SPERRY

PUBLICATIONS RELEASE

Integrated Scientific Processor System Processor and Storage Reference

UP-11006

This Library Memo announces the release of the SPERRY *Integrated Scientific Processor System, Processor and Storage, Reference,* UP-11006.

The SPERRY Integrated Scientific Processor System consists of a scientific processor subsystem merged with standard components of an 1100/90 system. The scientific processor attaches to the 1100/90 system through a scientific processor storage unit, which is substituted for the standard 1100/90 main storage units.

This manual describes the functional characteristics of the scientific processor subsystem components only, and is a companion manual with the *1100/90 Systems Processor and Storage Reference,* UP-9667.

This manual is intended primarily as a reference manual for experienced systems programmers and systems analysts. It includes the following information:

- scientific processor description
- interrupt handling information
- instruction conflict classes and conflict types
- scientific processor instructions
- scientific storage unit description
- multiple unit adapter description
- instruction summary
- scientific storage and address interleave configurations
- glossary
- index

Order additional copies of this manual through your Sperry representative.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists AC, BZ, CZ (less DE, GZ, HA), MZ, MBR, MBSU, MU1G	Mailing Lists DE, GZ, HA, 82, MBW, MU1H, MU1Y (193 pages)	Library Memo for UP-11006 RELEASE DATE: April 1986

Integrated Scientific Processor System

Processor and Storage

Reference

SPERRY

UP-11006

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry representative.

Sperry reserves the right to modify or revise the content of this document. No contractual obligation by Sperry regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry.

Page Status Summary

Section	Pages	Update
Cover/Disclaimer		
User Comment Form		
PSS	1	
Preface	1 – 2	
Contents	1 – 7	
Section 1	1 – 22	
Section 2	1 – 55	
Section 3	1 – 14	
Section 4	1 – 39	
Section 5	1 – 14	
Section 6	1 – 4	
Appendix A	1 – 10	
Appendix B	1 – 7	
Glossary	1 – 4	
Index	1 – 10	

Section	Pages	Update

Preface

Intent of This Manual

This SPERRY Integrated Scientific Processor System Processor and Storage Reference manual describes the SPERRY Integrated Scientific Processor Subsystem components only. It is a companion manual with the SPERRY *1100/90 Systems Processor and Storage Reference,* UP–9667. These manuals together fully describe a SPERRY Integrated Scientific Processor System.

This manual is intended primarily as a reference manual for experienced systems programmers and systems analysts.

Refer to the Glossary for a definition of terms used in this manual.

Manual Organization

This manual contains the following sections.

- **Section 1. Introduction**

  Describes the scientific processor components, features, configurations, and characteristics.

- **Section 2. Scientific Processor**

  Describes the scientific processor functions; including the scalar and vector processors.

- **Section 3. Operations**

  Describes activity switching, interrupt handling, and instruction conflict operations.

- **Section 4. Scientific Processor Instructions**

  Describes the instruction word formats and the instruction set.

- **Section 5. Scientific Processor Storage**

  Describes general storage operations including characteristics and modes of operation.

- **Section 6. Multiple Unit Adapter**

  Describes the relation of the multiple unit adapter with the scientific processor system.

- **Appendix A. Instruction Summary**

  Provides an alphabetically arranged summary of the instructions.

- **Appendix B. Storage Configurations and Address Interleave**

  Provides scientific storage and address interleave configurations.

- **Glossary**

  Defines key terms used in this manual.

- **Index**

  Lists key terms used in the manual with corresponding page number references.

Related SPERRY Manuals

Throughout this manual, when you are referred to another manual, you should use the version that applies to the software level in use at your site. If you need more information, the following Sperry manuals may be useful.

- *Integrated Scientific Processor System, System Description,* UP-11547
- *1100/90 Systems, System Description,* UP-9288
- *1100/90 Systems Processor and Storage, Reference,* UP-9667
- *1100/90 Systems, System Support Processor (SSP), Level 4R6, Operator Guide,* UP-10096.2
- *Series 1100 Executive System Operator Reference,* UP-7928
- *Series 1100 Hardware/Software Mini-Reference,* UP-7824
- *Series 1100 Meta-Assembler for the Scientific Processor, MASP Reference,* UP-10985

Copies of these manuals may be ordered through your Sperry representative.

Contents

Figures

Tables

1. Introduction

This manual provides detailed descriptions of the SPERRY Integrated Scientific Processor Subsystem components and characteristics only. Similar detailed information for the SPERRY 1100/90 system is provided in a separate reference manual. (See 1100/90 Systems Processor and Storage Reference, *UP-9667.)*

The integrated scientific processor (scientific processor) system consists of standard 1100/90 system hardware components plus the scientific processor subsystem hardware components (Figure 1-1).

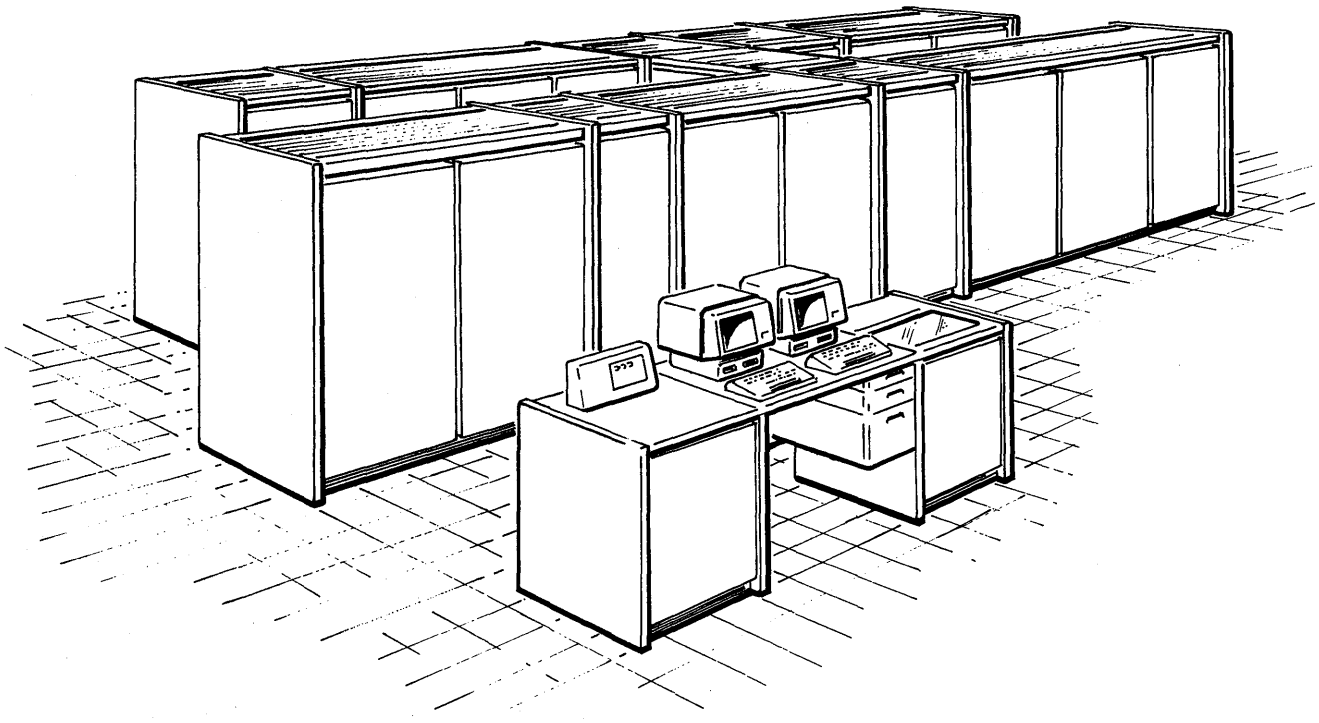


Figure 1-1. Integrated Scientific Processor System

1.1. Hardware Components

Figure 1-2 shows a simplified block diagram of the scientific processor system. The system configuration includes the scientific processor subsystem components and the 1100/90 system components.

The scientific processor system includes the following scientific processor subsystem components and standard 1100/90 system hardware components.

Scientific Processor Subsystem Components

- Integrated Scientific Processor (ISP)
- Scientific Processor Storage Unit (SPSU)
- Multiple Unit Adapter (MUA), optional
- Instruction Processor Cooling Unit (IPCU)

1100/90 System Components

- Instruction Processor (IP)
- Input/Output Processor (IOP)
- System Support Processor (SSP)
- Instruction Processor Cooling Unit (IPCU)
- System Clock Unit (SCU)
- Console
- Motor Alternator

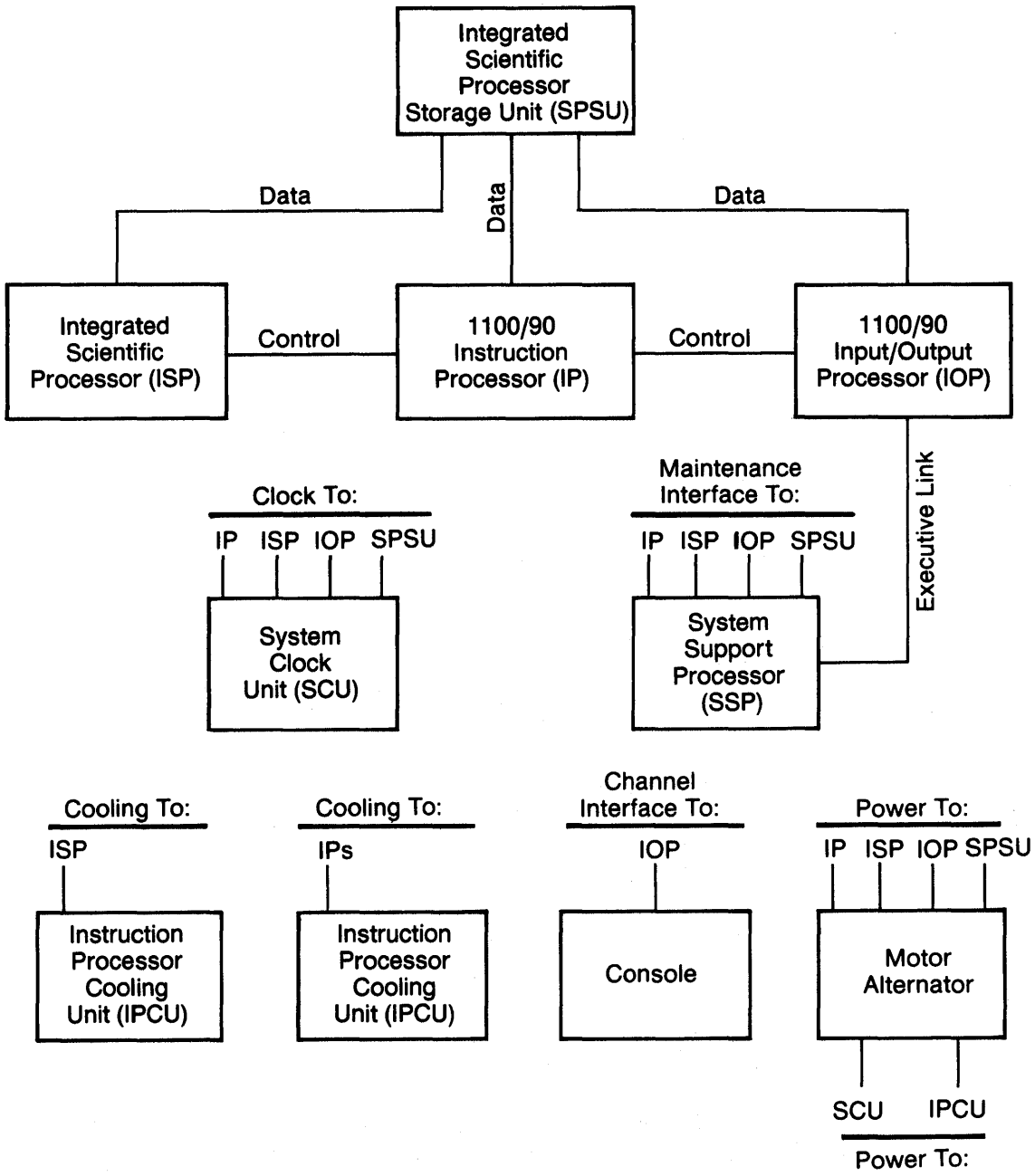


Figure 1-2. Scientific Processor System Simplified Block Diagram

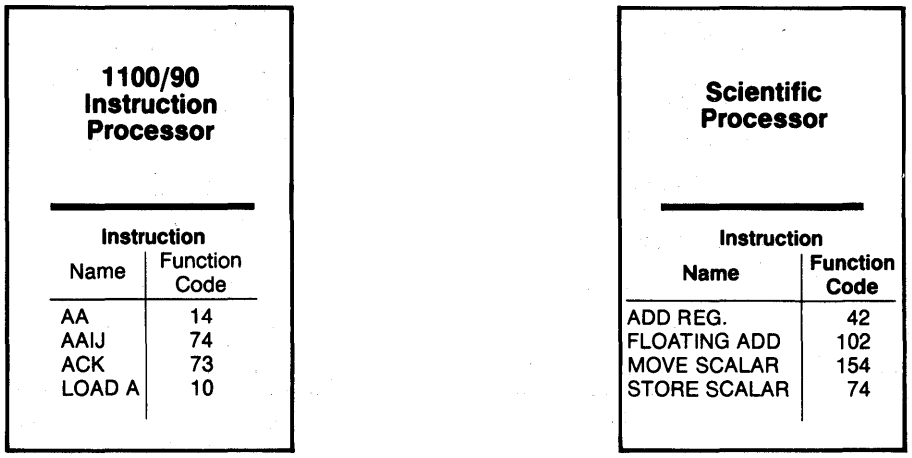
1.2. Features

The scientific processor consists of a scalar processor module and a vector processor module which support scalar and vector high-speed processing. The scalar module provides the overall scientific processor instruction execution control and performs one operation per instruction type (an example of this is one Add, one Multiply, etc.); the vector module performs the multiple operations per instruction type operations (an example of this is up to 64 Adds, up to 64 Multiplies, etc.).

Other features include:

1. Source language compatibility is provided at the FORTRAN level. Data representation is identical to that of the Series 1100 systems.

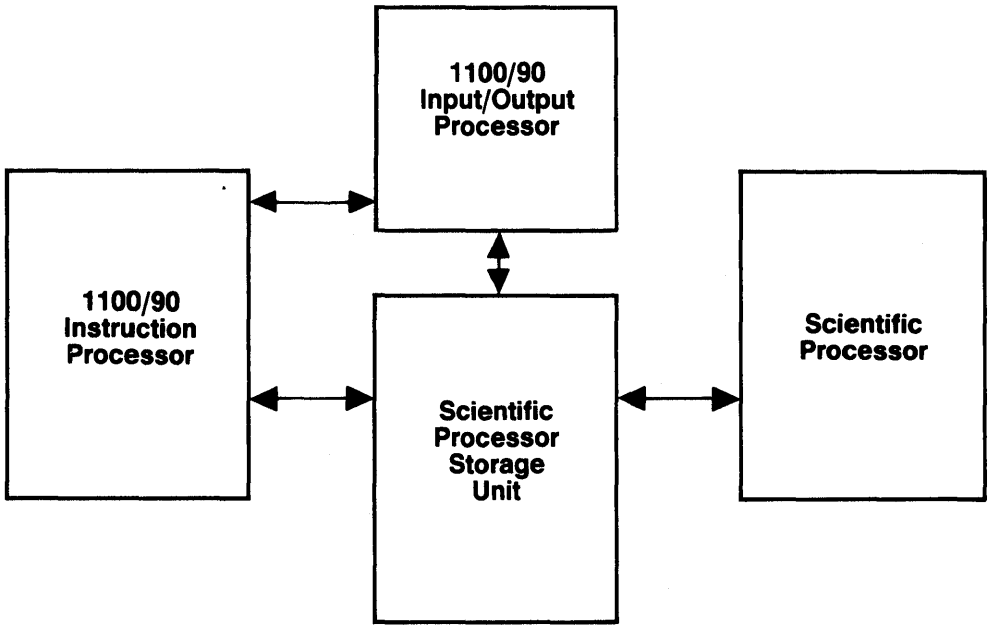
2. The scientific processor has its own unique instruction set tailored for scientific computing.



Unique Instruction Set

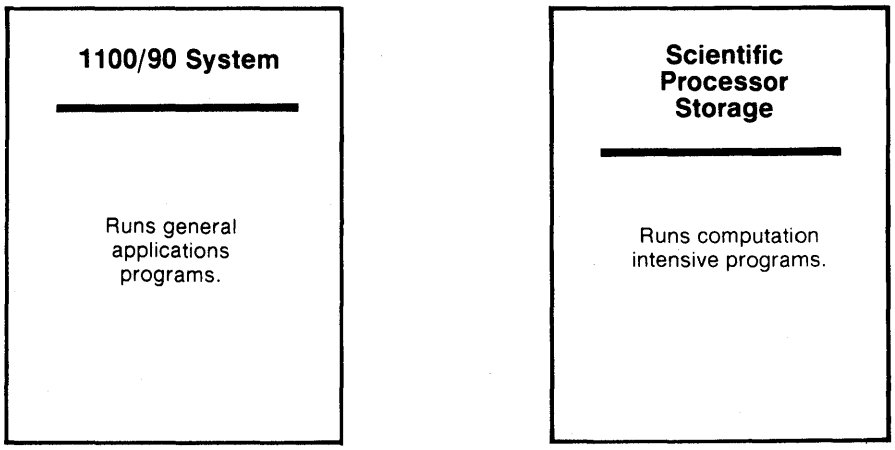
3. The Series 1100 Executive system, running on the instruction processor, manages all the tasks. The scientific processor executes user code only. No system software runs on the scientific processor. Its entire computing power is devoted to the computations of the user's application.

4. A scientific storage unit is directly addressable by the instruction processor, input/output processor, and the scientific processor. File or data transfers between the instruction processor and the scientific processor is therefore not necessary.



Scientific Processor Shared Storage

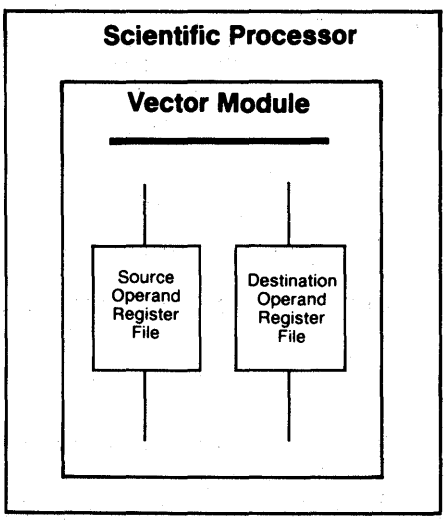
5. Because the scientific processor is totally integrated within a standard 1100/90 system, movement of data between systems is eliminated. You can therefore run your computational-intensive programs on the scientific processor, thereby freeing the general-purpose processor to address other applications.



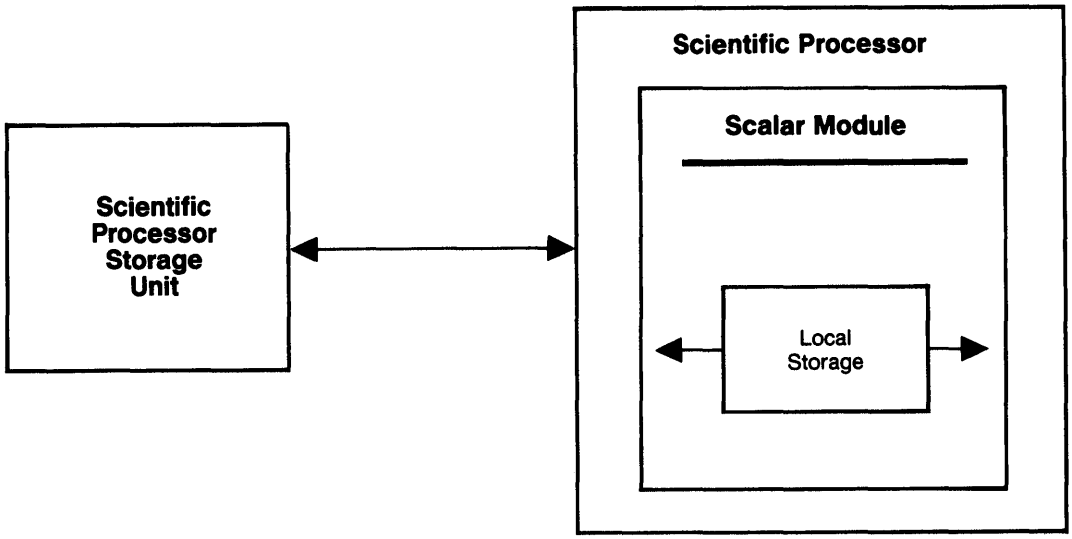
6. Instructions are available for computations that involve single and double-precision integers or floating-point numbers in both scalar and vector operations.

7. Repetitive operations are optimized using registers and control structures for efficient handling of the nested loop processing of a vector FORTRAN program. Primary emphasis is given to optimizing the innermost loop.

8. The vector processor has register files for both the source and destination operands, which minimizes the storage bandwidth problem of high-speed processing.



9. The scalar module uses local storage for fast access to frequently used scalar variables in a program.



1.3. Scientific Processor

The scientific processor uses parallel processing techniques to perform both scalar (singular) and vector (multiple) operations simultaneously at very high speeds. It uses a unique instruction set to perform floating-point, integer, and loop control instructions to control dedicated, independent pipelines to perform its calculations.

The scientific processor unit of work is referred to as an activity. Each activity is under control of the instruction processor and is explicitly dispatched for execution on the scientific processor.

The scientific processor has no privileged mode or privileged instructions. System services, such as mass storage file retrieval and storage or print output, are provided by the executive system running on the instruction processor in response to a scientific processor interrupt.

The following paragraphs explain some general concepts and characteristics of the scientific processor.

The scientific processor has a peak performance of 133 million single-precision floating-point operations per second and 67 million double-precision floating-point operations per second.

Although the scientific processor is a single entity performing its calculations in a highly overlapped fashion from a single instruction stream, it is in some cases useful to describe specific functions in terms of its two major processor modules: the scalar module and the vector module.

Instructions

The scientific processing instructions include: scalar, vector, and control instructions.

- The scalar instructions include arithmetic, logical, compare, convert, shift, and move instructions.
- The vector instructions include arithmetic, logical, compare, convert, reduction, and move instructions.
- The control instructions include loop control, jump, and state instructions.

(For a description of each instruction, see Section 4.)

Data and Addressing Formats

Data and addressing formats are identical to the Series 1100 data formats. Because the scientific processor is tightly coupled to the 1100/90 system, it uses addressing methods consistent with the instruction processor.

All user program and data addresses, on the scientific processor, are 36-bit virtual addresses. The information to translate a virtual address to a real storage address is placed in an area of the scientific processor control block referred to as the activity segment table (AST) by the instruction processor prior to activating the scientific processor.

The virtual address format is the same as the instruction processor extended mode, consisting of three fields: Level, Bank Descriptor Index, and Offset. However, the mechanism for translating virtual addresses to real addresses is unique to the scientific processor. Segment protection comparable to that used in the instruction processor is ensured by the translation mechanism.

Initialization

The scientific processor must have a real storage address that points to a dispatching mailbox to become operational. This address is loaded into the scientific processor internal hardware by the system support processor (SSP). When the mailbox has been loaded and the scientific processor is made operational by the SSP, it is initially dormant until it receives a universal processor interface (UPI) interrupt. In response to the interrupt it commences the activity switch algorithm by accessing the mailbox which in turn acquires the scientific processor control block associated with the activity to be initiated.

Instruction Execution

The scientific processor is a single instruction multiple data type, meaning it possesses a single thread of control, though each instruction may specify several operations affecting many data elements. The thread of execution is controlled by the contents of the Program Address Register. This register contains the 36-bit virtual address of the instruction to be executed next and its value is incremented by one following each non-jump instruction initiated.

Programmable Registers

The programmable registers consist of a Vector register set, a General register set, a Mask register, a State register set, and a Loop control register set.

Vector Register Set

There are 16 vector registers identified as V0–V15 respectively. Each register has space for 64 36–bit words. It may be formatted as 64 single–word elements, or as 32 double–word elements. Each register is understood to contain the elements of a single vector or array.

General Register Set

There are 16 General (G) registers of two words each. They are used as general accumulators for single–precision or double–precision scalar data operands and for holding virtual address and stride information for addressing operands in storage. For single–precision data operations, only the leftmost word of the pair participates, leaving the rightmost word unaffected. For addressing, the leftmost word contains the 36–bit virtual address, and the rightmost word provides the stride value for those vector accesses that require variable strides.

Mask Register

The Mask register is a 64–bit or 32–bit register (one–bit file) that has one bit position corresponding to each element of a vector file. The mask value is generated and placed into the Mask register by execution of a Compare Vector instruction, or it is loaded from a G–register. The mask value is used to conditionally control the execution of individual elements within vector operations. For example, a conditional vector addition adds only those pairs of elements whose corresponding Mask register bit is a 1.

Loop Control Register Set

The Loop Control registers hold parameters that determine iteration and indexing of program loops. There are eight 45–bit Vector Loop registers for controlling the iteration over strips of a vector, and there are eight 14–bit Element Loop registers for controlling the indexing over elements of a strip. When these registers are referenced within main storage as a part of activity initiation, activity termination, or the Store Loop Control Register instruction, they are packed side–by–side into eight double words.

The Loop Control register set also includes two 3–bit pointer registers, called Current Vector Loop Pointer (CVLP) and Current Element Loop Pointer (CELP). These registers are used to determine which of the Vector Loop registers and which of the Element Loop registers are used.

State Register Set

There are sixteen 36-bit State registers (0–15) that contain program visible states relating to internal interrupts, condition codes, etc. The Scalar Move instruction allows program access and control of the State registers. The word formats for the State registers are shown in 2.3.8.

Register Save Area

A real address in the scientific processor storage for the register save area is pointed to by word 5 of the scientific processor control block. It is used to initialize or maintain the following register contents when an activity is switched on to or out of the scientific processor.

The register save area format is:

Words	Contents
0–1023	Vector registers 0–15
1024–1055	General registers 0–15
1056–1071	Vector Loop and Element Loop registers
1072–1087	State registers
1088–1119	Jump history file

NOTE: The Mask register is not stored separately in the register save area but is mapped into the State registers.

The register save area must be located on a 16-word real address boundary. Thus, if bits 32–35 of control block word 5 are not 0's, an RSA Boundary Error (type 21) external interrupt is generated immediately.

Universal Processor Interface (UPI)

The scientific processor uses the UPI to communicate with instruction processors only. It does not communicate with input/output processors or other scientific processors. The scientific processor receives directed interrupts and transmits broadcast interrupts through the UPI. It uses the UPI mechanism strictly as a signalling interface rather than a message interface.

Chaining

By having distinct hardware for additions, multiplications, storage references, and scalar operations, processor throughput is enhanced by its ability to overlap the execution of consecutive instructions. Also, consecutive vector instructions can be chained together. Each element produced by one vector operation can be used as a source to the following operation (without waiting for the first operation to complete all elements) by simply coding the same vector register as the destination of the first operation and the source of the second operation.

Instrumentation

Instrumentation includes a quantum timer, an internal interval timer, a jump history file, and breakpoint interrupt functions.

Quantum Timer

The quantum timer limits the execution time of an activity in the scientific processor. Each scientific processor instruction issued causes the quantum timer value to be decreased by a specific amount. The amount of time varies depending on the instruction. When the value crosses zero, an external interrupt is caused.

The initial activity quantum timer value is loaded and controlled by the operating system and provided to the scientific processor through word 15 of the scientific processor control block.

The quantum timer granule is one cycle of the current scientific processor clock rate. Each instruction has an internally stored base execution time that is deducted from the quantum timer. Also, instructions have to account for these additional deductions:

- vector element count
- scientific storage stride value
- time for taking a jump
- double precision operations
- slow mode operation

The quantum timer is updated when the primary instruction holding register is loaded.

The quantum timer is not charged for page misses and conflicts at the scientific processor storage interface, also interrupt sequence time is not charged against the quantum timer.

NOTE: There are no scientific processor programmable instructions that can access the quantum timer.

Internal Interval Timer

The internal interval timer is located in word 15 of the State registers. It is available for general use and it can be read and written using standard scientific processor instructions. The interval timer counts machine cycles, even if storage conflicts occur. When it crosses zero, an internal interrupt is caused and the decrementing continues. By spacing these interrupts appropriately, and simply noting the return address each time one occurs, a good statistical evaluation is obtained of where a program spends its time.

Jump History File

The jump history file contains the virtual address of the 32 most recent jumps internally executed by an activity. This provides useful information for debugging purposes. A new entry is written to this jump history file each time a jump is taken. Entries are placed into succeeding higher locations, wrapping around from location 31 to location 0. Entries are stored only for jumps, not for interrupts. No disable of this feature is provided.

The file pointer is contained in word 11 of the scientific processor control block. It points to the next available location for storing an entry. Pointer wrap-around and the consequential overwriting of previous entries occurs without notice, the effect being simply to preserve the most recent 32 values. When an activity is switched off in the scientific processor, its jump history file is stored in the register save area.

Breakpoint Functions

Word 14 of the State register set contains a breakpoint virtual address. There are three types of breakpoint interrupts: Instruction Breakpoint Compare, Read Data Breakpoint Compare, and Write Data Breakpoint Compare. They are identified by internal interrupt indicator bits 1, 11, and 12 respectively, and independently enabled by appropriate bits in the Internal Interrupt Control Mask register (State register 11). Refer to 3.2.4. All types share the same comparison address in State register 14.

The instruction breakpoint condition occurs when the breakpoint virtual address exactly matches the program address value for an instruction being executed. A data breakpoint condition occurs when there is a match with the virtual address on an operand being read from or written to main storage or local storage. The operand may be a scalar or a vector element. In case of double precision, the match may be with either word of the pair.

In all breakpoint cases the interrupt condition does not prevent completion of the instruction execution. Rather the condition is held pending until instruction completion before taking effect.

NOTE: This is different from the recognition of other interrupt types.

When a data breakpoint interrupt is caused by a vector storage instruction, the element index of the comparing element is stored as status in word 8, bits 30–35 of the State register. Because the breakpoint condition (instruction or data) is held pending until instruction completion, it is possible for another interrupt or even another breakpoint condition to arise before the instruction completes. In such cases the original breakpoint condition may be lost or the element index overwritten.

Program Faults

Program faults include: instruction decode faults, arithmetic faults, and vector register length overflow faults.

Instruction Decode Faults

When each instruction to be executed is decoded, the leftmost 10 bits are examined for any combination of bits not defined or marked reserved. If any of these combinations exist, an Undefined Instruction interrupt occurs. Undefined combinations of the remaining 26 bits in an otherwise valid instruction will not cause this internal interrupt, but instead produces unpredictable execution results. A synchronous Undefined Instruction interrupt is always clean, meaning that the undefined instruction does not alter any registers or storage. Thus the interrupt handler can essentially define the instruction by means of a software routine and expect consistent operation.

Arithmetic Faults

Arithmetic faults result from computational instructions, including reductions, and from type conversions. In general they indicate that the correct algebraic result value cannot be expressed in the specified data format. Possible responses to these fault conditions are described in 3.2.5. For a fault that is to be ignored, a substitute value is stored in the specified register. These values are given here. When a fault causes an asynchronous interrupt, the result is undefined. When a fault causes a synchronous interrupt, the destination is unaltered.

The possible arithmetic faults are described here. They do not distinguish between whether the fault condition arose from a scalar or a vector instruction.

Divide Fault – produced by a divide instruction when the divisor value is all 0's. If a divide fault condition exists all other arithmetic faults are blocked. If the divide fault is ignored (no interrupt taken) 0's are stored as the arithmetic result.

Integer Overflow Fault – produced by integer arithmetic operations and conversions to integer when the magnitude of the correct algebraic result exceeds $2^{35}-1$ (single–precision) or $2^{71}-1$ (double–precision). When the interrupt is ignored, the result stored is the rightmost 36 or 72 bits of the correct algebraic result, unless the specific instruction specification provided otherwise.

Characteristic Overflow Fault – produced by floating–point operations, including type conversions to floating point, when the magnitude of the correct result would require a characteristic value that is too large for the selected data type format. When the fault is ignored, the result stored is all 0's.

Characteristic Underflow Fault – produced by floating–point operations and conversions to floating point when the result is not exactly 0, but is too small to be represented in properly normalized form. When the interrupt is ignored, the stored result is all 0's.

Vector Register Length Overflow Fault

The vector registers each have 64 words and can therefore contain no more than 64 single–word or 32 double–word elements. Nevertheless there exists a variety of ways that a vector instruction can specify strips, element indices, or mask lengths in excess of these limits. Any attempt to do so, either as a source or a destination, causes this fault condition. When the interrupt is taken synchronously, processing is aborted prior to storing any destination values or detecting any arithmetic faults. In this case the Element index status bits (State register word 8, bits 30 through 35) are not defined. When the interrupt is ignored, operation is not defined.

Performance Monitoring

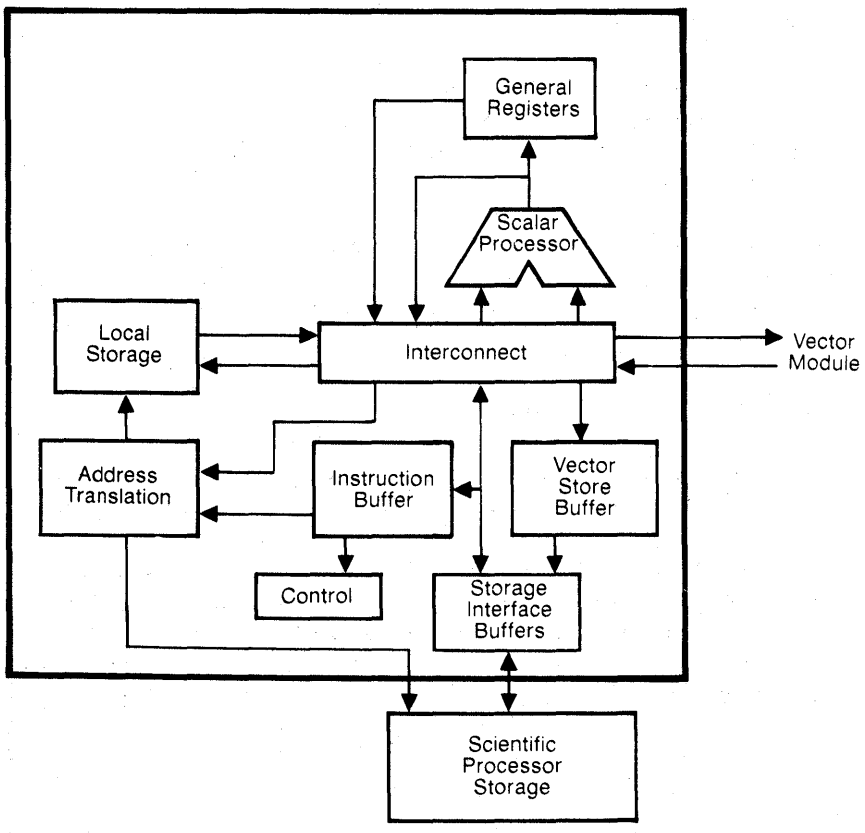
The scientific processor contains an interval timer and instruction breakpoint interrupt capability. The timer is available for performance monitoring since it can be read and written using scientific processor instructions. In conjunction with the instruction breakpoint capability, its use for performance monitoring is very flexible.

In addition, certain timing, control, status, and data signals are available for use by an external monitoring system capable of performing a necessary amount of logic at normal operating rates without affecting the electrical characteristics of the observed logic signals.

Scalar Module

The scalar module has the overall control function including: instruction fetch, decode, and issue. It also serves as an interface to the scientific storage unit.

Scalar Module



The scalar module includes the following functional components:

- instruction buffer
- instruction flow control
- local storage
- general and state register sets
- address translation and generation logic
- vector store buffers
- storage interface buffers
- scalar computational logic
- loop and interrupt control logic
- arithmetic logic unit
- floating-point exponent computational section
- multiplication section
- shifter and associated registers
- data paths and controls

When instructions are decoded, they are issued to the scalar or the vector modules or both for execution. Scalar instructions are placed into execution as rapidly as one every cycle, providing the execution facilities are available and no data conflicts exist.

Scalar processing proceeds in a highly overlapped manner with many instructions in process simultaneously. Arithmetic and control operations are initiated in sequential program order but may complete out of order because of the use of different paths through the computational logic.

Most data path and computational entities are two words wide such that the majority of single-precision or double-precision scalar operations are performed in the same amount of time.

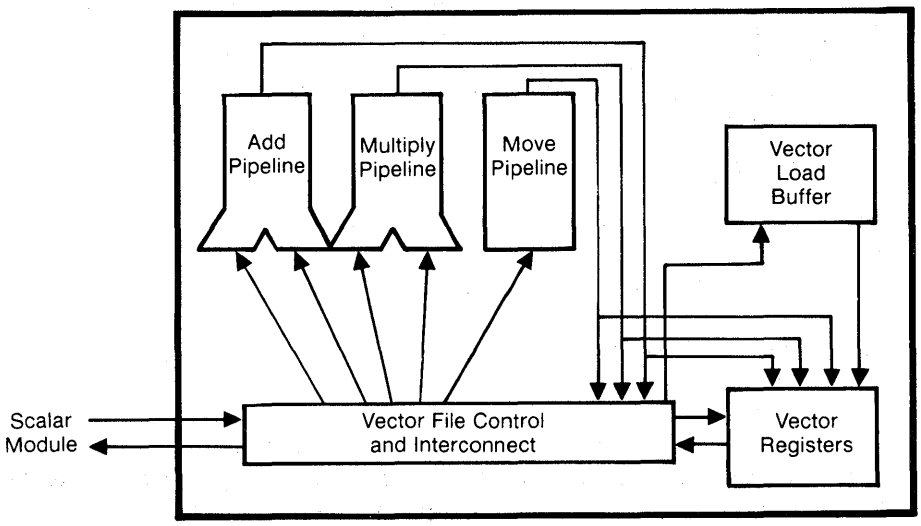
The vector store buffer serves to buffer performance differences between the vector module vector files and the scientific storage unit. It also provides non-consecutive scientific storage addressing. The vector store buffer is considered as a vector module facility.

Vector Module

The vector module receives instructions from the scalar processor and places them into execution as soon as facilities and operands are available. The main facilities are:

- vector register control and interconnect,
- the add pipeline,
- the multiply pipeline,
- the move pipeline,
- vector registers, and
- vector load buffer.

Vector Module



The add pipeline is used for add, subtract, logical, convert, reduction, compare, and shift operations.

The multiply pipeline is used for multiply, product reduction divide, and population count operations.

The move pipeline is used:

- for vector moves between vector registers,
- during single–precision to double–precision and double–precision to single–precision data type conversions,
- to compress vectors,
- to distribute vectors, and
- for calculating population parity.

The vector load buffer serves to buffer performance differences between scientific storage references and vector file accesses during vector file loads from storage. It also provides non-consecutive scientific storage addressing for vector data.

The vector module usually places instructions into execution in program sequential order. As vector instructions are received from the scalar module decoder, they are sent to the appropriate processing facility as soon as the processing facility and operands are available. If the facility is currently busy, the instruction must wait.

Normally the references to the Vector registers involve two words, either a double-precision element or two single-precision elements. The Vector registers are organized so that they can support up to eight such references each clock cycle. Also, the conflict detection logic permits use of an element written into a register at any time after it is written, thereby permitting multiple references to the same Vector register.

Unit Control Module

The unit control module has two functional subdivisions: a unit support controller and a power and cooling controller. These subdivisions provide power, cooling, and support interfaces for operator control and maintenance facilities.

1.4. Scientific Processor Storage Unit

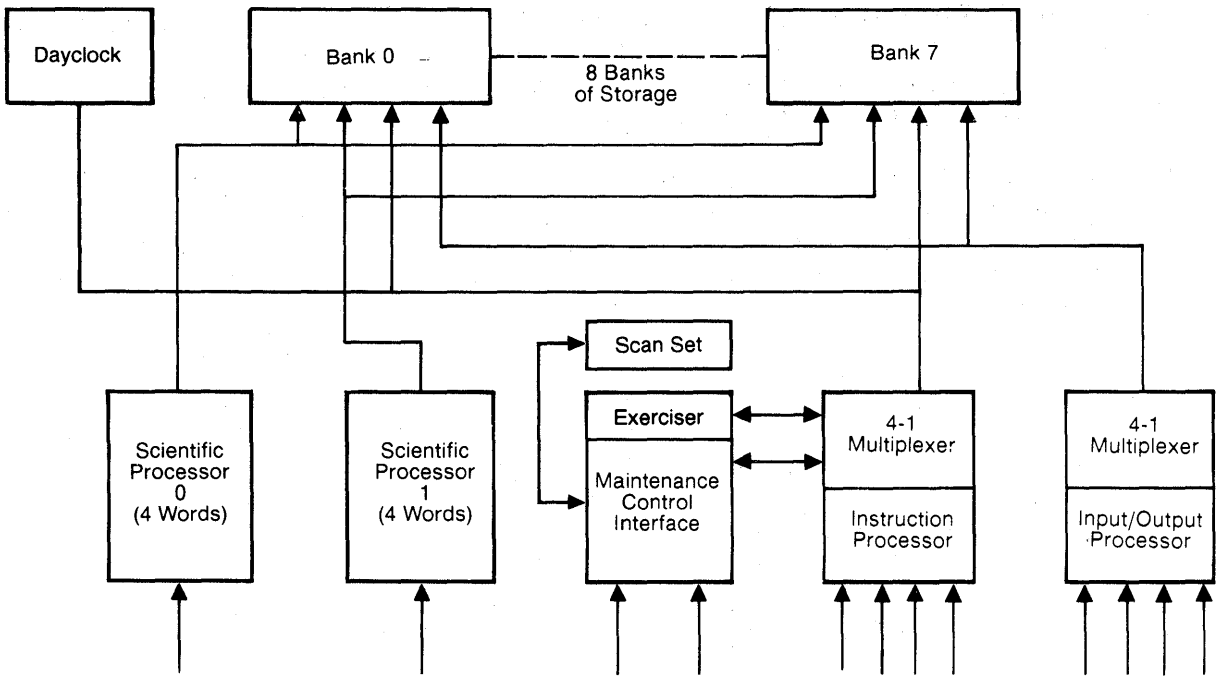
The scientific processor storage unit has eight storage banks. Each bank contains 524,288 (524K) words. Each word has 44 bits (36 data, 6 check, 1 check parity, and 1 data parity). One storage unit contains a maximum of 4,194,304 (4194K) words. Up to four scientific storage units can be used on a system.

When communicating with the scientific storage unit, the instruction processor can use one, two, or eight-word block transfers, the input/output processor uses one or two word block transfers, and the scientific processor always uses four-word block transfers.

Each scientific storage unit used replaces a possible main storage unit. The two types of storage units can be intermixed, but the scientific processor can only interface with the scientific storage. If two or more scientific storage units are to be accessed by a scientific processor, a multiple unit adapter is required.

The scientific storage unit IP and IOP ports provide identical functions for the instruction processor and input/output processor as the main storage unit does for the instruction processor. The functions include: block read or write operations (eight words per block, IP only); double-word read operations; and partial-, single-, or double-word write operations. The partial-word write capability is bit addressable for variable-length fields. The scientific processor ports provide four-word read operations and one-, two-, three-, and four-word write operations.

Scientific Processor Storage Unit



Requester Ports

The scientific storage unit can have up to ten requester ports that consists of:

- four instruction processor ports
- four input/output processor ports
- two scientific processor/multiple unit adapter ports

Interfaces

The scientific storage unit has these interfaces:

- instruction processor (up to four)
- input/output processor (up to four)
- scientific processor/multiple unit adapter (up to two)
- system support processor (two)
- system panel
- system clock
- other scientific processor storage or main storage units in the system

1.5. Multiple Unit Adapter

The multiple unit adapter is the interface between the scientific processor and one to four scientific processor storage units. Scientific processor systems with two or more scientific storage units require a multiple unit adapter for each scientific processor. A multiple unit adapter may also be used in a system with one scientific processor storage unit.

1.6. System Configurations

The scientific processor subsystem with the standard 1100/90 central complex equipment forms an integrated scientific processor system. A typical hardware configuration is shown in Figure 1–3.

A minimum system requires the following complement of system components:

- one instruction processor
- one input/output processor
- two instruction processor cooling units
- one scientific processor
- one scientific processor storage unit
- one system support processor
- one console with system panel
- one system clock
- two motor alternators

The basic system can be expanded by adding one, two, or three instruction processors; one, two, or three input/output processors; one scientific processor; one, two, or three scientific storage or main storage units for a total of 16 million words; and one or two multiple unit adapters. Additional cooling units and motor alternators may be required.

One scientific storage unit is required per system. When two scientific storage units are used, a multiple unit adapter is required. If a second scientific processor is used in a system using scientific storage units, a second multiple unit adapter is required.

Scientific storage units and main storage units can be mixed within a system. The scientific processor addresses only the scientific storage unit. The instruction processor and input/output processor address both types of storage units.

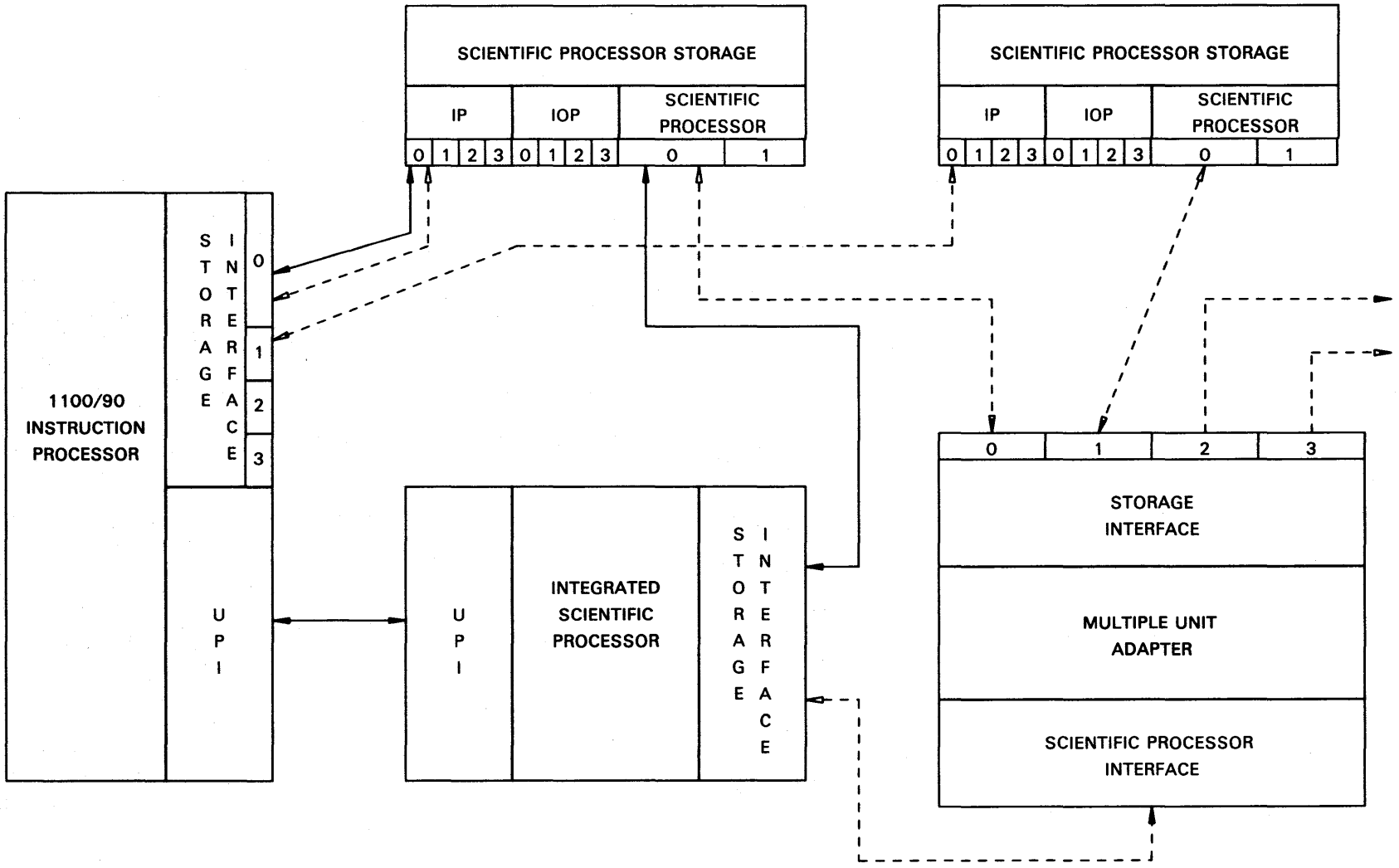
1.7. System Interfaces

The scientific processor system interfaces (Figure 1-4) provide functional paths to other units of the system. These interfaces are:

- scientific processor to scientific storage or to multiple unit adapter if two or more scientific storage units are to be accessed
- system clock unit interface
- universal processor interface (UPI) to each instruction processor

The UPI is the communications link with the instruction processor. The scientific processor uses the UPI as a signaling interface rather than as a message interface.

- maintenance and control interface to one or two system support processors
- instruction processor cooling unit interface to one or two IPCUs
- system panel interface to the system support processor through the maintenance and control interface



NOTE: Dashed lines indicate configuration for multiple scientific processor storages.

Figure 1–3. Integrated Scientific Processor System Typical Configuration

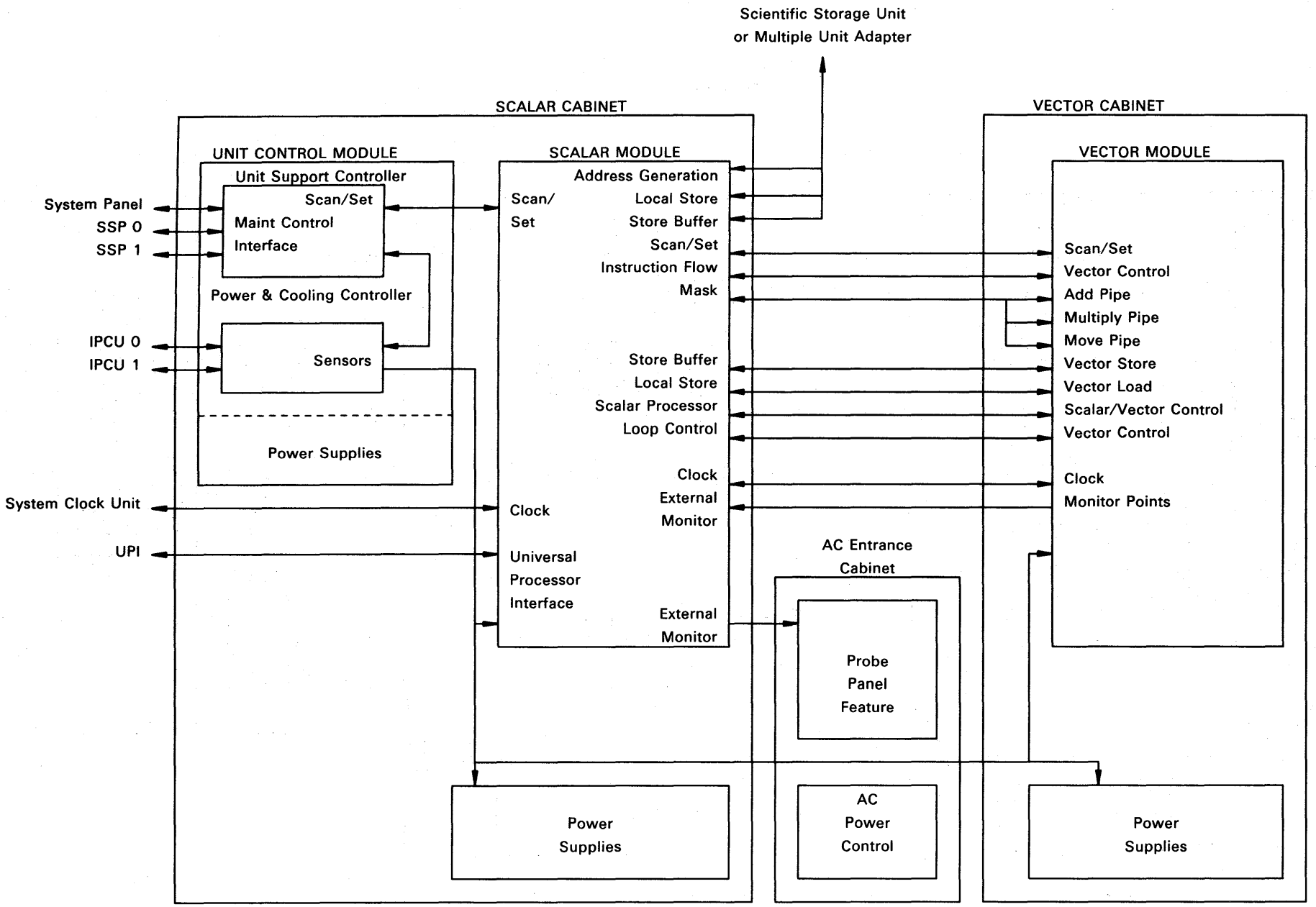


Figure 1–4. Integrated Scientific Processor Interfaces

2. Scientific Processor

This section describes the integrated scientific processor functional sections.

2.1. Functional Organization

The integrated scientific processor (scientific processor) is a free-standing unit that includes a scalar module, a vector module, a unit control module, and an AC entrance unit. The scientific processor cabinet is a three-section cabinet as shown in Figure 2-1.

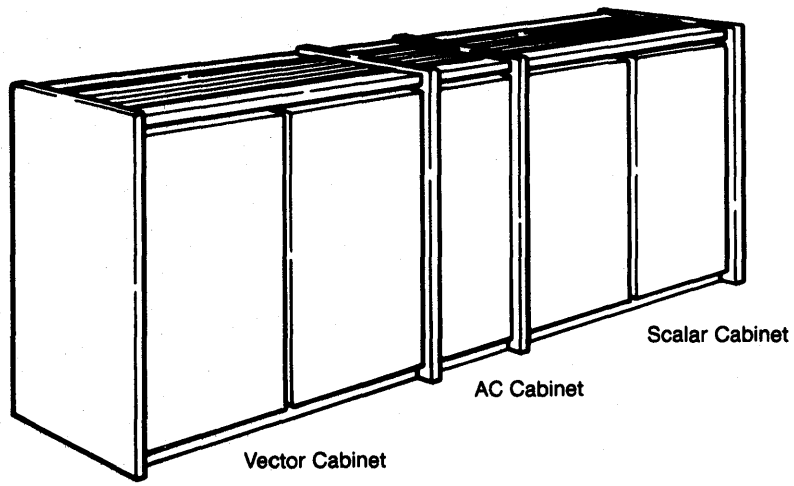


Figure 2-1. Integrated Scientific Processor Cabinet

The scientific processor's major functional sections are:

- a scalar module that performs scalar operations and has the overall scientific processor control function, including instruction fetch, decode, and issue
- a vector module to perform vector computations, the vector computations are performed on vector operands loaded in vector registers. Having been fetched from storage, a vector (or portion of) may participate in more than one computation without requeuing more storage references.
- a unit control module that provides power monitoring, cooling and support interfaces for operator control and maintenance facilities for both the scalar and vector modules. The unit control module also has several external interfaces that provide functional paths to other units of the system; and
- an AC entrance unit that contains the power circuit breaker assembly and probe panel for external performance monitoring.

In addition to the hardware components, the scientific processor uses various control structures to define the operation of the hardware and software within the scientific processor.

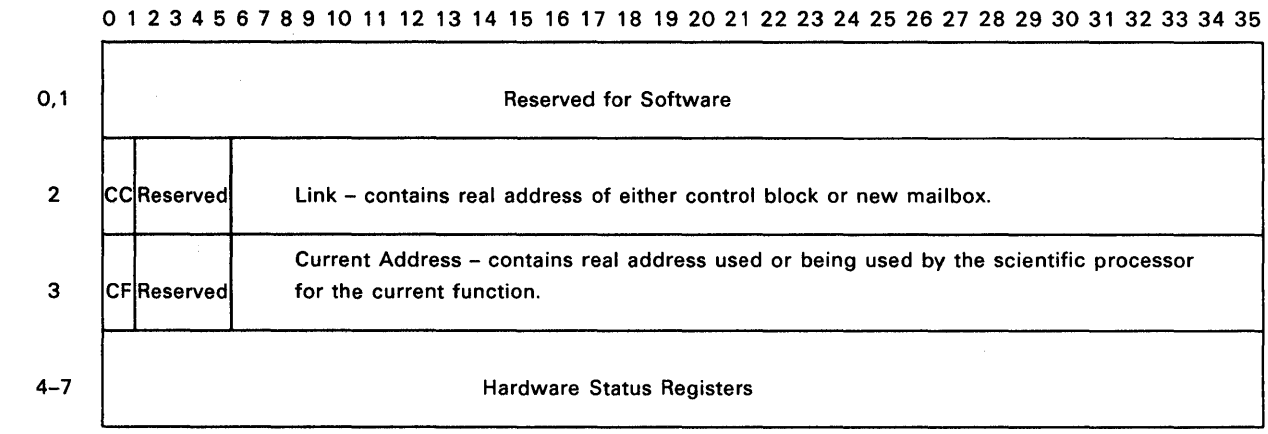
2.2. Control Structures

The unit of work that is scheduled for a scientific processor is called an activity. The scientific processor has neither a software control program nor a privileged mode of execution. Therefore scheduling of scientific processor activities is done by the instruction processor and each activity is explicitly dispatched. The instruction processor software control program and the scientific processor hardware therefore must be aware of the formats of activity control structures, in particular the mailbox, hardware status registers, and the scientific processor control block.

2.2.1. Mailbox

The mailbox control structure transfers limited initial and termination information between the instruction processor and the scientific processor. It points to the scientific processor control block for an activity at activity initiation. When the activity is terminated, the mailbox is loaded with the contents of the hardware status registers to report the activity termination status.

The real address used by the scientific processor to locate the mailbox is provided by the system support processor at initialization time. The real address of the mailbox must be on a 16-word boundary. The mailbox control structure has the following format:



where:

Word 2

Bits 0,1 CC – Control Code

The control code is entered into the mailbox by the instruction processor to indicate the desired operation to be performed by the scalar processor.

0 – Undefined
1 – Change mailbox address
2 – Report current status
3 – Start or accelerate activity

Word 3

Bits 0,1 CF – Current Function

The control function is entered into the mailbox by the scientific processor to indicate its current or most recent condition relative to control code requests.

0 – Mailbox access problem. Error status stored in mailbox words 4–7.
1 – Mailbox changed.
2 – Activity decelerated or status stored.
3 – Activity executing.

NOTE: Word 3 cannot be altered by software while the scientific processor is executing. The processor needs the control block address to decelerate the activity, and it may retain it internally or obtain it from word 3 when needed.

2.2.2. Hardware Status Registers

The hardware status registers are four 36-bit registers that exist in total or in part within the scientific processor hardware, within mailbox words 4–7 and the scientific processor control block words 8–11. They are used to hold status applicable to the hardware and the present (or most recent) activity. At activity initiation (barring a status indication that prevents activity initiation) the hardware copy of the status registers are loaded from the scientific processor control block for that activity. At activity termination the contents of the hardware copy of the status registers is written both into the control block and into mailbox words 4 through 7.

Hardware Status Register 0

Register 0 contains the external interrupt type indicators in bits 0–34, and bit 35 is the flag that indicates a pending internal interrupt. The occurrence of each external interrupt condition is reflected by setting the corresponding bit in this register.

Bit	External Interrupt Cause Indicated
0	Mailbox Valid bit not set or control block address boundary violation
1	Critical environmental fault (power loss, coolant loss, etc.)
2	Scientific processor hardware check
3	Reserved
4	Reserved
5	IPL reboot interrupt received
6	Error on information from storage
7	Error on information to storage
8	Multiple uncorrectable errors in storage
9	Real address not available
10	Storage internal check
11	Interface sequence error
12	Reserved
13	Storage interface timeout error
14	Reserved
15	Reserved
16	An internal interrupt is taken as external
17	Generate Interrupt (GI) instruction
18	UPI interrupt received
19	Quantum timer runout
20	Program segment alignment or length error
21	Register save area not on correct storage boundary
22	Local storage base address not on correct boundary
23	Local storage length granularity incorrect
24	Program address register fault (activity segment table entry not found)
25	Data address fault (activity segment table entry not found)
26	Address limits error
27	Storage protection check - Execute
28	Storage protection check - Read
29	Storage protection check - Write
30	Data Alignment error
31	Attempted Test and Set/Clear on local storage segment

Bit	External Interrupt Cause Indicated
32	Program address register bits 18,19 not zero
33	Reserved
34	Reserved
35	Pending Internal Interrupt Flag

Hardware Status Register 1

Register 1 holds the instruction that caused the specific external interrupt.

Hardware Status Register 2

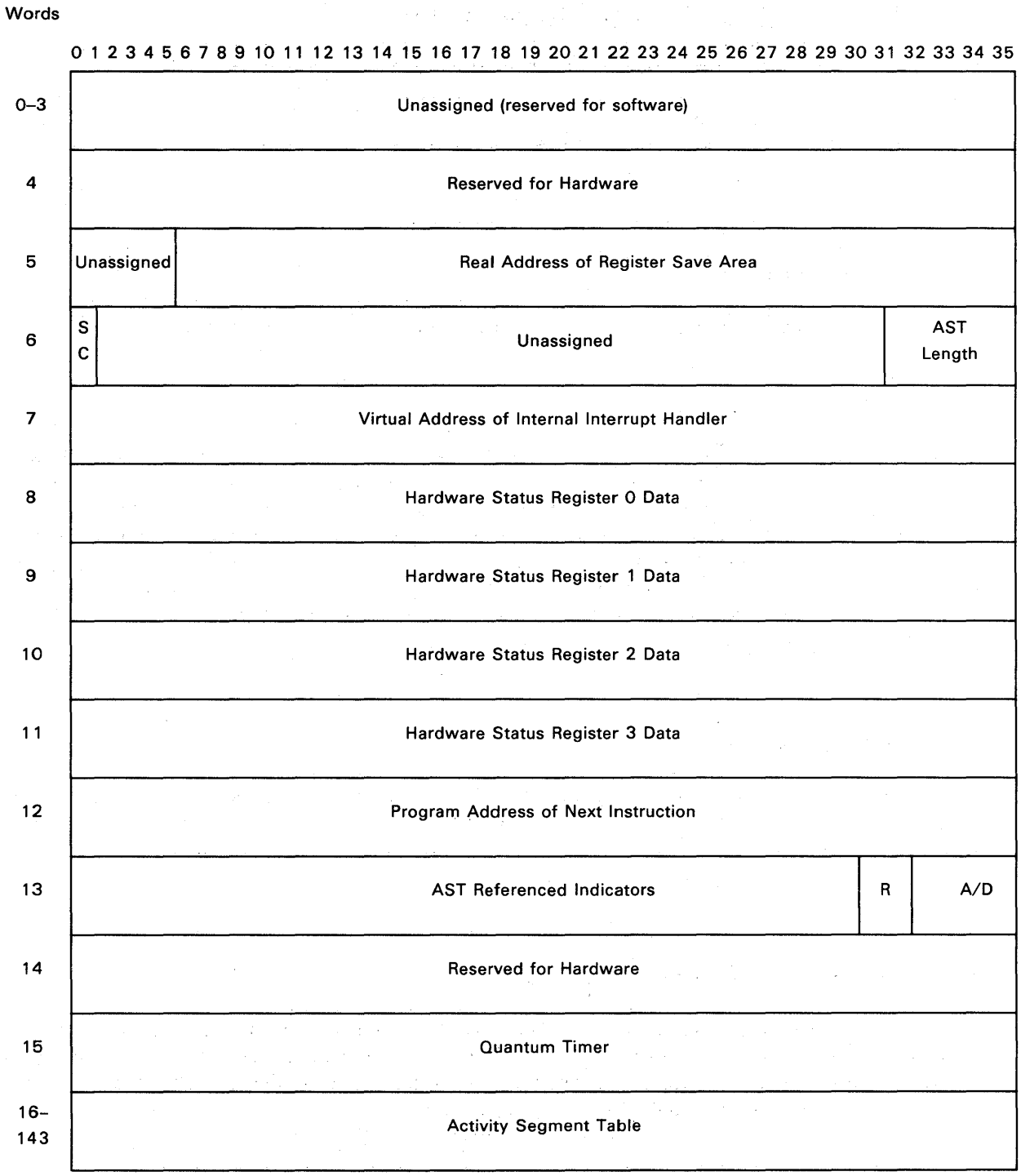
Register 2 holds status pertaining to the external interrupt. The first six bits contain the interrupt type code, which is the bit number of the indicator in register 0. The remaining bits contain interrupt type-dependent information. Once the type code field is set, subsequent interrupts do not alter register 2 or register 3 bits 0-28, but are indicated only by setting the indicator bit of register 0. Thus the status from only the first interrupt is stored and retained. External interrupt types, bits 0 and 16-19 of word 0 are excluded from this in that they never alter register 2 or register 3, but merely set their register 0 bits and in some cases register 1.

Hardware Status Register 3

Bits 0-28 are an extension of register 2. Bit 29 is the Diagnostic Instruction Executed indicator. Bit 30 is the Non-Local Jump (NLJ) indicator and bits 31-35 are the Jump History File pointer.

2.2.3. Scientific Processor Control Block

The scientific processor control block is created by the instruction processor associated with each activity. The control block contains all the information or pointers to the information needed by the scientific processor to process the activity. It resides in the scientific storage. The control block structure has the following format:



SC – speed control bit
R – reserved
A/D – acceleration/deceleration status

NOTE: Words 0–3 must remain available to the instruction processor. Words 4–7 may be accelerated, but need not be decelerated because the scientific processor will never alter them.

2.2.4. Jump History File

The scientific processor contains a 32-entry circular list of virtual addresses indicating the history of jumps taken by the activity's execution. A new entry is written to this list each time a jump is taken. Entries are placed into succeeding higher locations, wrapping around from location 31 to location 0. The value placed into each entry is the virtual address of the instruction to which the jump was made. Entries are stored only for jumps, not for interrupts. No disable of this feature is provided.

The file pointer is contained in scientific processor control block word 11 (see 2.2.3). It points to the next available location for storing an entry. Pointer wrap-around and the consequential overwriting of previous entries occurs without notice, the effect being simply to preserve the most recent 32 values. When an activity is switched off the scientific processor, its jump history file is stored in the register save area.

2.3. Scalar Module

The scalar module uses the General registers for single-precision and double-precision scalar data operands, and for holding virtual address information. Frequently used scalar variables in a program are stored in a local storage to provide fast access to this data. Local storage is not shared with other scientific or instruction processors.

Scalar operations include those portions of code that are not independent and must execute serially. The scalar module also serves as the control for both the scalar and vector modules and as the interface to the scientific processor storage.

As instructions are decoded, they are issued to either the scalar or the vector module for execution. Scalar instructions are issued and placed into execution as rapidly as one every cycle providing there is no data dependency conflict and the execution facilities are available.

Arithmetic and control operations are initiated in sequential program order but may complete out of that order because different paths are used through the computational logic.

Most data paths and computational entities are two words wide such that most scalar operations are performed in the same amount of time regardless of being single- or double-precision.

The scalar module contains the following functional sections:

- a scalar processor that includes:
  - an arithmetic logic unit,
  - a floating-point exponent arithmetic unit,
  - a multiplication unit,
  - a shifter, and
  - data paths.
- General registers that hold scalar integer or floating-point data, virtual addresses, and stride values;
- a high-speed local storage that holds frequently-used scalar variables, and subroutine parameters and constants. Its location in the scalar module ensures that the scalar processor is not slowed down waiting for data from main storage;

- an address generation section receives information to generate a real address, load either data or instructions from main storage, or store data to main storage;
- an instruction buffer that holds a working set of instructions;
- an instruction flow control that reads, decodes, and dispatches instructions;
- an interconnect that routes data and instructions throughout the scientific processor; and
- storage interface buffers that hold data going to and from registers and scientific processor storage.

2.3.1. Instruction Flow Control

The instruction flow control section acquires and decodes all instructions based upon the Program Address register or jump/branch evaluation results. It is divided into instruction addressing, instruction buffer, and instruction control subsections.

The instruction addressing subsection contains the program address generation register and its control. The Program Address register holds the 36-bit virtual address. The instruction buffer holds 16 pages of 256 instructions per page (all instructions initially reside in the scientific storage) to accelerate instruction availability in increments of 256 words. The control logic contains the instruction decode and sequence initiation logic, and it creates control words such as: vector control words, mask control words, and address generation control words that are sent to various sections of the scientific processor.

The operations performed by the instruction flow control section are:

- maintains program addresses;
- initiates and sustains page fetches until the page is resident in the instruction buffer;
- issues instructions to the scalar and vector modules;
- resolves instruction issue conflicts;
- sequences the scalar instruction pipeline;
- selects one of four scalar instruction operand sources:
  - vector file elements
  - scientific processor storage operands
  - local store operands
  - data in buffer for overlapped requests, Load G-Multiple and Load Loop Control
- decodes instructions;
- executes jump instructions;
- initiates the interrupt process; and
- controls starting and stopping of instructions.

Addressing and Control

The program starting address for an activity comes from the scientific processor control block (see 2.2.3). This address is compared with the contents of the content addressable memory to determine if the page containing the instruction is resident. If a comparison is made, a valid instruction is fetched by the instruction buffer and a new program address (starting address +1) is generated by the instruction flow address subsection.

The control logic generates the valid control words for the instruction and issues the instruction to the sections participating in its execution. However, anytime a program address does not compare to the contents of the content addressable memory, a miss is generated and a page of instructions (256 words) is loaded into the instruction buffer. Then, instruction execution resumes at the program address that generated the miss.

Program Addressing

Program addressing is normally required with the execution of jump instructions. Jump instructions are of two types: those that specify the jump address as simply an immediate offset within the current 65K instruction segment, and those that specify a complete 36-bit virtual address as the jump target. In any case, program addressing is done entirely in terms of virtual addresses, and these are translated by the activity table just as are data addresses.

To enable convenient hardware acceleration of program code, the scientific processor does not support writing into the code segment by any means. Also, code segments are restricted to be mapped starting on 64-word boundaries of main storage and to be allocated with length granularities in integral multiples of 256 words and must not exceed 65K total length (16-bit address range).

Instruction Buffer

The instruction buffer is a 4096-word cache storage containing up to 16 blocks of 256 consecutive instructions each. These blocks are loaded from the scientific processor storage during normal activity progression. All instruction requests are checked against a record of all blocks currently resident in the instruction buffer.

The instruction buffer is divided into four, 1K-word (40-bit word) buffers. Each instruction buffer has a separate input data register. Four, 40-bit (36 data plus 4 parity bits) instruction data words come from the scientific processor storage. The words are stored in the four instruction buffers simultaneously. This operation is done every time an Acknowledge comes from the scientific processor storage. A total of 64 Acknowledges is required to write the whole page (256 instructions) into the instruction buffers.

All address requests from the Program Address register in the scientific processor for an instruction are made to an internal instruction buffer with an access time of one clock cycle.

An instruction address request that is resident in the last block referenced from the instruction buffer makes that instruction requests, and all consecutive requests to that block, available in one clock cycle. A request for an address resident in an instruction buffer block that was not used for the last instruction request is delayed for one clock cycle. All consecutive requests to that block are then available at the single clock cycle rate.

An instruction request not resident in one of the blocks currently stored in the instruction buffer generates a miss. This requires a scientific processor storage request for that block through the address generation section of the scalar module. The single address request that generated the miss causes the block in the scientific processor storage containing that address to be loaded into the instruction buffer on a modified first-in first-out basis (block aging algorithm) if current block residency has exceeded 16. (Blocks are loaded consecutively from 0-15. The last used block is never overwritten if it is the current block selected for aging.)

If a parity fault occurs in the instruction buffer, one attempt is made to reload the page where the error happened and the faulty instruction is executed again.

Instruction Generation

The internal page address is generated from the comparison of Program Address register bits 0–27, four parity bits and the validity bit to the contents of the content addressable memory. If a comparison is made, one of the 16 lines becomes active and is encoded into the 4-bit page address. The page address changes when Program Address register bits 20–27 do not compare to the Last Page register and the content addressable memory indicates a "hit". Every time a page compare is made, a page compare Hit designator is set indicating that the corresponding page in the instruction buffer containing the instruction is resident.

2.3.2. Address Generation

The address generation section generates absolute storage addresses for storing and retrieving all program instructions and data in the scientific processor storage.

This section receives virtual addresses that must be converted to absolute (real) addresses before referencing the main storage unit. The sequence required to complete this operation is determined by an address generation control word from the instruction flow control section.

During the address translation operation, the most significant 18 bits of the virtual address (equivalent to a 256K block of addresses within the total address range) is matched with one of up to 32 active blocks of real addresses contained in the activity segment table.

The information that defines the scientific storage addressing environment for a specific activity is stored in the activity segment table. This information includes:

- the real address location in scientific storage,
- read/write/execute permission bits,
- block length, and
- virtual block number.

Operations in the address generation section are controlled by the control word received from the instruction flow control section. When the address generation section receives a valid control word it starts executing operations.

Storage Referencing

The scientific processor, through the address generation section, provides a pipelined interface to the scientific processor storage unit. The data interface is four words wide and is capable of transmitting a new storage request every 30 nanoseconds. (Figure 2–2 shows the request–acknowledge interface.) Successive requests, within limits, do not depend upon the acknowledgement of previous requests. Up to eight (or sixteen) requests may be issued before an Acknowledge is received for the first request. As acknowledgements are received, additional requests may be issued in order to keep the pipeline operating at its maximum rate.

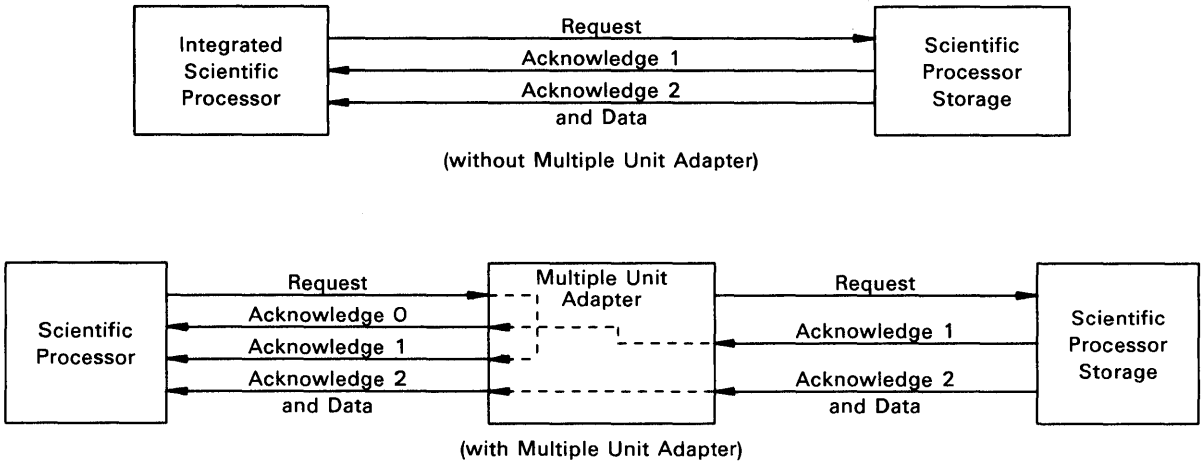


Figure 2-2. Storage Request-Acknowledge Interface

If access to more than one scientific processor storage is desired from the scientific processor, a multiple unit adapter is required. The adapter provides an interface between a scientific processor and up to four scientific processor storage units. The adapter contains an eight deep request buffer and decodes address bits 0 and 1 to determine which scientific processor storage to forward the request to. The adapter continues requesting a scientific processor storage until its request buffer is empty, there are eight unacknowledged storage requests outstanding, or a request for a different scientific processor storage unit is requested. When a different scientific processor storage is selected, the adapter waits until all requests from the first scientific storage are acknowledged before making the first request to the new selected scientific storage.

In a configuration with a multiple unit adapter, a scientific processor may have a maximum of 16 outstanding storage requests (eight requests in the scientific storage unit and eight requests in the multiple unit adapter).

When the multiple unit adapter sends a request to a scientific storage unit, it also sends an acknowledgement to the scientific processor. The scientific processor treats an acknowledgement without a multiple unit adapter the same as an acknowledgement with a multiple unit adapter. When the scientific processor receives either acknowledgement, it transmits another request because the request buffer has an empty position.

Segment Mapping

The local storage segment is defined to be local to the activity, meaning that data in it is not shared among activities. The data is accelerated into local storage at the beginning of the activity. If the length of the segment is less than or equal to 4096K words, then the entire segment is loaded there, and all references to the segment are directed to the local storage. If the segment length exceeds 4096K words, then the local storage is filled beginning at the start of the segment. References to the first part of the segment are handled internally, but references beyond this length are directed to the appropriate place in main storage.

The first entry of the activity segment table defines the addressing space of the local storage segment, which has the following restrictions:

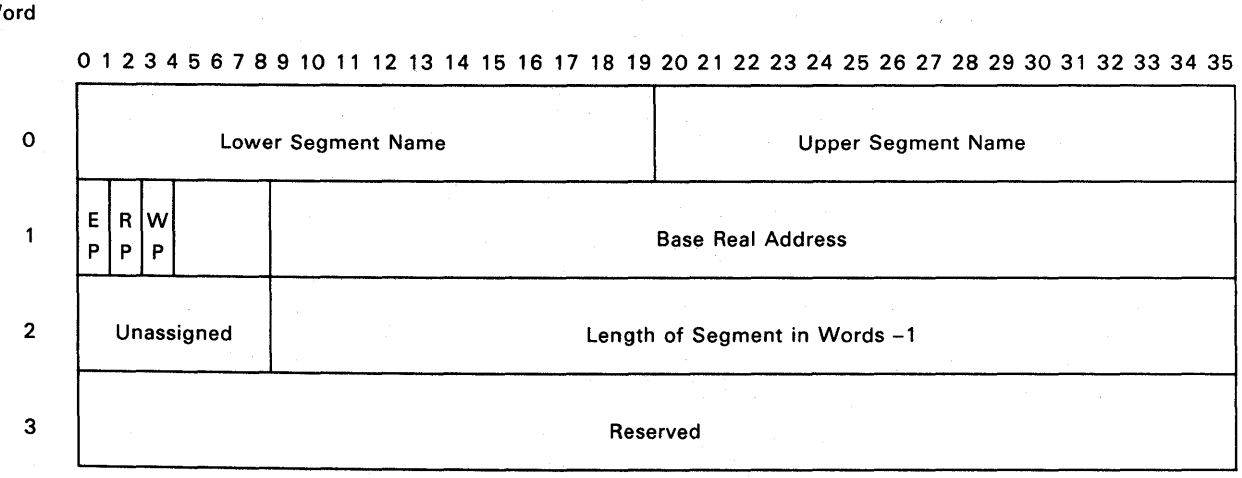
- The length must be a multiple of 32 words, for example, the least significant five bits of the Length field must be all 1's.
- The Base real address must be on a 16-word boundary, for example, the least significant four bits must be all 0's.
- The Permission bits are ignored, and implied values of 011 are used (read and write, but not execute).

Activity Segment Table

The activity segment table provides the information for translating virtual addresses into absolute (real) addresses.

Each activity has an activity segment table that resides in the scientific processor control block, and defines the addressing environment available to the activity. Each table entry is four words in length. The number of entries in a particular table is restricted to a maximum of 32, and is specified by word 6 of the scientific processor control block.

Activity Segment Table Entry Word Format



Word 1

Bit 0	EP – Execute permission
Bit 1	RP – Read permission
Bit 2	WP – Write permission
Bits 3–5	Unassigned

The activity segment table provides the information for translating virtual addresses for purposes of referencing main storage. Given a 36-bit virtual address, the left 18 bits of it are defined as the segment name. To translate the virtual address, the table is searched

until a segment entry is found between the lower and upper segment name boundaries contained in word 0. If no entry in the table satisfies this condition, then an addressing fault interrupt is caused. The table cannot contain more than one entry satisfying this condition.

Once the table entry is found, a 36-bit quantity called the virtual segment offset is calculated by subtracting the lower segment name value from the leftmost bits of the virtual address. The real address for the reference is then given by the base address plus the virtual segment offset. However, before the reference itself can be made, the permission bits must be inspected and the virtual segment offset must be less than or equal to the value in the length field. Failure to pass these checks causes an error interrupt.

One exception exists: if a read reference is attempted and the table entry used is the same table entry used to translate the current program address (for example, address of the instruction), then checking of the read permission bit is not performed, though the other checks still are required. The purpose of this special case is to permit access to constants in proprietary (execute-only) code.

Each table entry defines the translation for a contiguous extent of the virtual address space. Such an extent may cover more than one 256K segment, as the term segment is used in the context of the instruction processor translation process. However, since the entire extent must be handled as a single entity, the term segment is used to refer to the entire extent, whatever its size.

Word 13 of the scientific processor control block contains 31 indicator bits; each bit corresponding to one of the 31 possible table entries, ignoring the first entry that always defines local storage (see 2.3.5). Whenever a particular table entry is successfully used to perform a translation, the corresponding indicator bit is set to 1 and the other bits remain unaltered.

NOTE: For efficiency of implementation, actual limits checking is done on a block basis rather than word basis. Here a block means a contiguous 4-word group of words on a 4-word boundary in real storage. Access to any word of a block implies access to any other word of that block.

Address Limits Error Interrupt

Length checking is applied to all references to the local storage segment, whether the reference is directed to local storage or the scientific processor storage, and regardless of the method of specification. If the Base and Length fields do not meet the specified parameters, then an Address Limits Error interrupt is caused.

2.3.3. Scalar Processor

The scalar processor section performs all functions that relate to scalar arithmetic and logical operations. These operations may be either single-precision (36 bits) or double-precision (72 bits) integer or floating-point.

This section contains a control logic unit, an integer arithmetic logic unit (integer ALU), floating-point unit, and a multiply unit. The integer ALU contains a G-register file consisting of sixteen 72-bit registers. These registers are used as addressing base registers, scalar arithmetic accumulators, or as a source of operands for vector instructions.

The following types of instructions are executed in the scalar processor:

- Register to storage (RS) and register to register (RR) format scalar computational instructions: Add, Subtract, Multiply, Divide, Convert, Shift, Compare, Logical, Absolute Value, Count Leading Signs.
- Scalar Move instructions (Load, Store, and Move) and conditional jumps using G-registers.
- Increment and Jump Less (IJL) and Decrement and Jump Greater (DJG)
- Generate Index Vector (GXV) index value generation
- Read G-register when G-register data is required by other instructions, for example, vector-vector (VV) format instructions requiring one or more G-operands instead of Vector operands.

Integer ALU

The integer ALU performs part of or all of the required operations when the following types of scalar instructions are executed:

Load
Store
Move
Integer Add
Integer Subtract
Integer Divide (with floating-point unit)
Floating Point Divide (with floating-point unit)
Absolute Value
Logical
Compare

The G-register file consists of sixteen 72-bit (two words) registers with a 30 nanosecond read and write cycle time. Separate inputs from the instruction flow control section for read addresses and for write addresses enable a read operation and a write operation to be performed in the same 30 nanosecond time slot. The read address and write address cannot be the same. A read operation reads 72 bits. A write operation can write the upper word (bits 0-35), lower word (bits 36-71), or both words (bits 0-71).

These registers are used as general accumulator registers for double-precision and single-precision operands and as holding registers for virtual address and stride information required in the address generation section.

In addition to its arithmetic capabilities, the scalar ALU section also contains a Branch and Scalar Condition Code circuit which performs a compare function on data read from the Augend register. Results of the compare are transferred to the instruction flow section where it is determined if a jump is to be made. A two-bit Scalar Condition code is generated during the compare and is transferred to the control block section. This code contains results of the compare between operand 1 (OP1) and operand 2 (OP2) as follows:

Code	Compare Results
0	OP1 = OP2
1	OP1 > OP2
2	OP1 < OP2
3	Reserved

Floating-Point Unit

The floating-point unit performs floating-point add, floating-point subtract, floating-point data conversion, count leading signs, and shift operations. Also, part of the multiply and divide operations are performed in the floating-point unit.

During a divide instruction, the floating-point unit is controlled by the scalar processor control. Divide instructions must be completed before the floating-point unit will accept another instruction. Two cycles are required to complete execution of an integer instruction in the floating-point unit. Four cycles are required for a floating-point instruction.

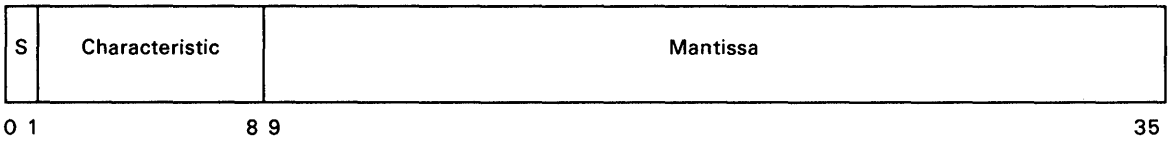
Data paths to and from the floating-point unit are two words wide. Single-precision operations are performed in the upper 36-bit portion of data paths and registers. Parity is checked on all data paths. Adders and shift matrices incorporate parity predict circuitry. Output data from normalization counters and count registers is duplicated and compared as a check against errors because they do not contain parity.

Number Representation and Data Types

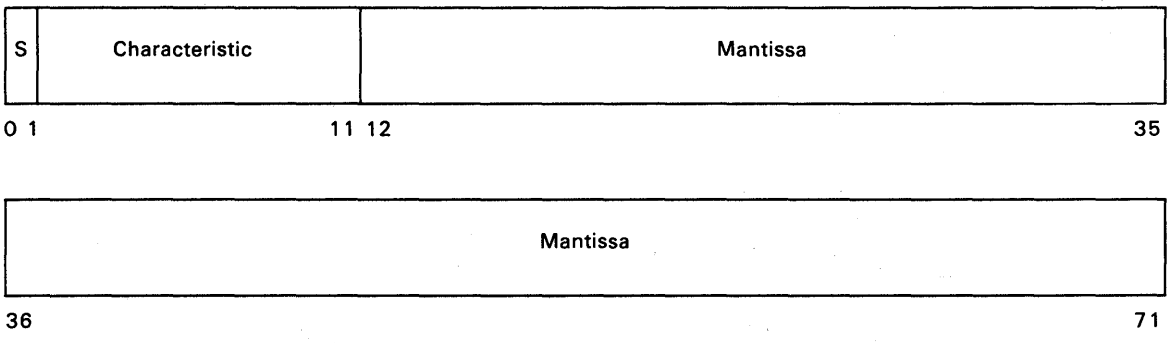
Integers are single-precision or double-precision numbers in which the binary point is to the right of the lowest order bit, for example, whole numbers.

Floating-point numbers are represented in single-precision format as a 27-bit fractional quantity (mantissa) multiplied by a power of 2, and in double-precision format as a 60-bit fractional quantity multiplied by a power of 2.

Single-Precision Floating Point



Double-Precision Floating Point



where:

S (Sign) — The S-bit is the sign of the numerical quantity represented by the floating-point number. If S is 0, the quantity is positive and if S is 1, the quantity is negative.

Characteristic — The characteristic represents both the numerical value and sign of the characteristic.

The 8-bit characteristic of a single-precision floating-point number is a value in the range of +127 through -128. The characteristic is made positive by adding 200_8 to the characteristic.

The 11-bit characteristic of a double-precision floating-point number is a value in the range of +1023 through -1024. The characteristic is made positive by adding 2000_8 to the characteristic. If the S-bit is a 1, the characteristic is complemented.

Mantissa — The mantissa is the fractional portion of a floating-point number and is normalized so that the absolute value is greater than or equal to one-half but less than one. A normalized number is one in which the most-significant bit of the mantissa does not equal the sign bit. The exception to this is a normalized zero. A normalized zero is always a word of all zeros, or all ones. Negative mantissas are expressed in ones-complement form.

Four types of integer and floating-point data is operated on in the scalar processor. The type of data is defined by the value in the instruction word t-field. They are as follows:

t-field	Data Type
0	Single-precision integer
1	Double-precision integer
2	Single-precision floating point
3	Double-precision floating point

Sign Manipulation

When a floating-point instruction begins executing in the floating-point unit, the first action is to separate the exponents from the mantissas and to save the signs of the operands. These signs are saved and used to compute the sign of the result, which is applied to the result at the end of the instruction.

Four input factors partially determine the sign of the result, they are:

Is it an Add instruction?
Is it a Subtract instruction?
Is the sign of the Augend positive or negative?
Is the sign of the Addend positive or negative?

From these four input factors, four interim results are determined. These interim results are:

Mantissa adder operation (add or subtract)
Sign for Add or Convert instruction
Sign for Divide instruction
Sign for multiply instruction (transferred to multiply unit)

The interim results along with other information and timing data continue the sign determination process. Because Divide instructions are not pipelined, timing is necessary to control instruction execution. The timing also controls the sign logic. In integer division, the number of iterations necessary to complete the division process is determined by the number of significant bits. In floating-point division, the number of iterations is either 27 or 28 (single-precision) or 60 or 61 (double-precision). When an instruction completes execution and the results are transferred to the integer ALU, the output of the sign logic determines if the result should be complemented. If complementing the result is necessary, it is complemented in the integer ALU.

Rules for number representation and floating-point instructions are:

1. Integers are represented in ones complement format.

2. Floating-point operands are converted to sign-magnitude notation, for example, all numbers made positive, signs retained, and applied to result at the end of the instruction.

3. Signs of floating-point operands and the result of arithmetic calculations are used to calculate the sign of the result.

4. if necessary, result is complemented in the integer ALU.

5. If a floating-point instruction produces a result with a positive or negative zero mantissa, the integer ALU is cleared to all zeros to produce a correctly normalized zero.

6. Results of floating-point instructions are always normalized.

7. A normalized zero is added to an unnormalized number and the result is correctly normalized. Results are undefined if unnormalized operands are used in other floating-point instructions.

8. Results are undefined if invalid shift counts are used for shifts.

Faults that may occur during floating-point operations are:

Divide Fault - Occurs during integer and floating-point divide operations when the divisor is positive or negative zero.

Characteristic Overflow - Occurs during floating-point divide, addition, subtraction, and double-precision floating-point to single-precision floating-point conversion when the characteristic of the result exceeds 177_8 for single-precision and 1777_8 for double-precision.

Single-Precision Characteristic Underflow - Occurs during floating-point divide, addition, subtraction, and double-precision floating-point to single-precision floating-point conversion when the exponent of the result is less then -200_8 and the mantissa of the result is not zero.

Double-Precision Characteristic Underflow - Occurs during floating-point divide, addition, and subtraction when the exponent of the result is less then -2000_8 and the mantissa of the result is not zero.

Integer Overflow - Occurs during integer divide with the t-field equal to 1, convert floating-point to integer, and convert double-precision integer to single-precision integer when the result has more than 35 significant bits.

Multiply Unit

The multiply unit processes either integers or floating-point mantissas. Sign determination and addition of the two characteristics is performed in the floating-point unit. The sign of the result is transferred to the multiply unit and is merely staged through the unit and combined with the product before being transferred to the integer ALU. The sum of the characteristics is also transferred to the multiply unit where the extra bias is subtracted in the Bias Subtract register. Then the characteristic is also staged through the unit and combined with the product in the CHAR Select register.

2.3.4. Local Storage

The scientific processor has an internal 4096-word local storage area. This storage is intended for frequently used scalar variables and constants by providing fast access to this data. The local storage is loaded from the first activity defined segment of the scientific processor storage unit during acceleration and stored back into scientific processor storage during deceleration.

The local storage contents is accelerated (moved on activity initial start-up from scientific processor storage) and provides high speed access with no main store delays for high use constants or scratch pad. The local storage content is decelerated (returned to scientific processor storage) on activity completion or exit from the scientific processor.

The local storage logic is controlled externally by the address generation, control block, and instruction flow control sections; it has no internal control logic.

When an activity switches into the scientific processor, the segment (bank) defined by activity segment table entry 0 is loaded into local storage (up to 4K of that segment). This data is used internally and is not shared or sharable among activities.

Addressing Local Storage

Local storage is specified either directly, using an absolute local storage address from the instruction, or indirectly, by specifying a full virtual address that maps into the local storage segment.

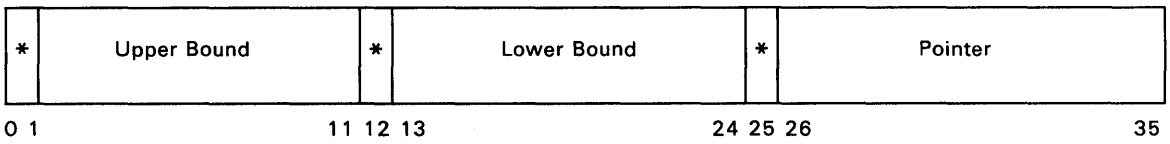
Length checking is applied to all references to the local storage segment, whether the reference is directed to the physical local storage or the main storage, and regardless of the method of specification. Failure to pass this check results in an address limits error interrupt.

Automatic Storage Stack

A scientific processor activity may set aside an area in the lower 4K of the local storage segment for temporary values (automatic storage) to be managed as a stack. Scalar instructions that directly reference local storage have an option in which the specified location is treated as an offset relative to the current position pointer associated with the stack. This pointer is maintained through the scientific processor instructions that acquire and release stack frames.

The local storage stack definition word, shown below, defines the area. Instructions in the RS format (see 4.2.2) have an option (b=15) that allows them to access locations in this defined area. Specifically, the local storage address, is formed by concatenating the five rightmost bits of the instruction u-field on the right of the 11-bit pointer value taken from state register 7 (S7). This forms the 16-bit direct offset in the local storage segment. The pointer, along with the upper and lower limits for it, are maintained in State register 7.

Local Storage Stack Definition Word Format (S7)



*indicates bits not used.

Instructions are provided to increment and decrement the pointer, checking it against the appropriate limit, see the Advance Local Storage Stack (4.18.5) and Retract Local Storage Stack instructions (4.18.6).

NOTE: The stack effect occurs on a frame basis, not on a word basis. For each value of the pointer, a block of 32 words is available with this mechanism. Changing a pointer makes available a different 32-word block.

The Stack Definition word is not protected, and its contents can be changed freely by the activity. Also the local storage area described by the Stack Definition word is not protected and can be accessed by the other local storage addressing mechanisms.

Obtaining Data from Local Storage

Data stored in local storage can be accessed by scalar or vector instructions. Local storage can be addressed directly or indirectly. Direct addressing bypasses the virtual to real address translation in address generation by supplying the real address at time of instruction decode. Indirect addressing requires translation of the virtual address in the instruction or pointed to by the r_1-field.

A RS formatted scalar instruction with a b-field of 0 or 15 can directly access local storage for operand 2 of the instruction. Operand 1 is stored in the G-register specified by the r_1-field of the same instruction. The b-field of 0 requires the u-field of that instruction to be equal to or less than 4096. The b-field of 15 requires that the Local Storage Stack Definition word contain a value in its 11-bit pointer (bits 25-35 of S7) that when the 5 least significant bits of the u-field (bits 31-35) are catenated onto the right of the pointer the

resultant real address is a value equal to or less than 4096. These two direct access methods bypass the normal address generation translation and allow an access time to local storage of 30 nanoseconds.

A RS formatted scalar instruction with a b-field of 1 through 14 indirectly accesses local storage through address generation. Both operands 1 and 2 have to use the address generation section for virtual to real address translation. The virtual address associated with a b-field of 1 through 14 must map into the local storage segment (activity segment table 0) with virtual segment offset plus the contents of the u-field resulting in a value equal to or less than 4096.

Vector instructions cannot directly access local storage. Indirect addressing requires an address generation translation of the virtual vector instruction address into the local storage segment with the result of virtual segment offset plus or minus its stride being equal to or less than 4096. The same 5 cycle access time attributed to all indirect accesses to local storage applies to all vector instructions.

Local Storage Acceleration

When an activity is switched into the scientific processor, the values in the first activity segment table entry, which defines the local storage segment, are checked. If the local storage segment exceeds 4K, portions above 4K remain in the scientific processor storage.

When the activity is switched off, the same amount of storage that was accelerated is copied back (decelerated) to the scientific processor storage. No attempt is made to update the resident portion of local storage in the main storage area while the activity is executing.

2.3.5. Store Buffer

The store buffer section provides a buffer between the high burst transfer rate of the vector file section and the slower potentially start stop rate of the scientific storage interface. This buffering allows the vector section to proceed with the next vector instruction if required resources are available after unloading the referenced vector file to the store buffer area.

The store buffer section contains two separate 16 address by 4-word store buffers (store buffer 0 and store buffer 1). The store buffers can perform simultaneous read/write operations. Data from the vector files is temporarily written into the store buffer for a store vector instruction, and then sent to the scientific storage. The 4-word output of the store buffers can be realigned; take the condensed vector file data and expand it to the desired storage address stride desired.

Vector Store controls the writing of the data from the vector files and the address generation section reads the data out of the store buffers. During the buffer load sequence each write address is for two words; two addresses for each buffer for a total of four write-address registers. Each word can be read individually; therefore, there are eight read address registers. Words are normally read from a buffer at 1 to 4 per cycle. The Store Vector Indexed instruction however, reads only one word per cycle.

The store buffer section:

1. provides a buffer between the vector files and the scientific processor storage,
2. provides a data path between the G-registers and the scientific processor storage,
3. provides a data path between the control block section and the instruction buffer, and
4. matches a vector of index values (address offset) with the corresponding data during indexed store vector operations.

Write Data Registers

Data received in the Write Data register is the four words from a vector file or the two words from a G-register in the scientific processor requested by a Store Multiple instruction. The data words are unscrambled in the Write Data registers depending on the requirements of the operation. Two words are transferred to Index Buffer 0 or 1 from the Write Data register during a Store Indexed Vector Instruction.

Write Data registers for the store vector instructions accept words 0 and 1 from the primary vector files without delay but, words 0 and 1 from the secondary vector files are delayed one cycle to allow for correction to the word to address arrangement in the store buffers. The Write Data registers have a two-input selector with each selector fed by a four way selector to re-arrange the data for proper alignment in the store buffers and index buffers for the Store Vector and Store Vector Indexed instructions.

For the Store Alternating Elements Vector and Store Vector Indexed instructions both primary and secondary vector file words are fed straight in with no delays. For the Store Alternating Elements Vector instruction the data is fed in and out without any modification to the data.

Data Out Registers

The data output registers select the address boundaries that the words are to be stored on. This register routes data to the scientific processor storage, local storage, or instruction buffer. All data written into the store buffers is controlled from the vector store section; all data read out of the buffers is under the control of the address generation section. Feedback between the read and write controls prevents buffer from overrun.

These registers consist of:

- instruction buffer data out registers,
- scientific processor storage data out registers, and
- local storage data out registers.

An eight-way selector is used to transfer the store buffer output to the instruction flow control or scientific processor storage via one of the two input selectors of the data out register. Local store input is fed to the other input of the data out register for transfer to scientific processor storage or instruction flow control. Local store data, which is passed straight through word 0, stays as 0, and 1 stays as 1. Local store data is always selected during any scalar instruction. The read toggle function selects which of the two buffers the words are coming from. The 2-bit address generation selector controls the data transfer and selects the word from the store buffer. The output from the data out registers is even parity for local storage and instruction buffer and odd parity for scientific processor storage.

2.3.6. Loop Control

The loop control section manages the vector operations. This involves processing vector strips and elements within strips. The loop control provides a vector length and an element pointer to sections needing them. The loop control also processes a set of conditional jumps that help efficient processing of vector strips and elements in each strip. Since the loop control deals mainly with vector properties, for the most part it affects the vector section. Some instructions, however, will affect the scalar section.

The loop control section contains the eight loop control registers and the control logic that operates with the Build Vector Loop (BVL), Build Element Loop (BEL), Jump Vector Loop (JVL), and Jump Element Loop (JEL) instructions. The loop stack can be loaded and stored via the Multiple Load/Store instructions and is a part of the scientific processor acceleration/deceleration sequences. The loop control broadcasts the current loop count/element count data to the various vector sections so that current loop and element positions or ending can be determined.

Loop Control Register Formats

There are two sets of loop control registers: eight 45-bit Vector Loop registers and eight 14-bit Element Loop registers. There are two 3-bit loop control register pointers: the Current Vector Loop Pointer (CVLP) and the Current Element Loop Pointer (CELP). The CVLP selects one of the eight Vector Loop registers that provides the current vector loop parameters and the CELP selects one of the eight Element Loop registers for controlling element loops.

To the user the Vector Loop and Element Loop double-word fields as seen in storage are usable for a particular register set identified by CVLP and CELP. The values of CVLP and CELP are not directly readable by the user, save through the use of the Store Loop Control Registers (SLCR) instruction. The values can however, be changed to any value within the range 0 to 7 by the user either with the adjust instructions (for example, CVLP, CELP, or CVELP) or through use of BVL (B_xVL_x) and BEL instructions. In addition, JEL and JVL may change the register address depending on value of the loop control parameters.

Vector Loop Registers

Each of the eight Vector Loop registers can be used to hold parameters defining the iteration of a loop over strips of a vector. Each register consists of the following fields:

Field	Description
Maximum Size	The value of this field is declared upon entry into the loop. When the field is 0, up to 64 elements are processed. When the field is a 1, either single-word or double-word operands are processed, however, the element count and the next element count are limited to 32 elements.
Remaining Length	This field specifies the number of elements remaining to be processed by this loop. The field is set to the starting vector length and is decremented under control of the maximum size field on each pass through the loop. The number of elements

Field	Description
	processed on each pass is called element count and is determined by the maximum size and remaining length fields. The number of elements processed on the next pass is called next element count and is determined by the maximum size, remaining length, and the element count.
First Alternate Element Count	Can be used instead of the element count or next element count by certain vector instructions. A program may calculate a value and place it into this field. The value is not affected by the Maximum Size field and its validity is evaluated only when used by execution of an instruction.
Second Alternate Element Count	Similar to the First Alternate Element Count field.

Element Loop Registers

Each of the eight Element Loop registers can be used to hold parameters defining the iteration over elements of a strip. Each register consists of the following fields:

Field	Description
Maximum Element Count	This field determines the number of element loops. This field is initialized upon entry into the loop and is set to values between 0 and 64, however, values higher than 64 do not cause fault conditions.
Element Pointer	This field specifies the number of passes made through an element loop. After each pass through the loop, it's count is increased by one and compared to the maximum element count.

CVLP and CELP

Although each Vector Loop and Element Loop register set is composed of 8 elements only a single element can be accessed at one time. Two pointers exist to indicate the current element of each register set: CVLP and CELP. While these parameters cannot be affected or read in isolation, such as a move or load, these parameters are affected by several instructions.

Loop Control Register Mapping

The contents of the Vector Loop and Element Loop registers are manipulated only by certain loop control instructions. Therefore it is unnecessary to define internal formats for these registers, other than to indicate the number of bits of information per field. However when this information is placed into scientific storage, its format must be defined. These registers can be stored by means of a Store Loop Control Register instruction and also upon activity deceleration when they are stored in the register save area. The format used is the same in both cases.

NOTE: The CVLP and CELP values are always stored and loaded along with the Vector Loop and Element Loop register contents.

When placed into storage, Vector Loop register zero and Element Loop register zero share the first double word; Vector Loop register one and Element Loop register one share the next double word, and so on. The format of each double word is:

Word	Bits	Description
Even	0–2	Not used (except first two double words)
Even	3	Not used
Even	4	Maximum size from Vector Loop (0–64, 1–32)
Even	5	Not used
Even	6–35	Remaining length from Vector Loop
Odd	0–1	Not used
Odd	2–8	First alternate from Vector Loop
Odd	9–10	Not used
Odd	11–17	Second alternate from Vector Loop
Odd	18–19	Not used
Odd	20–26	Maximum element count from Element Loop
Odd	27–28	Not used
Odd	29–35	Element pointer from Element Loop

Bits not used are written to zeros on stores and ignored on loads, with the following exceptions: bits 0–2 of the first even word are used for CVLP, and bits 0–2 of the second even word are used for CELP.

Loop Control Instructions

The Build Vector Loop and Build Element Loop instructions establish vector loop and element loop parameters, respectively. The Jump Vector Loop and the Jump Element Loop instructions establish the termination conditions for the vector and element loops, respectively.

The Adjust Loop Register Pointer instruction changes the contents of the CVLP or the CELP or both pointers. The Store Loop Control Register instruction saves the values of the loop control registers and the Load Loop Control Register instruction restores the values.

Loop Control Register Operation

The Vector Loop and Element Loop registers are used for vector loop and element loop parameters, respectively. Though functionally independent, corresponding Vector Loop and Element Loop registers are mapped together and generally used together. The CVLP always identifies one of the eight Vector Loop registers as the current vector loop definition. This definition supplies four values (element count, next element count, first alternate, and second alternate) which are available to all vector instructions for use as strip lengths. The 2-bit l field of the instruction selects one of these four.

Execution of a Build Vector Loop (BVL) instruction sets CVLP to a new value specified in the instruction and establishes the loop parameters in the selected Vector Loop register. The corresponding Jump to Vector Loop (JVL) instruction modifies these parameters for each new pass until the full length is completed. It then exits from the loop, and in so doing sets CVLP to the new value specified in the JVL. For correct nested operation, this new value should be the value that CVLP had prior to the corresponding BVL. Compiling this correct value is trivial in all cases except the last one just prior to a subroutine return. That case is resolved by the reasonable convention that saves and restores the entire L-register contents at all subroutine boundaries. This is conveniently handled by the SLCR and LLCR instructions.

The CELP and the Element Loop registers operate similar. The CELP selects an Element Loop register, the Element Pointer field value of which is used by scalar instructions to pick one of the elements of a vector register.

NOTE: The loop control registers and instructions have been defined such that zero is a valid loop count value. A zero value for either a vector loop or an element loop causes the enclosed operations to be executed zero times. (This is not strictly true, since reductions and scalar operations in a vector loop are still executed.)

2.3.7. Mask Processor

Many instructions in the scientific processor repertoire, primarily the vector instructions, use individual element execution control by using a mask. The scientific processor uses a single 64-bit Mask register that is used in several different instructions concurrently. The mask processor section allows the single source mask to provide multiple references at several different bit positions, one for each active destination, under one mask controller. Also, several mask status condition instructions aid in logical branching within vector loops.

The mask processor section operates in conjunction with the add pipeline section, multiply pipeline section, and move pipeline section in the vector module to control writing of data into the Vector File on an element-by-element basis. The mask processor consists primarily of a 64-bit register located in Special Registers S4 and S5. There is one parity bit for eight mask bits for a total of eight parity bits. As instructions are executed in each pipeline; the mask processor provides applicable mask bits, as determined by the instruction word t-field and c-field, to each pipe at the appropriate time.

During single-precision instructions, two mask bits are transferred to the applicable pipeline each cycle. During double-precision instructions, only one mask bit is transferred each cycle. Only mask bits used in an individual pipe is transferred to that pipe allowing concurrent control of multiple instructions from a single mask register.

The mask processor may also be used to determine jump conditions during Conditional Jump instructions; it may also be the source or destination during Scalar Move instructions.

2.3.8. Control Block

The control block section provides the interface and control within the scientific processor to accelerate an activity (task) on the scientific processor from an interrupt received through the universal processor interface (UPI).

The control block controls the acquisition of all the scientific processor activities initial state and local storage data. On fault or normal activity termination it controls the state save and interrupt operation (deceleration) of the scientific processor.

State Operations

The scientific processor states are maintained in an internal hardware State register located in the control block section. This register is not accessible by any instruction processor. Figure 2–3 provides a general overall diagram of state switching.

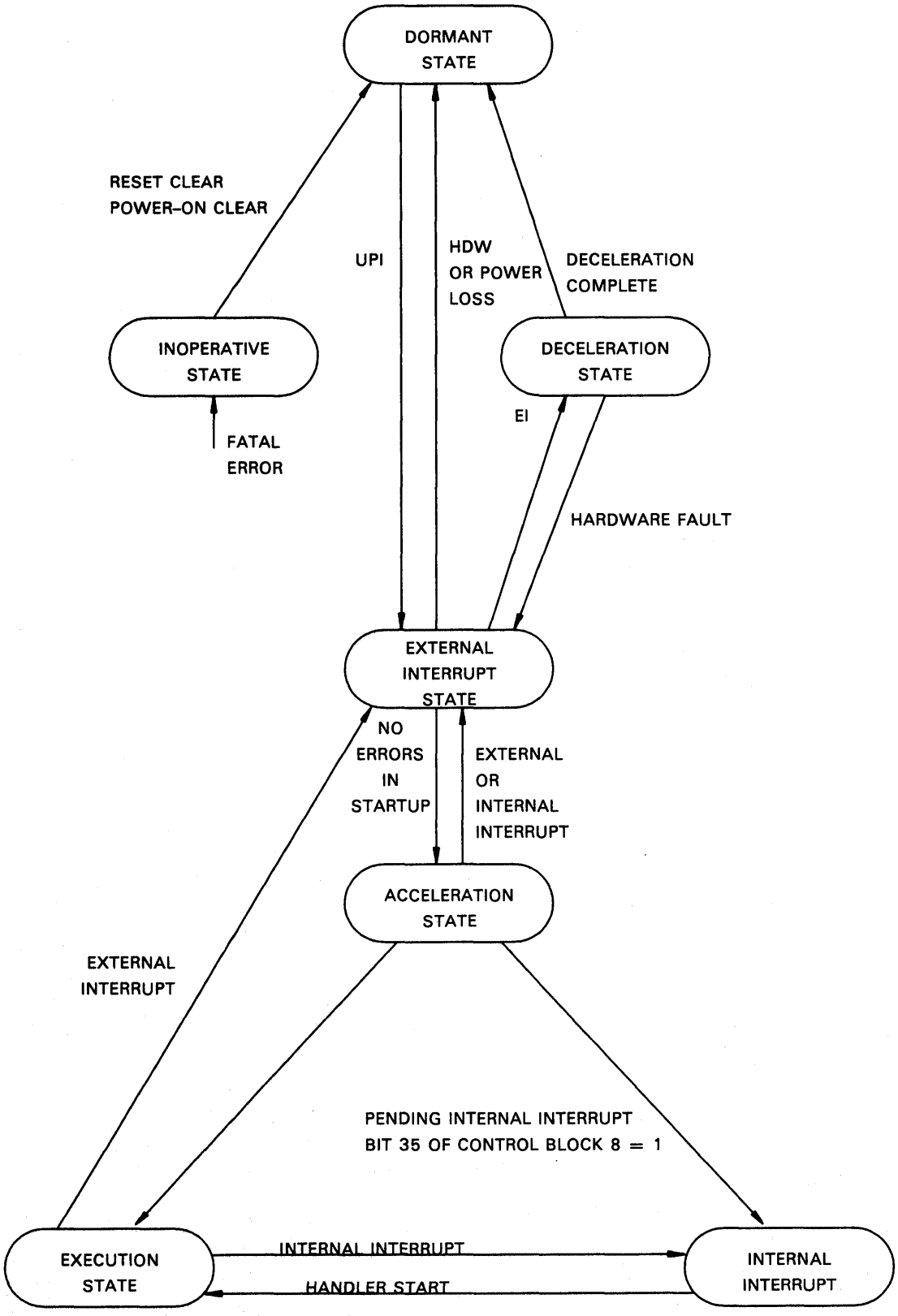


Figure 2-3. Integrated Scientific Processor State Switching

Dormant

The dormant state occurs when the scientific processor is powered up. In the dormant state no activity occurs except for monitoring and logging of interrupts and issuing Reset clears. An initial program load causes a mailbox address change and a universal processor interface clear causes a Reset Clear signal to be issued to the scientific processor. Any other external interrupts (except a UPI) is logged in hardware status register 0. This monitoring, logging, and clear issuance continues until a UPI interrupt is received from the instruction processor.

Acceleration

The acceleration state does the actual loading of a scientific processor activity. The only path available for entering this state is from the dormant state through the acceleration startup check of the external state.

Acceleration first reads control block words 8–15 to get hardware status register 0, word 8 in place to log any external interrupts that may occur while in the acceleration state. If an external interrupt was in word 8 when it was loaded, the acceleration process is immediately halted and a switch to the external state occurs. If no external interrupts are present, acceleration continues loading data from the scientific processor storage.

After all the data is loaded and no external interrupts are present, a single bit (bit 35) in hardware status register 0 is checked to see if any internal interrupts are pending (interrupts that have occurred before the activity was decelerated). If an internal interrupt is not pending the execution state is entered.

Acceleration is always done with real address references (no virtual addresses) and no updates of the R registers for address translation takes place.

If at any time an error is found in the addresses needed for acceleration (Register Save Area, Local Storage Segment Address), a switch to the external state occurs.

If an internal or external interrupt occurs during acceleration, which is not loaded from the scientific processor storage, an immediate switch to the external state is made. These types of interrupts are caused by the acceleration process itself or by an external source such as a power loss or the scientific processor storage interface error.

Deceleration

Deceleration is the process of storing pertinent scientific processor activity and state data to the scientific processor storage for use by the instruction processor. No address checking is done in this state because the addresses were verified during acceleration.

Before any actual storage takes place, the scientific processor checks for a hardware fault because of an undetected parity error in a previous state. If this parity error had occurred during acceleration, control block 13, bit 33 is set indicating that the previous acceleration has not completed and no instructions have been executed. If the parity error had not occurred during acceleration, bit 34 of control block 13 is set indicating that a full deceleration (all scientific processor data) had not completed.

In either case, bit 33 or 34 set, the unit support controller is notified of the error and proceeds to logout the scientific processor via the system support processor scan set; provided its system support processor option for logout is set. When the logout is complete, the scientific processor receives an Interrupt Clear to clear out the parity error and then starts again. From this point, the short deceleration is done and sends only control block 8–15 to the scientific processor storage.

If the hardware check finds no parity errors, and no subsequent errors occur, deceleration stores the following registers or files in the scientific processor storage:

- Vector files
- Local store
- G–registers
- Loop Control registers
- Scientific Processor Control Block registers
- State registers
- Jump History Stack

Deceleration is always done with real address references (no virtual addresses) and no updates of the R–registers for address translation takes place.

Several types of errors can occur during deceleration that will interrupt the normal process flow:

1. Scientific Processor Parity Error and the Scientific Processor Storage Error – The first occurrence after deceleration causes the logout and a short deceleration sequence occurs.
2. Scientific Processor Parity Error – The second occurrence causes a switch to the inoperative state. No further deceleration is done.
3. Scientific Processor Storage Error (Second occurrence) – These types of errors are considered nonfatal because the scientific processor storage does not lock up. System support processor intervention to clear the error is not required. The action taken here is to retry a short deceleration. (One additional retry of short deceleration is provided for the scientific processor storage errors.)
4. Scientific Processor Storage Error (Third occurrence) – A switch to the inoperative state is made.

Execution

When a scientific processor activity has been loaded (accelerate state) it is executed in this state. Either an internal or external interrupt causes the instruction execution to cease and a state switch to Internal or External is made depending on the type of interrupt. External interrupts have the highest priority should both types of interrupts occur simultaneously.

Internal Interrupt

This state is entered when an internal interrupt is present without an external interrupt. The previous state may have been accelerate, for the pending internal interrupt, or Execute, for the actual occurrence of an internal interrupt. The action taken here is to store the interrupting instruction and Address and save a return address. Then a jump to an internal interrupt handler is done and the State register is switched to the Execute state. The saved return address is used when the interrupt handler is complete.

Inoperative

This state indicates that the scientific processor is nonfunctioning and cannot be used until a Power-up clear is performed. The state is entered whenever a fatal error of any type is received. These fatal errors include:

- Page miss during acceleration or deceleration.
- Parity errors from instruction translate RAM.
- Parity errors from the instruction buffer during acceleration or deceleration.
- Parity errors on data to instruction buffer during acceleration or deceleration.

External Interrupt

The external interrupt state is the center for most state switch decisions. That is, when a state is exited, the external state is entered to make a decision as to what the next state should be. Abort deceleration and accelerate startup check are part of the external Interrupt state.

The different sequences that occur in the external Interrupt state are most conveniently examined by looking at the previous state.

If dormant was the previous state and there is an outstanding power loss or hardware fault interrupt, the hardware status registers are stored in the mailbox and the dormant state is reentered. If no hardware or Power Loss interrupts are outstanding, then the hardware status registers are cleared and the instruction processor is given the information that a new activity is being loaded by writing mailbox word 2 to mailbox word 3. This places the next scientific processor control block activity address into the present scientific processor control block activity address.

The entry to the acceleration state is contingent on the next scientific processor control block activity address (mailbox word 2) being on the correct boundary and has a valid address (valid bit set). If either of these checks fail, the same action for hardware fault or power loss interrupt is taken.

If execute was the prior state then at this point the interrupting instruction and return address, for the interrupt that caused the state switch, have been stored to scientific processor control block registers. Also, if an Internal interrupt is present, the pending internal interrupt bit in hardware status register 0 is set. From here, if the interrupt was not an initial program load, the deceleration state is entered. The state register switches to Suspend if an initial program load is present.

If decelerate was the previous state, the first parity error that had occurred would cause a short deceleration. The second parity error changes the state to inoperative. The first and second occurrence of a the scientific processor storage/multiple unit adapter error results in a short deceleration while the third error causes a switch to the inoperative state.

If accelerate was the previous state the internal or external interrupt that had occurred would require that a short deceleration be entered.

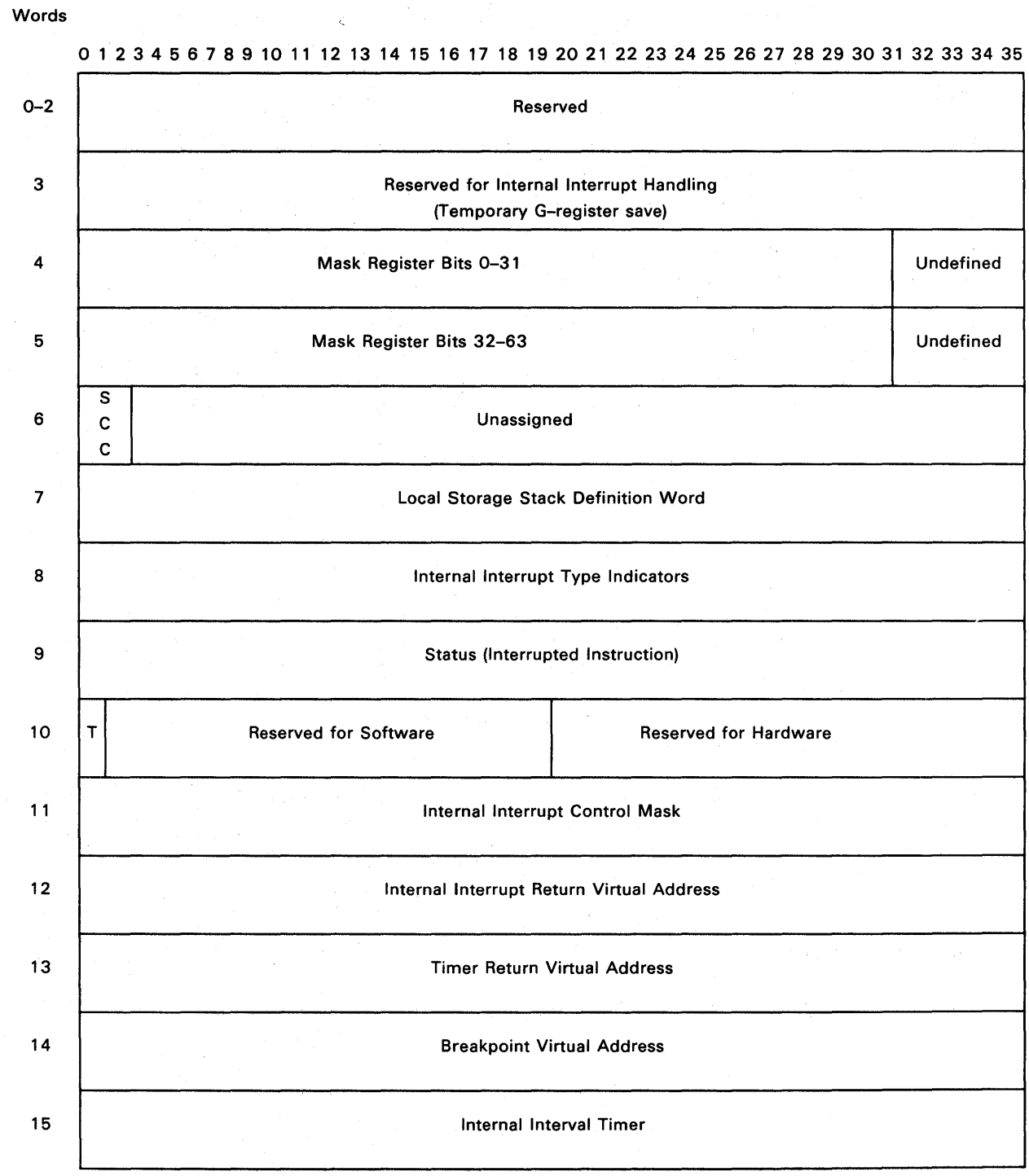
Special Considerations

An activity is decelerated only in response to an external interrupt (host serviceable interrupt) of some kind, which causes hardware status register 0 to become non-zero. Deceleration normally stores the contents of hardware status register 0-3 into both the control block (words 8-11) and the mailbox words 4-7. Software can examine and compare both status areas following deceleration, and thereby deduce the existence of possible scientific processor control errors.

External interrupt types bits 1 and 2 of status register 0 may occur even while the scientific processor is dormant, and these are recorded in hardware status registers 0-3. Any subsequent universal interface interrupt causes this status to be reported in mailbox words 4-7, and prevents an activity from starting.

State Register Set

The state register set word format is shown in Figure 2-4.



NOTE: Bits 32–35 of words 4 and 5 will have 0's in these bits whenever these registers are used as source operands. For consistent operation, software should never write data other than 0's into these bits.

Figure 2–4. State Register Word Format

2.4. Vector Module

The vector module has sixteen sets of 64-word vector registers used for holding vector operands. Each register has space for 64, 36-bit data words. Vector computations are defined as register-to-register, using vector register contents as source operands, and depositing the vector result into a vector register. Special loop control instructions control breaking long vectors into strips small enough to fit in a vector register.

The vector module contains the following functional sections:

- an add pipeline that:
  - adds,
  - subtracts,
  - converts,
  - shifts, and
  - executes logicals;
- a multiply pipeline that:
  - multiplies,
  - divides, and
  - calculates population counts;
- a move pipeline that:
  - moves vectors between registers,
  - moves and compresses data,
  - moves and distributes data,
  - calculates population parity, and
  - assists the add pipeline when a convert operation involves a precision change;
- vector registers that hold vector data. Each register can hold up to 64 single-precision or 32 double-precision vector elements; and
- an interconnect that routes data and instructions throughout the scientific processor.

2.4.1. Vector Register

The vector register section contains register file storage space for 128 vector files. These are divided into two copies of 64 primary and 64 secondary vector files.

Each vector file is structured as follows:

- Each file contains sixty-four 36-bit words. The file can contain up to 64 single-precision elements or 32 double-precision elements.
- Addressing for the primary and secondary vector files is independent of each other.
- Each vector file has an eight-way interleave, that is, eight read or write operations can occur simultaneously each clock cycle.

- Only the first 16 of the primary and secondary vector files are accessible to the programmer, the remaining 48 files of both primary and secondary files are reserved for hardware scratch area.

Primary and Secondary Vector Files

The primary vector file is used for operand data for the augend in the add pipeline, for the multiplicand in the multiply pipeline, scalar data in the scalar vector control. It is also used for primary storage where data for vector store instructions are routed to the store buffer.

The secondary vector file is used for the operand data for the addend for the add pipeline, and is available at the same time the augend is available from a primary vector file. The secondary vector file also supplies the multiplier for the multiply pipeline, the single operand for the move pipeline, and a second word pair for the store buffer.

The vector file has a data register set of eight separate registers that provide input to the primary and secondary vector files. Both copies of the vector files are written with identical data at the same time in each copy.

Data for the vector files originates from one of the following inputs:

- Vector File Scalar Input register (this data is from the scalar processor and could be the result of a RR format instruction).
- Two inputs from the load buffer during vector load instructions (this data is passed to the vector files at four words per clock cycle).
- Resultant operands from the multiply pipeline, the move pipeline, or the add pipeline.

Vector Register Memories

The vector register section has four independent memories (primary) with a duplicate copy of each memory (secondary). The primary and secondary memories contain storage space for a complete copy of all 16 vector files.

Of the four memories, memory 0 is the programmer visible (real) vector file set and memories 1 through 3 are exclusively hardware visible (shadow) vector file sets. Memories 1 through 3 provide instruction overlap capabilities, performance, and implementation enhancement techniques for the instructions that require vector file access.

Duplicate copies (primary and secondary) of each memory can reference data concurrently from two vector files, one from the primary copies and one from the secondary copies. These references may be in the same memory or in different memories.

During a write operation, both the primary and secondary vector file select is forced by the control logic to the same vector file within the same memory. This insures that the duplicate copy approach is maintained in the primary and secondary copies within the selected memory.

The vector file logic includes multiple vector file read data and write data registers to support concurrent operation of the vector load, vector store, vector arithmetic, and scalar

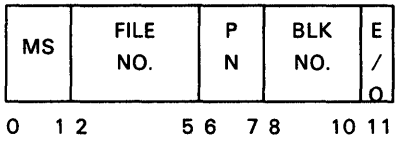
vector control sections. Also, the physical to logical arrangement of a 64-word vector file is divided into four rows of 16 words with the 16 words divided into eight blocks of two words each.

Each of the two-word blocks is treated as an independently controlled, vertical slice read/write and address through memories 0 through 3, primary or secondary.

Each individual reference to any vector file can read or write 72 bits (one double-precision data word or two single-precision data words) in a vector file reference cycle. Using a primary copy of one vector file and a secondary copy of a second vector file concurrently allows dual vector file operand referencing in a single vector file reference cycle.

Vector File Addressing

Addressing for the read and write operations of the vector files is done by a 12 bit address from vector control. This address selects a single vector file element and is broken into five fields as shown.



where:

Bits 0,1 — MS - Memory Select.
Selects memories 0 through 3.

Bits 2-5 — File Number
Selects 1 of 16 files in selected memory. This field is supplied by one of the four vector operand address fields of the vector control word.

Bits 6,7 — PN - Pass Number
Selects one of four passes in the selected file.

Bits 8-10 — BLK NO. - Block Number
Selects one of eight blocks in the selected pass.

Bit 11 — E/O - Even/Odd
Since a word pair is read or written every cycle no odd or even selection is required unless an odd number of single precision elements is written back into a vector file.

NOTE: Bits 6-11 select one of the 64 elements in the vector file chosen by bits 2-5 that resides in the vector file memory selected by bits 0 and 1.

Because of the eight block arrangements common to all four memories only a single vector file address is required for each of the four passes in one vector file. The 8-bit vector file

address (bits 0–7) remains the same for blocks 0–7 and only the pass number is incremented by one after each pass then reverted to 0 after the fourth pass is completed. (If bits 0–7 of the vector file address were all zeroes, then the first word pair 0,1 (pass 0) of vector file 0 memory 0 is selected.) Thirty–two cycles are required to read all elements of one vector file. Four addresses in each vector file use 64 vector file addresses for the 16 vector files. Read Data selection of the corresponding block is made by the Read Multiplexer Address counter in vector control.

2.4.2. Vector Control

The vector control section contains the vector control interface, the vector file control, and conflict detection logic.

The vector control interface performs the following operations:

- receives and accepts instructions, element pointer, element count, and abort information from the instruction flow control and loop control sections,
- acknowledges instructions when they are placed into execution, and
- forms the second instruction for multiple–pipe instructions where two pipelines participate in execution of instructions such as single to double conversions and indexed load vector instructions.

The vector file control and conflict detection logic:

- reserves vector file timeslots as required for instructions (the pipelines release the timeslot when the instruction is completed),
- selects vector file addresses, and
- detects data usage conflicts (chaining and anti–chaining conflicts).

Vector control accepts a 50–bit vector control word from the instruction flow control section. This control word is derived from the 36–bit scientific processor instruction word coded into a format interpreted by vector control. Instructions that require two pipelines for resolution, delay the acknowledge to the instruction flow control until both instructions have started in their respective pipelines. (The vector control receives only one control word for the dual pipeline request and forms the second instruction itself.)

Receive and Acknowledge Control

The instruction receive and acknowledge control function receives the following information from the instruction flow control and the loop control sections:

- vector control word,
- element count,
- element pointer,
- element count=0, and
- element count or element pointer out of range abort.

Vector control stores and interprets this instruction and parameter information for use in addressing and controlling the vector files and pipelines within the vector module to perform the requested vector related instructions. The element count information is forwarded to all pipes except scalar control, which requires element pointer information.

The acknowledge control accepts signals signifying start of execution from the individual pipelines and generates an acknowledge to the scalar processor module to allow the initiation of the next vector related instruction via a subsequent control word.

In the case where one instruction necessitates use of two pipelines to complete the operation, the acknowledge to the scalar module requires receipt of pipeline acknowledges from both pipelines. The first selected pipeline acknowledge starts execution of its portion of the single control word request and this initiates formation of a second control word within vector control and selection of the appropriate second pipeline. Not until the acknowledge from the second pipeline is received does vector control issue the acknowledge to the scalar module.

Receipt of an instruction request provides the following parameters from the vector control word:

- vector file timeslot pattern
- pipe selection
- source and destination vector file addresses
- operation modifier

Vector Control Interface

The vector control interface is used by the scalar module to issue vector control words to the vector module. These control words are derived from instruction decode in the instruction flow control section. Depending on the instruction mix, the instruction flow section can issue a vector control word every 30-nanoseconds to the vector module. However, the vector module can only interpret and dispatch these control words at a 60-nanosecond rate. An interface control protocol allows the instruction flow logic to fill the pipeline at a 30-nanosecond rate and keep it full at a 60-nanosecond rate, once the pipeline has been filled. The vector control interface also includes the loop parameter data provided from the loop control section.

Vector File Addressing

The vector control generates the vector file select from the vector operand fields in the vector control word.

The vector file address selection function receives the vector file address from the vector operand field in the control word and accesses the requested vector file address, selects the control word requested pipeline, and selects the pass number and possible shadow memory request from the selected pipeline.

File Number registers hold all the read/write operand file numbers for each pipeline (OP1, OP2 for source or read file numbers and OP3, OP4 for destination or write file numbers). These registers are loaded at start of instruction execution and held until the pipeline completes the instruction.

The primary and secondary Vector File Address Select registers are enabled by the vector control timeslot 0 register. Inputs to this register for vector file address selection are the File Number registers from each pipeline and the pass number and shadow memory select from each affected pipeline.

The five pipeline Read Multiplexer counters that enable the block read for the Read Out register are selected by the decode of the pipeline select field of the timeslot management word. Each pipeline has a Read Multiplexer counter that is synchronized with the start of an instruction so the mux address is applied to the Read Out register at the output Vector File RAM at the proper time. The counters are incremented each clock cycle and continue until a new instruction is started on that pipeline.

The pipeline Write Multiplexer counters (Add, Multiply, Move, Vector Load A–B, and scalar–vector control) select the appropriate Vector File Write Data register to channel the correct data into the vector file from an active pipeline. The counters are initialized to zero prior to the first available Vector File write data word and incremented each clock cycle. When an active pipe completes execution it clears the counters in a way similar to releasing vector file timeslots.

Vector File Logical and Facilities Usage Conflicts

The vector control section detects file access write/read, read/write, write/write conflicts on each file cycle. The instruction that encounters the conflict stops for one or more eight–cycle increment(s) and proceeds when the conflict has been resolved.

The vector control detects logical usage conflicts caused by asynchronous system operations and overlapped subsystem instruction execution. The major source of asynchronous operation is the system multiprocessor environment that creates request contention at the scientific processor storage. Within the scientific processor, asynchronous conflicts occur because of the varying instruction execution times of the scalar processor section and of the multiple pipelines in the vector module. The vector module, for example, does not execute a divide instruction in the same span of time as an addition or multiplication. The asynchronous interface for the scalar processor module and vector module causes conflict problems and in turn generates conflicts between those vector module sections (vector load, vector store, and scalar–vector control) that interface with the scalar processor module.

Facilities usage conflicts are a general category of conflicts that occur within the normal design constraints. An example would be an instruction held up in vector control with access to its available pipeline prevented because all of the vector file access timeslots are presently assigned.

Vector File Conflict Detection

The vector control conflict detection logic monitors all active instructions for conflicts that could occur if the instructions request the same vector file. Conflicts that can occur include: write/read, read/write, and write/write. Parallel reads (read/read) of different elements of the same vector file are allowed because they will not alter the data.

Monitoring for conflicts starts at the time an instruction becomes active in a pipeline and checks all subsequent instructions that become active in other pipes for similar Vector File usage. If two active instructions are allowed to share a common vector file element, the

validated instruction would prematurely alter the data or prematurely read data from that element. This element sharing would destroy the validity of both instructions. (The add pipeline or multiply pipeline can generate a conflict within themselves; generally, the conflict is between different pipelines.)

Conflict resolution logic shuts down the operand sequencing of the pipeline associated with the latter instruction in eight clock cycle increments until the conflict is gone or cleared; then, the instruction is allowed to progress.

Conflict detection is shown in the following block diagram in Figure 2–5. Each block of the diagram is numbered. The numbers correspond to the explanation following the block diagram.

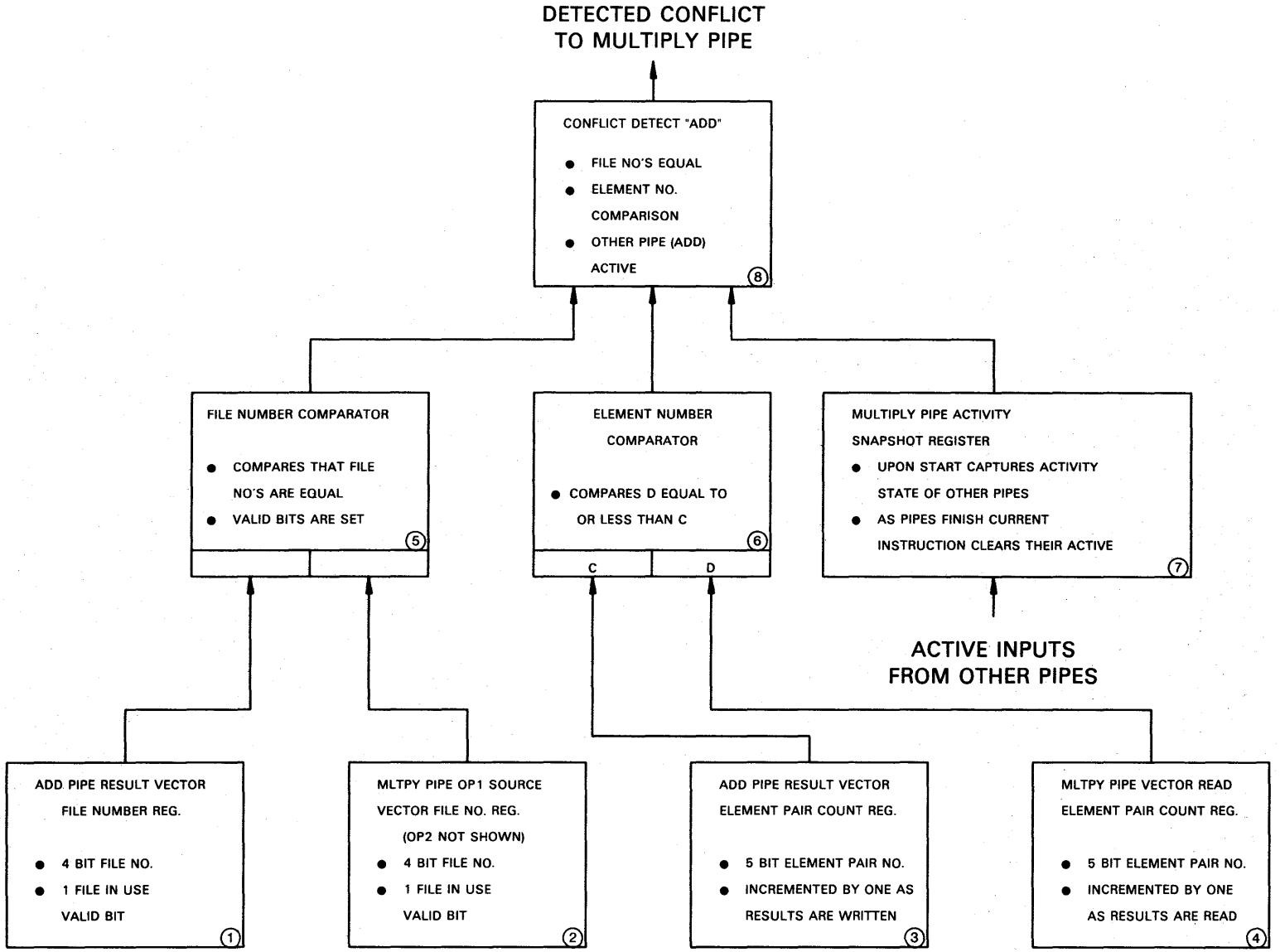


Figure 2-5. Typical Conflict Detector

① Add Pipe Result Vector File Number Register

This 5-bit register contains the 4-bit file number assigned to the Add instruction and a single valid bit that indicates this vector file is in use by this instruction.

② Multiply Pipe OP1 Source Vector File Number Register

This 5-bit register contains a 4-bit vector file number and a single valid bit indicator.

The output of this register and the register in block 1 are fed to block 5, the File Number Comparator.

③ Add Pipe Result Vector Element Pair Count Register

This register is the 5-bit add pipe vector element (word pair) count register for the OP3 result. The counter starts at zero and is increased by one when results are written in the selected vector file for the add pipe.

④ Multiply Pipe Vector Read Element Pair Count Register

This register is the 5-bit multiply pipe vector element (word pair) counter register. The register count is increased by one as results are read.

The output of this register and the register in block 3 are fed to block 6, the Element Number Comparator.

⑤ File Number Comparator

The OP3 and OP1 file numbers from the registers in blocks 1 and 2, respectively, are compared if both instructions are valid.

⑥ Element Number Comparator

The output of the registers in blocks 3 and 4 are fed to this comparator. The element counts from these two registers are compared to see if the multiply pipe has reached the point that the elements have been written into by the add pipe or have exceeded the add pipe element count.

⑦ Multiply Pipe Activity Snapshot Register

This register monitors the activity states of the other five pipes.

⑧ Conflict Detector

Inputs to this detector come from the registers in blocks 5, 6, and 7.

Vector Instruction Bypass

The vector instruction bypass function allows instructions to be placed into execution out of the normal program order in a certain case. This case requires that the following conditions be present.

1. Vector load busy.

2. Another instruction for vector load is queued into vector control and is in the Instruction Receive register.

3. An instruction for vector store is queued into the instruction flow control Function 0 register.

4. Vector store is inactive.

5. No conflict is possible (file numbers must not be equal) between the Vector Store instruction in instruction flow control Function 0 register and the Vector Load instruction queued in vector control.

If the previous five conditions are met then the following steps are taken:

1. Move the Vector Load instruction from the Vector Control Instruction Receive register into the Vector Load Instruction Hold register.

2. Transfer the Vector Store instruction from instruction flow control into the vector control Instruction Receive register.

3. Place the Vector Store instruction into execution in the normal manner and send the acknowledge to instruction flow control.

4. Move the Vector Load instruction back from the vector control Instruction Hold register into the vector control Instruction Receive register.

5. Continue in the normal manner.

Vector Control Word

All instructions that require the use of the vector module facilities cause the creation of a vector control word that is issued to the vector module by the scalar module. The vector control word contains 50 bits of control information and 7 parity bits. Also accompanying the vector control word is a vector parameter data word.

Selection of parameters from loop control is done concurrent with vector control word generation so the associate parameter information can be sent with the vector control word. While instruction flow control in the scalar module can issue control words at a 30–nanosecond rate maximum the vector module can only decode and dispatch them at a 60–nanosecond rate. Instructions that decode into vector control words exclusively are issued by instruction flow control without considering the state of the scalar module. However, some instructions require both the scalar module and vector module for execution. Thus, control words to both modules must be issued simultaneously and requires that instruction flow control checks the state of both modules.

Vector Parameter Word

The vector parameter contains 13 bits of data and 2 parity bits. While a control word is being created for the vector module, parameter information is selected from loop control. This information is available when the vector control word is issued.

2.4.3. Vector Add Pipeline

The add pipeline performs all arithmetic and logical operations on vector data, except multiplies, divides and product reductions. Data may be in single-precision (36 bits) or double-precision (72 bits) integer or floating-point format. The add pipeline contains control logic and two similar hardware units: a single-precision add pipeline with a 36-bit Augend and Addend interface and a double-precision add pipeline with a 72-bit Augend and Addend interface. All data manipulation is completed in one pass through a pipeline. Each cycle, the add pipeline operates on two single-precision operand or one double-precision operand.

Internal control of the add pipeline is provided by two internal random-access memories: operand control and data control.

Operand Control

The operand control memory controls sequencing of events necessary to complete an instruction.

When the add pipeline executes an instruction, the operand control logic controls sequence of events and data flow through the add pipeline after receiving a request and appropriate control information from the vector control section. When the add pipeline is not executing an instruction, it is in an idle state and will not become active until it receives a request from vector control. Before the add pipeline or other pipeline in the vector module starts executing an instruction, vector control receives the vector control word from the scalar module instruction flow control section. This control word completely defines the operation to be performed. When vector control receives a control word, it decodes it and determines what hardware resources are required to execute that instruction.

The instruction to be executed and the existing operating conditions determine what control information is transferred to the add pipeline. Some information is transferred each time the add pipeline is selected, for example, pipeline select; while other information is transferred only when specific conditions exist, for example, G-operand wait. The following information may be transferred from vector control to the add pipeline when it is selected to perform an instruction:

- Pipe select
- Operand sequencing microcode entry address
- Element count
- Single- or double-precision
- Timeslot reservation
- G-operand wait
- Element count out of range
- Extended sequence

Data Control

Data control logic operates in parallel with the operand control logic to control sequencing of data through the add pipeline. Flow of data through the pipeline is controlled by the data control memory. The address transferred from the vector control Entry Address register to the add pipeline is distributed to both the operand control and data control memories, but not at the same time. This address defines a location which contains the first step in a sequence of actions required to manipulate the data.

Conflict Detection

Vector module conflict resolution is controlled by vector control using conflict detection and status information provided by subsections within the vector module. For example, counters and registers that detect and store status information for the add pipeline are physically located in vector control but are controlled by the add pipeline. Detection of a conflict causes the following events to occur in the add pipeline:

- Halts activities in increments of eight cycles. Each inactive period is eight cycles long so that the add pipeline stays in synchronization with its reserved vector file access timeslot.
- Saves current operand control address.
- Loads zeros into Operand Sequencing Data register. This clears all counters in the add pipeline.
- Stops reading data from the vector file.
- Stops writing data to the vector file when all data currently in the add pipeline is through the pipeline.

Four Conflict File Number registers and four Conflict Element counters store status information pertinent to the instruction that is currently executing. The File Number registers are: OP1, OP2, OP3, and OP5. Status information in these registers identify the vector file elements currently being read from or written to. The Conflict Element counters are: RD OP, RD WR, $OP3_L$, and $OP5_L$. Status information in these counters identify the vector file word pairs currently being read from or written to.

Status	Description
RD OP	Contains number of element currently being read. This number defines the upper boundary and element number in RD WR counter defines the lower boundary of a range of elements currently reserved for use by the add pipeline.
RD WR	Contains element number defining lower boundary of range of elements currently reserved for use by the add pipeline.
$OP3_L$ and $OP5_L$	Identifies word pair currently being written. When an instruction completes a read from the vector file, but the write has not completed the value in $OP3_L$ counter transfers to $OP5_L$ counter. Both of these counters are disabled during extended sequence (multi-pass) instructions.

During single-pass instructions, values in the RD OP and RD WR counters are equal. During multi-pass instructions, the RD OP counter counts up to the maximum number of word pairs during the first pass, but the RD WR counter does not start counting until beginning of the last pass.

Number Representation and Data Types

This operation is the same as described in the scalar processor section (see 2.3.3).

2.4.4. Vector Move Pipeline

The vector move pipeline executes move, compress, and distribute instructions. It also participates in generate index and conversion instructions. The add pipeline does the active conversion of the single-precision to double-precision and double-precision to single-precision conversion instructions. The move pipeline stores the results of the conversions in the vector files. The move pipeline move buffer size is 32 addresses by two words wide.

The move pipeline executes the following instructions:

- Move Vector (MV)
- Double Move Vector (DMV)
- Move Negative Vector (MNV)
- Double Move Negative Vector (DMNV)
- Move and Compress Vector (MCV)
- Double Move and Compress Vector (DMCV)
- Move and Distribute Vector (MDV)
- Double Move and Distribute Vector (DMDV)
- Generate Index Vector (GXV)
- Population Parity (EBPV)

The move and add pipelines jointly share in the execution of the following instructions:

- Convert Floating to Double Floating, Vector (CFDFV)
- Convert Double Floating to Floating, Vector (CDFFV)
- Double Extract Sign Count, Vector (DESCV)

The add pipeline performs the arithmetic manipulation for these three instructions and places the result into a shadow memory of the vector files where it is accessed by the move pipeline, restructured and written back into a vector file in real memory (memory 0).

The move pipeline moves source elements from one vector file into a destination vector file. The element quantities for the moves are specified by vector control and range from 1 to 64. The element moves can be broadcast G-operands that uses a single G-register as the source vector and replicate that data into every specified element in the destination vector file. Conditional transfers of the elements use the Mask register for individual selection of those elements that transfer to the destination vector.

The compress instructions use the Mask register to select individual elements from a source vector file and transfer those to a destination vector file. Figure 2-6 shows the first seven elements of a source vector file and the seven corresponding bits of the Mask register. The source elements that have a corresponding one bit in the Mask register are transferred into the destination vector file starting at element 0. In this example elements 0, 3 and 6 are transferred to the destination. Alternately, zero is selectable as the element transfer select bit, which would transfer elements 1, 2, 4, and 5.

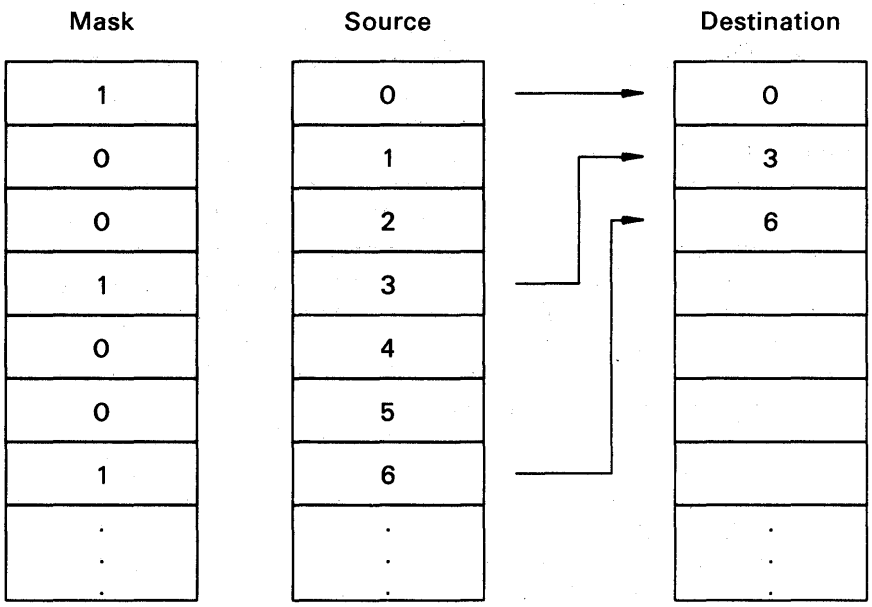


Figure 2-6. Basic Compress Instruction Element Transfer

The source element arrangement at the destination vector file for the distribute instructions is shown in Figure 2-7. Mask register bits of one move the corresponding source element into the same corresponding element of the destination vector file.

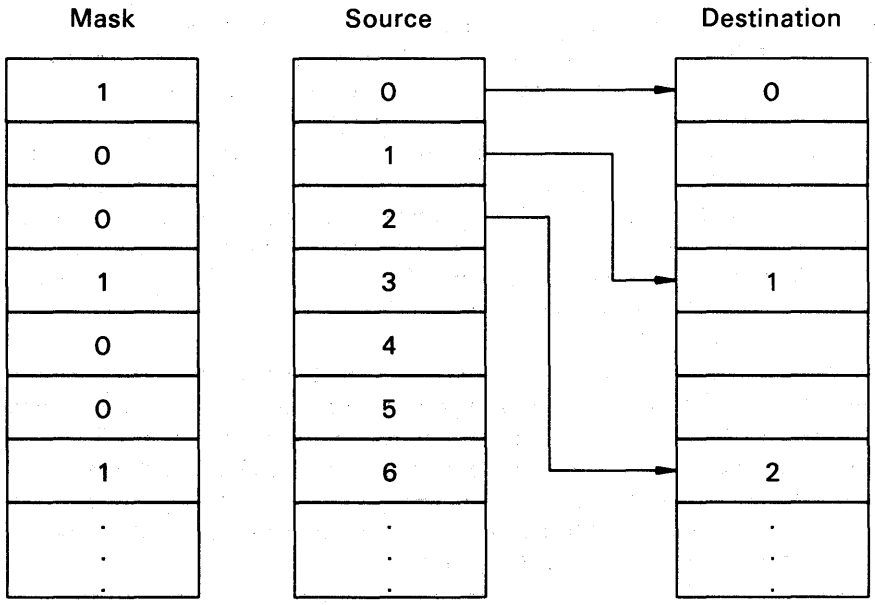


Figure 2-7. Basic Distribute Instruction Element Transfer

The Generate Index Vector instruction (GXV) generates a vector of indexes that are defined by a base (virtual address) and a stride. The base is contained in a G register specified by g_3 of the instruction. The stride is from the right half of that same G register if j=0 or a constant value of one if j=1. The vector of indexes is generated such that the first element of the vector is simply the base. The second element of the vector is the base plus the stride and the third element is the base plus twice the stride and so forth up to base plus 63 times the stride. This process generates a vector of uniform distanced indexes. (See Figure 2-8.)

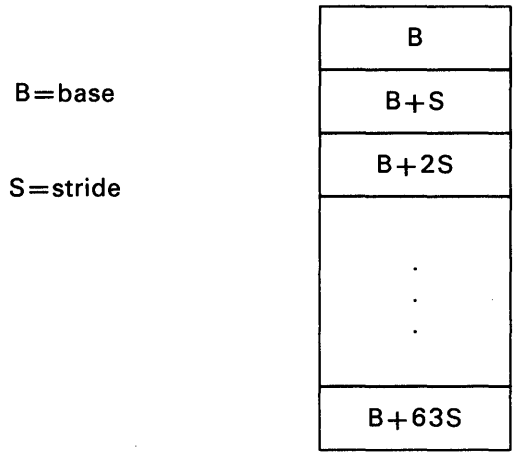


Figure 2-8. GXV Base and Stride Vector Arrangement

The type conversion instructions for vector data type and the count leading signs vector instructions, which move pipeline and add pipeline jointly execute, are handled similarly by move pipeline. Figure 2-9 represents the source and destination vector for these instructions. There are two type conversion instructions: Convert Floating to Double Floating, Vector (CFDFV) and Convert Double Floating, Vector (CDFFV).

For a single-precision to double-precision type conversion the source vector has 32 elements of 36 bit words and the result of the conversion is 32 double-precision elements filling one vector file. Figure 2-9 shows elements 0, 1, 2, through 31 in the source vector and 32 double-precision words resident in the destination vector after the conversion. For a double-precision to single-precision type conversion the opposite transfer is performed.

The move pipeline also assists in the execution of the count leading sign instruction for double-precision. The Count Leading Sign instruction is the Double Extract Sign Count, Vector (DESCV). The move pipeline's part in the execution of this instruction is very similar to the double-precision to single-precision type conversion. The DESCV instruction makes up to 32 double-precision elements available to the add pipe for examination of each count leading signs vector. This count is the number of consecutive bits, starting at bit number one, that are equal to bit zero. The add pipeline moves the elements for examination into a temporary location and move pipeline writes them back into a destination vector.

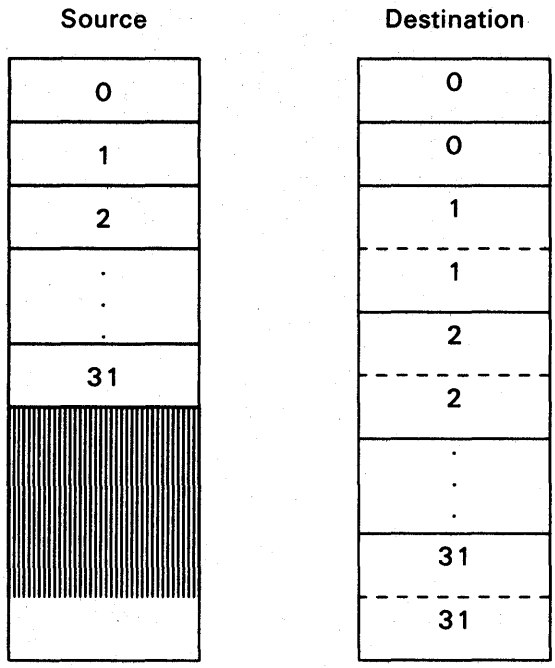


Figure 2-9. Type Conversion and Count Leading Signs Vector Arrangement

Figure 2-10 is a block diagram of the move pipeline and associated scientific processor sections that require information to initiate and execute an instruction requiring assistance from the move pipeline. The move pipeline consists of a move control and a move buffer. The move buffer is a temporary location for data transfers in and out of the vector files. The scalar vector control section controls movement of data between the move pipeline and the scalar processor for the GXV instruction. The scalar processor section reads the base and stride from an instruction specified G-register and generates the indexes for the GXV and transfers them to a vector file. (The indexes are actually transferred to the move buffer and then written into the vector file.) This transfer of data from the scalar processor to the vector file via the move buffer is controlled by the scalar vector control.

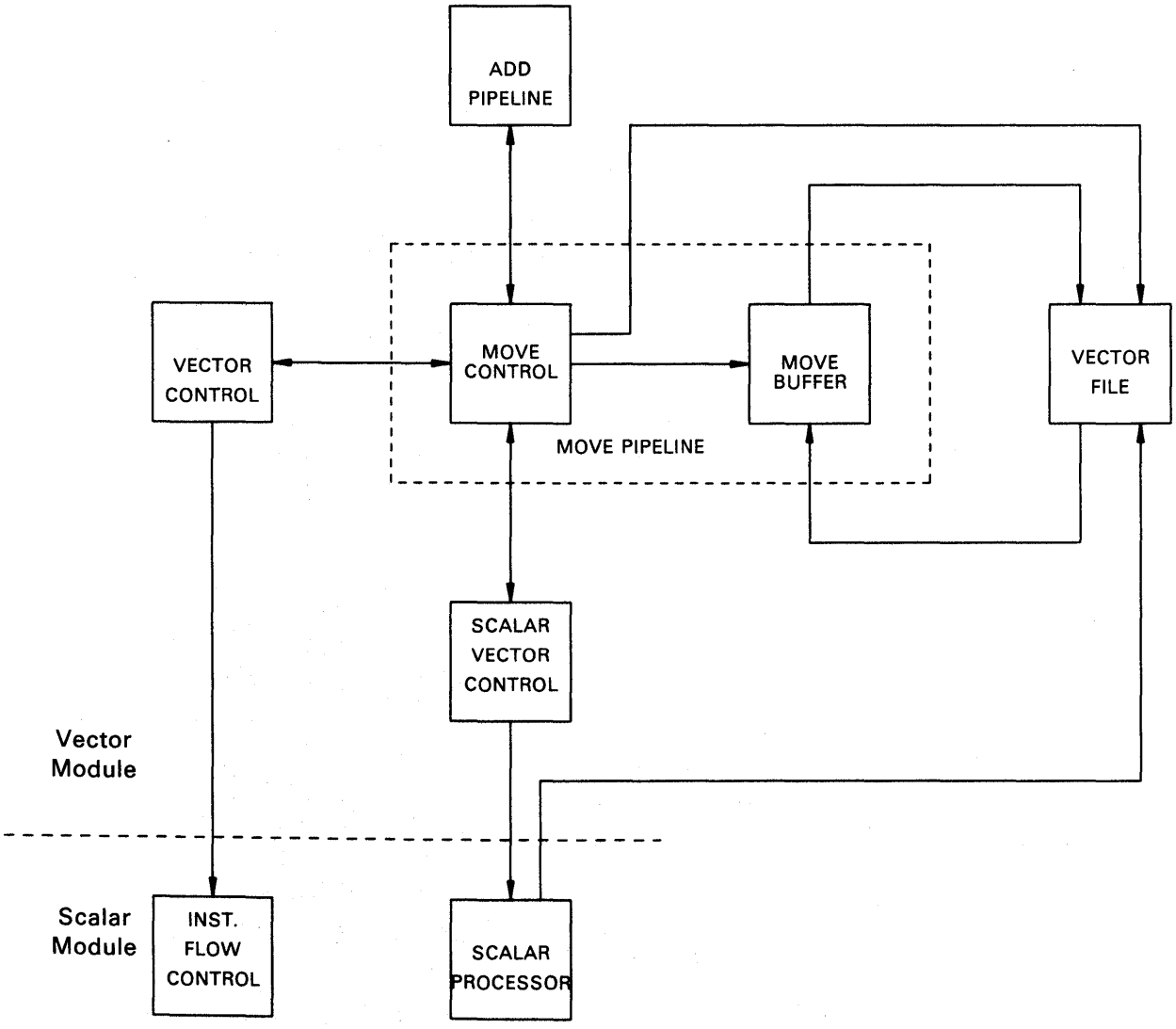


Figure 2-10. Move Pipeline to Vector Module Interface

2.4.5. Vector Multiply Pipeline

The vector multiply pipeline section executes multiply, divide, product reduction, population count, single-precision floating-point, double-precision floating-point, and single-precision integer instructions. It is basically a one-pass multiplier that provides a burst rate of two products per clock cycle for single-precision calculations and one product per clock cycle for double-precision calculations. Multiple passes are required to perform divide and product reduction instructions.

The following data types are provided for the arithmetic instruction types:

- single-precision integer
- single-precision floating-point
- double-precision floating-point

The multiply pipeline consists of sequence control, pipeline control, and pipeline data path subsections.

The sequence control subsection:

- receives and acknowledges instructions from vector control
- controls vector file addressing
- responds to vector file conflict notification
- sends control information to the pipeline control subsection.

The pipeline control subsection receives a control word from the sequence control subsection and stages it in parallel with the data passing through the pipeline data path subsection.

The pipeline data path subsection:

- receives a data stream from the vector file section, processes it as directed by the pipeline control, and presents the resulting stream to the vector file section
- sends write control pulses to the vector file section
- receives mask bits from the mask processor section
- receives and holds arithmetic fault interrupt mask bits from the control block section
- sends arithmetic fault indicator bits and the failing element pointer to the control block.

2.4.6. Vector Load

The vector load section executes the following instructions:

- Load Vector (LV, DLV)
- Indexed Load Vector (LVX, DLVX)
- Load Alternating Elements Vector (LAEV, DLAEV)

The vector load section receives load data from local storage or the scientific processor storage by way of local storage and transfers this data into the vector files. The data transfer between local storage and vector load is four words per cycle at a transfer rate of 133 megawords per cycle.

The vector load section supports single-precision and double-precision data types, and it has a Vector Load buffer that allows simultaneous read and write operations.

2.4.7. Vector Store

The vector store section moves data from the vector files to the store buffer section.

The vector store section executes the following instructions:

- Store Vector (SV)
- Store Vector, Indexed (SVX)
- Store Alternating Elements Vector (SAEV)
- Load Vector, Indexed (LVX)

Vector store controls the vector file primary and secondary store read registers that allow a four-word transfer per clock cycle to the store buffer section. The store buffer section then transfers the data to the scientific storage or to the local storage section.

Store Vector Instruction

The Store Vector (SV) instruction moves vector elements from the primary and secondary of a single vector file or from a G-register broadcast to replicate a vector element to storage locations in the scientific processor storage or local storage in the scalar processor module. The virtual storage addresses are contained in a G-register that is translated in the address generation section to a real address prior to execution. The instruction l-field selects the number of elements to be transferred. It can be from 1 to 64 per instruction.

The Store Vector instruction when performing a broadcast G takes one element from the specified G-register and replicates it across sequential addresses in storage for a length determined by the specified element count. An instruction requesting a broadcast G is handled exclusively by the scalar processor module as are Mask transfers that allow conditional transfers of certain elements. The SV instruction transfers either four single-precision (two double-precision) elements or two single-precision (one double-precision) elements per cycle to the store buffer.

Store Vector, Indexed Instruction

The Store Vector, Indexed (SVX) instruction transfers data from a vector file or a broadcast G register to a location in scientific processor storage or local storage exactly like the Store Vector instruction. The difference is that the SVX instruction uses an index quantity in addition to the virtual address in the SVX specified G-register. This index quantity is stored in the primary and secondary copies of the second Vector File requested by the SVX instruction. The base virtual address in the specified G-register is modified by adding the index value in the second vector file that corresponds to the same element in the first vector file that contains the index data. The SVX instruction transfers two single-precision elements or one double-precision element and two indexes per cycle (64 single-precision or 32 double-precision elements per instruction).

The SVX instruction uses 2 separate vector files; one for the index data and the other for the index. For each index data (operand) there is a corresponding index. This instruction is

essentially reading operand 1 out of a primary vector file and the index value for offsetting the address of operand 1 in storage is located in the secondary vector file. If the requested SVX instruction has an element count of 64 the Index Buffer secondary path is toggled to the alternate Index Buffer to accommodate the remaining 32 elements.

Load Vector, Indexed Instruction

The Load Vector, Indexed (LVX) instruction is the opposite of the SV instruction; the elements of a vector file are loaded from locations in storage (scientific processor storage or local storage). LVX specifies the vector file for the storage of the index data and the vector file that contains the index value. The index value in the LVX instruction vector file is transferred to the index buffer in the store buffer section for addition to the base address value to determine the real address of the index data location in storage. The vector store section transfers this index value to the address generation section for addressing. The vector load section loads the vector element.

Store Alternating Elements Vector Instruction

The Store Alternating Elements Vector (SAEV) instruction stores pairs of elements that are formed from corresponding elements of two vector files. These pairs of elements are stored in adjacent memory locations. In this case the element count specifies the number of element pairs that are transferred.

The SAEV instruction requires reading two separate vector files whose length is determined by element count. Element count specifies the number of element pairs that are to be transferred. For an element count of 64, 128 elements are transferred to the store buffers. Alternating pairs of elements are formed at the output of the store buffers in the store buffer section so that corresponding elements of the two vector files are stored in adjacent memory locations in storage.

Vector Store Interface

Figure 2–11 shows the relationship of the vector store section to the other sections that it interfaces with. Vector control receives the vector control word and vector parameter word from instruction flow control and loop control and initiates the request for the vector store pipeline to participate in the handling of data for the requested Vector instruction. Vector control also monitors for potential conflicts between vector store and the other pipelines and supplies the vector file addresses. Vector control informs vector store that it has an instruction that requires its participation and if a potential Vector File conflict exists between vector store and one of the other pipelines Vector control will suspend one of the pipelines temporarily. The vector store acknowledges the instruction requests from vector control and keeps track of where it is currently reading in the vector files and controls its own addressing of the vector files. The vector store feeds back to vector control any store buffer and index buffer conflict detection. The vector store also handles conflict counters and file registers control used in conflict detection. (The path between vector file and the store buffer section is the data path for the storage of 144 bits of data plus 24 bits of parity.)

Vector File Conflicts

When two of the active pipelines attempt to use the same vector file a potential vector file conflict is recognized and vector control suspends the operand sequencing of the pipeline whose current instruction was started most recently. This suspension is maintained in eight cycle increments to allow the affected pipeline to resume at the point of interruption when the conflict is resolved.

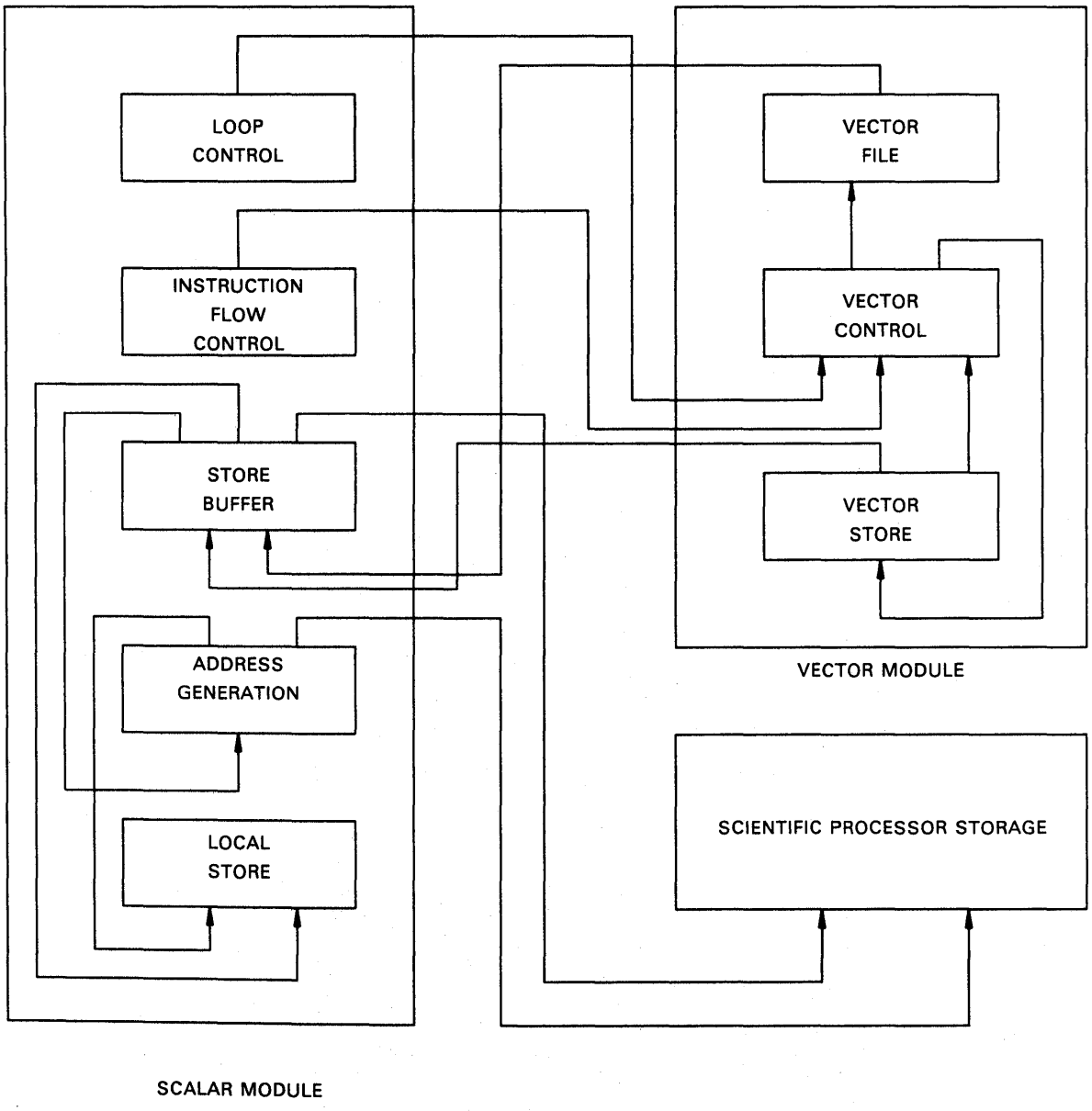


Figure 2-11. Vector Store Interface

2.4.8. Scalar Vector Control

The scalar vector control section provides the interface control for RR scalar format instructions and VV format instructions that move G-operand and vector operand data between the scalar and vector modules. It controls date transfers to the Move buffer in move pipeline for Generate Index Vector instructions and controls transfer of the results of reduction operations from multiply pipeline and the add pipeline to the scalar processor.

The scalar control contains the logic for controlling the data flow for instructions requiring data exchanges between the scalar and vector modules. Scalar control controls the flow of data to or from the scalar module and to the respective vector module pipeline (add pipeline, multiply pipeline, move pipeline) for instructions requiring G operands. This section also controls the flow of data to or from the scalar module and vector file for scalar instructions that require a vector operand.

G-Operand

The G-operand is used in the VV format vector instructions to create vector or scalar data for use in elementwise arithmetic functions. There are three general types of G-operands.

G-operands 1 and 2 are referred to as broadcast Gs and are used to create a vector for use by the add pipeline, multiply pipeline, or move pipeline. In a single G-operand operation, a value in a G-register becomes the source for one of the operands in a VV formatted vector instruction. This G-register value is used as a constant for one operand source and a vector file element is the other operand source. In a dual G-operand operation both operands are from individual G-registers in the scalar processor module. The G-operands transfer directly to the affected pipeline and are not stored in the vector file prior to the operation.

G-operand 3 is a reduction instruction that reduces a vector to one scalar value and sends it over the interface to the scalar processor module. The reduction instructions add or multiply the elements of a single operand and produce a single scalar result. This result transfers to the scalar processor module interface.

Scalar control acts as controller for data transferred over the scalar processor module/vector module interface for the GXV (Generate Index Vector) instruction and for the reduction instructions.

A GVX instruction is a variation of a G-operand where the G-register value is incremented by a fixed amount (stride) in the scalar processor module and scalar control is used to control the passing of these values to the move pipeline in the vector module. The values are transferred to elements in the vector file; with no intervening manipulation.

Scalar control acts as one of the six pipelines in vector module for instructions requiring V operands.

Vector Operand

The vector operand operation of scalar control is used with scalar instructions for controlling two types of RR format instructions: RR format scalar arithmetic instructions and RR format scalar moves.

The RR format scalar arithmetic instructions perform arithmetic operations in the scalar processor module using operands that originate as an element in the vector file or the operands element pointer are sourced from G-registers and the result is stored in the vector file. The element pointer is specified by the vector parameter word.

The RR format scalar move instructions transfer one vector file scalar element to another or between G-registers and a vector file scalar element. At least one operand is always in a vector file. Source and destination locations are specified in the vector control word. The data in these operations is unchanged only the location is changed.

The vector operand control is used exclusively for RR format instructions (vector operands) and the Interface Register control handles the scalar data exchange between scalar control and the scalar processor module as it does in the G-operand instructions.

Instruction Issue and Receive in vector control receives the vector control word from the instruction flow control section, decodes its fields, and disseminates the appropriate information to the affected pipelines and scalar control. Conflict Detect monitors for vector file conflicts and manages vector file access timeslot reservations. Vector File addressing generates the addresses to vector file memory 0.

3. Operations

This section describes activity switching, interrupt handling, and instruction conflict operations.

3.1. Activity Switching

There are seven mutually exclusive states defined for the scientific processor. If more than one state is active at the same time, the scientific processor is considered inoperative.

State	Function
Dormant	Non-executing mode where monitoring for a universal processor interface is done to begin an activity.
External	State transition decisions are made
Accelerate	Loads activity information from the scientific processor storage to the scalar module.
Decelerate	Store activity information back into the scientific processor storage.
Execute	Activity is executed
Inoperative	Entered when fatal errors are detected and can only be exited by a power-up clear signal.

The scientific processor uses activity switching to go from one internal machine state to another (dormant to execution or vice versa). Activity acceleration, used during activity start-up, is the process of:

- loading the scientific processor with information from the control block;
- loading the internal programmable registers from the register save area; and
- loading information into local storage.

Activity deceleration at activity termination is the reverse of this process.

Initially a scientific processor is in a dormant (idle) state, in which it executes no instructions and makes no storage references. To place an activity onto a dormant scientific processor, the instruction processor sets up the appropriate activity data structures, places the control block real address into mailbox word 2 with a control code of 3, and signals the scientific processor on the UPI.

Upon receipt of the universal interface interrupt the scientific processor reads the mailbox and proceeds to switch in and execute the activity. It continues executing until the occurrence of an external interrupt. (In the context of scientific processor interrupts external and internal refer not to the interrupt source, but to the type of response required.) At that point the scientific processor commences to switch out that activity, notifies the instruction processor through the UPI, and returns to the dormant state.

The scientific processor can receive initial program load and reset clear signals any time regardless of its state. Response to them takes precedence over normal state switching. The scientific processor responds by immediately ceasing all storage references, clearing itself internally including the hardware status registers, and becoming dormant.

3.1.1. Acceleration

When no activity is being executed, the scientific processor remains in a dormant state until a universal processor interface interrupt is received from the instruction processor. Upon receipt of the interrupt, the scientific processor begins loading a new activity that is scheduled by the instruction processor.

Communications between the scientific processor and the instruction processor is accomplished through the mailbox (2.2.1). The most significant bit in word two of the mailbox indicates whether a valid activity is available. If this bit is set, the scientific processor takes the new activity address from word two of the mailbox and begins loading data. If the control code was not valid the Dormant state is reentered.

The following items are loaded for a new activity:

1. Register Save Area
    a. General registers
    b. State registers
    c. Loop Control registers
    d. Vector registers
    e. Jump History File
2. Scientific Processor Control Block
    a. Words 8–15
    b. Activity Segment Table
3. Local Storage

After completing these loads the scientific processor begins executing. If hardware errors occur during acceleration, the acceleration is immediately aborted.

3.1.2. Deceleration

The scientific processor continues to execute an activity until an External interrupt is received. This interrupt forces the scientific processor to decelerate, which stores all pertinent data back to the scientific processor storage.

There are five possible sources of interrupts:

1. Generate Interrupt instruction executed by activity instruction stream when an activity is completed.

2. Universal processor interface from the instruction processor to abort an activity.

3. External interrupt caused by a fault.

4. Internal interrupt that was masked to be handled as an External interrupt.

5. Internal interrupt handler that issues a Generate Interrupt instruction.

The data registers and files that are stored in the scientific processor storage during deceleration are the same as those loaded during acceleration (except the activity segment table is not decelerated).

When deceleration is completed, the scientific processor enters the Dormant state.

Hardware errors (Parity and the scientific processor storage Interface errors) that are detected during deceleration causes deceleration to be aborted and only the scientific processor control block (words 8–15) registers are stored in the original scientific processor control block and mailbox area in the scientific processor storage.

3.1.3. Activity Switch Algorithm

Activity switching is interrupt driven and is governed by the state of the scientific processor and type of interrupt encountered.

3.1.4. Special Considerations

When accelerating an activity, control block words 8–15 must be accelerated first, and when decelerating an activity, control block words 8–15 must be decelerated last, so that the Interrupt Indicators include all interrupts received up to that point.

If an error occurs or an interrupt is received while accelerating an activity, the scientific processor aborts the acceleration. Bit 33 of control block word 13 is set to indicate that no instructions have been executed. Control block words 8–15 are decelerated.

If any scientific processor hardware check (hardware status register 0, bit 2) or any storage check (hardware status register 0, bits 6–12) occur while decelerating an activity, further deceleration of that activity is abandoned. Bit 34 of control block word 13 is set to indicate that deceleration was aborted, and the algorithm goes immediately to store the control block.

If part of the scientific processor internal state is destroyed while performing a rapid deceleration, the scientific processor sets control block word 13, bit 35. The internal state is destroyed when it is overlaid with the state information. Bit 35 set indicates that reconstruction of state is required and it can be done from a combination of information from scientific storage and its internal registers.

If the scientific processor encounters errors while in the dormant state, it sets its indicators in hardware status registers 0–3. If these errors cause it not to start, the errors are reported in mailbox words 4–7 when the next universal processor interface interrupt is received. Words 8–15 of the scientific processor control block reports status of an activity, and words 4–7 of the mailbox use this same format to report status of the hardware. (Words 4–7 of the mailbox are pertinent only if the scientific processor is dormant and the instruction processor only reads them as part of universal processor interface processing.)

If mailbox word 4 is not identical to control block word 8 following deceleration, an error has occurred while writing control block words 8–15 or mailbox words 4–7.

3.2. Interrupt Handling

Interrupts that occur on the scientific processor are classified into two types: external (hardware serviceable interrupt) and internal. External interrupts include certain contingencies such as non-recoverable program faults and hardware errors that must be handled by the instruction processor. Internal interrupts include others such as arithmetic overflows and divide faults that may optionally be handled on the scientific processor.

3.2.1. Interrupt Responses

The response to an external interrupt consists of switching out the activity. The process of decelerating state stores a status word indicating the cause of the interrupt into hardware status register 0, and additional status is placed in hardware status register 2–4. The return address program address register value is stored into scientific processor control block word 12. When an activity is switched onto a scientific processor it always begins executing at the address in control block word 12.

The response to an internal interrupt is determined by the Internal Interrupt Control Mask located in State register 11. Bits 0–17 of State register 8 are the indicators for the 18 possible internal interrupt causes (see 3.2.5). Each of the possible 18 internal interrupt causes is represented by a corresponding 2-bit field in the mask.

State register 11 contains the Internal Interrupt Control Mask. For each of the interrupts listed in 3.2.5 there are two bits in the mask word, which together control how the interrupt is to be handled. Mask i bits and i+18 control the indicator bits as follows:

i-field	i+18	Interrupt Response
0	0	Ignore entirely (do not change State register 8)
0	1	Cause asynchronous external interrupt
1	0	Cause synchronous internal interrupt
1	1	Cause synchronous external interrupt

Basically, they allow each interrupt to be: a) ignored, b) taken as if it were external, that is, by causing an activity switch, or c) taken internally. If the synchronous internal interrupt is selected, the response is essentially a forced branch to the virtual address in control block word 7, with the return address saved in State registers 12 or 13.

If mask bits 4 (Undefined Instruction) and 22 (reserved) are 0, hardware reaction is the same as it is when bit 4 is 0 and bit 22 is 1.

3.2.2. Interrupt Identification

Interrupts are identified by interrupt indicator bits. Hardware status register 0 contains the External Interrupt indicators, and State register 8 contains the Internal Interrupt indicators.

When an interrupt occurs that cannot be ignored, further issuing of instructions is suspended, however, other operations in process are allowed to finish. When a quiescent state is reached, the interrupt indicators are examined. If there is only one interrupt present, then the appropriate status is stored and the interrupt is taken as previously described.

It is conceivable that more than one interrupt could be indicated. That would result if a second interrupt arose while trying to complete the other operations, or if an interrupt from an outside source (for example, a universal processor interface) happened to coincide with one generated locally. If the interrupts are only internal interrupts, a jump is made to the virtual address in word 7 of the scientific processor control block, and the return address is saved in State register 12, except on a timer interrupt the return address is saved in State register 13. The interrupt handling software tests the indicators to determine the number and identity of the interrupts. If all interrupts are external, operation is analogous except that, of course, an activity switch occurs.

Should both internal and external interrupts be present, both internal and external status and indicators are set. Also, bit 35 of hardware status register 0 (Pending Internal Interrupt flag) is set and the external switch is taken. Upon being switched back in at a later time the flag is checked by the scientific processor before executing any instructions; if set, the internal interrupts are then taken and the flag is cleared by the hardware.

NOTE: In all cases the interrupt indicators must be cleared by software handlers.

When an activity is switched onto a scientific processor, word 8 of the control block is tested. If any bit 0–34 is set, the external interrupt is immediately taken, causing the activity to be switched back out without executing any instructions. If bits 0–34 are clear but bit 35 is set, the internal interrupt is taken as previously described. The Internal Interrupt indicators (State register word 8) are not tested by the scientific processor hardware. They themselves do not cause interrupts but are merely status bits set by hardware for use by software.

3.2.3. External Interrupts

Hardware status register 0 contains the External Interrupt indicators in bits 0–34, and bit 35 is the flag that indicates a pending internal interrupt. The occurrence of each external interrupt condition is reflected by setting the corresponding bit in hardware status register 0.

Bit	External Interrupt Cause Indicated
0	Mailbox Valid bit not set or control block boundary violation
1	Critical environmental fault (power loss, coolant loss, etc.)
2	Scientific processor hardware check
3,4	Reserved
5	IPL Reboot Interrupt received
6	Error on information from storage
7	Error on information to storage
8	Multiple uncorrectable errors in storage
9	Real address not available
10	Storage internal check
11	Interface sequence error
12	Reserved
13	Storage interface timeout error
14,15	Reserved
16	An internal interrupt is taken as external
17	Generate Interrupt (GI) instruction
18	UPI interrupt received
19	Quantum timer runout
20	Program segment alignment or length error
21	Register save area not on correct storage boundary
22	Local storage base address not on correct boundary
23	Local storage length granularity incorrect
24	Program address register fault (activity segment table entry not found)
25	Data address fault (activity segment table entry not found)
26	Address limits error
27	Storage protection check - Execute
28	Storage protection check - Read
29	Storage protection check - Write
30	Data Alignment error
31	Attempted Test and Set/Clear or local storage segment
32	Program address register bits 18,19 not zero
33,34	Reserved
35	Pending Internal Interrupt flag

3.2.4. Internal Interrupts

Following are the definitions of the Internal Interrupt indicators, which occupy State register word 8. Bits 0–17 are Internal Interrupt indicators and bits 30–35 store an element count if an internal interrupt occurs during a vector instruction.

Bit	Internal Interrupt Cause Indicated
0	Interval Timer Runout
1	Instruction Breakpoint Compare
2	Local Storage Stack Overflow
3	Local Storage Stack Underflow
4	Undefined Instruction

Bit	Internal Interrupt Cause Indicated
5	Vector Register Length Overflow
6	Divide Fault
7	Integer Overflow
8	Floating Characteristic Overflow
9	Single–Precision Characteristic Underflow
10	Double–Precision Characteristic Underflow
11	Data Breakpoint Compare – Read
12	Data Breakpoint Compare – Write
13	Remaining Length Overflow (Build Vector Loop)
14–17	Reserved for additional internal interrupts
18–29	Reserved
30–35	Element index status

Stack overflows simply continue counting and wrap around eventually. These cases are not particularly useful. The terms synchronous and asynchronous have special meanings in this context, and are defined 3.2.5.

For an Interval Timer Runout interrupt, the return address is placed into State register 13. For all other internal interrupts, the return address is placed into State register 12. If both kinds of interrupts occur simultaneously, both words are written. A Timer interrupt never alters the status (interrupted instruction) in State register 9.

3.2.5. Interrupt Synchrony

Vector operations generally take longer to perform than scalar operations and usually use different hardware. Because of this, the scientific processor permits scalar instructions following a vector instruction to be executed as soon as the vector operation is initiated, rather than waiting for it to be completed. Registers ensure that the functional effect of this is no different than if instructions had been executed singly and completely.

This asynchronous initiation of scalar instructions following a vector instruction can cause the interrupt return address captured to point to any one of many instructions (including jumps) beyond this vector instruction if a fault occurs during execution of the original vector instruction.

The scientific processor provides two operational modes for program execution a fast mode and slow mode. It is constrained to run in slow mode any time one or more of the following bits is set; bits 5–13 of the Internal Interrupt Mask, or bit 0 of control block word 6 (called the speed control bit). If none of these is set, then the scientific processor is permitted to run in fast mode.

NOTE: The interrupt causes indicated by bits 5–13 can be recognized without forcing slow mode by setting the corresponding asynchronous control bits (bits 23–31).

The scientific processor must have overlap disabled to capture arithmetic faults synchronously. Most other faults that need to be captured synchronously are captured during overlapped execution. For those external faults that are not captured synchronously, the general speed control is provided.

NOTE: Synchronous and slow mode are not synonymous. Synchronous refers to the way an interrupt is handled, and slow is a processor mode that may or may not be required.

The following paragraphs define synchronous and asynchronous interrupt handling.

When a synchronous internal interrupt occurs, the return address points to the instruction that logically follows the instruction that caused the interrupt. The instruction pointed to, and those after it, have not been executed. Therefore a jump to the return address following the interrupt servicing causes correct program resumption. In this situation, State register 9 holds a copy of the instruction that caused the interrupt. If it is an Elementwise Vector instruction, bits 30–35 of State register 8 hold the index of the element whose computation caused the interrupt. Elements preceding that point will have been correctly performed, and operands for the element whose computation caused the interrupt and those past that point will not have been altered.

A synchronous (overlapped) internal interrupt may be handled either internally or externally. In either case the status and return addresses are stored in State registers 8, 9 and 12 or 13. For external handling, the return address is also placed into control block word 12 and bit 16 of hardware status register 0 is set to one.

For asynchronous handling of an internal interrupt (internal interrupt taken as external) the contents of State registers 9, 12, 13, and bits 30 to 35 of state register 8 are not defined. The address in control block word 12 points to the next logical instruction to be processed.

Of the external interrupt types, the GI (bit 17) and the Address Faults (bits 24,25) are required to appear synchronous regardless of machine mode. For the GI and Data Address Fault, the instruction causing the interrupt is copied into hardware status register 1, and the return address points to the next logical instruction. For a Program Address Fault, the register contains the faulting address, and hardware status register 1 is not defined.

In fast mode all other external interrupts appear asynchronous. For them hardware status register 1 is not defined, but the Program Address register contains the valid return address (next logical instruction).

In slow mode all external interrupts that are caused by scientific processor instructions become synchronous. For them hardware status register 1 contains the instruction causing the interrupt and the Program Address register points to the next logical instruction.

If an interrupt is caused by the attempt to read an instruction, the return address in the Program Address register points to the address where the read attempt failed. However, if an interrupt is caused by the attempt to execute an instruction that was successfully read and the interrupt is to be taken synchronously, the return address is one plus the address of the interrupting instruction.

No attempt is made to completely synchronize the Internal Interval Timer when bit 0 of the Internal Interrupt Control Mask is set. Instead, fast mode is still permitted, and the interrupt appears asynchronous. It appears that fast mode is more desirable than perfect synchrony when taking this interrupt internally. However, if the processor is constrained to slow mode due to some other cause, then this interrupt will also appear synchronized.

3.2.6. Interrupt Status

In either fast or slow mode, additional status may be placed into hardware status registers 2 and 3 by the scientific storage unit and scientific processor parity and hardware checks.

3.2.7. Internal Interrupt Handling

Because the scientific storage is not directly addressed, internal interrupt handlers need a free G-register on which to base their data areas. State register 3 is specifically reserved as the save area for the G-register that is used for internal interrupt handler addressing. Only the left half of the G-register may be stored in State register 3.

3.3. Instruction Conflict Classification and Types

Instruction conflict classification is based on the different phases of instruction execution. As each phase of execution is entered, a specific type of conflict may suspend instruction execution. Suspension, within an instruction, may occur at any time within a phase of execution. The resolution of the conflict may allow the instruction to enter the next phase of execution or to resume execution within the current phase of execution. Instruction conflicts are grouped into the following classes:

- Instruction Issue Class
- Control Word Dispatch Class
- Execution Class

Figure 3-1 shows the instruction conflict classes and conflict types.

3.3.1. Instruction Issue Class

The Instruction issue class of conflicts are detected and resolved by the instruction flow control section in the scalar processor. Each instruction is analyzed to determine if any conflicts exist that must be resolved before the instruction is executed in the vector processor or the scalar processor. The existence of no conflicts allows the instruction to be executed so any conflicts associated with the execution are detected in the next phase of execution. The instruction issue class of conflicts are caused by the following types of conflicts:

- Destination/source register conflict
- Source/destination register conflict
- Vector control word queue (facility conflict)

3.3.2. Control Word Dispatch Class

Once an instruction has been placed into execution (issued), the control word dispatch class of instruction conflicts are detected by the vector control, instruction flow, or scalar processor sections. During this phase of execution the availability of facilities and operands is examined. When the control word is dispatched the instruction is moved into its final phase of execution.

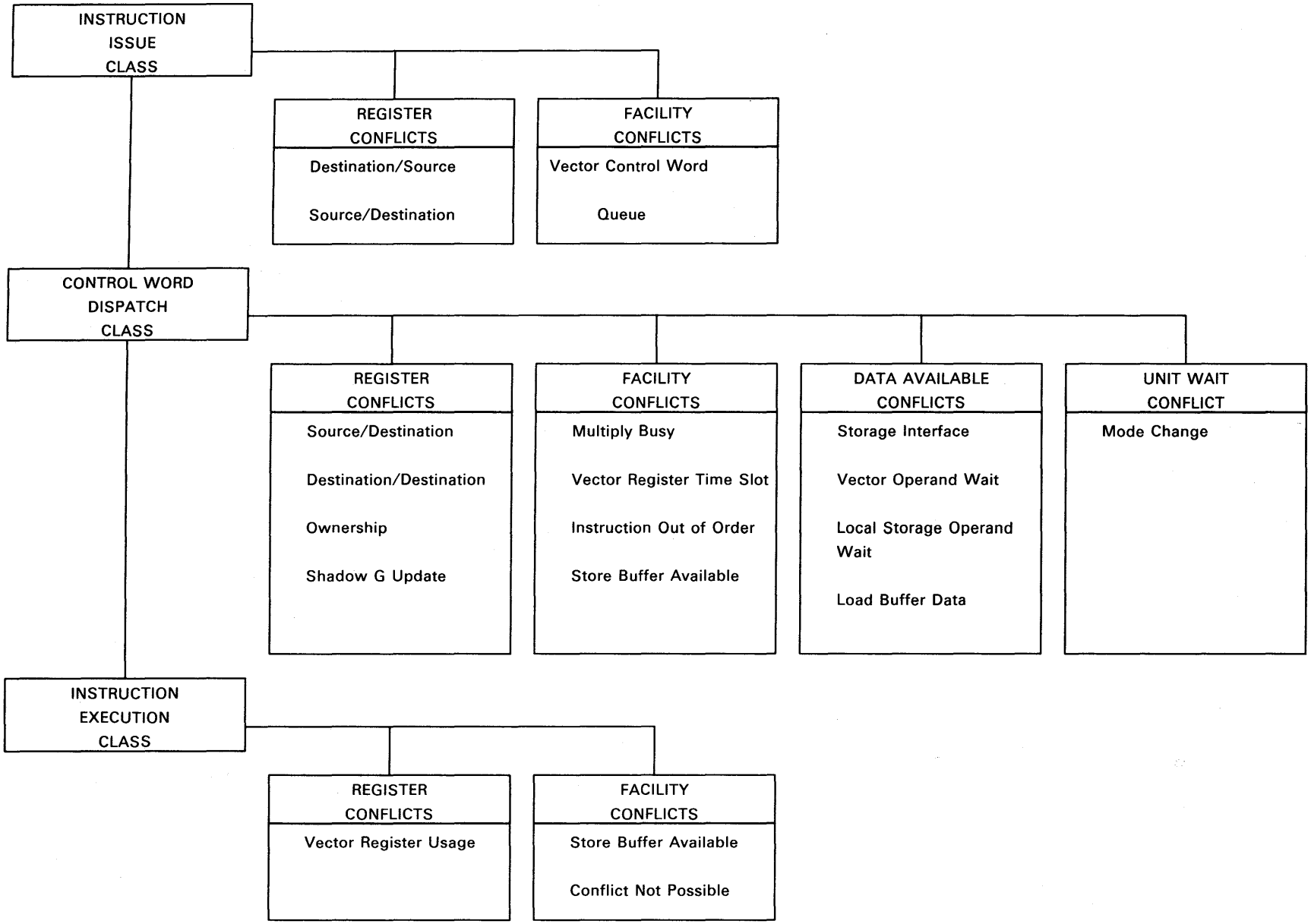


Figure 3–1. Instruction Conflict Classes and Conflict Types

The control word dispatch class of conflicts are caused by the following types of conflicts.

- Register conflict
- Facility conflict
- Data available conflict
- Unit wait conflict

3.3.3. Instruction Execution Class

The instruction execution class of conflicts are detected and resolved during Vector instruction execution. Detection does not start until the vector control section has dispatched the instruction to one of the following pipelines for execution:

- Add Pipe
- Multiply Pipe
- Move Pipe
- Vector Load
- Vector Store
- Scalar Control

The instruction execution class of conflicts are caused by register conflicts and facility conflicts. Register conflicts are detected by the vector control section at any point during instruction execution. Facility conflicts are detected by the pipeline at specific points of execution. All are resolved by the pipeline executing the instruction.

3.3.4. Register Conflicts

Instruction execution is suspended when one of the following register conflicts occurs:

Destination/Source

A register destination/source conflict occurs when a previously issued instruction is using the information stored in the same register that an instruction to be issued or waiting to continue execution is referencing.

Source/Destination

A register source/destination conflict occurs when an instruction to be issued or waiting to continue execution is transferring information from the same register that is being referenced by a previously issued instruction.

Destination/Destination

The register destination/destination conflict occurs when an instruction to be issued or waiting to continue execution is referencing the same register that is being referenced by a previously issued instruction. This conflict detection ensures that a register ends up with the corrected results when instructions complete out of the order they were issued.

Ownership

A register ownership conflict occurs when an instruction requires a working register for execution that is in use by a previously issued instruction. When this conflict occurs the order of register ownership priority is the order in which the instructions were issued.

Vector Register Usage

An instruction attempts to use or alter a Vector register element before some prior instruction is finished with it.

3.3.5. Facility Conflicts

Facility conflicts occur when an instruction is delayed because a previously issued instruction is in the logic section. The following are the specific types of facility conflicts:

Vector Control Word Queue

The vector control section holds an instruction, because the pipeline that executes it is busy with a prior instruction (or because of some other facilities conflicts). The instruction flow control section may issue another instruction to the vector processor, however this one is held by the instruction flow control section until the vector control is ready to accept it. Both instruction flow control and vector control need to detect this situation and hold an instruction.

Multiply Busy

The multiply busy facility conflict occurs in the scalar processor when the multiply logic is executing a scalar multiply instruction, and a second instruction is waiting to use the multiply logic.

Vector Register Time Slot

The vector register time slot reservation mechanism in the vector control section is used as each instruction is executed. If the required time slots are not available, then the instruction is held in vector control.

Instruction Out of Order

Instructions for the vector processor are issued by the instruction flow control section in the scalar processor in program order. Sometimes, the vector control section initiates those instructions into execution out of program order. When this occurs, there cannot be a conflict in the bypassed instruction.

Store Buffer Available

The store buffer available facility conflict is used by all instructions that the vector store section executes. The vector store section detects that the Store/Index buffer in the scalar processor module does not have address space available to transfer data from the vector register.

In the scalar processor, multiple store operations are allowed to complete when the Store Buffer is available.

Conflict Not Possible

The conflict not possible facility conflict determines that an instruction cannot have a vector register logical data usage conflict with some prior instruction. The five pipelines use this to determine when the vector register time slot reservations can be released. The early release of the vector register timeslots is necessary to allow proper overlay operation of subsequent instructions.

3.3.6. Data Available Conflicts

Instruction execution may be suspended because the operand requested at instruction issue has not arrived. Operand wait conflicts may occur because of one of the following:

Storage Interface Operand Wait

This conflict is the result of the scalar processor waiting on an operand from the scientific storage unit.

Vector Operand Wait

This conflict in the scalar processor occurs because a RR format instruction requiring an element from a vector register as an operand or a Reduction instruction results to a G-register.

Local Storage Operand Wait

This conflict occurs in the scalar processor when the results of a Store instruction to local storage is required as an operand by the instruction following the store.

Load Buffer Data

Certain required data for an instruction is not yet available. This is not a conflict between instructions, rather it is because two (or more) asynchronous (to each other) sections of logic participating in the execution of an instruction. This mechanism is used by all instructions that the vector load pipeline executes. The vector load section detects that data is not available in the load buffer to be transferred into the Vector registers. An example of how the conflict can occur is an interruption in the data from the scientific processor storage due to system contention.

3.3.7. Unit Wait Conflicts

The unit wait conflict occurs in the scalar processor when issued instruction may change the mode of the scientific processor. Under this condition the processor must reach a quiescent state before the instruction can continue execution.

4. Scientific Processor Instructions

This section describes the instruction word formats and the instruction set.

4.1. Introduction

The scientific processor instruction set consists of scalar, vector and control instructions. The scalar instructions compute or move one or more scalar data operands. A two-operand instruction format specifies operations for the scalar instructions. The vector instructions compute and move one or more vector operands located in vector registers. The vector instruction operands are specified by a three-operand instruction format. The scientific processor control instructions include: loop control, jump, and state instructions.

In vector computations the first element of one source vector is operated on together with the first element of the other source vector to produce the first element of the result vector. Then the second element of the sources are operated on to form the second element of the result vector, and so on. Necessarily both source vectors must be of the same length, and the operation produces a result vector of that length.

The vector instructions specify a length parameter to allow faster operation in those cases where fewer than the maximum number of elements in a file are to be processed. The two-bit l-field uses (see VV format in 4.2) values 0-3 respectively to select ELCNT, NELCNT, ALT1 and ALT2 from the Vector Loop register identified by the current value of CVLP. The selected parameter specifies the number of elements to process. Processing begins with the first elements and proceeds for the designated number of elements.

NOTE: Zero is a valid length parameter value, and indicates no elements.

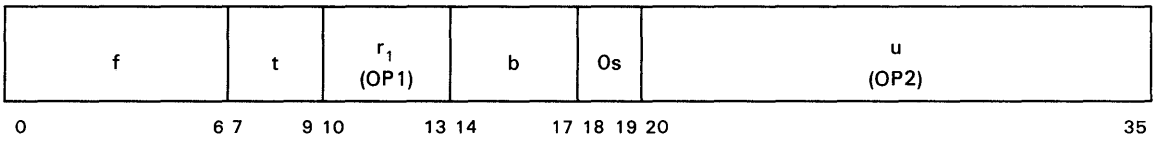
4.2. Instruction Word Formats

All scientific processor instructions are 36 bits in length and have three basic formats: register-to-storage (RS), register-to-register (RR), and vector-to-vector (VV). The common fields of all formats is described followed by a description of each format. Detailed variations are included with each instruction definition in Section 4.3 through 4.19.

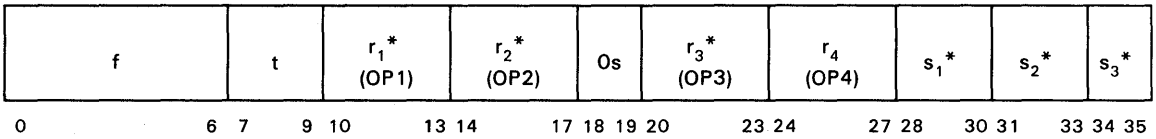
Bit positions in the instruction diagrams not used by the instruction on non-vector instructions require that zeros be put in these positions. Correct operation is not guaranteed should a non-zero appear in any position. For vector instructions, in which the three-bit s_1 or s_2 fields are not used, a value of 001 should be encoded into each 3-bit s-field as shown in 4.2.1.

The three formats and their respective fields are:

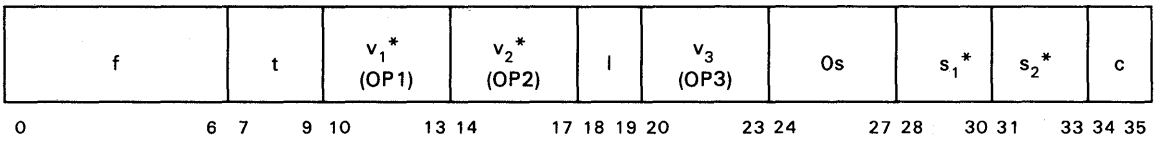
RS Format



RR Format



VV Format



* *The s-, r- or v-fields with corresponding subscripts operate together.*

4.2.1. Common Fields

The common instruction fields are defined as follows:

f-field

The f-field (function code) specifies the operation to be performed, and also provides additional definition for the other instruction fields.

t-field

The t-field specifies the data type. In general, even values indicate single-precision data and odd values double-precision data. All operands are assumed to be of the specified type, with logical and arithmetic operations performed according to that type. Numerical formats are standard Series 1100 ones-complement.

t-field	Definition
0	Single-precision integer
1	Double-precision integer
2	Single-precision floating-point real
3	Double-precision floating-point real
4-7	Reserved

Some instructions use a one- or two-bit t-field rather than a three-bit t-field. Some other instructions need additional data and assign special meanings to those bits of the t-field not required to specify size or data type.

With the exception of a floating-point add, all floating-point input operands must be in correctly normalized form. When normalized inputs are used, results are in correct normalized form. The normalized form of zero is all zeroes for the sign, characteristic, and mantissa. An unnormalized input operand produces undefined results. An exception occurs during floating-point adds. When a normalized 0 is added to an unnormalized operand, the result will be normalized.

s-field

The s-field value selects the associated operand as being a part of a particular register set (G-register, S-register, or the vector register set). In many instructions only a subset of the eight possible s-field values is allowed. In those cases the s-field is shown with less than three bits.

r-field

The r-field value selects the appropriate register number within the register set implied by the instruction or the s-field select.

v-field

The v-field value selects the appropriate vector file number within the vector file set implied by the instruction or the s-field select.

l-field

The l-field value selects the source for the maximum number of elements to be processed according to the vector instruction. Values 0 through 3 select the element count, next element count, first alternate element count, and second alternate element count respectively. These values are based on the contents of the Vector Loop register pointed to by the current value of the Current Vector Loop Pointer (CVLP).

c-field

The c-field value specifies how the 64 single-precision or 32 double-precision Mask register bits are used.

If the c-field value equals 0 or 2, write operations to all vector file elements used by an instruction are controlled by that instruction, that is, mask data has no effect.

If the c-field value equals 1, a WRITE ENABLE signal is active only for those elements that have 0 in their associated bit positions in the Mask register.

If the c-field value equals 3, a WRITE ENABLE signal is active only for those elements that have a 1 in their associated bit positions in the Mask register.

Operations disabled by the Mask control do not produce results, do not generate interrupts (for example, zero divide, address faults, etc), and do not alter the corresponding destination vector file location.

4.2.2. Register-to-Storage (RS) Format

The RS format is used for most scalar operations. The b- and u-fields together specify a storage location for operand 2 (OP2) in either local storage or scientific storage. The r_1-field selects a register from the G-register set to be used as operand 1 (OP1). Arithmetic and logical operations generally consist of combining the storage operand (OP2) and the r_1-field selected operand (OP1) and storing the result in the r_1-OP1 location. The u-field is used as the offset in the storage space selected under control of the b-field interpretation to locate OP2.

The b-field value selects either local storage or the scientific storage as the address space for the OP2 location. If the b-field is 0, then the u-field forms the 16-bit offset into the local storage segment. If the b-field is 15, then the 11-bit Pointer field of State register 7 has the five rightmost bits of the instruction u-field catenated on its right to form the local storage segment address.

When the resultant offset for b=0 or 15 is less than 4K, the local storage is referenced; otherwise the scientific storage is accessed using the activity segment table 0 base plus offset.

If the b-field value is 1 through 14, the u-field value is treated as a non-negative offset and to it is added the 36-bit virtual address from the G-register specified by the b-field to form the target virtual address. In order for the reference to be valid, both the target virtual address and the original virtual address in the G-register must be translated to a real address by the activity segment table entries and also meet the access permission conditions.

4.2.3. Register-to-Register (RR) Format

The RR format is used for scalar instructions that do not specify a storage location local storage or scientific storage as any of the three operands. The source operands and the destination are always registers or Vector files. This format permits three distinct locations for the three operands of a dyadic operation. Each of the three operand locations is specified by a corresponding s-field and r-field. The s-field value selects the set of registers or files and the related accessing information, and the r-field specifies the register number within the s-field selected set. The fourth r-field provides additional specification in certain special instances.

The s-field definitions vary somewhat from one class of RR format instructions to another, and usually only a subset of all possible values is defined for a particular instruction.

4.2.4. Vector-to-Vector (VV) Format

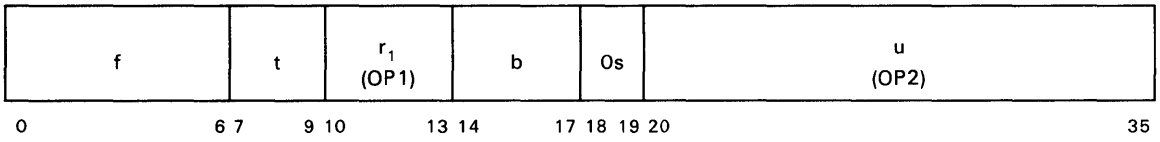
The VV format is used to specify vector operations. Operand 2, specified by s_2 and v_2, is combined with operand 1 (s_1, v_1), and the result of each individual element within the vector is placed into the operand 3 location.

The s_1- and s_2-fields each specify one of two possible ways for obtaining the corresponding source vector. For an s-field value of 1, the elements of the vector are taken from successive locations of the vector file specified by the corresponding v-field. For an s-field value of 0, the scalar value from the G-register specified by the v-field is broadcast, that is it is replicated internally to create an artificial vector of identical values. For vector computations the result is always a vector (even if $s_1=s_2=0$). It is placed in the vector file specified by v_3 (an s_3-field is not required for this result).

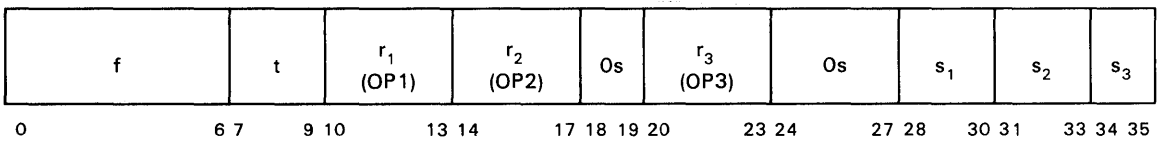
4.3. Scalar Arithmetic Computational Instructions

The scalar arithmetic computational instructions provide instructions to perform addition, subtraction, multiplication, division, and absolute value arithmetic operations. These instructions use the following register-to-storage (RS) and register-to-register (RR) formats:

RS Format



RR Format



1. Common fields (f-, t-, r-, s-, and v-) are defined in 4.2.1.

2. OP1=operand 1, OP2=operand 2, and OP3=operand 3.

In the RS format, OP1 is the G-register specified by the r_1-field, and OP2 is the storage location specified by the b-and u-fields. In the RR format, OP1, OP2, and OP3 are each specified by a 4-bit r-field and 1-bit of the s-field numbered accordingly. A value of 0 in the s-bit specifies the G-register selected by the r-field. A 1 bit in the identified s-field specifies a vector file element whose file number is given by the r-field and whose element number is given by element pointer of the loop control stack entry pointed to by the current element loop pointer. An element pointer value of 64 or greater for single-precision operands, or 32 or greater for double-precision operands causes a vector register length overflow.

In both formats OP1 and OP2 are sources. In the RS format, OP1 is also the destination for the result. In the RR format, the destination is OP3. Single-precision G-register operands are always assumed to be in the left half of the register.

The t–field for this instruction type has the following definition:

t–field	Definition
0	Single–precision integer
1	Double–precision integer
2	Single–precision floating–point real
3	Double–precision floating–point real
4–7	Reserved (causes undefined instruction interrupts)

The t–field applies equally to all source and destination operands unless otherwise specified.

4.3.1. Add (A, DA, FA, DFA, AR, DAR, FAR, DFAR)

Adds OP1 and OP2 source operands together and places the result in the destination location (either OP1 or OP3). This operation is defined for all values of the t–field.

The result is of the same precision as the sources; there is no residue word. In floating–point operations, a result of all 0s (sign, characteristic, and mantissa) is forced if the mantissa value is computed as +0 or –0. For integer operations, ones complement, subtractive addition is used so that a –0 result can be produced only when both source operands are –0. On an overflow, if the interrupt is not enabled, the truncated value is the result.

4.3.2. Add Negative (AN, DAN, FAN, DFAN, ANR, DANR, FANR, DFANR)

Subtracts the OP2 source operand from OP1. This is done by subtractive addition of the ones complement of OP2 to OP1. In all other respects the operation is similar to Add.

4.3.3. Multiply (MSI, MI, FM, DFM, MSIR, MIR, FMR, DFMR)

Forms the product of the source operands and places it in the destination. This operation is similar to scalar add, except that a standard double–precision integer is not supported. Instead, a t–field value of 1 specifies a form that the source operands are accepted as single–precision integers and the product is a double–precision integer. This operation is not defined if both s_3 and either s_1 or s_2 are 1 in the RR format.

NOTE: Single–precision integer type (t=0) produces a single–precision product; a value that exceeds single–word capacity causes an overflow.

4.3.4. Divide (DSI, DI, FD, DFD, DSIR, DIR, FDR, DFDR)

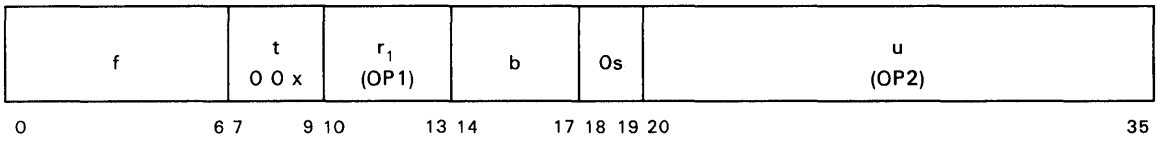
Divides the OP1 source operand by OP2 with no remainder produced. All operands are of the same type and precision except when the t–field equals 1. A standard double–precision integer is not supported.

In the t-field equals 1, case the dividend is accepted as a double-precision integer, but the divisor and quotient are single-precision integers. If the divisor is not zero but smaller in magnitude than the dividend by a factor of 2^{35} or more, then an integer overflow fault occurs. If the fault is ignored, the result stored is not defined. Double-precision integer operations (t=1) are undefined if s_1 and either s_2 or s_3 equal 1 in the RR format.

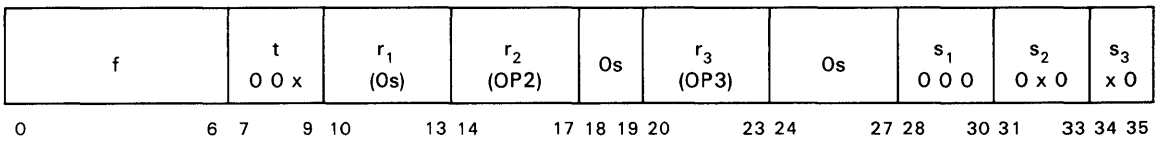
4.3.5. Absolute Value (EM, DEM, EMR, DEMR)

The OP2 source is examined for its algebraic sign. If it is non-negative, the source value is copied to the destination. If the sign is negative (including -0), the complemented source value is copied to the destination. Destination is OP1 in RS format, OP3 in RR format. In the RR format, OP1 is not used, and the r- and s-fields should contain 0s. These instructions use the following RS and RR formats:

RS Format



RR Format

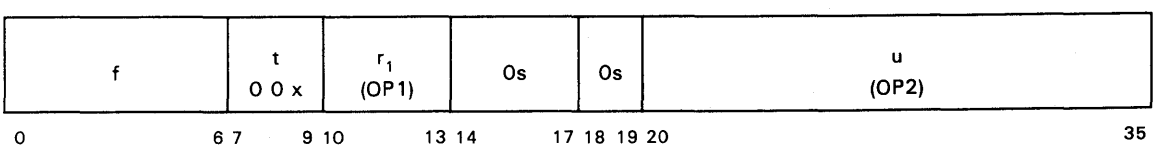


NOTE: x indicates bits not used in the field.

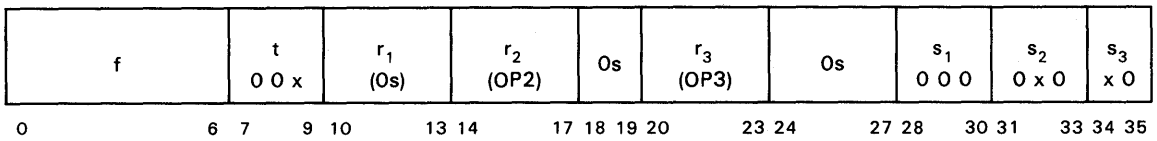
4.3.6. Count Leading Signs (ESC, DESC, ESCR, DESCR)

The OP2 source is examined and a count is made of the number of consecutive bits beginning with bit number one that have the same value as bit zero. The count value is in the range 0-35 for a single-precision source and 0-71 for a double-precision source. Only t-field values of 0 and 1 are permitted. The count value is placed into the destination (OP1 in RS format, OP3 in RR format). In the RR format, OP1 is not used, and the r_1- and s_1-fields must contain 0s. The destination is defined to be single-precision regardless of the source (single-precision or double-precision). Double-precision operations (t=1) are undefined if both s_2 and s_3 are 1 in the RR format. These instructions use the following RS and RR formats.

RS Format



RR Format

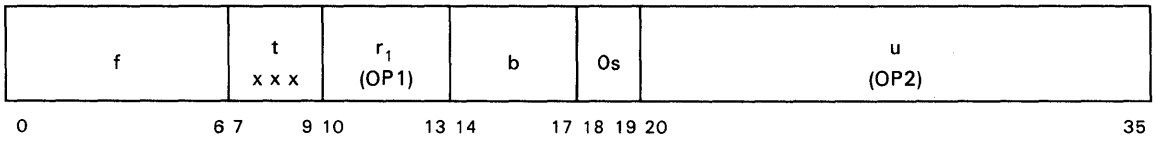


NOTE: x indicates bits not used in the field.

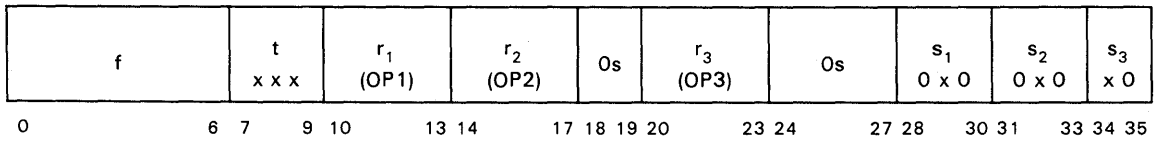
4.4. Scalar Logical Computational Instructions

The scalar logical computational instructions provide instructions to perform basic Boolean functions bit-for-bit on the two source operands to form the result operands. These instructions use the following RS and RR formats:

RS Format



RR Format



NOTE: x indicates bits not used in the field.

The s_1-, s_2-, and s_3-fields determine the location of operands OP1, OP2, and OP3 as either a G-register or a vector file element.

The t-field also acts as a partial OP code modifier in that it specifies the particular logical function, thereby allowing all logical instructions to share a common f-field value. The t-field encoding is:

t-field	Definition
0	Single-precision logical AND
1	Double-precision logical AND
2	Single-precision logical OR
3	Double-precision logical OR
4	Single-precision logical XOR
5	Double-precision logical XOR
6-7	Not used (causes undefined operation)

4.4.1. Logical AND (AND, DAND, ANDR, DANDR)

Forms the logical product (AND) of the two sources and places it in the destination. Operation is bit-for-bit combining each of the 36 or 72 bits of one source with the corresponding bit from the second source to produce the 36- or 72-bit result.

4.4.2. Logical OR (OR, DOR, ORR, DORR)

Forms the logical sum (OR) of the two sources and places it in the destination.

4.4.3. Exclusive OR (XOR, DXOR, XORR, DXORR)

Forms the logical difference (XOR) of the two sources and places it in the destination.

4.5. Scalar Comparison Instruction

The scalar comparison instruction compares the values of the OP1 and OP2 sources. The result of the comparison is used to set a 2-bit value in the scalar condition code (SCC) field in State register 6.

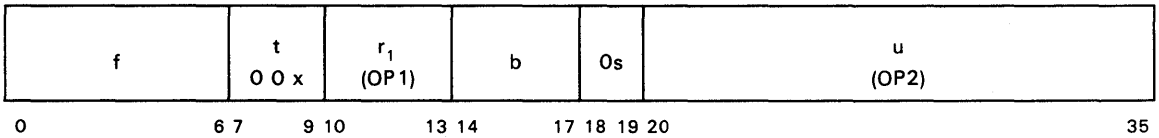
SCC	Interpretation
0	OP1 = OP2
1	OP1 > OP2
2	OP1 < OP2
3	Reserved

For scalar comparisons +0 and -0 are considered equal to each other.

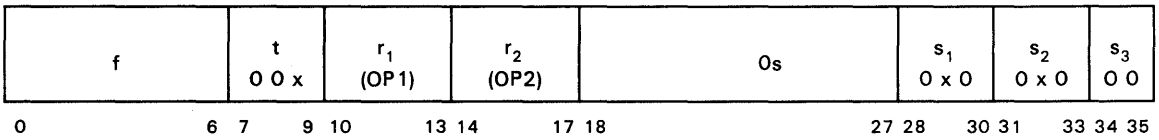
4.5.1. Compare (C, DC, CR, DCR)

Compares the value of OP1 to OP2 sources by subtracting OP2 from OP1, and indicates the result via the SCC. The t-field specifies either single-precision (t=0) or double-precision (t=1). In biased-exponent one's complement, two floating-point values can be correctly compared using integer comparison circuitry. The compare instruction uses the following RS and RR formats:

RS Format



RR Format



NOTE: x indicates bits not used in the field.

4.6. Scalar Type Conversion Instruction

The scalar type conversion instruction allows an operand of one type to be directly converted to an equivalent value in another type.

4.6.1. Convert (CIDIR, CIFR, CIDFR, CDIIR, CDIFR, CDIDFR, CFIR, CFDIR, CFDFR, CDFIR, CDFDIR, CDFFR)

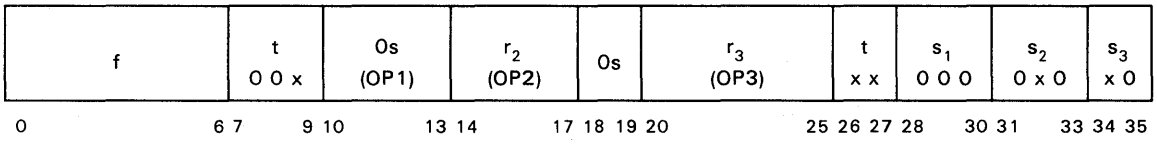
Converts the OP2 source operand from type t_2 to t_3. The result is placed into the OP3 location. The results are undefined when t_2 is equal to t_3. The s-field and r-field definitions are as for the RR format. Definitions of both t-fields (t_2 and t_3) are any of the values shown:

t-field	Definition
0	Single-precision integer
1	Double-precision integer
2	Single-precision floating-point real
3	Double-precision floating-point real
4-7	Reserved (causes undefined operation)

Floating-point inputs must be in correct normalized form. Given correct inputs, floating-point outputs are always in correct normalized form.

Conversions from double-precision to single-precision, and from floating-point to integer, may cause overflow conditions. Loss of precision does not constitute an overflow situation. Conversions involving a reduction in precision are performed by truncating rather than by rounding. The convert instruction uses the following RR format:

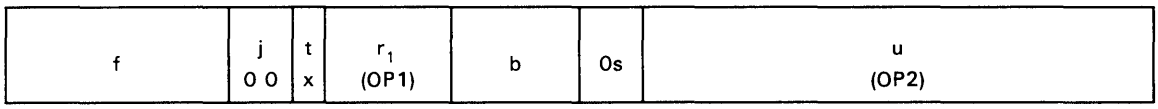
RR Format



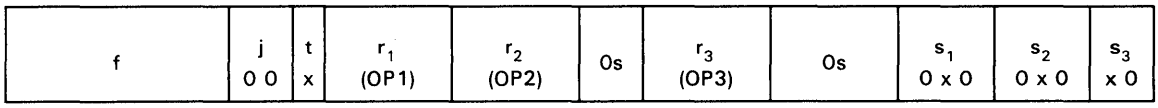
4.7. Scalar Shift Instructions

The scalar shift instructions allow data to be displaced, that is, scaled left or right with respect to an implied reference point, while optionally moving the data from one container to another. Both three-operand and two-operand formats are provided. The source operand to be shifted is OP1. The shift count is given by the rightmost seven bits of the OP2 source operand, which are interpreted as an unsigned binary quantity ranging in value from 0 through 127. However, correct operation is guaranteed only for shift count values in the range 0 through 72. These instructions use the following RS and RR formats:

RS Format



RR Format



NOTE: x indicates bits not used in the field.

In the RS format, OP1 is also the destination for the shifted data. In the RR format, the result is placed into OP3, and the OP1 contents are unaltered.

Since shift operands are essentially typeless data, the t-field specifies only single-word (t=0) and double-word (t=1). However, the t-field applies to OP1 and OP3 only. Double-precision operations (t=1) are undefined if s_2 and either s_1 or s_3 are 1 in the RR format. OP2 is considered single-word regardless of the t-field value. The j-field specifies the nature of the shift as:

j-field	Kind of Shift
0	Shift right logical (zero fill)
1	Shift right algebraic (sign fill)
2	Shift left logical (zero fill)
3	Shift left circular

4.7.1. Shift Right Logical (SSL, DSL, SSLR, DSLR)

Shifts the OP1 source value to the right the number of bit positions specified by the OP2 operand and places the result into the destination location. Bits shifted off the right are discarded, and 0s are used for fill on the left.

4.7.2. Shift Right Algebraic (SSA, DSA, SSAR, DSAR)

Shifts the OP1 source value to the right the number of bit positions specified by OP2, and places the result in the destination location. Bits shifted off the right are discarded, and bits supplied to the left are replicated sign bits.

4.7.3. Shift Left Logical (LSSL, LDSL, LSSLR, LDSLR)

Shifts the OP1 source value to the left the number of bit positions specified by OP2, and places the result in the destination location. Bits shifted off the left are discarded (there is no overflow) and bits supplied to the right are always 0s.

4.7.4. Shift Left Circular (LSSC, LDSC, LSSCR, LDSCR)

Operates the same as the Shift Left Logical instruction, except that bits supplied to the right are exactly those shifted to the left.

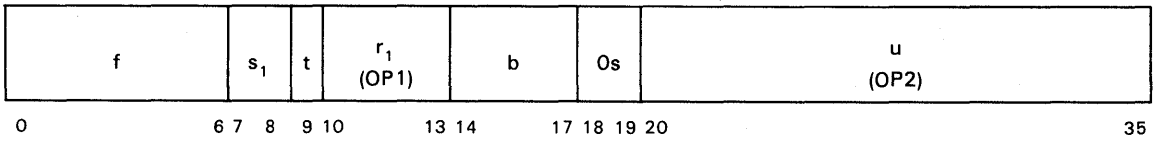
4.8. Scalar Move Instructions

The scalar move instructions move data from one location to another. In general they can access more locations than the computational instructions. The scalar move instructions are divided into two types of instructions: storage move instructions and move register-to-register instructions.

4.8.1. Storage Move Instructions

The storage move instructions use the following RS format:

RS Format



The operand is either one word (t-field equals 0) or two words (t-field equals 1) long. The s_1-field locates the OP1 operand as follows:

s_1-field	OP1 Location
0	G-register $<r_1>$; location is either left half (t=0) or entire double word (t=1).
1	G-register right half $<r_1>$; t-field must be 0 (an undefined instruction interrupt occurs if t-field is not 0).
2,3	Reserved.

Load Register (L, DL, LS)

The storage operand (OP2) specified by the b- and u-fields is obtained and placed into the OP1 location specified by the t-, s_1-, and r_1-fields.

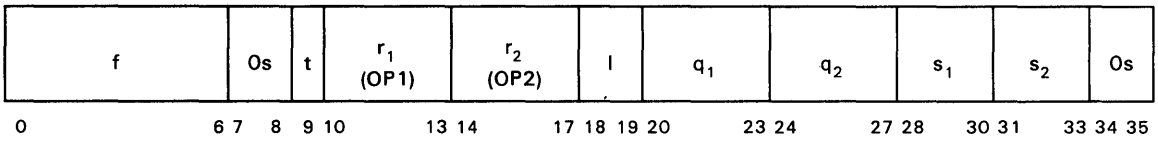
Store Register (S, DS, SS)

The operation of this instruction is the inverse of the Load instruction. This instruction moves the operand OP1 from the register location to the storage location OP2.

4.8.2. Move Register-to-Register Instructions

The source operand, specified by s_2–, r_2–, and q_2–fields is obtained and copied into the destination location specified by the s_1–, r_1–, and q_1–fields. This instruction uses the following RR format:

RR Format



The t–field specifies either single–precision (t=0) or double–precision (t=1). The l–field selects one of four sources from which to acquire additional pointer or length information by specified combinations of the s– and r–fields. The s_1– and s_2–fields are defined as follows:

s–field	Operand Location
0	G–register, same as for scalar Load instruction (based on the t–field).
1	S registers 0–15, selected by r–field values of 0–15 respectively. Valid only for t=0.
2	The vector file element where the r–field specifies the file number, and element pointer of the Element Loop register pointed at by the current element loop pointer specifies the element number.
3	Reserved.
4	Right half of G–register specified by the r–field; the t–field value must be 0.
5	The t–field value must be 0.*
6	Element 0 of file r.
7	The element of the r–field specified vector file that is specified by bits 31–35 (t=1) or bits 30–35 (t=0) of the G–register specified by q.

**An s–field value of 5 makes available certain parameter values selected by the following values of the r_2–field specified source operands or the r_1 specified destination operand and l–fields:*

r-field	Operand Location or Parameter Value	s=5 Parameter Selection
0,1	Reserved.	None
2	Number of 1s in the mask	Specified by r_2
3	Number of 0s in the mask	Specified by r_2
4	Index of the first 1 in the mask	Specified by r_2
5	Index of the first 0 in the mask	Specified by r_2
6	Index of the last 1 in the mask	Specified by r_2
7	Index of the last 0 in the mask	Specified by r_2
8	Length parameter value (RL)	Specified by r_2
9	Strip size (MZ)	Specified by r_2
10	Maximum element count value	Specified by r_2
11	Element pointer value	Source specified by r_2
12	Alternate element pointer 1 field	Destination specified by r_1
13	Alternate element pointer 2 field	Destination specified by r_1
14,15	Reserved	None

The r-field values are computed as follows:

In computing the values for r=2-7, the mask is treated as a bit string whose length is determined by the l-field of the instruction. Values of 0-3 in the l-field select ELCNT, NELCNT, ALT1, and ALT2 respectively, and the selected parameter specifies the number of mask bits to inspect. Mask bits beyond this length are entirely ignored. Index refers to the bit's position within the mask. Bits are numbered from the left starting with zero. If, when r=4-7, the desired bit does not exist (e.g., the mask is all zeroes and r=4), then the operand value defaults to the value of the length parameter defined by the l-field.

The values produced by r=8,9 are based upon the contents of the Vector Loop register currently pointed at by CVLP (r=8 also makes use of the instruction l-field as previously defined). The strip size value is 64 if MZ=0 and 32 if MZ=1. The values produced by r=10,11 are copied from the Element Loop register pointed at by the current value of the current element loop pointer. The fields specified by r=12,13 are those in the Vector Loop register currently pointed at by CVLP.

Only r-field values of 2-11 are permitted as source operands, and only r-field values of 12-13 are permitted as destinations. If other values are used the results are not defined. When writing into the ALT fields, only the rightmost seven bits of the source operand are transferred, other bits being discarded and ignored.

Move Scalar (MS, DMS)

Interprets move instructions to move the source operand to the destination operand.

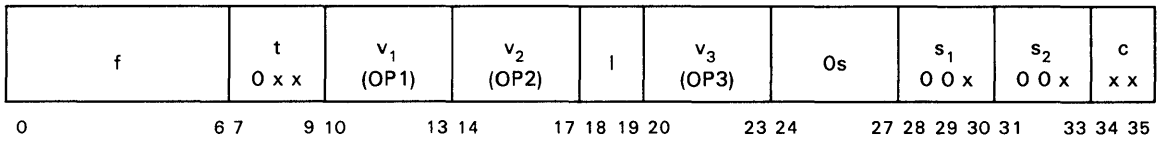
Move Negative Scalar (MNS, DMNS)

This instruction is similar to the Move Negative Scalar instruction, except that the one's complement of the source operand is moved to the destination.

4.9. Vector Arithmetic Instructions

The vector arithmetic instructions provide instructions to perform the basic arithmetic operations on vector operands. The term *operand* refers to an entire strip of a vector; the individual data values in the vector are called elements. These instructions use the following VV format:

VV Format



The two source operands, OP1 and OP2, are each specified by the v-field and the s-field. If the s-field is 1, then the v-field specifies the vector file whose contents are the source vector. If the s-field is 0, then the v-field selects a G-register whose value is replicated to form a vector of identical values. If both s-field bits are 0, the result vector is unpredictable.

The l-field specifies the length limitation parameter as explained in the preceding section. The c-field allows a further selection of elements within this length. Encoding of this field is as follows:

c-field	Elements upon which Processing Occurs
0	All (mask is ignored).
1	Elements whose positions correspond to 0s in the mask
2	All (mask is ignored).
3	Elements whose positions correspond to 1s in the mask

The destination location is a vector file, and is specified by the v_3 field (OP3). Element locations in the destination file for which the operation is not completed, because the locations are beyond the current strip length limit, or being masked out by the conditional mask, are left undisturbed by the instruction execution. The computations for masked elements are logically performed, but they cannot cause fault conditions (such as overflow or zero divide).

The t-field specifies the data type and operation type for the vector arithmetic instructions as follows:

t-field	Definition
0	Single-precision integer
1	Double-precision integer
2	Single-precision floating-point real
3	Double-precision floating-point real
4-7	Reserved

Except as noted within the instructions, the t-field applies to both sources and also to the result produced.

4.9.1. Add Vector (AV, DAV, FAV, DFAV)

Adds the OP1 and OP2 source vectors together and places the resultant vector into the destination file. In floating-point, an all-zero (sign, characteristic, and mantissa) element is forced if the computed mantissa value is +0 or -0. Integers use subtractive addition as for scalar adds.

4.9.2. Add Negative Vector (ANV, DANV, FANV, DFANV)

Subtracts the OP2 source vector from the OP1 source vector. In other respects operation is similar to the Add Vector instruction.

4.9.3. Multiply Vector (MSIV, MLIV, FMV, DFMV)

Multiplies the OP1 and OP2 source vectors together to form the resultant product vector. In other respects operation is similar to the Add Vector instruction, except that double-precision integer type is not directly supported. Instead, t=1 defines an operation in which the source operands are accepted single-precision integers, and a single-word result is produced. Its value is the full algebraic product shifted 36 places right, sign extended (t=0 produces the right half and t=1 produces the left half of the full double-word product.)

NOTE: The integer overflow interrupt must be masked off when using t=0 to produce the right half of the double-word product.

4.9.4. Divide Vector (DSIV, FDV, DFDV)

Divides the OP1 source vector by the OP2 source vector. No remainder is produced. A quotient vector is produced that is placed into the destination file. Division by zero causes a divide fault condition. In other respects, operation is similar to the Add Vector instruction, except that the double-precision integer type is not supported. The t=1 value is reserved.

4.9.5. Absolute Value (EMV, DEMV)

Computes the absolute value of source operand OP2 and places the result into the destination file. Source operand OP1 is not used so the v_1-field should be 0 and the s_1-field should be 1.

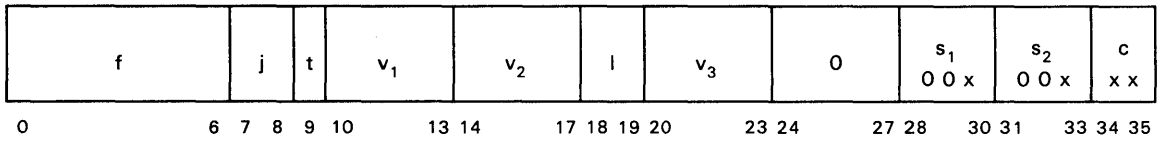
4.9.6. Negative Vector (MNV, DMNV)

Places the ones complement of the OP2 source vector into the destination file. Source operand OP1 is not used so the v_1-field should be 0 and the s_1-field should be 1.

4.9.7. Vector Shifts (SSLV, DSLV, SSAV, DSAV, LSSLV, LDSLV, LSSCV, LDSCV)

The vector shift instructions allow a vector of elements (OP1) to be shifted by a vector of shift counts (OP2). The resultant vector is placed into OP3, and OP1 is not altered. The instruction format is:

VV Format



The t-field specifies whether the shift data (OP1,OP3) is single-word (t=0) or double-word (t=1). In either case OP2 is always treated as single-word, and the shift count is extracted from the rightmost seven bits and interpreted as an unsigned binary quantity in the range 0-127. Operation is not defined if OP2 and OP3 specify the same vector register and t=1, or if the shift count exceeds 72.

The j-field specifies the kind of shift:

j-field	Kind of shift
0	Shift Right Logical (zero fill)
1	Shift Right Algebraic (sign fill)
2	Shift Left Logical (zero fill)
3	Shift Left Circular

Shift definitions are the same as the scalar shift instructions, see 4.7.

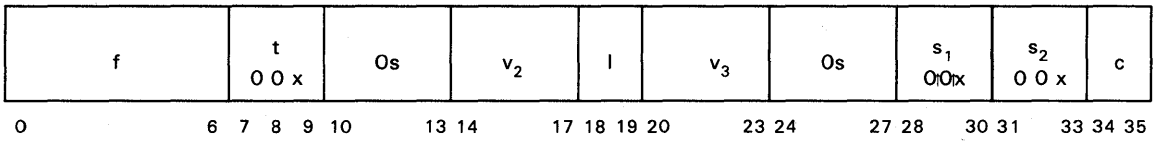
4.10. Vector Bit Evaluation Instructions

The vector bit evaluation instructions provide instructions to perform count leading signs vector, population count vector, and population parity count operation.

4.10.1. Count Leading Signs Vector (ESCV, DESCV)

The elements of the source vector (OP2) are examined, and for each a count is made of the number of consecutive bits, beginning with bit 1, that have the same value as bit 0. The vector of counts is placed into the destination (OP3). The source is either single- or double-word, but the result is always a single word. The instruction format is:

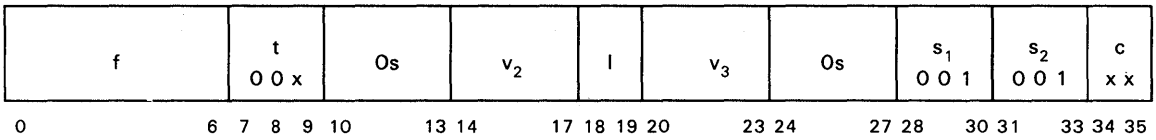
VV Format



4.10.2. Population Count Vector (EBCV)

This instruction uses the following VV format:

VV Format

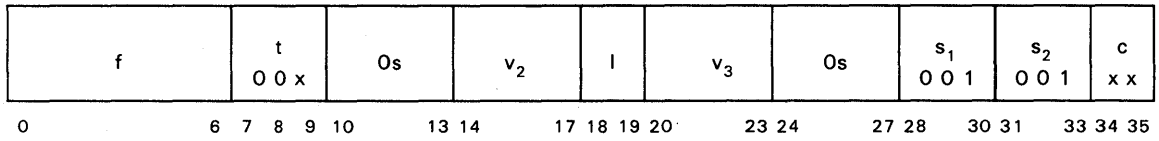


This instruction examines the elements of the source vector OP2, and a count is made for each of the number of bits set to 1. The vector containing the counts is placed into the destination operand OP3. This instruction is only supported with the t-field equal to zero. Both the source and the result must be single-precision.

4.10.3. Population Parity Count (EBPV)

This instruction uses the following VV format:

VV Format



This instruction examines the elements of the source vector, OP2, to determine the number of bits set to 1 in each element. If this number is odd, the result is set to the integer value plus one. If this number is even, the result is set to the integer value plus zero. The results are placed into the destination operand OP3. This instruction is only supported with the t-field equal to zero. Both the source and the destination must be single-precision.

4.11. Vector Logical Instructions

The vector logical instructions provide instructions to perform Boolean functions bitwise per element pair and elementwise through the vectors. They are specified by the VV format as defined for the vector arithmetic instructions. These instructions operate similar to the vector instructions, except that the t-field is defined as follows:

t-field	Definition
0	Single-precision logical AND
1	Double-precision logical AND
2	Single-precision logical OR
3	Double-precision logical OR
4	Single-precision logical XOR
5	Double-precision logical XOR
6,7	Not used

4.11.1. Logical AND Vector (ANDV, DANDV)

Forms the logical product (AND) of the source vectors and places the result into the destination file.

4.11.2. Logical OR Vector (ORV, DORV)

Forms the logical sum (OR) of the source vectors and places the result into the destination file.

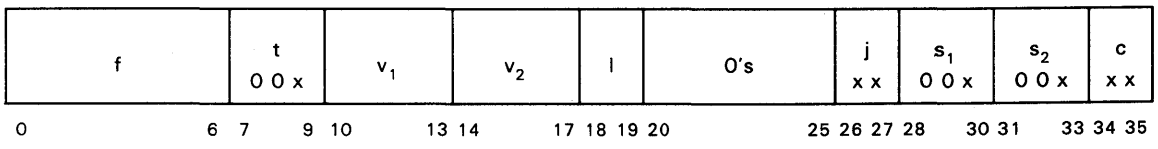
4.11.3. Logical Exclusive OR Vector (XORV, DXORV)

Forms the logical difference (XOR) of the source vectors and places the result into the destination file.

4.12. Elementwise Comparison Instruction

The elementwise comparison instruction uses a modified form of the VV format as shown here.

VV Format



The s-, v-, l-, and c-fields function as defined for the vector arithmetic instructions. The instruction compares OP1 to OP2 to produce a condition vector according to the specified relation (j-field), and places it into the mask register.

The selection of relation is done by the j-field encoded as:

j-field	Relation of Elements Causing 1 Value in Condition Bit
0	OP1 < OP2
1	OP1 ≤ OP2
2	OP1 = OP2
3	OP1 ≠ OP2

For purposes of this testing, a negative zero source element is considered to be equal to rather than less than a positive zero. The t-field specifies single-precision (t=0) or double-precision (t=1).

The comparison operation differs slightly from the subtraction operation in that conditions that would cause an overflow from the subtraction are handled correctly without overflow indication by the comparison.

4.12.1. Compare Vector (CLV, CLEV, CGV, CGEV, CEV, CNEV, DCLV, DCLEV, DCGV, DCGEV, DCEV, DCNEV)

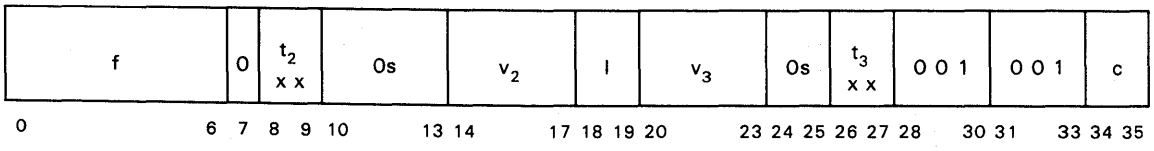
Compares the OP1 and OP2 source vectors by means of a vector subtraction of OP2 from OP1. The difference vector is not stored but is examined to produce a condition vector. The condition vector is a bit string in which each bit represents the outcome of the j-field specified comparison of the corresponding source elements. Because only one bit is produced per element comparison, the relation being tested must be specified in the compare instruction by the j-field.

Use of the conditional function provided by the c-field yields some useful, though not immediately obvious benefits. If the compare is executed only for elements opposite ones in the mask (c=3), then zeroes in the mask are unchanged by the comparison, whereas ones are replaced by the new bit values from the new comparison. The net result is to form the logical AND of the new mask with the previous one. Similarly c=1 results in the logical OR of the masks.

4.13. Vector Type Conversion Instructions

The vector type conversion instructions convert a vector of one data type to a vector of another data type. These instructions use the following VV instruction format.

VV Format



4.13.1. Convert Vector (CIDIV, CIFV, CIDFV, CDIIV, CDIFV, CDIDFV, CFIV, CFDIV, CFDFV, CDFIV, CDFDIV, CDFFV)

Converts the elements of the OP2 source vector element-by-element from type t_2 to type t_3 and places the result into the file specified by v_3. The results are undefined when t_2 is equal to t_3. The encoding of the t_2- and t_3-fields can be any of the following values:

t-field	Definition
0	Single-precision integer
1	Double-precision integer
2	Single-precision floating-point real
3	Double-precision floating-point real

The floating-point inputs must be in correct normalized form. With the correct inputs, floating-point outputs are always in correct normalized form.

Conversions from double-precision format to single-precision format and from floating-point to integer may cause an overflow condition. A loss of precision does not cause an overflow condition. Conversions involving a reduction in precision are performed by truncating rather than by rounding.

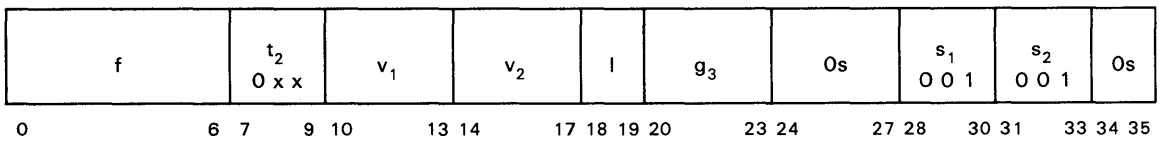
The l-field specifies the number of elements both for the source vector and the result vector independent of number of words per element.

If conversion from single-precision to double-precision is specified, v_2 and v_3 must not specify the same vector file since storing of the result would at some point destroy source elements before they could be used.

4.14. Vector Reduction Operation Instructions

Reduction operations are distinguished from elementwise operations in that they operate on a single source vector to produce a scalar (rather than vector) result. They are not conditional, having no c-field. These instructions use the following VV format.

VV Format



4.14.1. Sum Reduction (SUM, DSUM, FSUM, DFSUM)

The vector source operand, OP2, is taken from the vector file specified by v_2, and its elements are added together. That sum is then written into the G-register specified by g_3 (OP3). Both OP2 and OP3 are defined to be of type t where t is defined as the Convert instruction. The file specified by v_1 is allocated as a temporary scratch pad area for use by the hardware. Its entire contents may be altered by the instruction. If v_1 and v_2 specify the same file, then the file contents may be destroyed by the instruction but the numeric result is still correct.

The order of performing sum reduction is as follows: adjacent elements are combined pairwise; then adjacent partial sums are combined pairwise until only a single result emerges. If, at any stage, the number of elements or partial sums is odd, the first N/2 sums are produced, and the remaining element is carried forward to the next stage.

If the l-field indicates a length of zero, the contents of OP3 is left undisturbed.

4.14.2. Product Reduction (FPRD, DFPRD)

The operation of the Product Reduction instructions is similar to the Sum Reduction instructions, except that the elements of OP2 are multiplied rather than added together, and only floating-point operations (types 2 and 3 defined in 4.2.1) are supported. If the vector length as taken from the source selected by the l-field is zero, the contents of OP3 are left undisturbed.

4.14.3. Maximum Reduction (MAX, DMAX)

The elements of the vector file specified by v_2 within the length specified by the l-field (length source) value are compared. The index, that is, element number beginning with zero of the maximum value is placed into the G-register specified by g_3-field. The t-field is defined as in 4.2.1, but applies only to the source vector; t=0 for single-precision and t=1 for double-precision. (Values must be all integer or all real for valid comparisons.) The resultant index value is always a single-precision integer, and it lies in the range 0-63. When more than one element has the maximum value, the index stored is the lowest of the possible index values. Negative zero is considered as equal to, not less than, or positive zero. Overflow and underflow conditions do not occur from this instruction. If the vector length specified is zero, the contents of $G(g_3)$ are left undisturbed.

Programming Notes:

1. Once the index of the maximum is known, the maximum value itself can be easily obtained with a single scalar move instruction.

2. Extension to operation over multiple strips can be done as follows:

    a. Process the vector stripwise, comparing corresponding elements of consecutive strips with a Compare Vector instruction and select the maximum with a conditional move, then perform the maximum Reduction instruction after combining all strips into one.

    b. Process vectorwise, selecting the maximum of each strip with the maximum Reduction instruction and place its value into the element of a result file so that it corresponds to the strip number just processed. When the file is full or the end of the vector is reached, another maximum Reduction instruction can be performed on the result file, and so on if necessary.

4.14.4. Minimum Reduction (MIN, DMIN)

The operation of this instruction is similar to the maximum Reduction instruction, except that the index of the minimum element is returned.

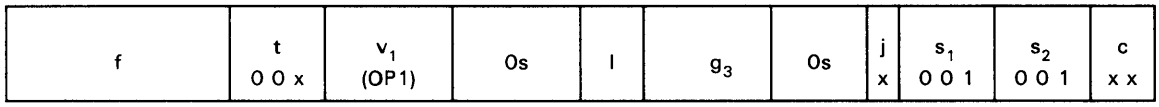
4.15. Vector Move Instructions

The vector move instructions move strips of vector data between the vector files or to and from storage locations, and the vector files. These instructions manipulate file elements without interpreting their values. The t-field only distinguishes single word (t=0) and double word (t=1).

4.15.1. Load Vector (LV, DLV)

The Load Vector instructions use the following VV format:

VV Format



A strip of vector elements is obtained from the storage locations specified by the virtual address contained in the G-register selected by g_3 and placed into the vector file specified by v_1.

The l-field selects the source of the length parameter indicating the upper limit on the number of elements to be transferred. The c-field allows certain of those transfers to be conditionally suppressed as defined in 4.9.

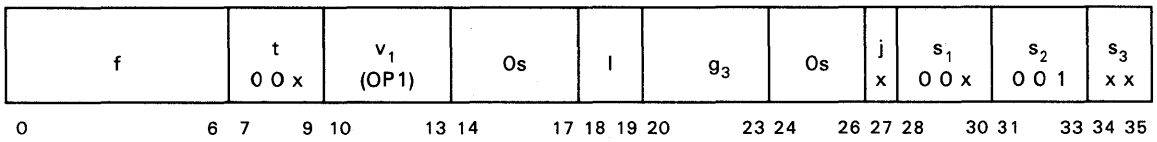
The j-field specifies how the stride value is obtained. For j=0 the stride value is taken from the right half of the G-register specified by g_3. The stride value is interpreted as an integer that may be positive, negative or zero. (Actually a zero stride value would normally not be used since the single source element being referenced can be entered once in a G-register and then broadcast during subsequent vector instructions as if it were in a vector file.) For j=1 a constant value of +1 is used for single-precision and +2 is used for double-precision. In both cases, the original virtual address initially points to the first vector element, and subsequent elements are addressed by stepping through storage by the stride value. Note that the stride is defined in terms of words, rather than elements. Also the stride value affects only the spacing of items in storage, not in the file. If the initial address plus accumulated stride values fall outside the segment limits, an Address Limits Error occurs.

Vector file elements beyond the limit determined by the l-field determined length source and those not loaded because of the c-field setting, are left unaltered by this instruction. Also, the contents of $G(g_3)$ are not altered by this instruction.

4.15.2. Store Vector (SV, DSV)

The Store Vector instructions use the following VV format:

VV Format



This instruction operates in reverse order of the Load Vector instruction, moving vector elements from a vector register (s_1=1) or a broadcast G-register (s_1=0) to locations in storage.

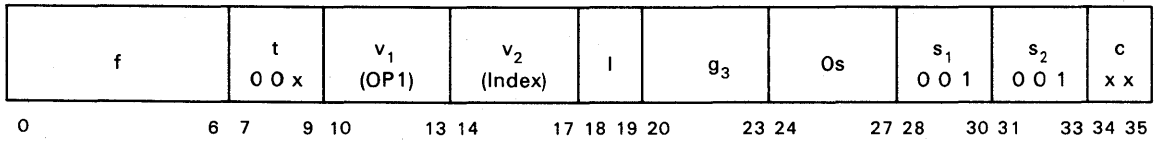
4.15.3. Generate Index Vector (GXV)

This instruction operates similar to the Load Vector instruction except that the values loaded into the file locations are the virtual addresses generated rather than the storage contents of those virtual addresses. Also address translation and checking are not done at all, so that any single-precision integer values can be used as starting address and stride. This instruction is defined only for single-precision (t=0). Overflow conditions are ignored, the truncated results are stored. This instruction may be used to generate any vector element of equally spaced integers. This instruction uses the same format as the Store Index instruction.

4.15.4. Indexed Load Vector (LVX, DLVX)

The Indexed Load Vector instruction uses the following VV format:

VV Format



This instruction loads a strip of elements into the file specified by v_1. The l-field determines the number of elements as already defined. The virtual address of each element is obtained by taking the base virtual address from the G-register specified by g_3 and adding to it the offset from the corresponding element of the file specified by v_2. Addition of offsets is not cumulative, but each is added to the base virtual address independently. Offsets may be positive or negative or zero but must not produce a virtual address outside the limits of the segment determined by the base virtual address of elements that are loaded or an address limits error interrupt will occur. OP2 is always assumed to be a single-precision integer regardless of the t-field value. The G-register is not altered by this instruction. File elements beyond the limit determined by the l-field and those not loaded due to the c-field setting are left unaltered by this instruction. The addresses of elements not loaded do not cause address limits error interrupts or data alignment error interrupts.

Specifying $v_1=v_2$ when t=1 produces unpredictable results.

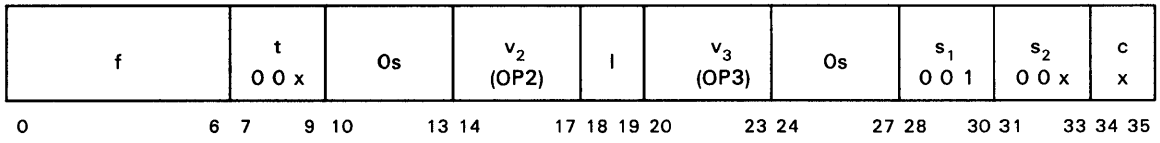
4.15.5. Indexed Store Vector (SVX, DSVX)

Operation is the inverse of Indexed Load Vector moving data from consecutive OP1 file locations to indexed storage locations. Operation is defined to proceed in the direction of increasing file addresses. Hence, if identical offset values exist in OP2, thus causing multiple OP1 elements to be written to the same storage location, then the OP1 element from the highest file address overwrites other elements stored there. This instruction uses the same format as the Indexed Load Vector instruction.

4.15.6. Move Vector (MV, DMV)

The Move Vector instruction uses the following VV format:

VV Format

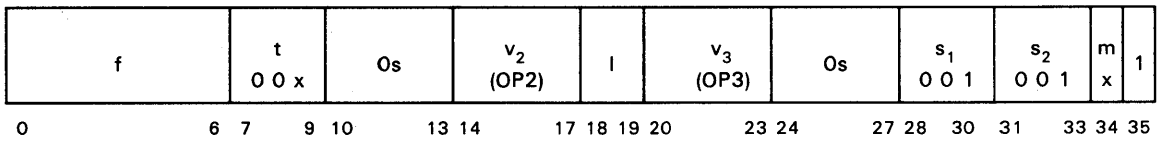


The OP2 source vector is copied to the file specified by v_3. OP2 may be either a broadcast G-register ($s_2=0$) or a vector file ($s_2=1$). The l and c-fields define the length and condition of the transfer.

4.15.7. Compress Vector (MCV, DMCV)

The Compress Vector instruction uses the following VV format:

VV Format



Elements of OP2, for which the corresponding mask register bit is a one (m=1) or zero (m=0), are selected and written into adjacent locations of the OP3 file beginning with location zero. Only OP2 elements within the length defined by the l-field are used. The number of OP3 locations written is given by the number of ones (m=1) or zeroes (m=0) in the mask within this defined length. Other OP3 locations are not altered. Bit 35 of the instruction must be one for operation.

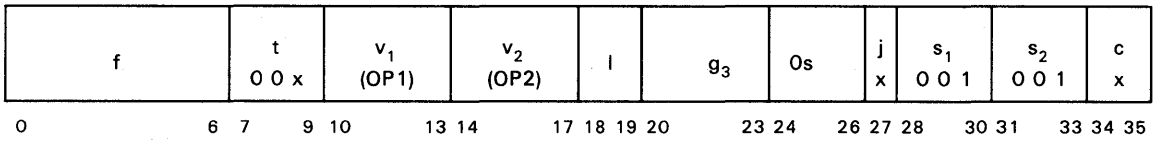
4.15.8. Distribute Vector (MDV, DMDV)

Distribute is essentially the inverse of Compress and has the same format. The elements of OP2 are taken sequentially and written into those locations of OP3 that correspond to ones (m=1) or zeroes (m=0) in the mask register. The length parameter determined by the instruction l-field applies to the mask and hence to OP3. The number of OP2 elements used is the number of ones (m=1) or zeroes (m=0) in the mask within the defined length. Locations of OP3 not explicitly written by this instruction are left unaltered. Operation when $v_2=v_3$ is not defined.

4.15.9. Load Alternating Elements Vector (LAEV, DLAEV)

The Load Alternating Elements Vector instructions uses the following VV format:

VV Format



Pairs of elements are obtained from storage beginning at the virtual address from G (g_3). The first element of each pair is placed into V (v_1), the second into V (v_2). Pairs are always from adjacent elements in storage. The stride value, which is determined by the j-field as in Load Vector (4.14.1), specifies the amount to add to the address of the second element of a pair to obtain the address of the first element of the next pair.

The length parameter (l-field selected) specifies the number of elements to be placed in each of the vector registers. Thus the total number of elements moved is twice that value. The c-field allows the transfer of certain pairs of elements to be conditionally suppressed as defined in 4.9. Suppressed transfers are included in the l-field count.

Addresses are constrained to pair boundaries. Thus when t=0 (single-word), the starting address must be even, and when t=1 (double-word elements) the starting address must be a four-word boundary. Otherwise a Type 30 external interrupt is caused. Also the stride must be an odd multiple of the element size. Operation is not defined for $v_1=v_2$.

4.15.10. Store Alternating Elements Vector (SAEV, DSAEV)

This is the inverse of the Load Alternating Elements Vector, forming pairs of elements from corresponding locations in the two vector files, and writing them to storage. This instruction uses the same format as the Load Alternating Elements Vector instruction.

4.16. Loop Control Instructions

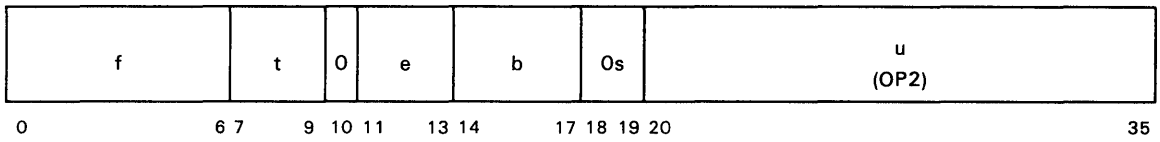
The principles of processing vectors in limited-length strips are discussed in 2.3.6. In particular it is describing how a single FORTRAN DO-loop is broken down into a vector loop and an element loop within it, the former processing one strip per pass, and the latter processing one element position per pass. The following four instructions are used to build and end vector and element loops, as their names imply.

NOTE: The Jump to Vector Loop, Build Element Loop, and Jump to Element Loop instructions all share the same OP code value. They are distinguished by the bits immediately following the OP code.

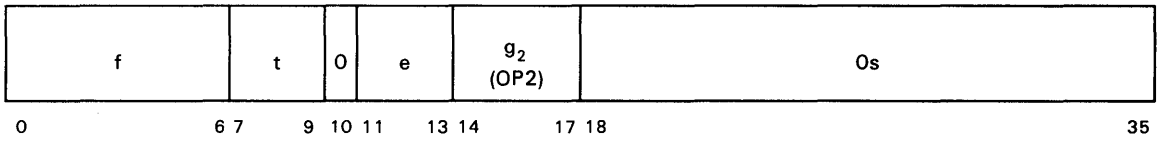
4.16.1. Build Vector Loop (BSVL, BLVL, BSVLR, BLVLR)

This instruction sets up the control parameters for a loop over vector strips, with each pass through the loop processing one strip of the vectors. The total number of elements to be processed per vector is the non-negative integer value taken from bits 6-35 of the OP2 operand. If bits 0-5 of the OP2 operand are other than 0s, a Remaining Length Overflow interrupt (3.2.4 - Type 13) is caused. Instruction formats for this instruction are:

RS Format



RR Format



In the first format, OP2 is from the storage location specified by the b and u-fields as for RS format instructions; in the second case it is from the G-register specified by g_2.

Essentially this instruction builds a new VL-register entry in the Vector Loop register specified by the e-field and sets the CVLP to point to it. The field values for the new entry are defined to be:

c-field	Description
MZ	Derived from the t-field of this instruction
RL	Value of OP2
ALT1	The ELCNT value from the previous Vector Loop entry
ALT2	The ALT1 value from the previous Vector Loop entry

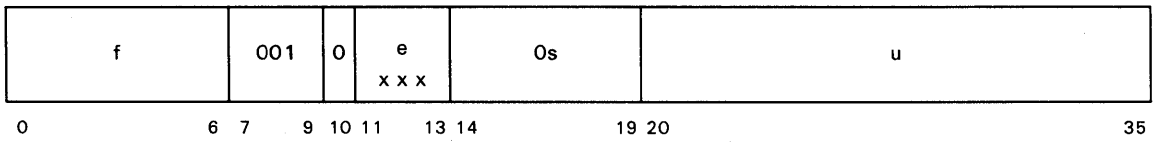
The phrase *previous Vector Loop entry* means the contents of the VL register pointed to by the CVLP before execution of this instruction.

This instruction is defined for t-field values of 5 and 6 that produce MZ bit values of 0 and 1 respectively. The t-field values 0-4 and 7 are reserved for future expansion.

4.16.2. Jump to Vector Loop (JVL)

This instruction is used to close the vector loop started by a Build Vector Loop instruction. For straightforward applications the jump address points to the instruction following the Build Vector Loop . Operation consists of subtracting the current ELCNT value from the RL value and placing the result back into RL. If the new RL value is greater than zero, then the jump is taken to the location pointed to by u. (See 4.17 for definition of jump address formation.) Otherwise the e-field value is put into the CVLP, and execution continues with the next sequential instruction.

RS Format



4.16.3. Build Element Loop (BEL)

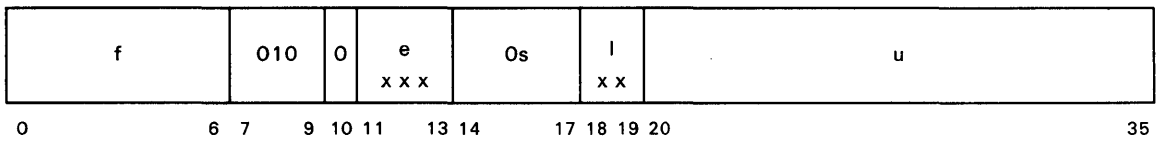
This instruction establishes the parameters for performing an element loop. An element loop is a loop over elements within a strip of a vector. It is used typically within the context of a vector loop, i.e., the BEL occurs logically between a Build Vector Loop and its JVL. Each element loop is associated with a particular vector loop in the sense that they are both derived from the same source language DO-loop. Due to the interchanging of nested loops for the sake of program execution efficiency, the vector loop for a particular element loop may or may not be the innermost vector loop existing at the time the BEL is executed. The l-field of the BEL specifies a strip length in the same way as for vector instructions, i.e., it is derived from the contents of the Vector Loop register pointed to by CVLP.

The BEL sets the current element loop pointer to point to the Element Loop register specified by the e-field value and then initializes the fields of that register as follows:

Status	Description
Element Pointer (ELPT)	Set to zero.
Maximum Element Count (MEC)	Set to the length parameter value from the source specified by the l-field.

The value placed into the MEC field is then tested. If the field contains zero, then a jump is made to the code offset specified by the u-field; if non-zero, then execution continues with the next sequential instruction. The branch address should be that of the JEL corresponding to the BEL. The net result is that element loops with a count of zero are executed zero times, as desired. A MEC value greater than 64 does not cause a fault condition.

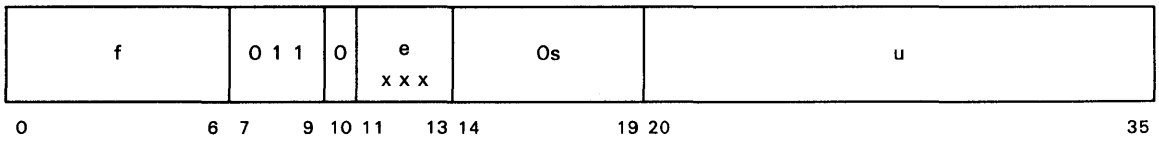
RS Format



4.16.4. Jump to Element Loop (JEL)

This instruction is used to close the element loop started by a BEL instruction. For straightforward applications the jump address points to the instruction following the BEL. Operation consists of adding one to the ELPT field of the Element Loop register entry pointed to by the current element loop pointer. If the sum is less than the MEC value of that same Element Loop register, then the sum is stored in the ELPT field and the jump is taken; otherwise the current element loop pointer is set to the value from the e-field of the JEL and execution proceeds with the next sequential instruction.

RS Format



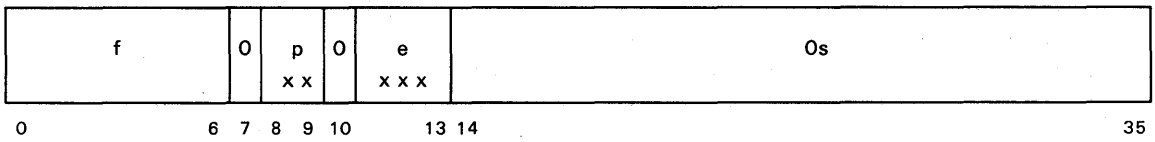
4.16.5. Adjust Loop Register Pointers (CELP, CVLP, CVELP)

This instruction is used to alter the contents of the Current Vector Loop Pointer (CVLP) and the Current Element Loop Pointer (CELP) which define locations in the VL and EL registers, respectively. The p-field selects the pointer to be altered:

p-field	Description
0	Not used
1	CELP only
2	CVLP only
3	Both

The selected pointer is set to the value from the instruction e field. The contents of the EL and VL registers are unaffected by this instruction.

RS Format



4.17. Jump Instructions

The jump instructions are used where it is desirable to alter the otherwise sequential nature of executing instructions. Instructions are defined to reside logically in main storage, and thus to be addressed with standard 36-bit virtual addresses. Three restrictions are placed on instruction code however:

1. Attempts to write into the code while the activity is executing yields unpredictable results;

2. Code mapped into the Local Storage segment causes a Storage Protection Check–Execute when an attempt is made to execute it;

3. The virtual address of each instruction (the value of Program Address register value used to fetch the instruction) must have 0s in bits 18 and 19.

The third restriction results from the fact that the branch instruction u–field is 16 bits. Therefore, code beyond 64K cannot be reached by segment–local branches and is of limited use. As a debug aid, any attempt to fetch an instruction with PAR bits 18,19 non–zero causes an external interrupt. In practice these bits can only become non–zero by PAR incrementation, the LAJ and JXS instructions or bad initial PAR value.

Most jump instructions allow target addresses that are constrained to be within the current segment. Specifically the 16–bit u–field specifies a word offset into the current segment. In this case the target virtual address is constructed of:

Bit	Description
0–17	Program Address Register (PAR) bits 0–17
18–19	Zeroes
20–35	Jump instruction u–field

The ability to jump between segments is provided by the LAJ and the JXS instructions, which allow full virtual addresses to be specified. These instructions form the target virtual address from the 16–bit instruction u–field and a 36–bit address from a G–register.

Bit	Description
0–17	Bits 0–17 from the G–register
18	OR of bits 18 and 19 from the G–register
19	Carry out from the following unsigned addition
20–35	Sum of the u–field and G–register bits 20–35

The JVL, BEL, and JEL instructions, though technically jump instructions, were defined under the section on loop control since they are used specifically in that context. In terms of address formation they follow the definitions in this section.

Each time a jump instruction is executed and the jump is taken, an entry is created and added to the Jump History File located in the scientific processor control block.

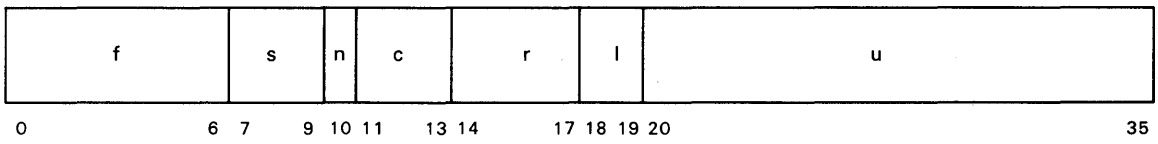
When execution of any jump instruction causes an interrupt that is to be taken synchronously, then the jump is not taken. Consequently the return address will be one beyond the interrupting jump. Faults on the jump target address are not considered part of the jump execution, but instead are associated with the fetch of the next instruction.

4.17.1. Conditional Jump (CJ)

The conditional jump instruction provides an assortment of pertinent program conditions to be tested as the basis for possible branching. The jump when taken is always segment-local as described in 4.17.

This instruction obtains an operand, which may be either a data element or a piece of machine state such as the scalar condition code, and tests it for a particular numeric condition. If the desired condition is obtained, the jump is taken; otherwise execution continues with the instruction immediately following the Conditional Jump instruction. The operand tested is never altered by the instruction.

RS Format



The operand for testing is specified by the s-field and the r-field values, and in a few cases the l-field value. The condition tested for is specified primarily by the c-field. The n-field selects a branch on true when it is 0 or inverts the use of the test result when it is 1.

The definitions of the c-field selections are shown in Table 4-1.

Table 4-1. Conditional Jump Instructions, c-field

c-field Value	Condition of Operand Causing True
0	Always TRUE.
1	-0, operand is all 1s.
2	+0, operand is all 0s.
3	+ and -0, operand is all 1s or all 0s.
4	<-0, operand left bit is 1 but not all bits are 1s.
5	<+0, operand left bit is 1.
6	≤+0, operand left bit is 1 or operand is all 0s.
7	Operand right bit is 1.

The definitions of the s-field values are shown in Table 4-2.

Table 4-2. Conditional Jump Instructions, s-field

s-field Value	Operand for Testing	Test Condition
0	G-register bits 0-35 specified by the r-field.	Specified by the c-field.
1	G-register bits 0-71 specified by the r-field.	Specified by the c-field.
2-4	Reserved for future expansion.	None.
5	Special (see Table 4-3).	Specified by the r- and c-fields.
6-7	Reserved for future expansion.	None.

NOTE: When the s-field value is 5, the r-field selects the operand for testing and specifies the test condition or selects the c-field that specifies the test condition. The s-field value of 5 does not alter the definition of the c-field or the n bit. The r-field definition for s=5 is given in Table 4-3.

The definitions of the r-field values when the s-field value equals 5 is given in Table 4-3.

Table 4-3. Conditional Jump Instructions, r-field Definitions (when s-field Equals 5)

r-field Value	Operand for Testing (s-field = 5)	Test Condition
0	Bits 0 through NL-1 of the mask register, where NL is the length parameter specified by the instruction I-field. If NL=0, then the operand test result is true for all values of the c-field. If NL is greater than 64, then a Vector Register Length Overflow condition exists and the Jump is not taken.	Specified by the c-field
1	The single bit value that is the mask bit corresponding to the element pointer value in the EL register pointed to by the current element loop pointer. If the element pointer is greater than 63, then a Vector Register Length Overflow condition exists and the Jump is not taken.	A single bit value of 1 produces a test result of true independent of the c-field.
2	The single bit has a value of 1 if and only if the current element count value is less than the full strip size.	A single bit value of 1 produces a test result of true independent of the c-field.
3	The single bit has a value of 1 if and only if the current value of the length parameter specified by the instruction I-field is zero.	A single bit value of 1 produces a test result of true independent of the c-field.
4	The 2-bit Scalar Condition Code (SCC).	Specified by the c-field.

NOTE: The values s=5, r=4 tests the value of the SCC itself, not the values of the operands that were compared to generate the SCC values. However the SCC has been defined such that if OP2 has value +0 in the Compare instruction; then the SCC will not only encode but will in fact duplicate the state of OP1 of the Compare. For example, an OP1 value that is positive non-zero will result in an SCC encoding of 01, which is itself positive non-zero, etc. The only exception is -0 which is encoded as if it were +0.

The s-field and r-field operand source selection, the c-field test condition, and the n-field true or false branch selection combine to select a single condition jump instruction. Tables 4-4 and 4-5 give all of the conditional jump instructions and the specific field values that select each instruction for n-field values of 0 (true) and 1 (false), respectively.

Table 4-4. Field Selection of Conditional Jump Instructions for n-field Equals 0

s-field/ r-field values	c-field value							
	0	1	2	3	4	5	6	7
s=0	J	JAB	JNB	JZ	JLZ	JHB	JLEZ	JLB
s=1		DJAB	DJNB	DLZ	DJLZ		DJLEZ	DJLB
s=5, r=0		JMAB	JMNB	JMXB				
s=5, r=1						JMCE		
s=5, r=2						JNFS		
s=5, r=3						JLRZ		
s=5, r=4			JCE, JTCC			JCL		

NOTE: Instruction mnemonics are defined in Appendix A.

Table 4-5. Field Selection of Conditional Jump Instructions for n-field Equals 1

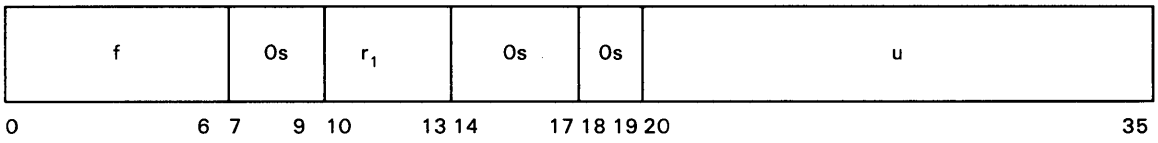
s-field/ r-field values	c-field value							
	0	1	2	3	4	5	6	7
s=0	NOP	JNAB	JNNB	JNZ	JGEZ	JNHB	JGZ	JNLB
s=1		DJNAB	DJNNB	DJNZ	DJGEZ		DJGZ	DJNLB
s=5, r=0		JMNAB	JMNNB	JMNXB				
s=5, r=1						JMNCE		
s=5, r=2						JZS		
s=5, r=3						JLNRZ		
s=5, r=4			JCNE, JTCS			JCGE	JCG	

NOTE: Instruction mnemonics are defined in Appendix A.

4.17.2. Increment and Jump Less (IJL)

The value in bits 36–71 of the G–register specified by r_1 is added to bits 0–35 of that register. The new value in bits 0–35 is then compared to the value in bits 0–35 of the G–register specified in r_2. If OP1 is less than OP2, then the jump is taken; otherwise execution proceeds with the next instruction. All values are single–precision integers. The jump is segment local. Negative zero is considered equal to positive zero in the comparison. If the incrementation of OP1 produces an overflow condition, an interrupt is not caused, the truncated result is stored, and execution proceeds to the next sequential instruction regardless of the OP2 value.

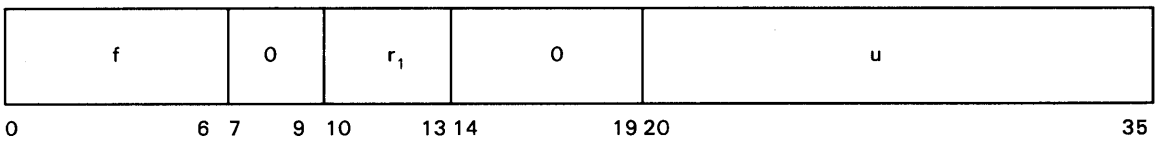
RS Format



4.17.3. Decrement and Jump Greater (DJG)

The integer value in bits 0–35 of the G–register specified by r_1 is decreased by one. If the new value is greater than zero, the jump is taken; otherwise execution proceeds to the next sequential instruction. If the decrementation of OP1 produces an overflow condition, no interrupt is caused, the truncated result is stored, and execution proceeds to the next instruction. The jump is segment local.

RS Format



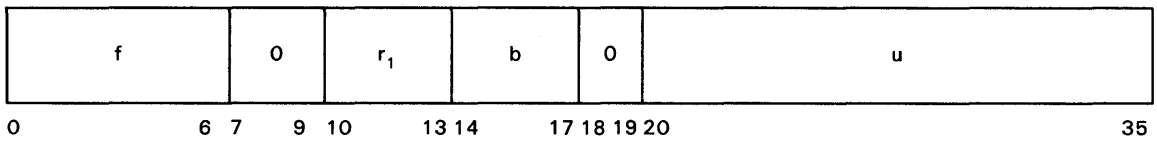
4.17.4. Load Address and Jump (LAJ, LANI)

For b–field values other than zero, the target instruction virtual address is formed by combining the u–field value and the virtual address from bits 0–35 of the G–register specified by the b–field. The virtual address of the location immediately following the LAJ is placed into bits 0–35 of the G–register specified by r_1, and the target virtual address is placed into the Program Address Register.

The LAJ itself does not do any limits checking or address translation; it merely computes a virtual address and places it into PAR. Address translation and access privilege faults, if any, occur in attempting to fetch the instruction to which the jump was made. However, to aid software in handling address faults, the NLJ (non–local jump) indicator bit is provided in word 11 (hardware status register 3) of the SPCB. This bit is set by execution of an LAJ instruction that loads PAR or any JXS. The bit is cleared by any other instruction execution.

When the b-field contains zero, the u-field is not used, PAR is not loaded (i.e., the jump is not done), but the next instruction address is placed into $G(r_1)$. This special case is useful for a subroutine establishing initial addressability of a group of constants without requiring any of them to be located in local storage.

RS Format



4.17.5. Jump to External Segment (JXS, JXSI)

This instruction is similar to the Load Address and Jump instruction but does not store the return address. Hence the r_1 field is not used. When the b-field is 0, the u-field is not used. Instead, the jump address is taken from State register 12. This provides a means for returning from internal interrupts after restoring all G-registers. The NLJ indicator is set by execution of this instruction as described for the Load Address and Jump instruction.

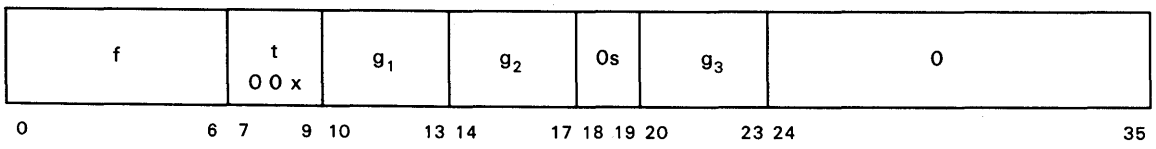
4.18. State Instructions

The state instructions are used for saving, restoring, or altering the general program state of an activity.

4.18.1. Load Multiple (LGM, DLGM)

This instruction loads a block of one or more G-registers from consecutive words of storage.

RR Format



Registers $G(g_1)$ through $G(g_2)$ are loaded from consecutive locations in storage beginning at the virtual address contained in $G(g_3)$ at the beginning of execution. Operation is not affected should $G(g_3)$ happen to be in the range of registers loaded by the instruction. $G(g_3)$ is not altered by being used as an address, but only if it happens to be in the range of registers loaded.

Register numbering is considered to be circular, wrapping around from 15 to 0. If $g_1=g_2$ then one register is loaded.

If t=0, consecutive single words are loaded into the left 36 bits of each of the G-registers in the specified range. If t=1, consecutive double words are loaded into the full 72 bits of the selected G-registers. In the latter case, the storage address must be an even address.

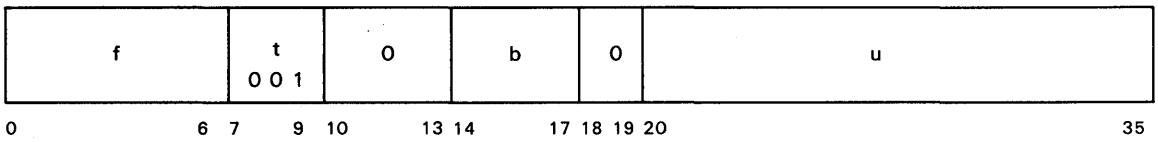
4.18.2. Store Multiple (SGM, DSGM)

The Store Multiple instructions are the inverse of the Load Multiple instructions. The Store Multiple instructions write registers $G(g_1)$ through $G(g_2)$ to consecutive storage locations beginning at the virtual address in $G(g_3)$.

4.18.3. Store Loop Control Registers (SLCR)

The eight Vector Loop and eight Element Loop registers provide space for parameters describing eight DO-loops in various stages of execution, which is sufficient for most subroutines individually. However, when calling another subroutine it is generally desirable to preserve certain parameters and to make available space for the called subroutine to use.

RS Format



This instruction stores eight double words beginning at the storage location specified by the b- and u-fields. This block contains the Vector Loop registers, Element Loop registers, CVLP, and the current element loop pointer in the format. The storage address must be an even address.

4.18.4. Load Loop Control Registers (LLCR)

This instruction is the inverse of SLR. It loads the Vector Loop and Element Loop registers, CVLP, and the current element loop pointer from the block of eight double words addressed by the b- and u-fields. The instruction format is identical. The storage address must be even.

4.18.5. Advance Local Storage Stack (ALSS)

This instruction advances the local storage stack to the next frame by adding one to the value in the Pointer field of the local storage stack definition word in register S7. Following the addition, the Pointer value is compared to the Upper Bound (UB) value. If Pointer is greater than UB or if the addition overflowed the Pointer field, then an internal interrupt is caused. Instruction format consists of a 7-bit OP code followed by 29, 0s.

4.18.6. Retract Local Storage Stack (RLSS)

The Retract is the inverse of the Advance. It subtracts one from the Pointer field and then compares Pointer to the Lower Bound (LB) value. If Pointer is less than LB or if the subtraction underflowed the Pointer field, then an internal interrupt is caused. Instruction format consists of the same OP code as ALSS followed by a 1 and 28, 0s.

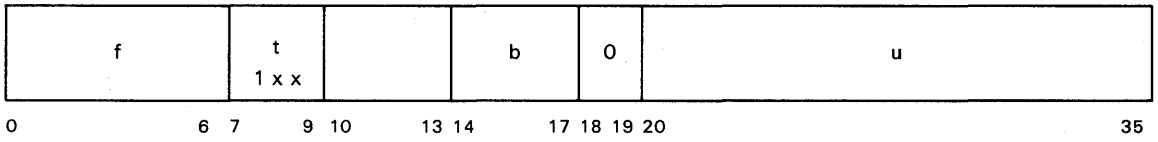
4.18.7. Generate Interrupt (GI, GIA, GIB)

An external interrupt is caused by this instruction to switch out this activity. The instruction format consists of a 7-bit OP code followed by three 0s. Use is roughly analogous to the ER instruction of the Series 1100. Bits 10–35 of the instruction need not be zeroes and may contain information for the operating system.

4.18.8. Test and Set (TS)

Bit 5 of the storage operand specified by the b- and u-fields is tested and its state is reflected in the resultant setting of the scalar condition code (SCC). SCC is set to 01 if the bit was one and to 00 if the bit was 0. The binary value 000001 is written into bits 0–5 of the storage operand regardless of the test results. Moreover the testing and setting are logically indivisible, meaning that no other processor or I/O unit can access the storage location between the testing and the setting. Bits 6–35 are not examined and are never altered by this instruction. This instruction is intended to be used for the synchronization of multiple real processors sharing data, so it is meaningful only for accesses to shared (main) storage. Therefore, any reference to the local storage segment by any means results in an interrupt (3.2.3).

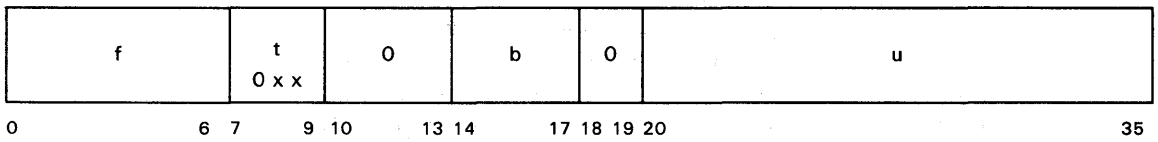
RS Format



4.18.9. Test and Clear (TC)

This instruction differs from Test and Set only in that the value written into bits 0–5 of the storage operand is all zeroes. All other statements apply.

RS Format



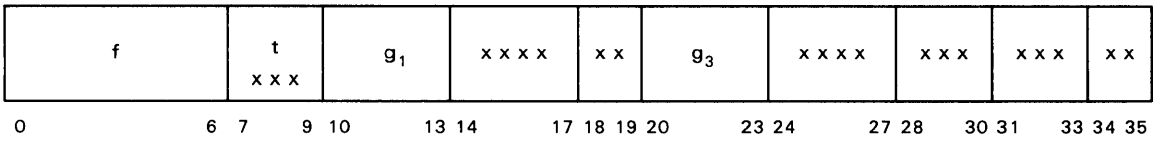
4.19. Diagnostic Instructions

The scientific processor system has two diagnostic instructions; Diagnose Read and Diagnose Write. Both instructions are used for the scientific processor-to-scientific storage interface. These instructions do not require any special mode of execution.

4.19.1. Diagnose Read (DGR)

The Diagnose Read instruction is used to test the storage read interface. It uses the following RS instruction format.

RS Format



A read request is issued to the storage location specified by the virtual address in the G-register specified by the g_3-field of the instruction with a function code to the scientific storage indicating a Diagnose Read. If the scientific storage is properly prepared, it performs the request but will deliberately place bad parity on some or all of the read data. Upon receipt of bad parity, the scientific processor will immediately cause an external interrupt. However, if bad parity is not detected, the corrupted read data is placed into the G-register specified by the g_1-field of the instruction and execution proceeds to the next instruction.

Execution of this instruction causes the Diagnose Instruction Executed indicator (Bit 29 of hardware status register 3) to be set.

4.19.2. Diagnose Write (DGW)

This instruction is identical in format and similar in intent to the Diagnose Read instruction. It attempts to store a word of data from G (g_1) into the storage location whose virtual address is in G(g_3). The scientific processor deliberately places bad parity on all parity bits of the address lines, function code, and write data lines going to the scientific storage. Upon detection of bad parity, the scientific storage causes an external interrupt in the processor.

The bad parity is not actually written into the scientific storage. If the referenced storage location happens to be local storage, bad parity is not caused. Since the error indication from the scientific storage may be delayed for an unspecified period of time, a test program using a Diagnose Read or Write instruction must not use a GI instruction until sufficient time has elapsed to ensure that the error indication and status have been received.

Execution of this instruction causes the Diagnose Instruction Executed indicator (Bit 29 of hardware status register 3) to be set.

5. Scientific Processor Storage

This section describes general scientific storage operations.

5.1. Introduction

The scientific processor storage unit is a free-standing unit with eight storage banks. Each bank contains 524,288 (524K) words with each bank reference capable of accessing four words per reference cycle. Each word has 44 bits (36 data, 6 check, 1 check parity, and 1 data parity). One scientific storage unit contains a maximum of 4,194,304 (4194K) words of random access storage. Up to four scientific storage units can be used on a system for a total of 16,777,216 words of storage. Figure 5-1 shows the scientific processor storage unit cabinet.

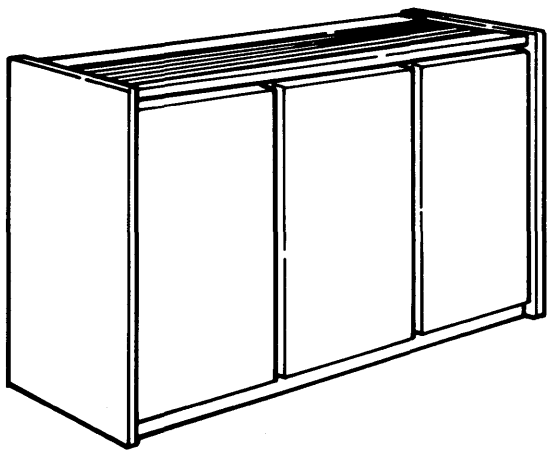


Figure 5-1. Scientific Processor Storage Unit

5.2. Storage Features

The scientific storage unit replaces or complements the main storage units. The scientific storage units can be treated identically for the purpose of executing Series 1100 code. The scientific storage unit is directly addressable by the scientific processor, instruction processor, and input/output processor. Both the scientific storage and the main storage units may be used in an scientific processor system. However, the scientific processor only interfaces with the scientific storage unit.

The data and addressing formats used with the scientific storage interfaces to the instruction processor and input/output processor are identical to those of the main storage unit. However, the internal addressing format differs due to the wider bank data interface (4 words) and different functional capability.

The scientific storage provides the higher bandpass required to support the speed of the scientific processors. Thus, the scientific processors can only access code or data loaded into the scientific storage units. This allocation is transparent to the applications programmer. The scientific storage provides the interfaces for up to two scientific processors, and up to four instruction processors and four input/output processors.

A multiple unit adapter (Section 6) is required when two or more scientific storage units are to be accessed by a scientific processor.

The scientific storage unit can have up to ten requester ports, consisting of:

- two scientific processor ports
- four instruction processor ports
- four input/output processor ports

The scientific storage unit has the following interfaces:

- instruction processor (up to four)
- input/output processor (up to four)
- scientific processor (up to two)
- system support processor (two)
- system panel
- system clock
- other main storage units in the system

5.3. Storage Functions

The scientific storage units appear to the instruction processor and input/output processors as if they are main storage units providing identical functions and preserving essential timing. The functions provided for instruction processor and input/output processor ports include: double-word read operations; partial-, single-, or double-word write operations, and block read or write operations (eight words per block) for instruction processors only. The partial-word write capability is bit addressable for variable length fields. The scientific processor ports provide four-word read operations; and one-, two-, three-, and four-word write operations.

5.4. Modes of Operation

Scientific storage unit operations are defined by a function code received in conjunction with a storage request. When a requester is granted priority, the scientific storage decodes the function code and initiates the requested operation.

Function codes for the instruction processor and input/output processor are given in Table 5-1.

Table 5-1. Instruction Processor and Input/Output Processor Function Codes

Function Code (Octal)	Type
02	Read Two Words
03	Read Block*
04	Read System Status Register*
05	Read Dayclock*
06	Test and Set
07	Test and Clear
10	Write One Word Partial
11	Write One Word
12	Write Two Words
13	Write Block*
14	Load Dayclock*
15	Load Dayclock Comparator*
16	Load Error Function Register*
17	Read Bank Status Register*
20	Selected Load Path 0*
21	Selected Load Path 1*
22	Initiate Auto Recovery*
23	Reset Auto Recovery Timer*
24	Set Dayclock Mode Normal*
25	Set Dayclock Mode Fast*
26	Set Dayclock Mode Slow*
31	Maintenance Write One Word
32	Maintenance Two-Word Read
33	Maintenance Read Block*
34	Read Dayclock Comparator*

**indicates IP function only.*

5.4.1. Write Functions

Write functions consist of partial-word, full-word, double-word, and block write operations.

Write One Word Partial (10_8)

An even or odd word partial write operation is performed within a word boundary as determined by address bit 23. The Start and End fields determine the bits within the word that are altered.

If address bit 23 is 0, the even word is altered and the Start and End field values of 0 through 35 represent data bit positions 0 through 35. If address bit 23 is 1, the odd word is altered and the Start and End field values of 0 through 35 represent data bit positions 36 through 71.

Write One Word (11_8)

A full-word write is performed. If address bit 23 is 0, the even word (bits 0 through 35) is altered. When address bit 23 is 1, the odd word (bits 36 through 71) is altered.

Write Two Words (12_8)

A double-word write operation is performed. Address bit 23 is ignored, and an even and an odd word is stored.

Write Block (13_8)

A write operation which alters eight consecutive locations which do not cross eight-word block boundaries. Address bits 21, 22, and 23 are ignored. The first two words of data are sent with the request. When the ACKNOWLEDGE 1 signal is received, the remaining three double words of write data is sent to the scientific storage in intervals.

5.4.2. Read Functions

Read functions consist of double-word read and eight-word read block operations.

Read Two Words (02_8)

A double word read operation is performed. Address bit 23 is ignored. The data is placed on the read data interface lines.

Read Block (03_8)

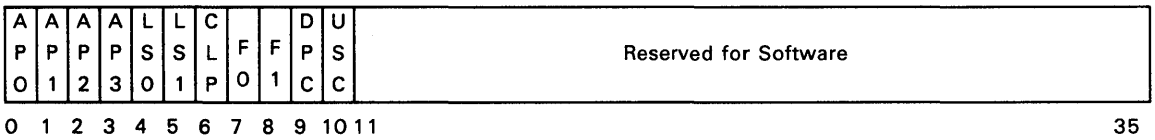
An eight-word read block operation is performed on block boundaries. Address bits 21, 22, and 23 are ignored by the scientific storage.

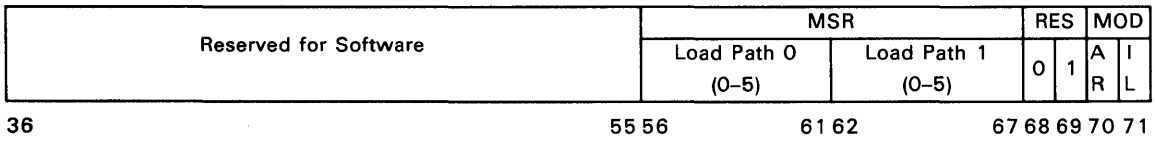
5.4.3. Status Functions

Status functions consist of reading the system status register and the bank status register.

Read System Status Register (04_8)

The scientific storage reads the System Status register and places the register contents on the read data lines. The system status register uses the following format:





where:

Bits 0–10 Application bits

- 0 – Application 0
- 1 – Application 1
- 2 – Application 2
- 3 – Application 3
- 4,5 – 00 – Load source is SSP IPL
  01 – Load source is system panel
  10 – Load source is auto recovery timer
  11 – Load source is requester IAR
- 6 – Current load path is 1, load path 0 cleared.
- 7,8 – 01 – Failed on first attempt, loaded on second attempt.
  10 – Failed on first and second attempts, loaded on third attempt.
  11 – Failed on first, second, and third attempts, loaded on fourth attempt.
- 9 – Dynamic partitioning change after status register was loaded
- 10 – Unit support controller fault.

Bits 11–55 Reserved for software.

Bits 56–67 MSR – Load path 0 and 1

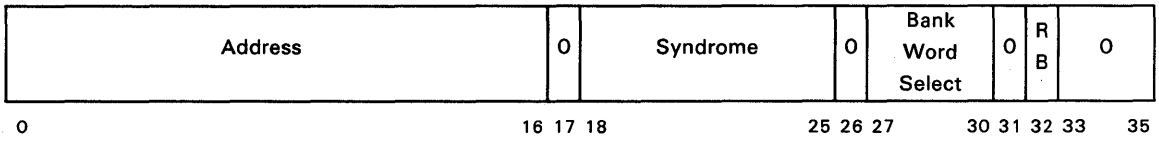
Bits 68,69 RES – Load path resident in storage

Bits 70,71 MOD – AR – Auto recovery enabled
IL – 8 – Bank interleave enabled

Read Bank Status Register (17_8)

The scientific storage places the contents of the selected Bank Status register on the requesters read interface lines. Each bank of scientific storage contains four status registers (one per data word). The request is directed to the register pair selected by address bit 22. Address bits 2, 20, and 21 (four bank interleave) or 19, 20, and 21 (eight bank interleave) select the bank.

The Bank Status register uses the following format for single bit errors:



where:

Bits 0–16 Address bits

0,1,2 – Row selected
3–16 – 16K address

Bits 18–25 Syndrome Code bits that point at the bit in error. These bits are 0s for no failure.

Bits 27–30 Bank/Word Select

27,28,29 – Bank selected
30 – Word pair selected

Bit 32 RB – Resident Bit

The resident bit is set when data is loaded and cleared when the requester reads out the data.

5.4.4. Dayclock Functions

The dayclock provides accurate elapsed time measurements and initiates system activities at a preselected time. The dayclock consists of a counter and a comparator.

Read Dayclock (05_8)

The scientific storage reads the current value of the dayclock and places the value on the read interface lines.

Load Dayclock (14_8)

The requester places the dayclock value to be loaded on the write data lines. The scientific storage stores the value in the dayclock and updates the dayclock from that value.

Load Dayclock Comparator (15_8)

The requester places the dayclock comparator value on the write data lines. The scientific storage stores the value in the comparator, and then broadcasts an interrupt whenever the dayclock value is equal to or greater than the value in the comparator.

Read Dayclock Comparator (34_8)

The scientific storage places the comparator value on the read data lines in the same bit locations as received.

Select Dayclock Rates

The dayclock rate can be selected to correct for clock variations by the following functions:

- Set Dayclock Mode Normal (24_8)
- Set Dayclock Mode Fast (25_8)
- Set Dayclock Mode Slow (26_8)

5.4.5. Auto Recovery Timer

The auto recovery timer monitors the system. If no Reset Auto Recovery Timer signal is received within a certain interval the timer initiates an auto recovery sequence.

Select Load Path 0 (20_8)

Select load path 0 forces the scientific storage to select auto recovery path 0 as the active load path.

Select Load Path 1 (21_8)

Select load path 1 forces the scientific storage to select auto recovery path 1 as the active load path.

Initiate Auto Recovery (22_8)

Simulates an immediate expiration of the auto recovery timer which forces an auto recovery initial confidence load program load attempt on the designated load path.

Reset Auto Recovery Timer (23_8)

The scientific storage clears the auto recovery timer and restarts the countdown. This function must be performed periodically to prevent the recovery sequence from starting.

5.4.6. Test and Set Functions

Test and set functions consists of Test and Set and Test and Clear. These two functions are the same, except upon completion, bit 5 (even words) and bit 41 (odd words) contain a 1 bit for the test and set function and a 0 bit for the test and clear function.

Test and Set (06_8)

The scientific processor performs a read operation followed by a partial write operation within one word boundary. The scientific processor provides the data to be written.

Test and Clear (07_8)

The test and clear operation is the same as for test and set, except that during the write operation 0's are written into data bits 0 through 5 (even words) and 36 through 41 (odd words).

5.4.7. Scientific Processor Functions

Function codes received on the scientific processor ports direct the scientific storage to perform operations for the processor. Table 5–2 shows the scientific processor function codes.

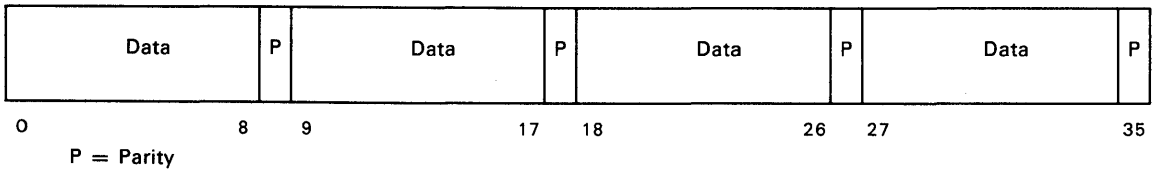
Table 5–2. Scientific Processor Function Codes

Function Code (Binary)	Type
00 0000	Read four words
00 0001	Write Word specified by Master Bit*
.	
.	
.	
00 1111	
01 0001	Test and Set specified by Master Bit**
.	
.	
.	
01 1000	

Table 5–2. Scientific Processor Function Codes (continued)

Function Code (Binary)	Type
10 0000	Maintenance read
11 0001	Test and Clear specified by Master Bit**
.	
.	
.	
11 1000	
*indicates any combination is legal.	
** only one may be selected.	

The scientific processor data word consists of four 36-bit words with four parity bits in each word. Word 1 is the most significant word and word 4 is the least significant word. An example of one word follows:



Read Four Words ($00–0000_2$)

The scientific storage reads the four words specified by address bits 2 through 21. The data is placed on the read data interface.

Write 1–4 Words ($00–XXXX_2$)

The scientific storage writes one word for each bit set (master bits). Any combination of the four words may be written. The function code ($XXXX_2$) can be any combination of 0000 to 1111.

Test and Set ($01–XXXX_2$)

The scientific processor performs a read operation followed by a partial write operation for the word specified by the master bit (only one bit is set at a time).

Test and Clear (11-$XXXX_2$)

This operation is the same as the test and set operation, except that during the write, 0's are written in data bits 0 through 5.

5.5. Address Translation

Address translation is the process of converting the relative operand address provided by an instruction to the absolute storage address of the operand, including verifying that the operand address is within the address space available to the instruction.

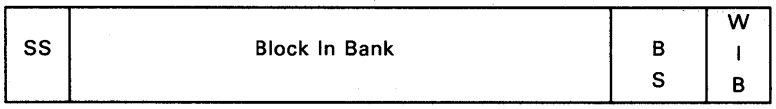
Address translation uses the following formats for scientific storage:

Four-Bank Interleave

where:

Bits 0,1	SS –Scientific Storage Select
Bit 2	BQS – Bank Quad Select
Bits 3–19	Block in Bank
Bits 20,21	BIQ – Bank in Quad
Bits 22,23	WIB – Word In Bank

Eight-Bank Interleave

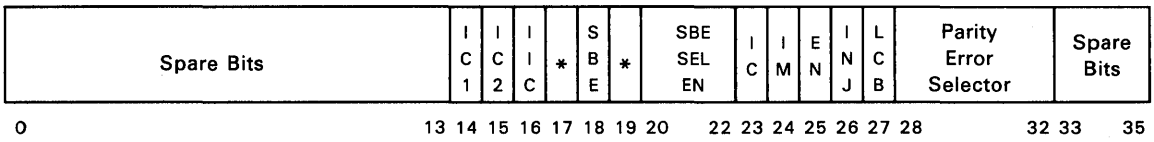


where:

Bits 0,1	SS – Scientific Storage Select
Bits 2–18	Block in Bank
Bits 19–21	BS – Bank Select
Bits 22–23	WIB – Word In Bank

5.6. Error Function Register

A load Error Function Register function (16_8) loads the Error Function register in the addressed bank pair to control the functions within the bank. The maintenance functions become sensitive only if the associated function register bits are enabled. The Error Function register uses the following format:



*indicates a spare bit

where:

Bit 14 IC1 – Partial Store Internal Check 1

Tests read data parity for a partial store when a maintenance read is requested.

Bit 15 IC2 – Partial Store Internal Check 2

Tests the partial store dual parity compare when a maintenance write one word is requested.

Bit 16 IIC – Inhibit Interface Check

Inhibits the interface parity check and allows the parity error to be undetected until it is propagated to the internal parity check.

Bit 18 SBE Lock

Inhibits all single–bit error reporting.

Bits 20–22 SBE Selective Enable

This field is enabled when bit 25 is set. Single–bit error reporting is selectively enabled on the word within the block for the bank–pair selected.

Bit 23 IC – Inhibit Correction

Inhibits correction on any word in error.

Bit 24 IM – Inhibit MUE

Inhibits sending multiple unit errors to the requester. This bit is set or cleared by the system support processor.

Bit 25 EN – Enable SBE

Enables bits 20–22.

Bit 26 INJ – Inject Parity

Enables bits 28–32 to inject a parity error in read data to the requester.

Bit 27 LCB – Lock Check Bits

The scientific storage does not write the check bits for a maintenance write one word.

Bits 28–32 Parity Error Selector

These bits select the byte in which a parity error occurs to the requester.

5.7. Error Reporting

Error reporting consists of checking parity on interface lines, and reporting external and internal errors.

5.7.1. Parity Checking

Parity is checked on lines from the scientific storage unit to the instruction processor, input/output processor, and the scientific processor.

Instruction Processor–Input/Output Processor to Scientific Storage

The write data word contains four 9–bit data bytes with one parity bit for each byte. The address word has four fields with one parity bit for each field. The control word has a parity bit for each group of control bits within the word as follows:

- function code
- start bit
- end bit

These parity bits determine if an error has occurred in a control word, address word, or the write data word during data transfer operations.

Scientific Processor to Scientific Storage

The data word contains four 9–bit bytes with one parity bit for each byte. The address word has three fields with one parity bit for each field, and the function word has one parity bit.

Scientific Storage Unit to Requester

One parity bit is generated for each 9–bits of the read data word, which allows the requester to check parity on errors that occurred during data transfer operations.

5.7.2. External Errors

External errors consist of interface parity errors, bank-not-available errors, and unsigned function errors.

Interface Parity Errors

Odd parity is maintained on all interfaces for all functions. Parity is checked on write data, function code, address, start field, and end field interface information. If an error is detected, the following events occur:

- the cycle is aborted
- an interface check is issued by the scientific storage unit, with the ACKNOWLEDGE 1 signal
- the interface error message is transmitted across the read data interface

A bank not available check is issued by the scientific storage unit if an error occurs on the address lines. This check preempts an address parity error.

Bank Not Available

Bank not available check is generated when an address is not in the requester's application, from partitioning, program error, or a hardware failure. Bank not available check is also generated if the scientific storage unit is partitioned to an eight-bank interleave and banks 0 through 3 or banks 4 through 7 are partitioned to maintenance.

Unassigned Functions

If the requester sends a function that is an unassigned octal code, the scientific storage unit performs a read operation. A function code parity error is indicated.

5.7.3. Internal Errors

Internal errors consist of single-bit errors, multiple bit errors, and partial-write errors detected in stored data.

Single-Bit Error

Each 36-bit data word is single-bit error corrected. When a single-bit error is detected and corrected within a bank, a STATUS CHECK signal is transmitted to the instruction processor within the application. The selected instruction processor responds to the scientific storage unit with a STATUS CHECK ACKNOWLEDGE signal.

Multiple Errors

Double-bit and multiple-bit read data errors that are decoded as undefined errors are defined as multiple uncorrectable errors. Read data is modified by error correction and cannot be used for error detection by the requester.

Partial-Write Errors

The partial write single-bit error read word is corrected and merged with the selected write data. Any single-bit error causes a status check to be broadcast to ports 4 through 7 in the application. Multiple bit errors in stored data causes the write cycle to abort.

5.8. Configuration

Each scientific storage unit used replaces an existing main storage unit. The two types of storage units can be intermixed, but the scientific processor can only interface with the scientific storage. If two scientific storage units are used, a multiple unit adapter is required. A second multiple unit adapter is required if a second scientific processor and two to four scientific storage units are used. Storage configurations and address interleave are given in Appendix B.

6. Multiple Unit Adapter

This section describes the relation of the multiple unit adapter with the scientific processor system.

6.1. Introduction

The multiple unit provides the interface between a scientific processor and one to four scientific processor storage adapter units. Scientific processor systems with two or more scientific storage units require the multiple unit adapter for each scientific processor. The adapter may also be used in a system with one scientific storage unit. Figure 6–1 shows the multiple unit adapter cabinet.

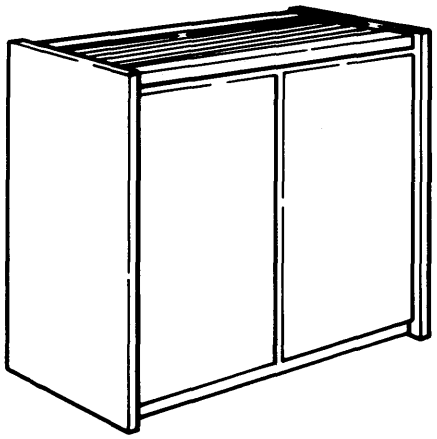


Figure 6–1. Multiple Unit Adapter

6.2. Request Stacking

The adapter provides an eight deep stack for scientific processor requests, and the associated address, write data, and function. The requests are taken off the stack in sequential order, that is first in first out.

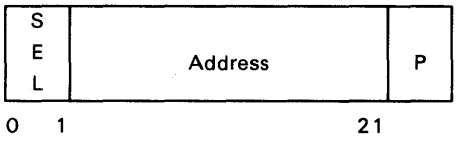
6.3. Request Acknowledgment

When the adapter sends a scientific processor request to the scientific storage, an acknowledgment is sent to the scientific processor. The acknowledgment informs the scientific processor that it can issue another request to the adapter.

6.4. Select Word Format

The adapter decodes the scientific processor most significant address bits (bits 0 and 1 of the scientific processor address format) to determine which one of the four scientific storage units to request.

The select word format is:

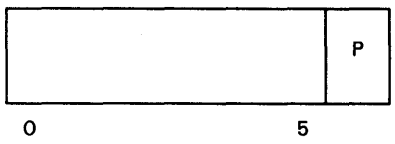


Bits 0 and 1 are the select bits.

The adapter continues requesting a scientific storage unit until the request stack is empty, eight storage requests remain outstanding (eight requests is the maximum that a scientific storage can stack), or a different scientific storage is decoded. When a different scientific storage unit is decoded, the adapter waits until an acknowledgement for the last storage request is received before requesting a different storage. This insures that the scientific storage references are not allowed to be serviced out of order.

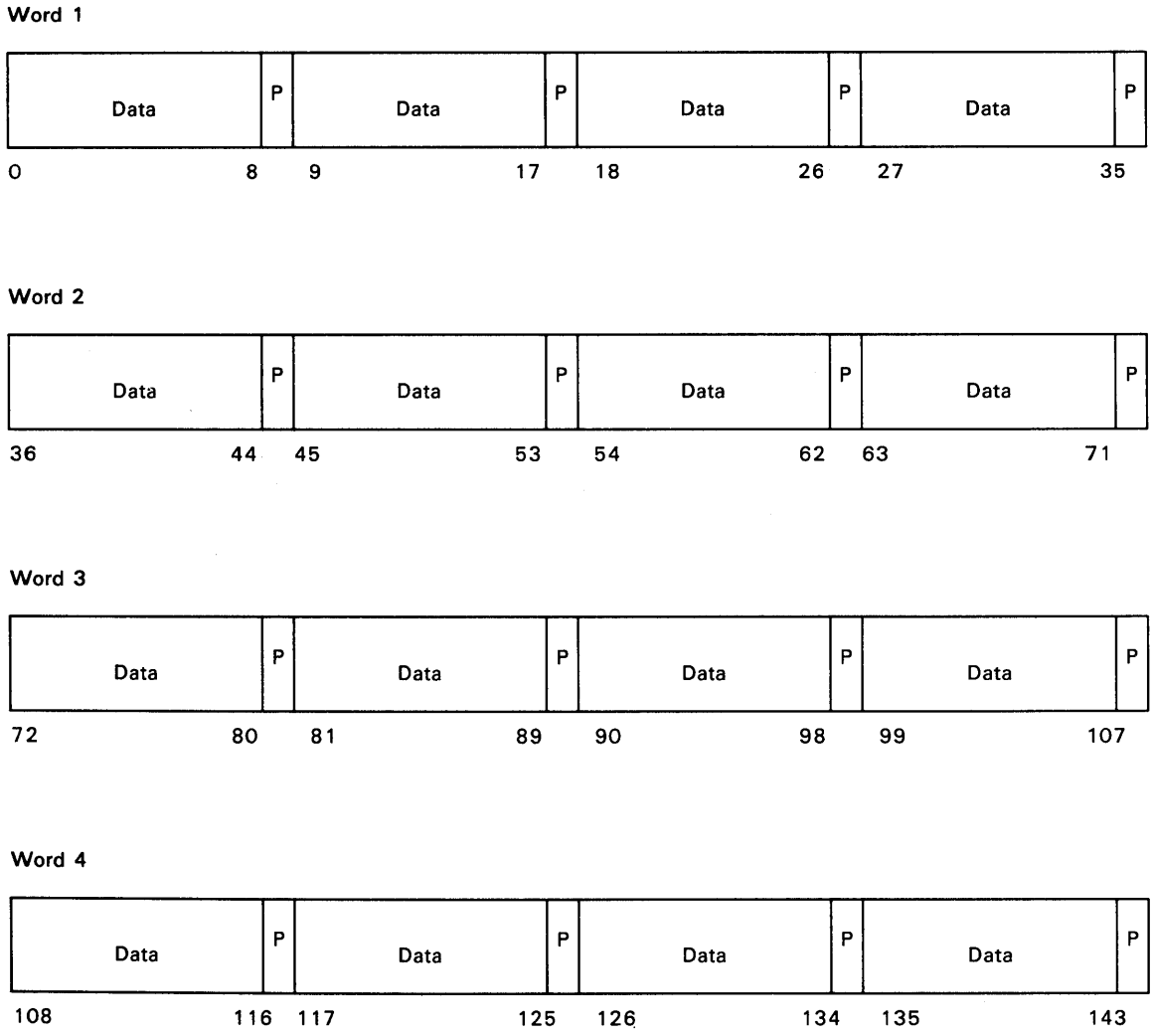
6.5. Function Word Format

The function word format is:



6.6. Write Data Format

The write data format is:



6.7. Read Data Format

The read data format is the same as the four words of the write data format in 5.7.

6.8. Parity

Parity is checked on these fields:

- Scientific processor address
- Scientific processor write data
- Scientific processor function
- Scientific processor storage read data

6.9. External Errors

Parity bits are checked from both the scientific processor and the scientific storage.

Scientific Processor to Multiple Unit Adapter

The adapter checks the parity bits of the address, write data, and function fields from the scientific processor and generates an interface check to the scientific processor when an error is detected.

The adapter passes the parity incorrect field to the scientific storage that generates an interface check. The parity error information is returned on the read data lines.

Scientific Processor Storage to Multiple Unit Adapter

The adapter checks the parity bits on read data from the scientific storage. If an error is detected, an interface check is generated and sent to the scientific processor.

6.10. Partitioning

The adapter can be partitioned into the scientific processor's application or partitioned into maintenance mode. The scientific storage units are partitioned into the adapter's application.

Appendix A. Instruction Summary

This appendix lists the scientific processor instructions by mnemonic (A.1) and by function codes (A.2). Detailed descriptions of the instructions are given in Section 4.

The instruction Opcode is shown in a three-character octal decode of the most significant seven binary bits of the 36-bit instruction word followed by the one-character octal decode of the next three binary bits generally known as the t-field. For decodes showing more than two fields, refer to Section 4. For more detailed information, refer to the *Meta-Assembler for the Scientific Processor, MASP Reference,* UP-10985.

A.1. Instruction Listing by Mnemonic

Mnemonic	Format	Opcode
A	RS	102 0
ALSS	RS	160 0
AN	RS	103 0
AND	RS	106 0
ANDR	RR	146 0
ANDV	VV	046 0
ANR	RR	143 0
ANV	VV	043 0
AR	RR	142 0
AV	VV	042 0
BEL	RS	123 2
BLVL	RS	131 5
BSVL	RS	131 6
BVLR	RS	161 5
C	RS	101 0
CDFDIR	RR	150 3 1
CDFDIV	VV	050 3 1
CDFFR	RR	150 3 2
CDFFV	VV	050 3 2
CDFIR	RR	150 3 0
CDFIV	VV	050 3 0
CDIDFR	RR	150 1 3
CDIDFV	VV	050 1 3
CDIFR	RR	150 1 2
CDIFV	VV	050 1 2
CDIIR	RR	150 1 0
CDIIV	VV	050 1 0
CELP	RS	163 1
CEV	VV	041 0
CFDFR	RR	150 2 3
CFDFV	VV	050 2 3
CFDIR	RR	150 2 1
CFDIV	VV	050 2 1
CFIR	RR	150 2 0
CFIV	VV	050 2 0
CIDFR	RR	150 0 3
CIDFV	VV	050 0 3
CIDIR	RR	150 0 1
CIDIV	VV	050 0 1
CIFR	RR	150 0 2
CIFV	VV	050 0 2
CLEV	VV	041 0
CLV	VV	041 0

Mnemonic	Format	Opcode
CNEV	VV	041 0
CR	RR	141 0
CVELP	RS	163 3
DA	RS	102 1
DAN	RS	103 1
DAND	RS	106 1
DANDR	RR	146 1
DANDV	VV	046 1
DANR	RR	143 1
DANV	VV	043 1
DAR	RR	142 1
DAV	VV	042 1
DBVLR	RR	161 6
DC	RS	101 1
DCEV	VV	041 1
DCLEV	VV	041 1
DCLV	VV	041 1
DCNEV	VV	041 1
DCR	RR	141 1
DEM	RS	111 1
DEMR	RR	151 1
DEMV	VV	051 1
DESC	RS	112 1
DESCR	RR	152 1
DESCV	VV	052 1
DFA	RS	102 3
DFAN	RS	103 3
DFANR	RR	143 3
DFANV	VV	043 3
DFAR	RR	142 3
DFAV	VV	042 3
DFD	RS	105 3
DFDR	RR	145 3
DFDV	VV	045 3
DFM	RS	104 3
DFMR	RR	144 3
DFMV	VV	044 3
DFPRD	VV	004 3
DFSUM	VV	002 3
DGR	RS	173 0
DGW	RS	177 0
DI	RS	105 1
DIR	RR	145 1
DJAB	RS	122 1 0 1
DJG	RS	121 0
DJGEZ	RS	122 1 1 4
DJGZ	RS	122 1 1 6
DJLB	RS	122 1 0 7
DJLEZ	RS	122 1 0 6
DJLZ	RS	122 1 0 4
DJNAB	RS	122 1 1 1
DJNB	RS	122 1 0 2
DJNLB	RS	122 1 1 7
DJNNB	RS	122 1 1 2
DJNZ	RS	122 1 1 3
DJZ	RS	122 1 0 3
DL	RS	130 1
DLAEV	VV	072 1
DLGM	RR	172 1
DLV	VV	070 1
DLVX	VV	073 1
DMAX	VV	006 1
DMCV	VV	022 1
DMDV	VV	023 1
DMIN	VV	007 1
DMNS	RR	155 1
DMNV	VV	055 1
DMS	RR	154 1
DMV	VV	054 1
DOR	RS	106 3
DORR	RR	146 3
DORV	VV	046 3
DS	RS	134 1
DSA	RS	107 3
DSAEV	VV	076 1
DSAR	RR	147 3
DSAV	VV	047 3
DSGM	RR	176 1
DSI	RS	105 0
DSIR	RR	145 0
DSIV	VV	045 0
DSL	RS	107 1
DSLR	RR	147 1
DSLV	VV	047 1
DSUM	VV	002 1
DSV	VV	074 1
DSVX	VV	077 1
DXOR	RS	106 5
DXORR	RR	146 5
DXORV	VV	046 5
EBCV	VV	056 0
EBPV	VV	057 0
EM	RS	111 0
EMR	RR	151 0
EMV	VV	051 0
ESC	RS	112 0
ESCR	RR	152 0
ESCV	VV	052 0

Mnemonic	Format	Opcode
FA	RS	102 2
FAN	RS	103 2
FANR	RR	143 2
FANV	VV	043 2
FAR	RR	142 2
FAV	VV	042 2
FD	RS	105 2
FDR	RR	145 2
FDV	VV	045 2
FM	RS	104 2
FMR	RR	144 2
FMV	VV	044 2
FPRD	VV	004 2
FSUM	VV	002 2
GI	RS	167 0
GXV	VV	060 0
IJL	RS	120 0
J	RS	122 0 0 0 0
JAB	RS	122 0 0 1
JCE	RS	122 5 0 2 4
JCG	RS	122 5 1 6 4
JCGE	RS	122 5 1 5 4
JCL	RS	122 5 0 5 4
JCLE	RS	122 5 0 6 4
JCNE	RS	122 5 1 2 4
JEL	RS	123 3
JFS	RS	122 5 1 5 2
JGEZ	RS	122 0 1 4
JGZ	RS	122 0 1 6
JHB	RS	122 0 0 5
JLB	RS	122 0 0 7
JLEZ	RS	122 0 0 6
JLRNZ	RS	122 5 1 5 3
JLRZ	RS	122 5 0 5 3
JLZ	RS	122 0 0 4
JMAB	RS	122 5 0 1 0
JMCE	RS	122 5 0 5 1
JMNAB	RS	122 5 1 1 0
JMNB	RS	122 5 0 2 0
JMNCE	RS	122 5 1 5 1
JMNNB	RS	122 5 1 2 0
JMNXB	RS	122 5 0 3 0
JMXB	RS	122 5 1 3 0
JNAB	RS	122 0 1 1
JNB	RS	122 0 0 2
JNFS	RS	122 5 0 5 2
JNHB	RS	122 0 1 5
JNLB	RS	122 0 1 7
JNNB	RS	122 0 1 2
JNZ	RS	122 0 1 3
JVL	RS	123 1
JXS	RS	124 0
JXSI	RS	124 0
JZ	RS	122 0 0 3
L	RS	130 0
LAEV	VV	072 0
LAJ	RS	125 0
LANI	RS	125 0
LDSC	RS	107 7
LDSCR	RR	147 7
LDSCV	VV	047 7
LDSL	RS	107 5
LDSLR	RR	147 5
LDSLV	VV	047 5
LGM	RR	172 0
LLCR	RS	132 1
LS	RS	130 2
LSSC	RS	107 6
LSSCR	RR	147 6
LSSCV	VV	047 6
LSSL	RS	107 4
LSSLR	RR	147 4
LSSLV	VV	047 4
LV	VV	070 0
LVX	VV	073 0
MAX	VV	006 0
MCV	VV	022 0
MDV	VV	023 0
MI	RS	104 1
MIN	VV	007 0
MIR	RR	144 1
MLIV	VV	044 1
MNS	RR	155 0
MNV	VV	055 0
MS	RR	154 0
MSI	RS	104 0
MSIR	RR	144 0
MSIV	VV	044 0
MV	VV	054 0
NOP	RS	122 0 1 0 0
OR	RS	106 2
ORR	RR	146 2
ORV	VV	046 2

Mnemonic	Format	Opcode
PRD	VV	004 0
RLSS	RS	160 4
S	RS	134 0
SAEV	VV	076 0
SGM	RR	176 0
SLCR	RS	136 1
SS	RS	134 2
SSA	RS	107 2
SSAR	RR	147 2
SSAV	VV	047 2
SSL	RS	107 0
SSLR	RR	147 0
SSLV	VV	047 0
SUM	VV	002 0
SV	VV	074 0
SVX	VV	077 0
TC	RS	137 0
TS	RS	137 4
XOR	RS	106 4
XORR	RR	146 4
XORV	VV	046 4

A.2. Instruction Listing by Function Code

Function Code	Mnemonic	Format	Instruction
002 0	SUM	VV	Sum Reduction
002 1	DSUM	VV	Double Sum Reduction
002 2	FSUM	VV	Floating-point Sum Reduction
002 3	DFSUM	VV	Double Floating Sum Reduction
004 0	PDR	VV	Product Reduction
004 2	FPDR	VV	Floating-point Product Reduction
004 3	DFPDR	VV	Double Floating Product Reduction
006 0	MAX	VV	Max Reduction
006 1	DMAX	VV	Double Max Reduction
007 0	MIN	VV	Min Reduction
007 1	DMIN	VV	Double Min Reduction
022 0	MCV	VV	Move and Compress Vector
022 1	DMCV	VV	Double Move and Compress Vector
023 0	MDV	VV	Move and Distribute Vector
023 1	DMDV	VV	Double Move and Distribute Vector
041 0	CEV	VV	Compare Equal Vector
041 0	CLEV	VV	Compare Less Than or Equal Vector
041 0	CLV	VV	Compare Less Than Vector
041 0	CNEV	VV	Compare Not Equal Vector
041 1	DCEV	VV	Double Compare Equal Vector
041 1	DCLEV	VV	Double Compare Less Than or Equal Vector
041 1	DCLV	VV	Double Compare Less Than Vector
041 1	DCNEV	VV	Double Compare Not Equal Vector
042 0	AV	VV	Add Vector
042 1	DAV	VV	Double Add Vector
042 2	FAV	VV	Floating Add Vector
042 3	DFAV	VV	Double Floating Add Vector
043 0	ANV	VV	Add Negative Vector
043 1	DANV	VV	Double Add Negative Vector
043 2	FANV	VV	Floating Add Negative Vector
043 3	DFANV	VV	Double Floating Add Negative Vector
044 0	MSIV	VV	Multiply Single Integer Vector
044 1	MLIV	VV	Multiply Left Half Integer Vector
044 2	FMV	VV	Floating Multiply Vector
044 3	DFMV	VV	Double Floating Multiply Vector
045 0	DSIV	VV	Divide Single Integer Vector
045 2	FDV	VV	Floating Divide Vector
045 3	DFDV	VV	Double Floating Divide Vector
046 0	ANDV	VV	AND Vector
046 1	DANDV	VV	Double And Vector
046 2	ORV	VV	OR Vector
046 3	DORV	VV	Double OR Vector
046 4	XORV	VV	Exclusive OR Vector
046 5	DXORV	VV	Double Exclusive OR Vector
047 0	SSLV	VV	Single Shift Logical Vector
047 1	DSLV	VV	Double Shift Logical Vector
047 2	SSAV	VV	Single Shift Algebraic Vector

Function Code	Mnemonic	Format	Instruction
047 3	DSAV	VV	Double Shift Algebraic Vector
047 4	LSSLV	VV	Left Single Shift Logical Vector
047 5	LDSLV	VV	Left Double Shift Logical Vector
047 6	LSSCV	VV	Left Single Shift Circular Vector
047 7	LDSCV	VV	Left Double Shift Circular Vector
050 0 1	CIDIV	VV	Convert Integer To Double Integer Vector
050 0 2	CIFV	VV	Convert Integer To Floating Vector
050 0 3	CIDFV	VV	Convert Integer To Double Floating Vector
050 1 0	CDIIV	VV	Convert Double Integer To Integer Vector
050 1 2	CDIFV	VV	Convert Double Integer To Floating Vector
050 1 3	CDIDFV	VV	Convert Double Integer To Double Floating Vector
050 2 0	CFIV	VV	Convert Floating To Integer Vector
050 2 1	CFDIV	VV	Convert Floating To Double Integer Vector
050 2 3	CFDFV	VV	Convert Floating To Double Floating Vector
050 3 0	CDFIV	VV	Convert Double Floating To Integer Vector
050 3 1	CDFDIV	VV	Convert Double Floating To Double Integer Vector
050 3 2	CDFFV	VV	Convert Double Floating To Floating Vector
051 0	EMV	VV	Magnitude Vector
051 1	DEMV	VV	Double Magnitude Vector
052 0	ESCV	VV	Sign Count Vector
052 1	DESCV	VV	Double Sign Count Vector
054 0	MV	VV	Move Vector
054 1	DMV	VV	Double Move Vector
055 0	MNV	VV	Move Negative Vector (Negate)
055 1	DMNV	VV	Double Move Negative Vector (Negate)
056 0	EBCV	VV	Extract Bit Count Vector
057 0	EBPV	VV	Extract Bit Parity Vector
060 0	GXV	VV	Generate Index Vector
070 0	LV	VV	Load Vector
070 1	DLV	VV	Double Load Vector
072 0	LAEV	VV	Load Alternate Elements Vector
072 1	DLAEV	VV	Double Load Alternate Elements Vector
073 0	LVX	VV	Load Vector Indexed
073 1	DLVX	VV	Double Load Vector Indexed
074 0	SV	VV	Store Vector (S1=0)
074 1	DSV	VV	Double Store Vector (S1=0)
076 0	SAEV	VV	Store Alternate Elements Vector
076 1	DSAEV	VV	Double Store Alternate Elements Vector
077 0	SVX	VV	Store Vector Indexed
077 1	DSVX	VV	Double Store Vector Indexed
101 0	C	RS	Compare
101 1	DC	RS	Double Compare
102 0	A	RS	Add
102 1	DA	RS	Double Add
102 2	FA	RS	Floating Add
102 3	DFA	RS	Double Floating Add
103 0	AN	RS	Add Negative
103 1	DAN	RS	Double Add Negative
103 2	FAN	RS	Floating Add Negative
103 3	DFAN	RS	Double Floating Add Negative

Function Code	Mnemonic	Format	Instruction
104 0	MSI	RS	Multiply Single Integer
104 1	MI	RS	Multiply Integer
104 2	FM	RS	Floating Multiply
104 3	DFM	RS	Double Floating Multiply
105 0	DSI	RS	Divide Single Integer
105 1	DI	RS	Divide Integer
105 2	FD	RS	Floating Divide
105 3	DFD	RS	Double Floating Divide
106 0	AND	RS	AND
106 1	DAND	RS	Double AND
106 2	OR	RS	OR
106 3	DOR	RS	Double OR
106 4	XOR	RS	Exclusive OR
106 5	DXOR	RS	Double Exclusive OR
107 0	SSL	RS	Single Shift Logical
107 1	DSL	RS	Double Shift Logical
107 2	SSA	RS	Single Shift Algebraic
107 3	DSA	RS	Double Shift Algebraic
107 4	LSSL	RS	Left Single Shift Logical
107 5	LDSL	RS	Left Double Shift Logical
107 6	LSSC	RS	Left Single Shift Circular
107 7	LDSC	RS	Left Double Shift Circular
111 0	EM	RS	Magnitude
111 1	DEM	RS	Double Magnitude
112 0	ESC	RS	Sign Count
112 1	DESC	RS	Double Sign Count
120 0	IJL	RS	Increment and Jump Less
121 0	DJG	RS	Decrement and Jump Greater
122 0 0 0	J	RS	Jump
122 0 0 1	JAB	RS	Jump All Bits – G
122 0 0 2	JNB	RS	Jump No Bits – G
122 0 0 3	JZ	RS	Jump Zero – G
122 0 0 4	JLZ	RS	Jump Less Than Zero – G
122 0 0 5	JHB	RS	Jump High Bit – G
122 0 0 6	JLEZ	RS	Jump Less Than or Equal To Zero – G
122 0 0 7	JLB	RS	Jump Low Bit – G
122 0 1 0	NOP	RS	No Operation
122 0 1 1	JNAB	RS	Jump Not All Bits – G
122 0 1 2	JNNB	RS	Jump Not No Bits – G
122 0 1 3	JNZ	RS	Jump Non Zero – G
122 0 1 4	DGEZ	RS	Jump Greater Than or Equal To Zero – G
122 0 1 5	JNHB	RS	Jump Not High Bits – G
122 0 1 6	JGZ	RS	Jump Greater Than Zero – G
122 0 1 7	JNLB	RS	Jump Not Low Bit – G
122 1 0 1	DJAB	RS	Double Jump All Bits – G
122 1 0 2	DJNB	RS	Double Jump No Bits – G
122 1 0 3	DJZ	RS	Double Jump Zero – G
122 1 0 4	DJLZ	RS	Double Jump Less Than Zero – G
122 1 0 6	DJLEZ	RS	Double Jump Less Than or Equal To Zero – G
122 1 0 7	DJLB	RS	Double Jump Low Bit – G

Function Code	Mnemonic	Format	Instruction
122 1 1 1	DJNAB	RS	Double Jump Not All Bits – G
122 1 1 2	DJNNB	RS	Double Jump Not No Bits – G
122 1 1 3	DJNZ	RS	Double Jump Non Zero – G
122 1 1 4	DGEZ	RS	Double Jump Greater Than or Equal To Zero – G
122 1 1 6	DJGZ	RS	Double Jump Greater Than Zero – G
122 1 1 7	DJNLB	RS	Double Jump Not Low Bit – G
122 5 0 1 0	JMAB	RS	Jump Mask All Bits
122 5 0 2 0	JMNB	RS	Jump Mask No Bits
122 5 0 2 4	JCE	RS	Jump On Condition Equal
122 5 0 3 0	JMNXB	RS	Jump Mask Not Mixed Bits
122 5 0 5 1	JMCE	RS	Jump Mask Current Element
122 5 0 5 2	JNFS	RS	Jump Not Full Strip
122 5 0 5 3	JLRZ	RS	Jump Length Register Zero
122 5 0 5 4	JCL	RS	Jump Condition Less Than
122 5 0 6 4	JCLE	RS	Jump Condition Less Than or Equal
122 5 1 1 0	JMNAB	RS	Jump Mask Not All Bits
122 5 1 2 0	JMNNB	RS	Jump Mask Not No Bits
122 5 1 2 4	JCNE	RS	Jump Condition Not Equal
122 5 1 3 0	JMXB	RS	Jump Mask Mixed Bits
122 5 1 5 1	JMNCE	RS	Jump Mask Not Current Element
122 5 1 5 2	JFS	RS	Jump Full Strip
122 5 1 5 3	JLRNZ	RS	Jump Length Register Non Zero
122 5 1 5 4	JCGE	RS	Jump Condition Greater Than or Equal
122 5 1 6 4	JCG	RS	Jump Condition Greater Than
123 1	JVL	RS	Jump To Vector Loop
123 2	BEL	RS	Begin Element Loop
123 3	JEL	RS	Jump To Element Loop
124 0	JXS	RS	Jump To External Segment
124 0	JXSI	RS	Jump To External Segment, Indirect
125 0	LAJ	RS	Load Address and Jump
125 0	LANI	RS	Load Address of Next Instruction
130 0	L	RS	Load
130 1	DL	RS	Double Load
130 2	LS	RS	Load Stride
131 5	BLVL	RS	Build Long Vector Loop
131 6	BSVL	RS	Build Short Vector Loop
132 1	LLCR	RS	Load Loop Control Registers
134 0	S	RS	Store
134 1	DS	RS	Double Store
134 2	SS	RS	Store Stride
136 1	SLCR	RS	Store Loop Control Registers
137 0	TC	RS	Test and Clear
137 4	TS	RS	Test and Set
141 0	CR	RR	Compare
141 1	DCR	RR	Double Compare Register
142 0	AR	RR	Add Register
142 1	DAR	RR	Double Add Register
142 2	FAR	RR	Floating Add Register
142 3	DFAR	RR	Double Floating Add Register
143 0	ANR	RR	Add Negative Register

Function Code	Mnemonic	Format	Instruction
143 1	DANR	RR	Double Add Negative Register
143 2	FANR	RR	Floating Add Negative Register
143 3	DFANR	RR	Double Floating Add Negative Register
144 0	MSIR	RR	Multiply Single Integer Register
144 1	MIR	RR	Multiply Integer Register
144 2	FMR	RR	Floating Multiply Register
144 3	DFMR	RR	Double Floating Multiply Register
145 0	DSIR	RR	Divide Single Integer Register
145 1	DIR	RR	Divide Integer Register
145 2	FDR	RR	Floating Divide Register
145 3	DFDR	RR	Double Floating Divide Register
146 0	ANDR	RR	And Register
146 1	DANDR	RR	Double And Register
146 2	ORR	RR	Or Register
146 3	DORR	RR	Double Or Register
146 4	XORR	RR	Exclusive Or Register
146 5	DXORR	RR	Double Exclusive Or Register
147 0	SSLR	RR	Single Shift Logical Register
147 1	DSLR	RR	Double Shift Logical Register
147 2	SSAR	RR	Single Shift Algebraic Register
147 3	DSAR	RR	Double Shift Algebraic Register
147 4	LSSLR	RR	Left Single Shift Logical Register
147 5	LDSLR	RR	Left Double Shift Logical Register
147 6	LSSCR	RR	Left Single Shift Circular Register
147 7	LDSCR	RR	Left Double Shift Circular Register
150 0 1	CIDIR	RR	Convert Integer To Double Integer
150 0 2	CIFR	RR	Convert Integer To Floating
150 0 3	CIDFR	RR	Convert Integer To Double Floating
150 1 0	CDIIR	RR	Convert Double Integer To Integer
150 1 2	CDIFR	RR	Convert Double Integer To Floating
150 1 3	CDIDFR	RR	Convert Double Integer To Double Floating
150 2 0	CFIR	RR	Convert Floating To Integer
150 2 1	CFDIR	RR	Convert Floating To Double Integer
150 2 3	CFDFR	RR	Convert Floating To Double Floating
150 3 0	CDFIR	RR	Convert Double Floating To Integer
150 3 1	CDFDIR	RR	Convert Double Floating To Double Integer
150 3 2	CDFFR	RR	Convert Double Floating To Floating
151 0	EMR	RR	Magnitude Register
151 1	DEMR	RR	Double Magnitude Register
152 0	ESCR	RR	Sign Count Register
152 1	DESCR	RR	Double Sign Count Register
154 0	MS	RR	Move Scalar
154 1	DMS	RR	Double Move Scalar
155 0	MNS	RR	Move Negative Scalar
155 1	DMNS	RR	Double Move Negative Scalar
160 0	ALLS	RS	Advance Local Storage Stack
160 4	RLLS	RS	Retract Local Storage Stack
161 5	BVLR	RR	Build Long Vector Loop Register
161 6	DBVLR	RR	Build Short Vector Loop Register
163 1	CELP	RS	Set Current Element Loop

Function Code	Mnemonic	Format	Instruction
163 3	CVELP	RS	Set Current Vector and Element Loop Pointers
167 0	GI	RS	Generate Interrupt
172 0	LGM	RR	Load G Multiple
172 1	DLGM	RR	Double Load G Multiple
173 0	DGR	RS	Diagnose Read
176 0	SGM	RR	Store G Multiple
176 1	DSGM	RR	Double Store G Multiple
177 0	DGW	RS	Diagnose Write

Appendix B. Storage Configurations and Address Interleave

This appendix describes storage and address interleave configurations for the scientific storage unit.

B.1. Storage Configuration

The scientific storage unit is available in increments of 4,194,304 (4 million) words only. It is divided into eight banks of 524,288 (524K) words each. The banks are divided into two main storage partitions (MSP 0 and MSP 1) with four banks in each MSP. There are two interleave options; a four-bank interleave or an eight-bank interleave. MSP 0 consists of a system status register, initial program load (IPL) circuitry, auto-recovery control, dayclock, and four storage banks (0,1,2, and 3). MSP 1 consists of control circuitry and four storage banks (4,5,6 and 7). Figure B-1 shows a 4-million word storage configuration.

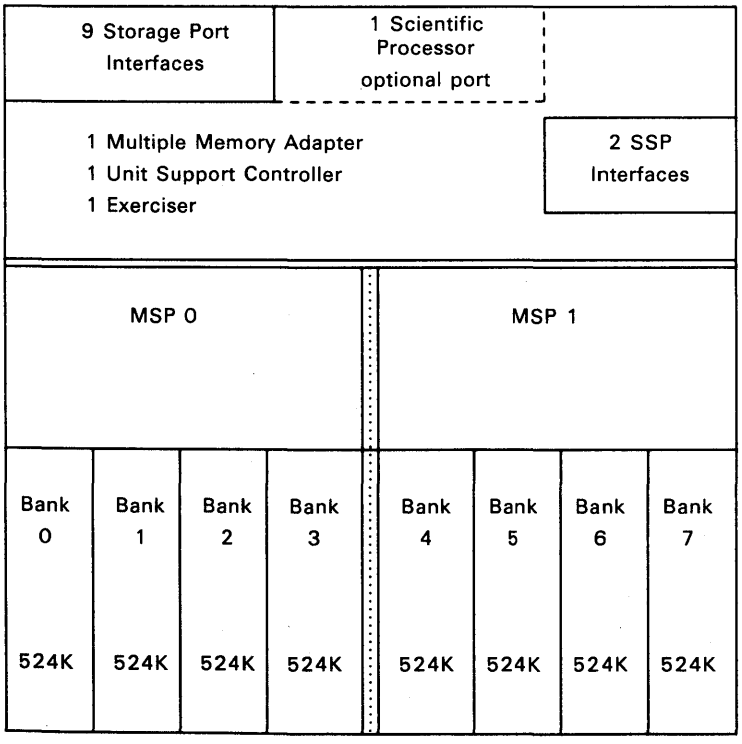
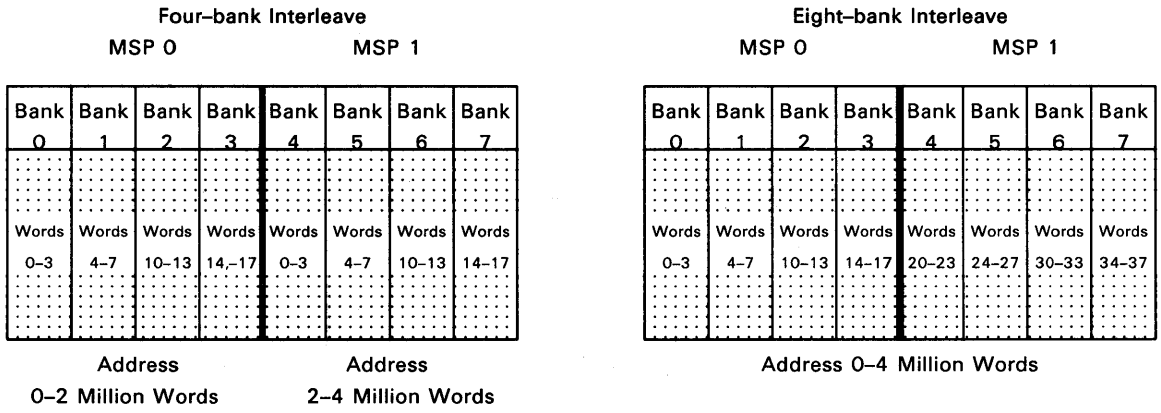


Figure B-1. Four Million Word Storage Configuration

B.2. Address Interleave

Interleave of addresses is done by MSP banks, such as banks 0 through 3, or banks 4 through 7. There is no address interleave between scientific storage units. The two methods available for address interleave are four-bank interleave (minimum interleave) or the eight-bank interleave (maximum interleave). Maximum interleave is recommended for scientific processor operations because it provides maximum performance by separating the bank request sequence to once every 32 words when using sequential addressing. The available interleave options are shown in Figure B-2.



NOTES: 1. *Minimum interleave is 4 banks.*
2. *Maximum interleave is 8 banks.*

Figure B-2. Address Interleave Configurations

B.3. Scientific Storage Address Range

The scientific storage unit is always configured with a full complement of storage array cards (that is 4 million words) and the address range is always contiguous. This is true whether the scientific storage is in minimum or maximum interleave (four-bank or eight-bank interleave). This address range includes 4 million words unless one of the MSPs is down because of internal hardware problems. In this case the remaining MSP must be in minimum interleave (four-bank interleave). With one MSP down (MSP 0 or 1) the scientific storage will have only 2 million words available for the system. If MSP 0 is down the 2-million to 4-million word range is available and if MSP 1 is down it is the 0- to 2-million word range. These are unit address ranges, the system address ranges depend on the scientific storage unit number (scientific storage 0, 1, 2, or 3).

B.4. System Notation of Storage Units and MSPs

The MSP designation on a system basis depends on which scientific storage the MSP is in. The individual unit MSP 0 identification is always an even system MSP number and the unit MSP 1 identification is always an odd system MSP number. It is possible to configure a system with both scientific storage units and MSUs, however, the system MSP numbers are

still determined by what the storage unit system number is. (Refer to Table B-1 for specific information.)

B.5. Storage Unit Maintenance

An individual scientific storage unit can only be configured to one system application at a time. It is not possible to assign (partition) just a part of a scientific storage to a system application. It is possible however, to assign an MSP (i.e. four banks) within a scientific storage unit to a unit maintenance mode. This is done by downing a specific MSP. The MSP when downed is not accessible by any requesters in the application; it is in an online-diagnose state. The storage exerciser may be run in this MSP, and it is possible to perform other maintenance activities such as replacing cards. The remaining MSP (that is not in the online-diagnose state) can be accessed by requesters in the system application.

All systems applications, however require to have accessible an individual unit MSP 0 (that is, storage banks 0, 1, 2, and 3 in a scientific storage or banks 0 and 1 in an MSU). This MSP 0 may be accessible in any storage unit in the application and in either a scientific storage or an MSU. Without an accessible MSP 0 (System MSP number 0, 1, 2, or 3) a system application cannot run. The reason for this is that IPL, dayclock, and auto-recovery circuitry are resident in an individual MSP 0 and a system application cannot run if none are configured in the application.

Table B-1. System and Unit MSP Notation

MSU/ Scientific Storage Number	Unit MSP Number	System MSP Number
MSU 0 or Scientific Storage 0	MSP 0 MSP 1	MSP 0 MSP1
MSU 1 or Scientific Storage 1	MSP 0 MSP 1	MSP 2 MSP 3
MSU 2 or Scientific Storage 2	MSP 0 MSP 1	MSP 4 MSP 5
MSU 3 or Scientific Storage 3	MSP 0 MSP 1	MSP 6 MSP 7

B.6. Scientific Storage Configuration Requirements

One to four scientific storages may be configured in an 1100/90 system. When only one scientific storage is configured it can be located anywhere in the configuration. For configurations with more than one scientific storage, they must be configured such that their address ranges are contiguous. They can be positioned in the center or at the upper or lower end of the system address range.

The 1100 Dynamic Allocator cannot handle address ranges of the two types of storage if the storage type address ranges are intermixed. The allocator makes a decision on each job as to which storage (such as system addresses) it is to be assigned. All jobs for the scientific processor must be assigned to scientific storage address ranges. Other jobs can be assigned to either scientific storage or MSU address ranges.

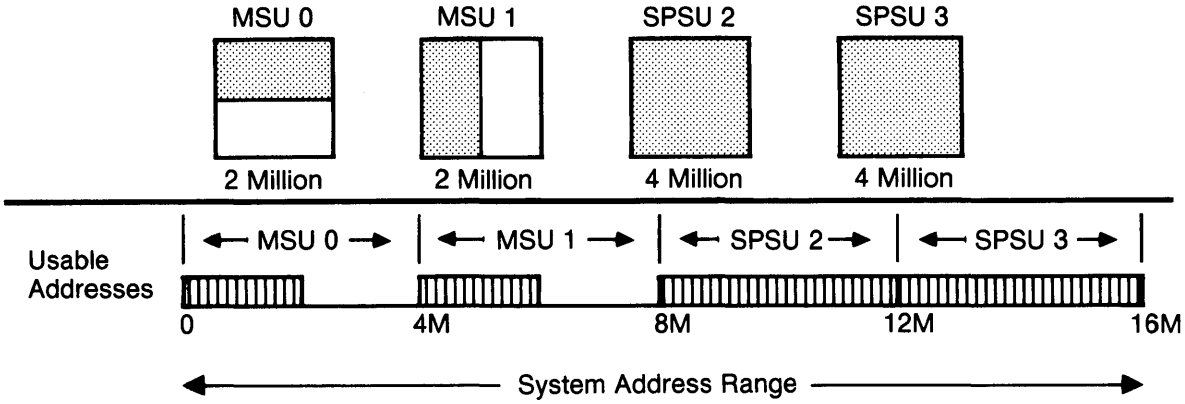
B.7. System Addressing Scientific Storage Interleave

Storage interleave on the 1100/90 exists only within the scientific storage units and MSUs. There is not storage interleave between storage units. Two types of storage interleave exist on the 1100/90 system; first is minimum interleave which is within MSPs and the second is maximum interleave which is across MSPs within a given storage unit. Each storage unit stands alone with its 4-million word address range. Whether scientific storages or MSUs, each storage has a 4-million word address range.

The 1100/90 with scientific processors can be configured with all scientific storage units or a combination of scientific storage units and MSUs. The interleave selection for each of the storage units is independent of the interleave selection for the other storage units. This is true regardless of whether the other units are scientific storages or MSUs. This means that some units may be in maximum interleave and some in minimum interleave. Maximum interleave is recommended for maximum performance particularly for multiprocessor systems.

B.8. Storage Related Address Ranges

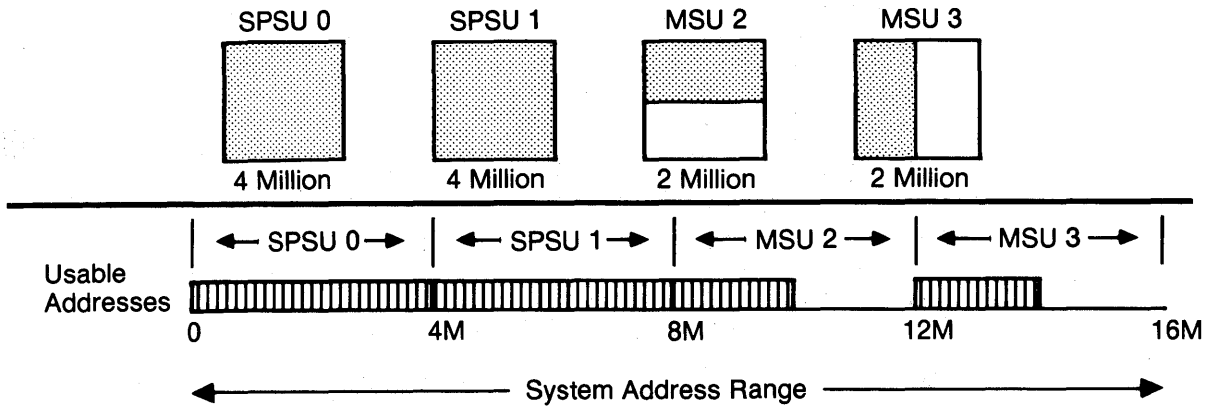
Each storage unit relates to a specific address range (i.e. storage unit 0, addresses 0 through 4 million; storage unit 1, addresses 4 million through 8 million). Only when the storage units are configured with the maximum 4 million words per unit is the system address range contiguous from 0 through 16 million words. With scientific processors configured, some of these units will be scientific processor storage units (SPSUs) and some may be main storage units (MSUs). Figures B-3 and B-4 illustrate configurations with both SPSUs and MSUs.



NOTE: SPSUs at upper range of addresses

Figure B-3. Example 1 of Storage Unit Configuration and System Addresses

NOTE: Scientific storages at upper range of addresses



NOTE: SPSUs at lower end of addresses

Figure B-4. Example 2 of Storage Unit Configuration and System Addresses

NOTE: Scientific storages at lower end of addresses

B.9. Partitioning Scientific Storage Units and Main Storage Units

Storage units are partitioned to system applications on a cabinet basis. An entire cabinet or number of cabinets are assigned to an application. There is no restriction as to which storage unit(s) (scientific storage or MSU) is assigned to a particular application with IPs and IOPs however applications with scientific processors must have scientific storages included or they cannot operate.

B.10. Storage Partitioning Implications on System Reboot

Storage partitioning can result in a system reboot, but only in one situation. That situation is for the downing of a storage unit which has EXEC code resident in its memory. If a storage unit is downed and EXEC code is not resident in its memory, then no reboot is required. Reboots are never required when storage units are brought into an application which is already running. This is because the additional storage unit has absolutely no effect on the address range on any storage unit already in the application.

B.11. Module Select Register (MSR) Settings for Scientific Storage Units

The setting of the MSR value follows strict rules which are shown in Table B-2. The MSR value allowable depends on which scientific storage unit is being used. The scientific storage unit is only available with 4 million words, therefore, the only possible change in MSR settings is when MSP 1 is downed for maintenance.

These MSR values are selected during the SSP /OPER program execution. For more specific information on how and when to select MSR values, refer to *1100/90 System, System Support Processor Operator Reference,* UP-9123.6.

Table B-2. MSR Values for Scientific Storage

SPSU* Number	Total Storage in SPSU*	Storage in MSP 0	Storage in MSP 1	Valid Range of Values in Octal for MSR Number			
				UNIT MSP 0		UNIT MSP 1	
0	4M	2M	**	00-07	System	—	System
0	4M	2M	2M	00-07	MSP 0	10-17	MSP 1
1	4M	2M	**	20-27	System	—	System
1	4M	2M	2M	20-27	MSP 2	30-37	MSP 3
2	4M	2M	**	40-47	System	—	System
2	4M	2M	2M	40-47	MSP 4	50-57	MSP 5
3	4M	2M	**	60-67	System	—	System
3	4M	2M	2M	60-67	MSP 6	70-77	MSP 7

**SPSU means scientific processor storage unit.*

***This condition only occurs when MSP-1 is downed for maintenance.*

Glossary

The glossary defines the key terms used in this manual.

A

accelerate
: Switching activities into the *scientific processor.*

activity
: The unit of work that is scheduled for the *scientific processor.* An activity consists of scalar and vector operations that execute in their respective processors.

activity switching
: The process of going from one *scientific processor* state to another.

add pipeline
: A section in the *vector module* that executes all single-precision and double-precision floating-point and integer instructions except multiply, divide, product reduction, vector move, and vector load instructions.

address generation
: A section in the *scalar module* that generates instructions to load either data or instructions from main storage.

B

buffer
: A storage area used to hold data temporarily as it is transmitted from one device to another. Buffers compensate for differences in the rate that data flows from one device to another.

C

chaining
: Overlapping the execution of consecutive instructions.

control block
: The control block has registers that contain the state of the *scientific processor* at certain stages of *activity* execution.

D

decelerate
: Switching activities out of the *scientific processor.*

E

Executive
: A Sperry supplied program that controls the 1100/90 system's operating environment. It is part of the operating system.

G

G–registers
: Registers in the *scalar module* used as accumulators for single–precision and double–precision scalar data operands.

H

hardware status register
: Four registers in the *scalar module* that are part of the control block. These registers hold hardware status information and external interrupt indicators.

I

instruction buffer
: An internal storage in the *scalar module* containing instructions loaded from the *scientific processor storage.*

instruction control
: The logic of the *instruction flow control* section that reads, decodes, and dispatches instructions.

instruction flow control
: A section in the *scalar module* that contains instruction buffer and control mechanisms.

instruction processor
: The main processing unit of the 1100/90 system.

J

jump history file
: A file in the *scientific processor* that contains the virtual address of the 32 most recent jumps internally executed by an *activity.*

L

local storage
: A storage area in the *scientific processor* that is used for frequently used scalar variables and constants.

loop control
: The loop control section in the *scalar module* broadcasts the current loop count and element count data to the various vector sections so that current loop and element positions or ending can be determined.

loop control registers
: Registers located in the *scalar module* that are used to hold parameters that determine iteration and indexing of program loops.

M

mailbox
: An eight–word register in the *scientific processor storage* that contains pointers to the information necessary to execute an *activity* located in the *scientific processor* control block.

mask processor
: A section in the *scalar module* allows a single source mask to provide multiple references at several different bit positions, one for each active destination, under one mask controller.

move pipeline
: A section in the *vector module* that executes the move, compress, and distribute instructions. It also participates in single to double and double to single–precision conversions.

multiply pipeline
A section in the *vector module* that executes the multiply, divide, and product reduction instructions.

P

pipeline
A functional unit that contains separate hardware stages to perform subfunctions. The stages operate independently so that the pipeline can process data in parallel.

Q

quantum timer
A timer that limits the execution time of an *activity* on the *scientific processor.*

R

RR format
The register–to–register format used for scalar instructions that do not specify a storage location.

RS format
The register–to–storage format used for most scalar operations that specify a storage location in either local storage or *scientific processor storage.*

S

scalar module
The processor module that performs scalar operations and has the overall control function in the *scientific processor.*

scalar processor
A section in the *scalar module* that executes most scalar instructions, performs floating–point characteristic manipulation, and performs integer and floating–point multiply operations.

scalar vector control
A section in the *vector module* that provides interface control for *RR format* and *VV format* instructions that move G–register operand and vector operand data between the *scalar module* and the *vector module.*

scientific processor
A processor that attaches to the system for performing scientific problems.

scientific processor system
A group of components characterized by a very large storage, high–speed arithmetic operations, and a large variety of floating–point arithmetic instructions.

scientific processor storage
A free–standing unit that provides main storage for both the 1100/90 instruction processor and the *scientific processor* in a *scientific processor system.*

state registers
Registers in the *scientific processor* that contain program visible states relating to internal interrupts and condition codes.

store buffer
A section in the *scalar module* that provides a *buffer* for data coming from the *vector module* and going to the *scientific processor storage.*

stride
A constant skip or increment through storage.

U

universal processor interface (UPI)
The communications interface between the *scientific processor* and the instruction processor.

V

VV format
: The vector-to-vector format used to specify vector operations.

vector control
: A section in the *vector module* that receives the vector control word, manages vector file time slots, and forms and selects vector file addresses.

vector files
: Sixteen files in the *vector module* that hold the elements of a single vector array. Each file has space for 64, 36-bit words.

vector load
: A section in the *vector module* that transfers data from local storage or the *scientific processor storage* into the vector files.

vector module
: See *vector processor.*

vector processor
: The processor that executes the vector portions of scientific applications. Vector scientific applications are those large portions of code that can execute in the vector processor's pipelines.

vector store
: A section in the *vector module* that controls moving data from the vector files to the *store buffer* in the *scalar module.*

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

SPERRY

USER COMMENTS

We will use your comments to improve subsequent editions.

NOTE: Please do not use this form as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update Level)*

Comments:

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage is necessary if mailed in the U.S.A.)
Thank you for your cooperation

CUT

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY CORPORATION

ATTN: Documentation Quality Control Group
C/O SYSTEM PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19422-9990

FOLD