASICs on each processor module. Each CC ASIC controls 2 paddle cards/slots: CC0 controls J1:J2 and CC1 controls J2:J3.

Table 8. Processor Modules and Associated Y1 Channel Numbers

Processor Module Number	CC0				CC1			
	Y1 Channels		Y1 Channels		Y1 Channels		Y1 Channels	
	Input	Output	Input	Output	Input	Output	Input	Output
0	20	21	22	23	24	25	26	27
1	30	31	32	33	34	35	36	37
2	40	41	42	43	44	45	46	47
3	50	51	52	53	54	55	56	57
4	60	61	62	63	64	65	66	67
5	70	71	72	73	74	75	76	77
6	100	101	102	103	104	105	106	107
7	110	111	112	113	114	115	116	117

Note: All channel numbers listed are octal numbers.

Y1 Channel Pairs

Each Y1 channel has two registers that can be loaded from any CPU. The channel address (CA) register contains the address of the next word in central memory to be transferred. When an I/O transfer begins, the CA register contains the address of the first word to be transferred. After the first word is transferred, the CA register increments. The next word is transferred and the CA register increments again. This process continues until all words are transferred.

The contents of the channel limit (CL) register determine the address of the last word in central memory to transfer. An I/O transfer completes when the contents of the CA register equal the contents of the CL register. The word at address (CL) is not transferred; address (CL) - 1 contains the last word transferred.

Channel Programming

Any CPU that is in monitor mode can initiate data transfers through a Y1 channel. Once a transfer is initiated, the transfer operates as a background activity and the CPU may resume other processing. When the transfer

completes, the channel sets an I/O interrupt request (IOI) flag in a CPU. The CPU that receives the interrupt request is not necessarily the same CPU that initiated the transfer.

Table 9 lists the instructions that are applicable to the Y1 channels. Instructions 0010*jk* through 0012*j*1 perform channel control and can be executed only by a CPU that is in monitor mode. There is no hardware interlock between CPUs; the programmer must ensure that two CPUs do not try to control the same channel at the same time. Instructions 033*i*00 through 033*ij*1 transmit I/O status information to register A*i*. These instructions are not limited to monitor mode, and any number of CPUs can execute them simultaneously.

Table 9. Y1 Channel Instructions

Machine Instructions	CAL Syntax	Description
0010*jk*[a]	CA,A*j* A*k*	Set channel (A*j*) CA register to (A*k*) and begin I/O sequence
0011*jk*[a]	CL,A*j* A*k*	Set channel (A*j*) CL register to (A*k*)
0012*j*0[a]	CI,A*j*	Clear channel (A*j*) interrupt and error flags Clear device master clear (output channel)
0012*j*1[a]	MC,A*j*	Clear channel (A*j*) interrupt and error flags Set device master clear (output channel) Clear device ready held (input channel)
033*i*00	A*i* CI	Transmit interrupting channel number to A*i*
033*ij*0	A*i* CA,A*j*	Transmit (CA) of channel (A*j*) to A*i*
033*ij*1	A*i* CE,A*j*	Transmit channel (A*j*) error flag to A*i*

[a] This instruction is privileged to monitor mode.

The following sequence of instructions initiates a data transfer through a Y1 channel.

Step	Machine Instruction	CAL	Comment
1	0011*jk*	CL,A*j* A*k*	Sets the CL register to (A*k*), where A*k* contains the address of one word after the last word to be transferred.
2	0010*jk*	CA,A*j* A*k*	Sets the CA register to (A*k*), where A*k* contains the address of the first word to be transferred.

This sequence starts the I/O transfer and increments the CA register after each data word transfers to or from the mainframe. On an output channel, the transfer stops when the contents of CA equal the contents of CL. On an input channel, the transfer stops when the contents of the CA register equal the contents of the CL register.

It is important to remember two characteristics of the Y1 channels when you program an I/O transfer. First, the CL register must be loaded before the CA register; the transfer begins when the CA register is loaded regardless of the contents of the CL register. Second, the CA register must be loaded with a value that is less than the contents of the CL register; if the CA register is loaded with a value that is equal to or greater than the CL register contents, unpredictable results occur.

Channel Operations

The SV1 has two types of logical channels: command and data. CPU instructions control command input and command output channels. The CPU uses command output channels to send commands to the IOS; it uses command input channels to receive status from the IOS. The IOS uses data channels to transfer data between mainframe memory and I/O buffer board (IOBB) memory.

The multiplexer I/O processor (MIOP) initiates all of the actual channel operations as I/O task control blocks (IOTCBs). The MIOP must set up an IOTCB in IOBB memory for every transfer between mainframe memory and the IOBB (a maximum of seven outstanding IOTCBs is allowed). There are two types of IOTCBs: I/O IOTCBs (S = 0) and console IOTCBs (S = 1). Figure 7 and Figure 8 show the formats of the IOTCBs.

Figure 7. I/O IOTCB Format

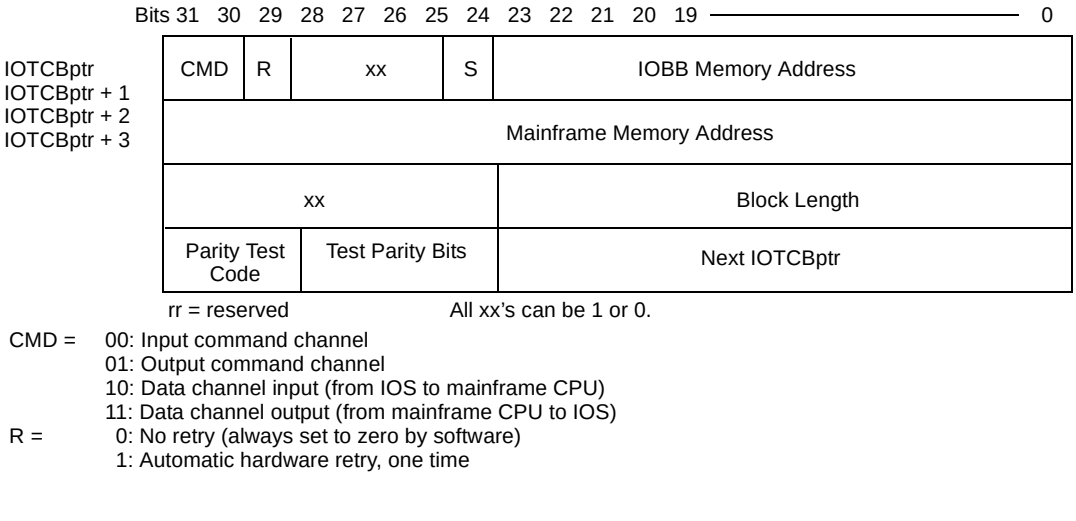
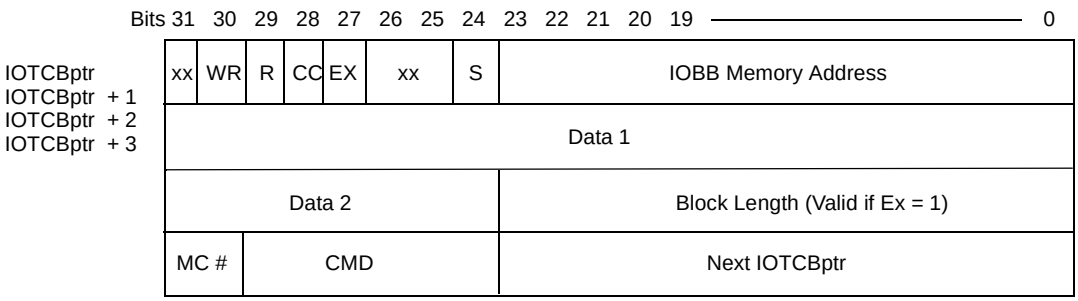


Figure 8. Console IOTCB Format



MC# = 00: Command is for MC ASIC of processor number 0
01: Command is for MC ASIC of processor number 1
10: Command is for MC ASIC of processor number 2
11: Command is for MC ASIC of processor number 3
CC = 1: Local operation (local to CC), no need to initiate any console bus cycles
EX = 1: Extra data; the following fields are valid:
IOBB memory address - starting address
length - number of 32-bit words transferred
WR = 0 - write from IOBB
WR = 1 - write to IOBB
R = 0: No retry (same as I/O IOTCB)
1: Automatic hardware retry, one time

IOBB Memory Address

The IOBB memory address is the starting address in IOBB memory from which data should be read or to which data should be written. The IOBB address must be divisible by 32 (that is, bits 0 through 4 = 00000) for a 32-, 64-, and 128-word burst.

Mainframe Memory Address

The mainframe I/O memory address is a starting address in mainframe memory from which data should be read or to which data should be written. This field is ignored by the CC ASIC if CMD = 0x.

Block Length

The number of 32-bit words to be transferred is limited by the total amount of memory on the IOBB. The length must be even (that is, the CC ASIC ignores bit 0). This field is ignored by the CC ASIC if CMD = 0x.

Next IOTCBptr

The next IOTCBptr (IOTCB pointer) is the IOBB memory address where the next IOTCB resides.

Note: The Next IOTCBptr signal is a 32-bit word address and must be divisible by 4 (that is, bits 1 and 0 = 00).

If the block length is 0, a no-operation instruction occurs (data channel only); however, the IOTCBptr is loaded and a completion interrupt is generated normally.

Command Input Channel

Five registers in the CC ASIC correspond to each input channel:

- Channel address (CA): the starting memory address in mainframe memory
- Channel limit (CL): the limiting memory address in mainframe memory
- Channel error flag (CE)
- Channel interrupt flag (CI)
- Channel number (C#)

The operating sequence of an input channel is as follows:

1. The CPU loads the CL register.
2. The CPU loads the CA register; the corresponding input channel (C#) is opened.
3. The CC ASIC sends a Ready to Receive Return Status Block (RSB) interrupt signal to the MIOP through the IOBB.
4. The MIOP sets up an RSB in IOBB memory.
5. The MIOP sets up an IOTCB (CMD = 00; IOBB memory address = starting address for RSB to be read).
6. The MIOP sends an IOTCB Pending interrupt signal to the CC ASIC through the IOBB.
7. The CC ASIC fetches IOTCB (designated by IOTCBptr), picks up the IOBB memory address, uses the CA and CL registers instead of the mainframe memory address from the IOTCB, and completes the transfer.
8. The CC ASIC interrupts the CPU when CA = CL.

9. The CC ASIC sends an IOTCB Done interrupt signal to the MIOP through the IOBB.

Command Output Channel

Five registers in the CC ASIC correspond to each output channel:

- Channel address (CA): the starting memory address in mainframe memory
- Channel limit (CL): the limiting memory address in mainframe memory
- Channel error flag (CE)
- Channel interrupt flag (CI)
- Channel number (C#)

The following steps describe the operating sequence of an output channel:

1. The CPU sets up a control block (CB) in mainframe memory.
2. The CPU loads the CL register.
3. The CPU loads the CA register; the corresponding output channel (C#) is opened.
4. The CC ASIC sends a CB Pending interrupt signal to the MIOP through the IOBB.
5. The MIOP sets up an IOTCB (CMD = 01; IOBB memory address = starting address for loading CB).
6. The MIOP sends an IOTCB Pending interrupt signal to the CC ASIC through the IOBB.
7. The CC ASIC fetches IOTCB (designated by IOTCBptr), picks up the IOBB memory address, uses the CA and CL registers instead of the mainframe memory address from IOTCB, and completes the transfer.
8. The CC ASIC interrupts the CPU when CA = CL.
9. The CC ASIC sends an IOTCB Done interrupt signal to the MIOP through the IOBB.

Data Channels (Input and Output)

The following steps describe the operating sequence of a data channel:

1. The MIOP sets up an IOTCB (CMD = 10 or 11,...).
2. The MIOP sends an IOTCB Pending interrupt signal to the CC ASIC through the IOBB.
3. The CC ASIC fetches IOTCB (designated by IOTCBptr), interprets all parameters from IOTCB, and completes the transfer.
4. The CC ASIC sends an IOTCB Done interrupt signal to the MIOP through the IOBB.

Error Handling

Error detection is applicable to parity across the data bus portion of the Y1 bus. Data as well as address and control are multiplexed into the 32-bit data bus. Therefore, all errors are parity errors that occur at different instances across the Y1 bus.

Two types of parity errors occur: those associated with IOTCB fetch and those associated with IOTCB execution. If a parity error occurs while an IOTCB is fetched, the IOTCB does not execute. The IOTCB controller clears its IOTCB pending queue, resets IOTCBptr to zero, and sends an IOTCB Fetch Error Interrupt signal to the MIOP through IOBB, which is the default beginning of the IOTCB chain.

Two types of parity errors occur during IOTCB execution: command channel errors (CMD = 0x) and data channel errors (CMD = 1x). In both cases, when a parity error is detected, the IOTCB execution continues. When the entire transfer is finished, if the retry bit in the IOTCB is set, hardware automatically retries once. If successful, IOBB receives a normal IOBB Done interrupt signal, and a scan-only flip-flop indicates that a successful retry occurred. If the retry is not successful, an IOTCB Execution Error interrupt signal is sent to the MIOP through the IOBB. The MIOP must then take appropriate actions. The IOTCB controller then fetches the next IOTCB if one is pending. In addition, if the IOTCB in error is command-channel related, the corresponding channel error (CE) flag does not set. The 033*ij*1 instruction reads the status of the CE flag.

High Performance Parallel Interface (HIPPI)

The High Performance Parallel Interface (HIPPI) is a 100-Mbyte/s channel that transfers data between data-processing equipment on multiple twisted-pairs of copper cable at distances up to 82 ft (25 m).

The HIPPI signal protocol is designed to be independent of distance; it therefore enables the average data rate to approach the peak data rate, even at distances longer than specified for the HIPPI channel.

The following list describes other characteristics of the HIPPI interface:

- The HIPPI is a simplex interface; it can transfer data in one direction only. Two HIPPI channels may be used to implement a full-duplex interface.
- Data transfers are performed and flow is controlled in increments of bursts; each burst normally contains 256 words.
- Signals and control sequences are simple, and a look-ahead flow control enables average transfer rates for large file transfers to approach the peak transfer rate, even at distances longer than specified for HIPPI cables.
- The HIPPI provides support for low-latency, real-time, and variable-size packet transfers.
- The HIPPI is also designed to transmit multiple packets after a connection is established. No round-trip cable delays are required between packets.

HIPPI Channel Operational Overview

The following paragraphs describe the seven channel instructions. All instructions except the 033*i*00, 033*ij*0, and 033*ij*1 instructions are privileged to monitor mode.

Load Control Registers and Start Channel (0010*jk*)

The content of A*j* specifies the channel number, and the contents of A*k* are loaded into the next register in the sequence. This sequence is specified by the bit map register, and processing occurs from the right to the left, or from the least-significant bit (LSB) to the most-significant bit (MSB).

This instruction is used to load the control word, D1 address, D1 block length, D2 address, D2 block length, and connection control information registers. This instruction is executed multiple times, once for each register that is to be loaded. The bit map register is used to specify how many and which registers

are to be loaded. The channel operation is started when the last specified register is loaded, unless bit 15 of the bit map register is set, which inhibits channel activation. There is no time limit in which to complete the loading. If the bit map register is reloaded, then this becomes the new load sequence and the previous sequence is ignored.

Load Register Bit Map (0011*jk*)

The content of A*j* specifies the channel number and the content of A*k* is loaded into the bit map register for that channel. This specifies the load and read sequence for the remaining registers. This instruction should be executed before each sequence of 0010*jk* and 033*ij*0 instructions because the register is cleared during processing. Each time this register is loaded, a new sequence is started.

Clear Pending Interrupt (0012*j*0)

The content of A*j* specifies the channel number for a clear pending interrupt and sets an error flag. This instruction clears the pending interrupt, error flags, and real-time status register and advances the sequence of operation in the channel control word to the next field. It also restarts the channel if processing was discontinued during a normal interrupt.

Reset Channel (0012*j*1)

The content of A*j* specifies the channel number for this instruction. This instruction resets the corresponding HIPPI channel. The interrupt is cleared if one is pending; any error flags are reset, and all channel operations are terminated. The channel returns to its initial state.

This instruction executes before the channel is used for the first time and after any sequence that places the channel into a test mode configuration. After some channel error conditions, a reset is required before the channel can be reactivated.

Read Highest Priority Interrupting Channel Number (033*i*00)

The channel number of the interrupting channel that has the highest priority is placed into A*i*. This instruction operates in the same manner as it does on the Y1 interface channels.

Read Control Registers (033*ij*0)

The content of A*j* specifies the channel number and A*i* is loaded from the register specified by the bit map register. The bit map register must be loaded for each new register selection. No sequencing is provided for read operations.

This instruction is used to read the control word, D1 address, D1 block length, D2 address, D2 block length, connection control information, real-time status, LLRC, operational status, sequence error idle (SEI) disable status, flow status word 1, and flow status word 2 registers (if they exist for that channel). This instruction is executed multiple times, once for each register that is to be read from. The bit map register specifies which register is to be read from.

If this instruction is executed with no bits set in the bit map register, then the channel address of the currently processing data area is returned. Execution of this instruction in non-monitor mode should return the current data area address. The bit map register should be cleared before user mode is entered.

The RT status register is read if this is the first channel operation after a reset operation. (0012*j*1)

Read Channel Error Flag (033*ij*1)

The contents of A*j* specify the channel numbers, and A*i* is loaded with the error flag for the specified channel. Bit 0 is the error flag; if it is equal to 1, that indicates an error was detected. If no error was detected, then A*i* contains 0’s for all bit locations.

HIPPI Channel Configurations

Table 10 shows the possible input/output channel configurations for the Y1 channel, the HIPPI input channel (HI-I), and the HIPPI output channel (HI-O). The configuration has a physical limitation because HIPPI channels and Y1 channels cannot be combined on the same CC ASIC.

Table 10. HIPPI or Y1 Channel Configurations

Proc Mod 0	Proc Mod 1	Proc Mod 2	Proc Mod 3	Proc Mod 4	Proc Mod 5	Proc Mod 6	Proc Mod 7
Paddle Card Slot J1							
*20/21, Y1	*30/31, Y1 *30, HI-I	40/41, Y1 40, HI-I	50/51, Y1 50, HI-I	60/61, Y1 60, HI-I	70/71, Y1 70, HI-I	100/101, Y1 100, HI-I	110/111, Y1 110, HI-I
Paddle Card Slot J2							
*22/23, Y1	*32/33, Y1 *33, HI-O	42/43, Y1 43, HI-O	52/53, Y1 53, HI-O	62/63, Y1 63, HI-O	72/73, Y1 73, HI-O	102/103, Y1 103, HI-O	112/113, Y1 113, HI-O
Paddle Card Slot J3							
24/25, Y1 24, HI-I	34/35, Y1 34, HI-I	44/45, Y1 44, HI-I	54/55, Y1 54, HI-I	64/65, Y1 64, HI-I	74/75, Y1 74, HI-I	104/105, Y1 104, HI-I	114/115, Y1 114, HI-I
Paddle Card Slot J4							
26/27, Y1 27, HI-O	36/37, Y1 37, HI-O	46/47, Y1 47, HI-O	56/57, Y1 57, HI-O	66/67, Y1 67, HI-O	76/77, Y1 77, HI-O	106/107, Y1 107, HI-O	116/117, Y1 117, HI-O

* Either paddle card 0 or paddle card 1 can be configured as the deadstart channel. By default, paddle card 0, Y1 channel 20/21, is configured as the deadstart channel.

VME IOS and HIPPI I/O Interrupts

I/O interrupts originate at the interrupting channel. The CI ASIC passes the necessary information to the global JS ASIC logic. When an I/O interrupt occurs, the CI sends an I/O interrupt command to the JS ASIC along with the number of the interrupting channel. Then, this information is sent over the JS/JS bus to the global JS logic on each shared resources JS ASIC.

The global JS logic routes the interrupt information to the I/O interrupt handling logic. There, the appropriate bit in the I/O interrupt register is set. An interrupt to one of the PV ASICs (SV1 series) or PVC ASICs (SV1ex series) is generated according to the following rules:

1. If any processor is in monitor mode, no interrupt is generated.
2. If all processors are in user mode and any processor has its sequence error idle (SEI) bit set, the interrupt is directed to the lowest-numbered processor that has its SEI bit set.
3. If all processors are in user mode, none of them have the SEI bit set, and a processor is waiting on semaphore, the interrupt is directed to the lowest-numbered processor that is waiting on semaphore.

4. If all processors are in user mode, none have SEI set, and none are waiting on semaphore, then the interrupt is directed to the last processor to clear an I/O channel. The interrupt is directed to the last processor that cleared any channel.

Once the software determines which processor to interrupt, the JS associated with that processor sends the I/O interrupt command over the PV/JS bus. If one of the processors is in monitor mode and no interrupt is generated, it is assumed that the processor in monitor mode will handle I/O interrupts before it exchanges into user mode. It is possible that an I/O interrupt could arrive after the processor in monitor mode has finished handling I/O interrupts but before it exchanges back to user mode. In that case, the I/O interrupt logic on the JS senses that none of the processors are in monitor mode and I/O interrupts are still pending. The criteria listed above are applied, and an I/O interrupt is posted to one of the processors.

A processor handles I/O interrupts by issuing 033*i*00 instructions until A*i* = 0. When a 033*i*00 instruction is executed, the PV sends the command to the local JS ASIC. The local JS ASIC determines the channel number of the highest-priority interrupting channel and returns it to the originating PV on the PV/JS bus.

When a processor clears a channel, the processor sends the clear channel command to the local JS. The local JS passes it on to the other JS ASICs on the JS/JS bus. This command is then forwarded to the CI, which handles the channel clear operation, and to the I/O interrupt logic, which clears the interrupt flag for that channel in its I/O interrupt register.

For each channel, there is a single priority bit that indicates whether it is a high- or low-priority channel. When a processor requests the highest priority channel, that channel access is determined as follows:

1. If any high-priority channels have an interrupt pending, the lowest-numbered channel is the one returned.

2. If no high-priority channels have an interrupt pending, the lowest-numbered, low-priority channel with an interrupt pending is returned.

The priority is set via the joint test action group (JTAG) control of the system. Normally, this priority scheme is configured at system power-up, but it is possible to change it while the system is running.

VME IOS and HIPPI I/O Memory Errors

Memory errors that occur during I/O operations present a challenge for Cray SV1 series systems. When a memory error occurs, the associated processor is notified. On the SV1 system, the I/O channels do not share memory ports with specific processors. Each I/O channel is loosely associated with the four processors that share its processor module.

One of the processors is selected to handle I/O memory errors by using JTAG. When a memory error occurs on I/O, the CI passes the error information to the local JS. The JS then posts the memory error to one of the four local processors that are configured to handle the I/O memory errors.

GigaRing I/O Section

Peripherals are connected to the SV1 system via the GigaRing I/O system. GigaRing technology defines a standard that enables a system integrator to connect various devices on a ring topology. Refer to the “GigaRing Overview” section for more information about the GigaRing channel and its operation.

The GigaRing node contains client logic, a GigaRing node chip, and a fully duplexed, bidirectional client-port interface. The GigaRing node chip, a single application-specific integrated circuit (ASIC), contains an input and an output link for both the positive and negative rings and a bidirectional client-port interface. The data path is 32 bits wide on each of the counter-rotating rings and 64 bits wide on the client port. The client port may be configured to operate in half-width mode (32 bits) for clients that do not require the bandwidth of the 64-bit interface.

A single-purpose node (SPN), a Cray computer system, or a multipurpose node (MPN-1) is referred to as a client node on the GigaRing channel. Each client node contains its own client logic and a GigaRing node chip. Each client node communicates with the other client nodes through the GigaRing node chip. (The client nodes are referred to hereafter as clients.)

The "GigaRing I/O Section" addresses the following related topics:

- MPN
- IPN
- FCN
- HPN-1 and HPN-2
- BMN
- ESN
- FOX
- Error Handling and Reporting

MPN-1 Functional Overview

The multipurpose node (MPN) connects specific industry standard SBus-based I/O peripherals or proprietary channels to the GigaRing channel to provide I/O services for the mainframe node.

The MPN logic components and SBus controllers reside inside the multipurpose node subrack (MPN-1). The MPN-1 provides forced-air cooling and supplies power to the MPN logic and SBus controllers.

A maximum of eight industry standard SBus controllers or Cray proprietary channels can reside within the MPN-1. The MPN-1 supports the following SBus controllers:

- Small computer system interface (SCSI) disk and tape drive interface
- Ethernet network interface
- Asynchronous transfer mode (ATM) network interface
- Fiber distributed data interface (FDDI) network interface
- Cray proprietary supervisory channel

The MPN-1 subrack and all associated MPN-1 peripheral subracks (such as SCSI disk or tape) reside in the PC-10 cabinet.

All GigaRing based systems require one MPN-1 subrack to be configured with one SBus Ethernet and one SBus SCSI disk interface.

All peripheral and GigaRing channel cable connections occur at the rear of the MPN-1. The front of the MPN-1 displays various MPN-1 messages and node activity.

MPN-1 Operation Overview

The MPN-1 is based on a memory-mapped bridge architecture that enables the SBus Peripheral Interface (SPI) *hyper*SPARC processor to address the memory of other IONs on the GigaRing channel.

The SBus controller takes information from its I/O peripheral device and places it on the SPI's SBus when requested. The SBus controller manages the peripheral data transfer. The SPI controls the SBus. The SPI converts the SBus data into MBus data and places it on the MBus. The MBus interface follows the level 1 device specification, which identifies how MBus transactions (MBus read or write operations) are performed. The MBus interface controls the data that is transferred to and from the translation windows and between the SPI and SSB.

The *hyper*SPARC processor opens enough translation windows to store the data from the peripheral device and generates the tag that each translation will use.

The tag is made up of the command tag and the address tag. The command tag contains the transfer type, transfer size, target node address, information used to manage the MBus translation, and GigaRing channel control information. The address tag contains the memory address of the target node to be accessed.

The event and receive processors generate or decode the GigaRing packet header and control the flow of information between the translation windows and the FIFOs to the GigaRing node chip. The processors also use parts of the tag to form the GigaRing packet header. This header is used by the GigaRing node chip to send data to or receive data from another node.

When the transfer is complete, the *hyper*SPARC processor closes the translation windows, which allows them to be reused for another transfer.

IPN-1 Functional Overview

The intelligent peripheral node interface (IPN-1) provides an interface between the GigaRing channel and the single-disk or disk array devices that support level 2 intelligent peripheral interface (IPI-2) protocol. Functionally, the IPN-1 is identical to five DCA-2 disk controllers, or to one DCA-3 disk controller that is installed with Cray IOS model E systems.

The IPN-1 allows existing Cray intelligent peripheral interface (IPI-2) products to connect to the GigaRing architecture. The IPN-1 supports the following single-drive configurations:

- DD-301 (1.377 Gbytes and 8.2 Mbytes/s)
- DD-302 (1.8 Gbytes and 9.3 Mbytes/s)
- DD-60 (1.96 Gbytes and 20 Mbytes/s)
- DD-62 (2.73 Gbytes and 8.1 Mbytes/s)

The IPN-1 supports the following RAID 3 (4 data units + 1 parity unit) configurations:

- DA-301 (5.5 Gbytes and 32.8 Mbytes/s)
- DA-302 (7.2 Gbytes and 37.0 Mbytes/s)
- DA-60 (7.84 Gbytes and 80.0 Mbytes/s)
- DA-62 (10.92 Gbytes and 32.5 Mbytes/s)

The IPN-1 attaches to existing disk enclosures such as the DE-60 and DE-100.

IPN-1 Components

The following paragraphs describe the major components of the IPN-1 GigaRing interface PCB (motherboard).

- The GigaRing option supports the data connection between the IPN-1 and the GigaRing channel.

- The motherboard uses two buses to transfer data between the various options. The SBus operates at 25 MHz. The IBus operates at 100 MHz. Neither bus has data protection.

- The client interface option (CLI) manages the data connection from the GigaRing option to the IPI-2 channel PCB. The CLI option also transfers data and control among the GigaRing node chip and the channel and SPARC support option (CSS) via the IBus.

- The microSPARC microprocessor chip is responsible for all control functions within the node. It communicates with the CSS option via the SBus.

- The microSPARC DRAM consists of 2 banks of 1, 4, or 16 Mbytes of memory mounted on single inline memory modules (SIMMs).

- The CSS option transfers data and control between the SBus and the IBus. The CSS also buffers information between the microSPARC chip and the rest of the IPN-1. The CSS controls the IPI logic and supports the microSPARC boot RAM, RS-232 interface, and SBus-to-channel memory.

FCN-1 and FCN-2 Functional Overview

The fibre channel node (FCN-1 and FCN-2) is an interface between a GigaRing channel and up to five fibre channel arbitrated loops (FC-AL). The FCN-1 uses the standard node/client interface that all GigaRing channel single-purpose nodes (SPNs) support. The FCN-1 and FCN-2 are nearly identical in design, except as noted below.

- The FCN-1 can connect to the DSF-1 or DSF-2 subracks via copper cables that have a maximum cable length of 98 ft (30 m).
- The FCN-2 has the added capability to connect, through fiber-optic cables, to DSF-2 subracks at a maximum distance of 983 ft (300 m).

FC-AL is an ANSI-standard serial communications channel that provides a peak bandwidth of 100 Mbytes/s. The ANSI FC-AL standard specifies a loop topology that supports up to 126 devices on a copper or a fiber-optic ring.

Cray's implementation of the FC-AL uses a copper connection and supports a maximum of 80 disk devices on each fibre channel (40 primary path, 40 alternate path). An FCN-1 and FCN-2 have connections for five FC-ALs; this provides a total peak bandwidth of 500 Mbytes/s.

The FCN-1 and FCN-2 support the following disk configurations:

- DD-308 single-spindle drives; 9.5-Gbyte capacity, with 8- to 12-Mbyte/s transfer rates
- DA-308 serial RAID 3 4+1 (4 data drives and 1 parity drive) arrays; 38-Gbyte capacity, with 32- to 48-Mbyte/s transfer rates
- DD-309 single-spindle drives; 18.2-Gbyte capacity, with 13.8- to 21-Mbyte/s transfer rates
- DA-309 serial RAID 3 4+1 (4 data drives and 1 parity drive) arrays; 72.8-Gbyte capacity, with 55- to 84-Mbyte/s transfer rates
- DD-331 single-spindle drives; 36-Gbyte capacity, with 18- to 30-Mbyte/s transfer rates
- DA-330 serial RAID 3 4+1 (4 data drives and 1 parity drive) arrays; 144-Gbyte capacity, with 72- to 120-Mbyte/s transfer rates

The FCN-1 and FCN-2 attaches to the DSF-1 subrack. The DSF-1 can contain a maximum of ten DD-308 disk drives. Up to eight DSF-1 subracks can be attached to a single FC-AL.

FCN-1 and FCN-2 Hardware Description

FCN-1 and FCN-2 modules plug into any one of the four I/O node slots in the scalable I/O (SIO) node subrack (NSR-1). GigaRing and FC-AL cables attach to the front panel of the module.

The FCN-1 and FCN-2 contain three printed-circuit boards (PCBs) that are enclosed in a metal canister. These three PCBs include the following components:

- Power supply board
- GigaRing interface board (motherboard)
- Fibre channel client board (daughter board)

The FCN-1 and FCN-2 hardware support CRC generation, parity data generation, and data reconstruction for RAID 3 and RAID 5 configurations. These features support the error-recovery functions of the disk devices that are attached to the fibre channel loop.

HPN Functional Overview

The HPN-1 and HPN-2 are single-purpose nodes (SPNs). They provide an interface between the GigaRing channel and the networks and between the channel and the network disk arrays that support HIPPI data transmission. The HPN-1 and HPN-2 support HIPPI networks and switched networks.

The HPN-1 transfers data at 100 Mbytes/s and provides two 32-bit input/output channels (one input and one output connection per channel with a total of two input and two output connections). An HPN-1 can simultaneously support a HIPPI network on one input/output channel and network disks on the other.

The HPN-2 transfers data at 200 Mbytes/s and provides one 64-bit input/output channel (2 input and 2 output connections). The HPN-2 can be configured as a single 32-bit, 100-Mbyte/s input/output channel with one input and one output connection.

Note: The HPN-2 can be configured either as a single 200-Mbyte/s channel or a single 100-Mbyte/s HIPPI channel but not both simultaneously.

Network Disk Arrays

Network disk array systems provide a large amount of data storage in a small area. The HPN-1 and HPN-2 support the following network disk arrays:

- ND-12 network disk array
- ND-14 network disk array
- ND-30 network disk array
- ND-40 network disk array

The HPN-1 and HPN-2 support HIPPI networks and switched networks. Additionally, the HPN-1 and HPN-2 attach to network disk array system enclosures.

Hardware Description

HPN-1 and HPN-2 modules plug into any of the four SPN slots in the scalable I/O (SIO) node subrack (NSR-1). GigaRing and HIPPI cables attach to the front panel of the module.

The HPN-1 and HPN-2 each consist of three printed circuit boards (PCBs) that are enclosed in a metal canister.

These three PCBs include the following components:

- Power supply board
- GigaRing interface board (motherboard)
- HIPPI channel board (daughter board)

The HPN-1 and HPN-2 hardware supports odd-byte parity and length/longitudinal redundancy checkword (LLRC) error detection.

BMN-1 Functional Overview

The BMN-1 node is a single-purpose node (SPN) that is located in the node subrack (NSR-1) in the PC-10 cabinet.

The BMN-1 node connects systems on a GigaRing channel to mass-storage magnetic tape devices. The BMN-1 supports the Federal Information Processing Standards (FIPS) 60 interface specification. The BMN-1 has two independent tape channels.

Each of the two channels supports the following transfer modes:

- Interlock mode
- 1.5-Mbyte/s high-speed mode
- 200-ft offset interlock mode
- 3-Mbyte/s data streaming mode
- 4.5-Mbyte/s data streaming mode

The BMN-1 node supports any tape drive system that supports the FIPS 60 specification, which includes the following devices and tape library robots:

- IBM 3480 (18-track)
- STK 4480 (18-track)
- IBM 3490 (36-track)
- STK 4490 (36-track)
- 9-track reel tapes (3420 compatible)
- STK 4400
- STK 9310
- STK 9360

BMN-1 Hardware Description

BMN-1 modules plug into any of the four I/O node slots in the scalable I/O (SIO) node subrack (NSR-1). GigaRing and tape channel cables attach to the front panel of the module.

The BMN-1 module contains three printed circuit boards (PCBs), which are enclosed in a metal canister:

- Power supply board
- GigaRing interface board (motherboard)
- Tape channel board (daughter board)

ESN-1 Functional Overview

The ESN-1 connects the mainframe node on a GigaRing channel to mass-storage magnetic tape devices via an Enterprise Systems Connection Architecture (ESCON) interface.

The ESN-1 provides an optical-fiber communication link between channels and control units that implement the Enterprise Systems Architecture/390 specification. The ESN-1 has four independent ESCON channels. Each ESCON channel has a bandwidth of 17 Mbytes/s.

The ESN-1 supports the following tape devices:

- IBM 3490E (36-track enhanced tape device)
- STK 4490 (36-track tape device)
- STK 9490 (TimberLine)
- STK SD-3 (RedWood)
- IBM 3590 (Magstar)

The ESN-1 supports the following libraries and robots:

- IBM 3494
- IBM 3495
- STK 4400
- STK 9310
- STK 9360

ESN-1 Hardware Description

ESN-1 modules plug into any of the four I/O node slots in the scalable I/O (SIO) node subrack (NSR-1). GigaRing and tape channel cables attach to the front panel of the module.

The ESN-1 module contains three printed circuit boards (PCBs), which are enclosed in a metal canister:

- Power supply board
- GigaRing interface board (motherboard)
- Tape channel board (daughter board)

FOX Overview

The FOX-1 is a transparent and nodeless extension to the standard GigaRing channel. The FOX-1 extends the distance between nodes on the GigaRing channel to 656 ft (200 m). The standard copper-based distance limit is 36 ft (11 m).

The FOX-1 can extend the length of the channel for any of the GigaRing interconnected nodes that reside within or communicate with the SIO architecture. The FOX-1 physically connects to the GigaRing channel like any node, but it does not interface directly with any of the GigaRing channel protocol.

A subrack houses the FOX-1 hardware. The FOX-1 subrack usually resides inside a PC-10 cabinet; however, for mainframe nodes, the FOX-1 subrack can be located outside the PC-10.

GigaRing Implementation

Two FOX-1 subracks complete the optical extension of the GigaRing channel. Standard GigaRing channel copper cables bring data into and out of the FOX-1 just as they do for other GigaRing interconnected nodes. Eight-fiber ribbon cables establish the optical link between FOX-1 subracks.

GigaRing Configurations

The FOX-1 subrack functions as a transparent node, even when the GigaRing channel is reconfigured. The FOX-1 does not affect the information flow in either a folded or masked ring.

Hardware Overview

The FOX-1 hardware resides in a 2-SU 19-in. rackmount enclosure. This subrack contains the necessary power supplies and cooling hardware needed to ensure reliable operation of the FOX-1.

The FOX-1 receives GigaRing information over standard copper GigaRing cables. The FOX-1 converts the information from an electrical format to an optical format, and then retransmits the information onto a fiber-optic cable to a receiving FOX-1, where the process is reversed. The FOX-1 uses an array of optical transceivers and supporting circuitry to convert the information that is transmitted on the GigaRing channel.

Error Reporting and Handling

Each GigaRing interface maintains an error monitor that any node on the ring can access. This feature enables error monitor software to access all nodes on a given ring and provide a cumulative status of the system. Table 11 provides a description of each MMR that is associated with error reporting.

Table 11. Error Reporting MMRs

Add. (Octal)	Bits in Field	Field Name	Description	Access
30	24	ERROR_COUNTER	Counter for ring and client errors	GR RAZ
31	19	NEG_RING_ERRORS	Ring error bit map	GR RAZ
32	19	POS_RING_ERRORS	Ring error bit map	GR RAZ
33	8	CLIENT_ERRORS	Client error bit map	GR RAZ
36	32	NEG_BUFFER_PE_STATUS	Bit map of buffer parity errors	GR RAZ
37	32	POS_BUFFER_PE_STATUS	Bit map of buffer parity errors	GR RAZ
40	32	CRC_CAPTURE	CRC and sendtag of send packet with CRC error	GR RAZ

Interprocessor Communication

The mainframe interprocessor communication section possesses three features that enable data and control information to transfer between CPUs:

- Shared registers
- Semaphore registers
- Interprocessor interrupts

Shared registers pass data between CPUs. Semaphore (SM) registers enable synchronization of programs that are operating in different CPUs. Interprocessor interrupts allow a CPU to initiate an exchange sequence in other CPUs. These features are especially useful in multitasking environments.

The shared and semaphore registers are arranged in groups called clusters. The following paragraphs explain clusters, the shared and semaphore registers, test and set control, and interprocessor interrupts.

Clusters

The shared and semaphore registers are divided into (number of CPUs +1) identical clusters. A CPU can reference only one cluster at a time. The cluster number (CLN) register in the exchange package determines to which cluster a CPU is assigned. Clusters are numbered beginning with 1 (octal). A CLN value of 0 prevents a CPU from accessing all shared and semaphore registers.

There are two ways to load the CLN register: automatically during an exchange sequence or by executing instruction 0014*j*3 when the CPU is in monitor mode.

Shared Registers

Shared registers provide a way to transfer data between operating registers in different CPUs; one CPU loads a shared register from its own operating registers. Other CPUs can then transfer the data from the shared register to their own operating registers. There are two types of shared registers: shared address (SB) and shared scalar (ST).

Each cluster contains eight 32-bit SB registers, numbered SB0 through SB7. Data is transmitted between the SB registers and the A registers in each CPU that are assigned to the cluster.

Each cluster contains eight 64-bit ST registers, numbered ST0 through ST7. Data is transmitted between the ST registers and the S registers in each CPU that is assigned to the cluster.

Table 12 lists all instructions that transmit data to or from the shared registers. In a CPU where the contents of the CLN register equal 0, instructions 026*ij*7 and 072*ij*3 return a value of 0, and instructions 027*ij*7 and 073*ij*3 perform no operation.

Table 12. Shared Register Instructions

Machine Instruction	CAL Syntax	Description
026*ij*7	A*i* SB*j*	Transmit (SB*j*) to A*i*
027*ij*7	SB*j* A*i*	Transmit (A*i*) to SB*j*
072*ij*3	S*i* ST*j*	Transmit (ST*j*) to S*i*
073*ij*3	ST*j* S*i*	Transmit (S*i*) to ST*j*

Semaphore Registers

SM registers allow a CPU to temporarily suspend program operation in order to synchronize operation with other CPUs. Each cluster contains thirty-two 1-bit SM registers numbered SM0 through SM37 (octal). Each CPU that is assigned to the cluster can set or clear each SM register and can perform a test

and set instruction, which is explained in the following paragraph. Each CPU in the cluster can also transmit the contents of all 32 SM registers to or from an S register. CPUs use the shared paths to set and clear semaphore registers.

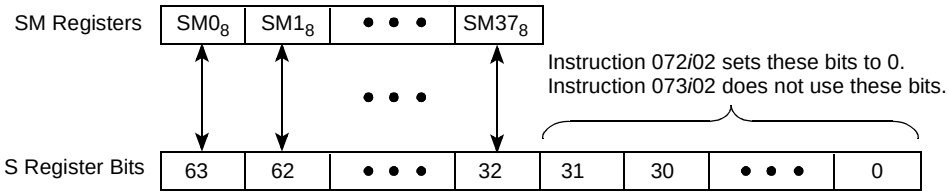
Table 13 lists all machine instructions that use the SM registers. The 0034*jk* test and set instruction tests the state of the SM*jk* register. If the content of the SM*jk* register is 0, the instruction executes immediately. If the content of the SM*jk* register is 1, the instruction holds issue until another CPU that is assigned to the same cluster clears the SM*jk* register. When the instruction issues, it sets the SM*jk* register. Instructions 0036*jk* and 0037*jk* clear and set the SM*jk* register.

Table 13. SM Register Instructions

Machine Instruction	CAL Syntax	Description
0034*jk*	SM*jk* 1,TS	Test and set semaphore *jk*
0036*jk*	SM*jk* 0	Clear semaphore *jk*
0037*jk*	SM*jk* 1	Set semaphore *jk*
072*i*02	S*i* SM	Transmit (SM) to S*i*
073*i*02	SM S*i*	Transmit (S*i*) to SM

Instructions 072*i*02 and 073*i*02 transmit the SM register contents to or from the upper half of the S register. Figure 9 shows the relation between the SM registers and the bits of an S register.

Figure 9. Relation between SM Registers and S Register Bits



If a CPU is not assigned to any cluster (that is, CLN = 0), instructions 0034*jk*, 0036*jk*, 0037*jk*, and 073*i*02 perform no operation. Instruction 072*i*02 sets register S*i* to 0.

The following example shows how an SM register is used to synchronize the operation of two CPUs in a multitasking program. In this example, CPU 0 computes a partial result needed by CPU 1 while CPU 1 computes a second partial result. CPU 1 then uses the two partial results as operands for further processing.

In Step 1, CPU 0 begins processing by setting register SM0, which indicates that it has not yet computed its partial result. In Steps 2 and 3, CPUs 0 and 1 begin to compute the partial results. At the end of Step 3, CPU 1 places its partial result in register S1. CPU 1 needs CPU 0's partial result before it can proceed. In Step 4, CPU 1 performs a test and set instruction on register SM0. Because register SM0 is already set, CPU 1 holds issue.

CPU 0	CPU 1
1. SM0 1 (003700)	
2. Compute partial result	3. Compute partial result
X	X
X	X
X	Place partial result in S1
X	
X	4. SM0 1, TS (003400)
X	
Place partial result in S1	
5. ST0 S1 (073103)	
6. SM0 0 (003600)	
7. Continue processing	8. S2 ST0 (072203)
X	
X	9. Continue processing
X	X
X	X
X	X

CPU 0 continues its computations and transfers its partial result to the S1 register. CPU 0 then transfers the partial result from S1 to register ST0 (Step 5). In Step 6, CPU 0 indicates that the partial result is ready in register ST0 by clearing register SM0. In Step 7, CPU 0 can now continue with other processing. SM0 is now cleared and the test and set instruction in CPU 1 issues, setting register SM0. CPU 1 then transfers CPU 0's partial result from register ST0 to register S2 (Step 8). CPU 1 now has its own partial result in register S1 and CPU 0's partial result in register S2 and can continue processing (Step 9).

Test and Set Control

The test and set control logic handles 0034*jk* instructions for the processors. When a processor executes an 0034*jk* instruction and the ECI (Enable Cache Invalidate) flag is set, it invalidates the cache for that CPU and sends a test and set command to the JS, which then passes it to the global logic via the interprocessor JS/JS bus. This command is passed to the global test and set control. The test and set command that is passed to the global JS logic is a 1-word command. It uses the JS/JS bus for 1 CP, after which the JS/JS bus is available for commands from other PV ASICs (SV1 series) or PVC ASICs (SV1ex series).

The global test and set logic contains the following information about each processor:

- Whether it is doing a 0034*jk* instruction
- Cluster number
- Semaphore register number

When the test and set command gets to the test and set logic, the logic checks the semaphore register to determine whether it is set. If the register is not set, the logic sets it and returns a completion command to the originating processor. If it is set, it enters a waiting-on-semaphore state and notifies the originating processor.

Whenever a 0036*jk* (clear SM) or 073*i*02 (load SM) instruction is executed, the status module for each CPU checks to determine whether the semaphore it is waiting on was cleared. If so, it requests to set it. One of the requesting CPUs receives access, sets the SM, and notifies its requesting CPU that it has completed.

Simultaneously, the processor that originated the 0034*jk* is holding issue. It holds issue until it receives a response from the JS. If the JS returns a completion command to the processor, then the 0034*jk* issues and execution continues. If the JS returns a deadlock command, the P register decrements and the processor exchanges with a deadlock flag set.

Deadlock

A deadlock condition occurs when all CPUs that are assigned to a cluster are holding issue on a test and set (0034*jk*) instruction; that is, each CPU within the cluster is waiting for another CPU to clear an SM register. When this condition occurs, no further execution is possible in any of the CPUs assigned to the cluster; each CPU waits for another CPU to clear an SM register.

Deadlock occurs in two situations:

- All CPUs in the same cluster hold issue on a test and set instruction.
- A single CPU holds issue on a test and set instruction and there are no other CPUs in the same cluster. This situation can occur in either of two ways:
  - Only one CPU is assigned to a particular cluster, and that CPU issues a test and set instruction for an SM register that is currently set.
  - Several CPUs are assigned to the same cluster, one of which is holding on a test and set instruction. Then, all the other CPUs exchange to new programs with different cluster numbers.

To resolve the deadlock condition, a deadlock interrupt occurs. This interrupt sets the deadlock (DL) flag in the current exchange package of each CPU that is assigned to the cluster in which the deadlock has occurred. This causes each affected CPU that is not in monitor mode to perform an exchange sequence. A deadlock chain passes the WS bit and CLN of each CPU to all of the CPU status modules.

Interprocessor Interrupts

Interprocessor interrupts allow a CPU to interrupt program execution in other CPUs. Table 14 shows the two instructions that involve interprocessor interrupts. These instructions can be executed only by a CPU in monitor mode.

Table 14. Interprocessor Interrupt Instructions

Machine Instruction	CAL Syntax	Description
0041*j*1[a]	SIPI A*j*	Set interprocessor interrupt request to CPU (A*j*)
001402[a]	CIPI	Clear interprocessor interrupt request

[a] These instructions are privileged to monitor mode.

When a CPU executes instruction 0014*j*1, the interrupt-from-internal CPU (ICP) request flag is set in the CPU that is designated by the contents of register A*j*. If this CPU is not in monitor mode, it begins an exchange sequence. The program that begins running as the result of the exchange sequence should be in monitor mode and should execute instruction 001402 to

clear the ICP flag. If this instruction is not executed, the ICP flag initiates another exchange sequence when the monitor mode program exits to a nonmonitor mode program.

There is one special case involving the 0014*j*1 instruction. If instruction 0014*j*1 is executed with the contents of register A*j* equal to the number of the CPU that is executing the instruction (that is, if a CPU tries to interrupt itself), the instruction performs no operation. The interprocessor interrupt logic is part of the global logic on the JS. It routes interprocessor interrupts to the correct CPU. When a processor issues a 0014*j*1 (SIPI) instruction, it sends a SIPI command to the local JS, which then passes it to the other JS ASICs via the interprocessor bus. The JS/JS bus interface logic in the global logic routes the SIPI to the correct processor.

Real-time Clock

Each JS contains a copy of the global real-time clock (RTC). When the RTC is written, all global copies are updated at the same time. Each individual JS is then responsible for updating the copies of the RTC that are local to each PV ASIC (SV1 series) or PVC ASIC (SV1ex series). This is done by requesting access to the PV/JS bus and sending a copy to the PV. When a processor reads the RTC, it reads from the local copy on the PV ASIC (SV1 series) or PVC ASIC (SV1ex series). The PV does not check whether a change to the RTC is pending. If one processor is changing the RTC at about the same time another is reading the RTC, the processor that is reading the RTC may not get the new value.

When the global RTC unit receives a load RTC command, it causes all other JS activity to halt. When all has gone quiet, the new RTC value is transferred to all PV ASICs (SV1 series) or PVC ASICs (SV1ex series at the same time.

Table 15 shows the two instructions that write data to and read data from the RTC.

Table 15. RTC Instructions

Machine Instruction	CAL Syntax	Description
0014*j*0[a]	RT S*j*	Transmit (S*j*) to RTC[b]
072*i*00	S*i* RT	Transmit (RTC) to S*i*

[a] This instruction is privileged to monitor mode.

[b] RTC bits 0–63 are forced to 0's if j=0.

Instruction 0014*j*0 can be issued only by a CPU in monitor mode; the CPU that issues this instruction updates the value of the local clocks on all other CPUs. Two or more CPUs should not execute this instruction simultaneously because there is no hardware to detect this condition, and unpredictable results can occur. The programmer must avoid this situation. Instruction 072*i*00 may be issued simultaneously by any number of CPUs.

Note: On the SV1 CPU, the real-time clock increments at the system clock rate, not the CPU clock rate (three times the system clock rate). Therefore, on an SV1 CPU, two successive 072*i*00 instructions that issue during the same system clock period will return the same value.

The RTC is normally used to determine the running time of a program or a segment of program code. The following example shows an instruction sequence that is used to determine the running time of a program.

Step	Machine Instruction	CAL	Comment
1	072100	S1 RT	Load S1 with starting time.
2	-	-	Insert code to be timed here. Code must not use S1.
3	072200	S2 RT	Load S2 with ending time.
4	061121	S1 S2-S1	Load S1 with difference between ending and starting time.

At the end of this sequence, if no interrupts occur, register S1 equals 1 plus the number of system CPs required to execute Step 2.

Exchange Mechanism

Exchange sequences, fetch sequences, and issue sequences are closely related. When an initial deadstart program or a new program runs, an exchange sequence occurs. An exchange sequence brings several important parameters of the program into some of the central processing unit's (CPU's) operating registers. A fetch sequence begins immediately after the exchange sequence. A fetch sequence transfers a block of instructions from memory to an instruction buffer. The issue sequence then selects the instruction that is indicated by the program address (P) register, decodes it, and passes it on to be executed.

As the instruction executes, the P register increments, which causes new instructions to move through the issue sequence. When a desired address is not currently in an instruction buffer, another fetch sequence occurs. This overall process continues until the program either terminates or is interrupted, at which time another exchange sequence occurs and the entire process starts over.

This section describes the exchange mechanism, the instruction fetch sequence, and the instruction issue sequence, which are unique to each CPU. It also briefly describes the programmable clock, the status register, and the performance monitor.

Each CPU uses an exchange mechanism to switch instruction execution from program to program. This exchange mechanism uses blocks of program parameters called exchange packages and a CPU operation called an exchange sequence.

The following subsections explain the exchange package and the exchange sequence in more detail.

Exchange Package

The exchange package (refer back to Figure 6) is a 16-word block of data in memory that is associated with a particular computer program. The exchange package contains the basic parameters that provide continuity when a program stops one section of the program and starts the next.

The exchange package holds the contents of the address (A) and scalar (S) registers. The contents of the intermediate address (B), intermediate scalar (T), vector (V), vector mask (VM), shared B (SB), shared T (ST), and semaphore (SM) registers are not saved in the exchange package. Data in these registers must be stored and replaced as required by the program that is supervising the object program or by any program that needs this data.

Refer again to Figure 6 for the format of the exchange package. Table 16 lists the exchange package assignments. The following subsections define and explain the fields of the exchange package.

Note: For software, the exchange package bits are numbered from left to right with the high-order bit (most significant bit) assigned to bit position 0. For hardware, the exchange package bits are numbered from right to left with the low-order bit (least significant bit) assigned to bit position 0. This document uses the hardware convention.

Processor Number Field

The contents of the processor number (PN) field indicate which CPU performed the exchange sequence. This value is inserted into the exchange package from the configuration file bits (13 through 9) that are located on the PV ASIC (SV1 series) or PVC ASIC (SV1ex series).

P Register Field

The program address (P) register contents are stored in the program address register field of the exchange package. The instruction that is stored at this location is the first instruction to issue when the program that corresponds to the exchange package begins execution.

Memory Error Data Fields

Memory error data, which consists of six fields of information, is valid only if one of two conditions is met. The first condition is that the interrupt-on-correctable memory (ICM) bit is set in the mode (M) register and a correctable memory error is detected. The second condition is that the interrupt-on-uncorrectable memory (IUM) bit is set in the M register and an uncorrectable memory error is detected. The following subsections describe the memory error data fields.

Syndrome Field

The 8-bit syndrome field defines the syndrome code that is generated by SECDED logic for a memory read operation or an I/O channel transfer. The syndrome code will be all zeroes for a cache register parity error.

Memory Error Address

If an error occurs during a memory read operation, the number of the bank that is being read when the error occurred is stored in the 10-bit read address bank field. The bank number is contained in bits 0 through 9 of the read address.

The memory error address contains address bits 0 through 7, and the memory error address (continued) contains overflow bits 8 and 9. The address field identifies the memory section for a cache register parity error per CPU on a processor module as shown previously in Figure 6.

Read Error Type Field

The 2-bit read error type field determines the type of memory or I/O error that occurred; bit 0 sets if the error is uncorrectable, and bit 1 sets if the error is correctable.

Read Mode Field

The read mode bits are used with the port bits to determine what kind of read operation was in progress when the memory error occurred. Table 16 shows the read mode and port value translations. Although the ports are listed as A and B and D, only port D is an actual physical port. Ports A and B are used as identification codes because they do not physically exist external to the PV ASIC (SV1 series) or PVC ASIC (SV1ex series).

Memory Register Fields

Four registers test the area limits for memory references: the data base address (DBA) register, the data limit address (DLA) register, the instruction base address (IBA) register, and the instruction limit address (ILA) register.

Data Base Address Register Field

The DBA register holds the base address of the user’s data area (the location in memory where a program’s data area begins). Each time an instruction in the program makes a memory reference, the memory address that is generated by the instruction is added to the contents of the DBA register to form the absolute memory address. An address less than the address stored in the DBA generates an out-of-range condition.

Bits 10 through 31 of the DBA register are stored in the DBA field of the exchange package. Bits 0 through 9 of the register are always 0 and therefore do not need to be stored.

Table 16. Exchange Package Read Mode and Port Translations

Port Value	Mode Value	Type of Transfer when Error Occurred	Explanation
0	2	V	Error during vector read from memory
4 = A	0	EX	Error while reading the exchange package
4 = A	1	B	Error during block memory read to the B registers
4 = A	3	A, S	Error during a memory read to the A or S register
2 = B	0	Fetch	Error during memory read for fetch instructions
2 = B	1	T	Error during block memory read to the T registers
6 = A, B	3	CRPE	Cache register parity error during a read from one of two CA ASICs per CPU per processor module
1 = D	0	Y1 or HIPPI	SECDED error in CI ASIC during a memory read for channel (n) output
1 = D	1	Y1 or HIPPI	SECDED error in CI ASIC during a memory read for channel (n+3) output
1 = D	2	Y1 or HIPPI	SECDED error in CI ASIC during a memory read for channel (n+5) output
1 = D	3	Y1 or HIPPI	SECDED error in CI ASIC during a memory read for channel (n+7) output

n = Processor board number

Section 0, 2, 5, and 7 errors are on CA 0, 2, 4, or 6 per CPU 0, 1, 2, or 3; and section 1, 3, 4, and 6 errors are on CA 1, 3, 5, or 7 per CPU 0, 1, 2, or 3 respectively.

Data Limit Address Register Field

The DLA register holds the limit address of the user's data area, which is used to determine the highest absolute memory address that the program can use for reading or writing data. Each time an instruction makes a memory reference, the absolute memory address that the instruction generates is compared to the address in the DLA register. If the absolute memory address is less than the DLA register, the reference proceeds. If the absolute memory address is equal to or greater than the DLA register, an out-of-range condition exists. If the interrupt-on-operand range error (IOR) flag in the mode register is set, the out-of-range condition sets the operand range error (ORE) flag in the flag register, which initiates an exchange sequence.

A memory read reference that is beyond the limits of the assigned area issues and completes, but a zero value is transferred from memory. A memory write reference that is beyond the assigned area issues, but no write operation occurs.

Bits 10 through 31 of the DLA register are stored in the DLA field of the exchange package. Bits 0 through 9 of the register are always 0 and therefore do not need to be stored. The highest absolute memory address that a program can reference for data is defined by [(DLA) x 2 exp10] - 1 memory range.

Instruction Base Address Register Field

The IBA register holds the base address of the user's instruction area (the location in memory where a program's instruction area begins). During an instruction fetch sequence, an absolute memory address is formed by adding the relative address that is generated by the fetch control logic to the contents of the IBA register.

The absolute memory address for an instruction fetch is formed by adding the IBA register to the higher-order 22 bits of the P register. Bits 10 through 31 of the IBA register are stored in the IBA field of the exchange package. Bits 0 through 9 of the register are always 0 and therefore do not need to be stored.

Instruction Limit Address Register Field

The ILA register holds the limit address of the user's instruction area, which is used to determine the highest absolute memory address that can be accessed during an instruction fetch sequence.

If the absolute memory address used in an instruction fetch sequence is not between the area of addresses contained within the IBA and ILA registers of the active exchange package, the CPU generates a program range error interrupt. Bits 10 through 31 of the ILA register are stored in the ILA field of the exchange package. Bits 0 through 9 of the register are always 0 and therefore do not need to be stored. The highest absolute instruction address of a program is defined by [(ILA) x 2 exp10]-1 memory range.

Exchange Address Register Field

The 8-bit exchange address (XA) register specifies the first word address of a 16-word exchange package that is loaded by an exchange sequence. The XA register field contains the higher-order 8 bits of a 12-bit XA register that specifies the absolute memory address. The low-order bits of the area are always 0; an exchange package must begin on a 16-word boundary. The 12-bit limit requires that the absolute memory address be in the lower 10000 (octal) words of memory. (The DBA is not added to the XA register.) The exchange sequence exchanges the contents of the registers with the contents of the exchange package at the beginning XA register in memory.

Vector Length Register Field

The 7-bit vector length (VL) register specifies the length of all vector operations that are performed by vector instructions and the number of elements that are held in the V registers. The value in the VL register can be changed during program execution by using the 00200*k* instruction.

Cluster Number Register Field

The cluster number (CLN) register determines the CPU's cluster. There are 33 (decimal) clusters of SB, ST, and SM registers (for the largest system). A value of 1 (octal) through 41 (octal) in the CLN register determines which cluster the CPU can access. If the content of the CLN register is 0, then the CPU does not have access to any SB, ST, or SM register. The contents of the CLN register in all CPUs are also used to determine a deadlock interrupt condition. The formula for determining the number of the cluster for the various SV1 series configurations is the number of CPUs + 1.

Vector Not Used Field

The state of the vector not used (VNU) bit in the exchange package indicates whether instruction 077 or instructions 140 through 176 were issued during the execution intervals. The VNU bit is set if none of the instructions issued; it is not set if one or more of the instructions issued.

Waiting for Semaphore Field

The waiting for semaphore (WS) bit in the exchange package is set to indicate that the CPU exchanged when the test and set instruction was holding in the current instruction parcel (CIP) register.

Flag Register Field

The flag (F) register contains 11 flags for the active program. The setting of a flag can initiate an exchange sequence. The monitor program analyzes the flag to identify the cause of an exchange sequence. Before the monitor program exchanges back to the program, it must clear the flags in the F register area of the exchange package. If any flag remains set, another exchange occurs immediately. The contents of the F register are stored in memory with the rest of the exchange package.

Some of the F register flags are disabled when a program is running in monitor mode or interrupt in monitor mode (refer to the following “Mode Register Field” subsection in this section for more information on the MM and IMM bits). If a flag is disabled and the conditions for setting the flag are present, the flag remains clear and no exchange sequence is initiated.

The F register contains the following flags:

Bit Position SW/HW	Flag Description
8/55	(not used)
9/54	Interrupt-from-internal CPU (ICP) - is set when another CPU issues instruction 0014*j*1.
10/53	Deadlock (DL) - is set when all CPUs in a common cluster are holding issue on a test and set instruction.
11/52	Programmable clock interrupt (PCI) - is set when the interrupt countdown counter in the programmable clock equals 0.
12/51	MCU interrupt (MCU) - is set when the CPU Interrupt signal is active on Deadstart of a CPU. This signal is part of I/O channel 20.
13/50	Floating-point error (FPE) - is set when a floating-point range error occurs in any of the floating-point functional units and when the interrupt-on-floating-point error (IFP) bit in the M register is set.
14/49	Operand range error (ORE) - is set when a data reference is made outside the boundaries of the DBA and DLA registers and when the interrupt-on-operand range error bit in the M register is set.
15/48	Program range error (PRE) - is set when an instruction fetch is made outside the boundaries of the IBA or ILA registers.
16/47	Memory error (ME) - is set when a correctable or uncorrectable memory error occurs and the corresponding interrupt-on-correctable memory error (ICM) bit or interrupt-on-uncorrectable memory error (IUM) bit in the M register is set.
17/46	I/O interrupt (IOI) - is set when an I/O interrupt occurs after an I/O channel completes a transfer.
18/45	Error exit (EEX) - is set by an error exit (000) instruction if the program is not in monitor mode or the interrupt-in-monitor mode (IMM) is set.
19/44	Normal exit (NEX) - is set by a normal exit (004) instruction if the program is not in monitor mode.

Mode Register Field

The mode (M) register contains 10 user-selectable bits for the active program; it also contains 1 status bit: floating-point error status. The M register contains the following bits:

Bit Position SW/HW	Mode or Status Description
20/43	Enable second vector logical (ESVL) - when set, this bit enables the second vector logical functional unit. Instructions 140 through 145 use the second vector logical functional unit if it is enabled and not reserved by another instruction.
21/42	Bit matrix loaded (BML) - is set by the load bit matrix (1740*j*4) instruction to indicate to the operating system that the BMM register contains valid data. The 002210 instruction clears this bit.
22/41	Floating-point error status (FPS) - when set, this bit indicates that a floating-point error occurred, regardless of the state of the floating-point error flag.
23/40	Bidirectional memory (BDM) - when set, this bit indicates that block read and write operations can operate concurrently. The BDM bit can be set or cleared during a program by using instructions 002600 (enable bidirectional memory transfers) and 002500 (disable bidirectional memory transfers).
24/39	Interrupt-on-operand range error (IOR) - when set, this bit enables interrupt-on-operand address range errors. The IOR bit can be set or cleared during the execution interval of a program by using instructions 002300 (enable interrupt-on-operand range error) and 002400 (disable interrupt-on-operand range error).
25/38	Interrupt-on-floating-point error (IFP) - when set, this bit enables interrupts on floating-point errors. The IFP bit can be set or cleared during the execution interval of a program by using instructions 002100 (enable interrupt-on-floating-point error) and 002200 (disable interrupt-on-floating-point error).
26/37	Interrupt-on-uncorrectable memory error (IUM) - when set, this bit enables interrupts on uncorrectable memory data errors and/or cache register parity errors.
27/36	Interrupt-on-correctable memory error (ICM) - when set, this bit enables interrupts on correctable memory data errors and/or cache register parity errors.
28/35	This bit position is not used.

Bit Position SW/HW	Mode or Status Description
29/34	Selected for external interrupts (SEI) - when set, this CPU is preferred for I/O interrupts. When an I/O channel completes a transfer, the channel can interrupt only one CPU. The CPU with this bit set gets the interrupt. Refer to “I/O Interrupts” for more information on I/O interrupts.
30/33	Interrupt in monitor mode (IMM) - this bit is used only when the MM bit is set; this bit then enables the DL, FPE, ORE, and PRE interrupts along with interrupts allowed when MM is also set.
31/32	Monitor mode (MM) - when set, this bit allows access to privileged monitor mode instructions and inhibits all interrupts except ME, NEX, and EEX.

The FPS bit indicates the state of the CPU at the time of the exchange sequence. The remaining bits are not altered during the execution interval for the exchange package and can be altered only when the exchange package is inactive in memory.

The FPS, BML, BDM, IOR, and IFP bits can be read to an S register with instruction 073*i*01.

Cache Enable

For the SV1 series machines, each PV ASIC includes a 32-Kword or 256-Kbyte cache on two CA ASICs. The cache is enabled for data use when the ECD mode bit is set to a 1 and for instruction fetch use when the ECF bit is set to a 1.

For the SV1ex series machines, each PVC ASIC includes a 256-Kbyte, 4-way, set associative cache array.

Processor Type

The processor type is identified using bits 63 and 62 of word 7 of the exchange package:

00 - Cray J90
01 - Cray J90se
10 - Cray SV1
11 - Cray SV1e

Performance Monitor

The performance monitor tracks groups of hardware-related events. These results can be used to indicate the relative performance of a program. The performance monitor contains eight performance counters that track four groups of hardware-related events.

The SV1 series system has a large 32-Kword cache that is used for both data and instruction caching. Cache is enabled by 2 bits in word 7 of the exchange package: ECD and ECF. Effective cache hit counting is simplified by the addition of 4 bits in word 7 of the exchange package that selectively disables cache hit counting. These bits are:

- Disable scalar cache hits - scalar A or S data (DSCH)
- Disable B or T cache hits - B or T data (DBCH)
- Disable fetch cache hits - instruction operands (DFCH)
- Disable vector cache hits - vector data (DVCH)

For more information, refer to “Performance Monitor” on page 99.

Cache Invalidate

The Enable Cache Invalidate (ECI) bit in word 7 of the exchange package is used by the 0034*jk* instruction to invalidate cache.

A Register Fields

The current contents of all A registers are stored in bits 0 through 31 of words 0 through 7 during an exchange sequence.

S Register Fields

The current contents of all S registers are stored in bits 0 through 63 of words 8 through 15 during an exchange sequence.

Exchange Sequence

The exchange sequence moves an inactive exchange package from memory into the operating registers. Simultaneously, the exchange sequence moves the active exchange package from the operating registers back into memory. This swapping operation occurs in a fixed sequence when all computational activity associated with the active exchange package stops.

The exchange sequence involves 16 memory read references (sent in 4 CPs) and 16 memory write references (sent in 8 CPs). A single 16-word block of memory is the source of the inactive exchange package and the destination of the active exchange package. Word 0 of the active exchange package is swapped with word 0 of the inactive exchange package. The location of this block is specified by the contents of the XA register and is a part of the active exchange package.

Exchange Sequence Timing

The following subsections define the hold conditions, execution time, and special case conditions for an exchange sequence.

Hold Conditions

The following conditions can delay the start of an exchange sequence:

- Incomplete memory references
- Any active A, S, or V registers within the CPU

Execution Time

Any exchange takes a minimum of approximately 220 CPs: approximately 108 CPs for the exchange sequence and approximately 112 CPs for the fetch operation. (This time is longer when memory conflicts occur.) Memory conflicts are possible during an exchange sequence and a fetch operation.

Special Case Conditions

If the test and set instruction is holding in the current instruction parcel (CIP) register, both the CIP and next instruction parcel (NIP) registers are cleared. The exchange occurs with the WS flag set and the P register pointing to the address of the test and set instruction.

Initiating an Exchange Sequence

The exchange sequence can be initiated by a deadstart sequence, a program exit, or an interrupt. The following subsections describe conditions that cause an exchange sequence and the results of the exchange.

Deadstart Sequence

The deadstart sequence starts a program in the mainframe after a power-off/power-on operation or whenever the operating system is re-initialized in the mainframe. All control latches, words in memory, and the contents of all registers are invalid after a power-off/power-on operation. During the power-on sequence, the reset logic is asserted automatically to all flip-flops (FF options) in the system. JTAG control logic loads all of the configuration registers using JTAG scan circuitry and then enables the Reset and Stopclk signals. Next, JTAG control proceeds with the start-up sequence to synchronize all ASICs and leave the system idle. Memory can now be loaded through I/O channel 20 (octal).

The external device then loads an initial exchange package and monitor program. Because a deadstart sequence forces the contents of the XA register to 0, this initial exchange package must be located at memory address 0.

Through JTAG, the processor is chosen to do the deadstart exchange by forcing an interrupt on that CPU. These actions cause an exchange sequence that issues the exchange package at memory address 0. This exchange package then moves into the operating registers and initiates a program that uses these parameters.

The exchange package that was originally used as the deadstart sequence is swapped back into memory address 0 and is indeterminate because of the deadstart operation. New data is entered into this exchange package in preparation for deadstarting subsequent CPUs by an interprocessor interrupt. When instruction 0014*j*1 is issued in the first CPU, the next CPU exchanges to memory address 0. This sequence continues until all CPUs are deadstarted.

Each exchange package resides in an area that was defined during system deadstart. The defined area must be in the lower 4,096 (10000 [octal]) words of memory. The package at memory address 0 is the deadstart monitor’s exchange package. Only the monitor has a defined area so that it can access all of memory, including exchange package areas. This area allows the monitor to define or alter all exchange packages other than its own when it is the active exchange package. Other exchange packages provide for object programs and other monitor tasks and are located outside of the program’s instruction and data areas.

Program Exit Instructions

Two program exit instructions initiate an exchange sequence: error exit (000) and normal exit (004). The two instructions enable a program to request its own termination. A program usually uses the normal exit instruction to

exchange back to the monitor program. The error exit instruction allows termination of an object program if an abnormal condition occurs; the exchange address selected is the same address that is used for a normal exit instruction.

Depending on which instruction issues, either an error exit or normal flag is set in the F register, which forces an interrupt. (Refer to the previous "Flag Register Field" subsection for more information on the flags.) The appropriate flag is set only if the active exchange package is not in monitor mode. The inactive exchange package that is sent during the exchange sequence normally has its monitor mode bit set.

Interrupts

An exchange sequence can also be initiated by setting any of the interrupt flags in the F register (refer to "Flag Register Field" for more information on the flags). Setting one or more flags causes a Request Interrupt signal to initiate an exchange sequence.

Exchange Package Management

Exchange package management dictates that a user program always exchanges back to the monitor that caused the non-monitor program to start execution. This exchange back to the monitor ensures that the program information is always exchanged into its proper exchange package.

For example, a monitor begins an execution interval following a deadstart sequence. No interrupts (except memory) can terminate its execution interval because it is in monitor mode. Before the monitor program exits, the monitor sets the contents of the XA register to point to a user program's exchange package, so that a user program runs next. Then, the monitor sets the contents of the XA register in the user program's exchange package to the appropriate location in the monitor program. The monitor voluntarily exits by issuing a normal exit instruction (004).

The exchange sequence moves the inactive exchange package (in this case, the user program's) from memory into the operating registers and at the same time, moves the active exchange package (in this case, the monitor's) from the operating registers into memory. The contents of the XA register in the user program's exchange package point to the monitor that originally allowed the user program to exchange. When the exchange is complete, the user program begins to run.

If an interrupt occurs while the user program is running, an exchange sequence is initiated. Because the contents of the XA register in the user's program exchange package point to the monitor, the exchange is back to the monitor. (Note that a user program cannot alter the contents of the XA register.)

When the exchange back to the monitor is complete, the monitor determines which interrupt caused the exchange and sets the contents of the XA register to call the proper interrupt-processing program to run. To do this, the monitor sets the XA register to point to the exchange package for the relevant interrupt-processing program. The monitor then clears the interrupt and executes a normal exit (004) instruction, which causes the interrupt-processing program to run. Depending on the operating task, the interrupt-processing program can run in monitor mode or user mode.

Note: There is no interlock between an exchange sequence in a CPU and memory transfers in another CPU; therefore, avoid modifying exchange packages used by other CPUs except under software-controlled situations.

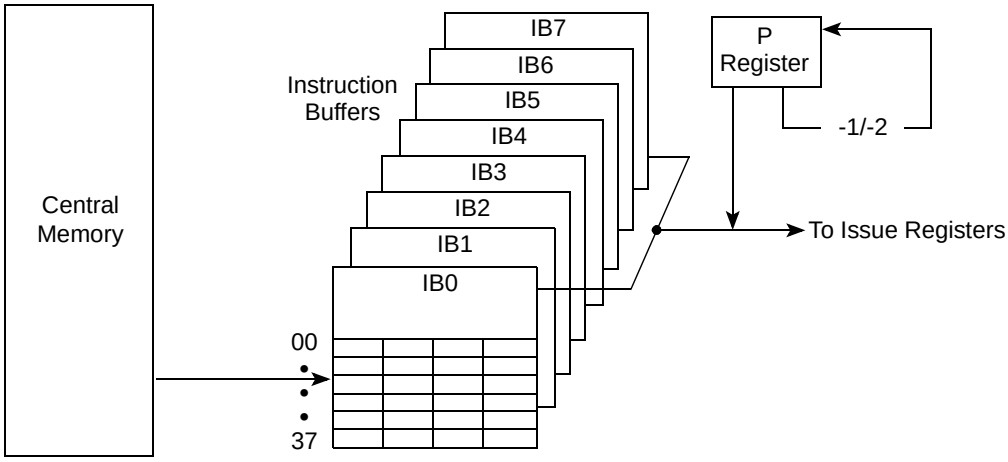
Instruction Fetch Sequence

An instruction fetch sequence retrieves program code from memory and places it in an instruction buffer. The program code is held in the instruction buffer before it is delivered to the instruction issue registers. The following subsections describe the hardware that is associated with the instruction fetch sequence and define the fetch operation.

Instruction Fetch Hardware

A Cray SV1 series system uses the P register to initiate an instruction fetch sequence; it uses eight instruction buffers to store the instructions that are retrieved from central memory. Figure 10 shows the P register and instruction buffers.

Figure 10. Instruction Fetch Block Diagram



Note: If the ECF bit is set cache is enabled.

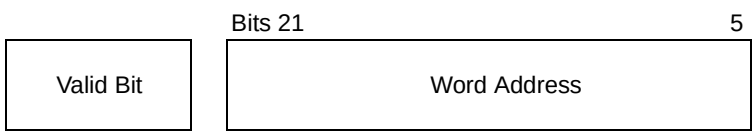
Instruction Buffers

Each of the eight instruction buffers (IB0 through IB7) holds 40 (octal) (00 through 37 octal) words. Each word contains four 16-bit instruction parcels; therefore, each buffer holds 128 parcels. Instruction parcels are held in the buffers before they are delivered to the issue registers.

The first instruction parcel in a buffer always has a word address that is a multiple of 40 (octal). This word address allows the entire area of addresses for instructions in a buffer to be defined by the high-order 17 bits of the P register.

Each instruction buffer has an associated instruction buffer address register (IBAR). The IBAR contains the high-order 17-bit address and an IBAR valid bit for the instruction in that buffer. When set, the IBAR valid bit indicates that the buffer contains valid data. During an exchange sequence, the IBAR valid bit is cleared to invalidate the previous program's instructions and to force the CPU to fetch new instructions. Once the fetch operation begins, the appropriate IBAR is loaded with the upper 17 bits of the P register, and its valid bit is set. Figure 11 shows the IBAR.

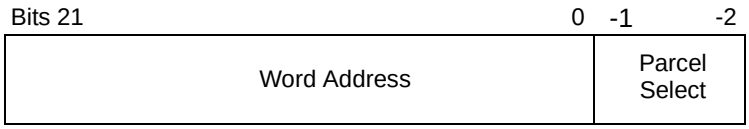
Figure 11. IBAR



Program Address Register

The 24-bit P register indicates the next parcel of program code to enter the next instruction parcel (NIP) register. As shown in Figure 12, the high-order 22 bits of the P register indicate the word address of the program code in memory relative to the base address. The low-order 2 bits indicate the parcel within the word. Because 22 bits specify the word address, the maximum program length is approximately 4 Mwords with approximately 16 million parcels.

Figure 12. P Register

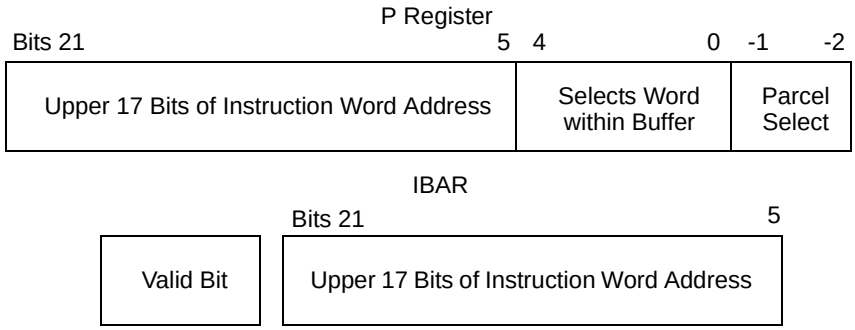


Under normal circumstances, the P register increments sequentially as instructions issue. For 1- and 2-parcel instructions, the P register increments by 1; for 3-parcel instructions, the P register increments by 2. These increments allow both 2- and 3-parcel instructions to issue in 2 CPU CPs. Branch instructions can load the P register with any value. When the program exchanges out, the saved P register contains the address of the instruction immediately after the last instruction that executed.

Instruction Fetch Operation

An instruction fetch operation refers to the series of steps that move program code from memory to an instruction buffer. Refer to Figure 13 for an illustration of the P register and IBAR address formats.

Figure 13. P Register and IBAR Address Formats



The P register always contains the parcel address of the next instruction to be decoded. The fetch operation is based on a comparison check of the P register against the values held in the eight IBARs; this comparison is done each clock

period (CP). If the content of one of the IBARs is equal to the upper 17 bits in the P register and the IBAR valid bit is set, an in-buffer (or coincidence) condition exists.

If the high-order 17 bits of the P register do not match any IBARs, or the valid bit is not set, an out-of-buffer (or no-coincidence) condition exists and the instruction fetch sequence starts.

Once the instruction buffers are loaded, or if the comparison between the P and IBAR registers produced a coincidence condition, the proper instruction parcel is selected from the instruction buffer. The instruction parcel is sent to the NIP register and then to the CIP register, from which the instruction issues. Refer to Figure 14. Instruction issue is explained later in this section.

The instruction fetch sequence sends 32 requests in 14 CPU CPs to transfer 32 words (128 parcels) from memory into the instruction buffer (refer to “Port Utilization” for more information on memory ports). Two words are transferred during each CPU CP.

The buffers are filled circularly: 128 parcels fill the first instruction buffer; then another fetch sequence occurs to fill the second instruction buffer with 128 parcels, and so on, until all eight buffers are filled. If the program code exceeds 1,024 parcels, the ninth fetch invalidates the first instruction buffer and reloads it with the new instructions.

The loading of the instruction buffer occurs in whatever order the instruction words arrive from memory (or cache).

Although optimizing the length of code segments for instruction buffers is not a prime consideration when programming a CPU, the number and size of the buffers and the capability for forward and backward branching can be used to minimize fetches. Large loops that contain up to 1,024 consecutive instruction parcels can be maintained in the eight buffers. Another method to minimize fetches is to have a main program sequence in one or two of the buffers make repeated calls to short subroutines in the other buffers. The program and subroutines remain undisturbed in the buffers as long as no out-of-buffer condition or exchange causes reloading of a buffer. The SV1 series CPU includes the 32-Kword cache that is available for both data and instruction use. Additional instructions may be available to the program for quick access from this cache.

Forward and backward branches are possible within buffers. Branching does not cause reloading of an instruction buffer if the address to which the instruction branches is within one of the buffers. Multiple copies of instruction parcels cannot occur in the instruction buffers.

Because instructions are held in instruction buffers and in cache, before issue and until the buffer is reloaded, self-modifying code should not be used. Self-modifying code may be impossible because of independent data and instruction memory protection. As long as the address of the unmodified instruction is in an instruction buffer, the modified instruction in memory is not loaded into an instruction buffer. Cache must be invalidated when self-modifying code is used.

Instruction Fetch Timing

During an instruction fetch sequence, instructions are moved from memory, or cache, to an instruction buffer at the rate of 2 words per CPU CP. It takes about 109 CPs for the first word to arrive at the instruction buffer and an additional 3 CPs for the first instruction to arrive in the current instruction parcel (CIP) register. Instruction issue can run concurrently with the fetch operation as long as the required instruction parcel is in the instruction buffer. If no memory conflicts occur, the instruction buffer is filled in about 124 CPs (109 CPs for the first word and 15 CPs for remaining words). Memory conflicts can lengthen the fetch sequence.

Instruction Issue

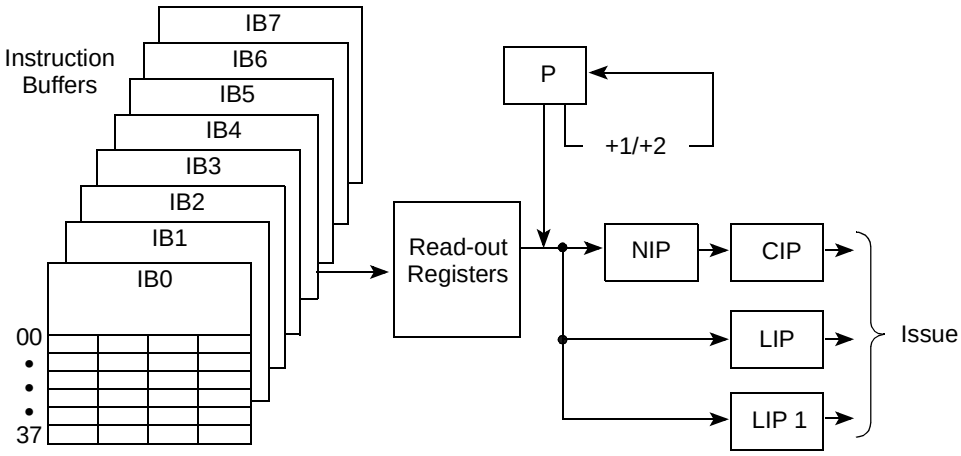
An instruction issue sequence is the series of steps that are performed to move an instruction from an instruction buffer through the issue registers and into execution.

Instruction Issue Hardware

The SV1 series system uses four registers to issue instructions. Figure 14 shows the registers and buffers, and the general flow of the instruction parcels through them. CPU instructions are 1-, 2-, or 3-parcel instructions; refer to "Instruction Formats" for information on instruction parcels.

Vector instructions are locally issued and dispatched to the vector unit for final issue. The vector unit can queue five such vector instructions, which are then issued in the order received. The vector issue unit checks for vector register and functional unit conflicts before issuing these instructions. Vector register and functional unit reservations are made by the vector issue unit upon final issue of the instructions.

Figure 14. Instruction Issue Block Diagram – General Flow



Instruction Buffers

The instruction buffers hold the program code after it is retrieved from memory and before it is passed to the issue registers. The instruction buffers have two associated read-out registers to streamline the flow of instructions from the buffers to the next instruction parcel (NIP) register. Even-numbered words are loaded into the even read-out register, while odd-numbered words are loaded into the odd read-out register. Bit 0 of the P register determines which read-out register is used, and bits -1 and -2 of the P register select the parcel to be sent to the NIP register.

Program Address Register

The 24-bit P register indicates the next parcel of program code to enter the NIP register. The high-order 22 bits of the P register indicate the word address for the program code in memory relative to the base address. The low-order 2 bits indicate the parcel within the word. Under normal circumstances, the P register increments sequentially as instructions issue. For 1- and 2-parcel instructions, the P register increments by 1; for 3-parcel instructions, it increments by 2. This allows both 2- and 3-parcel instructions to issue in 2 CPs. Branch instructions and exchange sequences can load the P register with any value.

Next Instruction Parcel Register

The 16-bit NIP register receives an instruction parcel from one of the instruction buffer read-out registers. While the parcel of program code is held in the NIP register, it is decoded to determine whether the instruction is a 1-, 2-, or 3-parcel instruction. The parcel is then passed to the CIP register.

The NIP register cannot be master cleared. An undetermined instruction can issue during the master clear sequence, before an interrupt condition blocks data entry into the NIP register.

Current Instruction Parcel Register

The 16-bit CIP register receives the parcel of program code from the NIP register and holds the instruction until it issues. Issue of an instruction that is held in the CIP register can be delayed until conflicting operations are completed (refer to "Reservations and Hold Issue Conditions").

The issue control hardware associated with the CIP register is master cleared; the register itself is not. An undetermined instruction can issue during the master clear sequence.

Lower Instruction Parcel and Lower Instruction Parcel 1 Registers

The 16-bit lower instruction parcel (LIP) register holds the second parcel of a 2-parcel instruction (the first parcel of this instruction is always held in the CIP register). The 16-bit LIP1 register holds the third parcel of a 3-parcel instruction (again, the first parcel is held in the CIP register, and the second parcel of this instruction is held in the LIP register).

Instruction Issue Operation

Control logic associated with the NIP register determines whether the instruction is a 1-, 2-, or 3-parcel instruction and steers subsequent parcels to the correct register. The general sequences for the three types of instructions are described in the following paragraphs; specific examples of 1-, 2-, and 3-parcel instructions are provided on the following pages.

For 1-parcel instructions, the P register sends the instruction parcel to the NIP register. From the NIP register, the instruction moves to the CIP register. If there are no conflicts, the instruction executes.

For a 2-parcel instruction, the P register sends the first parcel to the NIP register. Then the first parcel is sent to the CIP register, while the second parcel goes directly to the LIP register. When the two registers are properly loaded with the correct parcels and there are no conflicts, the first parcel issues from the CIP register and the second parcel issues from the LIP register at the same time. When the parcels of the 2-parcel instruction move from the CIP and LIP registers to execution, the NIP register sends a blank parcel to the CIP register. The control logic decodes this blank parcel as a no-operation instruction when it issues from the CIP register. While this blank parcel is loaded into the CIP

register, a new parcel is loaded into the NIP register, and the control logic determines whether the instruction is a multiparcel instruction. During this sequence, a delay can occur if the new instruction is in a different buffer than the previous instruction or if a fetch operation is required.

For a 3-parcel instruction, the P register sends the first parcel to the NIP register. Then the first parcel is sent to the CIP register, while the second parcel goes directly to the LIP register and the third parcel goes directly to the LIP1 register. When the three registers are properly loaded with the correct parcels and there are no conflicts, the first parcel issues from the CIP register, the second parcel issues from the LIP register, and the third parcel issues from the LIP1 register at the same time. When the parcels of the 3-parcel instruction move from the CIP and LIP registers to execution, the NIP register sends a blank parcel to the CIP register. The control logic decodes this blank parcel as a no-operation instruction when it issues from the CIP register. While this blank parcel is loaded into the CIP register, a new parcel is loaded into the NIP register and the control logic determines whether it is a multiparcel instruction. Delays can occur if the new instruction is in a different buffer than the previous instruction or if a fetch operation is required.

Figure 15 through Figure 24 and the following paragraphs show the steps that occur as 1-, 2-, and 3-parcel instructions are steered in sequence through the issue registers. The sequence assumes a 1-CP delay and is numbered CP*n* through CP*n*+9. An instruction buffer with its two read-out registers, the P register, and the relevant issue registers are shown for each CP.

Figure 15 shows parcels 20-0 through 21-3 being held in an instruction buffer and read-out registers. The P register is pointing to parcel 20-0 as the next parcel to be read into the NIP register.

Figure 15. Instruction Issue Block Diagram – Parcels Held

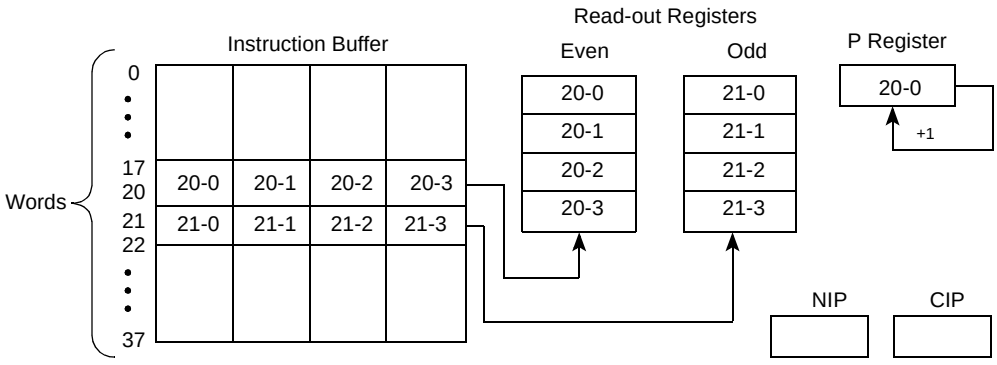
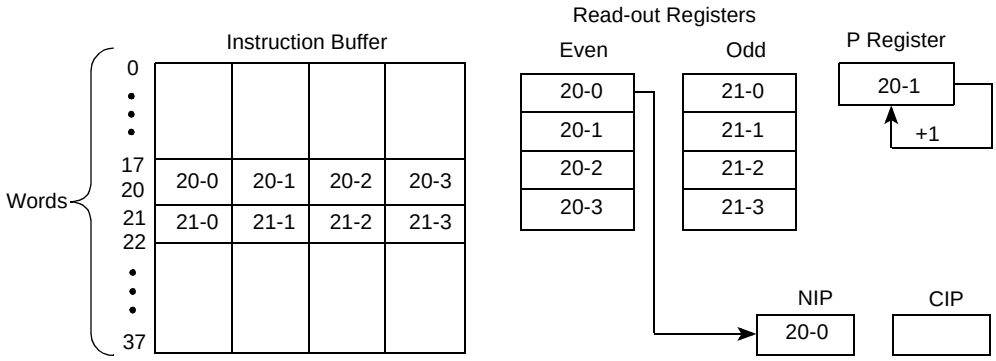


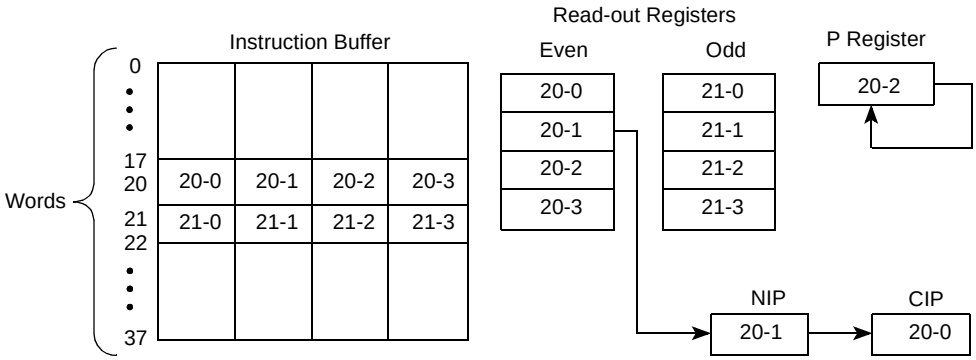
Figure 16 shows parcel 20-0 in the NIP register. The P register incremented by 1 and is pointing to parcel 20-1 to read out as the next parcel. While parcel 20-0 is in the NIP register, the issue hardware determines whether it is a 1-, 2- or 3-parcel instruction.

Figure 16. Instruction Flow through Issue Registers (CPn + 1)



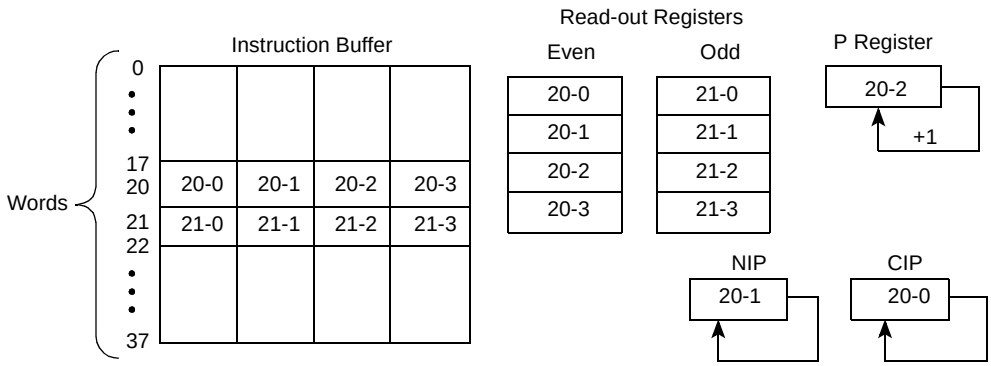
Because parcel 20-0 is a 1-parcel instruction, the logic steers parcel 20-0 into the CIP register and parcel 20-1 into the NIP register. The P register increments by 1 (refer to Figure 17).

Figure 17. Instruction Flow through Issue Registers (CPn + 2)



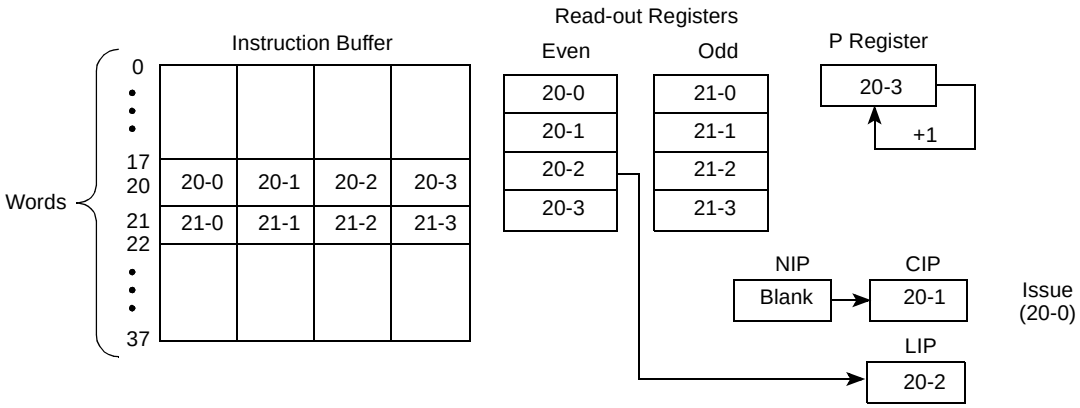
While the parcel in the NIP register is decoded to determine whether it is a 1-, 2-, or 3-parcel instruction, the issue hardware checks for any conflicts that might prevent the instruction in the CIP register from issuing. If there are conflicts, both the CIP and NIP registers hold their parcels, and the P register does not increment (refer to Figure 18).

Figure 18. 1-parcel Instruction Holding 1 CP for Conflict (CPn + 3)



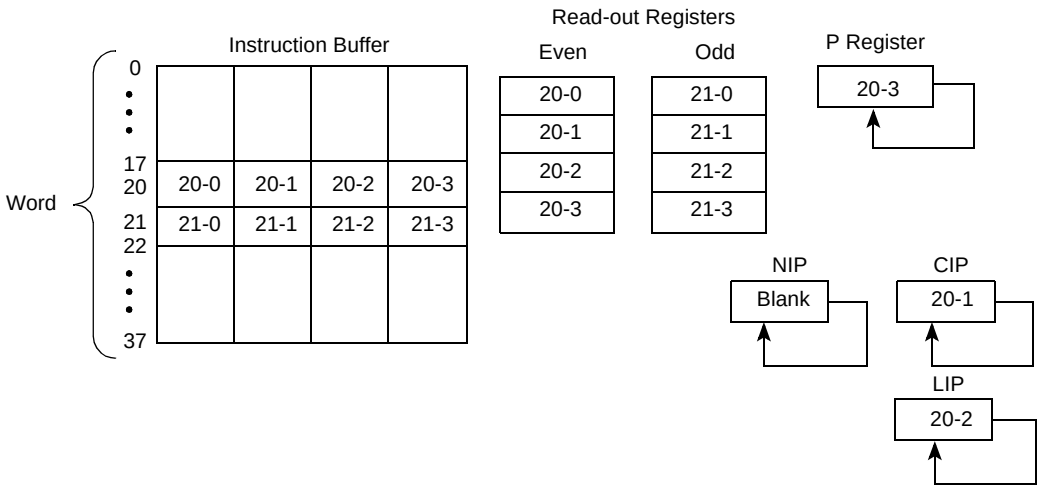
This holding state is maintained until the conflict is resolved. If there are no conflicts, or when the conflict is resolved, parcel 20-0 issues from the CIP register (refer to Figure 19).

Figure 19. Instruction Flow through Issue Registers (CPn + 4)



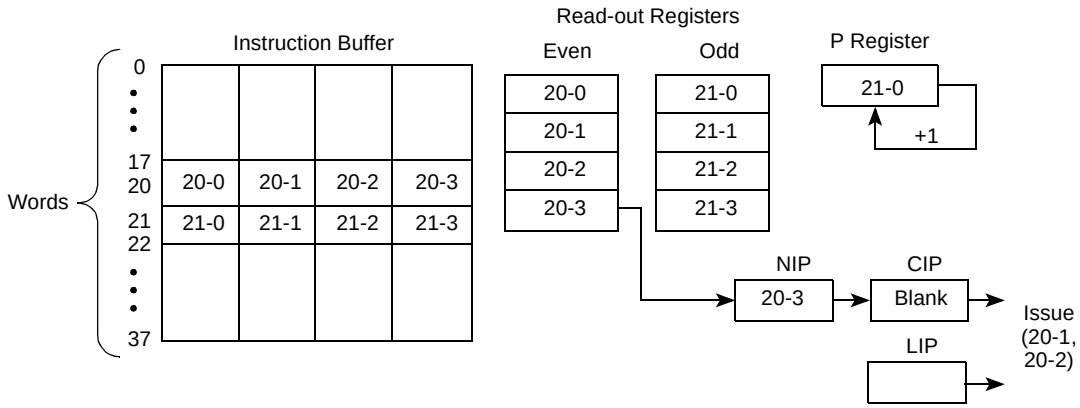
Because parcel 20-1 is the first parcel of a 2-parcel instruction, the logic steers parcel 20-2 into the LIP register and parcel 20-1 into the CIP register. Also, a blank parcel is generated in the NIP register. The P register increments by 1 to point to the next parcel (in this case, parcel 20-3). Issue hardware checks for conflicts. If any conflicts are found, the CIP, LIP, and NIP registers hold their parcels and the P register does not increment (refer to Figure 20).

Figure 20. 2-parcel Instruction Holding 1 CP for Conflict (CPn + 5)



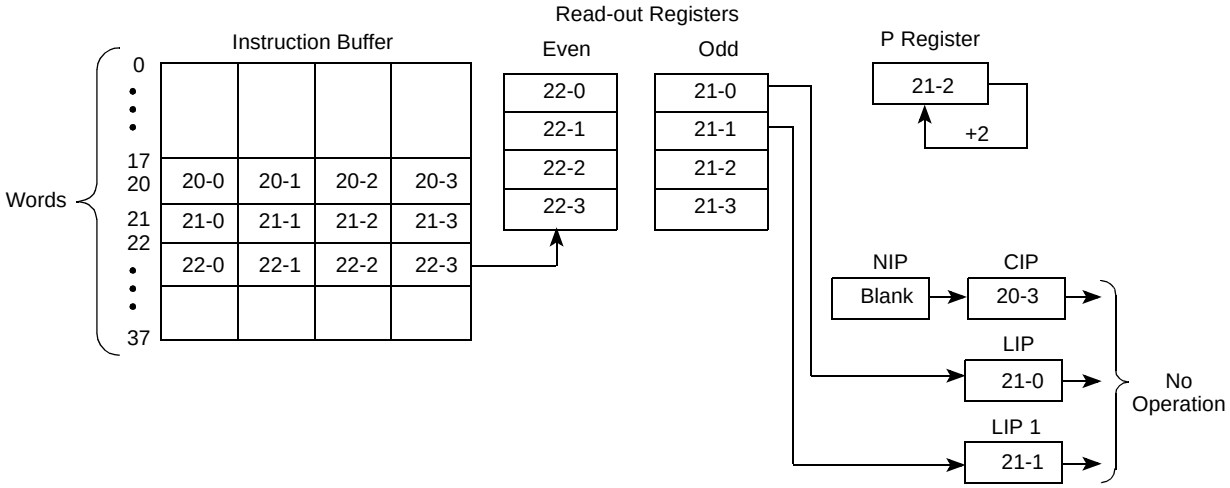
This holding state is maintained until the conflict is resolved. If there are no conflicts, or when the conflict is resolved, parcels 20-1 and 20-2 issue together in the next CP (refer to Figure 21).

Figure 21. Instruction Flow through Issue Registers (CPn + 6)



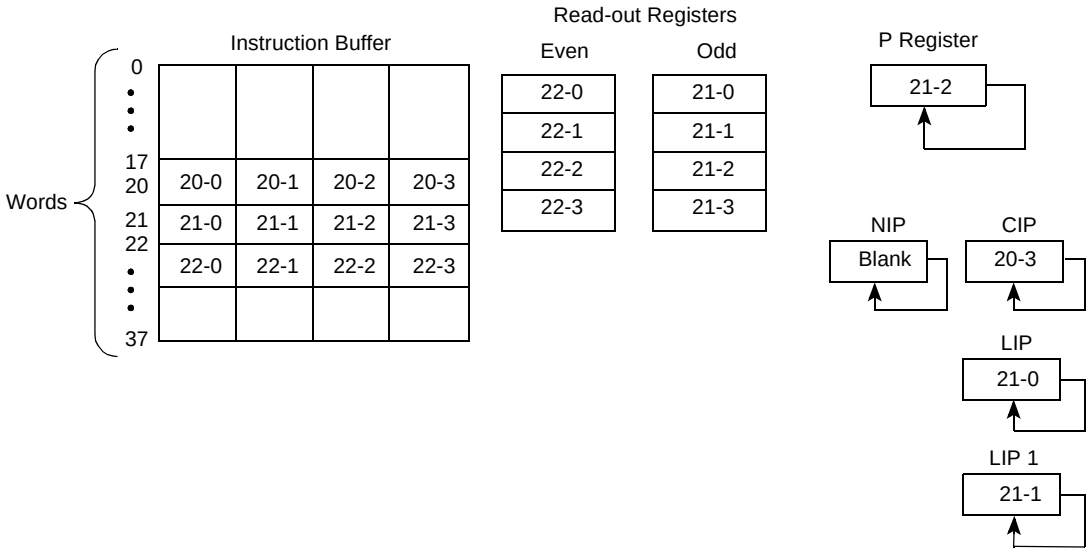
As the 2 parcels move from the CIP and LIP registers to issue, parcel 20-3 is loaded into the NIP register and a blank parcel is loaded into the CIP register. The P register increments by 1 and points to the next parcel (in this case, parcel 21-0). Because the P register no longer points to a parcel in word 20, a new word is loaded into the even read-out register during the next CP. The blank parcel in the CIP register is decoded as a no-operation instruction when it issues during CP*n*+7 (refer to Figure 22).

Figure 22. Instruction Flow through Issue Registers (CPn + 7)



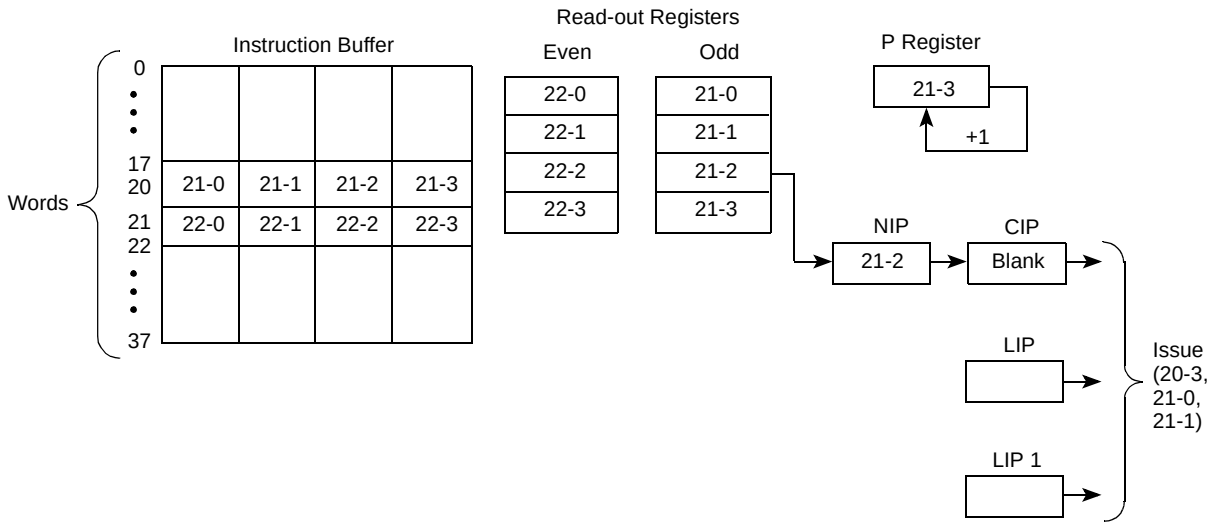
Because parcel 20-3 is the first parcel of a 3-parcel instruction, the logic steers parcel 21-1 into the LIP1 register, parcel 21-0 into the LIP register, and parcel 20-3 into the CIP register. A blank parcel is generated in the NIP register. The P register increments by 2 and points to the next parcel (in this case, parcel 21-2). Issue hardware checks for conflicts. If any conflicts are found, or when the conflict is resolved, the issue registers hold their parcels, and the P register does not increment (refer to Figure 23).

Figure 23. 3-parcel Instruction Holding 1 CP for Conflict (CPn + 8)



This holding state is maintained until the conflict is resolved. If there are no conflicts, parcels 20-3, 21-0, and 21-1 issue together in the next CP (refer to Figure 24).

Figure 24. Instruction Flow through Issue Registers (CPn + 9)



As the 3 parcels move from the CIP, LIP, and LIP1 registers to execution, parcel 21-2 enters the NIP register, and a blank parcel enters the CIP register. The P register increments by 1 to point to the next parcel (in this case, parcel 21-3).

Instructions continue to flow through the issue registers until the program code exits normally or is interrupted. In either case, an exchange sequence and a fetch operation bring new code into the instruction buffers and a new value into the P register, and the issue sequence starts over again.

Table 17 shows the issue sequence that is explained and illustrated in the previous paragraphs. This chart shows the movement of the instruction parcels at each CP as they pass through the issue registers.

Table 17. Instruction Issue Sequence

	CPN	*n*+1	*n*+2	*n*+3	*n*+4	*n*+5	*n*+6	*n*+7	*n*+8	*n*+9
P reg	20-0	20-1	20-2	20-2	20-3	20-3	21-0	21-2	21-2	21-3
NIP		20-0	20-1	20-1	Blank	Blank	20-3	Blank	Blank	21-2
CIP			20-0	20-0	20-1	20-1	Blank	20-3	20-3	Blank
LIP					20-2	20-2		21-0	21-0	
LIP 1								21-1	21-1	

Reservations and Hold Issue Conditions

When the first parcel of an instruction is in the CIP register, hardware determines whether any conflicts are preventing the instruction from executing. These conflicts are referred to as hold issue conditions and cause the instruction to be held in the issue registers until the conflict is resolved. Once the instruction issues, reservations are immediately placed on the local appropriate registers, paths, ports, or functional units as needed. These reservations are usually held a few CPs before the instruction finishes execution; the exact timing depends on the type of instruction.

Register reservations are placed in the following cases:

- A and S registers are reserved as result registers, but not as operand registers.
- Access to the B or T registers is reserved during block transfers.
- Input paths are reserved for the CP during which the data is expected to enter the A or S registers.

Port reservations are placed when the following conditions occur:

- Port A is reserved for memory reads to the B registers.
- Port B is reserved for memory reads to the T registers.
- Port A or port B is reserved for memory reads to the V registers.
- For a write to memory, if port A or port B is busy with a write reference, or if ports A and B are busy.

Conflicts also occur when more than one CPU tries to access the shared path at the same time. The shared path is used by all shared and semaphore registers, and by I/O instructions, interprocessor interrupt signals, and the real-time and programmable clocks.

For a detailed description of the hold issue conditions for each instruction, refer to the “CPU Instruction Descriptions” section of this document for more information. In several cases, these conditions are limited to a specific instruction or instruction sequence. The following list describes a few generalized hold issue conditions.

Scalar instructions hold issue if one of the following conditions occurs:

- The A or S register needed for a result is reserved.
- The input path is reserved for the CP during which incoming data enters the register.
- The instruction references memory, and port A or port B is reserved.

Vector instructions hold issue if the following conditions occur:

- The instruction references memory and the needed port is reserved. Five vector instructions have been dispatched to the vector unit and await issue there.

For B and T register block transfers, a hold issue condition exists if the needed port is reserved. For multiparcel instructions, a hold issue condition exists if the second or third parcel of the instruction is in a different buffer (2-CP delay) or not in any buffer.

Programmable Clock

Each CPU has a programmable clock that generates periodic interrupts at specific preset intervals. Available intervals range between 9 and 2^{32} -1 CPs. Intervals shorter than 100 ms are not practical because of the monitor overhead involved in processing the interrupt. Table 18 lists the monitor mode instructions that are used to enable and disable the programmable clock.

Table 18. Programmable Clock Instructions

CAL Code	Octal Code	Description
PCI S*j*	0014*j*4	Enter interrupt interval register with (S*j*)
CCI	001405	Clear PCI request
ECI	001406	Enable PCI request
DCI	001407	Disable PCI request

Note: On the SV1 CPU, the programmable clock operates at the system clock rate. Operation at the CPU clock rate is possible via a configuration bit change.

Interrupt Interval Register

The 32-bit interrupt interval (II) register is loaded with the number of CPs that elapse between programmable clock interrupt requests. Instruction 0014*j*4 transfers the low-order 32 bits of the S*j* register into the II register. Bit 3 is always forced to a logical 1 for instruction 0014*j*4. The binary value entered into the II register is the number of CPs. The interval is actually one more than the value in the II register. For example, if S*j* equals 0, the II register equals 8 (because bit 3 is always forced set), and the interval equals 9.

This value is held in the II register and is transferred to the programmable clock each time the counter reaches 0 and generates an interrupt request. The contents of the II register are changed only by another 0014*j*4 instruction.

Operation

The 32-bit programmable clock is preset to the value that the II register contains when instruction 0014*j*4 executes. This clock runs continuously and decrements by 1 at each CP until the content of the clock is 0. The programmable clock then sets the programmable clock interrupt (PCI) request and reads the interval value that is held in the II register. The programmable clock repeats the countdown cycle and sets the PCI request at the intervals that the contents of the II register determine.

A PCI request can set only if it is enabled (by instruction 001406), and it remains set until instruction 001405 executes and clears the request. The PCI request causes an interrupt only if the system is not in monitor mode. A request set in monitor mode is held until the system exchanges out of monitor mode.

Following a deadstart sequence, the monitor program ensures the state of the PCI request by issuing instructions 001405 and 001407 to clear and disable the PCI request.

Status Register

The status register holds the status of several flags and bits. The contents of the status register can be sent to the high-order bits of an S register with instruction 073*i*01. Table 19 shows the bit positions and describes the bits and flags in the S register.

Instruction 073*i*01 sets the low-order 32 bits to 1’s and returns the status bits to the high-order bits of the S*i* register. The 073*i*01 instruction is not privileged to monitor mode; the processor number and cluster number bit position return a

value of 0 if the instruction is not executed in monitor mode. The processor number is derived from the configuration register bits (bits 13 through 9) on the PV ASIC (SV1 series) or PVC ASIC (SV1ex series).

The PN and CLN flags return a value of 0 if the system is not in monitor mode when instruction 073*i*01 executes. The UME and CME flags are cleared during an exchange or when any 073 instruction is issued.

Table 19. Si Bit Positions and Bit Descriptions

S*i* Bit Position	Description
63	Clustered, CLN not equal to zero (CL)
57	Bit matrix loaded (BML)
53	Uncorrectable memory error occurred (UME)
52	Correctable memory error occurred (CME)
51	Floating-point error occurred (FPS)
50	Floating-point interrupt enabled (IFP)
49	Operand range interrupt enabled (IOR)
48	Bidirectional memory enabled (BDM)
44	Processor number bit 4 (PN4)
43	Processor number bit 3 (PN3)
42	Processor number bit 2 (PN2)
41	Processor number bit 1 (PN1)
40	Processor number bit 0 (PN0)
37	Cluster number bit 5 (CLN5)
36	Cluster number bit 4 (CLN4)
35	Cluster number bit 3 (CLN3)
34	Cluster number bit 2 (CLN2)
33	Cluster number bit 1 (CLN1)
32	Cluster number bit 0 (CLN0)

Performance Monitor

The performance monitor tracks groups of hardware-related events. These results can be used to indicate the relative performance of a program. The performance monitor contains eight performance counters that track four groups of hardware-related events.

The SV1 series system has a large 32-Kword cache that is used for both data and instruction caching. Cache is enabled by 2 bits in word 7 of the exchange package: ECD and ECF. Effective cache hit counting is simplified by the addition of 4 bits in word 7 of the exchange package that selectively disables cache hit counting. These bits are:

- Disable scalar cache hits - scalar A or S data (DSCH)
- Disable B or T cache hits - B or T data (DBCH)
- Disable fetch cache hits - instruction operands (DFCH)
- Disable vector cache hits - vector data (DVCH)

Note: Selectively apply these disables to increase the effectiveness of cache hit counting.

Cache hits are counted in either Group 0 or Group 2. Both 48-bit counters are identical except that the Group 2 counter may (will) lose counts (due to a small 5-bit accumulator). The Group 0 counter will not lose any cache hit increments and should be used when no cache hits are disabled.

Performance events are monitored only when the CPU is operating in nonmonitor mode. Entering monitor mode disables the performance counters. The groups are selected by the *j* field in instruction 0015*j*0; refer to Table 20 for events that are tracked in each group.

Note: Vector Length (VL) is the vector length of an instruction and ai is the length of the B or T block transfer instruction operation. Cache can be incremented by 1, 2, 3, or 4.

Refer to the “CPU Instruction Descriptions” section for more information on the instructions. Two types of instructions are used with the performance monitor: user instructions and maintenance instructions. The user instructions allow the user to select and read the performance monitor. The maintenance instructions test the logic of the performance monitor. The following subsections explain how these instructions are used with the performance monitor.

Table 20. Performance Counter Group Descriptions

Group	Performance Counter	Monitored Event	Incremented
Number of:			
0	0	Instruction issued	+1
0	1	Clock period holding issue	+1
0	2	Instruction fetches	+1
0	3	Floating-point add operation	+1 thru +2, +VL
0	4	Floating-point multiply operation	+1 thru +2, +VL
0	5	Floating-point reciprocal operation	+1 thru +2, +VL
0	6	CPU Memory references	+1 thru +4, +VL, +A*i*
0	7	Cache hits	+1 thru +4
Holding issue on:			
1	0	A registers	+1
1	1	S registers	+1
1	2	V registers	+1
1	3	B, T registers	+1
1	4	V functional units	+1
1	5	Shared registers	+1
1	6	Memory ports	+1
1	7	Miscellaneous	+1
Number of:			
2	0	Instruction fetches	+1
2	1	Cache hits	+1 thru +4
2	2	Scalar memory writes	+1
2	3	B, T memory references	+A*i*
2	4	Scalar memory references	+1 thru +4
2	5	CPU memory writes	+1 thru +2, +VL, +A*i*
2	6	CPU memory references	+1 thru +4, +VL, +A*i*
2	7	CPU memory conflicts	+1 thru +4
Number of:			
3	0	000 – 017 instructions	+1
3	1	020 – 077 instructions	+1
3	2	100 – 137 instructions	+1
3	3	140 – 157, 174 ($k \neq 0$) instructions	+1
3	4	160 – 173, 174 ($k = 0$) instructions	+1
3	5	176, 177 instructions	+1

Table 20. Performance Counter Group Descriptions (continued)

Group	Performance Counter	Monitored Event	Incremented
3	6	Vector integer operation (from #3)	+1, +VL
3	7	Vector floating-point operation (from #4)	+1, +VL

Note: The events monitored in Group 1, performance counter 7 are listed as miscellaneous. These miscellaneous events are defined as CMR instruction issue conflicts, vector unit instruction conflicts, test mode conflicts, vector logical busies, split parcel holds, floating-point busies, memory busies, and status busies.

Selecting and Reading Performance Events

Table 21 lists the two instructions that select and read the performance monitor. The primary function of instruction 0015*j*0 is to select one of the four groups of performance events to be tracked. It also clears the performance counters and the performance counters' pointer (explained later in this subsection). After instruction 0015*j*0 selects a group, the performance counters advance their totals according to the number of monitored events that occur. The performance counters can continuously monitor events for approximately 65 hours before they must be reset. Fifty CPs must elapse before another performance monitor instruction issues. The performance counter operates at the CPU clock rate (CPU CPs)

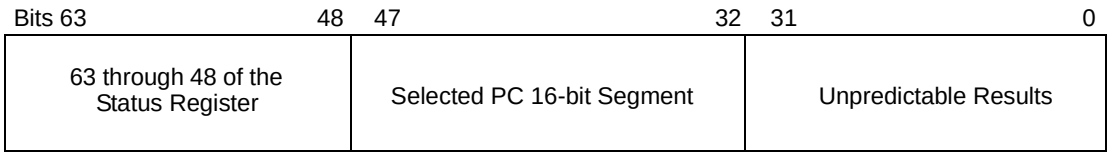
Table 21. Performance Monitor User Instructions

Octal Instruction	Primary Function	Secondary Functions
0015*j*0[a]	Selects the performance monitor. The *j* field selects the group to be monitored.	Clears all performance counters and clears the performance counter pointer.
073*i*11[a]	Reads 16 bits of the performance counter into S*i*.	Reads 16 bits of status register into S*i* and increments the performance counter pointer.

[a] This instruction is privileged to monitor mode.

Instruction 073*i*11 is used for performance monitoring and is privileged to monitor mode. Each execution of the 073*i*11 advances a pointer. Instruction 073*i*11 performs two functions. Its primary function is to read 16-bit segments of the performance counters into bits 32 through 47 of an S register (S*i*) (refer to Figure 25). Its secondary function is to read bits 48 through 63 of the status register into bits 48 through 63 of the same S register.

Figure 25. Contents of an S Register During Execution of 073i11 Instruction



Each performance counter is 48 bits wide and is divided into three 16-bit segments. A performance counter pointer selects which 16-bit segment to read into the S register. The performance counter pointer is cleared either on entry from or exit to monitor mode, or by instruction 0015*j*0 or 073*i*31.

The following example shows a sequence for reading a set of performance counters:

Step	Octal Code	Description
1	073*i*11	Bits 0 through 15 of counter 0 to S*i* bits 32 through 47.
2		4-CP delay
3	073*i*11	Bits 16 through 31 of counter 0 to S*i* bits 32 through 47.
4		4-CP delay
5	073*i*11	Bits 32 through 47 of counter 0 to S*i* bits 32 through 47.
6		4-CP delay
7	073*i*11	Bit 0 through 15 of counter 1 to S*i* bits 32 through 47.
8		4-CP delay
-	-	-
-	-	-
-	-	-
n	073*i*11	Read bits 32 through 47 of counter n to the S*i* register.

In Step 1, instruction 073*i*11 reads bits 0 through 15 of counter 0 into the S*i* register and increments the performance counter pointer. In Step 3, instruction 073*i*11 reads bits 16 through 31 of counter 0 into the S*i* register and increments the performance counter pointer. In Step 5, instruction 073*i*11 reads bits 32 through 47 of counter 0 into S*i* and increments the performance counter pointer. In Step 7, the process begins again, transferring the three 16-bit segments of counter 1 into the S*i* register. After each 073*i*11 instruction, the performance counter pointer advances by 1; a 4-CP delay must occur between sequential issues of instruction 073*i*11.

Testing Performance Counters

Instructions 073*i*21, 073*i*31, and 073*i*61 test the operation of the performance counter. Instruction 073*i*21 adds 4000020000000 (octal) to the contents of the performance counter by injecting 1's at bit positions 22 and 38. Each of these bit positions contains bit 7 of the middle parcel and bit 7 of the most significant parcel. The performance counter pointer is advanced to the next counter. This instruction also reads the status register into the S*i* register.

Instruction 073*i*31 clears the performance counter pointer and clears all maintenance modes. It also reads the status register into S*i*.

Instruction 073*i*61 increments the selected performance counter by adding 1 to bit position 0. Instruction 073*i*61 also reads status register bits 32 through 63 to a selected S register.

Cache Memory

The concept of a cache memory allows operations of the main memory address space to be mapped into a small fast memory associated with, and located in or near a CPU. The cache is split into a number of lines, or groups, of data words, which represent main memory locations. Each cache line has a tag that identifies the main memory address that the line represents.

The SV1 series cache is organized as a 4-way set associative, single word line write-through cache, tag supported for single 64-bit references (coherency supported by software). Both scalar and vector data is cached along with instructions. Scalar read references prefetch 8 words from memory into cache by allocating 8 single word lines for each scalar reference. This provides spatial locality for scalar code.

The SV1 series, with up to 32 CPUs possible per mainframe, has main memory divided into 8 sections. These sections are selected by the lowest 3 address bits, bits [2:0], on a reference to memory. Each of the 32 CPUs has its own cache which is further divided into 8 separate units, one for each section of memory. Each CPU has 256 Kbyte, 32 Kwords, of cache divided into 4 Kwords per section. The 4 Kwords of cache for each section are arranged in a 1,024 by 4-word array. Address bits [12:3] select one of the 1,024 locations, and address bits [31:13] are part of the tag for each word. When all 4 words per location are reserved, then the least recently used (LRU) word is reallocated upon another request.

Cache is organized on a section basis. Due to hardware limitations, the 8 sections of cache per CPU must share four request buses, two sections per bus, which are arranged as follows:

- Sections 0 and 2 requests, and sections 5 and 7 requests share one write bus
- Sections 4 and 6 requests, and sections 1 and 3 requests share one write bus

This section grouping was selected to support stride references to memory. Each cache section is busy for 2 CPU CPs due to the cache chip design. This design requires one clock period between references and the request busses are assigned specific section pairs for efficiency.

The SV1 cache is enabled by 2 bits in word 7 of the exchange package. The Enable Cache Data (ECD) bit enables cache for data use by A, S, B, T, and vector requests. The Enable Cache Fetch (ECF) bit enables cache for instruction use. Cache for a CPU, when enabled, is common for all data references and instruction references. Although self-modifying code will be current in the cache, it will not be current in the instruction buffers for a CPU because these are not updated. Instruction buffers must be flushed to keep them current.

The SV1 cache is a write-through cache where the write request data is put into the cache and a write request is also sent to memory to update that location. A read request to cache that results in a cache hit (the data is in cache) will not send that read request to memory. This reduces unnecessary memory activity. Only read or write requests that “miss” in cache allocate cache lines. The read data from memory for a missed read request updates cache and is also sent to the CPU. Refer to Table 22, which summarizes the general cache operation.

Cache is invalidated in three ways:

- By another CPU using the 0016*j*1 instruction where (A*j*) is the CPU number
- At the beginning and end of each exchange operation for this CPU
- By the 0034*jk* test and set instruction for this CPU (if enabled). The 0034*jk* instruction invalidates cache only when enabled by the Enable Cache Invalidate (ECI) bit in word 7 of the exchange package.

Table 22. Cray SV1 Series Cache Operations

Operand/ Operation	Read/ Write	Hit/Miss	Request to Memory	Allocate	Update on Return	Update on Write	Invalidate Cache
Fetch	Read	Hit					
		Miss	X	X	X(b)		
Scalar B or T Vector	Read	Hit					
		Miss	X	X	X(b)		
	Write	Hit	X			X	
		Miss	X	X		X	
Exchange	-	-	X				X
Flush(a)	-	-					X

(a) Cache is flushed (invalidated) by another CPU (0016*j*1), 0034*jk* (if ECI set), and before and after every exchange operation.

(b) Update except for special Read/Write request sequence.

Detailed Operation of Cache Memory

Figure 26 shows the physical organization of the cache for a section of memory. Each cache line consists of a Tag Valid, a Data Valid, and an Address Tag and 1 word of data as shown in Figure 26. Not shown are bits used for LRU purposes. The 1024 X 4 cache array per section has 4 such cache lines per location. Address bits [12:3] of the read or write request select the location and address bits [31:13] are used for the Address Tag. Address bits [2:0] select the section. The key concepts concerning the operation of the cache are as follows:

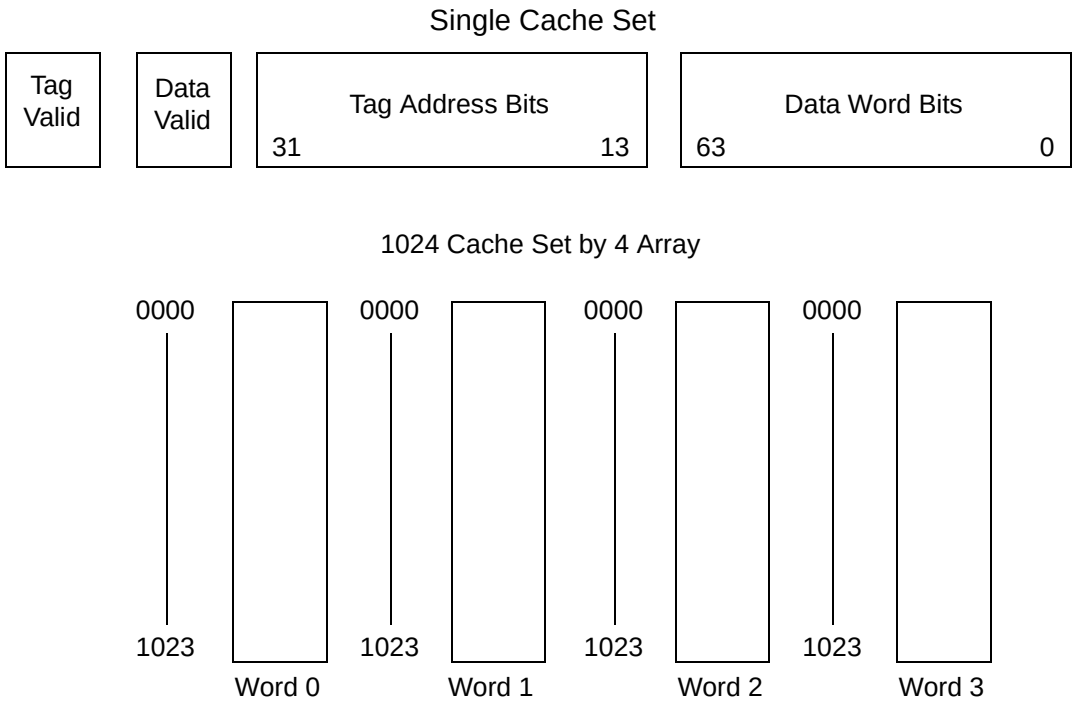
- A cache line can be allocated only when not already requested.
- A cache word is valid when both the Tag Valid and Data Valid bits are set.
- A cache line state is requested when either, not both, valid bits are set.
- A read request is sent to memory when not allocated or data is not present.
- A cache line hit normally occurs on a tag match with the Tag Valid bit set.
- A write request to memory following an outstanding read request for the same word writes the data into cache, sets the Data Valid bit, and clears the Tag Valid bit. The returned read data is sent to the CPU and only sets the Tag Valid bit.

Invalidating all of cache clears both state bits: Tag Valid and Data Valid. Replacing an allocated cache line, per the LRU, clears both Tag Valid and Data Valid (both must have been set), reloads the Address Tag, and resets the valids as required. A request to memory that misses the cache and cannot allocate a new cache line makes a normal memory request and does not affect the cache.

A read/write/read request combination for the same word results in the second read being queued in a cache first-in-first-out (FIFO) buffer without the read request being sent to memory.

The cache area for each section of memory has a FIFO port queue that is used for all read requests that are sent to memory. Read data from memory, per CPU section, is returned in the same order that the read requests were sent to memory. In addition to indicating where in the CPU the read data is to be delivered (if at all), the FIFO port queue also includes cache information that is related to the read request. A cached read request stores a return-to-cache valid, the request address bits [12:3], and a 2-bit code that identifies the one-of-four words per location for this request. The cache line itself then determines whether the word should be written to cache. Seven of eight scalar A or S read operands are normally written to cache but not sent to the CPU.

Figure 26. 1-word Line, 4-way Associative 4096-word Cache per Memory Section



Tag Valid	Data Valid	Definition
0	0	Not allocated
1	0	Allocated, but without data from memory
1	1	Allocated and data is present
0	1	Allocated and data is present, but earlier read data is not yet back from memory (for read/write combination)

Transitions: 00 can go to 10 on a read request, or 11 on a write request

10 can go to 11 on a read data from memory, or 01 on a write request

01 can go to 11 on a read valid from memory (for a read/write combination)

11 will remain at 11 on either a no change or a new write request

CPU Computation

Each CPU is an identical, independent computation section that consists of operating registers, functional units, and an instruction control network. The operating registers and functional units are associated with three types of processing: address, scalar, and vector.

Address processing operates on internal control information, such as addresses and indexes. The address (A) registers, intermediate address (B) registers, and two functional units are dedicated solely to address processing.

Scalar processing is sequential and uses one operand or operand pair to produce a single result. Scalar processing uses the scalar (S) registers and the intermediate scalar (T) registers. Scalar processing also uses four functional units that are dedicated to scalar processing and three floating-point functional units that it borrows from the vector area and uses as required.

Vector processing allows a single operation to be performed concurrently on a set (or vector) of operands, repeating the same function to produce a series of results. Vector processing uses the vector (V) registers. Vector processing also uses dual-pipe functional units that are dedicated to vector processing plus three dual-pipe floating-point functional units that it shares with scalar processing (pipe 0 units only).

Data flow in a computation section is from central memory to registers and from registers to functional units. Results flow from functional units to registers and from registers to central memory or back to functional units. Depending on the instruction sequence, data flows along either the scalar or vector path, or both simultaneously.

The computation section performs integer or floating-point arithmetic operations. Integer arithmetic is performed in two’s complement mode; floating-point quantities have signed magnitude representation.

Integer (or fixed point) operations are integer addition, integer subtraction, and integer multiplication. No integer division instruction is provided; the operation is accomplished through a software algorithm using floating-point hardware.

Floating-point instructions allow addition, subtraction, multiplication, and reciprocal approximation operations. The reciprocal approximation instructions provide a floating-point division operation that uses a multiple instruction sequence.

The instruction set includes logical operations for AND, inclusive OR, exclusive OR, exclusive NOR, and mask-controlled merge operations. Shift operations allow the manipulation of either 64-bit or 128-bit operands to produce 64-bit results. Most operations are used in vector or scalar instructions, including 32-bit integer arithmetic.

The 32-bit integer product is a scalar instruction that is designed for index calculation. A full-indexing capability is possible throughout central memory in either scalar or vector modes. The index can be positive or negative in either mode. Indexing allows matrix operations in vector mode to be performed on rows or on the diagonal as well as allowing conventional column-oriented operations.

The SV1 includes a bit-matrix multiply (BMM) unit that is available for both scalar and vector operations.

The following subsections describe the operating registers and their associated functional units.

Operating Registers

Each CPU has three primary and two intermediate sets of operating registers. The primary sets of operating registers are the address (A), scalar (S), and vector (V) registers. These registers are considered primary because functional units and central memory can access them directly.

For the A and S registers, an intermediate level of registers exists; they are not accessible to the functional units, but they act as a buffer for the primary registers. To reduce the number of memory reference instructions for scalar and vector operations, block transfers are possible between these intermediate registers and central memory. The intermediate address (B) registers support the A registers, while the intermediate scalar (T) registers support the S registers. The V registers do not have intermediate registers.

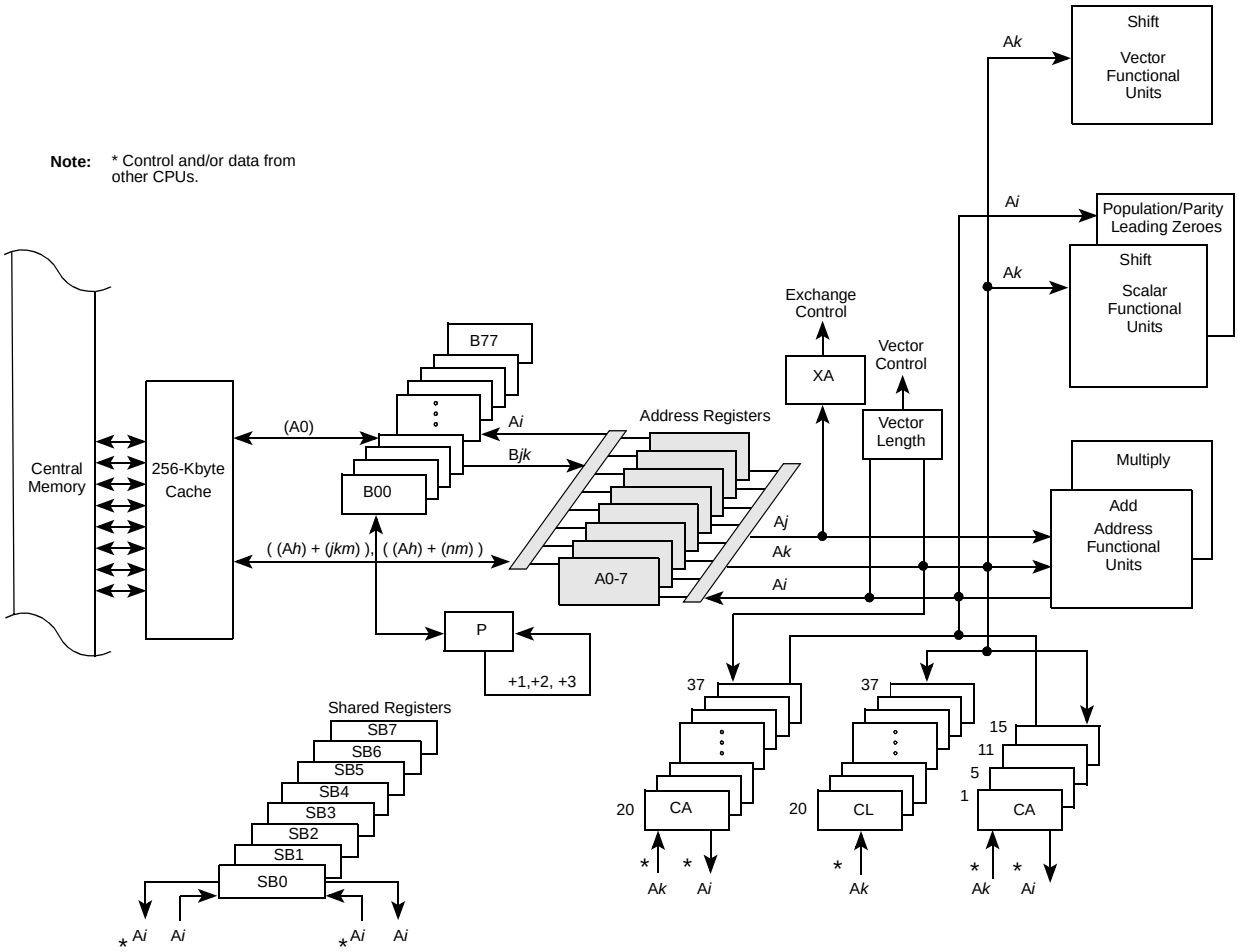
In the SV1 series system, the vector unit includes an additional set of 64 registers, the matrix BT registers, which are used with the scalar and vector bit-matrix multiply operations.

Address (A) Registers

Figure 27 shows the eight A registers and their associated CPU hardware. The A registers are designated A0 through A7.

The A registers and address functional units run at a full 32-bit width and the instruction set includes 3-parcel instructions. The following subsections explain A register functions, special uses, and instructions.

Figure 27. A Register Block Diagram



A Register Functions

The A registers serve as address registers for memory references and as index registers. A registers transfer and receive 32 bits. Refer to "Calculating Absolute Memory Address" for additional information. The A registers index the base address for scalar memory references and provide both a base address and an address increment for vector memory references. The A registers also provide values for shift counts, loop control, and channel I/O operations (setting the channel limit [CL] and current address [CA] registers) and serve as result registers for the scalar population/parity/leading zero functional unit.

The A registers are connected to the vector length (VL) and exchange address (XA) registers. The VL register is loaded by the 002 instruction. The XA register is loaded by the 0013*j*0 instruction only while the system is operating in monitor mode. Refer to the "Vector Length Register" section for more information on the VL register. Refer to the "Exchange Address Register Field" section for more information on the XA register.

Data either moves between central memory, a 32-Kword cache, and the A registers, or it is placed in the B registers. The B registers buffer the data between A registers and central memory. Data can also be transferred between A and S registers and between A registers and shared address (SB) registers.

The following list summarizes the functions of the A registers:

- Generate addresses for memory references and function as index registers.
- Set the CA and CL registers (I/O control).
- Provide values for shift counts and loop controls.
- Serve as result registers for the scalar population/parity/leading zero functional unit.
- Set the XA register (exchange control).
- Set and read the VL register (vector control).
- Transfer data between the A and S registers.
- Transfer data between the A and SB registers.

The address functional units support address and index generation by performing 32-bit integer arithmetic on operands that are obtained from A registers and by delivering the results to A registers. Refer to the "Address Functional Units" section for more information on the address functional units.

Special A Register Values

If register A0 is referenced in the *h*, *j*, or *k* fields of an instruction, the contents of the register are not used; instead, a special operand is generated. The special value is available regardless of existing A0 register reservations (they are not checked in this instance, and this special value does not alter the actual value of the A0 register. Table 23 shows the special A0 register values.

Table 23. Special A0 Register Values

Field	Operand Value
A*h*, *h* = 0	0
A*j*, *j* = 0	0
A*k*, *k* = 0	1

If the *i* field equals 0, then the contents of register A0 are used. The *i* field is not used as a special case.

A Register Instructions

Only one result per CP can be transferred to the A registers. When an instruction that delivers new data to an A register issues, a reservation is set for that register. The reservation prevents the issue of instructions that use the register until the new data is delivered. Instructions reference A registers by specifying the register number as the *h*, *i*, *j*, or *k* designator (refer to the "Instruction Formats" section for more information on instruction fields). A0 is the only A register that can be referenced when it is not specified in one of the instruction fields.

Table 24 lists A register instructions and provides octal and CAL codes. The content of the DBA register is added to instruction-generated memory addresses to form absolute memory addresses. Refer to the "Calculating Absolute Memory Address" section. Refer to the "CPU Instruction Descriptions" section for complete information on these instructions.

Only one input path to the A registers exists; therefore, all instructions that write data into the A registers must reserve the path for the CP when data arrives. The issue hardware determines which CP to reserve the path for the instruction, and it reserves the path for that CP. If the path is already reserved,

the instruction holds issue. The instruction continues to hold issue until the A register path is available in the CP when the data arrives. The instruction then issues and reserves the path for that CP.

Table 24. A Register Instructions

Machine Instruction	CAL Syntax	Description	Type of Instruction
020*i*00*mn*	A*i exp*	Transmit *nm* to A*i*	Register entry
021*i*00*mn*	A*i* #*exp*	Transmit one's complement of *exp* to A*i exp = nm*	
022*ijk*	A*i exp*	Transmit *jk* to A*i*	
031*i*00	A*i* -1	Transmit -1 to A*i*	
10*hi*00mn	A*i exp*,A*h*	Load from ((A*h*) + *exp*) to A*i exp = nm (h ≠ 0)*	Memory transfer (Load)
100*i*00*mn*	A*i exp*,0	Load from *(exp)* to A*i exp=nm*	
100*i*00*mn*	A*i exp,*	Load from *(exp)* to A*i exp=nm*	
10*hi*0000	A*i* ,A*h*	Load from (A*h*) to A*i (h ≠ 0)*	
11*hi*00*mn*	*exp*,A*h* A*i*	Store (A*i*) to ((A*h*) + *exp*) *exp = nm (h ≠ 0)*	Memory transfer (Store)
110*i*00*mn*	*exp*,0 A*i*	Store (A*i*)to *exp exp=nm*	
110*i*00mn	*exp*, A*i*	Store (A*i*) to *exp exp=nm*	
11*hi*0000	,A*h* A*i*	Store (A*i*) to (A*h*) *(h ≠ 0)*	
0013*j*0	XA A*j*	Transmit (A*j*) to XA register	Interregister transfer
0014*j*3	CLN A*j*	Transmit (A*j*) to CLN register	
00200*k*	VL A*k*	Transmit (A*k*) to VL register	
023*ij*0	A*i* S*j*	Transmit (S*j*) to A*i*	
023*i*01	A*i* VL	Transmit (VL) to A*i*	
024*ijk*	A*i* B*jk*	Transmit (B*jk*) to A*i*	
025*ijk*	B*jk* A*i*	Transmit (A*i*) to B*jk*	
027*ij*7	SB*j* A*i*	Transmit (A*i*)to SB*j*	
030*i*0*k*	A*i* A*k*	Transmit (A*k*) to A*i*	
031*i*0*k*	A*i* -A*k*	Transmit the negative of (A*k*) to A*i*	
071*i*0*k*	S*i* A*k*	Transmit (A*k*) to S*i* with no sign extension	
071*i*1*k*	S*i* + A*k*	Transmit (A*k*) to S*i* with sign extension	
071*i*2*k*	A*i* + FA*k*	Transmit (A*k*) to S*i* as unnormalized floating-point number	
030*ijk*	A*i* A*j* + A*k*	Transmit integer sum of (A*j*) and (A*k*) to A*i*	Integer operation
030*ij*0	A*i* A*j* + 1	Transmit integer sum of (A*j*) and 1 to A*i*	
031*ijk*	A*i* A*j*-A*k*	Transmit integer difference of (A*j*) and (A*k*) to A*i*	
031*ij*0	A*i* A*j* - 1	Transmit integer difference of (A*j*) and 1 to A*i*	
032*ij*k	A*i* A*j** A*k*	Transmit integer product of (A*j*) and (A*k*) to A*i*	

Table 24. A Register Instructions (continued)

Machine Instruction	CAL Syntax	Description	Type of Instruction
010*ijkm*	JAZ *exp*	Jump to *exp* if (A0) = 0(*i* 2 = 0)	Conditional jump
011*ijkm*	JAN *exp*	Jump to *exp* if (A0) ≠ 0 (*i* 2 = 0)	
012*ijkm*	JAP *exp*	Jump to *exp* if (A0) positive (*i* 2 = 0)	
013*ijkm*	JAM *exp*	Jump to *exp* if (A0) negative (*i* 2 = 0)	
026*ij*0	A*i* PS*j*	Transmit population count of (S*j*) to A*i*	Bit count
026*ij*1	A*i* QS*j*	Transmit population count parity of (S*j*) to A*i*	
027*ij*0	A*i* ZS*j*	Transmit leading zero count of (S*j*) to A*i*	
033*i*00	A*i* Cl	Transmit channel number of highest priority interrupt request to A*i* (*j* = 0)	Register channel
033*ij*0	A*i* CA,A*j*	Transmit current address of channel (A*j*) to A*i* (*j* ≠ 0, *k* = 0)	
033*ij*1	A*i* CE,A*j*	Transmit error flag of channel (A*j*) to A*i* (*j* ≠ 0, *k* = 1)	
0014*j*1	SIPI A*j*	Send interprocessor interrupt request to CPU (A*j*)	Interrupt
0016*j*1	IVC	Send invalidate cache request to CPU (A*j*)	Invalidate cache

Notes: In addition to the A register use by instructions, (A0) and (A*i*) are used with the B and T load and store instructions 034*ijk*, 035*ijk*, 036*ijk*, and 037*ijk*. Also, the (A*k*) is used as the shift count in scalar shift instructions 056 and 057, and in vector shift instructions 150, 151, 152, and 153.

Intermediate Address (B) Registers

Sixty-four 32-bit B registers are designated B0 (octal) through B77 (octal). The B registers serve as intermediate storage registers for the A registers. B registers typically contain data to be referenced repeatedly over a long time, which makes it inefficient to retain the data in either A registers or in central memory. Examples of data that B registers store are loop counts, variable array base addresses, and dimensions.

Instructions reference B registers by specifying the B register number in the *jk* field. Refer to “Instruction Formats” for more information on instruction fields.

Data transfers between an A and B register take 1 CP. A block of data transfers between B registers and central memory at a maximum rate of two registers per CP. During these block transfers, a reservation is made on all B registers that are used in the block transfer.

The *jk* fields of the instruction specify the first register that is involved in a block transfer; the low-order 7 bits of the contents of register A*i* specify the number of words that are transmitted. Successive transfers involve successive B register pairs until B76/B77 is reached. Registers B00/B01 are processed after registers B76/B77 if the count in register A*i* is not equal to zero. Other instructions can issue while a block of B registers is transferred to or from central memory. B00 is the only B register that can be referenced when it is not specified in one of the instruction fields. Table 25 lists the B register instructions.

There are 64 B registers, but the length of the B register load or store operation as specified by the 7-bit length in register A*i* can be greater than 64 (up to 127). For a load operation with a length greater than 64, the B register load control operates for the length divided by 2 cycles, but it actually sends load requests to cache/memory for the last 64 operands. For a store operation with a length greater than 64, the B register store control operates for the length divided by 2 cycles and sends data to cache/memory for every cycle.

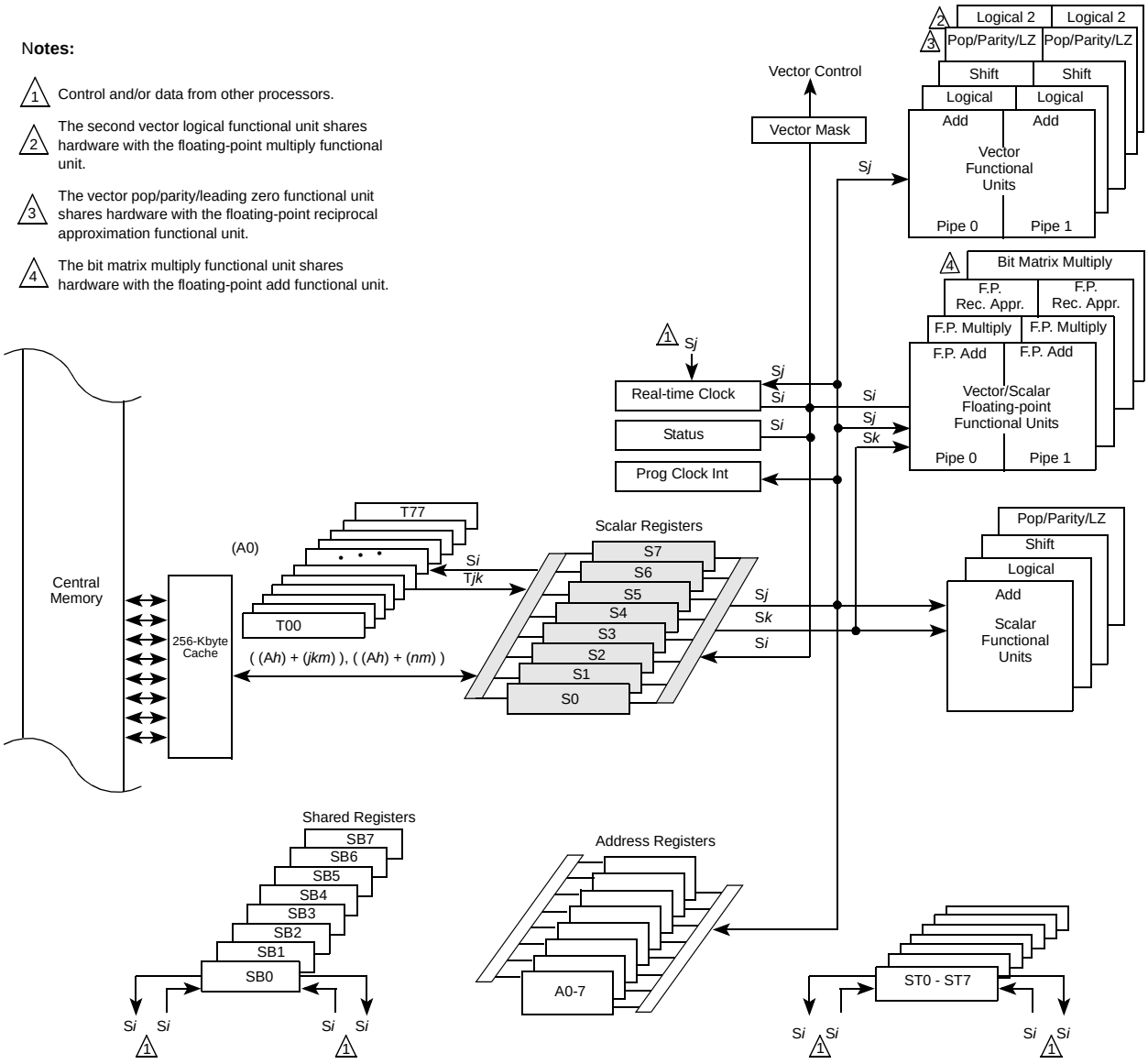
Table 25. B Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
024*ijk*	A*i* B*jk*	Transmit (B*jk*) to A*i*	Interregister transfer
025*ijk*	B*jk* A*i*	Transmit (A*i*) to B*jk*	
034*ijk*	B*jk*,A*i* ,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*	Block transfer
034*ijk*	B*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*	
035*ijk*	,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0)	
035*ijk*	0,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0)	
0050*jk*	J B*jk*	Jump to (B*jk*)	Jump
007*ijkm*	R *exp*	Return jump to *exp*; set B00 to (P) + 2	

Scalar (S) Registers

Figure 28 shows the eight S registers and their associated hardware. The S registers (S0 octal through S7 octal) are 64 bits wide. They are the principal scalar registers for a CPU and serve as the source and destination for operands that perform scalar arithmetic and logical operations.

Figure 28. Scalar Register Block Diagram



S Register Functions

The S registers provide the operands for scalar integer arithmetic, logical, shift, and pop/parity/leading-zero operations. They also provide the operands for scalar floating-point operations executed in the vector pipe 0 floating-point units, and the operands for the bit matrix multiply operation executed in the vector unit. Additionally, the S registers provide the constant that is used with some vector operations. Single-word transmissions of data, in either direction, between an S register and an element of a V register, or the vector mask (VM) register, are also possible. An S register provides the value to load into the real-time clock (RTC) via the JS ASIC; the RTC can be read to an S register. The interrupt interval (II) register in the programmable clock is also set from an S register.

Data moves directly between cache/central memory and S registers or is placed in the T registers. The T registers buffer scalar operands between S registers and cache/central memory. Data is also transferred between S and A registers, between S and shared scalar (ST) registers, and between S and semaphore (SM) registers.

The S registers can also read the contents of the status register; instruction 073*ij*1 sets the low-order 32 bits to 1's and returns certain status register bits to the high-order bits of the S*i* register. For more information on the 073*ijk* instruction, refer to "CPU Instruction Descriptions."

The S registers are primarily used for scalar operations. The following list summarizes other functions of the S registers:

- Provide a constant value for vector operations
- Set/read the RTC and VM registers
- Set the II register
- Transfer data between A and S registers
- Transfer data between S registers and ST or SM registers
- Read the contents of the status register

The scalar functional units support the S registers by performing integer arithmetic operations; scalar floating-point arithmetic is performed in the vector pipe 0 floating-point units. Refer to "Scalar Functional Units" for more information on the scalar functional units.

Special S Register Values

If register S0 is referenced in the *j* or *k* fields of an instruction, the contents of the register are not used; instead a special operand is generated. The special value is available regardless of the existing S0 register reservations (they are not checked in this instance). This use does not alter the actual value of the S0 register. Table 26 shows the special S0 register values.

Table 26. Special S0 Register Values

Field	Operand Value
S*j*, *j* = 0	0
S*k*, *k* = 0	2^{63}

If the *i* field equals 0, then the contents of the S0 register are used. The *i* field is not used as a special case.

S Register Instructions

Only one result per CP can be transferred to the S registers. When an instruction that delivers new data to an S register issues, a reservation is set for that register. This reservation prevents the issue of instructions that read the register until the new data is delivered. Instructions reference S registers by specifying the register number as the *i*, *j*, or *k* designator. Refer to "Instruction Formats" for more information on instruction fields. S0 is the only S register that can be referenced when it is not specified in one of the instruction fields.

Table 27 lists S register instructions and provides the octal and CAL codes. Refer to "CPU Instruction Descriptions" for complete information on these instructions. The contents of the DBA register are added to instruction-generated memory addresses to form physical memory addresses. Refer to "Address Range Checking."

Only one input path to the S registers exists; therefore, all instructions that write data into the S registers must reserve the path for the CP in which data arrives. The issue hardware determines the proper CP and reserves the path for that CP. If the path is already reserved, the instruction holds issue until the reservation is cleared. The instruction continues to hold issue until the S register path is available during the CP in which the data arrives. The instruction then issues and reserves the path for that CP.

Table 27. S Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
040*i*00*mn*	S*i exp*	Transmit *exp* to S*i exp* = *nm*	Register entry
041*i*00*mn*	S*i* #*exp*	Transmit one's complement of *exp* to S*i exp* = *nm*	
042*ijk*	S*i* <*exp*	Form ones mask in S*i exp* bits from right in *exp* = 100 octal - *jk* bits	
042*ijk*	S*i* #>*exp*	Form zeroes mask in S*i exp* bits; from left *exp* = *jk* bits	
042*i*77	S*i* 1	Transmit 1 to S*i*	
042*i*00	S*i* -1	Transmit -1 to S*i*	
043*ijk*	S*i* >*exp*	Form ones mask in S*i exp* bits from left; *exp* = *jk* bits	
043*ijk*	S*i* #<*exp*	Form zeroes mask in S*i* bits from right *exp* = 100_8 -*jk* bits	
043*i*00	S*i* 0	Clear S	
047*i*00	S*i* #SB	Transmit one's complement of sign bit to S*i*	
071*i*30	S*i* 0.6	Transmit 0.75 as normalized floating-point constant to S*i*	
071*i*40	S*i* 0.4	Transmit 0.5 as normalized floating-point constant to S*i*	
071*i*50	S*i* 1.	Transmit 1.0 as normalized floating-point constant to S*i*	
071*i*60	S*i* 2.	Transmit 2.0 as normalized floating-point constant to S*i*	
071*i*70	S*i* 4.	Transmit 4.0 as normalized floating-point constant to S*i*	
12*hi*00*mn*	S*i exp*, A*h*	Load from ((A*h*) + *exp*) to S*i* (*h* ≠ *0*) *exp* = *nm*	Memory transfer (Load)
120*i*00*mn*	S*i exp*, 0	Load from (*exp*) to S*i exp* = *nm*	
120*i*00*mn*	S*i exp*,	Load from (*exp*) to S*i exp* = *nm*	
12*hi*000	S*i* ,A*h*	Load from (A*h*) to S*i (h* ≠ *0)*	
13*hijkm*	*exp*,A*h* S*i*	Store (S*i*) to ((A*h*) + *exp) exp* = *nm* (*h* ≠ *0*)	Memory transfer (Store)
130*ijkm*	*exp*,0 S*i*	Store (S*i*) to *exp exp* = *nm*	
130*ijkm*	*exp*, S*i*	Store (S*i*) to *exp exp* = SS*nm*	
13*hi*000	,A*h* S*i*	Store (S*i*) to address(A*h*)	
0014*j*0	RT S*j*	Transmit (S*j*) to RTC	Interregister transfer
0014*j*4	PCI S*j*	Transmit (S*j*) to Interrupt Interval (II) register	
0030*j*0	VM S*j*	Transmit (S*j*) to VM register	
003000	VM 0	Clear the VM register	

Table 27. S Register Instructions (continued)

Machine Instructions	CAL Syntax	Description	Type of Instruction
023*ij*0	A*i* S*j*	Transmit (S*j*) to A*i*	Interregister transfer (cont.)
047*i*0*k*	S*i* #S*k*	Transmit one's complement of (S*k*) to S*i*	
047*i*00	S*i* #SB	Transmit one's complement of sign bit to S*i*	
051*i*0*k*	S*i* S*k*	Transmit (S*k*) to S*i*	
051*i*00	S*i* #SB	Transmit sign bit to S*i*	
072*i*00	S*i* RT	Transmit (RTC) to S*i*	
072*i*02	S*i* SM	Read semaphores to S*i*	
072*ij*3	S*i* ST*j*	Transmit (ST*j*) to S*i*	
073*i*00	S*i* VM	Transmit (VM) to S*i*	
073*i*11		Read performance counter into S*i*	
073*i*01	S*i* SR0	Transmit (SR0) to S*i*	
073*i*02	SM S*i*	Load semaphores from S*i*	
073*ij*3	ST*j* S*i*	Transmit (S*i*) to ST*j*	
074*ijk*	S*i* T*jk*	Transmit (T*jk*) to S*i*	
075*ijk*	T*jk* S*i*	Transmit (S*i*) to T*jk*	
076*ijk*	S*i* V*j*,A*k*	Transmit (V*j* element (A*k*)) to S*i*	
077*ijk*	V*i*,A*k* S*j*	Transmit (S*j*) to V*i* element (A*k*)	
077*i*0*k*	V*i*,A*k* 0	Clear element (A*k*) or register V*i*	
060*ijk*	S*i* S*j* + S*k*	Transmit integer sum of (S*j*) and (S*k*) to S*i*	Integer operation
060*i*0*k*	S*i* S*k*	Transmit (S*k*) to S*i*	
060*ij*0	S*i* S*j* + S0	Transmit integer sum of 2^{63} and (S*j*) to S*i*	
061*ijk*	S*i* S*j* -S*k*	Transmit integer difference of (S*j*) and (S*k*) to S*i*	
061*i*0*k*	S*i* -S*k*	Transmit the negative of (S*k*) to S*i*	
061*ij*0	S*i* S*j*-S0	Transmit integer difference of (S*j*) less 2^{63} to S*i*	
154*ijk*	V*i* S*j* + V*k*	Transmit integer sum of (S*j*) and (V*k*) to V*i*	
156*ijk*	V*i* S*j*-V*k*	Transmit integer difference of (S*j*) and (V*k*) to V*i*	
166*ijk*	V*i* S*j**V*k*	Transmit 32-bit integer product of (S*j*) and (V*k* elements) to V*i* elements	

Table 27. S Register Instructions (continued)

Machine Instructions	CAL Syntax	Description	Type of Instruction
062*ijk*	S*i* S*j* + FS*k*	Transmit floating-point sum of (S*j*) and (S*k*) to S*i*	Floating-point operation
062*i*0*k*	S*i* + FS*k*	Transmit normalized (S*k*) to S*i*	
063*ijk*	S*i* S*j*-FS*k*	Transmit floating-point difference of (S*j*) and (S*k*) to S*i*	
063*i*0*k*	S*i* -FS*k*	Transmit normalized negative of (S*k*) to S*i*	
064*ijk*	S*i* S*j* *FS*k*	Transmit floating-point product of (S*j*) and (S*k*) to S*i*	
065*ijk*	S*i* S*j* *HS*k*	Transmit half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*	
066*ijk*	S*i* S*j* *RS*k*	Transmit rounded floating-point product of (S*j*) and (S*k*) to S*i*	
067*ijk*	S*i* S*j* *lS*k*	Transmit two minus the floating-point product of (S*j*) and (S*k*) to S*i*	
070*ij*0	S*i* /HS*j*	Transmit floating-point reciprocal approximation of (S*j*) to S*i*	
071*i*0*k*	S*i* A*k*	Transmit (A*k*) to S*i* with no sign extension	
071*i*1*k*	S*i* +A*k*	Transmit (A*k*) to S*i* with sign extension	
071*i*2*k*	S*i* +FA*k*	Transmit (A*k*) to S*i* as unnormalized floating-point number	
160*ijk*	V*i* S*j* *FV*k*	Transmit floating-point products of (S*j*) and (V*k* elements) to V*i* elements	
162*ijk*	V*i* S*j* *HV*k*	Transmit half-precision rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	
164*ijk*	V*i* S*j* *RV*k*	Transmit rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	
170*ijk*	V*i* S*j* +FV*k*	Transmit floating-point sums of (S*j*) and (V*k* elements) to V*i* elements	
172*ijk*	V*i* S*j*-FV*k*	Transmit floating-point differences of (S*j*) and (V*k* elements) to V*i* elements	

Table 27. S Register Instructions (continued)

Machine Instructions	CAL Syntax	Description	Type of Instruction
044*ijk*	S*i* S*j*&S*k*	Transmit logical product of (S*j*) and (S*k*) to S*i*	Logical operation
044*ij*0	S*i* S*j*&SB	Transmit sign bit of (S*j*) to S*i*	
044*ij*0	S*i* SB&S*j*	Transmit sign bit of (S*j*) to S*i* (*j* ≠ 0)	
045*ijk*	S*i* #S*k*&S*j*	Transmit logical product of (S*j*) and complement of (S*k*) to S*i*	
045*ij*0	S*i* #SB&S*j*	Transmit (S*j*) with sign bit cleared to S*i*	
046*ijk*	S*i* S*j*\S*k*	Transmit logical difference of (S*j*) and (S*k*) to S*i*	
046*ij*0	S*i* S*j*\SB	Toggle sign bit of (S*j*), then transmit to S*i* (*j* ≠ 0)	
046*ij*0	S*i* SB\S*j*	Toggle sign bit of (S*j*), then transmit to S*i* (*j* ≠ 0)	
047*ijk*	S*i* #S*j*\S*k*	Transmit logical equivalence of (S*k*) and (S*j*) to S*i*	
047*ij*0	S*i* #S*j*\SB	Transmit logical equivalence of (S*j*) and sign bit to S*i*	
047*ij*0	S*i* #SB\S*j*	Transmit logical equivalence of (S*j*) and sign bit to S*i* (*j* ≠ 0)	
050*ijk*	S*i* S*j*!S*i*&S*k*	Transmit logical product of [(S*i*) and (S*k*) complement] ORed with logical product of [(S*j*) and (S*k*)] to S*i* (scalar merge)	
050*ij*0	S*i* S*j*!S*i*&SB	Transmit scalar merge of (S*i*) and sign bit of (S*j*) to S*i*	
051*ijk*	S*i* S*j*!S*k*	Transmit logical sum of (S*j*) and (S*k*) to S*i*	
051*ij*0	S*i* S*j*!SB	Transmit logical sum of (S*j*) and sign bit to S*i*	
051*ij*0	S*i* SB!S*j*	Transmit logical sum of (S*j*) and sign bit to S*i* (*j* ≠ 0)	
140*ijk*	V*i* S*j*&V*k*	Transmit logical product of (S*j*) and (V*k* elements) to V*i* elements	
142*ijk*	V*i* S*j*!V*k*	Transmit logical sum of (S*j*) and (V*k* elements) to V*i* elements	
144*ijk*	V*i* S*j*\V*k*	Transmit logical difference of (S*j*) and (V*k* elements) to V*i* elements	
146*ijk*	V*i*! S*j*V*k*&VM	Transmit (S*j*) if VM bit = 1, or (V*k*) if VM bit = 0 to V*i*	

Table 27. S Register Instructions (continued)

Machine Instructions	CAL Syntax	Description	Type of Instruction
052*ijk*	S0 S*i* < *exp*	Shift (S*i*) left *exp* places to S0; *exp* = *jk*	Register shift
053*ijk*	S0 S*i* > *exp*	Shift (S*i*) right *exp* places to S0; *exp* =100_8 - *jk*	
054*ijk*	S*i* S*i* < *exp*	Shift (S*i*) left *exp* places to S*i*; *exp* = *jk*	
055*ijk*	S*i* S*i* > *exp*	Shift (S*i*) right *exp* places to S*i*; *exp* = 100_8 -*jk*	
056*ijk*	S*i* S*i*, S*j* < A*k*	Shift (S*i*) and (S*j*) left by (A*k*) places to S*i*	
056*ij*0	S*i* S*i*, S*j* < 1	Shift (S*i*) and (S*j*) left one place to S*i*	
056*i*0*k*	S*i* S*i* < A*k*	Shift (S*i*) left (A*k*) places to S*i*	
057*ijk*	S*i* S*j*,S*i* > A*k*	Shift (S*j*) and (S*i*) right by (A*k*) places to S*i*	
057*ij*0	S*i* S*j*,S*i* > 1	Shift (S*j*) and (S*i*) right by one place to S*i*	
057*i*0*k*	S*i* S*i* > A*k*	Shift (S*i*) right (A*k*) places to S*i*	
014*ijkm*	JSZ *exp*	Jump to *exp* if (S0) = 0 (*i* bit 2 = 0) *exp* = *ijkm*	Conditional jump
015*ijkm*	JSN *exp*	Jump to *exp* if (S0) ≠ 0 (*i* bit 2 = 0) *exp* = *ijkm*	
016*ijkm*	JSP *exp*	Jump to *exp* if (S0) ≥ (*i* bit 2 = 0) *exp* = *ijkm*	
017*ijkm*	JSM *exp*	Jump to *exp* if (S0) < 0 (*i* bit 2 = 0) *exp* = *ijkm*	
070*ij*6	S*i* S*j**BT	Transmit bit-matrix product of (S*j*) and transpose of (BMM) to S*i*	Bit-matrix multiply

Intermediate Scalar (T) Registers

Sixty-four 64-bit T registers are designated T0 through T77 octal. The T registers are used as intermediate storage registers for the S registers. Data transfers between T and S registers and between T registers and central memory. A data transfer between a T register and an S register takes 1 CP.

Instructions reference T registers by specifying the T register number in the *jk* designator. Refer to "Instruction Formats" for more information on instruction fields.

A block of T registers transfers to or from cache or central memory at a maximum rate of two 64-bit register locations per CP. The *jk* fields of the instruction specify the first T register that is used in the block transfer; the low-order 7 bits of the contents of register A*i* specify the number of words that are transmitted. Successive transfers involve successive T register pairs until T76/T77 is reached. Registers T00/T01 are processed after register T76/T77 if the content of register A*i* is not equal to zero. Other instructions can issue while a block of T registers is transferred to or from cache or central memory. During these block transfers, a reservation is made on all T registers that are used in the block transfer. Table 28 summarizes the T register instructions.

There are 64 T registers, but the length of the T register load or store operation as specified by the 7-bit length field in register A*i* can be greater than 64 (up to 127). For a load operation with a length greater than 64, the T register load control operates for length divided by 2 cycles, but it actually sends load requests to cache/memory for the last 64 operands. For a store operation with a length greater than 64, the T register store control operates for length divided by 2 cycles and sends data to cache/memory for every cycle.

Table 28. T Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
074*ijk*	S*i* T*jk*	Transmit (T*jk*) to S*i*	Interregister transfer
075*ijk*	T*jk* S*i*	Transmit (S*i*) to T*jk*	
036*ijk*	T*jk*,A*i* ,A0	Load (A*i*) words from memory starting at (A0) to T registers starting at *jk*	Block transfer
036*ijk*	T*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at (A0) to T registers starting at *jk*	
037*ijk*	,A0 *Tjk*,A*i*	Store (A*i*) words from T registers starting at *jk* to memory starting at (A0)	
037*ijk*	0,A0 T*jk*,A*i*	Store (A*i*) words from T registers starting at *jk* to memory starting at (A0)	

Vector (V) Registers

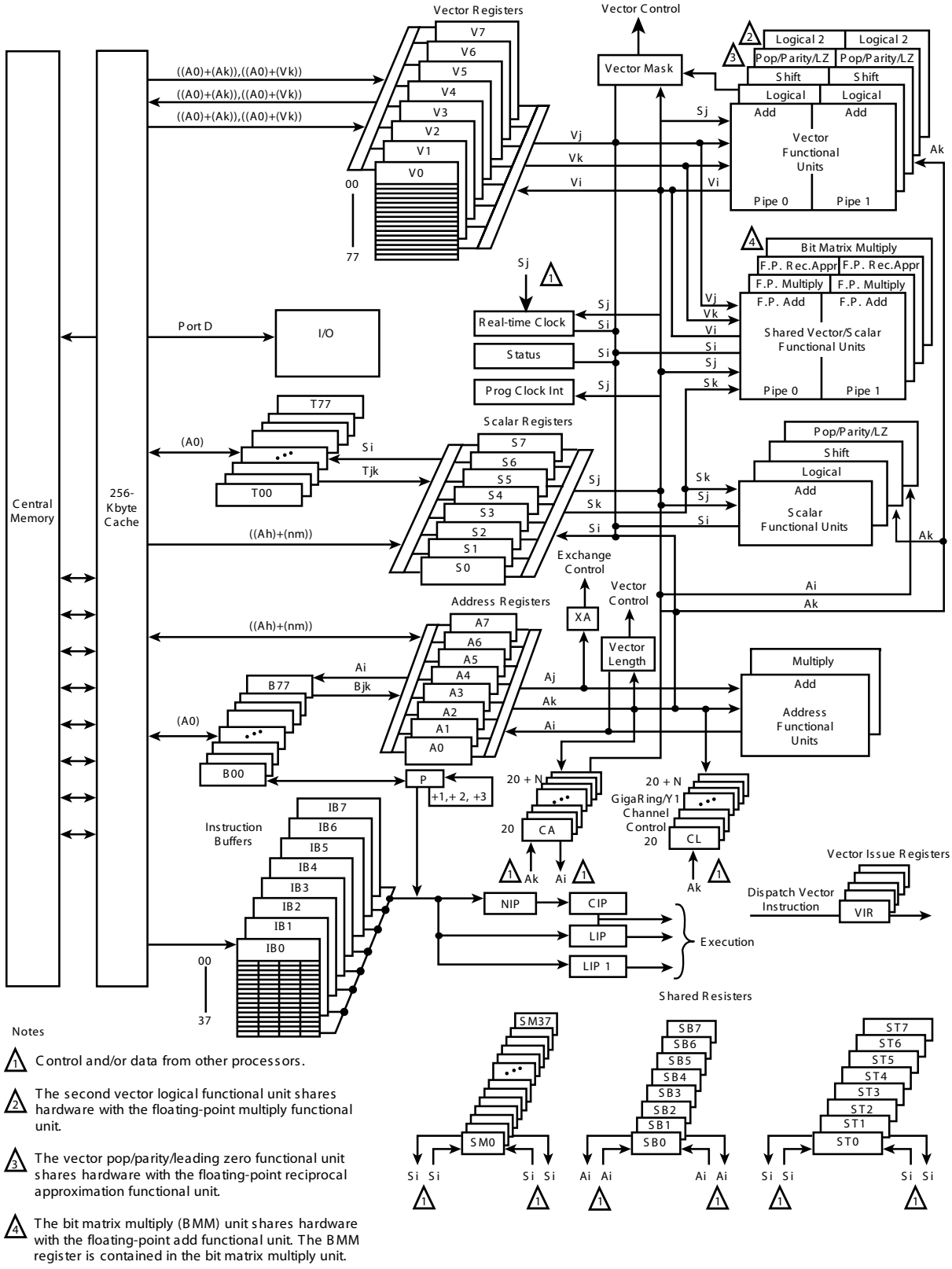
Figure 29 shows the eight V registers and their associated hardware. The V registers are designated V0 through V7. Each V register has 64 elements that are 64 bits wide. The V registers are used for vector processing. The following subsections explain vector processing, the V register functions, the V register instructions, and vector chaining and tailgating.

Vector Processing

Vector processing increases processing speed and efficiency by allowing an operation to be performed sequentially on a set (or vector) of operands by using a single instruction.

A vector is an ordered set of elements; each element is represented as a 64-bit word. A vector is distinguished from a scalar, which is a single 64-bit word. Examples of structures in Fortran that can be represented as vectors are one-dimensional arrays and rows, columns, and diagonals of multidimensional arrays. Vector processing occurs when arithmetic or logical operations are applied to vectors; it differs from scalar processing because it operates on many elements rather than on one.

Figure 29. V Register Block Diagram



Notes

1. Control and/or data from other processors.
2. The second vector logical functional unit shares hardware with the floating-point multiply functional unit.
3. The vector pop/parity/leading zero functional unit shares hardware with the floating-point reciprocal approximation functional unit.
4. The bit matrix multiply (BMM) unit shares hardware with the floating-point add functional unit. The BMM register is contained in the bit matrix multiply unit.

In vector processing, successive elements are provided each CP; as each operation is completed, the result is delivered to a successive element of the result register. The vector operation continues until the number of operations performed by the instructions equals the count specified by the vector length (VL) register.

Parallel vector operations allow the generation of two or more results per CP. Parallel vector operations can be processed by the following methods:

- Using different functional units and different V registers.
- Using the result stream from one vector register as the operand of another operation using a different functional unit; this process is known as *chaining* and is explained later in this subsection.
- Using one V register, from which operands are currently being read and sent to one functional unit, as the result register for the result stream from a second function. This process is known as tailgating and is explained later in this subsection. To tailgate, the operand processing for the first instruction must always remain ahead of the second instruction operand processing.

Vector Dual Pipe Processing

The SV1 series CPU is designed for dual pipe vector processing. There are two of each type of vector functional units: one set is designated as Vector Pipe 0 and the second as Vector Pipe 1. The parallel use of Pipes 0 and 1 reduces by half the number of CPs required to process the same set of vector operands (producing 2 results per CP).

The vector registers, to support the dual functional unit pipes, are each divided into two parts with all even elements in one part and all odd elements in the other. As an example, Vector register 0 elements 0, 2, 4, 6, and so on are sent as operands to and received as results from Pipe 0 functional units, while elements 1, 3, 5, 7, and so on are sent to and received from Pipe 1 functional units. There are 8 vector registers, each with a length of 64 elements, but each is divided into paired halves to support a duplicated set of vector functional units. The two halves are controlled by a common control unit.

Advantages of Vector Processing

In general, vector processing is faster and more efficient than scalar processing. Vector processing reduces the overhead that is associated with maintenance of the loop-control variable (for example, incrementing and

checking the count). In many cases, loops that process vectors are reduced to a simple sequence of instructions without branching backwards. Vector processing reduces central memory access conflicts. It also exploits functional unit segmentation processing because results from the units can be obtained at the rate of two results per CP with the SV1 series systems.

Vectorization typically speeds up a code segment by approximately a factor of ten. If a segment of code that previously used 50% of a program's run time is vectorized, the overall run time is 55% of the original run time (50% for the unvectorized portion plus 0.1 X 50% for the vectorized portion). Vectorizing 90% of a program reduces the run time to 19% of the original execution time.

V Register Functions

The V registers are used for vector processing. Unlike the A and S registers that have secondary functions, the V registers are used only for vector processing. Vector processing allows a single instruction to sequentially perform a specified operation on a set (vector) of operands, to produce a series of results. Examples of these sets or vectors may be rows or columns of a matrix or elements of a table.

Vector instructions reference V registers by specifying the register number as the *i*, *j*, or *k* designator. Refer to "Instruction Formats" for information about instruction fields. Vector registers always start with element 0. Individual elements of a V register are designated by octal numbers that range from 00 through 77. These numbers appear as subscripts to vector register references. For example, $V6_{27}$ refers to element 27 (octal) of V register 6.

Single-word data transfers can be made between an S register and an element of a V register. In block transfers, the contents of a V register are transferred to or from cache or central memory by specifying a first word address in central memory, an increment or decrement value for the central memory address, and a vector length. The transfer begins with the first element of the V register at a maximum rate of 2 words per CPU CP; this rate can be affected by central memory conflicts. A central memory conflict interrupts the vector data stream and can occur in chained operations (although they do not inhibit chaining). Any interruption in the vector data stream adds proportionally to the total execution time of vector operations.

Vector Instructions

All vector instructions are dispatched to the vector unit, which finally issues these instructions after conflict checks. Vector instructions reserve V registers as either operands or results. If the register is reserved as an operand register, it cannot be used as an operand register until the operand reservation clears. A

vector register can be used as both an operand and result register for the same vector instruction. If a register is reserved as a result register, it can be used as an operand register through a process called *chaining*. Refer to "Vector Chaining" for more information on chaining. A register that is reserved as an operand may be used as a result register by a later instruction, if conditions permit, through a process called *tailgating*.

No reservation is placed on the VL register during vector processing. If a vector instruction uses an S register as an operand, no reservation is placed on the S register. Conflicts can occur between vector and scalar operations that access memory. With the exception of these operations, the functional units are always available for scalar operations. The S and VL registers can be modified after the vector instruction issues without affecting the earlier vector operations. The A0 and A*k* registers in a vector memory reference can also be modified after the instruction issues from the current instruction parcel (CIP).

Instructions reference V registers by specifying the register number as the *i*, *j*, or *k* designator. Refer to "Instruction Formats" for more information about the instruction fields. Because most transfers to or from registers are done in blocks of data, instructions that transfer data between V registers and central memory reserve a port, and functional unit instructions reserve the appropriate functional unit.

Table 29 summarizes the types of V register instructions and provides the machine instruction, the CAL code, a description of the instruction, and the type of instruction. Included in this table are instructions that are not directly vector but that support vector operations. Refer to "CPU Instruction Descriptions" for a detailed description of these instructions.

Table 29. V Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
00200*k*	VL A*k*	Transmit (A*k*) to Vector Length register	Register entry
002000	VL 1	Transmit 1 to Vector Length register	
0030*j*0	VM S*j*	Transmit (S*j*) to the Vector Mask register	
003000	VM 0	Clear Vector Mask register	
076*ijk*	S*i* V*j*,A*k*	Transmit (V*j* element (A*k*)) to S*i*	
077*ijk*	V*i*,A*k* S*j*	Transmit (S*j*) to V*i* element (A*k*)	
077*i*0*k*	V*i*,A*k* 0	Clear element (A*k*) of register V*i*	

Table 29. V Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
176*i*0*k*	V*i* ,A0,A*k*	Load (VL) words from address (A0) incremented by (A*k*) elements to V*i* elements	Memory transfer (Load)
176*i*00	V*i* ,A0,1	Load (VL) words from address (A0) incremented by 1 to V*i* elements	
176*i*1*k*	V*i* ,A0,V*k*	Load (VL) words from address ((A0) + (V*k* elements)) to V*i* elements	
1770*jk*	,A0,A*k* V*j*	Store (VL) words from V*j* elements to address (A0) incremented by (A*k*)	Memory transfer (Store)
1770*j*0	,A0,1 V*j*	Store (VL) words from V*j* elements to address A0 incremented by 1	
1771*jk*	,A0,V*k* V*j*	Store (VL) words from V*j* elements to address ((A0) + (V*k* elements))	
154*ijk*	V*i* S*j* + V*k*	Transmit integer sums of (S*j*) and (V*k* elements) to V*i* elements	Integer operation
155*ijk*	V*i* V*j* + V*k*	Transmit integer sums of (V*j* elements) and (V*k* elements) to V*i*	
156*ijk*	V*i* S*j*-V*k*	Transmit integer differences of (S*j*) and (V*k* elements) to V*i* elements	
156*i*0*k*	V*i* -V*k*	Transmit twos complement of (V*k* elements) to V*i* elements	
157*ijk*	V*i* V*j*-V*k*	Transmit integer differences of (V*j* elements) and (V*k* elements) to V*i* elements	
160*ijk*	V*i* S*j**FV*k*	Transmit floating-point products of (S*j*) and (V*k* elements) to V*i* elements	Floating-point operation
161*ijk*	V*i* V*j**FV*k*	Transmit floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	
162*ijk*	V*i* S*j**HV*k*	Transmit half-precision rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	
163*ijk*	V*i* V*j**HV*k*	Transmit half-precision rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	
164*ijk*	V*i* S*j**RV*k*	Transmit rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	
165*ijk*	V*i* V*j**RV*k*	Transmit rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	
166*ijk*	V*i* S*j** V*k*	Transmit 32-bit integer product of (S*j*) and (V*k* elements) to V*i* elements	
167*ijk*	V*i* V*j**IV*k*	Transmit reciprocal iteration of two minus the floating-point product of (V*j* elements) and (V*k* elements) to V*i* elements	
170*ijk*	V*i* S*j* + FV*k*	Transmit floating-point sums of (S*j*) and (V*k* elements) to V*i* elements	

Table 29. V Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
170*i*0*k*	V*i* +FV*k*	Transmit normalized (V*k* elements) to V*i* elements	Floating-point operation (cont.)
171*ijk*	V*i* V*j* +FV*k*	Transmit floating-point sums of (V*j*) and (V*k* elements) to V*i* elements	
172*ijk*	V*i* S*j*-FV*k*	Transmit floating-point differences of (S*j*) and (V*k* elements) to V*i* elements	
172*i*0*k*	V*i* -FV*k*	Transmit normalized negative of (V*k* elements) to V*i* elements	
173*ijk*	V*i* V*j*-FV*k*	Transmit floating-point differences of (V*j* elements) and (V*k* elements) to V*i* elements	
174*ij*0	V*i* /HV*j*	Transmit floating-point reciprocal approximation of (V*j* elements) to V*i* elements	
140*ijk*	V*i* S*j*&V*k*	Transmit logical product of (S*j*) and (V*k* elements) to V*i* elements	Logical operation
141*ijk*	V*i* V*j*&V*k*	Transmit logical products of (V*j* elements) and (V*k* elements) to V*i* elements	
142*ijk*	V*i* S*j*!V*k*	Transmit logical sums of (S*j*) and (V*k* elements) to V*i* elements	
142*i*0*k*	V*i* V*k*	Transmit (V*k* elements) to V*i* elements	
143*ijk*	V*i* V*j*!V*k*	Transmit logical sums of (V*j* elements) and (V*k* elements) to V*i* elements	
144*ijk*	V*i* S*j* \ V*k*	Transmit logical differences of (S*j*) and (V*k* elements) to V*i* elements	
145*ijk*	V*i* V*j* \ V*k*	Transmit logical differences of (V*j* elements) and (V*k* elements) to V*i* elements	
146*ijk*	V*i* S*j* !V*k*&VM	Transmit (S*j*) if VM bit = 1, (V*k* element) if VM bit = 0, to V*i* elements	
146*i*0*k*	V*i* #VM&V*k*	Vector merge of (V*k* elements) and 0 to V*i* elements	
147*ijk*	V*i* V*j* !V*k*&VM	Transmit (V*j* element) if VM bit =1, (V*k* element) if VM bit = 0, to V*i* elements	
150*ijk*	V*i* V*j* < A*k*	Shift (V*j* elements) left by (A*k*) places to V*i* elements	Register shift
150*ij*0	V*i* V*j* < 1	Shift (V*j* elements) left one place to V*i* elements	
151*ijk*	V*i* V*j* > A*k*	Shift (V*j* elements) right by (A*k*) places to V*i* elements	
151*ij*0	V*i* V*j* > 1	Shift (V*j* elements) right one place to V*i* elements	
152*ijk*	V*i* V*j*,V*j* < A*k*	Transmit double shift of (V*j* elements) left (A*k*) places to V*i* elements	
152*ij*0	V*i* V*j*,V>1	Transmit double shift (V*j* elements) left one place to V*i* elements	
153*ijk*	V*i* V*j*,V*j* > A*k*	Transmit double shift of (V*j* elements) right (A*k*) places to V*i* elements	
153*ij*0	V*i* V*j*,V*j* > l	Transmit double shift of (V*j* elements) right one place to V*i* elements	

Table 29. V Register Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
174*ij*1	V*i* PV*j*	Transmit population count of (V*j* elements) to V*i* elements	Pop/Parity leading 0 operating
174*ij*2	V*i* QV*j*	Transmit population count parity of (V*j* elements) to V*i* elements	
174*ij*3	V*i* ZV*j*	Transmit the leading zero count of (V*j* elements) to V*i* elements	
1740*j*4	BMM V*j*	Transmit (V*j* elements) to BMM	Bit matrix multiply operation
174*ij*6	V*i* V*j**BT	Transmit the bit matrix product of (V*j* elements) and transpose of (BMM) to V*i* elements	
070*ij*6	S*i* S*j**BT	Transmit bit matrix product of (S*j*) and transpose of (BMM) to S*i* (This scalar instruction executes as a vector operation.)	

Vector Instruction Issue Timing

The CIP is the central issue point for all instructions. Instructions that require use of the vector unit are issued to the vector unit instruction queue. Table 29 lists all of these instructions. The vector issue register (VIR) issues these vector instructions in the order it receives them. The vector unit instruction queue (VIQ) can buffer a maximum of five instructions issued to it by the CIP. CIP issue of any additional vector instructions must wait until the queue count is less than five.

The CIP issues vector instructions to the VIR instruction queue without checking for a vector functional unit conflict or vector register busy condition. These conflicts delay issue of the instruction from the VIR.

Vector instruction issue timing has two categories:

- Issue from the CIP directly to the appropriate vector unit, via the VIR, when no conflicts exist to delay issue
- Issue from the VIR after a delay caused by vector register conflicts, functional unit conflicts, or vector instruction queuing

The execution time for vector instructions that issue directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue from the VIR.

Vector Instruction Issue Conflict Timing

The general rules that apply to the next vector instruction (NVI) issue from the VIR are as follows:

For Functional Unit Busy

- Functional unit is ready in (VL/2) + 1 CP. The exceptions are a BMM load instruction, 1740*j*4, which is ready in 8, 16, 24, or 32 CPs +2 CPs, and the logical unit that executes a 175 instruction to be followed by a 140 through 147 instruction and ready in (VL/2) + 2 CPs.

For Vector Register Busy

- V*i* is ready for V*i* use in (VL/2) + functional unit depth in CPs + 2 CPs.
- V*i* is ready for V*j* or V*k* use immediately (due to chaining).
- V*j* or V*k* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* or V*k* is ready for V*i* use in (VL/2) + 2 CPs, or V*j* or V*k* for V*i* use immediately when V*j* or V*k* is not involved in chaining (when tailgating is permitted).

Note: Chaining cannot occur unless the data is already available in V*i*, and tailgating cannot occur unless those V*i* locations that are to be written have already read those data elements from V*j* or V*k*.

Vector Chaining

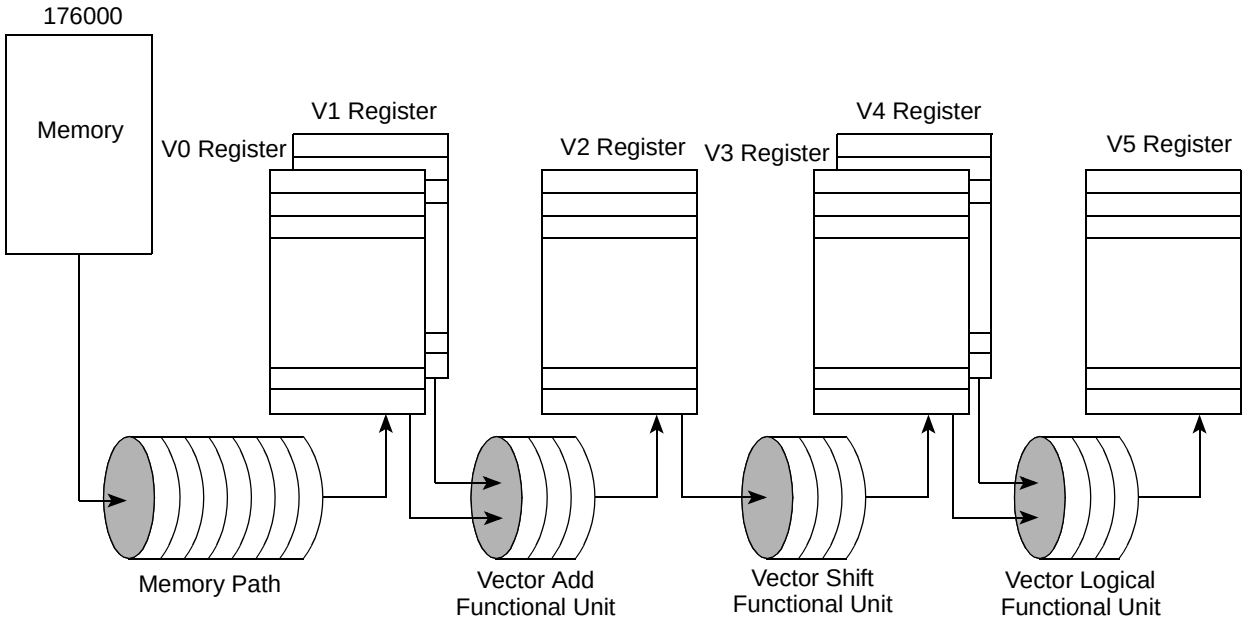
A vector register that is reserved for results can become the operand register of a succeeding instruction. This process, called chaining, allows a continuous stream of operands to flow through the vector registers and functional units. Even when a vector load operation pauses because of memory conflicts, chained operations may proceed as soon as data is available. A vector register can be read during the same cycle that it is written into.

This chaining mechanism allows chaining to begin at any point in the result vector data stream. The amount of concurrency in a chained operation depends on the relationship between the issue time of the chaining instruction and arrival time of the result data stream. For full chaining to occur, the chaining instruction must issue and be ready to use elements 0 and 1 of the result at the same time that elements 0 and 1 arrive at the V register. Partial chaining occurs if the chaining instruction issues after the arrival of elements 0 and 1 of the result vector data stream.

Elements are loaded into register V0. As soon as elements 0 and 1 have arrived from central memory into register V0, they are added to the first element pair of vector register V1. Subsequent elements are pipelined through the segmented functional unit, so that a continuous stream of results is sent to the destination register, which is register V2. As soon as the first pair of elements arrives at register V2, it becomes the operand pair for the shift operation. The results are sent to register V3, which immediately becomes the source of one of the operand pairs necessary for the logical operation between registers V3 and V4. The results of the logical operation are then sent to register V5.

Figure 30 shows how the results of four instructions are chained together. The instruction chaining sequence performs the following operations:

Figure 30. Vector Chaining Example



1. Read a vector of integers from central memory to register V0 (176000 instruction).

2. Add the contents of register V0 to the contents of register V1 and send the results to V2 (155210 instruction).

3. Shift the results obtained in Step 2 and send the results to register V3 (150327 instruction).

4. Form the logical product of the shifted sum obtained in Step 3 with the contents of register V4 and send the results to register V5 (141543 instruction).

Elements are loaded into register V0. As soon as individual elements 0 and 1 arrive from central memory into register V0, they are added to the first element pair of vector register V1. Subsequent element pairs are pipelined through the segmented functional units, so that a continuous stream of results is sent to the destination register V2. As soon as the first result element pair arrives at register V2, it becomes the operand pair for the shift operation. The results are sent to register V3, which immediately becomes the source of one of the operand pairs necessary for the logical operation between registers V3 and V4. The results of the logical operation are then sent to register V5.

Vector Tailgating

The mainframe design also incorporates vector register tailgating. Tailgating is the process of writing the result from a later vector instruction into a vector register after that location has been read by an earlier vector instruction. The SV1 series system permits tailgating to any vector register that is not already reserved as a result register. The tailgating control ensures that the read of a vector register location always occurs before the write to that vector register location can occur. Refer to Figure 31 for a vector tailgating example.

Figure 31. Vector Tailgating Example

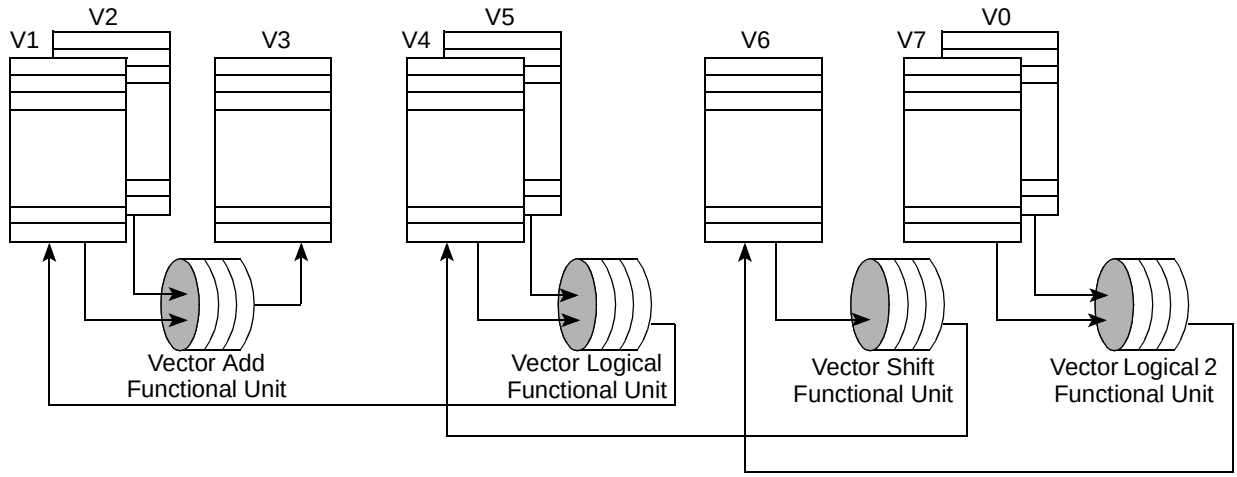


Figure 31 shows the results of three instructions that are tailgating their results into a previously issued instruction. The instruction sequence performs the following operation:

1. Vector integer add (155321 instruction) of V1 and V2 to V3 starts the sequence.

2. Vector logical (141145 instruction) of V4 and V5 puts the results into V1. The tailgating control assures that the read of V1 will remain ahead of the write to V1.

3. Vector shift (150465 instruction) of V6 puts the results into V4.

4. Vector logical 2 (141670 instruction) of V7 and V0 puts the results into V6.

Vector Control Registers

The vector length (VL) register and vector mask (VM) register provide control information that is needed to perform vector operations. The following subsections describe the VL and VM registers. Table 30 lists the vector mask instructions and provides octal and CAL codes. Refer to the "Functional Units Instruction Summary" for complete information on these instructions.

Table 30. Vector Mask Instructions

Machine Instructions	CAL Syntax	Description	Type of Instruction
0030*j*0	VM S*j*	Transmit (S*j*) to VM register	Register entry
003000	VM 0	Clear VM register	
073*i*00	S*i* VM	Transmit (VM) to S*i*	
146*ijk*	V*i* S*j*!V*k*&VM	Transmit to V*i* elements (S*j*) if VM bit = 1 or (V*k* element) if VM bit = 0	Logical operation
146*i*0*k*	V*i* #VM&V*k*	Transmit vector merge of (V*k* elements) and 0 to V*i* elements	
147*ijk*	V*i* V*j*!V*k*&VM	Transmit to V*i* elements (V*j* element) if VM bit = 1 or (V*k*) if VM bit = 0	
1750*j*0	VM V*j*, Z	Set VM = 1, if (V*j* element) = 0	
1750*j*1	VM V*j*, N	Set VM = 1, if (V*j* element) ≠ 0	
1750*j*2	VM V*j*, P	Set VM = 1, if (V*j* element) ≥ 0 (positive)	
1750*j*3	VM V*j*, M	Set VM = 1, if (V*j* element) < 0 (negative)	
175*ij*4	V*i*,VM V*j*, Z	Set VM bit = 1, if (V*j* element) = 0, and store the compressed indices of the V*j* elements = 0 in V*i*.	
175*ij*5	V*i*,VM V*j*, N	Set VM bit = 1, if (V*j* element) ≠ 0, and store the compressed indices of the V*j* elements ≠ 0 in V*i*.	
175*ij*6	V*i*,VM V*j*, P	Set VM bit = 1, if (V*j* element) ≥ 0, and store the compressed indices of the V*j* elements ≥ 0 in V*i*.	
175*ij*7	V*i*,VM V*j*, M	Set VM bit = 1, if (V*j* elements) < 0, and store the compressed indices of the V*j* elements < 0 in V*i*.	

Vector Length Register

The 7-bit VL register is set from 1 through 100 octal (VL = 0 gives VL = 100) to specify the length of all vector operations performed by vector instructions and the length of the vectors held by the V registers. The VL register controls the number of operations performed by instructions 140 through 177. The VL register is loaded and its contents are saved by an exchange sequence. The VL register is set by instruction 00200*k* and is read by instruction 023*i*01.

Vector Mask Register

The VM register has 64 bits; each bit corresponds to a word element in a vector register. Bit 63 corresponds to element 0, and bit 0 corresponds to element 63. The mask is used with vector merge and test instructions to allow operations to be performed on individual vector elements.

The VM register can be set from an S register through instruction 003 or can be created by testing a vector register for a condition using instruction 175. The mask controls element selection in the vector merge instructions (146 and 147). Instruction 073 reads the contents of the VM register to an S register.

User Mode Vector Instruction Timing

Table 31 describes the user mode vector instruction issue and execution information. The following definitions apply to Table 31.

- The current instruction parcel (CIP) is the same instruction issue register that was used in previous Cray designs.
- The CIP dispatches vector instructions to the vector instruction queue (VIQ). The VIQ can hold up to five vector instructions.
- The vector issue register (VIR) is the fifth stage of the VIQ. The VIR issues vector instructions in the order in which they are received from the CIP. The VIR checks for the “busy” conditions in the vector unit that hold issue in the VIR. The CIP does not monitor these “vector unit only” busy conditions.
- The timings shown are relative to a vector instruction issued from the VIR. An instruction issued from the CIP through a nonbusy VIR is 2 CPs longer.

Table 31. Vector Instruction Issue and Execution

Code	CAL	CIP Hold Issue Conditions					VIR Hold Issue Conditions					FCN Unit Busy Time	Used Vj/Vk Ready	Result in VR**	Comments and Special Conditions
		Ak Busy (Except A0)	Sj Busy (Except S0)	035/037 Instr in Progress	077 Issued Previous CP	VIQ Full	VR Busy* Vi	VR Busy* Vj	VR Busy* Vk	FCN Units Busy	Instr Not in VIR				
00200*k*	VL A*k*	x			x	x		x			x				VIR issue 3 CP
003C*j*0	VM S*j*		x		x	x				Log	x	1 CP		5CP in VM	VM busy for 3 CP
073*i*00	S*i* VM		S*i* Busy			x	x			VM	x	2 CP		3 CP in S*i*	VIR issue 2 CP
070*ij*6	S*i* S*j**BT		S*i*/S*j* Busy		x	x				BMM	x	1 CP			Scalar BMM - see note 1
076*ijk*	S*i* V*j*,A*k*	x	S*i* Busy		x	x		x			x		2 CP	5 CP in S*i*	VIR issue 2 CP. Refer to note 2.
077*ijk*	V*i*,A*k* S*j*	x	x		x	x	x				x		1 CP	4 CP	CIP hold issue on gather/scatter
140*ijk*	V*i* S*j*&V*k*		x		x	x	x		x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
141*ijk*	V*i* V*j*&V*k*					x	x	x	x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
142*ijk*	V*i* S*j* |V*k*		x		x	x	x		x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
143*ijk*	V*i* V*j* |V*k*					x	x	x	x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
144*ijk*	V*i* S*j*\V*k*		x		x	x	x		x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
145*ijk*	V*i* V*j*\V*k*					x	x	x	x	Log&FM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	FM/Log 2 unit
146*ijk*	V*i* S*j* |V*k*&VM		x		x	x	x		x	Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	Merge instructions
147*ijk*	V*i* V*j* |V*k*&VM					x	x	x	x	Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	Merge instructions
150*ijk*	V*i* V*j*<A*k*	x			x	x	x	x		Shift	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	
151*ijk*	V*i* V*j*>A*k*	x			x	x	x	x		Shift	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	
152*ijk*	V*i* V*j*,V*j*<A*k*	x			x	x	x	x		Shift	x	VL/2+1 CP	VL/2+2 CP	VL/2+8 CP	
153*ijk*	V*i* V*j*,V*j*>A*k*	x			x	x	x	x		Shift	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	

Table 31. Vector Instruction Issue and Execution (continued)

Code	CAL	CIP Hold Issue Conditions					VIR Hold Issue Conditions					FCN Unit Busy Time	Used *Vj*/*Vk* Ready	Result in VR**	Comments and Special Conditions
							VR Busy*			FCN Units Busy	Instr Not in VIR				
		Ak Busy (Except A0)	*Sj* Busy (Except S0)	035/037 Instr in Progress	077 Issued Previous CP	VIQ Full	*Vi*	*Vj*	*Vk*						
154*ijk*	V*i* S*j*+V*k*		x		x	x	x		x	V Add	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	
155*ijk*	V*i* V*j*+V*k*					x	x	x	x	V Add	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	
156*ijk*	V*i* S*j*-V*k*		x		x	x	x		x	V Add	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	
157*ijk*	V*i* V*j*-V*k*					x	x	x	x	V Add	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	
160*ijk*	V*i* S*j* *FV*k*		x		x	x	x		x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Floating multiply
161*ijk*	V*i* V*j* *FV*k*					x	x	x	x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Floating multiply
162*ijk*	V*i* S*j* *HV*k*		x		x	x	x		x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Half precision
163*ijk*	V*i* V*j* *HV*k*					x	x	x	x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Half precision
164*ijk*	V*i* S*j* *RV*k*		x		x	x	x		x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Rounded floating multiply
165*ijk*	V*i* V*j* *RV*k*					x	x	x	x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	Rounded floating multiply
166*ijk*	V*i* S*j**V*k*		x		x	x	x		x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	32-bit integer
167*ijk*	V*i* V*j* *IV*k*					x	x	x	x	F Mply	x	VL/2+1 CP	VL/2+2 CP	VL/2+11 CP	2 - product
170*ijk*	V*i* S*j* +FV*k*		x		x	x	x		x	F Add&BMM	x	VL/2+1 CP	VL/2+2 CP	VL/2+10 CP	
171*ijk*	V*i* V*j* +FV*k*					x	x	x	x	F Add&BMM	x	VL/2+1 CP	VL/2+2 CP	VL/2+10 CP	
172*ijk*	V*i* S*j* -FV*k*		x		x	x	x		x	F Add&BMM	x	VL/2+1 CP	VL/2+2 CP	VL/2+10 CP	
173*ijk*	V*i* V*j* -FV*k*					x	x	x	x	F Add&BMM	x	VL/2+1 CP	VL/2+2 CP	VL/2+10 CP	
174*ij*0	V*i* /HV*j*					x	x	x		FR/P/P/LZ	x	VL/2+1 CP	VL/2+2 CP	VL/2+18 CP	Floating reciprocal
174*ij*1	V*i* PV*j*					x	x	x		FR/P/P/LZ	x	VL/2+1 CP	VL/2+2 CP	VL/2+ 7 CP	Pop count
174*ij*2	V*i* QV*j*					x	x	x		FR/P/P/LZ	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	Parity
174*ij*3	V*i* ZV*j*					X	X	X		FR/P/P/LZ	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	Leading zero
1740*j*4	BMM V*j*					x		x		F Add&BMM	x	L+2 CP	VL/2+2 CP		L=8, 16, 24, or 32 CPs
174*ij*6	V*i* V*j**BT					x	x	x		F Add&BMM	x	VL/2+1 CP	VL/2+2 CP	VL/2+6 CP	Vector BMM

Table 31. Vector Instruction Issue and Execution (continued)

Code	CAL	CIP Hold Issue Conditions					VIR Hold Issue Conditions					FCN Unit Busy Time	Used V*j*/V*k* Ready	Result in VR**	Comments and Special Conditions
		A*k* Busy (Except A0)	S*j* Busy (Except S0)	035/037 Instr in Progress	077 Issued Previous CP	VIQ Full	VR Busy*			FCN Units Busy	Instr Not in VIR				
							Vi	V*j*	V*k*						
1750*j*0	VM V*j*,Z					x		x		Log	x	VL/2+1 CP	VL/2+2 CP		Test,1/0 to VM
1750*j*1	VM V*j*,N					x		x		Log	x	VL/2+1 CP	VL/2+2 CP		Test,1/0 to VM
1750*j*2	VM V*j*,P					x		x		Log	x	VL/2+1 CP	VL/2+2 CP		Test,1/0 to VM
1750*j*3	VM V*j*,M					x		x		Log	x	VL/2+1 CP	VL/2+2 CP		Test,1/0 to VM
175*ij*4	V*i*,VM V*j*,Z					x	x	x		Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	VM/Compress I
175*ij*5	V*i*,VM V*j*,N					x	x	x		Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	VM/Compress I
175*ij*6	V*i*,VM V*j*,P					x	x	x		Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	VM/Compress I
175*ij*7	V*i*,VM V*j*,M					x	x	x		Log	x	VL/2+1 CP	VL/2+2 CP	VL/2+7 CP	VM/Compress I
176*i*0*k*	V*i*, ,A0,A*k*	x		x	x	x	x				x			VL/2+109 CP VL/2+25 CP	Refer to note 3. Memory/Cache CPs
176*i*1*k*	V*i* ,A0,V*k*			x	x	x	x		x		x		VL/2+2 CP VL/2+2CP	VL/2 +115 CP VL/2+28 CP	Gather. Refer to note 4.
1770*jk*	,A0,A*k* V*j*	x		x	x	x		x			x				Refer to note 5.
1771*jk*	,A0,V*k* V*j*			x	x	x		x	x		x		VL/2+2 CP		Scatter. Refer to note 6.

Notes:

* Vector register busy does not delay issue from VIR if chaining or tailgating is permitted.

** The cycles shown for the “Result in VR” are from VIR instruction issue time until the result data is written into the VR. The result data is available for chaining to the next vector instruction as source data (operands) as soon as it reaches the VR.

1 The 070*ij*6 instruction is the scalar BMM instruction that must execute as a vector instruction (be issued from the VIR) and return the result to an S register.

2 The 076 instruction implemented on SV1 series systems provides effective communication between the vector unit and scalar unit. The instruction is issued to the VIQ without checking for VR busy. The 076 is not executed (data to S*i*) until the V*j* vector register is not busy. Following the 076 instruction with a transfer instruction (the S register to itself) will hold issue at the CIP until the VR is not busy. CIP will hold issue of the next vector instruction using an S or A operand register by 1 CP.

3 CIP hold issue of this vector load instruction occurs when no memory port is available, a vector store operation has been issued with bidirectional mode OFF, or a 076 instruction is waiting execution in the vector queue (essentially bidirectional mode OFF because of a 076 instruction) with a vector (with an additional 5 CPs with chaining/tailgating occurring store active). Also, scalar and vector requests to memory cannot occur at the same time. Port busy for a vector load is VL/2+13 CP minimum.

4 The conditions of number 3 apply. In addition, CIP cannot issue the gather instruction if a scatter instruction has not completed execution. Port busy for a gather is VL + 9 CP with a minimum of 12 CPs.

5 The conditions of number 3 apply. In addition, CIP cannot issue while another store is active. Port busy for a vector store is VL + 7 CP with a minimum of 9 CPs.

6 The conditions of number 3 apply. In addition, CIP cannot issue the scatter instruction if a gather instruction has not completed execution. Port busy for a scatter is VL + 16 CP minimum.

Bit Matrix Multiply (BMM) Register

There is one bit matrix multiply register (BMM). The register holds a square matrix of 64 x 64 bits. The register is internal to the bit matrix multiply functional unit. Refer to the "Bit-matrix Multiply Functional Unit" section for more information.

Functional Units

Functional units perform instructions other than simple transfers or control operations. Functional units have independent logic, except for the reciprocal approximation, vector population count, floating-point multiply, and second vector logical units (described later in this section), which share some logic. All functional units can operate simultaneously. For more information, refer to the "Functional Unit Independence" section.

A functional unit receives operands from registers, performs an operation, and delivers the result to a register after the function is performed. Functional units operate in three-addressing mode, with source and destination addressing limited to register designators.

All functional units perform operations in a fixed amount of time; delays are impossible once the operands are delivered to the unit. The time from delivery of the operands to the functional unit until completion of the calculation is called the *functional unit time* and is measured in CPs.

Functional units are fully segmented. This means a new set of operands for unrelated computation can enter a functional unit in each CP even though the functional unit time can be more than 1 CP. Refer to "Pipelining and Segmentation" for more information about segmentation.

There are four groups of functional units: address, scalar, vector, and floating-point. The address, scalar, and vector functional units operate with one of the primary register types (A, S, and V) to support address, scalar, and vector processing. The floating-point functional units support either scalar or vector operations and accept operands from or deliver results to the S or V registers. In the SV1, the scalar floating-point instructions execute in the vector floating-point functional units. The scalar floating-point instructions "steal" a cycle from any vector instruction that is currently using that functional unit. This may delay that vector instruction execution by 1 CP. For timing purposes, cache or memory can also act as functional units for vector operations.

The following subsections define the function, the functional unit time, and the instructions that each functional unit executes. Refer to the following sections and subsections for additional information on functional units:

1. The "Pipelining and Segmentation" and the "Functional Unit Independence" subsections contain detailed information on functional unit segmentation/independence.

2. The "Functional Unit Operations" subsection contains detailed information on integer arithmetic, floating-point arithmetic, normalized floating-point numbers, floating-point range errors, addition algorithm, multiply algorithm, and the division algorithm.

3. The "CPU Instruction Descriptions" subsection contains detailed information on the instructions and instruction formats.

Address Functional Units

The address functional units operate with 32-bit operands and results.

Address functional units perform integer arithmetic on operands that are obtained from A registers and deliver the results to an A register. The address functional units use two's complement arithmetic.

Address Add Functional Unit

The address add functional unit performs 32-bit integer addition and subtraction. The unit executes instructions 030 (addition) and 031 (subtraction). The subtraction operation uses two's complement arithmetic. The A*k* operand is complemented and then added to the A*j* operand. A 1 is added to the low-order bit position of the result. The address add functional unit does not detect overflow conditions.

The address add functional unit time is 2 CPs. This functional unit time is measured from instruction issue to when the result is available.

Address Multiply Functional Unit

The address multiply functional unit performs 32-bit multiplication. The unit executes instruction 032, which forms a 32-bit integer product from two operands. No rounding is performed. The result consists of the least significant 32 bits of the product. The address multiply functional unit does not detect overflow conditions.

The address multiply functional unit time is 4 CPs. This functional unit time is measured from instruction issue to when the result is available.

Scalar Functional Units

Scalar functional units perform operations on 64-bit operands that are obtained from S registers and usually deliver the 64-bit results to an S register. The exception is the population/parity/leading zero count functional unit that delivers its 7-bit result to an A register.

The following subsections describe the four functional units that are exclusively associated with scalar operations. The scalar floating-point instructions use the vector pipe 0 floating-point units for instruction execution. When a scalar instruction uses a floating-point functional unit, it “steals” a cycle from any vector instruction that is using that unit, or its companion unit. The scalar instruction never checks whether the vector units are busy; it just issues and executes. Refer to “Floating-point Functional Units” for more information about these units.

Scalar Add Functional Unit

The scalar add functional unit performs 64-bit integer addition and subtraction. It executes instructions 060 (addition) and 061 (subtraction). The subtraction operation uses two’s complement arithmetic. The S*k* operand is complemented, and then added to the S*j* operand. A 1 is added to the low-order bit position of the result. The scalar add functional unit does not detect overflow conditions.

The scalar add functional unit time is 2 CPs. This functional unit time is measured from instruction issue to when the result is available.

Scalar Shift Functional Unit

The scalar shift functional unit shifts the entire 64-bit contents of an S register (single shift) or shifts the 128-bit contents of two concatenated S registers (double shift). For a single shift (instructions 052 through 055), the shift count is specified by the *jk* field. For a double shift (instructions 056 and 057), the A*k* register contains the shift count; only the lower 7 bits of the contents of the A*k* register are used. If any bits are set in the upper positions, they cause the result register S*i* to be zeroed out.

All single shifts and some double shifts are end-off with zero fill. A circular shift occurs if the shift count does not exceed 64 and the *i* and *j* designators are equal and nonzero.

Single-shift instructions have a functional unit time of 3 CPs, and double-shift instructions have a functional unit time of 3 CPs. These functional unit times are measured from instruction issue to when the result is available.

Scalar Logical Functional Unit

The scalar logical functional unit performs bit-by-bit manipulations of 64-bit quantities that are obtained from S registers. It executes instructions 042 through 043 (mask) and 044 through 051 (logical operations).

The scalar logical functional unit time is 1 CP. This functional unit time is measured from instruction issue to when the result is available.

Scalar Population/Parity/Leading Zero Functional Unit

This functional unit performs instructions 026 (population count and population count parity) and 027 (leading zero count). Instruction 026*ij*0 counts the number of bits in the S*j* register that have a value of 1 in the operand and returns a 7-bit result to the A*i* register; the maximum count is 100 octal (64 decimal), and the minimum count is 0.

Instruction 026*ij*1 counts the number of bits in the S*j* operand that have a value of 1, but returns only a 1-bit parity count to the A*i* register. If the S*j* operand has an even number of bits set, a 0 is returned to the A*i* register. If the S*j* operand has an odd number of bits set, a 1 is returned to the A*i* register.

The functional unit time for the population count parity is 4 CPs. This functional unit time is measured from instruction issue to when the result is available.

Instruction 027*ij*0 counts the number of 0 bits (left to right) that precede a 1 bit in the operand. For these instructions, the 64-bit operand is obtained from an S register, and the 7-bit result is delivered to an A register.

The functional unit time for the leading zero count is 4 CPs. This functional unit time is measured from instruction issue to when the result is available.

Vector Functional Units

Most vector functional units perform operations on operands that are obtained from one or two vector registers or from a vector and an S register. The shift, bit-matrix multiply, and population/parity/leading-zero functional units, which require only one operand, are exceptions. Results from a vector functional unit

are delivered to a vector register, with the 070*ij*6 instruction the exception. All vector functional units are duplicated and operate in a dual-pipe configuration that produces 2 results per CP.

Successive operand pairs are transmitted each CP to the functional units. The corresponding results emerge from the functional unit *n* CPs later, where *n* is the functional unit time and is constant for a given functional unit. The VL register determines the number of operand pairs to be processed by a functional units in VL/2 CPs.

The functional units that this section describes are associated with vector operations.

Vector Add Functional Unit

The vector add functional unit performs 64-bit integer addition and subtraction for a vector operation and delivers the results to elements of a V register. The unit executes instructions 154 and 155 (addition), and 156 and 157 (subtraction). Instructions 154 and 156 use one scalar register operand. The subtraction operation uses two's complement arithmetic in which the V*k* operand is complemented and then added to the A*j* operand. A 1 is added to the low-order bit position of the result. The vector add functional unit does not detect overflow conditions.

The vector add functional unit time is 2 CPs deep. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

Vector Shift Functional Unit

The dual pipe vector shift functional unit shifts the entire 64-bit contents of a vector register element (single-shift) or the 128-bit value formed from two consecutive elements of a V register (double shift). Shift counts are obtained from an A register and are end-off with zero fill. All shift counts are considered positive unsigned integers. If any bit higher than bit 6 is set, the shifted result is all 0's.

The dual pipe vector shift functional unit executes instructions 150 and 151 (single shift) and instructions 152 and 153 (double shift). The functional unit times are 4 CPs deep for instruction 152(left double shift) and 3 CPs deep for instructions 150, 151, and 153. These times are measured by the number of sequential registers from the input register through the output register in the functional unit.

Full Vector Logical Functional Unit

The full vector logical functional unit performs a bit-by-bit manipulation of the 64-bit quantities for instructions 140 through 147. The full vector logical functional unit also performs the logical operations that are associated with the vector mask (175) instruction.

The full vector logical functional unit time is 3 CPs deep for the 175 instructions and 2 CPs deep for the 140 through 147 instructions. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

Second Vector Logical Functional Unit

The second vector logical functional unit can be enabled or disabled by setting the enable second vector logical (ESVL) bit in the exchange package. When enabled, the second vector logical functional unit performs the same bit-by-bit manipulations of the 64-bit quantities that the full vector logical functional unit performs for instructions 140 through 145. When enabled and available, this logical unit is always selected for use before the full logical unit.

The second vector logical and floating-point multiply functional units cannot be used simultaneously because they share input data paths. In addition, because these two units have shared paths, some codes that rely on floating-point products may run slower if the second vector logical functional unit is enabled. If the floating-point multiply unit is busy and the full vector logical unit is not busy, a vector logical instruction uses the full vector logical functional unit.

The second vector logical functional unit is disabled through software by clearing bit 20 of word 6 in the flag register of the user's exchange package or by clearing bit 43 in the hardware exchange package of word 6. When the second vector logical unit is disabled, all 140 through 145 instructions use the full vector logical unit.

The second vector logical functional unit time is 2 CPs deep. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

Vector Population/Parity/Leading-zero Functional Unit

The vector population/parity/leading-zero functional unit performs population counts, parity counts, and leading-zero counts for vector operations. It executes instructions 174*ij*1 (vector population count), 174*ij*2 (vector population count

parity), and 174*ij*3 (vector leading-zero count). This functional unit shares the input data path with the reciprocal approximation functional unit. Therefore, the second unit is busy while the input data is in use with the first unit.

Instruction 174*ij*1 counts the 1 bits in each element of the V*j* register and returns this number to the V*i* register; the total number of 1 bits is the population count. This population count can be an odd or an even number, as indicated by its low-order bit.

Instruction 174*ij*2 counts the number of 1 bits in each element of the V*j* register and returns the low-order bit of the count as the result parity bit to elements of the V*i* register. The result count in the Vi element is 0 for even parity and 1 for odd parity.

Instruction 174*ij*3 counts the number of 0 bits (left to right) that precede the first 1 bit in the V*j* operands and returns this count as the result to the low order 7 bits in elements of the V*i* register.

The vector population/parity/leading-zero functional unit time is 3CPs deep for population and parity, and 2 CPs deep for leading-zero. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

Floating-point Functional Units

The dual-pipe vector floating-point functional units perform floating-point arithmetic for vector operations, and the pipe 0 floating-point units perform arithmetic for scalar operations. When a scalar instruction executes, operands are obtained from S registers and results are delivered to an S register. When most vector instructions execute, operands are obtained from pairs of V registers, or from an S register and a V register, and results are delivered to a vector V*i* register. When a floating-point functional unit is used for a vector operation, the general description of vector functional units applies.

Note: A scalar floating-point instruction issues from the CIP register and “steals” a cycle from any vector instruction is that currently using that floating-point functional unit. The CIP does check for any return path conflicts to the S registers.

Floating-point Add Functional Unit

The individual floating-point add functional units perform addition and subtraction of 64-bit operands in floating-point format. They execute instructions 062 (scalar add), 063 (scalar subtract), and 170 through 173

(vector add and subtract). A result is normalized even when operands are unnormalized. The floating-point add functional unit detects overflow and underflow conditions; only overflow conditions are flagged.

The scalar floating-point add functional unit time is 8 CPs. This functional unit time is measured from instruction issue to when the result is available. For vector instructions, the floating-point add functional unit is 6 CPs deep. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

Floating-point Multiply Functional Unit

The individual floating-point multiply functional units perform full- and half-precision multiplication of 64-bit operands in floating-point format. They execute instructions 064 through 067 (scalar multiplication) and instructions 160 through 167 (vector multiplication). The half-precision product is rounded; the full-precision product can be rounded or not rounded.

The vector floating-point multiply functional unit also executes instruction 166*ijk*. This instruction computes the 32-bit product of the contents of the S*j* register and the elements of the V*k* register and transmits the results to the V*i* register.

The vector floating-point multiply and second vector logical functional units share the input data paths to both functional units. The other functional unit is reserved when one functional unit is busy receiving operands over this data path. This reservation terminates when use of this input data path concludes.

Input operands must be normalized; the floating-point multiply functional unit delivers a normalized result only if both input operands are normalized. The floating-point multiply functional unit detects overflow and underflow conditions; only overflow conditions are flagged.

The scalar floating-point multiply functional unit time is 9 CPs. This functional unit time is measured from instruction issue to when the result is available. For vector instructions, the floating-point multiply functional unit is 7 CPs deep. This time is measured by the number of sequential registers from the input register through the output register in the functional unit.

The floating-point multiply functional unit recognizes both operands with zero exponents as a special case and performs an integer multiply operation. The result is considered an integer product, is not normalized, and is not considered out of range. This case provides a fast method of computing a 48-bit integer product, although the operands in this case must be shifted before the multiply

operation. Refer to the "Integer Arithmetic" subsection for more information on integer multiplication. Also described is the 32-bit integer multiply using the 166*ijk* instruction.

Reciprocal Approximation Functional Unit

The individual reciprocal approximation functional unit finds the approximate reciprocal of a 64-bit operand in floating-point format. These units execute instructions 070 and 174*ij*0. Because the vector population/parity/leading-zero functional unit shares the input data path with this unit, both functional units are busy while either is receiving operands over this data path.

The input operand must be normalized; the floating-point reciprocal approximation functional unit delivers a correct result only if the input operand is normalized. The high-order bit of the coefficient is not tested, but it is assumed to be a 1. The floating-point reciprocal approximation functional unit detects overflow and underflow conditions; both conditions are flagged.

The scalar reciprocal approximation functional unit time is 16 CPs. This functional unit time is measured from instruction issue to when the result is available. For vector instructions, the reciprocal approximation functional unit is 14 CPs deep. This time is measured according to the number of sequential registers that are accessed from the input register through the output register in the functional unit.

Bit-matrix Multiply Functional Unit

The bit-matrix multiply (BMM) functional unit performs bit-matrix multiply operations. This functionality is implemented with the instructions described in Table 32.

Table 32. Bit-matrix multiply instructions

Machine Code	CAL Syntax	Function Description
070*ij*6	S*i* S*j**BT	Transmit bit matrix product of (Sj) and transpose of (BMM) to *Si*
1740*j*4	BMM V*j*	Transmit (Vj elements) to BMM
174*ij*6	V*i* V*j**BT	Transmit bit matrix product of (Vj elements) and transpose of (BMM) to Vi elements

Instruction 1740j4 transmits the contents of the Vj register to the BMM register. This is the only way the contents of the BMM registers can be set. The vector length (VL) register determines how many elements of V*j* are transmitted to BMM. The rows of BMM beyond VL are zeroed. This instructions sets the bit matrix loaded (BML) bit in the mode register.

Instruction 070*ij*6 performs a scalar bit matrix multiply of the contents of the Sj register with the transpose of the contents of the BMM register and transmits the result to the S*i* register.

Instruction 174ij6 performs a vector bit matrix multiply of the contents of the V*j* register with the transpose of the contents of the BMM register and transmits the result to the V*i* register. The vector length (VL) register determines how many elements of V*j* are used.

The "B^T" in the CAL syntax for the two matrix multiply instructions is a short-hand notation that means *transpose of (BMM)*.

The following three sections describe how the results of the bit matrix multiplies are computed.

Because the operands are vectors and matrices of bits, mathematically the 070*ij*6 is a vector-matrix multiply and the 174i*j*6 is a matrix-matrix multiply. The 1740*j*4 instruction moves data into the functional unit to load matrix register BT.

Theory of Operation

The BMM function unit performs operations on matrices of bits as shown by the equation:

$A \times B^T = R$

where:

- A is either a row matrix of N bits or a square matrix of NxN bits
- B is a square matrix of NxN bits
- B^T is the transpose of B
- R is the bit matrix product of A and B^T
- R is the same size and shape as A
- N can have a value from 1 to 64 inclusive

Individual bits in a row matrix are denoted a_n where n is the column number. Individual bits in a square matrix are denoted a_{mn} where m is the row number and n is the column number. Normal mathematical convention numbers the columns in a matrix from left to right beginning with 1 and the rows from top to bottom beginning with 1.

If A is a row matrix, it can be represented as follows:

$[a_1\ a_2\ a_3\ ...a_n]$

If B is a square matrix, it can be represented as follows:

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ b_{31} & b_{32} & b_{33} & \dots & b_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & b_{n3} & \dots & b_{nn} \end{bmatrix}$$

Then the operation:

$A \times B^T = R$

can be represented as follows:

$$[a_1\ a_2\ a_3\ \dots\ a_n] \times \begin{bmatrix} b_{11} & b_{21} & b_{31} & \dots & b_{n1} \\ b_{12} & b_{22} & b_{32} & \dots & b_{n2} \\ b_{13} & b_{23} & b_{33} & \dots & b_{n3} \\ \vdots & \vdots & \vdots & & \vdots \\ b_{1n} & b_{2n} & b_{3n} & \dots & b_{nn} \end{bmatrix} = [r_1\ r_2\ r_3\ \dots r_n]$$

Notice that the B matrix has been transposed. The transpose of a matrix is formed by interchanging its rows and columns.

The individual result bits are calculated using the following equations:

$$r_1 = a_1 \bullet b_{11} \oplus a_2 \bullet b_{12} \oplus a_3 \bullet b_{13} \oplus \dots \oplus a_n \bullet b_{1n}$$
$$r_2 = a_1 \bullet b_{21} \oplus a_2 \bullet b_{22} \oplus a_3 \bullet b_{23} \oplus \dots \oplus a_n \bullet b_{2n}$$
$$r_3 = a_1 \bullet b_{31} \oplus a_2 \bullet b_{32} \oplus a_3 \bullet b_{33} \oplus \dots \oplus a_n \bullet b_{3n}$$
$$\vdots$$
$$r_n = a_1 \bullet b_{n1} \oplus a_2 \bullet b_{n2} \oplus a_3 \bullet b_{n3} \oplus \dots \oplus a_n \bullet b_{nn}$$

The symbol • is the AND operator.

The symbol ⊕ is the Exclusive-OR operator.

The • operator has higher precedence than the ⊕ operator.

If A is a square matrix then the operation:

$A \times B^T = R$

can be represented as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{21} & b_{31} & \dots & b_{n1} \\ b_{12} & b_{22} & b_{32} & \dots & b_{n2} \\ b_{13} & b_{23} & b_{33} & \dots & b_{n3} \\ \vdots & \vdots & \vdots & & \vdots \\ b_{1n} & b_{2n} & b_{3n} & \dots & b_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \dots & r_{1n} \\ r_{21} & r_{22} & r_{23} & \dots & r_{2n} \\ r_{31} & r_{32} & r_{33} & \dots & r_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \dots & r_{nn} \end{bmatrix}$$

The individual result bits are calculated using the following equations:

$$r_{11} = a_{11} \bullet b_{11} \oplus a_{12} \bullet b_{12} \oplus a_{13} \bullet b_{13} \oplus \dots \oplus a_{1n} \bullet b_{1n}$$
$$r_{12} = a_{11} \bullet b_{21} \oplus a_{12} \bullet b_{22} \oplus a_{13} \bullet b_{23} \oplus \dots \oplus a_{1n} \bullet b_{2n}$$
$$r_{13} = a_{11} \bullet b_{31} \oplus a_{12} \bullet b_{32} \oplus a_{13} \bullet b_{33} \oplus \dots \oplus a_{1n} \bullet b_{3n}$$
$$\vdots$$
$$r_{1n} = a_{11} \bullet b_{n1} \oplus a_{12} \bullet b_{n2} \oplus a_{13} \bullet b_{n3} \oplus \dots \oplus a_{1n} \bullet b_{nn}$$

$$r_{21} = a_{21} \bullet b_{11} \oplus a_{22} \bullet b_{12} \oplus a_{23} \bullet b_{13} \oplus \dots \oplus a_{2n} \bullet b_{1n}$$
$$\vdots$$
$$r_{2n} = a_{21} \bullet b_{n1} \oplus a_{22} \bullet b_{n2} \oplus a_{23} \bullet b_{n3} \oplus \dots \oplus a_{2n} \bullet b_{nn}$$
$$\vdots$$
$$r_{n1} = a_{n1} \bullet b_{11} \oplus a_{n2} \bullet b_{12} \oplus a_{n3} \bullet b_{13} \oplus \dots \oplus a_{nn} \bullet b_{1n}$$
$$\vdots$$
$$r_{nn} = a_{n1} \bullet b_{n1} \oplus a_{n2} \bullet b_{n2} \oplus a_{n3} \bullet b_{n3} \oplus \dots \oplus a_{nn} \bullet b_{nn}$$

These are the mathematical operations performed by the BMM function unit.

Bit Matrix Representation in SV1 Registers

The mathematical operations described above must use the S, V, VL, and BMM registers on SV1 hardware.

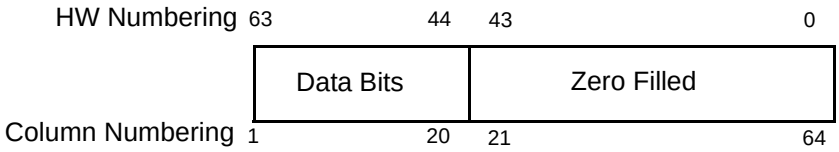
The size N of the matrices must first be set by setting VL to N. N can have a value from 1 to 64 inclusive.

The individual bits of a row matrix must be set in an S register, left justified (highest order bit is leftmost) within the register. The remainder of the row (if any) beyond column N must be zero filled in order to get valid results.

Normal mathematical convention numbers the positions in a row of a matrix from left to right beginning with 1. Cray hardware convention numbers bits of an S register beginning with the leftmost as the most significant bit and numbered as bit 63. This means that column 1 of the row matrix is bit 63 of the S register.

Figure 32 shows an example of a row matrix for N equals 20.

Figure 32. Row Matrix for N = 20

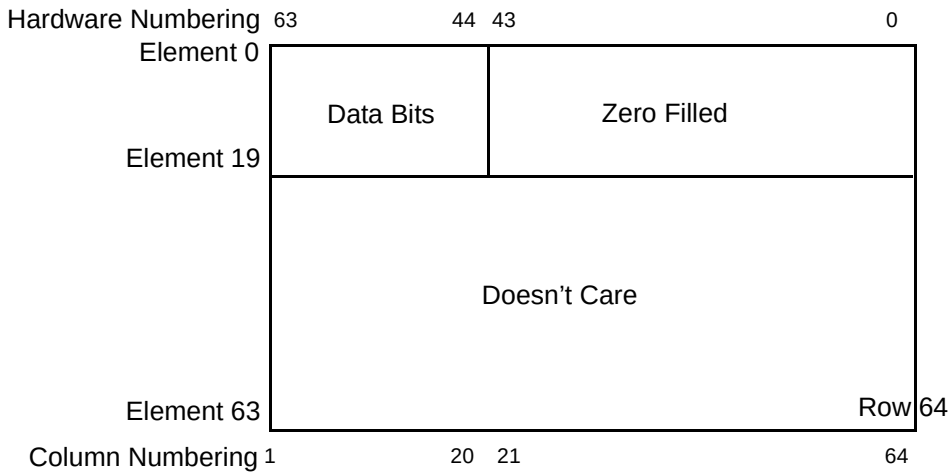


The individual bits of a square matrix must be set in a V register. The V register contains 64 elements and each element contains 64 bits. Therefore, a V register can be viewed as a square bit matrix of size 64x64. Normal mathematical convention numbers the rows of a matrix beginning with 1. Cray hardware convention numbers elements of a V register beginning with 0. This means that row 1 of the matrix is element 0 of the V register.

If the size of the square matrix is less than 64x64, that is if (VL) is less than 64, then the first N rows of the V register are used in the operations and rows of the V register beyond row N are not used in the operations. The bits in the first N rows must be left justified and zero filled as described above for a row matrix in order to get valid results.

Figure 33 shows an example of a square matrix for N equals 20.

Figure 33. Square Matrix for N = 20



The individual bits of the bit matrix in the BMM register are set with the BMM load instruction as described below.

Bit Matrix Multiply Operations on SV1

The mathematical operations described above must use instructions 070*ij*6, 1740*j*4, and 174*ij*6 on SV1 hardware.

The size of N of the matrices is the value of (VL).

The BMM load instruction, 1740j4, transmits the first N rows of the bit matrix in the Vj register to the first N rows of the BMM register. All remaining rows of the BMM register are set to all 0's.

When the BMM load instruction executes, the bit matrix loaded (BML) bit is set in the mode register. The clear BML bit instruction, 002210, clears the BML bit.

For the mathematical operations:

$A \times B^T = R$

the B matrix must first be loaded into the BMM register with the BMM load instruction where Vj contains the B matrix.

The scalar bit matrix multiply instruction, 070*ij*6, transmits the bit matrix product of the row matrix in the S*j* register and the transpose of the square matrix in the BMM register to the S*i* register.

If the A matrix in the mathematical operation is a row matrix, then the scalar bit matrix multiply instruction is used where S*j* contains the A matrix, BMM contains the B matrix, and S*i* contains the R matrix.

The vector bit matrix multiply instruction, 174*ij*6, transmits the bit matrix product of the square matrix in the V*j* register and the transpose of the square matrix in the BMM register to the V*i* register. Only the first N rows of V*j* are used in the operation.

If the A matrix in the mathematical operation is a square matrix, then the vector bit matrix multiply instruction is used where V*j* contains the A matrix, BMM contains the B matrix, and V*i* contains the R matrix.

To obtain valid results from the bit matrix multiply instructions the following conditions must exist:

- The two matrices must be left justified in the source registers.
- Bits beyond column N must be zero.
- For vector bit matrix multiply, the two matrices must be equal in size.

Both the scalar and vector bit matrix multiply instructions execute in the BMM functional unit as vector-type instructions. The S registers are used for operand and result registers for the scalar instruction. With the vector unit essentially quiet, the scalar bit matrix multiply function time is 7 CPs. This functional unit time is measured from instruction issue to when the result is available. For the vector bit matrix multiply, the functional unit is CPs deep. The time is measured according to the number of sequential registers that are accessed from the input register through the output register in the functional unit. The load of the BMM register takes 8, 16, 24, or 32 CPs, plus 24 CPs to complete as determined by VL/2.

Functional Unit Operations

Functional units in a CPU perform logical operations, integer arithmetic, floating-point arithmetic, and bit-matrix multiply operations. Integer and floating-point arithmetic are performed in two’s complement. The following subsections explain the logical operations, the integer arithmetic, and the floating-point arithmetic used by the system. Refer to the “Bit-matrix Multiply Functional Unit” subsection for that operation.

Logical Operations

Scalar and vector logical functional units perform bit-by-bit manipulation of 64-bit quantities. Instructions are provided for forming logical products, sums, differences, equivalences, and merges.

A logical product is the AND function. The following example shows an AND function.

Operand 1:	1 0 1 0
Operand 2:	1 1 0 0
Result:	1 0 0 0

A logical sum is the inclusive OR function. The following example shows an inclusive OR function.

Operand 1:	1 0 1 0
Operand 2:	1 1 0 0
Result:	1 1 1 0

A logical difference is the exclusive OR function. The following example shows an exclusive OR function.

Operand 1:	1 0 1 0
Operand 2:	1 1 0 0
Result:	0 1 1 0

A logical equivalence is the exclusive NOR function. The following example shows an exclusive NOR function.

Operand 1:	1 0 1 0
Operand 2:	1 1 0 0
Result:	1 0 0 1

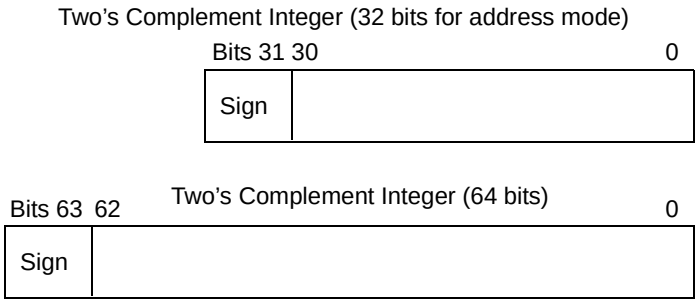
The merge operation uses two operands and a mask to produce results. The following example shows a merge operation. The bits of operand 1 pass where the mask bit is a 1. The bits of operand 2 pass where the mask bit is a 0.

Operand 1:	1 0 1 0 1 0 1 0
Operand 2:	1 1 0 0 1 1 0 0
Mask:	1 1 1 1 0 0 0 0
Result:	1 0 1 0 1 1 0 0

Integer Arithmetic

All integers, whether 32 or 64 bits, are represented in the registers as shown in Figure 34. The address add and address multiply functional units perform 32-bit arithmetic. The scalar add and vector add functional units perform 64-bit arithmetic.

Figure 34. Integer Data Formats



Multiplication of two scalar (64-bit) integer operands is done using the floating-point multiply instruction and one of two multiplication methods. The method used depends on the magnitude of the operands and the number of bits that are available to contain the product. The first method uses a standard floating-point multiply instruction (160*ijk* or 161*ijk*) to produce a 48-bit integer result from two properly shifted 24-bit integer operands. The second method uses the 166 instruction to perform a 32-bit integer multiply using two properly shifted 32-bit integer operands to produce a 32-bit integer result. The following paragraphs explain these two integer multiply operations.

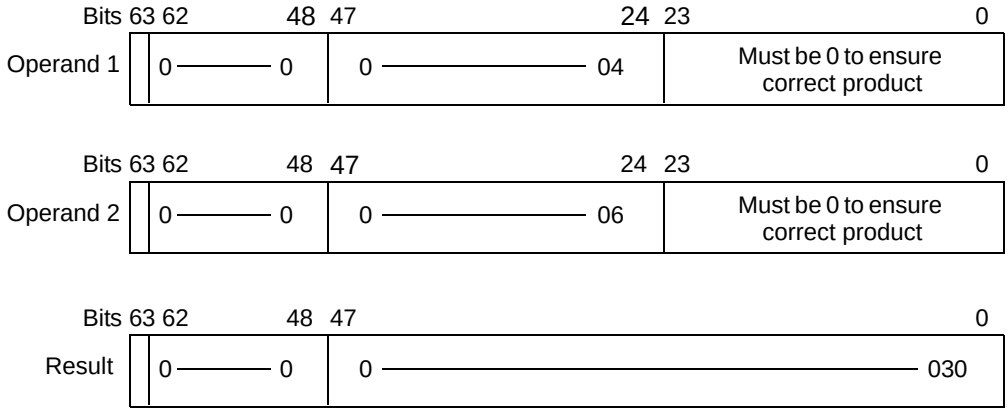
24-bit Integer Multiplication

The floating-point multiply functional unit recognizes a condition in which both operands have zero exponents as a special case; it is treated as an integer multiplication operation. A complete multiplication operation is performed with no truncation as long as the total number of bits in the two operands does not exceed 48 bit positions. To multiply two integer numbers, set the exponent of each operand (bits 48 through 62) equal to 0 and place each 32-bit integer value in bit positions 24 through 47 of the operand's coefficient field. To ensure accuracy, the least significant 24 bits must be 0's.

When the floating-point multiply functional unit performs the operation, it returns the high-order 48 bits of the product as the result coefficient and leaves the exponent field as 0. The result is a 48-bit quantity in bit positions 0 through 47; no normalization shift of the result is performed.

As shown in Figure 35, if operand 1 is 4 (octal) and operand 2 is 6 octal, a 48-bit result of 30 octal is produced. Bit 63 follows the rules for multiplying signs and the result is a signed-magnitude integer. Bits 63 of operands 1 and 2 are combined with an XOR function to derive the sign of the result. The format of integers expected by both the hardware and software is two's complement, not signed magnitude; therefore, negative products must be converted to two's complement form.

Figure 35. 24-bit Integer Multiply Performed in a Floating-point Multiply Functional Unit



If the 24 least significant bits of the operand coefficients are not shifted so that they are nonzero, the low-order 48 bits of the product could be nonzero, and the high-order 48 bits (the returned part) could be one larger than expected. This is caused by the truncation compensation constant that is added during a multiply. The truncation compensation constant is discussed in more detail in the "Floating-point Multiplication Algorithm" section.

Multiplication of Operands Greater than 24 Bits

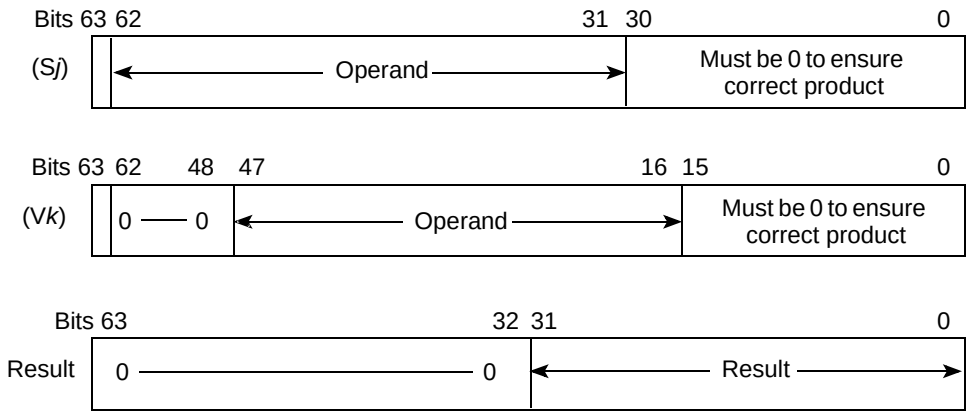
The second multiplication method is used when the operands are more than 24 bits long; multiplication is done by software that forms multiple partial products and then shifts and adds the partial products.

A second integer multiplication operation performs a 32-bit multiplication operation on the S*j* operand and the V*k* operand and puts the result in the V*i* register (166*ijk* instruction). The operands must be shifted left before the operation begins. The S*j* operand must be shifted left 31 decimal places, leaving the operand in bit positions 62 through 31; bit positions 30 through 0 must be equal to 0 to ensure accuracy (refer to Figure 36). The V*k* operand must be shifted left 16 decimal places, which places the operand in bit positions 16 through 47; bit positions 0 through 15 must be equal to 0 to ensure

accuracy. Bits 48 through 63 are zero filled. The result of the multiply is right justified into bit positions 0 through 31, while bit positions 32 through 63 are zero filled.

Although no integer division operation is provided, integer division can be carried out by converting the numbers to the floating-point format and then using the floating-point functional units. For more information on integer division, refer to the "Floating-point Division Algorithm" subsection.

Figure 36. 32-bit Integer Multiply Performed in a Floating-point Multiply Functional Unit



Floating-point Arithmetic

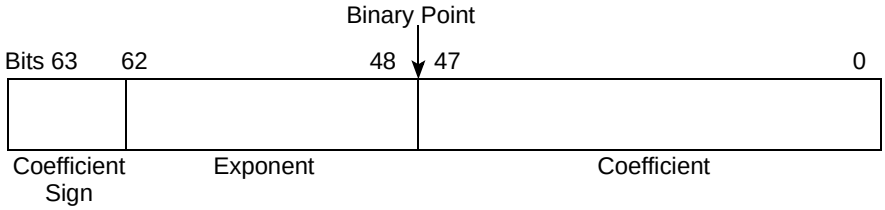
The scalar and vector instructions use floating-point arithmetic. The following subsections explain floating-point arithmetic:

- Floating-point data format
- Exponent ranges
- Normalized floating-point numbers
- Floating-point range errors
- Floating-point addition
- Multiplication and division algorithms
- Double-precision numbers

Floating-point Data Format

Floating-point numbers are represented in a standard format throughout the CPU; Figure 37 shows this format, which has three fields: coefficient sign, exponent, and coefficient.

Figure 37. Floating-point Data Format

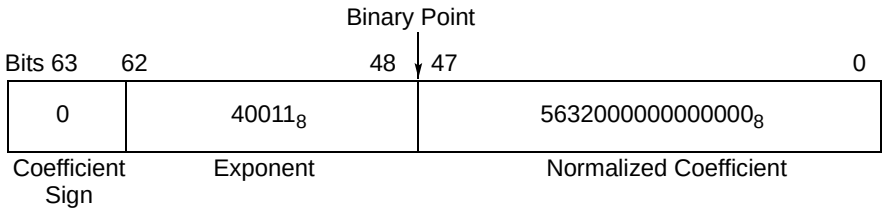


This format is a packed representation of a binary coefficient and an exponent (power of two). The coefficient sign is located in bit position 63 and is separated from the rest of the coefficient. If this bit is equal to 0, the coefficient is positive; if this bit is equal to 1, the coefficient is negative. The exponent is represented as a biased integer number in bit positions 62 through 48; each exponent is biased by 40000 (octal). Bit 61 is the sign of the exponent; a 0 indicates a positive exponent, and a 1 indicates a negative exponent. Bit 62 is the bias of the exponent.

The coefficient is a 48-bit signed fraction; the sign of the coefficient is located in bit position 63. Because the coefficient is in signed-magnitude format, it is not complemented for negative values. A normalized floating-point number has a 1 in bit position 47, and an unnormalized floating-point number has a 0 in this bit position (normalized numbers are discussed in more detail later in this section).

Figure 38 and the following steps show the relationship between the biased exponent and the coefficient. The following steps convert a floating-point number to its decimal equivalent.

Figure 38. Internal Representation of a Floating-point Number



1. Subtract the bias from the exponent to get the integer value of the exponent:

$$\begin{array}{rl} 40011 & \text{(octal)} \\ -40000 & \text{(octal)} \\ \hline 11 & \text{(octal)} \end{array} = 9 \text{ (decimal)}$$

2. Multiply the normalized coefficient by the power of 2 indicated in the exponent to get the result:

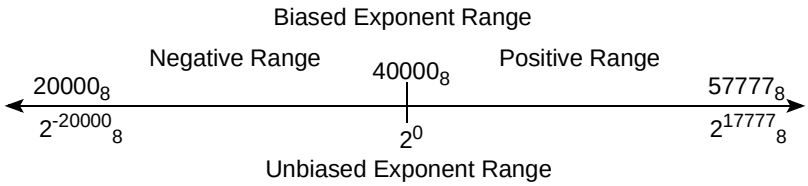
   0.5632 (octal) x 2 (exp 9) = 563.20 (octal) = 371.25 (decimal)

A zero value or an underflow result is not biased and is represented as a word of all 0's. A negative 0 is not generated by any floating-point functional unit, except the case in which a negative 0 is one operand going into the floating-point multiply or floating-point add functional unit.

Exponent Ranges

The exponent portion of the floating-point format is represented as a biased integer in bits 48 through 62. The bias added to the exponents is 40000 (octal), which represents an exponent of 2 (exp 0). Figure 39 shows the biased and unbiased exponent ranges.

Figure 39. Biased and Unbiased Exponent Ranges



In terms of decimal values, the floating-point format of the system allows the accurate expression of numbers to about 15 decimal digits in the approximate decimal range of 10 exp. -2466 through 10 exp +2466.

Normalized Floating-point Numbers

A nonzero floating-point number is normalized if the most significant bit of the coefficient (bit 47) is nonzero. This condition implies that the coefficient was shifted as far left as possible and that the exponent adjusted accordingly; therefore, a normalized floating-point number has no leading 0's in its coefficient. The exception is a normalized floating-point 0, which is all 0's.

When a floating-point number is created by inserting 40060 (octal) into the exponent and a 48-bit integer into the coefficient, normalize the result before using it in a floating-point operation. Normalization is accomplished by adding the unnormalized floating-point operand to 0. Because S0 provides a 64-bit zero when used in the S*j* field of an instruction, an operand in S*k* is normalized with the 062*i*0*k* instruction. S*i*, which can be the same register as S*k*, contains the normalized result.

The reciprocal approximation functional unit must have normalized numbers to produce correct results; using unnormalized numbers produces inaccurate results. The floating-point multiply functional unit does not require normalized numbers to get correct results; however, more accurate results occur when normalized numbers are used.

The floating-point add functional unit does not require normalized numbers to get correct results. The floating-point add functional unit does, however, automatically normalize all its results; unnormalized floating-point numbers may be routed through this functional unit to take advantage of this process.

Floating-point Range Errors

To ensure that the limits of the functional units are not exceeded, a range check for overflow and underflow conditions is made on the exponent of each floating-point number as it enters the functional unit. In the floating-point add and floating-point multiply functional units, bits 61 and 62 are checked; if both are equal to 1, the exponent is equal to or greater than 60000 (octal) and an overflow condition is detected.

In the reciprocal approximation functional unit, the exponent is complemented and the value of 2 is added before the operation proceeds. When the check is made in a reciprocal approximation operation, the exponent must be equal to or greater than 60002 (octal) for an overflow condition to occur.

When an overflow condition is detected, an interrupt occurs only if the interrupt-on floating-point error (IFP) bit is set in the mode register and the system is not in monitor mode. The IFP bit can be set or cleared by a user mode program; the Cray operating system (COS), or UNICOS, keeps a bit in the exchange package to indicate the condition of this mode bit. System software manipulates the mode bit and uses the exchange package bit to indicate how the mode is left to the user.

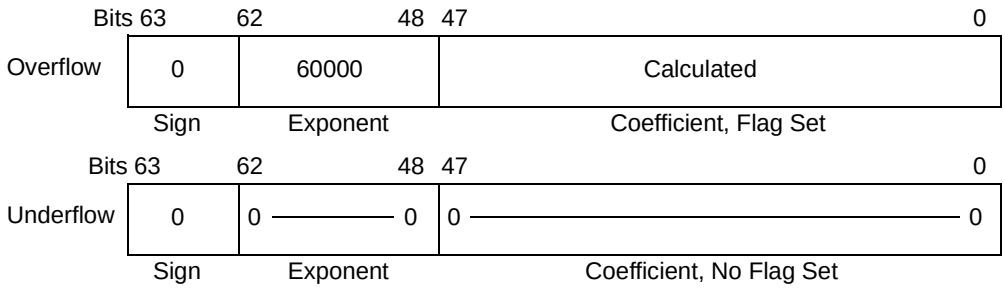
To check for an underflow condition in the floating-point add and multiply functional units, bits 61 and 62 are checked; if both are equal to 0, then the exponent is less than or equal to 17777 (octal), and an underflow condition is detected. No flag is set, but the exponent and coefficient are both set to 0’s.

Because the reciprocal approximation operation complements and adds 2 to a floating-point number, the result exponent must be less than or equal to 20001 (octal) for an underflow condition to occur. The underflow condition in the result exponent signals an overflow condition on the original exponent and forces the original exponent to 60000 (octal) and bit 47 to 0.

Floating-point Add Functional Unit Range Errors

A floating-point add range-error condition occurs in scalar operands when the larger incoming exponent is greater than or equal to 60000 (octal). This condition sets the floating-point error (FPE) flag in the flag register, and an exponent of 60000_8 is sent to the result register along with the computed coefficient (refer to Figure 40). If a floating-point addition or floating-point subtraction operation generates an exponent of less than 20000 (octal) or a coefficient of 0, the condition is considered an underflow. No fault is generated, and the word returned from the functional unit consists of all 0 bits.

Figure 40. Floating-point Add and Floating-point Multiply Range Errors



Floating-point Multiply Functional Unit Range Errors

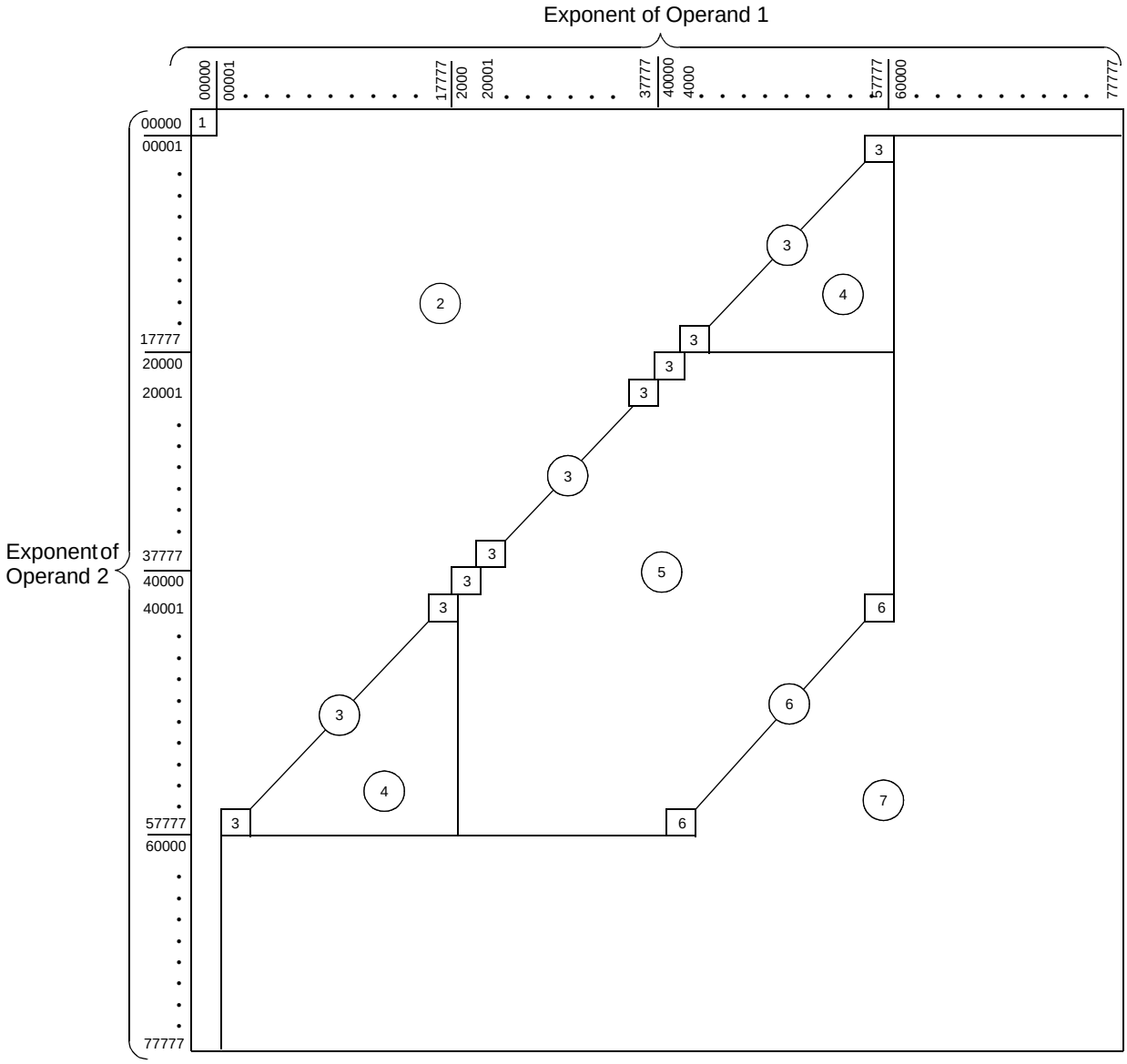
The floating-point multiply functional unit has the same range error conditions as the floating-point add functional unit (refer to Figure 41). The only exception is when both exponents are equal to 0; the multiply is allowed to proceed as an integer multiply, leaving the exponent and sign bits equal to 0.

Out-of-range conditions are tested before normalization in the floating-point multiply functional unit. The way in which the out-of-range conditions are handled can be determined by using the exponent matrix shown in Figure 41. The exponent of the result, for any set of exponents, falls into one of the following seven zones. Only zones 6 and 7 generate floating-point errors.

In Figure 41, a zone number is represented by a number inside of a circle. Octal number exponents of the two operands are represented by a number inside of a square.

A list of zones and their descriptions immediately follows Figure 41.

Figure 41. Exponent Matrix for a Floating-point Multiply Functional Unit



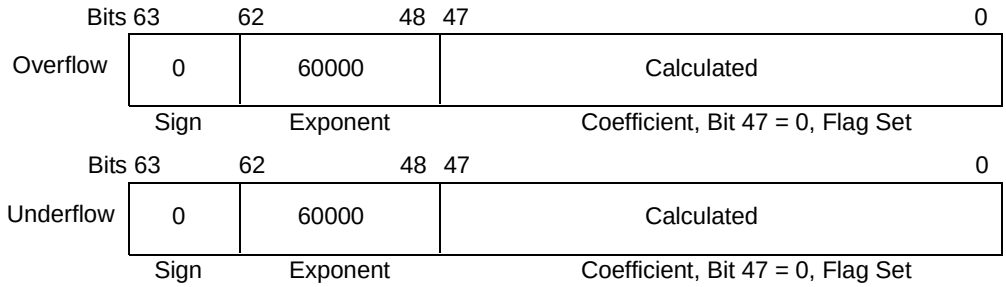
Note: In Figure 41, a zone number is represented by a number inside of a circle. Octal number exponents of the two operands are represented by a number inside of a square.

Zone	Description
1	Zone 1 indicates a simple integer multiply; no fault is possible.
2	Exponents in Zone 2 result in an underflow condition; the result is set to +0. (Multiply by 0 is in this group.)
3	An underflow condition may occur on this boundary in Zone 3. When a normalized shift is required, the underflow is not detected, and the coefficient and the exponent are not zeroed out. The exponent used before the shift is 2000 (octal); the exponent used after the shift is 17777_8. An underflow condition is detected on the exponent that is used for an unshifted product coefficient.
4	The use of an operand with an underflow exponent in Zone 4 is allowed if the final result is within the range 20000 (octal) to 57777 (octal).
5	Zone 5 is the normal operand range; normal results are produced.
6	An overflow condition is flagged on this boundary in Zone 6. If a normalized shift is required, the value should be within bounds if the exponent is 57777 (octal). Because overflow is detected, a 60000 (octal) is inserted in the product as the final exponent when the exponent for the unnormalized shift condition is used.
7	Within Zone 7, an overflow fault is flagged and the product exponent is set to 60000 (octal).

Floating-point Reciprocal Approximation Functional Unit Range Errors

For the floating-point reciprocal approximation functional unit, an incoming operand with an exponent less than or equal to 20001 (octal) or greater than or equal to 60002 (octal), causes a floating-point range error. The error flag is set and an exponent of 60000 (octal) and the computed coefficient with bit 47 set to 0 are sent to the result register (refer to Figure 42).

Figure 42. Floating-point Reciprocal Approximation Range Errors



Floating-point Addition Algorithm

Floating-point addition or subtraction is performed in a 49-bit register to allow for a sum that carries an additional bit position. The algorithm performs three operations: it equalizes exponents, adds coefficients, and normalizes results.

To equalize the exponents, the larger of the two exponents is retained. The coefficient of the smaller exponent is shifted right by the difference of the two exponents or until both exponents are equal. Bits shifted out of the register are lost; no roundup occurs. Because the coefficient is only 48 bits, any shift beyond 48 bits causes the smaller coefficient to become 0's.

After the two coefficients are equalized, they are added. Two conditions are analyzed to determine whether an addition or subtraction operation occurs. The two conditions are the sign bits of the two coefficients and the type of instruction (an add or subtract) issued. The following list shows how the operation is determined:

- If the sign bits are equal and an add instruction is issued, an addition operation is performed.
- If the sign bits are not equal and an add instruction is issued, a subtraction operation is performed.
- If the sign bits are equal and a subtract instruction is issued, a subtraction operation is performed.
- If the sign bits are not equal and a subtract instruction is issued, an addition operation is performed.

The last operation normalizes the results. To normalize the result, the coefficient is shifted left by the number of leading 0's (the coefficient is normalized when bit 47 is a 1). The exponent must also be decremented accordingly. If a carry operation across the binary point occurs during an addition operation, the coefficient is shifted right by 1 and the exponent increases by 1.

The normalization feature of the floating-point add functional unit is used to normalize any floating-point number. Simply pair the number with a zero operand and send both through the floating-point add functional unit.

A range check is performed on the result of all additions; refer to "Floating-point Range Errors" for more information on how the result is checked.

Floating-point Multiplication Algorithm

The floating-point multiply functional unit receives two 48-bit floating-point operands from either an S or V register as input into a multiply pyramid (refer to Figure 43). Multiplication is commutative, that is, A X B = B X A. The signs of the two operands are exclusively ORed, the exponents are added, the bias is subtracted, and the two 48-bit coefficients are multiplied. If the coefficients are both normalized, multiplying them produces a full product of either 95 or 96 bits. A 96-bit product is normalized as it is generated, but a 95-bit product requires a left shift of 1 to generate the final coefficient. If the shift is done, the final exponent is reduced by 1 to reflect the shift.

Because the result register (an S or V register) can hold only 48 bits in the coefficient, only the upper 48 bits of the 96-bit result are used. The lower 48 bits are never generated. The following paragraphs describe the truncation process that is used to compensate for the loss of bits in the product. It assumes that no shift was required to generate the final product; power-of-two designators are used.

The floating-point multiply functional unit truncates part of the low-order bits of the 96-bit product. To adjust for this truncation, a constant is unconditionally added above the truncation. The average value of this truncation is 9.25×2^{-56}, which was determined by adding all carries produced by all possible combinations that could be truncated and dividing the sum by the number of possible combinations. Nine carries are injected at bit position -56 to compensate for the truncated bits.

The effect of the truncation without compensation is at most a result coefficient 1 smaller than expected. With compensation, the results range from 1 too large to 1 too small in bit position -48. Approximately 99% of the values have zero deviation from the result if a full 96-bit product was present. Rounding is optional, but truncation compensation is not. The rounding method adds a constant so that the result is 50% high [0.25×2^{-48} (high)] 38% of the time, and 25% low [0.125×2^{-48} (low)] 62% of the time, which results in a near-zero average rounding error. In a full-precision rounded multiplication operation, 2 rounding bits are entered into the summation at bit positions -50 and -51 and are allowed to propagate.

For a half-precision multiplication operation, rounding bits are entered into the summation at bit positions -32 and -31. A carry bit that results from this entry is allowed to propagate upward, and the 29 most significant bits of the normalized result are transmitted back.

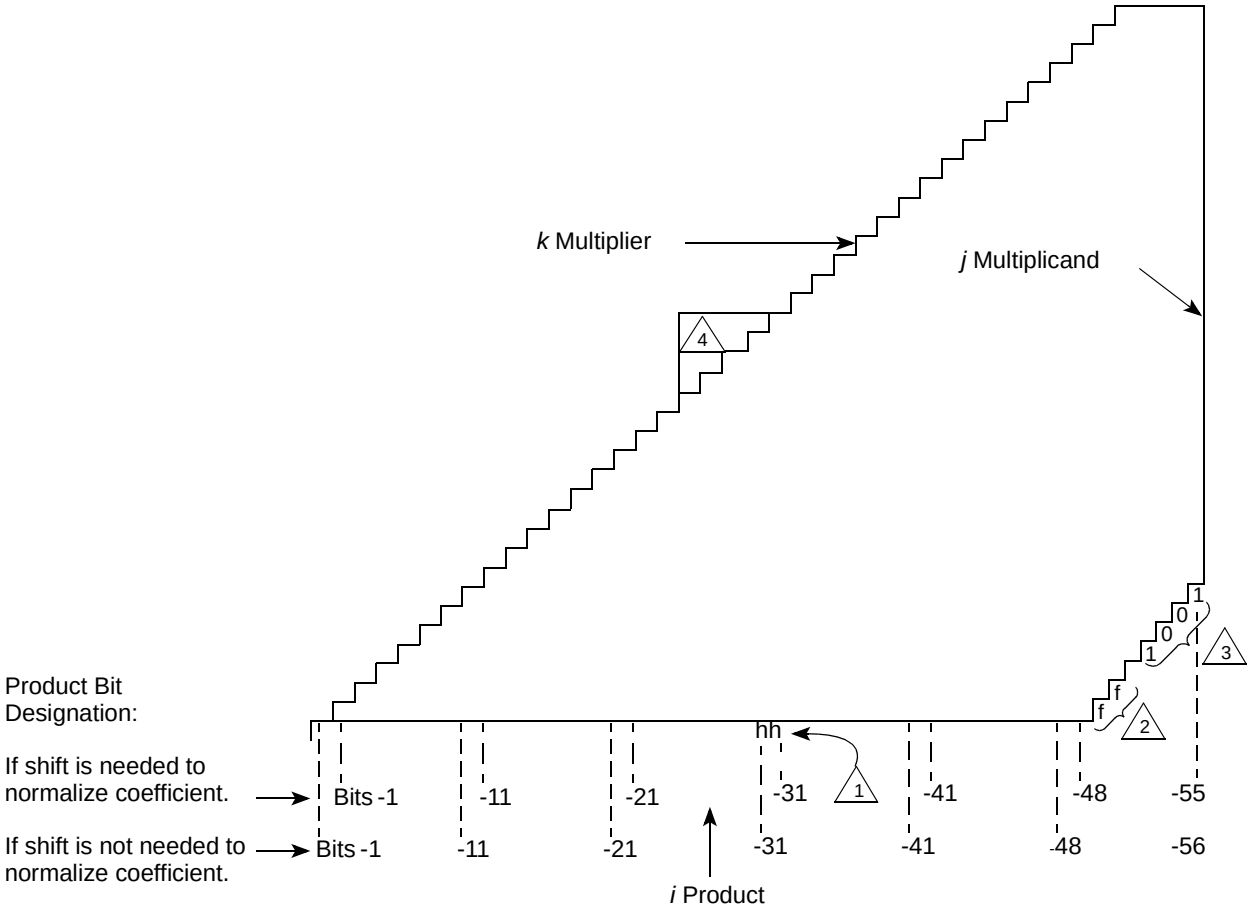
The result variations caused by this truncation and rounding are in one of the following ranges:

-0.23×2^{-48} to $+ 0.57 \times 2^{-48}$
or
-8.17×10^{-16} to $+ 20.25 \times 10^{-16}$

With a full 96-bit product and rounding equal to one-half the least significant bit, the following result variation is expected:

-0.5×2^{-48} to $+ 0.5 \times 2^{-48}$

Figure 43. Floating-point Multiply Partial-product Sums Pyramid



1. hh = 11_2 for half-precision round, 00_2 for full-precision rounded or full-precision unrounded multiplication operation.
2. ff = 11_2 for full-precision round, 00_2 for half-precision rounded or full-precision unrounded multiplication operation.
3. Truncation compensation constant; 1001_2 used for all multiplication operations.
4. Used only for 32-bit integer multiplication operation with instruction 166*ijk*. Summations for any other instructions are blocked.

Floating-point Division Algorithm

An SV1 series computer system does not have a single functional unit dedicated to the division operation. Rather, the floating-point multiply and reciprocal approximation functional units together carry out the algorithm. The following paragraphs explain the algorithm and how it is used in the functional units.

Finding the quotient of two floating-point numbers involves two steps. For example, to find the quotient A/B, first the B operand is sent through the reciprocal approximation functional unit to obtain its reciprocal, 1/B. Second, this result along with the A operand is sent to the floating-point multiply functional unit to obtain the product A X 1/B.

The reciprocal approximation functional unit uses an application of Newton's method for approximating the real root of an arbitrary equation F(x) = 0 to find reciprocals.

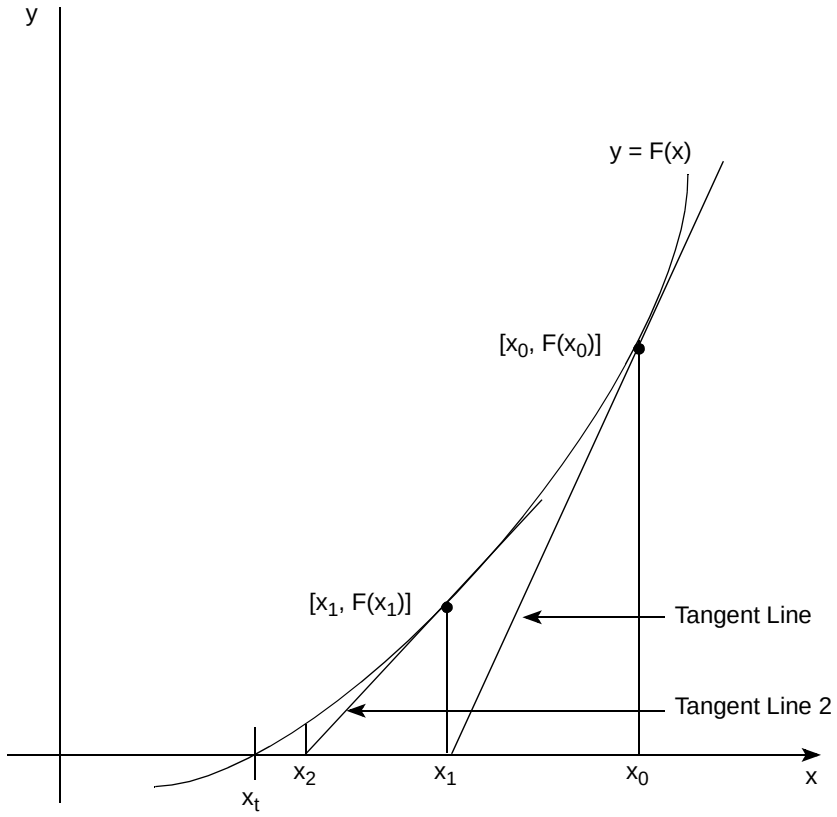
To find the reciprocal, the equation F(x) = 1/x - B = 0 must be solved. To do this, a number, A, must be found so that F(A) = 1/A - B = 0. That is, the number A is the root of the equation 1/x - B = 0. The method requires an initial approximation (or guess, which is shown as x_0 in Figure 44) sufficiently close to the true root (which is shown as x_t in Figure 44). x_0 is then used to obtain a better approximation; this is done by drawing a tangent line (line 1 in Figure 44) to the graph of y = F(x) at the point $[x_0, F(x_0)]$. The x-intercept of this tangent line becomes the second approximation, x_1. This process is repeated, using tangent line 2 to obtain x_2, and so on.

The following iteration equation is derived from this process:

$$x_{(i+1)} = 2x_i - x^2_i B = x_i (2 - x_i B)$$

In the equation, $x_{(i+1)}$ is the next iteration, x_i is the current iteration, and B is the divisor. Each $x_{(i+1)}$ is a better approximation than x_i to the true value, x_t. The exact answer is generally not obtained at once because the correction term is not exact. The operation is repeated until the answer becomes sufficiently close for practical use.

Figure 44. Newton's Method of Approximation



The mainframe uses this approximation technique based on Newton's method. A hardware look-up table provides an initial guess, x_0, with an accuracy of 8 bits to start the process. The following iterations are then calculated.

Iteration	Operation	Description
1	$x_1 = x_0(2 - x_0B)$	The first approximation is done in the reciprocal approximation functional unit and is accurate to 16 bits.
2	$x_2 = x_1(2 - x_1B)$	The second approximation is done in the reciprocal approximation functional unit and is accurate to 30 bits.
3	$x_3 = x_2(2 - x_2B)$	The third approximation is done in the floating-point multiply functional unit to calculate the correction term.

The reciprocal approximation functional unit calculates the first two iterations, while the floating-point multiply functional unit calculates the third iteration. The third iteration uses a special instruction within the floating-point multiply

functional unit to calculate the correction term. This iteration is used to increase accuracy of the reciprocal approximation functional unit's answer to full precision (the floating-point multiply functional unit can provide both full- and half-precision results).

The reciprocal iteration is designed for use once with each half-precision reciprocal that is generated. If the third iteration (the iteration performed by the floating-point multiply functional unit) results in an exact reciprocal, or if an exact reciprocal is generated by some other method, performing another iteration results in an incorrect final reciprocal. A fourth iteration should not be done.

The following example shows how the floating-point multiply functional unit provides a full-precision result, computing the value of S1/S2.

Step	Operation	Unit
1	S3 = 1/S2	Reciprocal approximation functional unit
2	S4 = [2 - (S3 * S2)]	Floating-point multiply functional unit
3	S5 = S4 * S3	Floating-point multiply functional unit using full-precision; S5 now equals 1/S2 to 48-bit accuracy
4	S6 = S5 * S1	Floating-point multiply functional unit using full-precision rounding

The reciprocal approximation in Step 1 is correct to 30 bits. By Step 3, it is accurate to 48 bits. This iteration answer is applied as an operand in a full-precision rounded multiplication operation (Step 4) to obtain a quotient accurate to 48 bits. Additional iterations may produce erroneous results.

When 29 bits of accuracy are sufficient, the reciprocal approximation instruction is used with the half-precision multiply to produce a half-precision quotient in only two operations, as shown in the following example.

Step	Operation	Unit
1	S3 = 1/S2	Reciprocal approximation functional unit
2	S6 = S1 * S3	Floating-point multiply functional unit in half-precision

The 19 low-order bits of the half-precision multiply results are returned as 0's with a rounding applied to the low-order bit of the 29-bit result.

Another method of computing division follows:

Step	Operation	Unit
1	S3 = 1/S2	Reciprocal approximation functional unit
2	S5 = S1 * S3	Floating-point multiply functional unit
3	S4 = [2 - (S3 * S2)]	Floating-point multiply functional unit
4	S6 = S4 * S15	Floating-point multiply functional unit

With this method, the correction to reach a full-precision reciprocal is done after the numerator is multiplied by the half-precision reciprocal rather than before the multiplication.

The coefficient of the reciprocal produced by this alternative method can be different by as much as 2×2^{-48} from the first method described for generating full-precision reciprocals. This difference can occur because one method can round up as much as twice, while the other method may not round at all. One rounding can occur while the correction is generated and the second rounding can occur when producing the final quotient. Therefore, use the same method to compare the reciprocals each time they are generated. The Cray Fortran CFT and CFT90 compilers use a consistent method to ensure that the reciprocals of numbers are always the same.

Double-precision Numbers

The CPU does not provide special hardware for performing double- or multiple-precision operations. Double-precision computations with 95-bit accuracy are available through software routines that Cray provides.

Parallel Processing Features

An SV1 series system has several special features that enhance the parallel processing capabilities inherent in the system. The following subsections discuss two types of parallel processing that SV1 series systems use:

- Parallel processing within a single CPU
- Parallel processing among two or more CPUs within a mainframe
- Parallel processing across mainframes within a system of up to 32 mainframes

Parallel processing features within a single CPU include instruction pipelining and segmentation, functional unit independence and duplication, and vector processing (vectorization). The first two features are inherent hardware features of the system; a programmer has little control over these features. However, the vector processing feature can be manipulated by the programmer to provide optimum throughput. Refer to "Vector Processing" for more information on vector processing.

Parallel processing among two or more CPUs is called multiprocessing, which is the capability of several programs to run concurrently on multiple CPUs of a single mainframe. Included in this category are multitasking and the Autotasking feature of the CF90 Fortran compiling system. Multitasking is the capability to run two or more parts (or tasks) of a single program in parallel on different CPUs within a mainframe. The Autotasking feature provides automatic multiprocessing; it automatically partitions user programs among multiple CPUs without user interface.

Because the intent of this document is to present programmers with system hardware information, the following subsections focus on the parallel processing features that are most closely related to the hardware (the parallel processing features that execute within a single CPU of a mainframe). A basic definition and explanation of multiprocessing, multitasking, and the Autotasking feature are included.

Pipelining and Segmentation

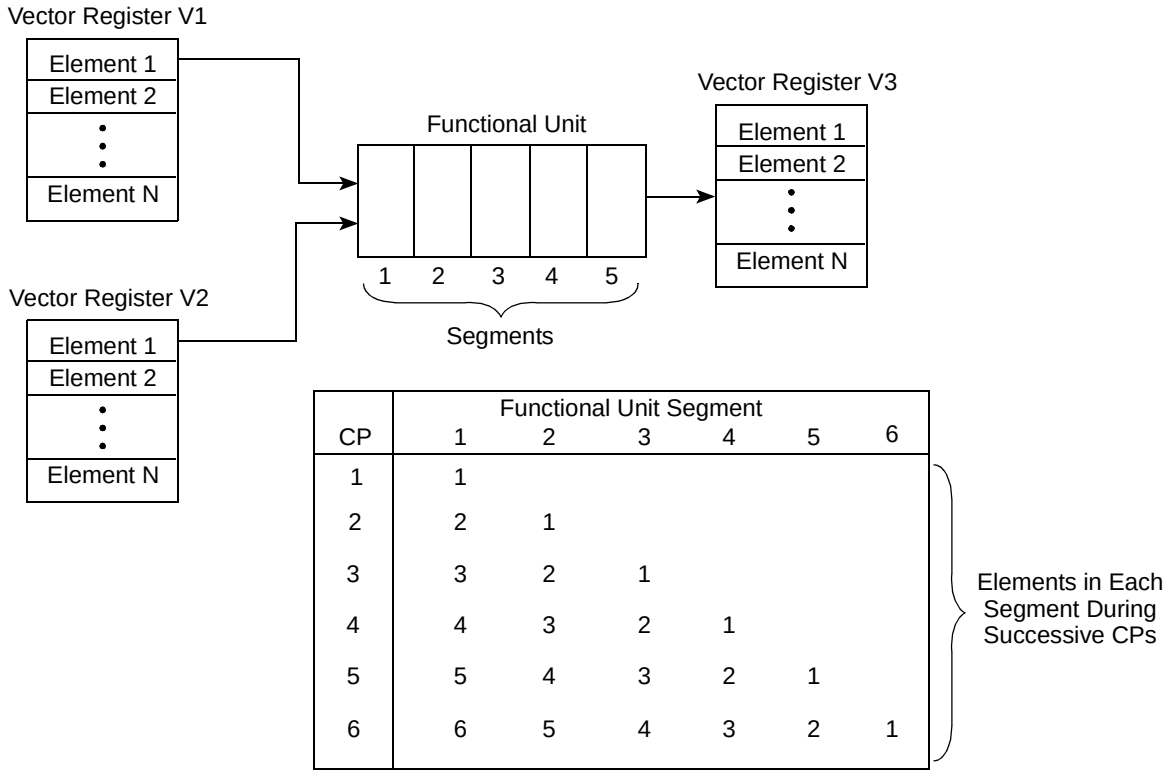
Pipelining is the process in which an operation or instruction begins before a previous operation or instruction completes. Pipelining requires fully segmented hardware. Segmentation is the process by which an operation is divided into a discrete number of sequential steps, or segments. Fully segmented hardware uses this feature to perform one segment of the operation during a single clock period (CP).

At the beginning of the next CP, the partial results are sent to the next segment of the hardware for processing in the next step of the operation. During this CP, the previous hardware segment processes the next operation. Without segmented hardware, an entire operation or instruction must complete before another operation or instruction starts. In the SV1 series system, all hardware is fully segmented.

Therefore, pipelining occurs during all hardware operations such as exchange sequences, memory references, instruction fetch sequences, instruction issue sequences, and functional unit operations. The pipelining and segmentation features are critical to the execution of vector instructions.

Figure 45 shows how a set of elements is pipelined through a segmented vector functional unit. In the first CP, element 1 of register V1 and element 1 of register V2 enter the first segment of the functional unit. During the next CP, the partial result is moved to the second segment of the functional unit, and element 2 of both vector registers enters the first segment. This process continues each CP until all elements are completely processed.

Figure 45. Segmentation and Pipelining Example



CP	Functional Unit Segment 1	2	3	4	5	6
1	1					
2	2	1				
3	3	2	1			
4	4	3	2	1		
5	5	4	3	2	1	
6	6	5	4	3	2	1

Elements in Each Segment During Successive CPs

In this example, the functional unit is divided into five segments; the functional unit can process up to five pairs of elements simultaneously. After 5 CPs, the first result leaves the functional unit and enters vector register V3; subsequent results are available at the rate of one result per CP.

Functional Unit Independence

The specialized functional units in the system handle the arithmetic, logical, and shift operations. Most functional units are fully independent; any number of functional units can process instructions concurrently. Functional unit independence enables different operations such as multiplications and additions to proceed in parallel.

For example, the equation A = (B + C) X D X E could be run as follows. If operands B, C, D, and E are loaded into the S registers, three instructions are generated for the equation: one that adds B and C, one that multiplies D and E, and one that multiplies the results of these two operations. The multiplication of D and E is issued first, followed by the addition of B and C. The addition and the multiplication proceed concurrently; because the addition takes less time to run than the multiplication, they complete at the same time. The addition operation is essentially hidden in that it occurs during the same time interval as the multiplication operation. The results of these two operations are then multiplied to obtain the final result.

Multiprocessing and Multitasking

Users can incorporate parallel processing features known as multiprocessing and multitasking; this category also includes microtasking.

Parallel processing among two or more CPUs is called multiprocessing, which is the capability of a program to run concurrently on multiple CPUs of a single mainframe. Applying more than one processor to a single job implies that the job has software tasks (parts) that can run in parallel. Such a job can be logically or functionally divided to allow two or more parts of the work to run simultaneously (that is, in parallel). One example of multiprocessing is a weather-modeling job in which the northern hemisphere calculation is one part and the southern hemisphere another part. Distinct code segments are not needed; the same code runs on multiple processors simultaneously, with each processor acting on different data.

Multitasking is the capability to run two or more tasks of a single program in parallel on different CPUs within a mainframe. The multitasking theory is that a program that runs on a dedicated system in wall clock time *t* can be multitasked to run in a time as short as *t*/*n*, if modified to use *n* or more parallel tasks on a machine with *n* CPUs.

In practice, however, a speedup factor of *n* is not quite attainable. In some instances, multitasking can actually increase a program’s execution time if the multitasking overhead decreases performance more than parallel execution time improves it. The following factors can limit the maximum improvement for a program:

- Not all parts of a program can be divided into parallel tasks.
- Those parts that can be multitasked may depend on one another so that, at run time, one or more tasks must wait until others complete some operation.

- Use of the multitasking features incurs overhead that cannot be recovered.

The CFT compiler on the system automatically uses the vector hardware to perform operations on inner DO loops that have no data dependencies. Once such optimizing is complete, a single processor can work no faster, but more than one processor could operate on separate parts of the data simultaneously to achieve results faster. Microtasking permits multiple processors to work on a Fortran program at the DO-loop level. The name *microtasking* was chosen because multiprocessing is efficient even at a DO-loop level where the task size, or granularity, may be small.

Microtasking also works well when the number of processors available is unknown or may vary during the program's execution. This means that microtasked jobs do not require a dedicated system, although they perform best in a dedicated environment with no competing jobs.

Advanced programming skills and tools are needed to successfully use multiprocessing, multitasking, and microtasking concepts in order to promote more efficient programs. These features are thoroughly discussed and explained in Cray software publications.

Autotasking Feature

System analysts and programmers can use the Autotasking component of the compilers, which uses automatic multitasking, to automatically detect whether portions of their programs can be run in parallel. The Autotasking feature is an extension of multiprocessing and microtasking and is designed to make parallel processing easier to use. The Autotasking feature alters a Fortran program to allow it to run simultaneously on multiple CPUs.

Enabling and Disabling the Maintenance Mode

The maintenance mode feature of an SV1 series system allows a programmer to write programs that assist in locating hardware failures in central memory. By using maintenance mode, a programmer can disable memory error correction and replace check bits with data bits.

A maintenance mode enable bit for each CPU is located in the configuration register for each CPU. Setting the maintenance bit enables the maintenance mode in the CPU. Maintenance functions can then be set and cleared by machine instructions.

Clearing the maintenance mode enable bit in the configuration register disables the maintenance mode of a CPU. With maintenance mode disabled, existing maintenance functions are cleared, and machine instructions cannot set any new maintenance functions.

Note: Memory error correction in the SV1 CPUs cannot be disabled by maintenance instructions, but it can be disabled via CA ASIC configuration bits. This restriction does not apply to I/O data error correction.

Using Maintenance Mode

Two instructions set and clear all maintenance functions within the maintenance mode. Instructions operate only in a CPU in monitor mode; they execute as no-operation (no-op) instructions in a CPU that is not in monitor mode. Instruction 0015*j*1 enables one of four maintenance functions. These instructions must be entered in machine code; Cray Assembly Language (CAL) does not support them. Allow 10 clock periods (CPs) for maintenance functions to be set or cleared. Ensure that no memory reference that may be affected by the maintenance functions can occur within 10 CPs after a 0015*j*1 or 073*i*31 instruction issues.

Table 33 lists the maintenance functions that the 0015*j*1 instruction sets for four values of *j*. Multiple maintenance functions can be set by executing the 0015*j*1 instruction more than once with different values of *j*. All maintenance functions remain set until they are cleared by a 073*i*31 instruction or until the maintenance mode is disabled.

Table 33. 0051j1 Instruction Operation

Maintenance Code	Description
001501	Do not use this instruction
001511	No-op instruction
001521	Disable port D error correction
001541	Replace check bits with data bits on memory writes Replace data bits with check bits on memory reads Replacement bits: **Data Bit** / **Check Bit** 0 / 64 8 / 65 16 / 66 24 / 67 32 / 68 40 / 69 48 / 70 56 / 71
001551	Replace check bits with V*k* data bits on the path to the VA ASIC during execution of 1771*jk* instructions Replacement bits: **Data Bit** / **Check Bit** 0 / 64 1 / 65 2 / 66 3 / 67 4 / 68 5 / 69 6 / 70 7 / 71

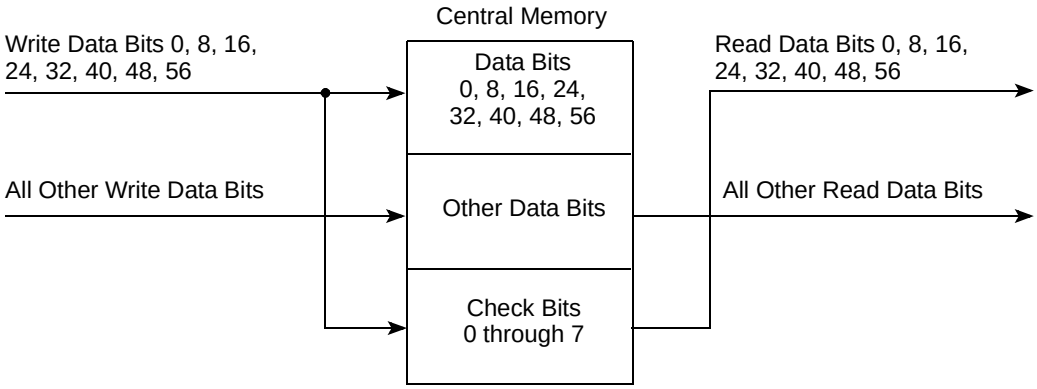
Note: These instructions are privileged to monitor mode.

Memory read and write error correction for a CPU is disabled via a configuration bit in the two CA ASICs per CPU. Instructions 001501 and 001511 DO NOT disable ports A and B error correction. After error correction has been disabled, memory read data passes through the error correction, regardless of the state of the check bits. Error detection and reporting continue as usual. The mode register bits can still enable or disable error interrupts.

Instruction 001541 allows the programmer to define the check bits that are stored with a data word, instead of allowing the check-bit generation logic to determine the check bits. It also allows the programmer to read the check bits (refer to Figure 46). After the 001541 instruction is executed, all memory write

references cause data bits 0, 8, 16, 24, 32, 40, 48, and 56 to be stored as check bits 0 through 7. Memory read references cause check bits 0 through 7 to replace the appropriate data bits.

Figure 46. Instruction 001541 Operation



Instruction 001551 modifies the operation of instruction 1771*jk*. After instruction 001551 executes, the 1771*jk* instruction no longer performs a scatter operation, but instead performs a stride operation that is similar to the 1770*jk* instruction. Register A0 contains the base address, A*k* contains the address increment, and V*j* contains the data to be written to memory. Register V*k* is no longer used for addressing. Instead, V*k* data bits 0 through 7 are written to memory as check bits 0 through 7. V*k* data bits 8 through 63 are not used.

Instructions 001531 and 001561 are not used. Instruction 001571 is reserved for future use.

Instruction 073*i*31 clears all maintenance functions set by the 0015*j*1 instructions, transfers the contents of the status register to register S*i*, and clears the performance monitor pointer.

CPU Instructions

The following subsections explain the instruction formats and special register values that the computer system uses. A central processing unit (CPU) instruction summary is included as well as a quick-reference table of all CPU instructions.

Quick-reference Table of CPU Instructions

Table 34. Quick-reference Table of CPU Instructions

Machine Instruction	CAL Syntax	Description
000000	ERR	Error exit
0010*jk*[a]	CA,A*j* A*k*	Set the CA register for the channel indicated by (A*j*) to (A*k*) and activate the channel.
001000	PASS	This is a no-operation instruction.
0011*jk*[a]	CL,A*j* A*k*	Set the CL register for the channel indicated by (A*j*) to (A*k*) address.
0012*j*0[a]	CI,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); clear device master clear (output channels only).
0012*j*1[a]	MC,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); set device master clear (output channels only); clear device ready-held (input channels only).
0013*j*0[a]	XA A*j*	Transmit (A*j*) to the XA register.
0014*j*0[a]	RT S*j*	Load the RTC register with (S*j*).
0014*j*1[a]	SIPI A*j*	Send an interprocessor interrupt request to CPU (A*j*).
001401[a]	SIPI	Send an interprocessor interrupt request to CPU 0.
001402[a]	CIPI	Clear the interprocessor interrupt.
0014*j*3[a]	CLN A*j*	Load the CLN register with (A*j*).
0014*j*4[a]	PCI S*j*	Load the II register with (S*j*).
001405[a]	CCI	Clear the programmable clock interrupt request.
001406[a]	ECI	Enable the programmable clock interrupt request.
001407[a]	DCI	Disable the programmable clock interrupt request.
0015*j*0 [a, c]		Select performance monitor.
001511[a, c]		No-op instruction.
001521[a, c]		Disable port D I/O error correction.
001541[a, c]		Enable replacement of checkbyte with data on ports for writes and the replacement of data with checkbytes on ports for reads (Ports A, B, and D).
001551[a, c]		Replace check bits with V*k* data bits on ports A and B writes to memory during execution of instruction 1771*jk*.
0016*j*1[b]	IVC	Send invalidate cache request to CPU (A*j*).
00200*k*	VL A*k*	Transmit (A*k*) to VL register.
002000	VL 1	Transmit 1 to VL register.
002100	EFI	Enable interrupt on floating-point error.
002200	DFI	Disable interrupt on floating-point error.
002210	CBL	Clear bit-matrix loaded bit in status and exchange package.
002300	ERI	Enable interrupt on operand range error.
002400	DRI	Disable interrupt on operand range error.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
002500	DBM	Disable bidirectional memory transfers.
002600	EBM	Enable bidirectional memory transfers.
002700	CMR	Complete memory references.
002703[f]	ETSI	Enable test and set invalidate
002704[f]	CPA	Hold if port A or port B busy
002705[f]	CPR	Hold if port A or port B busy
002706[f]	CPW	Hold instruction issue if block store or scalar load/store busy
002707[f]	DTSI	Disable test and set invalidate
0030*j*0	VM S*j*	Transmit (S*j*) to VM register.
003000[b]	VM 0	Clear VM register.
0034*jk*	SM*jk* 1,TS	Invalidate cache and test and set semaphore *jk*, $0 \le jk \le 31_{10}$.
0036*jk*	SM*jk* 0	Clear semaphore *jk*, $0 \le jk \le 31_{10}$.
0037*jk*	SM*jk* 1	Set semaphore *jk*, $0 \le jk \le 31_{10}$.
004000	EX	Normal exit from the operating system.
0050*jk*	J B*jk*	Jump to (B*jk*).
006*ijkm*	J *exp*	Jump to *exp*.
007*ijkm*	R *exp*	Return jump to *exp* and set register B00 to (P) + 2.
010*ijkm*[d]	JAZ *exp*	Jump to *exp* if (A0) = 0 ($i_2 = 0$).
011*ijkm*[d]	JAN *exp*	Jump to *exp* if (A0) ≠ 0 ($i_2 = 0$).
012*ijkm*[d]	JAP *exp*	Jump to *exp* if (A0) positive; (A0) ≥ 0 ($i_2 = 0$).
013*ijkm*[d]	JAM *exp*	Jump to *exp* if (A0) negative ($i_2 = 0$).
014*ijkm*[d]	JSZ *exp*	Jump to *exp* if (S0) = 0 ($i_2 = 0$)
015*ijkm*[d]	JSN *exp*	Jump to *exp* if (S0) ≠ 0 ($i_2 = 0$)
016*ijkm*[d]	JSP *exp*	Jump to *exp* if (S0) positive; ($i_2 = 0$)
017*ijkm*[d]	JSM *exp*	Jump to *exp* if (S0) negative ($i_2 = 0$)
020*i*00*mn*[e] or 022*ijk*[e]	A*i* *exp*	Transmit *exp* into A*i*
021i00mn	A*i* #exp	Transmit ones complement of *exp* into A*i*
023*ij*0	A*i* S*j*	Transmit (S*j*) to A*i*.
023*i*01	A*i* VL	Transmit (VL) to A*i*.
024*ijk*	A*i* B*jk*	Transmit (B*jk*) to A*i*.
025*ijk*	B*jk* A*i*	Transmit (A*i*) to B*jk*.
026*ij*0	A*i* PS*j*	Transmit the population count of (S*j*) to A*i*.
026*ij*1	A*i* QS*j*	Transmit the population count parity of (S*j*) to A*i*.
026*ij*7	A*i* SB*j*	Transmit (SB*j*) to A*i*.
027*ij*0	A*i* ZS*j*	Transmit leading zero count of (S*j*) to A*i*.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
027*ij*7	SB*j* A*i*	Transmit (A*i*) to SB*j*.
030*ijk*	A*i* Aj + A*k*	Transmit the integer sum of (A*j*) and (A*k*) to A*i*.
030*i*0*k*[b]	A*i* A*k*	Transmit (A*k*) to A*i*.
030*ij*0[b]	A*i* Aj+1	Transmit the integer sum of (A*j*) and 1 to A*i*.
031*ijk*	A*i* Aj-A*k*	Transmit the integer difference (A*j*) and (A*k*) to A*i*.
031*i*00[b]	A*i* -1	Transmit -1 to A*i*.
031*i*0*k*[b]	A*i* -A*k*	Transmit the negative of (A*k*) to A*i*.
031*ij*0[b]	A*i* Aj-1	Transmit the integer difference (A*j*) and 1 to A*i*.
032*ijk*	A*i* Aj*A*k*	Transmit the integer product of (A*j*) and (A*k*) to A*i*.
033*i*00	A*i* CI	Transmit the channel number of the highest priority interrupt request to A*i* (*j* = 0).
033*ij*0	A*i* CA,A*j*	Transmit the current address of the channel (A*j*) to A*i* ($j \neq 0$, $k = 0$).
033*ij*1	A*i* CE,A*j*	Transmit the error flag of channel (A*j*) to A*i* ($j \neq 0$, $k = 1$).
034*ijk*	B*jk*, A*i* ,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
034*ijk*[b]	B*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
035*ijk*	,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
035*ijk*[b]	0,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
036*ijk*	T*jk*,A*i* ,A0	Load (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
036*ijk*[b]	T*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
037*ijk*	,A0 T*jk*,A*i*	Store (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).
037*ijk*[b]	0,A0 T*jk*,A*i*	Store (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).
040*i*00*mn*	S*i* *exp*	Transmit *exp* into S*i*
041*i*00*mn*	S*i* #*exp*	Transmit ones complement of *exp* into S*i*
042*ijk*	S*i* <*exp*	Form ones mask in S*i* *exp* bits from right; *exp* = 100_8 - *jk*.
042*ijk*[b]	S*i* #>*exp*	Form zeroes mask in S*i* *exp* bits from left; *exp* = *jk*
042*i*77[b]	S*i* 1	Enter 1 into S*i* register.
042*i*00[b]	S*i* -1	Enter -1 into S*i* register.
043*ijk*	S*i* >*exp*	Form ones mask in S*i* *exp* bits from left; *exp* = *jk*
043*ijk*[b]	S*i* #<*exp*	Form zeroes mask in S*i* *exp* bits from right; *exp* = 100_8 -jk
043*i*00[b]	S*i* 0	Clear the S*i* register.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
044*ijk*	S*i* S*j*&S*k*	Transmit the logical product of (S*j*) and (S*k*) to S*i*.
044*ij*0[b]	S*i* S*j*&SB	Transmit the sign bit of (S*j*) to S*i*.
044*ij*0[b]	S*i* SB&S*j*	Transmit the sign bit of (S*j*) to S*i* ($j \neq 0$).
045*ijk*	S*i* #S*k*&S*j*	Transmit the logical product of (S*j*) and complement of (S*k*) to S*i*.
045*ij*0[b]	S*i* #SB&S*j*	Transmit the (S*j*) with sign bit cleared to S*i*.
046*ijk*	S*i* S*j*\S*k*	Transmit the logical difference of (S*j*) and (S*k*) to S*i*.
046*ij*0[b]	S*i* S*j*\SB	Toggle the sign bit of (S*j*), then transmit to S*i*.
046*ij*0[b]	S*i* SB\S*j*	Toggle the sign bit of (S*j*), then transmit to S*i* ($j \neq 0$)
047*ijk*	S*i* #S*j*\S*k*	Transmit the logical equivalence of (S*k*) and (S*j*) to S*i*.
047*i*0*k*[b]	S*i* #S*k*	Transmit the one's complement of (S*k*) to S*i*.
047*ij*0[b]	S*i* #S*j*\SB	Transmit the logical equivalence of (S*j*) and sign bit to S*i*.
047*ij*0[b]	S*i* #SB\S*j*	Transmit the logical equivalence of (S*j*) and sign bit to S*i* ($j \neq 0$).
047*i*00[b]	S*i* #SB	Transmit the one's complement of sign bit into S*i*.
050*ijk*	S*i* S*j*!S*i*&S*k*	Transmit the logical product of (S*i*) and (S*k*) complement ORed with the logical product of (S*j*) and (S*k*) to S*i*.
050*ij*0[b]	S*i* S*j*!S*i*&SB	Transmit the scalar merge of (S*i*) and sign bit of (S*j*) to S*i*.
051*ijk*	S*i* S*j*!S*k*	Transmit the logical sum of (S*j*) and (S*k*) to S*i*.
051*i*0*k*[b]	S*i* S*k*	Transmit the (S*k*) to S*i*.
051*ij*0[b]	S*i* S*j*!SB	Transmit the logical sum of (S*j*) and sign bit to S*i*.
051*ij*0[b]	S*i* SB!S*j*	Transmit the logical sum of (S*j*) and sign bit to S*i* ($j \neq 0$).
051*i*00[b]	S*i* SB	Transmit the sign bit into S*i*.
052*ijk*	S0 S*i* <*exp*	Shift (S*i*) left *exp* places to S0; *exp* = *jk.*
053*ijk*	S0 S*i* >*exp*	Shift (S*i*) right *exp* places to S0; *exp* = 100_8-*jk*.
054*ijk*	S*i* S*i* <*exp*	Shift (S*i*) left *exp* places to S*i*; *exp* = *jk*.
055*ijk*	S*i* S*i* >exp	Shift (S*i*) right *exp* places to S*i*; *exp* = 100_8-*jk*.
056*ijk*	S*i* S*i*,S*j* <A*k*	Shift (S*i*) and (S*j*) left by (A*k*) places to S*i*.
056*ij*0[b]	S*i* S*i*,S*j* <1	Shift (S*i*) and (S*j*) left one place to S*i*.
056*i*0*k*[b]	S*i* S*i*<A*k*	Shift (S*i*) left (A*k*) places to S*i*.
057*ijk*	S*i* S*j*,S*i* >A*k*	Shift (S*j*) and (S*i*) right by (A*k*) places to S*i*.
057*ij*0[b]	S*i* S*j*,S*i* >1	Shift (S*j*) and (S*i*) right one place to S*i*.
057*i*0*k*[b]	S*i* S*i* >A*k*	Shift (S*i*) right (A*k*) places to S*i*.
060*ijk*	S*i* S*j*+S*k*	Transmit the integer sum of (S*j*) and (S*k*) to S*i*.
061*ijk*	S*i* S*j*-S*k*	Transmit the integer difference of (S*j*) and (S*k*) to S*i*.
061*i*0*k*[b]	S*i* -S*k*	Transmit the negative of (S*k*) to S*i*.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
062*ijk*	S*i* S*j*+FS*k*	Transmit the floating-point sum of (S*j*) and (S*k*) to S*i*.
062*i*0*k*[b]	S*i* +FS*k*	Transmit the normalized (S*k*) to S*i*.
063*ijk*	S*i* S*j*-FS*k*	Transmit the floating-point difference of (S*j*) and (S*k*) to S*i*.
063*i*0*k*[b]	S*i* -FS*k*	Transmit the normalized negative of (S*k*) to S*i*.
064*ijk*	S*i* S*j**FS*k*	Transmit the floating-point product of (S*j*) and (S*k*) to S*i*.
065*ijk*	S*i* S*j**HS*k*	Transmit the half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*.
066*ijk*	S*i* S*j**RS*k*	Transmit the rounded floating-point product of (S*j*) and (S*k*) to S*i*.
067*ijk*	S*i* S*j**IS*k*	Transmit the reciprocal iteration: 2-(S*j*)*(S*k*)to S*i*.
070*ij*0	S*i* /HS*j*	Transmit the floating-point reciprocal approximation of (S*j*) to S*i*.
070*ij*6	S*i* S*j**BT	Transmit bit-matrix product of (S*j*) and transpose of (BMM) to S*i*
071*i*0*k*	S*i* A*k*	Transmit (A*k*) to S*i* with no sign extension.
071*i*1*k*	S*i* +A*k*	Transmit (A*k*) to S*i* with sign extension.
071*i*2*k*	S*i* +FA*k*	Transmit (A*k*) to S*i* as unnormalized floating-point number.
071*i*30	S*i* 0.6	Transmit 0.75 x 2^{48} as normalized floating-point constant into S*i*.
071*i*40	S*i* 0.4	Transmit 0.5 as normalized floating-point constant into S*i*.
071*i*50	S*i* 1.0	Transmit 1.0 as normalized floating-point constant into S*i*.
071*i*60	S*i* 2.0	Transmit 2.0 as normalized floating-point constant into S*i*.
071*i*70	S*i* 4.0	Transmit 4.0 as normalized floating-point constant into S*i*.
072*i*00	S*i* RT	Transmit (RTC) to S*i*.
072*i*02	S*i* SM	Transmit (SM) to S*i*.
072*ij*3	S*i* ST*j*	Transmit (ST*j*) to S*i*.
073*i*00	S*i* VM	Transmit (VM) to S*i*.
073*i*11[a, c]		Read the performance counter into S*i*.
073*i*21[a, c]		Increment upper performance counter.
073*i*31[a, c]		Clear all maintenance modes.
073*i*61[a, c]		Increment current performance counter (lower).
073*i*01	S*i* SR0	Transmit (SR0) to S*i*.
073*i*02	SM S*i*	Transmit (S*i*) to SM.
073*ij*3	ST*j* S*i*	Transmit (S*i*) to ST*j*.
074*ijk*	S*i* T*jk*	Transmit (T*jk*) to S*i*.
075*ijk*	T*jk* S*i*	Transmit (S*i*) to T*jk*.
076*ijk*	S*i* V*j*,A*k*	Transmit (V*j* element (A*k*)) to S*i*.
077*ijk*	V*i*,A*k* S*j*	Transmit (S*j*) to V*i* element (A*k*).

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
077*i*0*k*[b]	V*i*,A*k* 0	Clear element (A*k*) of register V*i*.
10*hi*00*mn*	A*i exp*,A*h*	Load from address ((A*h*) + *exp*) to A*i* (*h* ≠ 0).
100*i*00*mn*	A*i exp*,0	Load from address (*exp*) to A*i*.
100*i*00*mn*	A*i exp*,	Load from address (*exp*) to A*i*.
10*hi*0000	A*i* ,A*h*	Load from address (A*h*) to A*i* (*h* ≠ 0).
11*hi*00*mn*	*exp*,A*h* A*i*	Store (A*i*) to address (A*h*) + *exp* (*h* ≠ 0).
110*i*00*mn*	*exp*,0 A*i*	Store (A*i*) to address *exp.*
110*i*00*mn*	*exp*, A*i*	Store (A*i*) to address *exp*.
11*hi*0000	,A*h* A*i*	Store (A*i*) to address (A*h*) (*h* ≠ 0).
12*hi*00*nm*	S*i exp*,A*h*	Load from address ((A*i*) + *exp*) to S*i* (*h* ≠ 0).
120*i*00*mn*	S*i exp*,0	Load from address (*exp*) to S*i*.
120*i*00*mn*	S*i exp,*	Load from address (*exp*) to S*i*.
12*hi*0000	S*i* ,A*h*	Load from address (A*h*) to S*i* (*h* ≠ 0).
13*hi*00*mn*	*exp*,A*h* S*i*	Store (S*i*) to address (A*h*) + *exp* (*h* ≠ 0).
130*i*00*mn*	*exp*,0 S*i*	Store (S*i*) to address *exp*.
130*i*00*mn*	*exp*, S*i*	Store (S*i*) to address *exp*.
13*hi*0000	,A*h* S*i*	Store (S*i*) to address (A*h*) (*h* ≠ 0).
140*ijk*	V*i* S*j*&V*k*	Transmit logical products of (S*j*) and (V*k* elements) to V*i* elements.
141*ijk*	V*i* V*j*&V*k*	Transmit logical products of (V*j* elements) and (V*k* elements) to V*i* elements.
142*ijk*	V*i* S*j*!V*k*	Transmit logical sums of (S*j*) and (V*k* elements) to V*i* elements.
142*i*0*k*[b]	V*i* V*k*	Transmit (V*k* elements) to V*i* elements.
143*ijk*	V*i* V*j*!V*k*	Transmit logical sums of (V*j* elements) and (V*k* elements) to V*i* elements.
144*ijk*	V*i* S*j*\V*k*	Transmit logical differences of (S*j*) and (V*k* elements) to V*i* elements.
145*ijk*	V*i* V*j*\V*k*	Transmit logical differences of (V*j* elements) and (V*k* elements) to V*i* elements.
145*iii*[b]	V*i* 0	Clear V*i* elements.
146*ijk*	V*i* S*j*!V*k*&VM	Transmit (S*j*) if VM bit = 1; (V*k* element) if VM bit = 0 to V*i*.
146*i*0*k*[b]	V*i* #VM&V*k*	Transmit vector merge of (V*k* elements) and 0 to V*i*.
147*ijk*	V*i* V*j*!V*k*&VM	Transmit (V*j* elements) if VM bit = 1; (V*k* element) if VM bit = 0 to V*i* elements.
150*ijk*	V*i* V*j*<A*k*	Shift (V*j* elements) left by (A*k*) places to V*i* elements.
150*ij*0[b]	V*i* V*j*<1	Shift (V*j* elements) left one place to V*i* elements.
151*ijk*	V*i* V*j*>A*k*	Shift (V*j* elements) right by (A*k*) places to V*i* elements.
151*ij*0[b]	V*i* V*j*>1	Shift (V*j* elements) right one place to V*i* elements.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
152*ijk*	V*i* V*j*,V*j*<A*k*	Double shift (V*j* elements) left (A*k*) places to V*i* elements.
152*ij*0[b]	V*i* V*j*,V*j*<1	Double shift (V*j* elements) left one place to V*i* elements.
153*ijk*	V*i* V*j*,V*j*>A*k*	Double shift (V*j* elements) right (A*k*) places to V*i* elements.
153*ij*0[b]	V*i* V*j*,V*j*>1	Double shift (V*j* elements) right one place to V*i* elements.
154*ijk*	V*i* S*j*+V*k*	Transmit integer sums of (S*j*) and (V*k* elements) to V*i* elements.
155*ijk*	V*i* V*j* + V*k*	Transmit integer sums of (V*j* elements) and (V*k* elements) to V*i* elements.
156*ijk*	V*i* S*j*-V*k*	Transmit integer differences of (S*j*) and (V*k* elements) to V*i* elements.
156*i*0*k*[b]	V*i* -V*k*	Transmit two's complement of (V*k* elements) to V*i* elements.
157*ijk*	V*i* V*j*-V*k*	Transmit integer differences of (V*j* elements) and (V*k* elements) to V*i* elements.
160*ijk*	V*i* S*j**FV*k*	Transmit floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
161*ijk*	V*i* V*j**FV*k*	Transmit floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
162*ijk*	V*i* S*j**HV*k*	Transmit half-precision rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
163*ijk*	V*i* V*j**HV*k*	Transmit half-precision rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
164*ijk*	V*i* S*j**RV*k*	Transmit rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
165*ijk*	V*i* V*j**RV*k*	Transmit rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
166*ijk*	V*i* S*j**V*k*	Transmit 32-bit integer product of (S*j*) and (V*k* elements) to V*i* elements.
167*ijk*	V*i* V*j**IV*k*	Transmit reciprocal iterations: 2-(V*j* elements)*(V*k* elements) to V*i* elements.
170*ijk*	V*i* S*j*+FV*k*	Transmit floating-point sums of (S*j*) and (V*k* elements) to V*i* elements.
170*i*0*k*[b]	V*i* +FV*k*	Transmit normalized (V*k* elements) to V*i* elements.
171*ijk*	V*i* V*j*+FV*k*	Transmit floating-point sums of (V*j* elements) and (V*k* elements) to V*i* elements.
172*ijk*	V*i* S*j*-FV*k*	Transmit floating-point differences of (S*j*) and (V*k* elements) to V*i* elements.
172*i*0*k*[b]	V*i* -FV*k*	Transmit normalized negative of (V*k* elements) to V*i* elements.
173*ijk*	V*i* V*j*-FV*k*	Transmit floating-point differences of (V*j* elements) and (V*k* elements) to V*i* elements.
174*ij*0	V*i* /HV*j*	Transmit floating-point reciprocal approximation of (V*j* elements) to V*i* elements.
174*ij*1	V*i* PV*j*	Transmit population count of (V*j* elements) to V*i* elements.

Table 34. Quick-reference Table of CPU Instructions (continued)

Machine Instruction	CAL Syntax	Description
174*ij*2	V*i* QV*j*	Transmit population count parity of (V*j* elements) to V*i* elements.
174*ij*3	V*i* ZV*j*	Transmit the leading-zero count of (V*j* elements) to V*i* elements
1740*j*4	BMM V*j*	Transmit (V*j* elements) to BMM. Locations that are not loaded are cleared.
174*ij*6	V*i* V*j**BT	Transmit the bit-matrix product of (V*j* elements) and transpose of (BMM) to V*i* elements.
1750*j*0	VM V*j*,Z	Set VM bit if (V*j* element) = 0.
1750*j*1	VM V*j*,N	Set VM bit if (V*j* element) ≠ 0.
1750*j*2	VM V*j*,P	Set VM bit if (V*j* element) ≥ 0.
1750*j*3	VM V*j*,M	Set VM bit if (V*j* element) < 0 (V*j* is negative).
175*ij*4	V*i*,VM V*j*,Z	Set VM bit if (V*j* elements) = 0; also, the compressed indices of the V*j* element = 0 are stored in V*i*.
175*ij*5	V*i*,VM V*j*,N	Set VM bit if (V*j* elements) ≠ 0; also, the compressed indices of the V*j* element ≠ 0 are stored in V*i*.
175*ij*6	V*i*,VM V*j*,P	Set VM bit if (V*j* elements) ≥ 0; also, the compressed indices of the V*j* element ≥ 0 are stored in V*i*.
175*ij*7	V*i*,VM V*j*,M	Set VM bit if (V*j* elements) < 0; also, the compressed indices of the V*j* element < 0 are stored in V*i*.
176*i*0*k*	V*i* ,A0,A*k*	Load (VL) words from address (A0) incremented by (A*k*) to V*i* elements.
176*i*00	V*i* ,A0,1	Load (VL) words from address (A0) incremented by 1 to V*i* elements.
176*i*1*k*	V*i* ,A0,V*k*	Load (VL) words from address ((A0) + (V*k* elements)) to V*i* elements.
1770*jk*	,A0,A*k* V*j*	Store (VL) words from (V*j* elements) to address (A0) incremented by (A*k*).
1770*j*0	,A0,1 V*j*	Store (VL) words from (V*j* elements) to address (A0) incremented by 1.
1771*jk*	,A0,V*k* V*j*	Store (VL) words from (V*j* elements) to address (A0) + (V*k* elements).

[a] These instructions are privileged to monitor mode.

[b] Special CAL syntax.

[c] These instructions are not supported by CAL Version 2.

[d] Bit 2 of the *i* field is equal to 0.

[e] The value of the expression determines which instructions are generated.

[f] These instructions are for SV1s with enhanced processors and SV1ex machines only.

Notational Conventions

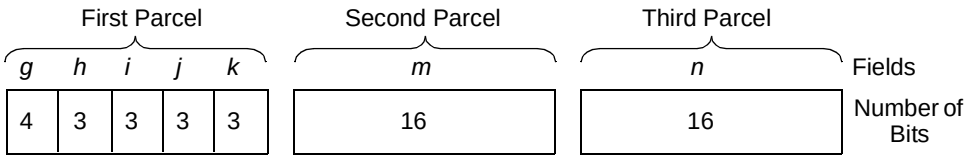
This section uses the following conventions:

- All numbers are decimal numbers unless otherwise indicated.
- Letters X or x or *x* represent an unused value.
- Register bits are numbered from low-order (least significant) to high-order (most significant).
- The letter n represents a specified value.
- The value in parentheses ( ) specifies the contents of a register or memory location as designated by value.
- Variable parameters are in *italic* type.
- Vector mask bit 63 corresponds to vector element 0, and bit 0 corresponds to vector element 63.

Instruction Formats

Instructions can be 1 parcel (16 bits), 2 parcels (32 bits), or 3 parcels (48 bits) long. Instructions are packed 4 parcels per word and parcels are numbered 0 through 3 from left to right. Any parcel position can be addressed in branch instructions. A 2- or 3-parcel instruction begins in any parcel of a word and can span a word boundary. For example, a 2-parcel instruction that begins in parcel 3 of a word ends in parcel 0 of the next word. No padding of word boundaries is required. Figure 47 shows the general instruction format.

Figure 47. General Instruction Format



The first parcel contains five fields, and the second and third parcels each contain a single field. Four variations of this format use the fields differently. The following subsections describe the formats of the following variations:

- 1-parcel instruction format with discrete *j* and *k* fields

- 1-parcel instruction format with combined *j* and *k* fields
- 2-parcel instruction format with combined *i*, *j*, *k*, and *m* fields
- 3-parcel instruction format with combined *m* and *n* fields

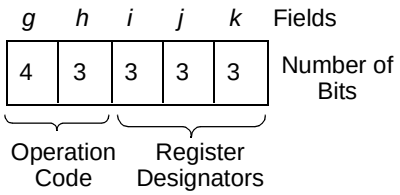
1-parcel Instruction Format with Discrete *j* and *k* Fields

The most common of the 1-parcel instruction formats uses the *i*, *j*, and *k* fields as individual designators for operand and result registers (refer to Figure 48). The *g* and *h* fields define the operation code, the *i* field designates a result register, and the *j* and *k* fields designate operand registers. Some instructions ignore one or more of the *i*, *j*, and *k* fields.

The following types of instructions use this format:

- Arithmetic
- Logical
- Vector shift
- Scalar double-shift
- Floating-point constant

Figure 48. 1-parcel Instruction Format with Combined j and k Fields

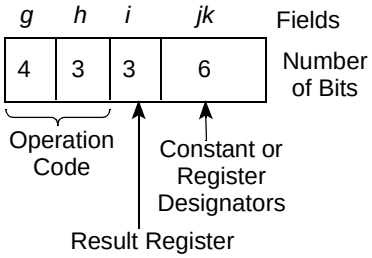


1-parcel Instruction Format with Combined *j* and *k* Fields

Some 1-parcel instructions use the *j* and *k* fields as a combined 6-bit field (refer to Figure 49). The *g* and *h* fields contain the operation code, and the *i* field is usually a destination register. The combined *j* and *k* fields usually contain a constant or an intermediate address (B) or intermediate scalar (T) register designator. The 005 branch instruction and the following types of instructions use the 1-parcel instruction format with combined *j* and *k* fields:

- 6-bit constant
- B or T register block memory transfer
- B or T register data transfer with address (A) or scalar (S) register
- Scalar single-shift
- Scalar mask

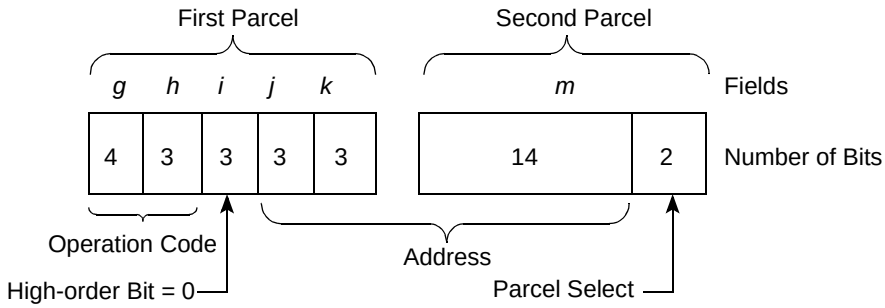
Figure 49. 1-parcel Instructions with j and k as a Combined 6-bit Field



2-parcel Instruction Format with Combined *i*, *j*, *k*, and *m* Fields

This 2-parcel format uses the combined *i*, *j*, *k*, and *m* fields to contain a 24-bit address that allows branching to an instruction parcel (refer to Figure 50). A 7-bit operation code (*gh*) is followed by an *ijkm* field. The high-order bit of the *i* field is equal to 0.

Figure 50. 2-parcel Instruction Format with Combined i, j, k, and m Fields



3-parcel Instruction Format with Combined *m* and *n* Fields

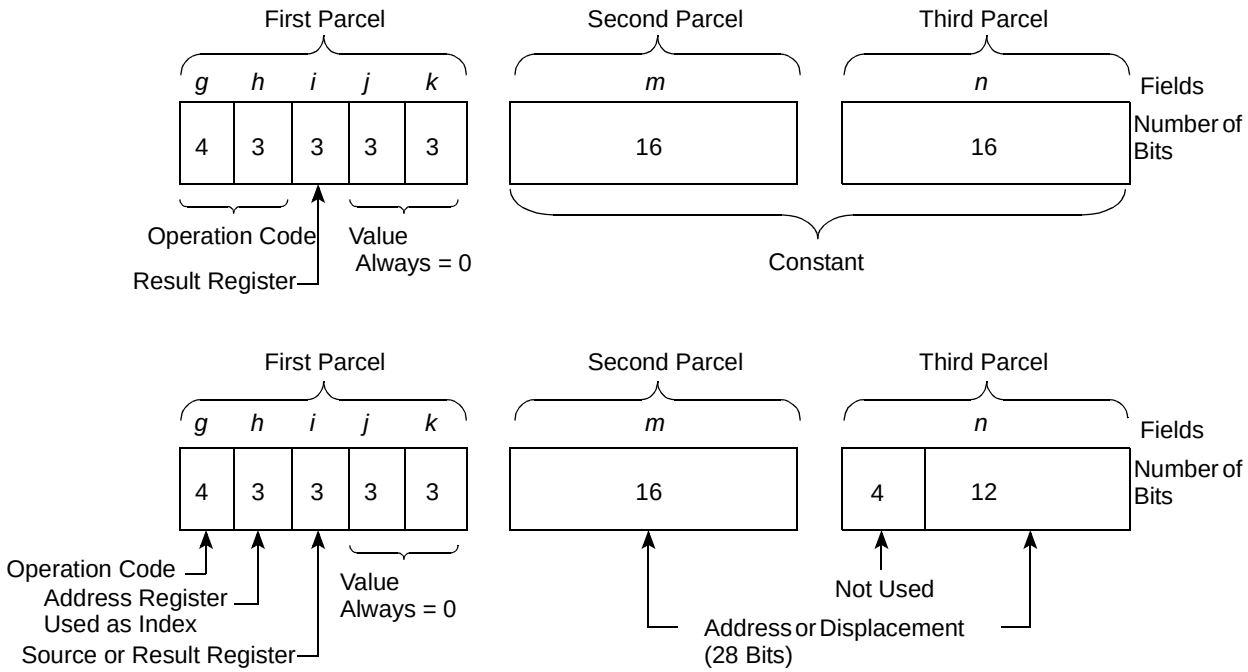
The format for a 32-bit immediate constant uses the combined *m* and *n* fields to hold the constant. The 7-bit *g* and *h* fields contain an operation code, and the 3-bit *i* field designates a result register; the *j* and *k* fields are a constant 0. The instructions that use this format transfer the 32-bit *mn* constant to an A or S register.

Note: The *m* field of the 3-parcel instruction contains bits 0 through 15 of the expression, while the *n* field contains bits 16 through 31 of the expression. When the instruction is assembled, the *mn* field is reversed and actually appears as the *nm* field when used as an expression.

When 3-parcel instructions are used to generate memory addresses, bits 31 through 0 of the *nm* field are used to calculate memory addresses. Refer to "Calculating Absolute Memory Address" for additional information. This format uses the 4-bit *g* field for an operation code, the 3-bit *h* field to designate an address index register, and the 3-bit *i* field to designate a source or result register.

Figure 51 shows the two applications for the 3-parcel instruction format with combined *m* and *n* fields. Remember that the *m* and *n* fields are reversed when a 3-parcel instruction is assembled.

Figure 51. 3-parcel Instruction Format with Combined m and n Fields



Special Register Values

If the S0 and A0 registers are referenced in the *h*, *j*, or *k* fields of certain instructions, the contents of the respective register are not used; instead, a special operand is generated. The special operand is available regardless of existing A0 or S0 reservations (and in this case is not checked). This special operand does not alter the actual value of the S0 or A0 register. If registers S0 or A0 are used in the *i* field as the operand, the actual value of the register is

provided. Cray Assembly Language (CAL) issues a caution-level error message for A0 or S0 when 0 does not apply to the *i* field. Table 35 lists the special register values.

Table 35. Special Register Values

Field	Operand Value
A*h*, *h* = 0	0
A*j*, *j* = 0	0
A*k*, *k* = 0	1
S*j*, *j* = 0	0
S*k*, *k* = 0	Bit 63 = 1

Monitor Mode Instructions

The monitor mode instructions (channel control, set real-time clock, programmable clock interrupts, and so on) perform specialized functions that are useful to the operating system. These instructions run only when the CPU is operating in monitor mode. If a monitor mode instruction issues while the CPU is not in monitor mode, it is treated as a no-operation instruction.

Special CAL Syntax Forms

Certain machine instructions can be generated from two or more different CAL instructions. Any of the operations performed by special instructions can be performed by instructions in the basic CAL instruction set.

For example, the following CAL instructions generate instruction 002000, which enters a 1 into the vector length (VL) register:

```
VL A0
VL 1
```

The first instruction is the basic form of the enter VL instruction, which takes advantage of the special case where (A*k*) = 1 if *k* = 0. The second instruction is a special syntax form that provides the programmer with a more convenient notation for the special case.

In several cases, a single CAL syntax can generate several different machine instructions. These cases are used for entering the value of an expression into an A register or an S register, or for shifting S register contents. The assembler determines which instruction to generate from characteristics of the expression.

The following subsection identifies CAL instructions that have a special syntax form.

CPU Instruction Descriptions

This subsection describes all the instructions that the mainframe uses. The instruction descriptions use acronyms and abbreviations that are defined in previous sections. The following information is included with each instruction description:

- Special cases
- Hold issue conditions
- Execution time
- Description

In some instructions, register designators are prefixed by the following letters that have special meaning to the assembler. The letters and their meanings are as follows:

Letter	Description
F	Floating-point operation
H	Half-precision floating-point operation
I	Reciprocal iteration
P	Population count
Q	Parity count
R	Rounded floating-point operation
Z	Leading-zero count

The following list defines some of the notations that the instruction set uses:

Character	Description
+	Arithmetic sum of specified registers
-	Arithmetic difference of specified registers
*	Arithmetic product of specified registers
/	Reciprocal approximation
#	Use one's complement
>	Shift value or form mask from left to right
<	Shift value or form mask from right to left
&	Logical product of specified registers
!	Logical sum of specified registers
\	Logical difference of specified registers

An expression (*exp*) occupies the *jk*, *jkm*, *ijkm*, or *mn* field. The *h*, *i*, *j*, and *k* designators indicate the field of the machine instruction into which the register designator constant or symbol value is placed.

Functional Units Instruction Summary

Instructions other than simple transmit or control operations are performed by specialized hardware known as functional units. The following instructions are performed by each of the functional units.

.

Functional Unit	Instructions
Address add (integer)	030, 031
Address multiply (integer)	032
Scalar add (integer)	060, 061
Scalar logical	042 through 051
Scalar shift	052 through 057
Scalar pop/parity/leading zero	026, 027
Vector add (integer)	154 through 157
Vector logical	140 through 147, 175
Second vector logical	140 through 145
Vector shift	150 through 153
Vector pop/parity/leading-zero	174*ij*1, 174*ij*2, 174*ij*3
Floating-point add	062, 063, 170 through 173
Floating-point multiply	064 through 067, 160 through 167
Floating-point reciprocal	070*ij*0, 174*ij*0
Memory (scalar)	100 through 130
Bit-matrix multiply	070*ij*6, 1740*j*4, 174*ij*6
Memory (vector)	176, 177

Instruction 000000

Machine Instruction	CAL Syntax	Description
000000	ERR	Error exit

Special Cases

There are no special cases.

Hold Issue Conditions

The instruction holds issue if any A, S, or V register is reserved or if an instruction fetch operation is in progress.

Execution Time

The 000 instruction issues in 1 CP. Following the instruction issue, an additional 220 CPs are required for an exchange sequence (108 CPs) and a fetch operation (112 CPs). Memory conflicts during the exchange sequence cause additional delays.

Description

The 000 instruction is treated as an error condition, and an exchange sequence occurs when the instruction is issued. The contents of the instruction buffers are voided by the exchange sequence. If the monitor mode is not in effect, the error exit flag in the F register is set. All instructions issued before this instruction are completed.

When the results of previously issued instructions arrive at the operating registers, an exchange occurs to the exchange package that is designated by the contents of the XA register. The program address that is stored during the final exchange sequence is the contents of the P register advanced by one count (the address of the instruction following the error exit instruction).

Instruction 000 is not generally used in program code. This instruction stops execution of an incorrectly coded program that branches to an unused area of memory (if memory was backgrounded with 0's) or into a data area (if data is positive integers, right justified ASCII, or floating-point 0's).

Instructions 0010 through 0013

Machine Instruction	CAL Syntax	Description
0010*jk*[a]	CA,A*j* A*k*	Set the CA register for the channel indicated by (A*j*) to (A*k*) and activate the channel.
001000	PASS	This is a no-operation instruction.
0011*jk*[a]	CL,A*j* A*k*	Set the CL register for the channel indicated by (A*j*) to (A*k*) address.
0012*j*0[a]	CI,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); clear device master clear (output channels only).
0012*j*1[a]	MC,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); set device master clear (output channels only); clear device ready-held (input channels only).
0013*j*0[a]	XA A*j*	Transmit (A*j*) to the XA register.

[a] These instructions are privileged to monitor mode.

Special Cases

The following special cases exist for instructions 0010 through 0013:

- If the program is not in monitor mode, these instructions become no-operation instructions with all A*j* or A*k* register hold issue conditions remaining effective.
- For instructions 0010, 0011, and 0012, if *j* = 0, the instruction performs no operation.
- For instructions 0010, 0011, and 0012, if *k* = 0, the CA or CL register is set to 1.
- Valid channel numbers are Y1 channel numbers 20 through 117_8 on the largest system (excluding channels 100_8 and 101_8).
- For instruction 0013, if *j* = 0, the XA register is cleared.

Hold Issue Conditions

The instructions hold issue when the A*j* register is reserved (except A0).

Instructions 0010 through 0011 hold issue when the A*k* register is reserved (except A0); instructions 0010 through 0012 hold issue when there is a shared register access conflict or if the JS ASIC buffer is full.

Execution Time

The instruction issue time for instructions 0010 through 0013 is 1 CP.

Note: In monitor mode, the software must ensure that only one CPU at a time is servicing an I/O channel.

Description

Instructions 0010 through 0013 are privileged to monitor mode and provide operations that are useful to the operating system. Functions are selected through the *i* designator. Instructions are treated as pass instructions if the monitor mode bit is not set. A monitor program activates a user job by initializing the XA register to point to the user's job exchange package and then executing a normal exit instruction.

When the *j* designator is 0, the functions are executed as pass instructions. When the *k* designator is 0, the CA register or CL register is set to 1. Valid channel numbers are 20 through 117 for Y1 channels on the largest systems with channels 100_8 and 101_8 excluded. GigaRing I/O uses channels X4 through X7 of the same channel range (24 through 27, 34 through 37, and so on through channels 114 through 117.

Instructions 0010, 0011, and 0012 control operation of the I/O channels. Each Y1 channel has a CA and a CL register to direct channel activity. The CA register contains the address of the current channel word; the CL register specifies the limit address. When the channel is programmed, the CL register is initialized first and then the CA register is set, which activates the channel. As the transfer continues, the CA register increments toward the CL register. When the contents of the CA register are equal to the contents of the CL register, the transfer is complete for all words from the initial contents of the CA register through the contents of the CL register minus 1.

The 0010*jk* instruction sets the CA register for the channel that is indicated by the contents of the A*j* register to the value in the A*k* register. The 0011*jk* instruction sets the CL register for the channel that is indicated by the contents of the A*j* register to the address that is specified in the A*k* register. The 0011*jk* instruction is usually issued before the 0010*jk* instruction is issued.

Instruction 0012*j*0 clears the interrupt and error flags for the channel that is indicated by the contents of the A*j* register. If the contents of the A*j* register represent an output channel, the device master clear is cleared.

Instruction 0012*j*1 also clears the interrupt and error flags for the channel that is indicated by the contents of the A*j* register. If the contents of the A*j* register represent an output channel, the device master clear is set. If the contents of the A*j* register represent an input channel, the device ready flag is cleared.

Instruction 0013*jk* transmits bits 13 through 4 of the A*j* register to the XA register. The XA register is cleared when the *j* designator is 0.

Instructions 0014 and 0016*j*1

Machine Instruction	CAL Syntax	Description
0014*j*0[a]	RT S*j*	Load the RTC register with (S*j*).
0014*j*1[a]	SIPI A*j*	Send an interprocessor interrupt request to CPU (A*j*).
001401[a]	SIPI	Send an interprocessor interrupt request to CPU 0.
001402[a]	CIPI	Clear the interprocessor interrupt.
0014*j*3[a]	CLN A*j*	Load the CLN register with (A*j*).
0014*j*4[a]	PCI S*j*	Load the II register with (S*j*).
001405[a]	CCI	Clear the programmable clock interrupt request.
001406[a]	ECI	Enable the programmable clock interrupt request.
001407[a]	DCI	Disable the programmable clock interrupt request.
0016*j*1[b]	IVC	Send invalidate cache request to CPU (A*j*).

[a] This instruction is privileged to monitor mode.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 0014 and 0016:

- If the program is not in monitor mode, these instructions perform no operation, and all S*j* or A*j* register hold issue conditions remain in effect.
- The RTC register will not be ready for some indeterminate number of cycles (over 100 CPs minimum with no delays).
- The following code ensures that RTC is ready:

```
RT      Sj
SBj     A0
JAZ     label
label Si  RT
```

Instruction 0014*j*0 is a global instruction and instruction 027*ij*7 is a local instruction. All local instructions are held in the JS ASIC until all global instructions are completed.

Hold Issue Conditions

The instruction holds issue for any of the following conditions:

- Instructions 0014*j*0, 0014*j*1, 0014*j*4, and 0016*j*1 hold issue when the A*j* register is reserved (except A0).
- Instructions 0014*j*0, 0014*j*1, and 0014*j*4 hold issue when the S*j* register is reserved (except S0).
- Instructions 0014*j*0, 0014*j*1, 0014*j*3, and 0016*j*1 hold issue when a shared register access conflict occurs or if the JS ASIC buffer is full.
- Instruction 0016*j*1 holds issue until an acknowledgment is received, which indicates that the cache in CPU (A*j*) is invalid.

Execution Time

The 0014 or 0016 instructions issue in 1 CP.

Description

The 0014 instruction performs specialized functions for managing the real-time and programmable clocks. These functions process interprocessor interrupt requests and cluster number operations. Instruction 0014 is privileged to monitor mode and is treated as a pass instruction if the monitor mode bit is not set.

The 0014*j*0 instruction loads the contents of the S*j* register into the RTC register. The RTC register is set to 0 when the *j* designator is 0.

The 0014*j*1 instruction sets the CPU interrupt request in the CPU that is specified by the contents of the A*j* register. If the CPU named in the contents of the A*j* register attempts to interrupt itself, the instruction performs no operation. If the other CPU is not in monitor mode, the interrupt-from-internal CPU flag sets in the F register, which causes an interrupt. The request remains until it is cleared when the receiving CPU issues instruction 001402. Instruction 001401 performs the same function as 0014*j*1, except that it sets the internal CPU interrupt request in CPU 0.

Instruction 001402 clears the internal CPU interrupt request that is set by another CPU.

The 0014*j*3 instruction sets the cluster number to the contents of the A*j* register to make 1 of 41_8 cluster selections. A cluster number of 0 causes all shared and semaphore register operations to be no-operation instructions (except SB, ST, or SM register reads, which return a zero value to the A*i* or S*i* register). A nonzero cluster has a separate set of SM, SB, and ST registers. A cluster number larger than 91 (octal) produces undefined results.

The 0014*j*4 instruction loads the low-order 32 bits from the S*j* register into the Interrupt Interval (II) register and programmable clock. The programmable clock is a 32-bit counter that decrements by 1 each system CP until the contents of the counter equal 0. The programmable clock interrupt request is then set. The counter is then set to the interval value held in the II register and the counter repeats the countdown to 0. When a programmable clock interrupt request is set, it remains set until a 001405 instruction is executed. Refer to the "Interrupt Interval Register" subsection for more information about the II register.

The 001405 instruction clears the programmable clock interrupt request if the request is set previously when the interrupt countdown (ICD) counts down to 0.

The 001406 instruction enables repeated programmable clock interrupt requests at a rate determined by the value stored in the II register.

The 001407 instruction disables repeated programmable clock interrupt requests until a 001406 instruction is executed to enable the requests.

The 0016*j*1 instruction invalidates the cache in the CPU that is specified by the contents of the A*j* register. If the CPU named in the contents of the A*j* register attempts to invalidate its own cache, the instruction performs no operation.

Instructions 0015 through 001551

Machine Instruction	CAL Syntax	Description
0015*j*0[a, c]		Select performance monitor.
001511[a, c]		This is a no-op instruction.
001521[a, c]		Disable port D I/O error correction.
001541[a, c]		Enable replacement of checkbyte with data on ports A, B, and D for writes and the replacement of data with checkbytes on ports for reads from memory.
001551[a, c]		Replace check bits with V*k* data bits on the path to the VAB ASIC during execution of instruction 1771*jk*.

[a] These instructions are privileged to monitor mode.

[c] These instructions are not supported by CAL Version 2.

Special Cases

The special case for instruction 0015 is that if the program is not in monitor mode, these instructions perform no operation, and all hold issue conditions remain in effect.

Hold Issue Conditions

The instruction holds issue when any A*j* register is reserved (except A0).

Execution Time

The 0015 instruction issues in 1 CP.

Description

All 0015 instructions are privileged to monitor mode. Instruction 0015*j*0 selects one of four groups of hardware-related events to be monitored by the performance counters and clears all performance counter pointers. Allow a 50-CP delay before issuing another performance monitor instruction.

Instruction 001501 is not used by the SV1 series system. Instead, a configuration bit in the CA ASICs disables checkbyte generation and error correction for all CPU requests to memory. The 001511 instruction is a no-op instruction in the SV1 series system. Instruction 001541 applies to all CPU and I/O requests to memory. This instruction allows certain bits to be replaced in either the checkbyte or data field. During write operations, bits in the checkbyte are replaced with corresponding data bits. During read operations, the data bits are replaced with corresponding checkbyte bits. The following list shows how the bits are replaced:

Data Bit	Checkbyte Bit
0	64
8	65
16	66
24	67
32	68
40	69
48	70
56	71

Instruction 001551 allows certain bits to be replaced in the checkbyte with V*k* data during the execution of instruction 1771*jk*. Instruction 1771*jk* executes in the same manner as instruction 1770*jk*; the content of the A*k* register is the increment value, and the V*k* data is not used as the address. The following list shows which V*k* data bit replaces each checkbyte bit:

Data Bit	Checkbyte Bit
0	64
1	65
2	66
3	67
4	68
5	69
6	70
7	71

Instruction 0020

Machine Instruction	CAL Syntax	Description
00200*k*	VL A*k*	Transmit (A*k*) to VL register.
002000[b]	VL 1	Transmit 1 to VL register.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instruction 0020:

- The maximum vector length is $64(100_8)$.
- If $k = 0$, (A*k*) = 1.
- If $k \neq 0$ and (A*k*) = 0 or a multiple of 100 (octal), then register VL = 100 (octal).

Hold Issue Conditions

The instruction holds issue under any of the following conditions:

- The A*k* register is reserved (except A0).
- A 077 instruction issued in the previous CP.

- The vector instruction queue is full.

VIR Hold Issue Conditions

This instruction issues without register or functional unit delays. Instruction 070*ij*6, 073*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The instruction issue time for the 0020 instruction is 1 CP.

The VIR instruction issue time for the 0020 is 3 CPs.

Description

The low-order 6 bits of the contents of the A*k* register are entered into the VL register; the seventh bit of the VL register is set if the 6 low-order bits of the contents of the A*k* register equal 0. For example, if the contents of the A*k* register equal 0 or a multiple of 100 (octal), then VL = 100 (octal). The contents of the VL register will always be between 1 and 100 (octal).

Instruction 002000 transmits the value of 1 to the VL register.

The 0020 instruction is issued from the VIR in sequence with any other vector instructions.

Instructions 0021 through 0027

Machine Instruction	CAL Syntax	Description
002100	EFI	Enable interrupt on floating-point error.
002200	DFI	Disable interrupt on floating-point error.
002210	CBL	Clear the Bit-matrix Loaded (BML) bit.
002300	ERI	Enable interrupt on operand range error.
002400	DRI	Disable interrupt on operand range error.
002500	DBM	Disable bidirectional memory transfers.
002600	EBM	Enable bidirectional memory transfers.
002700	CMR	Complete memory references.

Hold Issue Conditions

Instructions 002100 and 002200 hold issue if the floating-point functional units are busy or if the vector unit is not quiet.

Instructions 002300 through 002400 have no hold issue conditions.

Instruction 002700 holds issue if ports A or B are busy and if memory is busy; the hold issue lasts for a minimum of 45 CPs with no cache or memory activity; 77 CPs minimum with an A register load from cache, or, 87 CPs minimum with an A register store to memory.

Execution Time

Instructions 0021 through 0027 issue in 1 CP.

Description

Instructions 002100 and 002200 set and clear the interrupt-on-floating-point (IFP) error bit in the M register. When the IFP bit is set, it enables interrupts on floating-point range errors. These two instructions do not check the previous state of the flag. Either of these instructions also clears the floating-point error status bit.

The 002210 instruction clears the BML bit in the M register. The BML bit is automatically set by loading the BMM register with the 1740*j*4 instruction.

Instructions 002300 and 002400 set and clear the interrupt-on-operand range (IOR) error bit in the M register. These two instructions do not check the previous state of the IOR bit. When set, the IOR error bit enables interrupts on operand range errors.

Instructions 002500 and 002600 disable and enable the bidirectional memory mode. When this mode is enabled, block read and write operations can operate concurrently. When it is disabled, only block read operations can operate concurrently.

Instruction 002700 ensures completion of all memory references within the CPU that issues the instruction. Instruction 002700 does not issue until all previous memory references are confirmed to be complete. For example, a CPU is guaranteed to receive updated data when it issues a data load instruction after a 002700 instruction. The 002700 instruction synchronizes memory references between processors in conjunction with semaphore instructions.

Instructions 002703 through 002707

Machine Instruction	CAL Syntax	Description
002703	ETSI	Enable Test and Set Validate
002704	CPA	Hold if Port A or Port B busy
002705	CPR	Hold if Port A or Port B busy
002706	CPW	Hold instruction issue if block store or scalar load/store busy
002707	DTSI	Disable Test and Set Invalidate

Execution Time

Instructions 002703 and 002707 issue in 1 CP.

Execution time for instructions 002704 through 002706 varies depending on the time that is required to clear the memory ports in the PVC.

Description

Instruction 002703 sets bit 2 in the cache-enable field (bit 34 of word 7) of the exchange package; this allows any subsequent test-and-set instructions to invalidate cache.

Instructions 002704 and 002705 are identical. They perform a complete ports all (CPA) operation. The instructions hold issue until all previous memory operations exit the PV portion of the PVC (which guarantees memory ordering for this CPU). Compared to a complete memory references (CMR) operation, which guarantees that all previous memory operations are in memory, these instructions guarantee that no memory references **from this CPU** that are generated after the CPA can precede any memory references before the CPA.

Instruction 002706 performs a complete port write (CPW). The instruction holds issue until all previous block writes, vector writes, or scalar loads or stores exit the PV portion of the PVC (which guarantees ordering for this CPU). This instruction differs from CPA in that a block read or vector read can still be in the PV when this instruction issues.

Instruction 002707 clears bit 2 in the cache-enable field (bit 34 of word 7), which disallows any subsequent test-and-set instructions from invalidating cache.

Instructions 0030, 0034, 0036, and 0037

Machine Instruction	CAL Syntax	Description
0030*j*0	VM S*j*	Transmit (S*j*) to VM register.
003000[b]	VM 0	Clear VM register.
0034*jk*	SM*jk* 1, TS	Test and set semaphore *jk*, 0 ≤*jk* ≤Š 31_{10}.
0036*jk*	SM*jk* 0	Clear semaphore *jk*, 0 ≤*jk* ≥ 31_{10}.
0037*jk*	SM*jk* 1	Set semaphore jk, 0 ≥ jk ≥ 31_{10}.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 0030, 0034, 0036, or 0037:

- For instruction 0030*j*0, if *j* = 0 then (S*j*) = 0.
- Instructions 0034*jk*, 0036*jk*, and 0037*jk* perform no operation if CLN = 0.

Hold Issue Conditions

Instruction 0030*j*0 holds issue under any of the following conditions:

- The S*j* register is reserved (except S0).
- 077 instruction was issued in the previous CP.
- The vector instruction queue is full.

VIR Hold Issue Conditions

Instruction 0030*j*0 holds issue at the VIR under any of the following conditions:

- The primary vector logical unit is busy with 140-147 instructions.

- The primary vector logical unit is busy with 175 instruction.
- The VIR issued a 0030*j*0 instruction less than 4 CPs earlier.
- Instruction 070*ij*6, 073, 076, or 077 issued the previous CP.

Instruction 0034*jk* has the following hold issue conditions:

- This instruction holds issue when a shared register access conflict occurs or when the shared operation buffer is full.
- This instruction holds issue when the PV to JS interface is still busy due to a previous instruction.
- When the current cluster number does not equal 0 and SM*jk* is set, this instruction holds issue until a CPU in the same cluster clears the semaphore register.

Instructions 0036*jk* and 0037*ijk* have the following hold issue conditions:

- These instructions hold issue when a shared path access conflict occurs or if the shared operation buffer is full.
- These instructions hold issue when the PV to JS interface is still busy because of a previous instruction.

Execution Time

Instruction 0030 issues in 1 CP. The vector mask register is busy for 3 CPs for a 0030*j*0 instruction.

Instruction 0034 issues in a minimum of 77 CPs. The PV to JS shared interface remains busy for 1 additional CP. Instructions 0036 and 0037 issue in a minimum of 1 CP. The PV to JS interface remains busy for an additional 3 CPs.

Description

Instruction 0030*j*0 transmits the contents of the S*j* register into the VM register. The VM register is cleared if the *j* designator is 0 in instruction 003000. These instructions are used with the vector merge instructions (146 and 147), which perform operations that are determined by the contents of the VM register.

Instruction 0034*jk* tests and sets the semaphore (SM) register that is designated by the *jk* fields. There are thirty two 1-bit SM registers numbered SM0 through SM37 (octal); SM0 is the most significant semaphore register. If the SM register designated by the *jk* fields is set, this instruction holds issue until another CPU clears that SM register. If the SM register that is designated by the *jk* fields is clear, the instruction issues and sets the SM register. If all CPUs in a cluster are holding issue on a test and set instruction, the deadlock flag is set in the exchange package (if the system is not in monitor mode) and an exchange occurs.

Instruction 0034*jk* also invalidates cache for that CPU. The invalidation operation occurs at the beginning of the 0034*jk* instruction issue if enabled by the ECI bit in word 7 of the exchange package.

If an interrupt occurs while a test and set instruction is holding in the CIP register, the waiting-on-semaphore bit in the exchange package sets, the CIP and NIP registers clear, and an exchange occurs with the P register pointing to the test and set instruction.

Instruction 0036*jk* clears the SM register that is designated by the *jk* fields.

Instruction 0037*jk* sets the SM register that is designated by the *jk* fields.

Instruction 0040

Machine Instruction	CAL Syntax	Description
004000	EX	Normal exit from the operating system.

Special Cases

There are no special cases.

Hold Issue Conditions

The 0040 instruction holds issue when any A, S, or V register is reserved, if the vector unit is not quiet, or if an instruction fetch is in progress.

Execution Time

The 0040 instruction issues in 1 CP. Following the instruction issue, 220 CPs are required for an exchange sequence (108 CPs) and a fetch operation (112 CPs). Memory conflicts during the exchange sequence or fetch operation cause additional delays.

Description

Instruction 004 initiates an exchange sequence, which voids the contents of the instruction buffers. If the system is not in monitor mode, the normal exit flag in the F register sets. All instructions that issued before the 004 instruction are completed. Instruction 004 issues a monitor request from a user program or transfers control from a monitor program to another program.

When all results arrive at the operating registers of previously issued instructions, an exchange sequence occurs to the exchange package that is designated by the contents of the XA register. The program address that is stored in the exchange package advances one count from the address of the normal exit instruction.

Instruction 0050

Machine Instruction	CAL Syntax	Description
0050*jk*	J B*jk*	Jump to (B*jk*).

Special Cases

A special case may occur when instruction 0050*jk* executes as a 2-parcel instruction. The parcel that follows the single parcel of the 0050*jk* instruction is not used; however, a delay occurs if the second parcel is not in the instruction buffer.

Hold Issue Conditions

The 0050 instruction holds issue if any one of the following conditions occurs:

- A 025 instruction was issued in the previous CP.
- The second parcel is in a different buffer (a 3-CP delay occurs).
- The second parcel is not in an instruction buffer.
- Instruction 034 in progress with block length less than or equal to 100_8 and register B*jk* had not been written
- Instruction 034 is in progress with block length greater than 100_8.
- Instruction 035 is in progress.

Execution Time

The instruction issue times for the 0050 instruction are as follows:

- If the instruction parcel and following parcel are in the same buffer and the branch address is in a buffer, the issue time is 8 CPs.
- If the instruction parcel and the following parcel are both in a buffer and the branch address is not in a buffer, the issue time is a minimum of 112 CPs. Additional time is required if a memory conflict exists. The location of the branch address within the instruction buffer may add more CPs to this time.

Description

Instruction 005 sets the P register to the 24-bit parcel address specified by the contents of the B*jk* register, which causes the program to continue at that address. The instruction is used to return from a subroutine.

Instruction 0060

Machine Instruction	CAL Syntax	Description
006*ijkm*	J *exp*	Jump to *exp*.

Hold Issue Conditions

The 006 instruction holds issue if either one of the following conditions occurs:

- The second parcel is in a different buffer (the instruction holds issue for 3 CPs).
- The second parcel is not in a buffer.

Execution Time

Instruction issue times for the 006 instruction are as follows:

- If both parcels of the instruction are in the same buffer and the branch address is in a buffer, the issue time is 6 CPs.

- If both parcels of the instruction are in the same buffer and the branch address is not in a buffer, the issue time is a minimum of 111 CPs. The location of the branch address within the instruction buffer may add more CPs to this time. Additional time is required if a memory conflict exists.

Description

The 006*ijkm* instruction is a 2-parcel unconditional jump instruction. It sets the P register to the parcel address that is specified by the low-order 24 bits of the *exp* (*ijkm* field). The program continues at that address.

Instruction 0070

Machine Instruction	CAL Syntax	Description
007*ijkm*	R *exp*	Return jump to *exp* and set register B00 to (P) + 2.

Hold Issue Conditions

The instruction holds issue under any of the following conditions:

- A 025 instruction was issued in the previous 2 CPs.
- The second parcel is in a different buffer (a 3-CP delay occurs).
- The second parcel is not in a buffer.
- Instruction 034 is in progress with block length less than or equal to 100_8 and register B*jk* has not been written.
- Instruction 034 in progress with block length greater than 100_8.
- Instruction 035 is in progress.

Execution Time

The issue times for the 007 instruction are as follows:

- If both parcels of the instruction are in the same buffer and the branch address is in a buffer, the instruction issue time is 6 CPs.

- If both parcels of the instruction are in the same buffer and the branch address is not in a buffer, the instruction issue time is a minimum of 111 CPs. The location of the branch address within the instruction buffer may add more CPs to this time. Additional time is needed if a memory conflict exists.

Description

The 2-parcel 007*ijkm* instruction sets register B00 to the address of the parcel that follows the second parcel of the instruction. The P register is then set to the parcel address that is specified by the low-order 24 bits of the *exp* (*ijkm* field). Execution continues at that address.

This instruction provides return links for subroutine calls. The subroutine is entered through a return jump. The subroutine can return to the caller at the instruction following the call by executing a jump to the contents of register B00 (0050*jk*).

Instructions 010 through 013

Machine Instruction	CAL Syntax	Description
010*ijkm*[a]	JAZ *exp*	Jump to *exp* if (A0) = 0 (i_2 = 0).
011*ijkm*[a]	JAN *exp*	Jump to *exp* if (A0) ≠ 0 (i_2 = 0).
012*ijkm*[a]	JAP *exp*	Jump to *exp* if (A0) positive; (A0) ≥ 0 (i_2 = 0).
013*ijkm*[a]	JAM *exp*	Jump to *exp* if (A0) negative (i_2 = 0).

[a] Bit 2 of the *i* field is equal to 0.

Special Cases

The following special cases exist for instructions 010 through 013:

- (A0) = 0 is a positive condition.
- The high-order bit of the *i* designator (i_2) must be 0.
- Register A0 is 32 bits wide and bit 31 is the sign bit.

Hold Issue Conditions

Instructions 010 through 013 hold issue under any of the following conditions:

- Register A0 is busy in any one of the previous 3 CPs.

- The second parcel of the instruction is not in a buffer.
- The second parcel of the instruction is in a different buffer (holds issue for 3 CPs).

Execution Time

The following instruction issue times are for instructions 010 through 013, if the branch is taken (jump conditions are satisfied):

- If both parcels of the instruction are in the same buffer, the branch is taken, and the branch address is in a buffer, the issue time is 6 CPs.
- If both parcels of the instruction are in the same buffer, the branch is taken, and the branch address is not in a buffer, the issue time is a minimum of 111 CPs.
- If each parcel of the instruction is in a different buffer, the branch is taken, and the branch address is in a buffer, the issue time is 9 CPs.
- If each parcel of the instruction is in a different buffer, the branch is taken, and the branch address is not in a buffer, the issue time is a minimum of 114 CPs.
- If the second parcel of the instruction is not in a buffer, the branch is taken, and the branch address is in a buffer, the issue time is a minimum of 114 CPs.
- If the second parcel of the instruction is not in a buffer, the branch is taken, and the branch address is not in a buffer, the issue time is approximately 240 CPs.

The following instruction issue times are for instructions 010 through 013, if the branch is not taken (jump conditions are satisfied):

- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in the same instruction buffer, the issue time is 2 CPs.
- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in a different instruction buffer, the issue time is 5 CPs.

- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in memory, the issue time is a minimum of 111 CPs.
- If each parcel of the instruction is in a different buffer and the branch is not taken, the issue time is 5 CPs.
- If the second parcel of the instruction is not in a buffer and the branch is not taken, the issue time is approximately 111 CPs.

Note: Memory conflicts may produce a delay whenever a fetch operation occurs.

Description

The 2-parcel 010 through 013 instructions test the contents of the A0 register for the condition specified by the *h* field. If the condition is satisfied, the P register is set to the parcel address that is specified by the low-order 24 bits of the *exp* (*ijkm* field) and execution continues at that address. The high-order bit (i_2) of the *ijkm* field must be 0. If the condition is not satisfied, execution continues with the instruction that follows the branch instruction.

Instructions 014 through 017

Machine Instruction	CAL Syntax	Description
014*ijkm*[a]	JSZ *exp*	Jump to *exp* if (S0) = 0 (i_2 = 0)
015*ijkm*[a]	JSN *exp*	Jump to *exp* if (S0) ≠ 0 (i_2 = 0)
016*ijkm*[a]	JSP *exp*	Jump to *exp* if (S0) positive; (A0 ≥ 0) (i_2 = 0)
017*ijkm*[a]	JSM *exp*	Jump to *exp* if (S0) negative (i_2 = 0)

[a] Bit 2 of the *i* field is equal to 0.

Special Cases

The following special cases exist for instructions 014 through 017:

- (S0) = 0 is a positive condition.
- The high-order bit of the *i* designator (i_2) must be 0.

Hold Issue Conditions

Instructions 014 through 017 hold issue under any of the following conditions:

- Register S0 is busy in any one of the previous 3 CPs.
- The second parcel of the instruction is in a different buffer (holds issue for 3 CPs).
- The second parcel of the instruction is not in a buffer.

Execution Time

The following issue times are for instructions 014 through 017, if the branch is taken (jump conditions are satisfied):

- If both parcels of the instruction are in the same buffer, the branch is taken, and the branch address is in a buffer, the issue time is 6 CPs.
- If both parcels of the instruction are in the same buffer, the branch is taken, and the branch address is not in a buffer, the issue time is a minimum of 111 CPs.
- If each parcel of the instruction is in a different buffer, the branch is taken, and the branch address is in a buffer, the issue time is 9 CPs.
- If each parcel of the instruction is in a different buffer, the branch is taken, and the branch address is not in a buffer, the issue time is a minimum of 114 CPs.
- If the second parcel of the instruction is not in a buffer, the branch is taken, and the branch address is in a buffer, the issue time is a minimum of 114 CPs.
- If the second parcel of the instruction is not in a buffer, the branch is taken, and the branch address is not in a buffer, the issue time is approximately 240 CPs.

The following issue times are for instructions 014 through 017 if the branch is not taken (jump conditions are not satisfied):

- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in the same instruction buffer, the issue time is 2 CPs.
- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in a different instruction buffer, the issue time is 5 CPs.

- If both parcels of the instruction are in the same buffer, the branch is not taken, and the next instruction is in memory, the issue time is a minimum of 111 CPs.
- If each parcel of the instruction is in a different buffer and the branch is not taken, the issue time is 5 CPs.
- If the second parcel of the instruction is not in a buffer and the branch is not taken, the issue time is approximately 111 CPs.

Note: Memory conflicts produce delays when a fetch operation occurs.

Description

The 2-parcel 014 through 017 instructions test the contents of the S0 register for the condition specified by the *h* field. If the condition is satisfied, the P register is set to the parcel address that is specified by the low-order 24 bits of the *exp* (*ijkm* field) and execution continues at that address. The high-order bit (i_2) of the *ijkm* field must be 0. If the condition is not satisfied, execution continues with the instruction that follows the branch instruction.

Instructions 020 through 022

Machine Instruction	CAL Syntax	Description
020*i*00*mn*[d] or 021*i*00*mn* or 022*ijk*[d]	A*i* *exp*	Transmit *exp* into A*i* (020 or 022) or transmit one's complement of *exp* into A*i* (021).

[d] These instructions are generated depending on the value of the expression.

Hold Issue Conditions

Instructions 020 through 022 hold issue under any of the following conditions:

- The A*i* register is reserved.
- The second or third instruction parcel is not in a buffer.

Execution Time

The following instruction issue times apply to instructions 020 through 022:

- Register A*i* is ready in 1 CP.

- For instructions 020 and 021, the instruction issue time is 2 CPs.
- For instruction 022, the instruction issue time is 1 CP.
- If parcel 0 is in a different buffer than parcels 1 and 2, the instruction issue time is 5 CPs.
- If parcel 2 is in a different buffer than parcels 0 and 1, the instruction issue time is 6 CPs.

Description

Instructions 020 through 022 transmit a value that is determined by *exp* into the A*i* register. The syntax differs from most CAL symbolic instructions in that the assembler generates any of the previous Cray machine instructions depending on the form, value, and attributes of the *exp*.

The assembler generates the instruction 022*ijk* if all of the following conditions are true (the *jk* fields contain the 6-bit value of *exp*):

- The value of the expression is positive and less than 77 (octal).
- All symbols (if any) within the expression are previously defined.
- The expression has an absolute relative attribute.

If any one of the previous three conditions is not true, the assembler generates one of the following instructions:

- 3-parcel 020*i*00*mn* or 021*i*00*mn* instruction

If the *exp* has a positive value greater than 77 (octal) or either a relocatable or external relative attribute, the following condition occurs:

- Instruction 020*i*00*mn* is generated. The *exp* value is entered in the 32-bit *mn* field.

If the *exp* value is negative and has an absolute relative attribute, the following condition occurs:

- Instruction 021*i*00*mn* is generated. The one’s complement of the *exp* value is entered into the 32-bit *mn* field unless the *exp* value is -1. If the *exp* is -1, instruction 031*i*00 is generated.

Instruction 023

Machine Instruction	CAL Syntax	Description
023*ij*0	A*i* S*j*	Transmit (S*j*) to A*i*.
023*i*01	A*i* VL	Transmit (VL) to A*i*.

Special Cases

The following special cases exist for instruction 023:

- If *j* = 0 then (S*j*) = 0.
- If the low-order 6 bits of the VL register are 0, bit 6 in the VL register = 1.
- If any of the low-order 6 bits of the VL register are not 0, bit 6 = 0.

If (A1) = 0, the following CAL sequence produces (A2) = 100 (octal):

- VL A1
- A2 VL

If (A1) = 23 (octal), the following CAL sequence produces (A2) = 23 (octal):

- VL A1
- A2 VL

If (A1) = 123 (octal), the following CAL sequence produces (A2) = 23 (octal):

- VL A1
- A2 VL

Hold Issue Conditions

The 023 instruction holds issue under any of the following conditions:

- The A*i* register is reserved.
- Instruction 0020*xx* is issued in the previous CP.

The 023*ij*0 instruction holds issue if the S*j* register is reserved (except S0).

Execution Time

The instruction issue times are as follows:

- The instruction issue time is 1 CP.
- The A*i* register is ready in 1 CP.

Description

Instruction 023*ij*0 transmits the low-order 32 bits of the contents of the S*j* register into the A*i* register. The high-order bits of the S*j* register are ignored. Register A*i* = 0 if the *j* designator is 0. Instruction 023*i*01 transmits the contents of the VL register into the A*i* register.

Instructions 024 through 025

Machine Instruction	CAL Syntax	Description
024*ijk*	A*i* B*jk*	Transmit (B*jk*) to A*i*.
025*ijk*	B*jk* A*i*	Transmit (A*i*) to B*jk*.

Hold Issue Conditions

Instructions 024 and 025 hold issue under any of the following conditions:

- Register A*i* is reserved.
- Instruction 025*ijk* was issued in the previous CP (for instruction 024*ijk*).
- Instruction 034 is in progress with block length less than or equal to 100_8 and register B*jk* has not been written.
- Instruction 034 is in progress with block length greater than 100_8.
- Instruction 035 is in progress.

Execution Time

The issue times for instructions 024 and 025 are as follows:

- Register A*i* is ready in 1 CP after issuing a 024 instruction.
- Instruction issue time is 1 CP.

Description

Instruction 024 transmits the contents of the B*jk* register into the A*i* register, and instruction 025 transmits the contents of the A*i* register into the B*jk* register.

Instruction 026

Machine Instruction	CAL Syntax	Description
026*ij*0	A*i* PS*j*	Transmit the population count of (S*j*) to A*i*.
026*ij*1	A*i* QS*j*	Transmit the population count parity of (S*j*) to A*i*.
026*ij*7	A*i* SB*j*	Transmit (SB*j*) to A*i*.

Special Cases

The following special cases exist for instruction 026:

- For instructions 026*ij*0 and 026*ij*1, if *j* = 0 then (A*i*) = 0.
- For instruction 026*ij*7, if CLN = 0 then (A*i*) = 0.

Hold Issue Conditions

Instruction 026 holds issue under any of the following conditions:

- The A*i* register is reserved.
- For instructions 026*ij*0 and 026*ij*1 when the S*j* register is reserved (except S0).
- For instruction 026*ij*7 when a shared path conflict occurs or the shared operation buffer is full.

Execution Time

The instruction issue times for the 026 instruction are as follows:

- The instruction issue time is 1 CP.
- For instructions 026*ij*0 and 026*ij*1, register A*i* is ready in 4 CPs.
- For instruction 026*ij*7, register A*i* is ready in 41 CPs.

Description

Instruction 026*ij*0 counts the number of 1 bits in the S*j* register and enters the result into the low-order 7 bits of the A*i* register. The high-order bits of the A*i* register are cleared. If the S*j* register equals 0, then the value in the A*i* register equals 0.

Instruction 026*ij*1 enters a 0 in the A*i* register if the S*j* register has an even number of 1 bits. If the S*j* register has an odd number of 1 bits, a 1 is entered in the A*i* register. The high-order bits of the A*i* register are cleared. The actual population count is not transferred.

Instructions 026*ij*0 and 026*ij*1 are executed in the population/parity/leading-zero count functional unit.

Instruction 026*ij*7 transmits the contents of the SB*j* register to the A*i* register. The SB*j* register is shared between the CPUs in the same cluster.

Instruction 027

Machine Instruction	CAL Syntax	Description
027*ij*0	A*i* ZS*j*	Transmit leading zero count of (S*j*) to A*i*.
027*ij*7	SB*j* A*i*	Transmit (A*i*) to SB*j*.

Special Cases

The following special cases exist for instruction 027:

- If *j* = 0 for instruction 027*ij*0, register A*i* = 64.
- If S*j* is negative for instruction 027*ij*0, A*i* = 0.
- If CLN = 0 for instruction 027*ij*7, the instruction performs no operation.

Hold Issue Conditions

The 027 instruction holds issue under any of the following conditions:

- The A*i* register is reserved.
- For 027*ij*0 instruction when the S*j* register is reserved (except S0).
- For instruction 027*ij*7 when a shared path access conflict occurs or if the shared operations buffer is full.

Execution Time

The instruction issue times for instruction 027 are as follows:

- The instruction issue time is 1 CP.
- For instruction 027*ij*0, the A*i* register is ready in 4 CPs.
- For instruction 027*ij*7, the SB*j* register is ready in approximately 25 CPs.

Description

Instruction 027*ij*0 counts the number of leading 0's in the S*j* register and enters the result into the low-order 7 bits of the A*i* register. All bits above bit 6 in the A*i* register are cleared. The A*i* register is set to 64 if the *j* designator is 0, or if the content of the S*j* register is 0. Instruction 027*ij*0 executes in the population/parity/leading-zero count functional unit. Instruction 027*ij*7 transmits the contents of the A*i* register to the SB*j* register. The SB*j* register is shared between the CPUs in the same cluster.

Instructions 030 through 031

Machine Instruction	CAL Syntax	Description
030*ijk*	A*i* A*j* + A*k*	Transmit the integer sum of (A*j*) and (A*k*) to A*i*.
030*i*0*k*[b]	A*i* A*k*	Transmit (Ak) to A*i*.
030*ij*0[b]	A*i* A*j* + 1	Transmit the integer sum of (A*j*) and 1 to A*i*.
031*ijk*	A*i* A*j*-A*k*	Transmit the integer difference (A*j*) and (A*k*) to A*i*.
031*i*00[b]	A*i* -1	Transmit -1 to A*i*.
031*i*0*k*[b]	A*i* -A*k*	Transmit the negative of (A*k*) to A*i*.
031*ij*0[b]	A*i* A*j* -1	Transmit the integer difference (A*j*) and 1 to A*i*.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instruction 030:

- If $j = 0$ and $k \neq 0$, then A*i* = A*k*.
- If $j = 0$ and $k = 0$, then A*i* = 1.
- If $j \neq 0$ and $k = 0$, then A*i* = A*j* + 1.

The following special cases exist for instruction 031:

- If $j = 0$ and $k \neq 0$, then A*i* = -A*k*.

- If $j = 0$ and $k = 0$, then Ai = -1.
- If $j \neq 0$ and $k = 0$, then Ai = Aj - 1.

Hold Issue Conditions

Instructions 030 and 031 hold issue under any of the following conditions:

- The Ai register is reserved.
- The Aj or Ak register is reserved (except A0).

Execution Time

The issue times for instructions 030 and 031 are as follows:

- The instruction issue time is 1 CP.
- Register Ai is ready in 2 CPs.

Description

Instructions 030 and 031 execute in the address add functional unit, overflow is not detected by either instruction.

Instruction 030 forms the integer sum of the contents of the Aj and Ak registers and enters the result into the Ai register.

Instruction 031 forms the integer difference of the contents of the Aj and Ak registers and enters the result into the Ai register. Instruction 031i00 is generated in place of instruction 020$ijkm$ if the operand is -1.

Instruction 032

Machine Instruction	CAL Syntax	Description
032*ijk*	A*i* A*j**A*k*	Transmit the integer product of (A*j*) and (A*k*) to A*i*.

Special Cases

The following special cases exist for instruction 032:

- If $j = 0$, (Ai) = 0.
- If $k = 0$, (Ak) = 1.
- If $j \neq 0$ and $k = 0$, (Ai) = (Aj).

Hold Issue Conditions

The 032 instruction holds issue under any of the following conditions:

- The A*i* register is reserved.
- The A*j* or A*k* register is reserved (except A0).

Execution Time

The instruction issue times are as follows:

- The instruction issue time is 1 CP.
- Register A*i* is ready in 4 CPs.

Description

Instruction 032 forms the integer product of the contents of the A*j* and A*k* registers and enters the low-order 32-bit result into the A*i* register. Instruction 032 executes in the address multiply functional unit, and overflow conditions are not detected.

Instruction 033

Machine Instruction	CAL Syntax	Description
033*i*00	A*i* CI	Transmit the channel number of the highest priority interrupt request to A*i* (*j* = 0).
033*ij*0	A*i* CA,A*j*	Transmit the current address of channel (A*j*) to A*i* (*j* ≠ 0, *k* = 0).
033*ij*1	A*i* CE,A*j*	Transmit the error flag of channel (A*j*) to A*i* (*j* ≠ 0, *k* = 1).

Special Cases

The following special cases exist for instruction 033:

- If (A*j*) = 0, then (A*i*) = highest priority channel causing an interrupt.
- If (A*j*) ≠ 0 and *k* = 0, then (A*i*) = current address of channel (A*j*).
- If (A*j*) ≠ 0 and *k* = 1, then (A*i*) = I/O error flag of channel (A*j*).
- After instruction 0012*j*0 issues, 033*i*00 issues immediately because the JS ASIC ensures that all local instructions are held until all global instructions are completed.

All 033*ij*1 instructions return a 1-bit channel error flag, regardless of the type of channel.

Hold Issue Conditions

The 033 instruction holds issue under any of the following conditions:

- The A*i* or A*j* (except A0) register is reserved.
- A shared register conflict occurs or the shared operation buffer is full.

Execution Time

The instruction issue times for instruction 033 are as follows:

- The instruction issue time is 1 CP.
- For 033*i*00, register A*i* is ready in 46 CPs.
- For 033*ij*0, register A*i* is ready in 122 CPs if no conflicts occur with other CPUs.
- For 033*ij*1, register A*i* is ready in 124 CPs if no conflicts occur with other CPUs.

Description

Instruction 033 enters channel status information into the A*i* register. The *j* and *k* designators and the contents of register A*j* define the information. Instruction 033 does not interfere with channel operation and is not protected from user execution.

Instruction 033*i*00 enters the channel number of the highest priority interrupt request into the A*i* register. For each channel, there is a single priority bit that indicates whether it is a high- or low-priority channel. When a processor requests the highest-priority channel, that channel is determined as follows:

1. If any channel marked as high priority has an interrupt pending, the lowest-numbered, high-priority channel is the one returned.
2. If no channels marked as high priority have an interrupt pending, the lowest-numbered, low-priority channel with an interrupt pending is returned.

Instruction 033*ij*0 enters the contents of the CA register for the channel that is specified by the contents of the A*j* register into the A*i* register.

Instruction 033*ij*1 enters the error flag for the channel that is specified by the contents of the A*j* register into the low-order bit of the A*i* register. The high-order bits of the A*i* register are cleared. The error flag can be cleared only in monitor mode by using the 0012 instruction.

Instructions 034 through 037

Machine Instruction	CAL Syntax	Description
034*ijk*	B*jk*, A*i* ,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
034*ijk*[b]	B*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
035*ijk*	,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
035*ijk*[b]	0,A0 B*jk*,A*i*	Store (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
036*ijk*	T*jk*,A*i* ,A0	Load (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
036*ijk*[b]	T*jk*,A*i* 0,A0	Load (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
037*ijk*	,A0 T*jk*,A*i*	Store (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).
037*ijk*[b]	0,A0 T*jk*,A*i*	Store (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 034 through 037:

- If (A*i*) register = 0, initiate a zero-block transfer.
- If (A*i*) register is in a range greater than 100 (octal) and less than 200 (octal), a wrap-around condition occurs.
- If (A*i*) register is greater than 177 (octal), bits 7 through 23 are truncated and the block length is equal to the value of 0 through 6.

Hold Issue Conditions

The 034 through 037 instructions hold issue under any of the following conditions:

- The A0 register is reserved.
- The A*i* register is reserved.
- Instruction 034 holds issue if port A is busy, when instruction 035 is in progress, or a block write (035, 037, 177) is busy with memory access in unidirectional mode or with an uncompleted 076 instruction active.
- Instruction 035 holds issue when a block write (035, 037, 177) is busy, or when instruction 034 is in progress, or in unidirectional memory mode and port A or port B is busy. An uncompleted 076 instruction is equivalent to unidirectional memory mode.
- Instruction 036 holds issue if port B is busy, when instruction 037 is in progress, or when a block write (035, 037, 177) is busy with memory access in unidirectional mode or with an uncompleted 076 instruction active.
- Instruction 037 holds issue when a block write (035, 037, 177) is busy, when instruction 036 is in progress, or in unidirectional memory mode and port A or port B is busy. An uncompleted 076 instruction is equivalent to unidirectional memory mode.

Execution Time

The instruction issue times are as follows:

- The instruction issue time is 1 CP.
- For instruction 034 or 036,
  - If (A*i*) ≠ 0, B or T registers are reserved for a minimum of (A*i*/2) + 163 CPs if the data loads from memory or a minimum of (A*i*/2) + 17 CPs if the data loads from cache.
  - If (A*i*) = 0, B or T registers are reserved for 6 CPs.
  - If (A*i*) ≠ 0, port A or B is busy for (A*i*/2) + 5 CPs.

- If (A*i*) = 0, port A or B is busy for 6 CPs.

Note: Instructions 034 and 036 with a block length of less than or equal to 100_8 release the B or T registers individually as they are written.

- For instruction 035 or 037,
  - If (A*i*) ≠ 0, B or T registers are reserved for (A*i*/2) + 5 CPs.
  - If (A*i*) = 0, B or T registers are reserved for 6 CPs.
  - If (A*i*) ≠ 0, port A or port B is busy for (A*i*/2) + 5 CPs.
  - If (A*i*) = 0, port A or port B is busy for 6 CPs.

Description

Instructions 034 through 037 perform block transfers between cache or central memory and B or T registers. Instruction 034*ijk* transfers words from cache or central memory directly into the B registers. Instruction 035*ijk* stores words from B registers directly into cache and central memory.

Instruction 036*ijk* transfers words from cache or central memory directly into T registers. Instruction 037*ijk* stores words from T registers directly into cache or central memory.

For the 034 through 037 instructions, processing of B and T registers is circular. The first register involved in the transfer is specified by the *jk* fields; the low-order 7 bits of the contents of the A*i* register specify the number of words transmitted. Successive transfers involve successive B or T register pairs until B76/B77 or T76/T77 is reached. Register pair B00/B01 is processed after B76/B77 and register T00/T01 is processed after T76/T77 if the count in the content of the A*i* register is not exhausted.

The first memory location that is referenced by the transfer instruction is specified by the contents of register A0. The contents of register A0 are not altered by execution of the instruction. Memory references are incremented by 2 for successive transfers.

For transfers of B registers to cache and central memory, each 32-bit value is right adjusted in the word; the high-order 32 bits are cleared. When transferring from memory into B registers, only the 32 low-order bits are transmitted; the 32 high-order bits are ignored.

If the contents of the A*i* register equal 0, no words are transferred. If *i* = 0, the contents of register A0 are used for the block length and the starting memory address. The CAL assembler issues a warning message when *i* = 0.

Note: Instruction 034 uses port A, instructions 035 and 037 use either ports A or B, and instruction 036 uses port B for block transfers.

Instruction 040 through 041

Machine Instruction	CAL Syntax	Description
040*i*00*mn* or 041*i*00*mn*	S*i exp*	Transmit *exp* into S*i* (040) or transmit one's complement of *exp* into S*i* (041).

Hold Issue Conditions

Instructions 040 through 041 hold issue under any of the following conditions:

- S*i* register is reserved.
- The second or third parcel is not in a buffer.

Execution Time

The instruction issue times for instructions 040 and 041 are as follows:

- If both parcels are in the same buffer, the issue time is 2 CPs.
- If parcel 0 is in a different buffer than parcels 1 and 2, the issue time is 5 CPs.
- If parcels 0 and 1 are in a different buffer than parcel 2, the issue time is 6 CPs.
- The S*i* register is ready in 1 CP.

Description

These instructions transmit a quantity into the S*i* register. Depending on the instruction *exp* value, either the 040*i*00*mn* or the 041*i*00*mn* instruction is generated. If the expression has a positive value, or either a relocatable or external relative attribute, the following instruction is generated.

- Instruction 040*i*00*mn* is generated with the 32-bit *mn* field containing the expression value.

If the expression has a negative value and an absolute relative attribute, the following instruction is generated:

- Instruction 041*i*00*mn* is generated with the 32-bit *mn* field containing the one's complement of the expression value.

Instructions 042 through 043

Machine Instruction	CAL Syntax	Description
042*ijk*	S*i* <*exp*	Form ones mask in S*i* *exp* bits from right; *exp* = 100_8 - *jk* bits.
042*ijk*[b]	S*i* # >*exp*	Form zeroes mask in S*i* *exp* bits from left; *exp* = *jk* bits.
042*i*77[b]	S*i* 1	Enter 1 into S*i* register.
042*i*00[b]	S*i* -1	Enter -1 into S*i* register.
043*ijk*	S*i* >*exp*	Form ones mask in S*i* *exp* bits from left; *exp* = *jk* bits.
043*ijk*[b]	S*i* #<*exp*	Form zeroes mask in S*i* *exp* bits from right; *exp* = 100_8 - *jk* bits.
043*i*00[b]	S*i* 0	Clear the S*i* register.

[b] Special CAL syntax.

Hold Issue Conditions

Instructions 042 through 043 hold issue when the S*i* register is reserved.

Execution Time

The issue times for instructions 042 and 043 are as follows:

- The instruction issue time is 1 CP.
- Register S*i* is ready in 1 CP.

Description

Instruction 042 generates a mask of 100 (octal) - *jk* 1's from right to left in the S*i* register. For example, if *jk* = 0, the S*i* register contains all 1 bits (integer value = -1) and if *jk* = 77 (octal), the S*i* register contains 0's in all but the low-order bit (integer value = 1).

Instruction 043 generates a mask of *jk* 1's from left to right in the S*i* register. For example, if *jk* = 0, the S*i* register contains all 0 bits (integer value = 0) and if *jk* = 77 (octal), the S*i* register contains 1's in all bits except the low-order bit (integer value = -2).

The scalar logical functional unit executes instructions 042 and 043.

Instructions 044 through 051

Machine Instruction	CAL Syntax	Description
044*ijk*	S*i* S*j*&S*k*	Transmit the logical product of (S*j*) and (S*k*) to S*i*.
044*ij*0[b]	S*i* S*j*&SB	Transmit the sign bit of (S*j*) to S*i*.
044*ij*0[b]	S*i* SB&S*j*	Transmit the sign bit of (S*j*) to S*i* (*j* ≠ 0)
045*ijk*	S*i* #S*k*&S*j*	Transmit the logical product of (S*j*) and complement of (S*k*) to S*i*.
045*ij*0[b]	S*i* #SB&S*j*	Transmit the (S*j*) with sign bit cleared to S*i*.
046*ijk*	S*i* S*j*\S*k*	Transmit the logical difference of (S*j*) and (S*k*) to S*i*.
046*ij*0[b]	S*i* S*j*\SB	Toggle the sign bit of (S*j*), then enter into S*i*.
046*ij*0[b]	S*i* SB\S*j*	Toggle the sign bit of (S*j*), then enter into S*i* (*j* ≠ 0)
047*ijk*	S*i* #S*j*\S*k*	Transmit the logical equivalence of (S*k*) and (S*j*) to S*i*.
047*i*0*k*[b]	S*i* #S*k*	Transmit the one's complement of (S*k*) to S*i*.
047*ij*0[b]	S*i* #S*j*\SB	Transmit the logical equivalence of (S*j*) and sign bit to S*i*.
047*ij*0[2]	S*i* #SB\S*j*	Transmit the logical equivalence of (S*j*) and sign bit to S*i* (*j* ≠ 0).
047*i*00[b]	S*i* #SB	Transmit the one's complement of sign bit into S*i*.
050*ijk*	S*i* S*j* !S*i*&S*k*	Transmit the logical product of (S*i*) and (S*k*) complement ORed with the logical product of (S*j*) and (S*k*) to S*i*.
050*ij*0[b]	S*i* S*j* !S*i*&SB	Transmit the scalar merge of (S*i*) and sign bit of (S*j*) to S*i*.
051*ijk*	S*i* S*j*!S*k*	Transmit the logical sum of (S*j*) and (S*k*) to S*i*.
051*i*0*k*[b]	S*i* S*k*	Transmit the (S*k*) to S*i*.
051*ij*0[b]	S*i* S*j*!SB	Transmit the logical sum of (S*j*) and sign bit to S*i*.
051*ij*0[b]	S*i* SB!S*j*	Transmit the logical sum of (S*j*) and sign bit to S*i* (*j* ≠ 0).
051*i*00[b]	S*i* SB	Transmit the sign bit into S*i*.

[b] Special CAL syntax.

Note: For instructions 044 through 051, the abbreviation SB in the CAL syntax refers to the sign bit, not a shared address register.

Special Cases

The following special cases exist for instructions 044 through 051:

- If *j* = 0, (S*j*) = 0.
- If *k* = 0, (S*k*) = Bit 63 set to 1.

Hold Issue Conditions

Instructions 044 through 051 hold issue under the following conditions:

- The S*i* register is reserved.
- The S*j* or S*k* register is reserved (except S0).

Execution Time

The issue times for instructions 044 through 051 are as follows:

- The instruction issue time is 1 CP.
- Register S*i* is ready in 1 CP.

Description

The scalar logical functional unit executes instructions 044 through 051. Instruction 044 forms the logical product (AND) of the contents of the S*j* register and the contents of the S*k* register and enters the result into the S*i* register. Bits of the S*i* register are set to 1 when corresponding bits of the S*j* register and the S*k* register are 1, as in the following example:

```
(Sj)  = 1 1 0 0
(Sk)  = 1 0 1 0
-----------------
(Si)  = 1 0 0 0
```

The contents of the S*j* register are transmitted to the S*i* register if the *j* and *k* designators have the same nonzero value. The S*i* register is cleared if the *j* designator is 0. The sign bit of the contents of the S*j* register is transmitted to the S*i* register if the *j* designator is nonzero and the *k* designator is 0. The two special forms of instruction 044*ij*0 perform the same function; however, in the second form, *j* must not equal 0. If *j* equals 0, an assembly error results.

Instruction 045 forms the logical product (AND) of the contents of the S*j* register and the complement of the S*k* register and enters the result into the S*i* register. Bits of the S*i* register are set to 1 when corresponding bits of the S*j* register and the complement of the S*k* register are 1, as in the following example in which the contents of S*k*'= the complement of the contents of S*k*:

if (S*k*)	= 1 0 1 0
(S*j*)	= 1 1 0 0
(S*k*')	= 0 1 0 1
(S*i*)	= 0 1 0 0

S*i* is cleared if the *j* and *k* designators have the same value or if the *j* designator is 0. The content of the S*j* register with the sign bit cleared is transmitted to the S*i* register if the *j* designator is nonzero and the *k* designator is 0. Instruction 045*ij*0 performs the identical function.

Instruction 046 forms the logical difference (exclusive OR) of the contents of the S*j* register and the contents of the S*k* register and enters the result into the S*i* register.

Bits of the S*i* register are set to 1 when corresponding bits of the S*j* register and the S*k* register are different, as in the following example:

(S*j*)	= 1 1 0 0
(S*k*)	= 1 0 1 0
(S*i*)	= 0 1 1 0

S*i* is cleared if the *j* and *k* designators have the same nonzero value. The contents of the S*k* register are transmitted to the S*i* register if the *j* designator is 0 and the *k* designator is nonzero. The sign bit of the contents of the S*j* register is complemented and the result is transmitted to the S*i* register if the *j* designator is nonzero and the *k* designator is 0. The two special forms of instruction 046*ij*0 perform the same function; however, in the second form, *j* must not equal 0. If *j* equals 0, an assembly error results.

Instruction 047 forms the logical equivalent (exclusive NOR) of the contents of the S*j* register and the contents of the S*k* register, and enters the result into the S*i* register.

Bits of the S*i* register are set to 1 when corresponding bits of the S*j* register and the S*k* register are the same as in the following example:

(S*j*) = 1 1 0 0
(S*k*) = 1 0 1 0
(S*i*) = 1 0 0 1

S*i* is set to all 1's if the *j* and *k* designators have the same nonzero value. The complement of the contents of the S*k* register is transmitted to the S*i* register if the *j* designator is 0 and the *k* designator is nonzero. All bits except the sign bit of the contents of the S*j* register are complemented, and the result is transmitted to the S*i* register if the *j* designator is nonzero and the *k* designator is 0. The result is the complement produced by instruction 046. The two special forms of instruction 047*ij*0 perform the same function; however, in the second form, *j* must not equal 0. If *j* equals 0, an assembly error results.

Instruction 047*i*0*k* forms the one's complement of the contents of S*k* and enters the value into S*i*.

Instruction 050 merges the contents of the S*j* register with the contents of the S*i* register, depending on the ones mask in S*k*. The result is defined by the following Boolean equation in which S*k*' is the complement of S*k*, as shown in the following example:

(S*i*) = (S*j*)(S*k*) + (S*i*) (S*k*')

if (S*k*) = 1 1 1 1 0 0 0 0

(S*k*') = 0 0 0 0 1 1 1 1
(S*i*) = 1 1 0 0 1 1 0 0
(S*j*) = 1 0 1 0 1 0 1 0
(S*i*) = 1 0 1 0 1 1 0 0

Instruction 050 is used for merging portions of 64-bit words into a composite word. Bits of the S*i* register are cleared when the corresponding bits of the S*k* register are 1 if the *j* designator is 0 and the *k* designator is nonzero. The sign bit of the contents of the S*j* register replaces the sign bit of the S*i* register if the *j* designator is nonzero and the *k* designator is 0. The sign bit of the S*i* register is cleared if the *j* and *k* designators are both 0.

Instruction 051 forms the logical sum (inclusive OR) of the contents of the S*j* register and the contents of the S*k* register. Bits of the S*i* register are set when one of the corresponding bits of the S*j* register and the S*k* register are set, as in the following example:

```
(Sj)    = 1 1 0 0
(Sk)    = 1 0 1 0
-------------------
(Si)    = 1 1 1 0
```

The contents of the S*j* register are transmitted to the S*i* register if the *j* and *k* designators have the same nonzero value. The contents of the S*k* register are transmitted to the S*i* register if the *j* designator is 0 and the *k* designator is nonzero. The contents of the S*j* register with the sign bit set to 1 are transmitted to the S*i* register if the *j* designator is nonzero and the *k* designator is 0. A ones mask that consists of only the sign bit is entered into the S*i* register if the *j* and *k* designators are both 0.

Instructions 052 through 055

Machine Instruction	CAL Syntax	Description
052*ijk*	S0 S*i* <*exp*	Shift (S*i*) left *exp* places to S0; *exp* = *jk.*
053*ijk*	S0 S*i* >*exp*	Shift (S*i*) right *exp* places to S0; *exp* = 100_8-*jk*.
054*ijk*	S*i* S*i* <*exp*	Shift (S*i*) left *exp* places to S*i*; *exp* = *jk*.
055*ijk*	S*i* S*i* >exp	Shift (S*i*) right *exp* places to S*i*; *exp* = 100_8-*jk*.

Hold Issue Conditions

Instructions 052 through 055 hold issue under any of the following conditions:

- The S*i* register is reserved.
- For instructions 052 and 053, when the S0 register is reserved.

Execution Time

The issue times for instructions 052 through 055 are as follows:

- The instructions issue time is 1 CP.
- For instructions 052 and 053, register S0 is ready in 3 CPs.
- For instructions 054 and 055, register S*i* is ready in 3 CPs.

Description

The scalar shift functional unit executes instructions 052 through 055. The instructions shift values in an S register by an amount specified by *exp* (*jk* field); all shifts are end-off with zero fill.

Instruction 052 shifts the contents of the S*i* register *jk* places to the left and enters the result into the S0 register; the shift range is 0 through 63 left. If the shift count is 64, instruction 053000 is generated and register S0 is cleared.

Instruction 053 shifts the contents of the S*i* register to the right by 100 (octal) - *jk* places and enters the result into the S0 register; the shift range is 1 through 100 (octal) right. If the shift count is 0, then instruction 052000 is generated and the contents of register S0 are not altered.

Instruction 054 shifts the contents of the S*i* register to the left *jk* places and enters the result into the S*i* register; the shift range is 0 through 77 (octal) left. If the shift count is 100 (octal), instruction 055*i*00 is generated and the S*i* register is cleared.

Instruction 055 shifts (S*i*) to the right by 100 (octal) - *jk* places and enters the result into the S*i* register; the shift range is 1 through 100 (octal) right. If the shift count is 0, then instruction 054*i*00 is generated and the contents of the S*i* register are not altered.

Instructions 056 through 057

Machine Instruction	CAL Syntax	Description
056*ijk*	S*i* S*i*,S*j* <A*k*	Shift (S*i*) and (S*j*) left by (A*k*) places to S*i*.
056*ij*0[b]	S*i* S*i*,S*j* <1	Shift (S*i*) and (S*j*) left one place to S*i*.
056*i*0*k*[b]	S*i* S*i* <A*k*	Shift (S*i*) left (A*k*) places to S*i*.
057*ijk*	S*i* S*j*,S*i* >A*k*	Shift (S*j*) and (S*i*) right by (A*k*) places to S*i*.
057*ij*0[b]	S*i* S*j*,S*i* >1	Shift (S*j*) and (S*i*) right one place to S*i*.
057*i*0*k*[b]	S*i* S*i* >A*k*	Shift (S*i*) right (A*k*) places to S*i*.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 056 through 057:

- If $j = 0$, (Sj) = 0.
- If $k = 0$, (Ak) = 1.
- Perform a circular shift if $i = j \neq 0$ and Ak is greater than or equal to 0, and less than or equal to 64.

Hold Issue Conditions

Instructions 056 through 057 hold issue under any of the following conditions:

- The S*i* register is reserved.
- The S*j* or A*k* register is reserved (except S0 and/or A0).

Execution Time

The instruction issue times are as follows:

- The instruction issue time is 1 CP.
- Register S*i* is ready in 3 CPs.

Description

The scalar shift functional unit executes instructions 056 and 057. The instruction shifts 128-bit values that are formed by logically joining two S registers. Shift counts are obtained from the A*k* register. All shift counts are considered positive and all 32 bits of the contents of the A*k* register are used for the shift count.

Replacing the A*k* register reference with 1 is the same as setting the *k* designator to 0; a reference to register A0 provides a shift count of 1. Omitting the S*j* register reference is the same as setting the *j* designator to 0; the contents of the S*i* register are concatenated with a word of 0's.

The shifts are circular if the shift count does not exceed 64, and the *i* and *j* designators are equal and nonzero. For instructions 056 and 057, the contents of the S*j* register are unchanged, provided $i \neq j$. For shifts greater than 64, the shift is end-off with zero fill. Instruction 056 produces a 128-bit quantity by concatenating the contents of the S*i* register and the contents of the S*j* register. This instruction shifts the resulting value to the left by an amount specified by the low-order bits of the A*k* register and enters the high-order bits of the result into the S*i* register. The S*i* register is cleared if the shift count exceeds 127. Instruction 056 produces the same result as instruction 054 if the shift count does not exceed 63 and the *j* designator is 0. The special forms of 056 perform the same function.

Instruction 057 produces a 128-bit quantity by concatenating the contents of the S*j* register and the contents of the S*i* register. This instruction shifts the resulting value to the right by an amount specified by the low-order 7 bits of the contents of the A*k* register and enters the low-order bits of the result into the S*i* register. The S*i* register is cleared if the shift count exceeds 127.

Instruction 057 produces the same result as instruction 055 if the shift count does not exceed 63 and the *j* designator is 0. The special forms of 057 perform the same function.

Instructions 060 through 061

Machine Instruction	CAL Syntax	Description
060*ijk*	S*i* S*j*+S*k*	Transmit the integer sum of (S*j*) and (S*k*) to S*i*.
061*ijk*	S*i* S*j*-S*k*	Transmit the integer difference of (S*j*) and (S*k*) to S*i*.
061*i*0*k*[b]	S*i* -S*k*	Transmit the negative of (S*k*) to S*i*.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instruction 060 or 061:

- If $j = 0$ and $k = 0$, then (S*i*) = bit 63.
- For instruction 060, if $j = 0$ and $k \neq 0$, then (S*i*) = (S*k*).
- For instruction 060, if $j \neq 0$ and $k = 0$, then (S*i*) = (S*j*) with bit 63 complemented.
- For instruction 061, if $j = 0$ and $k \neq 0$, then (S*i*) = -(S*k*).
- For instruction 061, if $j \neq 0$ and $k = 0$, then (S*i*) = (S*j*) with bit 63 complemented.

Hold Issue Conditions

Instructions 060 through 061 hold issue under any of the following conditions:

- The S*i* register is reserved.
- The S*j* or S*k* register is reserved (except S0).

Execution Time

The instruction issue times are as follows:

- Register S*i* is ready in 2 CPs.
- The instruction issue time is 1 CP.

Description

The scalar add functional unit executes instructions 060 and 061. Instruction 060*ijk* forms the integer sum of the contents of the S*j* register and the contents of the S*k* register, and enters the result into the S*i* register; no overflow conditions are detected. The contents of the S*k* register are transmitted to the S*i* register if the *j* designator is 0 and the *k* designator is nonzero. The sign bit is entered in the S*i* register and all other bits of the S*i* register are cleared if the *j* and *k* designators are both 0.

Instruction 061*ijk* forms the integer difference of the contents of the S*j* register and the contents of the S*k* register, and enters the result into the S*i* register; no overflow is detected. The high-order bit of the S*i* register is set and all other bits of the S*i* register are cleared when the *j* and *k* designators are both 0.

Instruction 061*i*0*k* transmits the negative (two's complement) of the contents of the S*k* register into the S*i* register if the *j* designator is 0 and the *k* designator is nonzero. The sign bit is entered in the S*i* register and all other bits of the S*i* register are cleared if the *j* and *k* designators are both 0.

Instructions 062 through 063

Machine Instruction	CAL Syntax	Description
062*ijk*	S*i* S*j*+FS*k*	Transmit the floating-point sum of (S*j*) and (S*k*) to S*i*.
062*i*0*k*[b]	S*i* +FS*k*	Transmit the normalized (S*k*) to S*i*.
063*ijk*	S*i* S*j*-FS*k*	Transmit the floating-point difference of (S*j*) and (S*k*) to S*i*.
063*i*0*k*[b]	S*i* -FS*k*	Transmit the normalized negative of (S*k*) to S*i*.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instruction 062:

- If (S*k*) exponent is valid, $j = 0$ and $k \neq 0$, then (S*i*) = (S*k*) normalized.
- If (S*j*) exponent is valid, $j \neq 0$ and $k = 0$, then (S*i*) = (S*j*) normalized.

The following special cases exist for instruction 063:

- If (S*k*) exponent is valid, $j = 0$ and $k \neq 0$, then (S*i*) = -(Sk) normalized. The sign of (S*i*) is opposite of (S*k*) if (S*k*) $\neq 0$.
- If (S*j*) exponent is valid, $j \neq 0$ and $k = 0$, then (S*i*) = (S*j*) normalized.

Hold Issue Conditions

The 062 through 063 instructions hold issue under any of the following conditions:

- The S*i* register is reserved.
- The S*j* or S*k* register is reserved (except S0).

Execution Time

The instruction issue times are as follows:

- The instruction issue time is 1 CP.
- Register S*i* is ready in 8 CPs.

Description

The floating-point add functional unit executes instructions 062 and 063. The functional unit considers all operands to be in floating-point format; the result is normalized even if the operands are unnormalized. The *k* designator is normally nonzero. In the special forms, the *j* designator is assumed to be 0 so that the normalized contents of S*k* are entered into S*i*. For floating-point operands with the sign bit set (bit = 1), a 0 exponent and 0 coefficient are treated as 0 (all 64 bits = 0, which is considered -0). However, no floating-point unit generates a 0 except the floating-point multiply functional unit if one of the operands was a 0. Normally, -0 occurs in logical manipulations when a sign is attached to a number; that number can be 0.

Instruction 062*ijk* produces the floating-point sum of the contents of the S*j* register and contents of the S*k* register and enters the normalized result into the S*i* register. Instruction 062*i*0*k* transmits the normalized contents of the S*k* register to the S*i* register.

Instruction 063*ijk* produces the floating-point difference of the contents of the S*j* register and contents of the S*k* register and enters the normalized result into the S*i* register. Instruction 063*i*0*k* transmits the negative of the floating-point quantity in the S*k* register to the S*i* register as a normalized floating-point number.

Instructions 064 through 067

Machine Instruction	CAL Syntax	Description
064*ijk*	S*i* S*j**FS*k*	Transmit the floating-point product of (S*j*) and (S*k*) to S*i*.
065*ijk*	S*i* S*j**HS*k*	Transmit the half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*.
066*ijk*	S*i* S*j**RS*k*	Transmit the rounded floating-point product of (S*j*) and (S*k*) to S*i*.
067*ijk*	S*i* S*j**IS*k*	Transmit the reciprocal iteration: 2-(S*j*) to S*i*.

Special Cases

The following special cases exist for instructions 064 through 067:

- If $j = 0$, (Sj) = 0.
- If $k = 0$, (Sk) = bit 63.

If both exponent fields are 0, an integer multiplication operation is performed. Correct integer multiplication results are produced if any of the following conditions occurs:

- Both operand sign bits are 0.
- The number of the 0 bits to the right of the least significant 1 bit in the two operands is greater than or equal to 48.

The integer result obtained is the high-order 48 bits of the 96-bit product of the two operands.

Hold Issue Conditions

Instructions 064 through 067 hold issue under any of the following conditions:

- The S*i* register is reserved.
- The S*j* or S*k* register is reserved (except S0).

Execution Time

The issue times for instructions 064 through 067 are as follows:

- The instruction issue time is 1 CP.
- Register S*i* is ready in 9 CPs.

Description

The floating-point multiply functional unit executes instructions 064 through 067 and considers all operands to be in floating-point format. The result may not be normalized if the operands are not normalized.

Instruction 064*ijk* forms the floating-point product of the contents of the S*j* register and contents of the S*k* register and enters the result into the S*i* register.

Instruction 065*ijk* forms the half-precision rounded floating-point product of the contents of the S*j* and S*k* registers and sends the result to the S*i* register. The low-order 19 bits of the result are cleared. This instruction can be used in the division algorithm when only 30 bits of accuracy are required.

Instruction 066*ijk* forms the rounded floating-point product of the contents of the S*j* and S*k* registers and sends the result to the S*i* register. This instruction is used in the reciprocal approximation sequence.

Instruction 067*ijk* forms two minus the floating-point product of the contents of the S*k* register and contents of the S*j* register and enters the result into the S*i* register.

Instruction 070*ij*0

Machine Instruction	CAL Syntax	Description
070*ij*0	S*j* /HS*j*	Transmit the floating-point reciprocal approximation of (S*j*) to S*i*.

Special Cases

The following special cases exist for instruction 070:

- (S*i*) is invalid if (S*j*) is not normalized. A normalized value is indicated by bit 47 of (S*j*) = 1. No test is made of this bit to determine its value.
- If (S*j*) = 0, a range error occurs and the result is invalid.
- If *j* = 0, (S*j*) = 0.

Hold Issue Conditions

The 070 instruction holds issue under any of the following conditions:

- The S*i* register is reserved.
- The S*j* register is reserved (except S0).

Execution Time

The issue times for the 070 instruction are as follows:

- Register S*i* is ready in 16 CPs.
- The instruction issue time is 1 CP.

Description

The reciprocal approximation functional unit executes instruction 070. Instruction 070 forms an approximation to the reciprocal of the normalized floating-point quantity in the S*j* register and enters the result into the S*i* register. The result is invalid if the contents of the S*j* register are not normalized or are equal to 0.

The reciprocal approximation instruction produces a result of 30 significant bits. The low-order 18 bits are 0's. The number of significant bits is increased to 48 using the reciprocal iteration instruction and a multiplication operation.

Instruction 070*ij*6

Machine Instruction	CAL Syntax	Description
070*ij*6	S*i* S*j**BT	Transmit the bit-matrix product of (S*j*) and transpose of (BMM) to S*i*.

Special Cases

A special case exists when the operand *j* = 0, then (S*j*) = 0.

Hold Issue Conditions

- Register S*i* is reserved.
- Register S*j* is reserved (except S0).
- Instruction 077 was issued in the previous CP.
- The vector instruction queue (VIQ) is full.

VIR Hold Issue Conditions

- The bit-matrix multiply functional unit is busy with a 1740*j*4 or 174*ij*6 instruction.

- The floating-point add functional unit is busy with 170 through 173 instructions.
- Instruction 070*ij*6, 073, 076, or 077 was issued the previous CP.
- A 062 or 063 instruction issued from CIP 3 CPs earlier.
- Any 064 through 067 instruction issued from CIP 4 CPs earlier.
- A 070*ij*0 instruction issued from CIP 14 CPs earlier.

Execution Time

Execution Timing from CIP

- Instruction issue time is 1 CP from CIP.
- Register S*i* is ready in 7 CPs (from CIP when no delays).

Execution Timing from VIR

- Instruction issue time is 2 CP from CIP
- Register S*i* is ready in 5 CPs from VIR

Description

Instruction 070*ij*6 uses the BMM functional unit that it shares with the vector instructions. This functional unit contains the 64×64-bit BMM register, and it generates the bit-matrix product that is returned to the S*i* register. Refer to the "Bit-matrix Multiply Functional Unit" section for more information.

Instruction 070*ij*6 transmits the instruction and the S*i* register contents to the VIQ and VIR to be issued essentially as a vector instruction. The instruction issues from the VIR when the BMM unit is not busy and no conflicts exist on the result operand bus from the vector unit to the S registers. This instruction reserves no V registers.

Instruction 071

Machine Instruction	CAL Syntax	Description
071*i*0*k*	S*i* A*k*	Transmit (A*k*) to S*i* with no sign extension.
071*i*1*k*	S*i* +A*k*	Transmit (A*k*) to S*i* with sign extension.
071*i*2*k*	S*i* +FA*k*	Transmit (A*k*) to S*i* as unnormalized floating-point number.
071*i*30	S*i* 0.6	Transmit 0.75 x 2^{48} as normalized floating-point constant into S*i*.
071*i*40	S*i* 0.4	Transmit 0.5 as normalized floating-point constant into S*i*.
071*i*50	S*i* 1.0	Transmit 1.0 as normalized floating-point constant into S*i*.
071*i*60	S*i* 2.0	Transmit 2.0 as normalized floating-point constant into S*i*.
071*i*70	S*i* 4.0	Transmit 4.0 as normalized floating-point constant into S*i*.

Special Cases

The following special cases exist for instruction 071:

- If *k* = 0, (A*k*) = 1.
- If *j* = 0, (S*i*) = (A*k*).
- If *j* = 1, (S*i*) = (A*k*) sign extended.
- If *j* = 2, (S*i*) = (A*k*) unnormalized.
- If *j* = 3, (S*i*) = 0.6 x 2^{60} (octal).
- If *j* = 4, (S*i*) = 0.4 x 2^{0} (octal).
- If *j* = 5, (S*i*) = 0.4 x 2^{1} (octal).
- If *j* = 6, (S*i*) = 0.4 x 2^{2} (octal).
- If *j* = 7, (S*i*) = 0.4 x 2^{3} (octal).

Hold Issue Conditions

The 071 instructions hold issue under any of the following conditions:

- The S*i* register is reserved.
- The A*k* register is reserved (except A0). This hold issue condition applies when the *j* designators equal 0 through 7.

Execution Time

The issue times for the 071 instruction are as follows:

- Instruction issue time is 1 CP.
- Register S*i* is ready in 2 CPs.

Description

Instruction 071 performs functions that depend on the value of the *j* designator. These functions transmit information from an A register to an S register and generate frequently used floating-point constants.

Instruction 071*i*0*k* transmits the 32-bit value in the A*k* register to the low-order bits of the S*i* register; the high-order bits of the S*i* register are zeroed. The value is treated as an unsigned integer. A value of 1 is entered into the S*i* register when the *k* designator is 0.

Instruction 071*i*1*k* transmits the 32-bit value in the A*k* register to the low-order bits of the S*i* register. The value is treated as a signed integer. The sign bit of the A*k* register is extended through the high-order bits of the S*i* register. A value of 1 is entered into the S*i* register when the *k* designator is 0.

Instruction 071*i*2*k* transmits the 32-bit value in A*k* to S*i* as an unnormalized floating-point quantity. For this instruction, the exponent in bits 62 through 48 is set to 40060 (octal). The sign of the coefficient is set according to the sign of the contents in the A*k* register. If the sign bit is set, the two's complement of the contents of the A*k* register is entered into the S*i* register as the magnitude of the coefficient, and bit 63 of the S*i* register is set for the sign of the coefficient.

A sequence of instructions converts an integer whose absolute value is less than 32 bits to floating-point format. The following CAL code is an example of this instruction sequence:

```
CAL code:   A1 S1
            S1 +FA1
            S1 +FS1 14 CPs required
```

Instructions 071*i*30 through 071*i*70 are initially recognized by the assembler as the symbolic instruction S*i exp*. The assembler then checks the expression for any of the constant values (explained in following paragraphs). If it finds one of the instructions in the exact syntax shown, it generates the corresponding Cray machine instruction. If none of the indicated constant values are found, instruction 040*ijkm* or 041*ijkm* is generated. These constant values enable more efficient instructions when entering commonly used values into S*i*.

Instruction 071*i*30 transmits the floating-point constant of 0.75×2^{48} into S*i* (040060600000000000000 (octal)). This constant is used to create floating-point numbers from integer numbers (positive and negative) whose

absolute value is less than 47 bits. A sequence of instructions is used for conversion of an integer in S1. The following CAL code is an example of this instruction sequence.

```
CAL code:   S2 0.6
            S1 S2-S1
            S1 S2-FS1 13 CPs required
```

Instruction 071*i*40 transmits a floating-point constant 0.4 (040000400000000000000 (octal)) into the S*i* register.

Instruction 071*i*50 transfers the floating-point constant 1.0 (040001400000000000000 (octal)) into the S*i* register.

Instruction 071*i*60 transfers the floating-point constant 2.0 (040002400000000000000 (octal)) into the S*i* register.

Instruction 071*i*70 transfers the floating-point constant 4.0 (040003400000000000000 (octal)) into the S*i* register.

Instructions 072 through 073

Machine Instruction	CAL Syntax	Description
072*i*00	S*i* RT	Transmit (RTC) to S*i*.
072*i*02	S*i* SM	Transmit (SM) to S*i*.
072*ij*3	S*i* ST*j*	Transmit (ST*j*) to S*i*.
073*i*00	S*i* VM	Transmit (VM) to S*i*.
073*i*11[a, c]		Read the performance counter into S*i*.
073*i*21[a, c]		Increment upper performance counter.
073*i*31[a, c]		Clear all maintenance modes.
073*i*61[a, c]		Increment current performance counter (lower).
073*i*01	S*i* SR0	Transmit (SR0) to S*i*.
073*i*02	SM S*i*	Transmit (S*i*) to SM.
073*ij*3	ST*j* S*i*	Transmit (S*i*) to ST*j*.

[a] These instructions are privileged to monitor mode.

[c] These instructions are not supported by CAL Version 2.

Special Cases

The following special cases exist for instructions 072 through 073:

- Instructions 072*i*02 and 072*ij*3, (S*i*) = 0 if CLN = 0.
- Instructions 073*i*02 and 073*ij*3 perform no operation if CLN = 0.
- Instruction 072*i*00 transmits the real-time clock (RTC) to S*i*. The RTC will not be ready for some indeterminate number of cycles; the following code ensures that the RTC is ready:

```
RT      Sj (0014j0)
SBj     A0 (027ij7)
JAZ     label (010ijkm)
label Si  RT (072i00)
```

- The 0014*j*0 is a global instruction, and the 027*ij*7 is a local instruction. All local instructions are held in the JS ASIC until all global instructions are completed.

Hold Issue Conditions

The 072 through 073 instructions hold issue under any of the following conditions:

- The S*i* register is reserved.
- Instructions 072*i*02, 072*ij*3, 073*ij*3, and 073*i*02 hold issue when a shared register access conflict occurs or when the shared operation buffer is full or when the PV to JS interface is still busy due to a previous instruction.
- For instruction 073*i*00, hold issue if vector instruction queue is full.

VIR Hold Issue Conditions

The 073*i*00 instruction holds issue at the VIR under any of the following conditions:

- For instruction 073*i*00, when instruction 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.
- For instruction 073*i*00, when instruction 175 is in progress, the VM is busy for (VL/2) + 2 CPs.
- When instruction 003 is in progress, VM is busy for 3 CPs.

- The 073*i*00 instruction issue is delayed 1 CP because of the following conditions:
  - 076 VIR issue 2 CPs earlier
  - 076*ij*6 VIR issue 2 CPs earlier
  - 062 through 063 CIP issue 5 CPs earlier
  - 064 through 067 CIP issue 6 CPs earlier
  - 070*ij*0 CIP issue 16 CPs earlier

Execution Time

The issue times for instructions 072 through 073 are as follows:

- CIP instruction issue time is 1 CP.
- VIR 073*i*00 instruction issue time is 2 CPs.
- Instruction 073*i*02 and 073*ij*3 cause the PV to JS interface to be busy for 10 CPs and 16 CPs respectively.
- For instructions 072*i*00 and 073*i*11, the S*i* register is ready in 1 CP.
- For instruction 072*i*02, the S*i* register is ready in 64 CPs minimum.
- For instruction 072*ij*3, the S*i* register is ready in 48 CPs minimum.
- For instruction 073*i*00, the S*i* register is ready in 5CPs.
- For instruction 073*i*00, the S*i* register is ready in 3 CPs from the VIR issue.

Description

Instruction 072*i*00 transmits the 64-bit value of the RTC into the S*i* register. The RTC increments by 1 each CP and can be set only by the monitor through use of instruction 0014*j*0.

Note: The SV1 real-time clock increments at the system clock rate, not the CPU clock rate. Therefore, multiple 072*i*00 instructions that issue during the same system clock period will return the same value. The local copy of the RTC in the CPU can be made to increment at the CPU clock rate via a configuration bit.

Instruction 072*i*02 transmits the values of all the semaphores into the S*i* register. The 32-bit SM register is left-justified in the S*i* register with SM00 occupying the sign bit. The PV to JS interface is busy for 4 CPs when this instruction issues.

Instruction 072*ij*3 transmits the contents of the ST*j* register into the S*i* register. The PV to JS interface is busy for 4 PCs on issue of this instruction.

Instruction 073*i*00 transmits the 64-bit contents of the VM register into the S*i* register. The VM register is usually read after it is set by instruction 175. This instruction issues from the VIR (VL/2) + 3CPs after the 175 instruction.

Instruction 073*i*00 issues from the VIR without delay during the execution of a 140 through 147 instruction.

Instruction 073*i*11 is used for performance monitoring and is privileged to monitor mode. Each execution of the 073*i*11 instruction advances a pointer and enters 16 bits of a performance counter into bit positions 32 through 47. It also enters 16 bits of the status register into bit positions 48 through 63 of the S*i* register.

Instruction 073*i*21 is used to test the operation of the performance counters by incrementing the value stored in the counter while the CPU is in monitor mode. When instruction 073*i*21 executes, the value of the performance counter increments at bits 22 and 38. There must be an 8-CP delay between a 073*i*21 instruction and other performance monitor instructions. Instruction 073*i*21 also loads S*i* register bits 32 through 63 with status and advances the performance monitor pointer to the next counter.

Instruction 073*i*31 is used for performance monitoring and is privileged to monitor mode. Instruction 073*i*31 clears all maintenance modes that are set by the 0015*j*1 instruction; allow 10 CPs for the maintenance mode to become ineffective. It also clears the performance monitor pointer. Instruction 073*i*31 also reads status to bits 32 through 63 of the S*i* register.

Instruction 073*i*61 advances the current counter at bit position 0. This instruction also reads status to bits 32 through 63 of the S*i* register. For a 073*i*61 instruction, a carry does not propagate beyond bit 15.

Instruction 073*i*01 sets the low-order 32 bits to 1’s and returns the following status bits to the high-order bits of S*i* register. The 073*i*01 instruction is privileged to monitor mode; the processor number and cluster number bit

positions return a value of 0 if the instruction is not executed in monitor mode. The encoded processor number for bit positions 44 through 42 is defined in word 0 of the exchange package.

S*i* Bit Position	Description
63	Clustered, CLN not equal to zero (CL)
57	Bit-matrix loaded (BML)
53	Uncorrectable memory error occurred (UME)
52	Correctable memory error occurred (CME)
51	Floating-point error occurred (FPS)
50	Floating-point interrupt enabled (IFP)
49	Operand range interrupt enabled (IOR)
48	Bidirectional memory enabled (BDM)
44	Processor number bit 4 (PN4)
43	Processor number bit 3 (PN3)
42	Processor number bit 2 (PN2)
41	Processor number bit 1 (PN1)
40	Processor number bit 0 (PN0)
37	Cluster number bit 5 (CLN5)
36	Cluster number bit 4 (CLN4)
35	Cluster number bit 3 (CLN3)
34	Cluster number bit 2 (CLN2)
33	Cluster number bit 1 (CLN1)
32	Cluster number bit 0 (CLN0)

Instruction 073*i*02 sets the semaphore registers from 32 high-order bits of the S*i* register. SM00 receives the sign bit of the contents of the S*i* register.

Instruction 073*ij*3 transmits the contents of the S*i* register into the ST*j* register.

Instructions 074 through 075

Machine Instruction	CAL Syntax	Description
074*ijk*	S*i* T*jk*	Transmit (T*jk*) to S*i*.
075*ijk*	T*jk* S*i*	Transmit (S*i*) to T*jk*.

Hold Issue Conditions

The 074 through 075 instructions hold issue under any of the following conditions:

- The S*i* register is reserved.
- Instruction 075*ijk* issued in previous CP (for instruction 074*ijk*).
- Instruction 036 in progress with block length less than or equal to 100_8 and register T*jk* has not been written.
- Instruction 036 in progress with block length greater than 100_8.
- Instruction 037 in progress.

Execution Time

The issue times for instructions 074 through 075 are as follows:

- Instruction issue time is 1 CP.
- For instruction 074*ijk*, the S*i* register is ready in 1 CP.

Description

Instruction 074 transmits the contents of the T*jk* register into the S*i* register.

Instruction 075 transmits the contents of the S*i* register into the T*jk* register.

Instructions 076 through 077

Machine Instruction	CAL Syntax	Description
076*ijk*	S*i* V*j*,A*k*	Transmit (V*j* element (A*k*)) to S*i*.
077*ijk*	V*i*,A*k* S*j*	Transmit (S*j*) to V*i* element (A*k*).
077*i*0*k*[b]	V*i*,A*k* 0	Clear element (A*k*) of register V*i*.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 076 through 077:

- If *i* = 0 then (S*i*) = 0.
- If *k* = 0 then (A*k*) = 1.

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- The A*k* register is reserved (except A0) or the vector instruction queue is full.
- For instruction 076, register S*i* is reserved.
- For instruction 077, S*j* is reserved (except S0).
- A 077 instruction was issued in the previous CP.
- The vector instruction queue is full.

VIR Hold Issue Conditions

- V*i* and V*j* registers are reserved.
- Instructions 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.
- 062 or 063 instruction issued from CIP 3 CP earlier (076).
- 063 through 064 instruction issued from CIP 4 CP earlier (076).
- 070*ij*0 instruction issued from CIP 14 CP earlier (076).

Execution Time

The instruction issue times are as follows:

- For instruction 076, CIP issue time is 1 CP.
- For instruction 077, CIP issue time is 2 CPs.
- VIR issue time is 2 CPs.
- For the instruction 076, register S*i* is ready in 5 CPs from VIR issue, 7 CPs from CIP issue if no delay occurred in execution.
- For the instruction 077, register V*i* is ready in 4 CPs.

Description

For instruction 077, when followed by any other instruction, there is a 1-CP delay between the two instructions, caused by the PC sending S*j* and A*k* values to the vector unit.

Instructions 076 and 077 transmit a 64-bit quantity between a V register element and an S register.

Instruction 076*ijk* transmits the contents of an element of register V*j* that is indicated by the contents of the low-order 6 bits of A*k* to register S*i*.

Instruction 077*ijk* transmits the contents of register S*j* to an element of register V*i* as determined by the low-order 6 bits of the contents of the A*k* register. Element 1 (the second element of register V*i*) is selected if the *k* designator is 0.

Instruction 077*i*0*k* zeroes element (A*k*) of register V*i*. The low-order 6 bits of A*k* determine which element is cleared. The second element of register V*i* is cleared if the *k* designator is 0.

Instructions 10*h* through 13*h*

Machine Instruction	CAL Syntax	Description
10*hi*00*mn*	A*i exp*,A*h*	Load from address ((A*h*) + *exp*) to A*i*. *(h ≠ 0)*
100*i*00*mn*	A*i exp*,0	Load from address (*exp*) to A*i*.
100*i*00*mn*	A*i exp*,	Load from address (*exp*) to A*i*.
10*hi*0000	A*i* ,A*h*	Load from address (A*h*) to A*i*. *(h ≠ 0)*
11*hi*00*mn*	*exp*,A*h* A*i*	Store (A*i*) to address (A*h*) + *exp*. *(h ≠ 0)*
110*i*00*mn*	*exp*,0 A*i*	Store (A*i*) to address *exp.*
110*i*00*mn*	*exp*, A*i*	Store (A*i*) to address *exp*.
11*hi*0000	,A*h* A*i*	Store (A*i*) to address (A*h*). *(h ≠ 0)*
12*hi*00*mn*	S*i exp*,A*h*	Load from address ((A*h*) + *exp*) to S*i. (h ≠ 0)*
120*i*00*mn*	S*i exp*,0	Load from address (*exp*) to S*i*.
120*i*00*mn*	S*i exp,*	Load from address (*exp*) to S*i*.
12*hi*0000	S*i* ,A*h*	Load from address (A*h*) to S*i*. *(h ≠ 0)*
13*hi*00*mn*	*exp*,A*h* S*i*	Store (S*i*) to address (A*h*) + *exp*. *(h ≠ 0)*
130*i*00*mn*	*exp*,0 S*i*	Store (S*i*) to address *exp*.
130*i*00*mn*	*exp*, S*i*	Store (S*i*) to address *exp*.
13*hi*0000	,A*h* S*i*	Store (S*i*) to address (A*h*). *(h ≠ 0)*

Special Cases

The following special case exists for instructions 10*hi*00*mn*, 11*hi*00*mn*, 12*hi*00*mn*, and 13*hi*00*mn*:

- Only bits 0 through 31 of the A*h* register and the *mn* field are used to calculate the memory address. Refer to the "Calculating Absolute Memory Address" subsection for additional information.

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- Ports A or B busy.
- A*h* is reserved if $h \neq 0$.
- For instructions 10*h* and 11*h*, A*i* is reserved.
- For instructions 12*h* and 13*h*, S*i* is reserved.
- If the second or third parcel is not in a buffer, a 3-CP delay occurs.

Execution Time

The instruction issue times for the 10*h* through 13*h* instructions are as follows:

- If parcel 0 is in one buffer and parcels 1 and 2 are in a different buffer, the issue time is 5 CPs.
- If parcels 0 and 1 are in one buffer and parcel 2 is in a different buffer, the issue time is 6 CPs.
- If all parcels are in the same buffer, the issue time is 2 CPs.
- For instruction 10*h*, register A*i* is ready in 17 CPs from cache and 164 CPs from memory.
- For instruction 12*h*, register S*i* is ready in 17 CPs from cache and 164 CPs from memory.
- A bank is ready for the next scalar load or store operation in 15 CPs.

Description

Instructions 10*h* through 13*h* transmit data between memory and an A register or an S register.

For these instructions, only the value of the expression is used if the *h* designator is 0 or if a 0 or blank field is used in place of A*h*. Only the contents of A*h* are used if the expression is omitted. An assembly error occurs if an expression has a parcel-address attribute.

Instructions10*hi*00*mn* through 10*hi*0000 load the low-order 32 bits of a memory word directly into an A register. The memory address is determined by adding the address in the A*h* register to the expression value (*mn* field). Only the value of the expression is used if the *h* designator is 0, or a 0 or blank field is used in place of A*h*. Only the contents of A*h* are used if the expression is omitted. An assembly error occurs if an expression has a parcel-address attribute.

Instructions 11*hi*00*mn* through 11*hi*0000 store 32 bits from register A*i* directly into memory. The high-order bits of the memory word are cleared. The memory address is determined by adding the address in the A*h* register to the expression value (*mn* field).

Instructions 12*hi*00mn through 12*hi*0000 load the contents of a memory word directly into an S register. The memory address is determined by adding the address in register A*h* to the expression value (*mn* field). Only the value of the expression is used if the *h* designator is 0, otherwise a zero or blank field is used in place of the contents of register A*h*. Only the contents of register A*h* are used if the expression is omitted. An assembly error occurs if an expression has a parcel-address attribute.

Instructions 13*hi*00mn through 13*hi*0000 store the contents of register S*i* directly into memory. The memory address is determined by adding the address in the A*h* register to the expression value (*mn* field).

The *exp* used for addressing orders the parcels as follows: *nm*.

Instructions 140 through 147

Machine Instruction	CAL Syntax	Description
140*ijk*	V*i* S*j*&V*k*	Transmit logical products of (S*j*) and (V*k* elements) to V*i* elements.
141*ijk*	V*i* V*j*&V*k*	Transmit logical products of (V*j* elements) and (V*k* elements) to V*i* elements.
142*ijk*	V*i* S*j*!V*k*	Transmit logical sums of (S*j*) and (V*k* elements) to V*i* elements.
142*i*0*k*[b]	V*i* V*k*	Transmit (V*k* elements) to V*i* elements.
143*ijk*	V*i* V*j*!V*k*	Transmit logical sums of (V*j* elements) and (V*k* elements) to V*i* elements.
144*ijk*	V*i* S*j* \V*k*	Transmit logical differences of (S*j*) and (V*k* elements) to V*i* elements.
145*ijk*	V*i* V*j* \V*k*	Transmit logical differences of (V*j* elements) and (V*k* elements) to V*i* elements.
145*iii*[b]	V*i* 0	Clear V*i* elements.
146*ijk*	V*i* S*j*!V*k*&VM	Transmit (S*j*) if VM bit = 1; (V*k* elements) if VM bit = 0 to V*i* elements.
146*i*0*k*[b]	V*i* #VM&V*k*	Transmit vector merge of (V*k* elements) and 0 to V*i* elements.
147*ijk*	V*i* V*j*!V*k*&VM	Transmit (V*j* elements) if VM bit = 1; (V*k* elements) if VM bit = 0 to V*i* elements.

[b] Special CAL syntax.

Special Cases

The following special case exists for instructions 140 through 147:

- If *j* = 0, then (S*j*) = 0.

Hold Issue Conditions

Instructions 140 through 147 hold issue under any of the following conditions:

- For instructions 140, 142, 144, and 146, if S*j* register is reserved (except S0).
- For instructions 140, 142, 144, and 146 if a 077 instruction was issued in the previous CP, or if the vector instruction queue (VIQ) is full.

VIR Hold Issue Conditions

Instructions 140 and 147 hold issue at the VIR under any of the following conditions:

- V*i* and V*k* (V*j* for 141, 143, 145, and 147) registers are reserved unless chaining or tailgating is permitted.
- Available functional units are busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from VIR the previous CP.

Execution Time

The execution time for vector instructions that are issued directly from CIP to the functional unit through the vector issue register (VIR) is 2 CPs longer than the execution time of the instruction that is waiting to issue in the VIR. The issue times for instructions 140 through 147 from the VIR are as follows:

- For Functional Unit Busy
  - The functional unit is ready in (VL/2) + 1 CP (except for a 140 through 147 instruction following a 175 instruction which is ready in (VL/2) + 4 CPs.
- For Vector Register Busy
  - V*i* is ready for V*i* use in (VL/2) + 5 CPs.
  - V*i* is ready for V*j* or V*k* use immediately (because of chaining).
  - V*j* or V*k* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
  - V*j* or V*k* is ready for V*i* immediately (because of tailgating).
- Vector logical (140 through 147) execution time is (VL/2) + 5 CPs until all of the data is available for use by the next instruction.
- Unit busy time between the floating-point multiply and second vector logical functional units is (VL/2) + 1 CP.
- Unit busy time between the second vector logical and floating-point multiply functional units is (VL/2) + 1 CP.

- Instruction 073*i*00 borrows 1 CP from an active main logical unit during execution.

Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

The contents of the VL register determine the number of operations (VL/2) that are performed. All operations start with element 0 and 1 of the V*i*, V*j*, or V*k* registers and increment the element number by 2 for each operation that is performed. All results are delivered to register V*i*.

Instructions 140 through 145 can be executed in either the full vector logical or the second vector logical functional units, with the second unit being the first choice when the second vector logical unit is enabled. If the second vector logical unit is disabled, instructions 140 through 145 can be executed only in the full vector logical unit. Instructions 146 and 147 execute in the full vector logical unit only.

For instructions 140, 142, 144, and 146, a copy of S*j* is delivered first with the instruction to the vector unit and then to the functional unit. The copy is held as one of the operands until completion of the operation. Therefore, S*j* can be changed immediately after CIP issue of the instruction. For instructions 141, 143, 145, and 147, all operands are obtained from V registers.

The vector logical units are two registers deep for the 140 through 147 instruction execution data.

Instructions 140 and 141 form the logical products (AND) of operand pairs and enter the results into V*i*. Bits of an element of V*i* are set to 1 when the corresponding bits of S*j* or (V*j* element) and (V*k* element) are 1, as shown in the following example:

```
(Sj) or (Vj element)   = 1 1 0 0
(Vk element)           = 1 0 1 0
                         -------
(Vi element)           = 1 0 0 0
```

Instructions 142 and 143 form the logical sums (inclusive OR) of operand pairs and deliver the results to V*i*. Bits of an element of V*i* are set to 1 when one of the corresponding bits of (S*j*) or (V*j* element) and (V*k* element) is 1, as shown in the following example:

```
(Sj) or (Vj element)  = 1 1 0 0
(Vk element)          = 1 0 1 0
                        -------
(Vi element)          = 1 1 1 0
```

Instructions 144 and 145 form the logical differences (exclusive OR) of operand pairs and deliver the results to V*i*. Bits of an element are set to 1 when the corresponding bit of the contents of S*j* or (V*i* element) is different from (V*k* element), as shown in the following example:

```
(Sj) or (Vj element)  =  1 1 0 0
(Vk element)          =  1 0 1 0
                         -------
(Vi element)          =  0 1 1 0
```

Instructions 146 and 147 transmit operands to V*i*, depending on the contents of the VM register. Bit 63 of the mask corresponds to element 0 of a V register. Bit 0 corresponds to element 63. The operand pairs that are used for the selection depend on the instruction, (S*j*) and (V*k*) with the 146 instruction and (V*j*) and (V*k*) with the 147 instruction. If bit n of the vector mask is 1, the S*j* or V*j* operand is transmitted; if bit n of the mask is 0, the V*k* element is selected. The following two examples illustrate these points.

Example 1:

Instruction 146 is executed and the following register conditions exist:

```
(VL)      =   4
(VM)      =   06000000000000000000 (a 0110 bit pattern)
(S2)      =  -1
(V6, 00)  =   1
(V6, 01)  =   2
(V6, 02)  =   3
(V6, 03)  =   4
```

Instruction 146726 is executed. Following execution, the first four elements of V7 contain the following values:

(V7, 00) = 1
(V7, 01) = -1
(V7, 02) = -1
(V7, 03) = 4

The remaining elements of V7 are not altered.

Example 2:

Instruction 147 is executed and the following register conditions exist:

(VL) = 4
(VM) = 0600000000000000000000 (a 0110 bit pattern)

(V2, 00)	=	1	(V3, 00)	=	-1
(V2, 01)	=	2	(V3, 01)	=	-2
(V2, 02)	=	3	(V3, 02)	=	-3
(V2, 03)	=	4	(V3, 03)	=	-4

Instruction 147123 is executed. Following execution, the first four elements of V1 contain the following values:

(V1, 00) = -1
(V1, 01) = 2
(V1, 02) = 3
(V1, 03) = -4

The remaining elements of V1 are not altered.

Instructions 150 through 151

Machine Instruction	CAL Syntax	Description
150*ijk*	V*i* V*j*<A*k*	Shift (V*j* elements) left by (A*k*) places to V*i* elements.
150*ij*0[b]	V*i* V*j*<1	Shift (V*j* elements) left one place to V*i* elements.
151*ijk*	V*i* V*j*>A*k*	Shift (V*j* elements) right by (A*k*) places to V*i* elements.
151*ij*0[b]	V*i* V*j*>1	Shift (V*j* elements) right one place to V*i* elements.

[b] Special CAL syntax.

Special Cases

A special case exists for instructions 150 through 151; if *k* = 0, then (A*k*) = 1.

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- The A*k* register is reserved (except A0).
- Instruction 077 was issued in the previous CP.
- The vector instruction queue is full.

VIR Hold Issue Conditions

The 150 and 151 instructions hold issue at the VIR under any of the following conditions:

- V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- The vector shift functional unit is busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The execution time for vector instructions that are issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for the instruction that is waiting to issue in the VIR. The issue times for instructions 150 through 151 from the VIR are as follows:

For Functional Unit Busy:

- The functional unit is ready in (VL/2) + 1 CP.

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 6 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* is ready for V*j* or V*k* use (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- Vector Shift (150, 151) execution time is (VL/2) + 6 CPs until all of the data is available for use by the next instruction.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

Instructions 150 and 151 are executed in the vector shift functional unit. The contents of the VL register determine the number of operations performed. Operations start with element 0 and 1 of the V*i* and V*j* registers and end with elements specified by (VL)-1.

All shifts are end-off with zero fill. Unlike shift instructions 052 through 055, these instructions receive the shift count from A*k* rather than the *jk* fields and all 32 bits of A*k* are used for the shift count. Elements of V*i* are cleared if the shift count exceeds 63. All shift counts (A*k*) are considered positive.

Instruction 150*ijk* shifts the contents of the elements of register V*j* to the left by the amount specified by the contents of A*k* and enters the results into the elements of V*i*. The special form of this instruction shifts the contents of V*j* one place to the left and enters the results into V*i*.

Instruction 151*ijk* shifts the contents of the elements of register V*j* to the right by the amount specified by the contents of A*k* and enters the results into the elements of V*i*. The special form of this instruction shifts the contents of V*j* one place to the right and enters the results into V*i*.

The vector shift unit is three registers deep for execution data.

Instructions 152 through 153

Machine Instruction	CAL Syntax	Description
152*ijk*	V*i* V*j*,V*j*<A*k*	Double shift of (V*j* elements) left (A*k*) places to V*i* elements.
152*ij*0[b]	V*i* V*j*,V*j*<1	Double shift of (V*j* elements) left one place to V*i* elements.
153*ijk*	V*i* V*j*,V*j*>A*k*	Double shift of (V*j* elements) right (A*k*) places to V*i* elements.
153*ij*0[b]	V*i* V*j*,V*j*>1	Double shift of (V*j* elements) right one place to V*i* elements.

[b] Special CAL syntax.

Special Cases

A special cases exist for instructions 152 through 153; if *k* = 0, then (A*k*) = 1.

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- The A*k* register is reserved (except A0).
- Instruction 077 was issued in the previous CP.
- The vector instruction queue is full.

VIR Hold Issue Conditions

The instructions hold issue at the VIR under any of the following conditions:

- V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- The vector shift functional unit is busy.
- Instructions 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in the VIR. The issue times for instructions 152 through 153 from VIR are as follows:

For Functional Unit Busy:

- The functional unit is ready in (VL/2) + 1 CP (except for the 152 or 153 instruction combination, which requires an additional CP).

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 7CPs for the 152 instruction and (VL/2) + 6 CPs for the 153 instruction.

- V*i* is ready for V*j* or V*k* use immediately (because of chaining).

- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.

- V*j* is ready for V*i* use immediately (because of tailgating).

- Vector Shift 153 execution time is (VL/2) + 6 CPs until all the data is available for use by the next instruction.

- Vector Shift 152 execution time is (VL/2) + 7 CPs until all the data is available for use by the next instruction.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

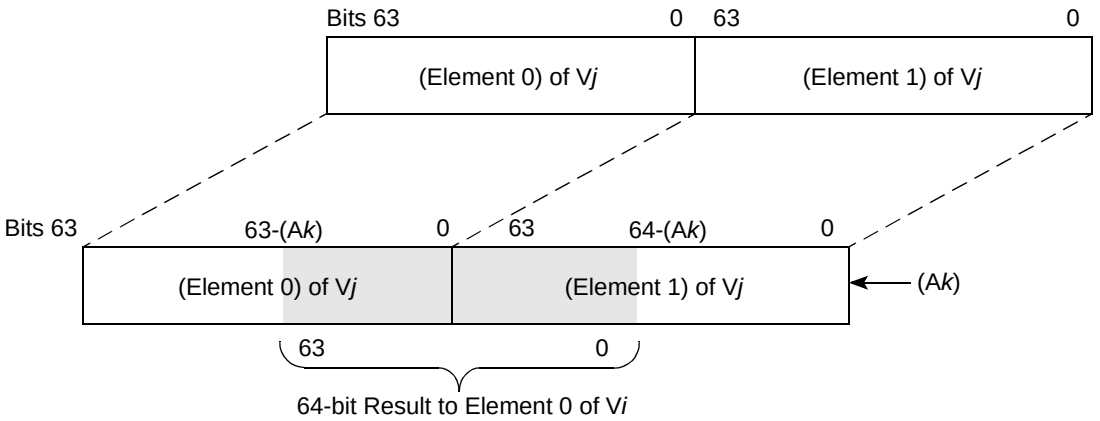
The vector shift functional unit executes instructions 152 and 153. The instructions shift 128-bit values that are formed by logically joining the contents of two elements of the V*j* register. The direction of the shift determines whether the high-order bits or the low-order bits of the result are sent to V*i*. Shift counts are obtained from register A*k*. All shifts are end-off with zero fill. The contents of the VL register determine the number of operations performed.

The vector shift functional unit is four or three registers deep for the 152/153 instruction execution data respectively.

Instruction 152 performs left shifts. The operation starts with element 0 and 1 of V*j*. If the content of VL is 1, element 0 is joined with 64 bits of 0's, and the resulting 128-bit quantity is then shifted left by the amount specified by the contents of A*k*. Only this one operation is performed. The 64 high-order bits that remain are transmitted to element 0 of V*i*.

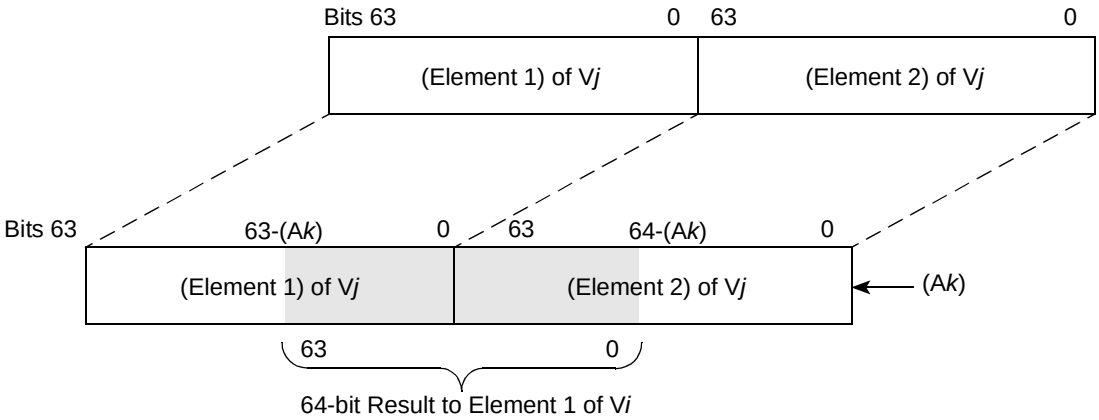
If the content of VL is 2, the operation starts by joining element 0 of V*j* with element 1; the resulting 128-bit quantity is then left shifted by the amount specified by the contents of A*k*. The high-order 64 bits that remain are transmitted to element 0 of V*i*. Figure 52 shows this operation.

Figure 52. Vector Left Double Shift, First Element, VL Greater than 1



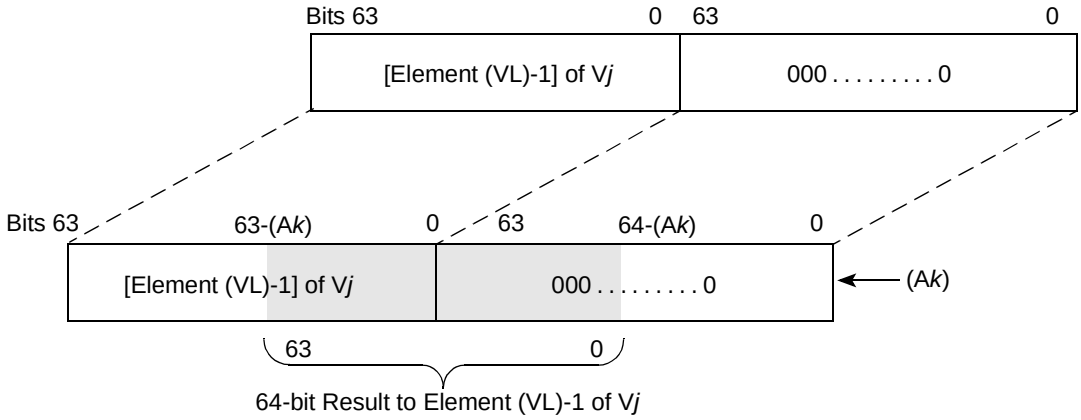
If the content of VL is greater than 2, the operation continues by joining element 1 with element 2 and transmitting the 64-bit result to element 1 of V*i*. Figure 53 shows this operation. Because the shift unit is a 2-pipe design, both shifts occur at the same time producing two 64-bit results to elements of 0 and 1 of V*i*.

Figure 53. Vector Left Double Shift, Second Element, VL Greater than 2



If the content of VL is 2, element 1 is joined with 64 bits of 0's and only two operations are performed. In general, the last element of V*j*, as determined by the contents of VL, is joined with 64 bits of 0's. Figure 54 shows this operation.

Figure 54. Vector Left Double Shift, Last Element



Note: The elements are numbered 0 through 63 in the V registers; therefore, element (VL)-1 refers to the VLth elements.

If the content of A*k* is greater than or equal to 128, the result is all 0's. If the content of A*k* is greater than 64, the result register contains at least the contents of A*k* -64 zeroes.

Example 1:

If instruction 152 is to be executed and the following register conditions exist, instruction 152541 is executed:

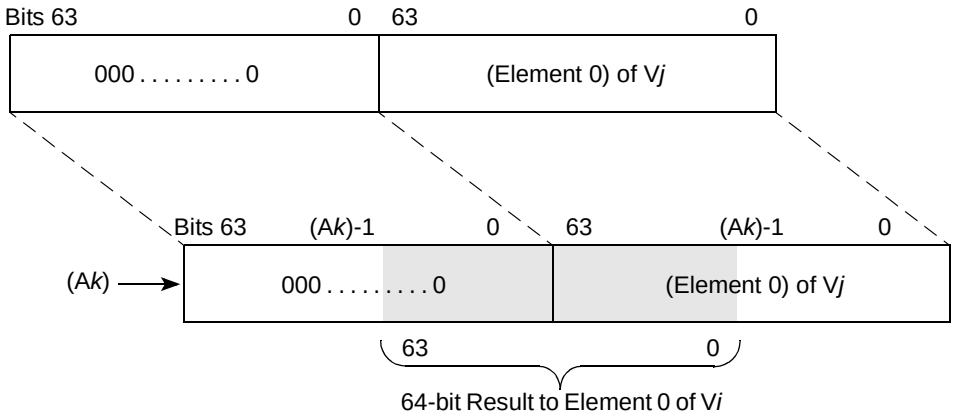
```
(VL)      =  4
(A1)      =  3
(V4, 00)  =  0  00000  0000  0000  0000  0007
(V4, 01)  =  0  60000  0000  0000  0000  0005
(V4, 02)  =  1  00000  0000  0000  0000  0006
(V4, 03)  =  1  60000  0000  0000  0000  0007
```

Following execution, the first four elements of V5 contain the following values:

```
(V5, 00)  =  0  00000  0000  0000  0000  0073
(V5, 01)  =  0  60000  0000  0000  0000  0054
(V5, 02)  =  0  00000  0000  0000  0000  0067
(V5, 03)  =  0  60000  0000  0000  0000  0070
```

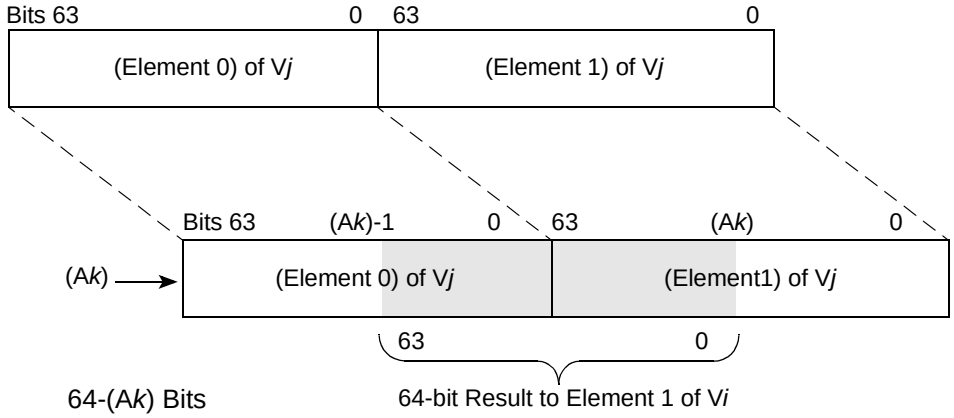
Instruction 153 performs right shifts. The original element 0 of V*j* is joined with 64 high-order bits of 0's and the 128-bit quantity is shifted right by the amount specified by (A*k*). The 64 low-order bits of the result are transmitted to element 0 of V*i*. Figure 55 shows this operation.

Figure 55. Vector Right Double Shift, First Element



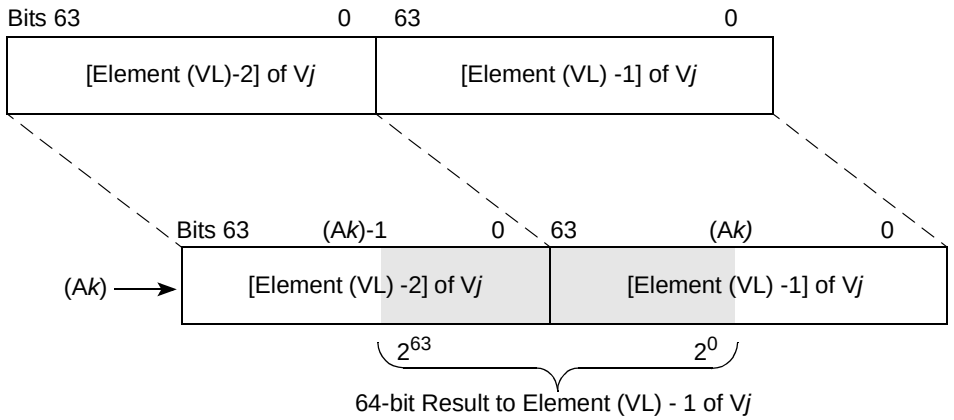
If the content of VL equals 1, only one operation is performed. However, instruction execution continues by joining element 0 with element 1, shifting the 128-bit quantity by the amount specified by (A*k*), and transmitting the result to element 1 of V*i*. Figure 56 shows this operation. (Both operations occur during the same CP because of the 2-pipe design of the shift unit.)

Figure 56. Vector Right Double Shift, Second Element, VL Greater than 1



The last operation performed by the instruction joins the last element of V*j*, as determined by the contents of VL, with the preceding element (refer to Figure 57).

Figure 57. Vector Right Double Shift, Last Operation



Note: Elements are numbered 0 through 63 in the V registers; therefore, element (VL)-1 refers to the VLth element.

Example 2:

If instruction 153 is executed and the following register conditions exist, then instruction 153026 is executed:

```
(VL)      =  4
(A6)      =  3
(V2, 00)  =  0  00000  0000  0000  0000  0017
(V2, 01)  =  0  60000  0000  0000  0000  0006
(V2, 02)  =  1  00000  0000  0000  0000  0006
(V2, 03)  =  1  60000  0000  0000  0000  0007
```

Following execution, register V0 contains the following values:

```
(V0, 00)  =  0  00000  0000  0000  0000  0001
(V0, 01)  =  1  66000  0000  0000  0000  0000
(V0, 02)  =  1  50000  0000  0000  0000  0000
(V0, 03)  =  1  56000  0000  0000  0000  0000
```

The remaining elements of register V0 are not altered.

Instructions 154 through 157

Machine Instruction	CAL Syntax	Description
154*ijk*	V*i* S*j*+V*k*	Transmit integer sums of (S*j*) and (V*k* elements) to V*i* elements.
155*ijk*	V*i* V*j* + V*k*	Transmit integer sums of (V*j* elements) and (V*k* elements) to V*i* elements.
156*ijk*	V*i* S*j*-V*k*	Transmit integer differences of (S*j*) and (V*k* elements) to V*i* elements.
156*i*0*k*[b]	V*i* -V*k*	Transmit two's complement of (V*k* elements) to V*i* elements.
157*ijk*	V*i* V*j*-V*k*	Transmit integer differences of (V*j* elements) and (V*k* elements) to V*i* elements.

[b] Special CAL syntax.

Special Cases

The following special cases exist for instructions 154 through 157:

- For instruction 154, if *j* = 0, then (S*j*) = 0 and (V*i* element) = (V*k* element).

- For instruction 156, if *j* = 0, then (S*j*) = 0 and (V*i* element) = -(V*k* element).

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- For instructions 154 and 156, if the S*j* register is reserved (except S0).
- For instructions 154 and 156, if a 077 instruction was issued in the previous CP.
- The VIQ is full.

VIR Hold Issue Conditions

The instructions hold issue in the VIR under any of the following conditions:

- V*i* and V*k* (V*j* for 155 and 157) registers are reserved unless chaining or tailgating is permitted.
- Vector add functional unit is busy.
- Instruction 076*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The execution time when the vector instruction issues directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in the VIR. The issue times for instructions 154 through 157 from VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP.

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 5CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* or V*k* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.

- V*j* or V*k* is ready for V*i* use immediately (because of tailgating).
- Execution time for the vector add/differences instructions (154 through 157) is (VL/2) + 5 CP until all the data is available.

Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

The vector add functional unit executes instructions 154 through 157. Instructions 154 and 155 perform integer addition. Instructions 156 and 157 perform integer subtraction. The contents of the VL register determine the number of additions or subtractions that are performed. All operations start with element 0 of the V registers and increment the element number by 2 for each operation performed. All results are delivered to elements of V*i*. No overflow is detected.

Instructions 154 and 156 deliver a copy of the contents of S*j* to the functional unit, where the copy is retained as one of the operands until the vector operation completes. The other operand is an element of V*k*. For instructions 155 and 157, both operands are obtained from V registers.

Instruction 154*ijk* adds the contents of S*j* to each element of V*k* and enters the results into elements of V*i*. Elements of V*k* are transmitted to V*i* if the *j* designator is 0.

Instruction 155*ijk* adds the contents of the elements of register V*j* to the contents of the corresponding elements of register V*k* and enters the results into the elements of register V*i*.

Instruction 156*ijk* subtracts the contents of each element of V*k* from the contents of register S*j* and enters the results into the elements of register V*i*. Instruction 156*i*0*k* transmits the negative (two’s complement) of each element of V*k* to V*i*.

Instruction 157*ijk* subtracts the contents of the elements of register V*k* from the contents of the corresponding elements of register V*j* and enters the results into the elements of register V*i*.

The integer add/subtract functional unit is two registers deep for execution data.

Instructions 160 through 167

Machine Instruction	CAL Syntax	Description
160*ijk*	V*i* S*j**FV*k*	Transmit floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
161*ijk*	V*i* V*j**FV*k*	Transmit floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
162*ijk*	V*i* S*j**HV*k*	Transmit half-precision rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
163*ijk*	V*i* V*j**HV*k*	Transmit half-precision rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
164*ijk*	V*i* S*j**RV*k*	Transmit rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements.
165*ijk*	V*i* V*j**RV*k*	Transmit rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements.
166*ijk*	V*i* S*j**V*k*	Transmit 32-bit integer product of (S*j*) and (V*k* elements) to V*i* elements.
167*ijk*	V*i* V*j**IV*k*	Transmit reciprocal iterations: 2-(V*j* elements)*(V*k* elements) to V*i* elements.

Special Cases

The following special case exists for instructions 160, 162, 164, and 166:

- If $j = 0$, then (Sj) = 0.

Hold Issue Conditions

The instructions hold issue under the following conditions:

- For instructions 160, 162, 164, and 166, when the S*j* register is reserved (except S0), or when a 077 instruction was issued the previous CP.
- The VIQ is full.

VIR Hold Issue Conditions

The instructions hold issue at the VIR under any of the following conditions:

- V*i* and V*k* (V*j* for 161, 163, 165, and 167) registers are reserved unless chaining or tailgating is permitted.
- Floating-point multiply functional unit is busy.

- Instruction 070*ij*6, 073, 076, or 077 issued from VIR the previous CP.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in VIR. The issue times for instructions 160 through 167 from VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP.

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 10 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* or V*k* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* or V*k* is ready for V*i* use immediately (because of tailgating).
- For floating-point multiply instructions (160 through 167), execution time is (VL/2 + 10 CPs until all of the data is available for use by the next instruction.
- Unit busy time between the floating-point multiply and second vector logical functional units is (VL/2) + 1 CP.
- Unit busy time between the second vector logical and floating-point multiply functional units is (VL/2) + 1 CP.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

The floating-point multiply functional unit executes instructions 160 through 167. The contents of the VL register determine the number of operations performed by an instruction. All operations start with element 0 and 1 of the V registers and increment the element number by 2 for each successive operation.

The functional unit operates under the assumption that operands are in floating-point format. Instructions 160, 162, 164, and 166 send a copy of the contents of S*j*, with the instruction, to the functional unit, where the copy is retained as one of the operands until the completion of the operation. Therefore, the contents of S*j* can be changed immediately without affecting the vector operation. The other operand is an element of V*k*. For instructions 161, 163, 165, and 167, both operands are obtained from V registers. All results are delivered to elements of V*i*. If either operand is not normalized, there is no guarantee that the product is normalized. If neither operand is normalized, the product is not normalized.

Instruction 160*ijk* forms the floating-point products of the contents of S*j* and elements of V*k* and enters the results into elements of V*i*.

Instruction 161*ijk* forms the floating-point products of the contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*.

Instruction 162*ijk* forms the half-precision rounded floating-point products of the contents of the S*j* register and the contents of elements of the V*k* register and enters the results into elements of V*i*. This instruction can be used in a divide algorithm when only 30 bits of accuracy are required.

Instruction 163*ijk* forms the half-precision rounded floating-point products of the contents of elements of the V*j* register and elements of the V*k* register and enters the results into elements of V*i*. This instruction can be used in a divide algorithm when only 30 bits of accuracy are required.

Instruction 164*ijk* forms the rounded floating-point products of the contents of the S*j* register and the contents of elements of V*k* and enters the results into elements of V*i*.

Instruction 165*ijk* forms the rounded floating-point products of the contents of elements of the V*j* register and elements of the V*k* register and enters the results into elements of V*i*.

Instruction 166*ijk* forms the 32-bit product of the contents of S*j* and the elements of V*k* and enters the result into elements of V*i*. The S*j* operand must be left-shifted by 31 (decimal) places and the V*k* operand must be left-shifted by 16 (decimal) places before the 166*ijk* instruction executes.

Instruction 167*ijk* forms 2 minus the floating-point products of the contents of the elements of V*j* and elements of V*k* and enters the results into the elements of V*i*. This instruction is used in the division operation sequence of instructions.

The floating-point multiply functional unit is 7 registers deep for execution data.

Instructions 170 through 173

Machine Instruction	CAL Syntax	Description
170*ijk*	V*i* S*j*+FV*k*	Transmit floating-point sums of (S*j*) and (V*k* elements) to V*i* elements.
170*i*0*k*[b]	V*i* +FV*k*	Transmit normalized (V*k* elements) to V*i* elements.
171*ijk*	V*i* V*j*+FV*k*	Transmit floating-point sums of (V*j* elements) and (V*k* elements) to V*i* elements.
172*ijk*	V*i* S*j*-FV*k*	Transmit floating-point differences of (S*j*) and (V*k* elements) to V*i* elements.
172*i*0*k*[b]	V*i* -FV*k*	Transmit normalized negative of (V*k* elements) to V*i* elements.
173*ijk*	V*i* V*j*-FV*k*	Transmit floating-point differences of (V*j* elements) and (V*k* elements) to V*i* elements.

[b] Special CAL syntax.

Special Cases

The following special case exists for instructions 170 and 172:

- If *j* = 0, then (S*j*) = 0.

Hold Issue Conditions

The instructions hold issue under any of the following conditions:

- For instructions 170 and 172, if the S*j* register is reserved (except S0).
- For instructions 170 through 173, 077 issued last CP, or VIQ is full.

VIR Hold Issue Conditions

The instructions hold issue at the VIR under any of the following conditions:

- V*i* and V*k* (V*j* for 171, 173) registers are reserved unless chaining or tailgating is permitted.

- Floating-point add functional unit is busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from VIR the previous CP.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in the VIR. The issue times for instructions 170 through 173 from the VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP.

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 9 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* or V*k* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* or V*k* is ready for V*i* use immediately (because of tailgating).
- The execution time for the floating add/difference (170 through 173) instructions is (VL/2) + 9 CPs until all the data is available for use by the next instruction.

  Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

The floating-point add functional unit executes instructions 170 through 173. Instructions 170 and 171 perform floating-point addition; instructions 172 and 173 perform floating-point subtraction. The contents of the VL register determine the number of additions or subtractions that are performed by an instruction. All operations start with element 0 and 1 of the V registers and

increment the element number by 2 for each operation performed. All results are delivered to V*i* in normalized state, and the results are normalized even if the operands are not normalized.

Instructions 170 and 172 deliver a copy of (S*j*) to the functional unit, where it remains as one of the operands until the completion of the operation. The other operand is an element of V*k*. For instructions 171 and 173, both operands are obtained from V registers.

Instruction 170*ijk* forms the floating-point add by summing the contents of the S*j* and the elements of V*k* and enters the results into elements of V*i*.

The special form of the instruction (170*i*0*k*) normalizes the contents of the elements of V*k* and enters the results into elements of register V*i*.

Instruction 171*ijk* forms the floating-point sums of the contents of the elements of V*j* and elements of V*k* and enters the results into the elements of register V*i*.

Instruction 172*ijk* forms the floating-point differences of the contents of S*j* and elements of register V*k* and enters the results into elements of register V*i*. Instruction 172*i*0*k* transmits the negatives of floating-point quantities in the elements of V*k* to elements of V*i*.

Instruction 173*ijk* forms the floating-point differences of the contents of the elements of register V*j* less the contents of the elements of registers V*k* and enters the results into the elements of register V*i*.

The floating point add functional unit is 6 CPs deep for execution data.

Instruction 174

Machine Instruction	CAL Syntax	Description
174*ij*0	V*i* /HV*j*	Transmit floating-point reciprocal approximation of (V*j* elements) to V*i* elements.

Special Cases

When a 174 instruction issues, if the V*j* register element is not normalized, the V*i* register element is invalid. Bit 47 of the V*j* register element must be 1. This bit is not tested.

Hold Issue Conditions

The 174 instruction holds issue when a the VIQ is full.

VIR Hold Issue Conditions

The instruction holds issue at the VIR under any of the following conditions:

- The V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- The reciprocal or pop/parity/leading-zero functional units are busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from VIR the previous CP.

Execution Time

The execution time for a vector instruction that issues directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for an instruction that is waiting to issue in the VIR.

The issue times for the 174 instruction from the VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP (for either a Pop/Parity/Leading-zero or a reciprocal).

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 17 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- Execution time for the floating-point reciprocal (174) instruction is (VL/2) + 17 CPs until all the data is available for use by the next instruction.

Description

The reciprocal approximation functional unit executes instruction 174. The instruction forms an approximate value of the reciprocal of the normalized floating-point quantity in each element of V*j* and enters the result into elements of V*i*. The contents of the VL register determine the number of elements for which approximations are found.

Instruction 174 occurs in the divide sequence to compute the quotients of floating-point quantities. The reciprocal approximation instruction produces results of 30 significant bits. The low-order 18 bits are 0's. The number of significant bits can be extended to 48 by using the reciprocal iteration instruction and a multiply instruction.

The reciprocal approximation functional unit is 14 CPs deep for execution data.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Instruction 174ij1 through 174ij2

Machine Instruction	CAL Syntax	Description
174*ij*1	V*i* PV*j*	Transmit population count of (V*j* elements) to V*i* elements.
174*ij*2	V*i* QV*j*	Transmit population count parity of (V*j* elements) to V*i* elements.

Hold Issue Conditions

These instructions hold issue when the VIQ is full.

VIR Hold Issue Conditions

These instructions hold issue at the VIR under any of the following conditions:

- The V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- The reciprocal unit or pop/parity/leading-zero functional units are busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from VIR the previous CP.

Execution Time

The execution time for a vector instruction that issues directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in the VIR.

The issue times for instructions 174*ij*1 through 174*ij*2 issued from the VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP.

For Vector Register Busy:

- V*i* is ready for V*i* use in (VL/2) + 6 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- Execution time for the pop/parity (174) instruction is (VL/2) + 6 CPs until all the data is available for use by the next instruction.

Description

The vector population/parity/leading-zero functional unit executes instructions 174*ij*1, 174*ij*2 and 174*ij*3 and shares some logic with the reciprocal approximation functional unit.

Instruction 174*ij*1 counts the number of bits that are 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 cleared.

Instruction 174*ij*2 counts the number of bits that are set to 1 in each element of V*j*. The least significant bit of each element result shows whether the result is an odd or even number. Only the least significant bit of each element is transferred to the least significant bit position of the corresponding element of register V*i*. The remainder of the element is set to 0’s. The actual population count results are not transferred.

The pop/parity/leading-zero functional unit is 3 CPs deep for pop/parity and 2 CPs deep for leading-zero execution data.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Instruction 174*ij*3

Machine Instruction	CAL Syntax	Description
174*ij*3	V*i* ZV*j*	Transmit leading-zero count of (V*j* elements) to V*i* elements.

CIP Hold Issue Conditions

- The VIQ is full.

VIR Hold Issue Conditions

This instruction holds issue at the VIR under any of the following conditions:

- The V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- Instruction 076*ij*6, 073, 076, or 077 is issued from VIR the previous CP.
- The reciprocal unit or pop/parity/leading-zero functional unit is busy.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer that the execution time for instructions that are waiting to issued in VIR.

The issue time for the 174*ij*3 instruction is as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP (for either a pop/parity/leading-zero or reciprocal).

For Vector Register Busy

- V*i* is ready for V*i* use in (VL/2) + 5 CPS

- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- Unit busy time for leading-zero instruction is (VL/2) + 5 CPs until all data is available for use by the next instruction.
- Instruction issue time is 1 CP from CIP.

Execution Timing from VIR

- Instruction issue time is 1 CP from VIR.
- Execution time for leading-zero is (VL/2) + 5 CPs until all data is available for use by the next instruction (with no delays).

Note: Vector instructions may or may not start to execute immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain starting with that load.

Description

Instruction 174*ij*3 counts the number of leading zeros (left to right) in the (V*j* elements) and enters the result into the low-order 7 bits of the V*i* elements. All bits above bit 6 in the V*i* elements are cleared. The V*i* element is set to 64 if the contents of the V*j* element is zero. Instruction 174*ij*3 executes in the pop/parity/leading-zero functional unit.

The pop/parity/leading-zero functional unit is three registers deep for pop/parity execution data and two registers deep for leading-zero execution data.

Instruction 1740*j*4

Machine Instruction	CAL Syntax	Description
1740*j*4	BMM V*j*	Transmit (V*j* elements) to BMM.

CIP Hold Issue Conditions

- The VIQ is full.

VIR Hold Issue Conditions

This instruction holds issue at the VIR under any of the following conditions:

- The V*j* registers are reserved unless chaining is permitted.
- Instruction 070*ij*6, 073, 076, or 077 is issued from VIR the previous CP.
- The floating-point add or bit-matrix multiply functional unit is busy with a 170 through 173, 1740*j*4, or 174*ij*6 instruction.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer that the execution time for instructions that are waiting to issue in VIR.

The issue time for the 1740*j*4 instruction is as follows:

For Functional Unit Busy:

- Functional unit ready in VL/2 rounded up to 8, 16, 24, or 32 CPs + 2 CPs.

For Vector Register Busy

- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- Unit busy time for both the floating-point add and the BMM functional units is (VL/2) +1 CP and 10, 18, 26, or 34 CPs (per VL/2) respectively.
- Instruction issue time is 1 CP from CIP.

Execution Timing from VIR

- Instruction issue time is 1 CP from VIR.
- BMM register is ready for use in 10, 18, 26, or 34 CPs from VIR (when no delays).

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain starting with that load.

Description

Instruction 1740*j*4 loads 64 BMM registers of the BMM functional unit with the contents of V*j* elements. The (VL) specifies the number of V*j* elements to be loaded into the BMM register; all remaining BMM registers are cleared to all 0's. The BMM register is loaded two at a time in this dual-port design. BMM loads occur in multiples of 8 CPs per (VL/2). A VL count of 1 through 16 used 8 CPs, 17 through 32 uses 16 CPs, 33 through 48 uses 24 CPs, and 49 through 64 uses 32 CPs to load (if no data delays occur). The functional unit is busy for an additional 2 CPs. This BMM load sets the BML bit in the status register and the exchange package. BMM operations are described in an earlier section of this manual.

The 070*ij*6 and 174*ij*6 instructions use the loaded BT registers to generate the bit-matrix product with (S*j*) and (V*i* elements) that is sent to the S*i* and V*i* registers respectively.

The BMM functional unit shares the operand input bus with the floating-add functional unit. The functional unit is busy if either unit is receiving data from the V registers. This busy time exists for (VL/2) +1 CP for the floating-add unit (when no delays are encountered). The busy time for the BT register load is (VL/2 rounded up to 8, 16, 24, or 32 CPs) + 2 CPs (when no delays are encountered).

The BMM functional unit is two registers deep for execution data.

Instruction 174*ij*6

Machine Instruction	CAL Syntax	Description
174*ij*6	V*i* V*j**BT	Transmit the bit-matrix product of (V*j* elements) and transpose of (BMM) to V*i*.

CIP Hold Issue Conditions

- The VIQ is full.

VIR Hold Issue Conditions

This instruction holds issue at the VIR under any of the following conditions:

- The V*i* and V*j* registers are reserved unless chaining or tailgating is permitted.
- Instruction 070*ij*6, 073, 076, or 077 was issued from VIR the previous CP.
- The floating-point add or bit-matrix multiply functional unit is busy with a 170 through 173, 1740*j*4, or 174*ij*6 instruction.

Execution Time

The execution time for a vector instruction that is issued directly from the CIP to the functional unit (through the VIR) is 2 CPs longer that the execution time for instructions that are waiting to issue in VIR. The issue time for the 174*ij*6 instruction is as follows:

For Functional Unit Busy:

- Functional unit ready in (VL/2) + 1CP.

For Vector Register Busy

- V*i* is ready for V*i* use in (VL/2) + 5 CPs.
- V*i* is ready for V*j* or V*k* use immediately (because of chaining).
- V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
- V*j* is ready for V*i* use immediately (because of tailgating).
- The BMM 174*ij*6 execution time is (VL/2) + 5 CPs until all the data is available for use by the next instruction.
- Unit busy time for both the floating-point add and the BMM functional units is (VL/2) +1 CP.
- Instruction issue time is 1 CP from CIP.
- Register V*i* is ready in (VL/2) + 7 CPs (from CIP when no delays occur).

Execution Timing from VIR

- Instruction issue time is 1 CP from VIR.
- Register V*i* is ready with the first data in 5 CPs from VIR (when no delays occur).

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain starting with that load.

Description

Instruction 174*ij*6 uses the BMM functional unit. This functional unit contains the 64×64-bit BMM register and it generates the bit-matrix product that is returned to the V*i* register. The BMM register must be loaded with the 1740*j*4 instruction before use. This load sets the BML bit in the status register and the exchange package. BMM operations are described in an earlier section of this manual.

The 174*ij*6 instruction uses the contents of the VL register to determine the number of operations performed by an instruction. All operations start with element 0 and 1 of the V registers and increment the element number by 2 for each successive operation. The instruction executes in (VL/2) CPs if no delays occur.

The BMM functional unit shares the operand input bus with the floating-add functional unit. The functional unit is busy if either unit is receiving data from the V registers. This busy time exists for (VL/2) +1 CP when no delays are encountered.

The BMM functional unit is designed as a dual-pipe unit with the same set of 64 BT registers used for both pipes. The BMM functional unit is two registers deep for execution data.

Instruction 175

Machine Instruction	CAL Syntax	Description
1750*j*0	VM V*j*,Z	Set VM bit if (V*j* element) = 0.
1750*j*1	VM V*j*,N	Set VM bit if (V*j* element) ≠ 0.
1750*j*2	VM V*j*,P	Set VM bit if (V*j* element) ≥ 0.
1750*j*3	VM V*j*,M	Set VM bit if (V*j* element) < 0 (V*j* is negative).
175*ij*4	V*i*,VM V*j*,Z	Set VM bit if (V*j* elements) = 0; also, the compressed indices of the V*j* element = 0 are stored in V*i*.
175*ij*5	V*i*,VM V*j*,N	Set VM bit if (V*j* elements) ≠ 0; also, the compressed indices of the V*j* element ≠ 0 are stored in V*i*.
175*ij*6	V*i*,VM V*j*,P	Set VM bit if (V*j* elements) ≥ 0; also, the compressed indices of the V*j* element ≥ 0 are stored in V*i*.
175*ij*7	V*i*,VM V*j*,M	Set VM bit if (V*j* elements) <0; also, the compressed indices of the V*j* element <0 are stored in V*i*.

Special Cases

The following special cases exist for instruction 175:

- If the V*j* element *n* = 0, and *k* = 0 or 4, then VM bit *n* = 1.
- If the V*j* element *n* ≠ 0, and *k* = 1 or 5, then VM bit *n* = 1.
- If the V*j* element *n* is positive (0 is a positive condition), and *k* = 2 or 6, then VM bit *n* = 1.
- If the V*j* element *n* is negative, and *k* = 3 or 7, then VM bit *n* = 1.
- If the V*j* element *n* = 0 and *k* = 4, then the V*i* compressed element = *n*.
- If the V*j* element *n* ≠ 0 and *k* = 5, then the V*i* compressed element = *n*.
- If the V*j* element *n* is positive (0 is a positive condition), and *k* = 6, then V*i* compressed element = *n*.
- If the V*j* element *n* is negative and *k* = 7, then V*i* compressed element = *n*.
- Only the indices of successful V*j* element tests (for which VM = 1) are written into contiguous elements of V*i*.

Hold Issue Conditions

The 175 instruction holds issue when the VIQ is full.

VIR Hold Issue Conditions

This instruction holds issue at the VIR under any of the following conditions:

- V*i* (V*i* for 175*ij*4 through 175*ij*7) register is reserved unless vector chaining or tailgating is permitted.
- The main vector logical functional unit is busy.
- Instruction 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The execution time for a vector instruction that issues directly from the CIP to the functional unit (through the VIR) is 2 CPs longer than the execution time for instructions that are waiting to issue in the VIR. The instruction issue times for the 175 instruction that is issued from the VIR are as follows:

For Functional Unit Busy:

- Functional unit is ready in (VL/2) + 1 CP (except (VL/2) + 4 CP for a 140 through 147 instruction following a 175).
- For Vector Register Busy
  - V*i* is ready for V*i* use in (VL/2) + 6 CPs.
  - V*i* is ready for V*j* or V*k* use immediately (because of chaining).
  - V*j* is ready for V*j* or V*k* use in (VL/2) + 2 CPs.
  - V*j* is ready for V*i* use immediately (because of tailgating).
- Execution time for the vector logical (175 with *k* = 0 through 3) instruction is (VL/2) + 3 CPs until the vector mask is available for use by the same vector logical unit.
- Execution time for the vector logical (175 with *k* = 4 through 7) instruction is (VL/2) + 5 CPs until all the data is available for use by the next instruction.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Description

The full vector logical functional unit executes the vector mask and compressed index instruction 175. Instructions 1750*j*0 through 1750*j*3 create a mask in the VM register. The 64 bits of the VM register correspond to the 64 elements of V*j*. Elements of V*j* are tested for the specified condition. If the condition is true for an element, the corresponding bit is set to 1 in the VM register. If the condition is not true, the bit is set to 0.

Instructions 175*ij*4 through 175*ij*7 create an identical vector mask (as in instructions 1750*j*0 through 1750*j*3) and a compressed index list in register V*i*, based on the results of testing the contents of the elements of register V*j*.

The contents of the VL register determine the number of elements that are tested; however, the entire VM register is cleared before elements of V*j* are tested. If the content of an element is 0, it is considered positive. Element 0 corresponds to bit 63, element 1 to bit 62, and so on, from left to right in the VM register.

The type of test made by the instruction depends on the low-order 2 bits of the *k* designator. The high-order bit of the *k* designator is used to select the compressed index option.

For instruction 1750*j*0, if the V*j* register element is 0, the VM bit is set to 1. If the V*j* register element is not 0, the VM bit is set to 0.

For instruction 1750*j*1, if the V*j* register element is not 0, the VM bit is set to 1. If the V*j* register element is 0, the VM bit is set to 0.

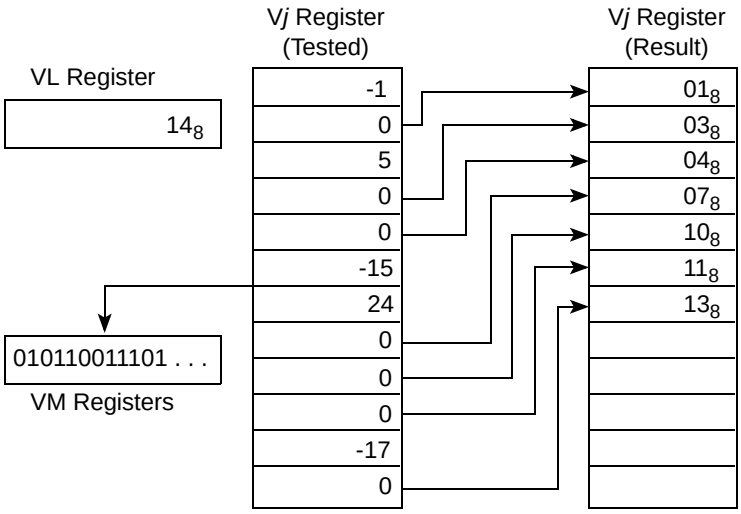
For instruction 1750*j*2, if the V*j* register element is positive, the VM bit is set to 1. If the V*j* register element is negative, the VM bit is set to 0. A value of 0 is positive.

For instruction 1750*j*3, if the V*j* register element is negative, the VM bit is set to 1. If the V*j* register element is positive, the VM bit is set to 0. A value of 0 is positive.

Instructions 175*ij*4, 175*ij*5, 175*ij*6, and 175*ij*7 are compressed index instructions. These instructions test for zero, nonzero, positive, and negative elements, respectively. A vector mask and a compressed index are generated, based on the tested condition.

For instruction 175*ij*4, if the V*j* register element is 0, the VM bit is set to 1 and the V*i* register compressed element is set to the V*j* register element index. If the V*j* register element is 0, data is written to the V*i* register elements, and the V*i* register element pointer is advanced. Refer to Figure 58 for an example of the 175*ij*4 instruction.

Figure 58. Compressed Index Example



For instruction 175*ij*5, if the V*j* register element is not 0, the VM bit is set to 1 and the V*i* register compressed element is set to the V*j* register element index. If the V*j* register element is not 0, data is written to the V*i* register elements, and the V*i* register element pointer is advanced.

For instruction 175*ij*6, if the V*j* register element is positive, the VM bit is set to 1 and the V*i* register compressed element is set to the V*j* register element index. If the V*j* register element is positive, data is written to the V*i* register elements, and the V*i* register element pointer is advanced (a value of 0 is positive).

For instruction 175*ij*7, if the V*j* register element is negative, the VM bit is set to 1 and the V*i* register compressed element is set to the V*j* register element index. If the V*j* register element is negative, the V*i* register elements are written to the V*i* register elements, and the V*i* register element pointer is advanced.

The contents of the VL register determine the number of elements that are tested. The VM register bits that correspond to the untested elements of the V*j* register are cleared.

Vector mask instruction 175i*jk*, *k* = 0 through 3, and the compressed index instructions 175*ijk*, *k* = 4 through 7, are a vector counterpart to the scalar conditional branch instructions.

The main vector logical unit is 3 CPs deep for the VM and 2 CPs deep for the compressed index data path to V*i* for the 175 instruction.

Instruction 176 through 177

Machine Instruction	CAL Syntax	Description
176*i*0*k*	V A0,A*k*	Load from memory starting at (A0) increased by (A*k*) and load into V*i* elements.
176*i*00	V*i* ,A0,1	Load from consecutive memory addresses starting with (A0) and load into V*i* elements.
176*i*1*k*	V*i* ,A0,V*k*	Load from memory using memory address ((A0) + (V*k*)) and load into V*i* elements.
1770*jk*	,A0,A*k* V*j*	Store (V*j* elements) to memory starting at (A0) increased by (A*k*).
1770*j*0	,A0,1 V*j*	Store (V*j* elements) to memory in consecutive addresses starting with (A0).
1771*jk*	,A0,V*k* V*j*	Store (V*j* elements) to memory using memory address ((A0) + (V*k*)).

Special Cases

The following special cases exist for instructions 176 through 177:

- For instructions 176*i*0*k* and 1770*jk*, increment (A0) by 1 if *k* = 0.
- Instructions 176 and 177 use port B. If port B is busy, instructions 176 and 177 use port A.
- Only bits 0 through 31 of the A0, A*k*, and V*k* registers are used to calculate memory addresses. Refer to the "Calculating Absolute Memory Address" subsection for additional information.
- Memory conflicts slow the loading or storing of individual vector elements.

- For instruction 176, if there is an instruction that uses the 176 result register as a source (chaining), the execution of that instruction is delayed whenever there is a delay in instruction 176 results.

Hold Issue Conditions

The 176 through 177 instructions hold issue under any of the following conditions:

- The A0 register is reserved.
- For instruction 176, when ports A and B are busy.
- For instruction 177, when port A or B is busy with a write reference or if ports A and B are busy.
- For instructions 176*i*1*k* and 1771*jk*, when either 176*i*1*k* or 1771*jk* is in progress.
- For instructions 176*i*0*k* and 1770*jk*, when A*k* is reserved when *k* = 1 through 7.
- If the system is not in bidirectional memory mode, or if an uncompleted 076 instruction exists, then instruction 176 holds issue when port A or B is busy with a write reference, and instruction 177 holds issue when port A or B is busy.
- The VIQ full.

VIR Hold Issue Conditions

These instructions hold issue at the VIR under any of the following conditions:

- V*i* (and V*k* for 176*i*1*k*) register is reserved for a 176 instruction and chaining or tailgating is not permitted.
- V*j* (and V*k* for 1771*ijk*) register is reserved for a 177 instruction and chaining is not permitted.
- Instruction 070*ij*6, 073, 076, or 077 issued from the VIR the previous CP.

Execution Time

The execution time for vector instructions issued directly from CIP to the vector load and store control section through the VIR is 2 CPs longer than the execution time for the instruction that is waiting issue in the VIR. The issue times for instructions 176 and 177 from the VIR are as follows:

- For instruction 176*i*0*k*:
  - The instruction issues in 1 CP.
  - The V*i* register is ready in a minimum of (VL/2) + 176 CPs if memory is available and a minimum of (VL/2) +22 CPs minimum if the load data is in the cache.
  - Port A or B is busy (VL/2) + 10 CPs minimum.
- For instruction 1770*jk*:
  - The instruction issues in 1 CP.
  - The V*j* register is ready in (VL/2) + 2 CPs minimum.
  - Port A or B is busy (VL/2) + 13 CPs minimum.
- For instruction 176*i*1*k*:
  - The instruction issues in 1 CP.
  - The V*i* register is ready in (VL/2) + 172 CPs minimum, if memory is available and (VL/2) + 25 CPs minimum if load data is available in cache.
  - The V*k* register is ready in (VL/2) + 2 CPs minimum.
  - Port A or B is busy (VL/2) +13 CPs minimum.
- For instruction 1771*jk*:
  - The instruction issues in 1 CP.
  - The V*j* and V*k* registers are ready in (VL/2) + 2 CPs, if data is available.
  - Port A or B is busy (VL/2) + 16 CPs minimum.

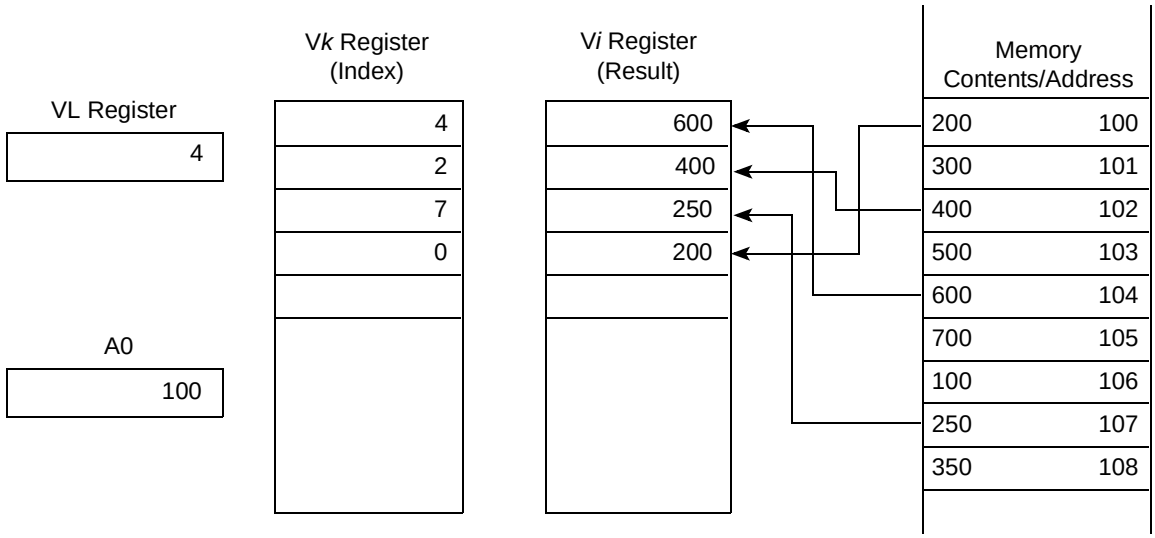
Description

Instructions 176 and 177 transfer blocks of data between V registers and memory. Instruction 176 reads data from memory to elements of register V*i*. Instruction 177 stores data from elements of register V*j* to memory. The contents of the VL register determine the number of elements that are transferred. Tailgating is possible with the 176 instruction, and chaining is possible with the 177 instruction.

Instructions 176*i*0*k* and 176*i*00 load words into elements of register V*i* directly from memory. A0 contains the starting memory address; it is 32 bits wide. This address is incremented by the contents of register A*k* (which is 32 bits wide) for each word that is transmitted. The contents of A*k* can be positive or negative, which allows both forward and backward streams of references. If the *k* designator is 0, or if 1 replaces A*k* in the operand field of the instruction, the address is increased by 1 for each word of data.

Instruction 176*i*1*k* gathers words from nonsequential memory locations and loads them into sequential elements of register V*i*. Elements of vector register V*k* and register A0 generate the nonsequential memory address. The low-order bits of each element of V*k* contain a signed integer, which is added to the contents of A0 to obtain the memory address. Figure 59 shows an example of the 176*i*1*k* instruction.

Figure 59. Gather Instruction Example



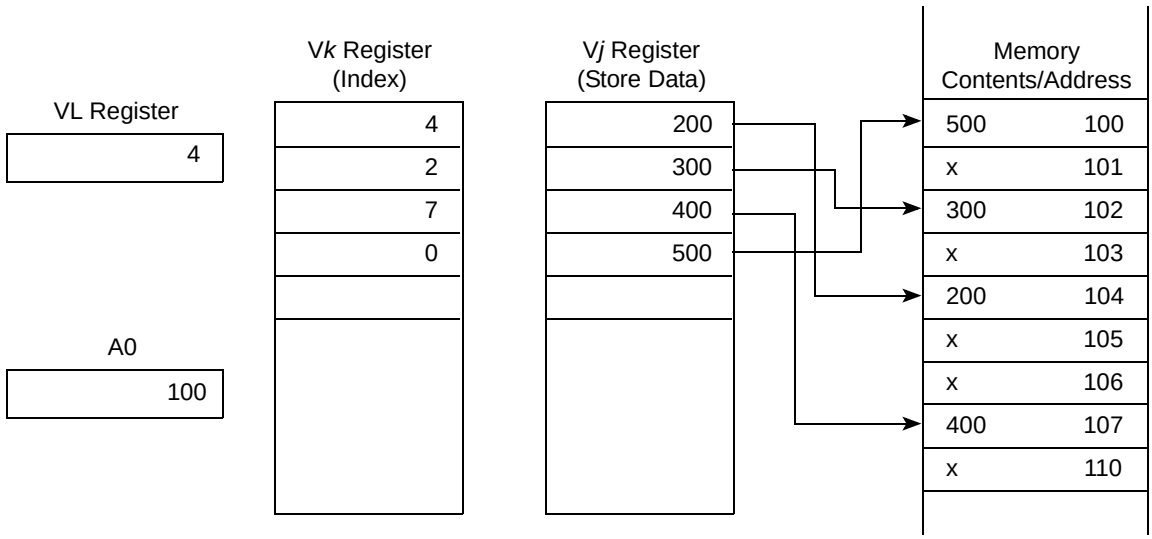
In Figure 59, the VL register is set to 4, which results in a transfer of 4 elements. The 176*i*1*k* instruction adds the contents of A0 to the contents of each element of register V*k* to form a memory address. The contents of that address are then loaded into the V*i* register. Because A0 = 100 and V*k* element

0 = 4, the content of address 104 is loaded into V*i* element 0. Similarly, A0 + V*k* element 1 = 102, and the content of memory location 102 is loaded into V*i* element 1. This process continues until the number of elements that are transferred equals the VL count.

Instructions 1770*jk* and 1770*j*0 store words from elements of register V*j* directly into memory. A0 contains the starting memory address. This address is incremented by the contents of register A*k* for each word that is transmitted. The contents of A*k* can be positive or negative, allowing both forward and backward streams of references. If the *k* designator is 0, or if 1 replaces A*k* in the result field of the instruction, the address is increased by 1 (consecutive locations of memory).

Instruction 1771*jk* scatters words from elements of register V*j* to nonsequential memory locations. V*k* elements and register V*k* and A0 generate the nonsequential memory address. The low-order bits of each element of V*k* contain a signed integer, which is added to the contents of A0 to obtain the memory address. Figure 60 shows an example of the 1771*jk* instruction.

Figure 60. Scatter Instruction Example



In Figure 60, the VL register is set to 4, which results in a transfer of 4 elements. The 1771*jk* instruction adds the contents of A0 to the contents of each element of register V*k* to generate a memory address. An element of V*j* is stored at the resulting memory address. Because A0 = 100 and V*k* element 0 = 4, the content of V*j* element 0 is stored in address 104. Similarly, A0 + V*k* element 1 = 102, and the content of V*j* element 1 is stored in memory location 102. This process continues until the number of elements that are transferred equals the VL count.

Figure 59 and Figure 60 describe data being loaded from, and stored into memory. Cache will also be used if enabled in the exchange package. Data is read from cache if it is available there. Data is stored into both cache and memory (write through cache).

The 176 instruction may execute with, or without chaining or tailgating (chaining for the 177 instruction). The times in CPs listed for these instructions is without chaining or tailgating. Data times with cache and memory increase by 5 CPs when chaining or tailgating occurs. However, these instructions may begin execution many CPs earlier.

Note: Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

Appendix A - Block Transfer Engine and Translate Look-Aside Buffer

This appendix includes a description of the software protocol for using the block transfer engine (BTE) and translate look-aside buffer (TLB) for the SV1ex memory module. It also contains detailed information on memory mapped register (MMR) addresses and data fields and procedures for writing and using the BTE and TLB.

BTE

The BTE transfers data between memory locations within a section in 64-word blocks, 4 words per subsection. This means that when you start up the BTEs in all eight memory sections in a system with a block length of 1, the BTEs will transfer 512 words. Therefore, data must be aligned on 512-word boundaries.

The BTE uses four MMRs per section. MMR0 is the source address, MMR1 is the destination address, MMR2 is the block length, and MMR3 is the enable/disable register. MMR3 is also the TLB enable/disable register. The BTE MMRs for each section of memory must be loaded separately.

As soon as the block length register is loaded, the BTE starts up and uses the last starting address and last destination address it received. Therefore, if multiple transfers are desired using the same source address and multiple destination addresses, the source address need not be reloaded. However, if multiple source or destination addresses are loaded without a block length between them, the BTE retains only the last source and destination addresses.

An individual BTE transfers data at a rate of 2.13 gigawords per second. If the BTEs in each section of memory are started at the same time, the system BTE transfer rate is 17 gigawords per second in an 8 x 8 system and 8.5 gigawords per second in a 4 x 4 system.

The following sections describe the registers of the MMRs.

MMR0 - Source Address

Address Field

- 31:18 - Must be all 1’s
- 17:15 - Must be set to the processor module number
- 14:12 - Binary 100
- 11:5 - Must be all 0’s
- 4:3 - Binary 00

Data Field

- 26:0 - Source address bits 35:9

MMR1 - Destination Address

Address Field

- 31:18 - Must be all 1’s
- 17:15 - Must be set to the processor module number
- 14:12 - Binary 100
- 11:5 - Must be all 0’s
- 4:3 - Binary 01

Data Field

- 26:0 - Destination address bits 35:9

MMR2 - Block Length

Address Field

- 31:18 - Must be all 1’s
- 17:15 - Must be set to the processor module number
- 14:12 - Binary 100
- 11:5 - Must be all 0’s
- 4:3 - Binary 10

Data Field

- 26:0 - Block length

MMR3 - BTE/TLB Enable/Disable

Address Field

- 31:18 - Must be all 1’s
- 17:15 - Must be set to the processor module number
- 14:12 - Binary 100
- 11:5 - Must be all 0’s

- 4:3 - Binary 11

Data Field

- 7:4 - All 1’s to enable BTE, all 0’s to disable BTE
- 3:0 - All 1’s to enable TLB, all 0’s to disable TLB

TLB

The TLB is used to extend the addressing capability of the SV1ex system beyond the 32 bits that come across the backplane. The TLB allows users to address 512 pages of 8 megawords of memory (total of 4 gigawords). These pages can reside anywhere in a 64-gigaword system image.

Address bits 31:23 are used to address data into the TLB (when enabled) and the TLB then returns 13 bits that are used as address bits 35:23.

The TLB is enabled/disabled for a processor module; each source (4 processors and an I/O) has its own unique TLB. If you want to enable the TLB for I/O operations and want the CPUs to continue to access the lower 4 gigawords of memory, you must write the CPUs’ TLB entries with the “unity” pattern. This is simply a one-for-one match for bits 31:23 and bits 35:32 cleared.

You enable/disable the TLB by writing MMR3 (described in the BTE section).

When you write the TLB, the system must be in noninterruptible monitor mode; otherwise interrupts may occur during the write and all sections may not be mapped to the same memory range.

TLB entry 0 for each CPU and I/O is unwritable and always points to the lower 8 megawords of memory. This ensures that on interrupts the exchange package information comes from the correct source.

The upper 262,144 words of TLB entry 511 for each CPU and I/O are unusable and will be interpreted as an MMR address (because bits 31:18 are all 1’s).

Writing the TLB

On power-up or after a system reset, the TLB entries in each section of memory must be written for every processor and I/O in the system. Any processor, or the I/O on a processor module, can write the TLBs for all four processors and the I/O for that particular module. However, a processor cannot write the TLBs for any processors or I/Os for a different processor module.

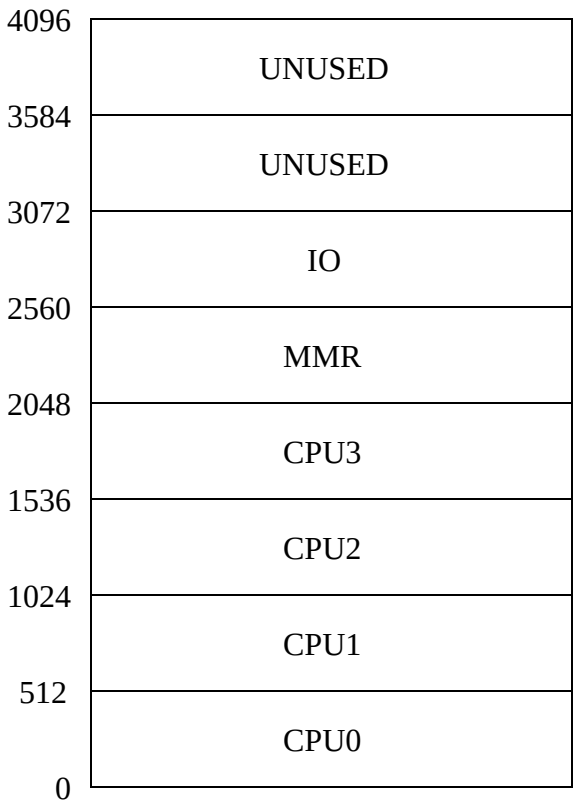
The writes are also passed into the SDRAM DIMMs to allow reading of the TLB.

Address Field

- 31:18 - Must be all 1’s
- 17:15 - Must be set to the processor module number
- 14:3 - Address into the TLB for this processor module

The lower 512 words are CPU 0’s TLB, the next 512 words are CPU 1’s TLB, the next 512 words are CPU 2’s TLB, the next 512 words are CPU 3’s TLB, the next 512 words are reserved for MMR space, the next 512 words are the I/O’s TLB, and the top 1024 words are unused. Refer to Figure 61.

Figure 61. TLB Address Fields



Data Field

- 12:0 - These bits are written into the TLB and are used as address bits 35:23 for this particular page.

Reading the TLB

When a TLB entry is written, the write is also passed on to SDRAM memory to create a “shadow copy” of the TLBs. This enables TLB reads to return the correct data. The read address is the same as the write address described earlier. The reads are treated as normal memory reads and are passed to main memory, which contains the “shadow copy” of the TLBs.

JTAG Interface

The JTAG USER1 register contains 2 bits that are used to permanently disable the BTE and TLB. If these bits are disabled and MMR3 is written to enable the BTE or TLB, they remain disabled. When Bit 0 of the USER1 register is set, it enables MMR3 to enable/disable the TLB. Bit 1 of the USER1 register does the same for the BTE. When bit 0 of the USER1 register is cleared, the TLB remains disabled no matter what the state of MMR3. When bit 1 of the USER1 register is cleared the BTE remains disabled no matter what the state of MMR3.

Power up/Reset Procedures

On power-up or after a system master clear the TLB and BTE are disabled. All entries must be written as described in the TLB section before enabling and using the TLB. However, after a system master clear the “shadow copy” of the TLBs that is held in main memory remains intact so this information can be read for system dump information; this must be done before writing the TLB.

Memory Clear Process

Because the SV1ex system can have very large memory sizes, it is not feasible to clear all of memory as you do on the SV1 by using the Jclr command. Use BTE as follows to speed up the memory clear procedure:

1. Clear an aligned 64-Kword block of memory.
2. Set the BTE source address to the beginning of this 64-Kword block.
3. Set the BTE destination address to the first word after this 64-Kword block.
4. Set the BTE block length to the total number of 512-word blocks in the system minus the 64-Kword block.

This procedure enables the BTE to clear the rest of memory.

Reader Comment Form

Title: System Programmer Reference (Cray SV1™ Series) **Number: 108-0245-003**

Your feedback on this publication will help us provide better documentation in the future. Please take a moment to answer the few questions below.

For what purpose did you primarily use this document?

_____Troubleshooting
_____Tutorial or introduction
_____Reference information
_____Classroom use
_____Other - please explain

_____

Using a scale from 1 (poor) to 10 (excellent), please rate this document on the following criteria and explain your ratings:

_____Accuracy _____
_____Organization _____
_____Readability _____
_____Physical qualities (binding, printing, page layout) _____
_____Amount of diagrams and photos _____
_____Quality of diagrams and photos _____

Completeness (Check one and explain your answer)

_____Too much information _____ Too little information _____Correct amount

_____
_____
_____

You may write additional comments in the space below. Mail your comments to the address below, fax them to us at +1 715 726 4991, or e-mail them to us at *fiona@cray.com.* When possible, please give specific page and paragraph references. We will respond to your comments in writing within 48 hours.

_____
_____
_____

NAME _____
JOB TITLE _____
E-MAIL ADDRESS _____
SITE/LOCATION _____
TELEPHONE _____
DATE _____

Technical Training and Documentation
900 Lowater Rd.P.O. Box 6000
Chippewa Falls, WI 54729
USA