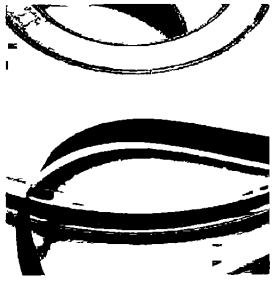
Prime Computer, Inc.

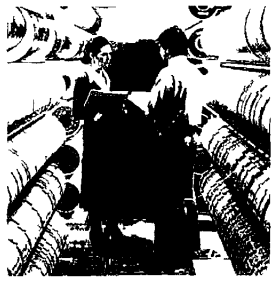
DOC3060-192P

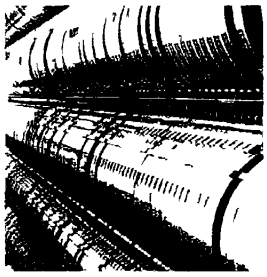
System Architecture Reference Guide

Revision 19.2

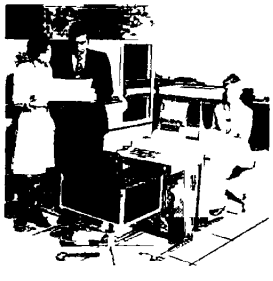











System Architecture Reference Guide

DOC3060-192

Third Edition

by

Martha August, Alice Landy, and Marilyn Hammond

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.2 (Rev. 19.2).

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

COPYRIGHT INFORMATION

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000 X4837

Customers Outside U.S.

Contact your local Prime
subsidiary or distributor.

PRINTING HISTORY

System Architecture Reference Guide

Edition	Date	Number	Software Release
First Edition	April 1979	IDR3060	n/a
Second Edition	April 1981	PDR3060-182	18.2
Third Edition	July 1983	DOC3060-192	19.2

SUGGESTION BOX

All correspondence on suggested changes to this document should be directed to:

Alice Landy
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

Contents

About This Book

Prime's 50 Series family is a sophisticated group of totally compatible supermini computers. Its members are the Prime:

- 2250
- 250-II
- 550-II
- 750
- 850
- 9950

The 50 Series systems embody an advanced 32-bit architecture that grants the user the ability to perform complex tasks efficiently and quickly. This document describes the 50 Series architecture from a functional point of view.

NOTES TO THE READER

Several groups of people will find this document useful: engineers, programmers, designers, and technicians. To read this book, you should have a basic understanding of computers, but not necessarily of Prime computers. Prime stresses a high degree of compatibility across its product line; therefore, you can apply much of the information contained in this book to other Prime machines, as well as to the 50 Series machines.

ORGANIZATION OF THIS GUIDE

Because this guide stresses the functional aspects of the 50 Series processors, the topics are organized according to function. Chapter 1 presents a general overview. Chapters 2 through 12 each describe one aspect of the system, beginning with memory configuration and addressing and ending with the I/O system. Each chapter builds on the information contained in the previous one. Chapters 1 through 12 may be summarized as follows:

- Chapter 1: Overview of the 50 Series systems
- Chapter 2: Configuration of the 50 Series physical and virtual memory
- Chapter 3: Virtual addressing, modes and formats, and address traps
- Chapter 4: Memory management
- Chapter 5: Control data structures and restricted instructions
- Chapter 6: Datatypes supported on the 50 Series systems
- Chapter 7: Branch instructions and the stack
- Chapter 8: Procedure calls, the stack, and argument transfers
- Chapter 9: Single-stream process exchange
- Chapter 10: Dual-stream (850) process exchange
- Chapter 11: Interrupts, faults, checks, and traps
- Chapter 12: The I/O system (DMA, DMC, DMT, and DMQ)

Throughout these chapters are lists of Prime assembly language instructions that pertain to the topics under discussion. These lists briefly define the instructions' actions and show how they relate to the topics. In addition to these lists, Chapters 13 and 14 contain detailed information about each instruction -- name, format, mnemonic, and required operands -- and a complete description of each of the instruction's actions. These chapters are summarized as follows:

- Chapter 13: Instructions executable in S, R, and V mode
- Chapter 14: Instructions executable in I mode

Appendix A discusses system power-up and the initialization of registers.

1

System Overview

The CPUs of all 50 Series systems share a common architecture and one operating system. This commonality is what makes the 50 Series a line of completely upward- and downward-compatible systems. The implementation of the common architecture, however, is slightly different for each member, allowing the 50 Series systems to address a wide variety of user needs as well as remain compatible. The first part of this chapter explores the single-stream CPU implemented on the 2250, 250-II, 550-II, and 750. The second part discusses the dual-stream 850 CPU. The third part discusses Prime's newest CPU, the 9950.

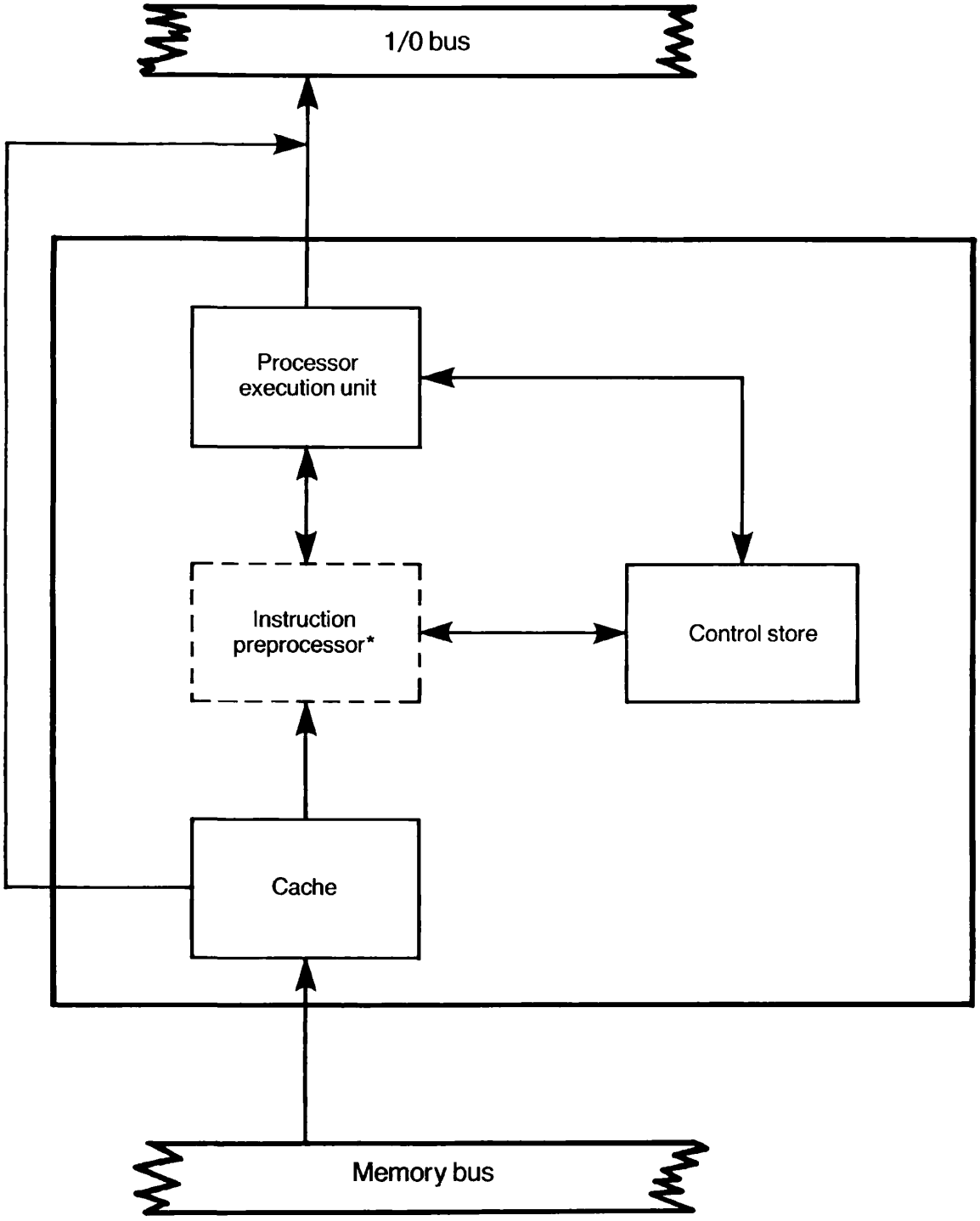
SINGLE-STREAM ARCHITECTURE

The CPU can be divided into four major units. The first three of these are implemented on all single-stream members of the 50 Series family:

- Cache memory
- Control store
- Processor execution unit

The fourth, the instruction preprocessor unit, is a feature of the 750 and 850 systems. It serves as a speedup mechanism to process instructions at a greater speed.

Figure 1-1 diagrams this architecture.



* = 750 and 850 only

Block Diagram of Single-processor Architecture
Figure 1-1

Cache and STLB

The 50 Series uses a virtually addressed, write-through cache. Each of the cache entries contains the contents of and additional information about two bytes (2250, 250-II, and 550-II) or four bytes (750, 850, and 9950) of recently accessed physical memory. If the contents of a specified location can be found in the cache, the system saves a great deal of time: it takes only 80 nanoseconds to access a cache entry, a vast improvement over the approximately 600 nanoseconds needed to access physical memory. The time saved can be spent performing other operations rather than waiting for a memory reference to complete.

To speed up the virtual to physical address translation, the STLB (Segmentation Table Lookaside Buffer) contains the results of the last 64 translations (128 translations on the 9950). Since programs tend to reference the same set of locations during their execution, the system can perform a translation once, store the result in the STLB, and then have it for reference the next time the user specifies the same location. Since the STLB has a much faster access time than physical memory does, referencing it saves translation time as well as access time.

See Chapter 4, Memory Management, for more information about cache, STLB, and address translation.

The Control Store Unit

To speed up execution, the 50 Series systems implement many functions, such as procedure calls, in hardware and firmware. (Procedure calls are explained in Chapter 8.) The firmware that governs instruction execution is contained in the control store ROM. Each 50 Series system can support up to 64 Kbytes of firmware address space. The exception is the 9950, which uses a loadable control store of 50 Kbytes of RAM.

The Processor Execution Unit

This unit performs the computation required during instruction execution. Elements of the processor execution unit include:

- Integer arithmetic logic unit (ALU)
- Decimal ALU
- Floating point unit
- Register file
- Program counter

Figure 1-2 shows an expanded block diagram of the processor execution unit.

ALUs: The integer arithmetic logic unit (ALU) performs the desired operation on the user's two's complement data. In a similar fashion, the decimal ALU and the floating-point unit handle decimal and floating-point operations, respectively. These units can perform tests and checks as well as arithmetic operations.

Register File: The register file contains four sets of registers. Each set contains 32 32-bit registers. Two of these are user register sets that contain information about a process and about the system as the process sees it. These user register sets contain information about the general registers a process can use, addresses of fault handlers, contents of system registers, and other useful information.

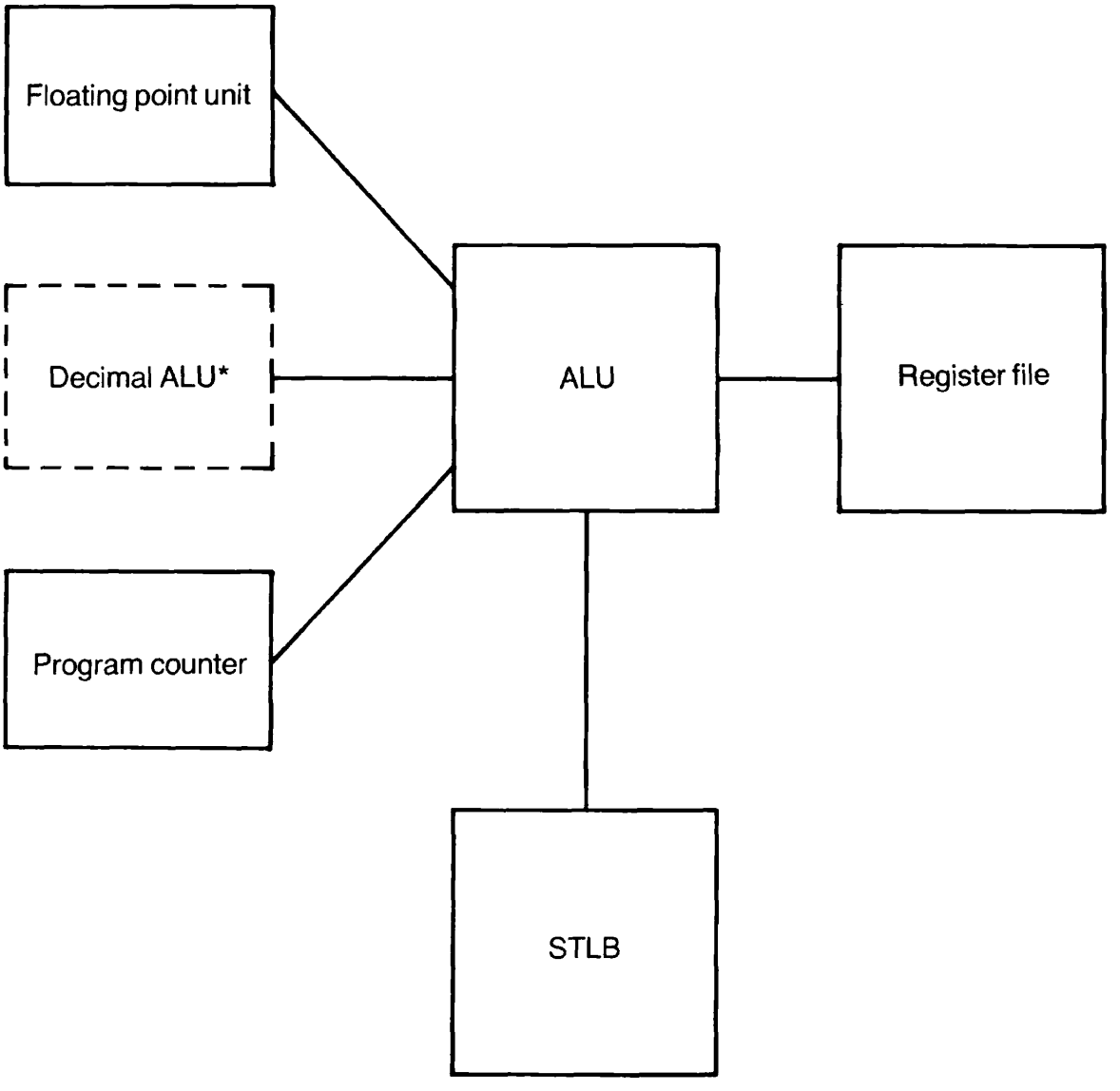
One of the remaining register sets contains microcode scratch and system status registers. The fourth set contains direct memory access (DMA) channels to speed I/O operations. (See Chapter 12.)

The 9950 has eight register sets: four sets of user registers, three sets of mircrocode scratch registers, and one set of direct memory access registers.

Program Counter: The program counter contains the address of the next instruction to be executed.

The Instruction Preprocessor Unit

The 750 has a special instruction preprocessor unit, designed to speed up execution by processing information about instructions before execution. While the processor execution unit is performing an add or similar operation for instruction n, the instruction preprocessor is working on the next two instructions. It is decoding instruction n+1, calculating its address, and determining what registers, if any, are to be accessed. It is also fetching instruction n+2 from the cache so that it can be decoded when instruction n+1 begins to execute. This means that, in most cases, when the processor execution unit finishes one operation, the instruction preprocessor unit has already done the calculations necessary to allow the execution unit to perform the next instruction without delay.



***= 550-II, 750, 850, and 9950 only.**

Processor Execution Unit
Figure 1-2

DUAL-STREAM IMPLEMENTATION

The 850 system implements a dual-stream version of the common 50 Series architecture. The system's dual-stream nature enables it to provide 60-80% more service than the 750. Figure 1-3 shows a block diagram of the 850 dual-stream architecture.

Instruction Stream Units

The 850 contains two instruction stream units (ISUs), each of which is similar in capabilities to a 750 CPU. Each ISU executes an independent stream of instructions simultaneously, synchronized by a stream synchronization unit (SSU). (See below.) Each ISU is responsible for:

- Full instruction decode.
- Effective address calculation.
- Instruction execution.
- Calculating data for the anticipated next instruction.

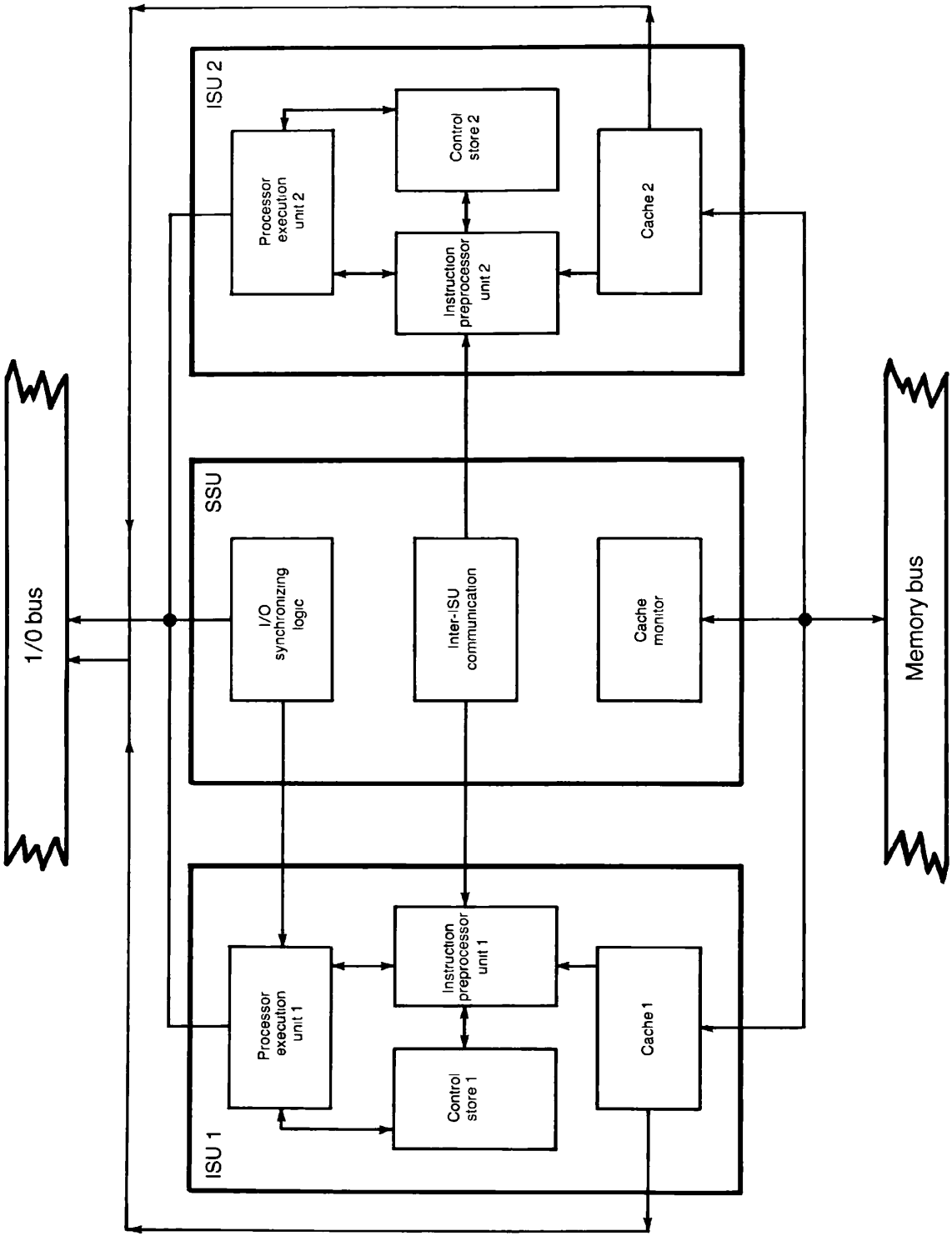
The four blocks shown in each ISU contain the same elements and perform the same functions as those described in the first part of this chapter.

Note that the two ISUs share one copy of the operating system. PRIMOS is reentrant and can run on either ISU (as can any user program), so duplicate copies are not needed. System actions are also simplified, since there is no need to check for or handle discrepancies caused by different versions of the operating system.

Stream Synchronization Unit

The primary task of the SSU is to prevent improper information from being loaded into the cache of either ISU. It does this by maintaining a list of the contents of both caches. When data is written into either cache, the SSU detects it and invalidates the contents of the appropriate entry in its list of cache contents. This means that the SSU always knows which cache locations contain current information and which do not.

When a cache location in one of the ISUs contains information that is out-of-date, the SSU notifies that ISU of the discrepancy. That ISU invalidates the stale entry, thus forcing a memory read to the current information the next time that location is referenced.



Dual-stream Architecture
Figure 1-3

In addition to synchronizing cache references, the SSU also coordinates references to memory and system handlers. The two ISUs share one main memory, one operating system, and one copy of several system handlers. To ensure that these resources are used effectively and efficiently, the SSU contains four locks. The process exchange lock aids the process exchange mechanism (see Chapter 10) to transfer control smoothly between processes on both ISUs. The queue lock controls situations in which simultaneously executing queue instructions (one on each ISU) are vying for access to a single queue. It ensures that both instructions get access, but that neither one interrupts or interferes with the other. The check lock allows only one ISU to signal a check at a time, thus guaranteeing that the single set of check handlers services all checks. The fourth lock, the mutual exclusion lock, can be used by software to prevent both ISUs from trying to access a particular procedure or piece of data at the same time.

Diagnostic operations and communications between ISUs are also handled through the SSU. The former feature aids in system monitoring and testing; the latter enhances the 850's ability to execute independent instruction streams without high system overhead.

SPECIAL FEATURES OF THE 9950

Although the 9950 follows the general architecture of the 50 Series, as shown in the previous discussions, it contains several features designed for outstanding performance. These include:

- ECL design. The 9950 uses emitter coupled logic (ECL) for swift execution of instructions. Memory parts using ECL are about 50-60% faster than those made of TTL or NMOS; all other ECL parts are twice as fast, on the average, as their Schottky counterparts.

- Dedicated backplane. To minimize delay when instructions flow from one system unit to another, each of the five PC boards that make up the 9950 processor is assigned a specific slot in the CPU chassis.

- Pipeline. The 9950 uses a pipeline technique for executing instructions in parallel, thus speeding up instruction execution considerably. The pipeline is explained in the next section.

- Branch cache. The 9950 uses a memory called the branch cache to record and predict the target address for jump and branch instructions. The branch cache contains 256 entries.

Because the 9950 executes instructions in parallel, it might begin to execute instructions down an incorrect path, following a branch, before it had determined the correct branch address. If this occurs, the processor must flush the pipeline of all instructions from the wrong branch path, and then must begin execution down the correct branch path. This sequence of steps causes a delay.

To minimize the chance of such an occurrence, the 9950 branch cache contains information about the branches that have previously occurred in the program. The processor uses this information to determine which branch was most recently taken for each conditional instruction. The 9950 then assumes that the same branch will be taken this time. If the prediction is wrong, the processor adds a new entry in the cache, specifying the correct branch for future use.

- Environmental sensors. These are explained in the final section of this chapter.

The 9950 Pipeline

The execution of each 9950 instruction is divided into ten stages, as shown in Table 1-1. Each stage takes 40 nanoseconds to complete. This is called the beat rate of the system.

The 9950 executes instructions in parallel. This means that the processor does not have to complete the entire ten-stage sequence for one instruction before it can begin executing the next. Rather, instructions are processed somewhat like cars in a factory assembly line. The cars travel past a number of specialized stations. At each station a specific operation takes place. Then the car moves on. After a certain length of time the next car arrives at the same station where the same operation occurs.

The 9950 ten-stage pipeline processes instructions in a similar fashion. After every other 40-nanosecond beat, a new instruction arrives at a station, and that station's operation is performed on it.

Using the pipeline in this fashion, the 9950 executes Stages 1 and 2 of the first instruction. When it begins on Stage 3 of the first instruction, it can also begin Stage 1 of the second instruction. Likewise, when it begins Stage 3 of the second instruction, it can also begin Stage 1 of the third, and so on. This means that the pipeline can begin a new instruction every other beat.

The rate of instruction flow through the pipeline is determined by the processor's use of system elements at each stage. As shown in Table 1-1, Stages 2 and 7 both use the cache, and Stages 7 and 10 both use the register file. When two instructions in the pipeline request the same element at the same time, a conflict occurs. Starting a new instruction every other beat minimizes this type of conflict.

When there are no conflicts in the pipeline, simple instructions complete execution every 80 nanoseconds. Some instructions, however, require more than 80 nanoseconds to complete execution. When this occurs, the pipeline holds up operations on the subsequent instructions until it has completed the extra operation for the first instruction. During the holdup, the processor still forms control store addresses and fetches microcode words, but it performs no prefetch or effective address calculations.

Flushing the Pipeline

If an instruction stores data into the stream of instructions that follows it, the 9950 pipeline may have to be flushed before further calculations take place. S and R mode store instructions automatically flush the pipeline; therefore, no further actions are required and performance is reduced substantially. V and I mode store instructions, however, do not automatically flush the pipe. Either an E64V (V mode) or an E32I (I mode) instruction will perform the flush.

Prime systems are designed for pure procedure. All translator-generated code avoids storing into the instruction stream.

UPS and Environmental Sensing Support on the 9950

The 9950 has a diagnostic processor system that supports inputs from the UPS (uninterruptable power supply) system and environmental sensors. These allow the 9950 to be brought to an orderly shutdown in the event of an overtemperature or a main AC power loss with messages appearing on the supervisor terminal. In order to conserve power, the diagnostic processor does not accept typed commands during system shutdown or while the UPS is active.

UPS Support: The UPS uses two signals, UPS active and UPS battery low. UPS active means that main AC power has been interrupted. The low battery condition means that 5 or 6 minutes remain before system power is lost.

When the UPS is powering the entire system, including peripherals, and a battery low condition occurs, the diagnostic processor sends a processor check to the CPU (as explained in Chapter 11), and waits for a CPU halt or for up to 5 minutes before powering down the system.

When a UPS active condition occurs and the UPS is powering only the CPU, memory, and diagnostic processor, the diagnostic processor sends a power failure signal to the processor, causing the processor to log the power failure condition and then halt.

Table 1-1
Stages in the 9950 Instruction Execution Pipeline

Stage	Action
1	Send the contents of the lookahead program register to the memory address register.
2	Read the next instruction from the cache.
3	Start decoding the address of the next instruction.
4	Read the contents of the base and index registers.
5	Form the effective address and the control store address.
6	Send the contents of the effective address register to the memory address register and fetch the contents of the next microword.
7	Read the operand from the cache and register file.
8	Execution, phase 1 (ALU).
9	Execution, phase 2. (Transfer results to RS.)
10	Store the results of the operation.

When the UPS condition changes from active to inactive, it implies that AC power has returned. The diagnostic processor initiates a warm start to the CPU provided that the operator keyed an AWARMON command at the supervisor terminal before main AC power loss. Otherwise, the diagnostic processor initiates a master clear and enters control panel mode.

Environmental Sensing Support: There are three environmental sensors: a cabinet overtemperature sensor, a processor board overtemperature sensor, and an air flow sensor that detects failures in the cabinet blowers. The cabinet temperature and airflow sensors are warning indicators; the processor board sensor is a critical indicator.

When the cabinet temperature is too high or the airflow sensor detects a failure in the cabinet blowers, the diagnostic processor initiates an orderly system shutdown by sending the processor an appropriate environmental check code (explained in Chapter 11) that initiates the PRIMOS system shutdown. The diagnostic processor waits for a CPU halted message or for a specified timeout (10 minutes for cabinet overtemperature, 1 minute for air blower failure). If an air blower failure occurs while there is more than 1 minute to timeout, the timeout is set to 1 minute.

When an overtemperature condition is detected on the processor board, the diagnostic processor initiates an immediate system powerdown that includes powering down the processor.

2

Physical and Virtual Memory

The 50 Series processors are virtual memory systems. This means that a very large, protected, virtual address space is available to each user who is logged onto the system. This virtual address space is supported by a much smaller physical address space invisible to the user.

Virtual memory has several advantages. To the user logged onto the system, there appears to be an address space of almost unlimited size, which can support very large applications without using overlays. This address space is protected against unauthorized accesses in hardware. To the system owner, a virtual memory scheme provides the ease of use of a large memory at the cost of a much smaller amount of hardware.

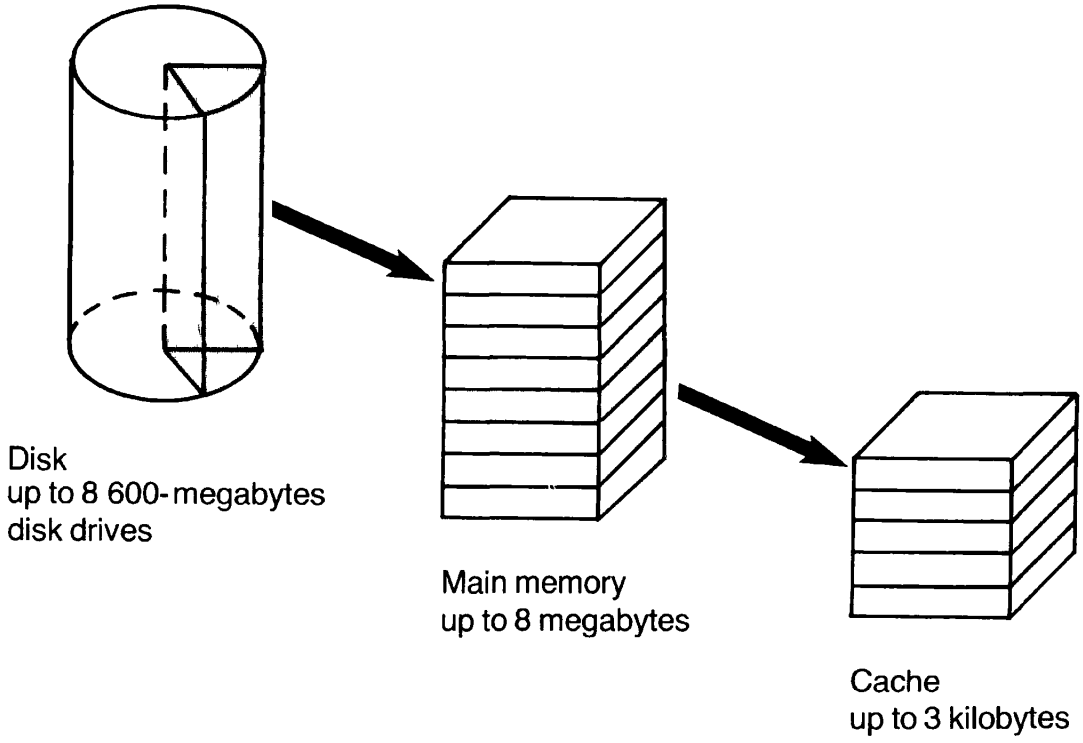
The three key parts to a virtual memory scheme are physical memory, virtual memory, and a manager to control the virtual memory scheme. The manager is the operating system, PRIMOS, and its attendant hardware and firmware support. This chapter describes the characteristics of the 50 Series physical and virtual memory, and shows how PRIMOS coordinates the 50 Series virtual memory scheme. It also describes some of the hardware protection mechanisms implemented in the 50 Series virtual memory.

PHYSICAL MEMORY

Physical memory encompasses all hardware parts of the system used to store large blocks of information. There are three types of physical memory:

- Cache
- Main memory
- Disk

Figure 2-1 shows the relationship between the three elements of physical memory.



Elements of Physical Memory
Figure 2-1

Cache

The cache is a data buffer that stores copies of the information contained in the most frequently referenced memory locations. Its size varies from system to system as shown in Table 2-1. During program execution, this buffer is used to speed up memory references.

Since cache is a form of very high speed memory, it takes only 80 nanoseconds to access data stored there. In contrast, it takes about 600 nanoseconds to access data stored in main memory. This difference in access times makes it very advantageous to access cache whenever possible.

Three factors determine how often the cache contains the correct data (known as the cache hit rate):

- The size of the cache (2-32 Kbytes)
- The information fetch rate (16-64 bits, depending on the system and the amount of memory interleaving)
- Locality of reference (the tendency of a program to execute within a small part of itself at any time)

The 50 Series cache hit rate varies from system to system. See Table 2-1 for details.

Table 2-1
Cache Sizes and Hit Rates

System	Cache Size	Hit Rate
2250	2 Kbytes	85%
250-II	2 Kbytes	85%
550-II	8 Kbytes	90%
750	16 Kbytes	95%
850	32 Kbytes	95%
9950	16 Kbytes	95%

Main Memory

The 50 Series main memory is high speed MOS with error checking and correction built in to correct single bit errors and detect double bit errors. The memory is packaged on boards in units of 512 Kbytes or 1 Mbyte. (The 9950 allows up to 2 Mbytes per board in units of 64K RAM chips. Since the 9950 can contain up to eight memory boards, it can have up to 16 Mbytes of main memory.)

All systems use two-way interleaving. This doubles the amount of data that can be fetched with one operation. Thus, it speeds up memory references and makes more efficient use of the I/O bus.

On the 9950, interleaving takes place within each memory board. On all other systems, interleaving is done between pairs of boards. In this type of interleaving, consecutive physical locations are placed on alternate memory boards; when a reference to memory is made, the system fetches the same location on each board. Systems with an odd number of memory boards use interleaving for all but the odd board.

Main memory is divided into units called pages. Each page is 2 Kbytes in size. The pages subdivide main memory into uniform pieces that PRIMOS can manage conveniently and efficiently. Since all pages are the same size, PRIMOS can reply to all requests for space in the same way, regardless of who or what makes the request. In addition, disk records (called virtual pages) are the same 2 Kbytes in size, so transfers between main memory and disk are simplified. Chapter 4, Memory Management, describes other advantages of pages.

Disk

Disks provide storage for all of virtual memory. Either the system or the user can access any of this information at any time (given the proper access rights). When accessed, a copy of the information is moved from disk to main memory. The Paging section in Chapter 4 describes how the information is moved.

VIRTUAL MEMORY

Virtual memory is divided into units called segments. Each segment can contain up to 128 Kbytes. Segments are virtual units, not physical ones, that aid both the user and the system in organizing their virtual address spaces and the information contained there. For example, the user can organize program code in one segment and program data in a second one. Segments make it possible to allow extra room in a program for variable length data structures, such as arrays whose dimensions can change each time the program runs. They also allow the user to build modular programs, one module to a segment. PRIMOS uses segments in a similar way to organize its own code into modules.

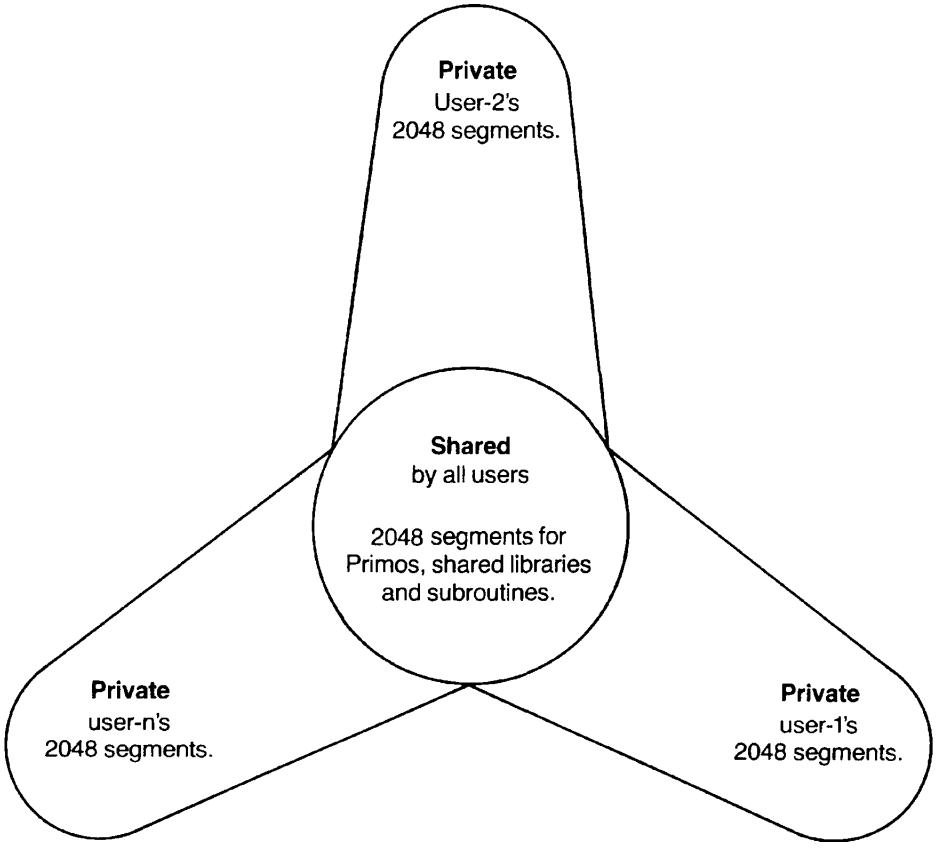
The virtual address space of each user contains 4096 segments. These are subdivided into four groups of 1024 each. The segments are subdivided to make address translation and segment sharing easier. (See Shared and Unshared Segments, below, and Chapter 4, Memory Management.)

Shared and Unshared Segments

In the Prime virtual memory scheme (diagrammed in Figure 2-2), each user address space of 4096 segments is divided into shared and unshared space. The first 2048 segments are shared with all other users. This allows the operating system, shared libraries, and shared subsystems to be seen by all users.

The second 2048 segments are private, containing information unique to each user. This means that if two users reference segment 4000, they are specifying completely different locations.

This arrangement of shared and unshared segments means that there is no possibility of one user's private space conflicting with that of another user. It also means that only one copy of PRIMOS and the shared system software need be maintained, and thus reduces memory use. Moreover, it means that PRIMOS is embedded in the virtual address space of each user and is directly accessible via a normal procedure call. (See Chapter 8, Stacks and Procedure Calls.) No interrupts, special supervisor calls, or system traps are necessary when the user accesses PRIMOS or any utility, library, or subsystem residing in shared space.



50 Series Virtual Memory Space
Figure 2-2

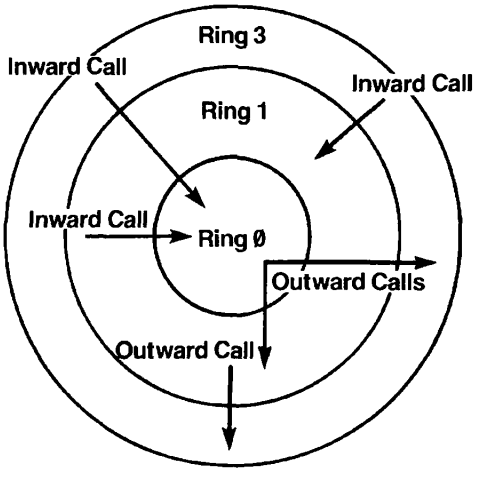
Protection Rings

Designating shared and unshared segments is not the only form of protection available to the 50 Series virtual memory. Three hardware implemented rings provide a simple, unbreakable form of security that checks each memory reference for its right to access the specified part of memory.

The rings represent levels of protection. Ring 0 represents the highest level of protection and grants the greatest number of privileges. The kernel of PRIMOS runs under Ring 0 protection, which means that its segments cannot be accessed by the user except through protected entry points, and that it has read, write, and execute privileges to all segments. PRIMOS can access any information in the system, invoke special routines, and so on.

Users run under Ring 3 protection, which means that they cannot arbitrarily access Ring 0 routines or items contained in the private segments of other users' address spaces. Each segment under Ring 3 protection may have a different combination of read, write, and execute access rights.

Ring 1 provides privileges less powerful than those of Ring 0 but more powerful than those of Ring 3.



Protection Rings
Figure 2-3

Rings provide a simple, effective way to protect critical parts of the system. Without them, a Ring 3 procedure could directly access any Ring 0 procedure, which could potentially corrupt system operation. Screening out such references protects the integrity of the entire system.

See Chapter 4, Memory Management, for information about how rings govern the virtual-to-physical address translation to prevent invalid accesses.

Segmentation Table Lookaside Buffer

Virtual memory has its counterpart of the cache, the STLB. The system uses this buffer with the cache to reduce the time needed to access information. Where a cache entry contains information about a recently accessed physical memory location, an STLB entry contains the information the system needs to find the physical location from the virtual address the user specified. Each entry also specifies the protection attributes associated with the location. Chapter 4 describes more about how the STLB is used.

SUMMARY

This chapter described the configuration of the 50 Series physical and virtual memories. Chapter 3, Addressing, shows how to form a virtual address that references a location within the virtual address space. Chapter 4, Memory Management, shows how the 50 Series systems use the virtual address and the virtual-to-physical address translation process to integrate virtual and physical memory.

3
Addressing

INTRODUCTION

The 50 Series processors support several kinds of addressing: direct addressing, indexed addressing, indirect addressing, and indirect indexed addressing. They also support several modes of addressing, each with its own uses and benefits. This chapter:

- Provides an overview of virtual addressing and of effective address calculation.
- Explains how effective address calculation is done for each type of addressing, and what registers are involved.
- Explains the various modes of addressing.
- Provides summaries of instruction forms for each type of addressing in each mode.

UNITS

The basic units of information are bits, bytes, halfwords, and words. A byte contains eight bits. One halfword contains two bytes; the bits are labelled from 1 (most significant bit) to 16 (least significant bit). A word contains four bytes. The bits are labelled from 1 to 32.

Memory is measured in bytes. The 50 Series physical memory size can be up to 16 Mbytes; the virtual address space contains 512 Mbytes.

COMPONENTS OF A VIRTUAL ADDRESS

A virtual address refers to a unique location in a user's virtual address space. The location is characterized by three elements: a ring number, a segment number, and an offset within that segment. (All offsets are relative to the first location within a segment, and are expressed in units of halfwords.) The format of a virtual address is shown in Figure 3-1.

When an instruction makes a memory reference, it provides information from which the virtual address can be calculated. This is frequently referred to as calculating the effective address. Depending on the type of instruction, the information can be provided in several different formats, and the calculation done in various ways. This section explains the various ways in which the ring number, segment number, and offset can be specified. It also explains the use of the indirect bit. The section, Forming an Address, explains how each of the four types of addressing uses these components to calculate the effective address.

Ring Number

Ring numbers are found in the program counter, in the base register, and within indirect addresses. When an effective address is calculated, the highest numbered ring referenced in any of these locations is chosen as the ring field for the effective address. (For more information on rings, and on the process of calculating ring numbers, see Chapter 4.)

Segment Number

The segment number is generally provided in one of three ways:

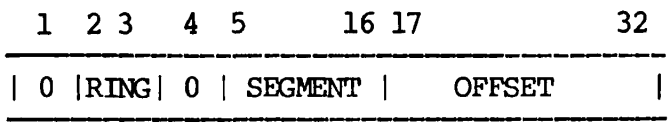
- If the instruction contains a base register field, the segment number is found in the specified base register.
- If the instruction does not contain a base register field, the segment number is found in the program counter.
- In indirect addressing, the segment number field contains the segment number.

Base Registers: There are four base registers available for use in address calculation:

- The procedure base register (PB)
- The stack base register (SB)

- The link base register (LB)
- The auxiliary base register (XB)

All of these are 32-bit registers. Their format is shown in Figure 3-1.



Bits	Name	Description
1	——	Must be 0. (See the F bit in the section on Calculating Indirect Pointers, in Chapter 8, for the explanation of this.)
2-3	Ring	Specifies the ring number.
4	——	Must be 0. (See the E bit in the section on Calculating Indirect Pointers, in Chapter 8, for the explanation of this.)
5-16	Segment	Specifies the segment number.
17-32	Offset	Specifies the offset value.

Format of Virtual Addresses and Base Registers
Figure 3-1

The PB contains the address of the currently active procedure. It is unique among the four base registers because its offset is always 0.

The program counter always contains a trusted copy of the segment number in the PB. Therefore, an instruction that contains no base register field uses the same segment number as one that specifies the PB.

SB contains the starting address of the stack for the currently active stack frame. LB contains the starting address of a save area for static variables, such as an entry control block. (See Chapter 8.) XB usually contains a temporary pointer, such as that to a FORTRAN common block. These three registers usually have non-zero offsets. Thus, they supply not only the segment number but also an offset address relative to that number.

Offset

The offset portion of an effective address is supplied by one or more of the following components:

- Displacement: a 16-bit number given explicitly within the instruction.
- Base register: if the base register is SB, LB, or XB, it will contain an offset to be added to the displacement given within the instruction.
- Index register: if an index register is used, then the contents of that index register are to be added to whatever other offset has been calculated.
- Indirect address: if indirect addressing is used, the indirect address will contain the offset.

In summary, an offset can be calculated in any of the following ways:

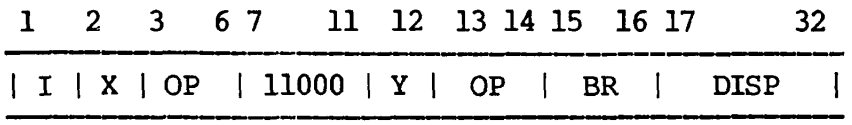
- Displacement
- Displacement + offset from BR
- Displacement + index register
- Displacement + offset from BR + index register
- Indirect address
- Indirect address + index register

The instruction format tells the processor which method to use.

COMPONENTS OF AN INSTRUCTION

Instruction Format

Figure 3-2 diagrams a typical instruction format. Thus, it shows how all the fields described in this chapter fit together into a single instruction.



Bits	Mnem	Name	Description
1	I	Indirect bit	Specifies indirect addressing.
2	X	Index field	Specifies use of an index register.
3-6	OP	Opcode	Specifies the operation to perform.
7-11	——	——	Specifies instruction format.
12	Y	Index field	Specifies use of an index register.
15-16	BR	Base register	Specifies the base register to use.
13-14	OP	Opcode	Specifies the operation to perform.
17-32	DISP	Displacement	Specifies a 16-bit offset.

Format of a Typical Instruction (V Mode, Long)
Figure 3-2

Indirect Bit

An instruction may contain an indirect bit. If this bit is 1, it signifies that the address being calculated is an indirect address. If this bit is 0, the address is a direct address. (Indirect addresses are explained under Forming an Address, later in this chapter.)

Index Register Field

An instruction may specify two index registers by using the X and Y fields. Each of these fields is one bit long. These fields are encoded with the I field to specify indexing. (See Table 3-6 for the encoding.) If an index register is specified, then the contents of that index register are added to whatever other offset has been calculated.

Base Register Field

The base register field of an instruction may contain one of the following four values:

Value	Base Register
00	PB (Procedure Base)
01	SB (Stack Base)
10	LB (Link Base)
11	XB (Auxiliary Base)

The value tells the processor which base register to check for the correct segment number (and, perhaps, offset).

Displacement

The displacement field contains a 16-bit number representing an offset within a segment. As the section on Offset explained, the value given by the displacement may either stand alone or have other values added to it to provide the actual offset for the effective address.

FORMING AN ADDRESS

The processor uses the contents of the fields in a memory reference instruction to select which of the four types of address formations to use:

- Direct
- Indexed
- Indirect
- Indirect indexed

Direct Addressing

In direct addressing, the processor forms the effective address by adding the contents of the base register to the displacement.

Indexed Addressing

The processor adds the contents of the base register, index register, and displacement to produce the effective address.

S, R, and V mode instructions that contain 1101 in bits 3-6 cannot specify indexing. See the tables at the end of this chapter for specific information.

Indirect Addressing

Short Form Indirection: Depending on the addressing mode, indirect addressing takes one of two forms. In the first, the processor treats the displacement as the address of a location in the procedure segment. The processor uses the contents of the addressed location as the effective address. This is called short form, or 16-bit, indirection.

Some addressing modes allow more than one level of indirection. (See the 16S, 32S, and 32R sections at the end of this chapter.) In these cases, the processor uses the displacement as the address of some location in the address space. If this addressed location contains another indirect address, then the processor uses these contents as the address of another location in memory. This indirection chain is followed until one addressed location does not contain an indirect address; these contents are called the result of the chain. The processor uses the result of the chain as the effective address.

The tables at the end of this chapter specify the number of levels of indirection supported by each addressing mode.

Long Form Indirection: In long form indirect addressing, the instruction points to a location in memory that contains a 32-bit (or, more rarely, 48-bit) pointer. These long pointers contain not only addresses but also 2 or 3 bits that provide additional information.

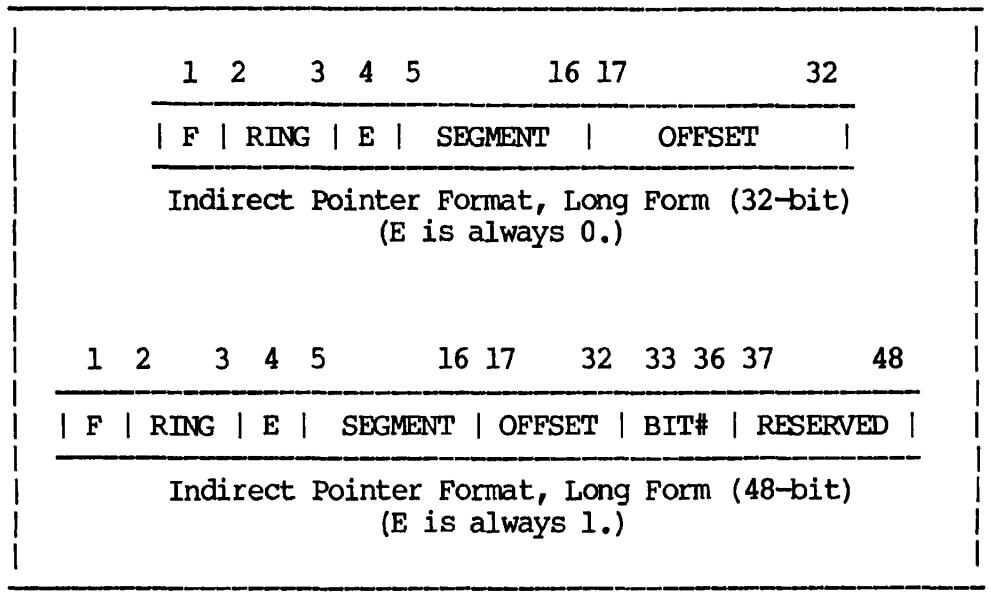
Figure 3-3 shows the format of those pointers. The bits of special interest are the extension bit (or E bit), the fault bit (or F bit), and the bit number field.

The functions of these three fields are as follows:

F bit — If F = 1, a pointer fault is generated when this indirect address is used. (See Chapter 11 for information on pointer faults.)

E bit — If E = 0, the pointer is a 32-bit pointer. If E = 1, the pointer is a 48-bit pointer. (Throughout the rest of the chapter, discussions will assume that the 32-bit format is being used.)

Bit number — Permits you to specify (or point to) a particular bit within an address offset.



Pointer Formats for Long Form Indirection
Figure 3-3

Indirect Indexed Addressing

This type of addressing takes one of two forms: indirect preindexed, or indirect postindexed.

When calculating a preindexed indirect address, the processor adds the value of the index register to the contents of the base register and displacement and uses the sum as an indirect address. It resolves any indirection chain and uses the result of the chain (or the indirect address itself, if there was no chain to follow) as the effective address.

When calculating a postindexed indirect address, the processor adds the contents of the base register and displacement and uses the result as an indirect address. It resolves any indirection chain, then adds the result of the chain (or the indirect address itself, if there was no chain to follow) to the contents of the specified index register to form the effective address.

ADDRESSING MODES

The first part of this chapter described several ways to specify an address with information contained within an instruction. Once the processor calculates the effective address, it can reference whatever information is contained in the location specified by the effective address. This section describes the ways to specify an address in an instruction and how the processor forms the effective address.

The 50 Series processors support four modes of addressing, each of which forms addresses differently. Depending on the program and personal preference, one or two of these modes may be more useful than another. The three most important modes are:

- V, or virtual
- I, or general register
- R, or relative

The fourth mode — S, or sectored, mode — is supported for historical reasons.

V Mode

V mode performs short and long operations and has a wide variety of registers to use. A short (halfword) instruction in this mode can reference the first 256 locations of both the stack and link, as well as the 224 locations on either side of the current location in the procedure segment. A long (word) V mode instruction can directly reference all locations in four segments. Indirect addressing can reference all locations in up to 4096 128-Kbyte segments.

I Mode

When referencing memory, I mode is similar to 32-bit V mode. The difference is that I mode short operations reference 8 32-bit general purpose registers for use as index registers, accumulators, counters, or the like. I mode long operations have the same referencing power as V mode long operations. They can also use five additional index registers and immediate forms.

R Mode

A sector is a block of 512 (1000 octal) contiguous memory locations. Sector 0 starts on location 0 and ends on location '777; Sector 1 begins on location '1000 and ends on location '1777; and so on.

An R mode instruction can reference any location in Sector 0, as well as a group of locations relative to the current value of the program counter. When the sector bit (S) in an R mode instruction is 0, the instruction can only reference locations in Sector 0. When S is 1, the instruction references locations relative to the current value of the program counter. The range of these relative locations is PC - '360 to PC + '377, inclusive.

Note that an R mode instruction that specifies a location in the range PC - '361 to PC - '400, inclusive, selects a special addressing code, such as stack register. These special codes are explained in more detail in Tables 3-6 and 3-7.

S Mode

Like R mode instructions, S mode instructions contain a sector bit. When S is 0, references are to Sector 0. When S is 1, however, references are only to those locations within the sector containing the instruction.

Note that S mode is a holdover from early Prime machines that were based on the Honeywell 316 and 516 minicomputers. When operating in S mode, the 50 Series processors act exactly as these early machines do.

SUMMARY OF ADDRESSING MODES

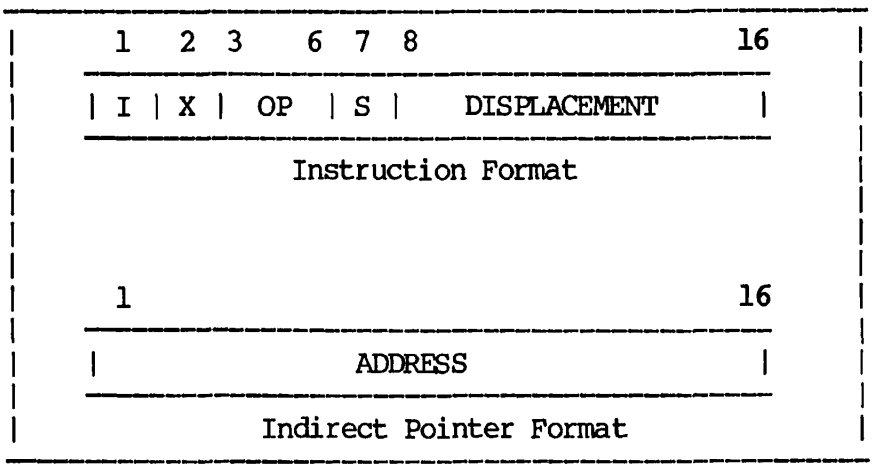
The figures and tables in the rest of this chapter present summaries of each addressing mode. Table 3-1 summarizes useful information about all the modes.

Table 3-1
Summary of Addressing Modes

Mode	Address Length	Addressing Range	# Index Regs	Indirection Levels
16S direct	14 bits	1024 halfwords	One	
16S indirect	14 bits	16K halfwords	One	Multiple
32S direct	15 bits	1024 halfwords	One	
32S indirect	15 bits	32K halfwords	One	Multiple
32R direct	15 bits	1008 halfwords	One	
32R indirect	15 bits	32K halfwords	One	Multiple
64R direct	16 bits	1008 halfwords	One	
64R indirect	16 bits	64K halfwords	One	One
64V short	16 bits	64K halfwords: +256 SB relative +256 LB relative +/-256 PC relative +512 PB absolute	One	One
64V long	28 bits	4 segments*	Two	One
64V indirect	28 bits	4096 segments*	Two	One
32I long	28 bits	4 Segments*	Seven	One
32I indirect	28 bits	4096 segments*	Seven	One

* All segments contain 128 Kbytes.

64V Mode Short Form



64V Mode Formats, Short Form
Figure 3-4

Table 3-2
64V Mode Short Form Summary

I	X	S	Disp	Inst Type	Form of EA	Example
0	0	0	0-'7@	Direct	LDA ADR	REG
			'10-'377	Direct		SB+D
			'400-'777	Direct@@		LB+D
0	1	0	0-'7@	Indexed	LDA ADR,1	REG, if D+X<'7;@
						SB+D+X, if D+X>'7@
			'10-'377	Indexed		SB+D+X
			'400-'777	Indexed@@		LB+D+X
1	0	0	0-'7@	Indirect	LDA ADR,*	I(REG)
			'10-'777	Indirect		I(PB+D)
1	1	0	0-'77	Indirect, preindexed	LDA ADR,1*	I(REG), if D+X<'7;@
						I(PB+D+X), if D+X>'7@
			'10-'77	Indirect, preindexed	LDA ADR,1*	I(PB+D+X)
			'100-'777	Indirect, postindexed	LDA ADR,*1	I(PB+D)+X
0	0	1	'-340-'+377	Direct	LDA ADR	PC+D
0	1	1	'-340-'+377	Indexed	LDA ADR,1	PC+D+X
1	0	1	'-340-'+377	Indirect	LDA ADR,*	I(PC+D)
1	1	1	'-340-'+377	Indirect, preindexed	LDA ADR,1*	I(PC+D+X)

Notes to Table 3-2

@ This table assumes segmented mode (bit 14 of the modals = 1). For nonsegmented mode, the displacement range is 0-'37, rather than 0-'7. This means that the range '10-'377 changes to '40-'377 in nonsegmented mode. The range '400-'777 remains unchanged.

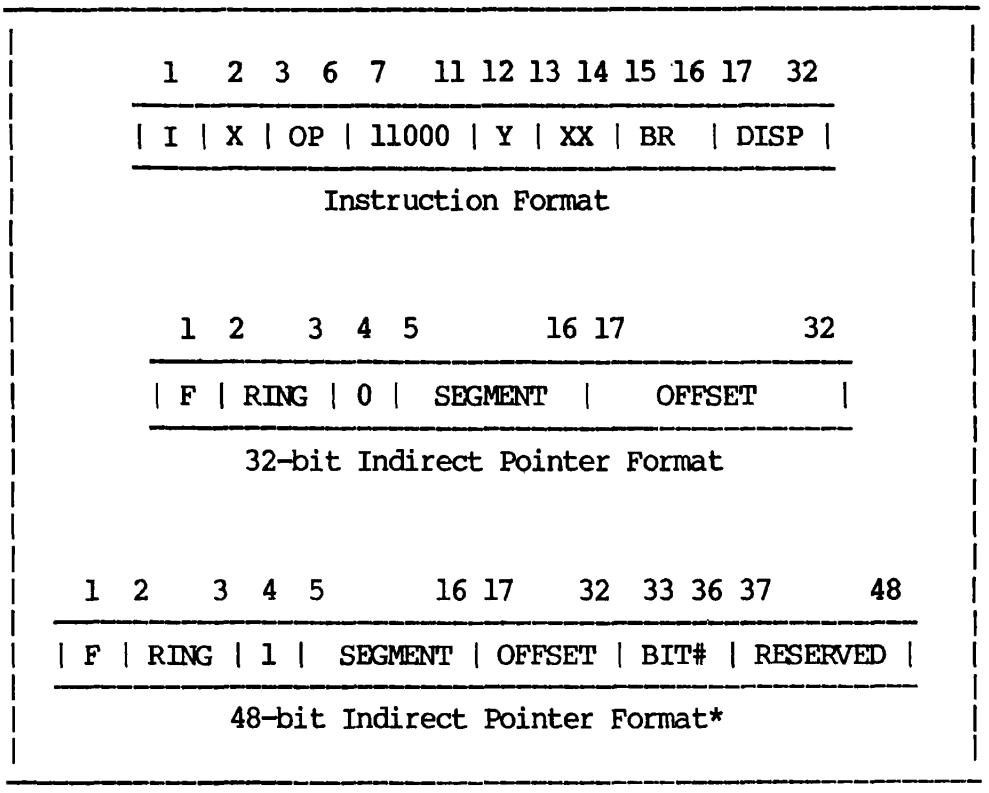
@@ In these address forms, the displacement offsets the contents of LB by '400 (bit 8=1). To compensate for this, set the contents of LB to the current value of the link frame minus '400. For example, if the segment number in LB is '4002 and the word number in the displacement is '177400, the offset of '400 gives the location of the link frame as segment number '4002, word number 0.

This mode allows one level of indexing, and one of indirection.

REG refers to a location in the register file. See Address Traps at the end of this chapter.

The instructions STX, FLX, DFLX, LDX, LDY, STY, and JSX do not do indexing. The effective address is formed as if bit 2 = 0.

64V Mode, Long Form and Indirect Form



* This indirect format is used only by a few instructions; most use the 32-bit form.

64V Mode Formats, Long Form and Indirect Form
Figure 3-5

Table 3-3
64V Mode Long Form, Indirect Summary

I	X	Y	BR	Instruction Type	Example	Form of EA
0	0	0	00	Direct	LDA ADR	PB/D
			01			SB+D
			10			LB+D
			11			XB+D
0	0	1	00	Indexed by Y	LDA ADR,Y	PB/D+Y
			01			SB+D+Y
			10			LB+D+Y
			11			XB+D+Y
0	1	0	00	Indexed by X	LDA ADR,X	PB/D+X
			01			SB+D+X
			10			LB+D+X
			11			XB+D+X
0	1	1	00	Indirect	LDA ADR,*	I(PB/D)
			01			I(SB+D)
			10			I(LB+D)
			11			I(XB+D)
1	0	0	00	Preindexed by Y	LDA ADR,Y,*	I(PB/D+Y)
			01			I(SB+D+Y)
			10			I(LB+D+Y)
			11			I(XB+D+Y)
1	0	1	00	Postindexed by Y	LDA ADR,*Y	I(PB/D)+Y
			01			I(SB+D)+Y
			10			I(LB+D)+Y
			11			I(XB+D)+Y
1	1	0	00	Preindexed by X	LDA ADR,X,*	I(PB/D+X)
			01			I(SB+D+X)
			10			I(LB+D+X)
			11			I(XB+D+X)
1	1	1	00	Postindexed by X	LDA ADR,*X	I(PB/D)+X
			01			I(SB+D)+X
			10			I(LB+D)+X
			11			I(XB+D)+X

Notes to Table 3-3

The processor performs X and Y indexing and 32-bit word (inter-segment) indirection.

PB/D indicates that the displacement is relative to the origin of PB. PB specifies the segment number (the offset must be 0); the displacement specifies the offset.

All displacements are within the range 0-'177777.

The instructions STX, FLX, DFLX, LDX, LDY, STY, and JSX do not do indexing. The effective address is formed as shown in Table 3-4. Bit 2, the X bit, is used as part of the opcode in these instructions.

Table 3-4
Address Formation for Nonindexing Instructions

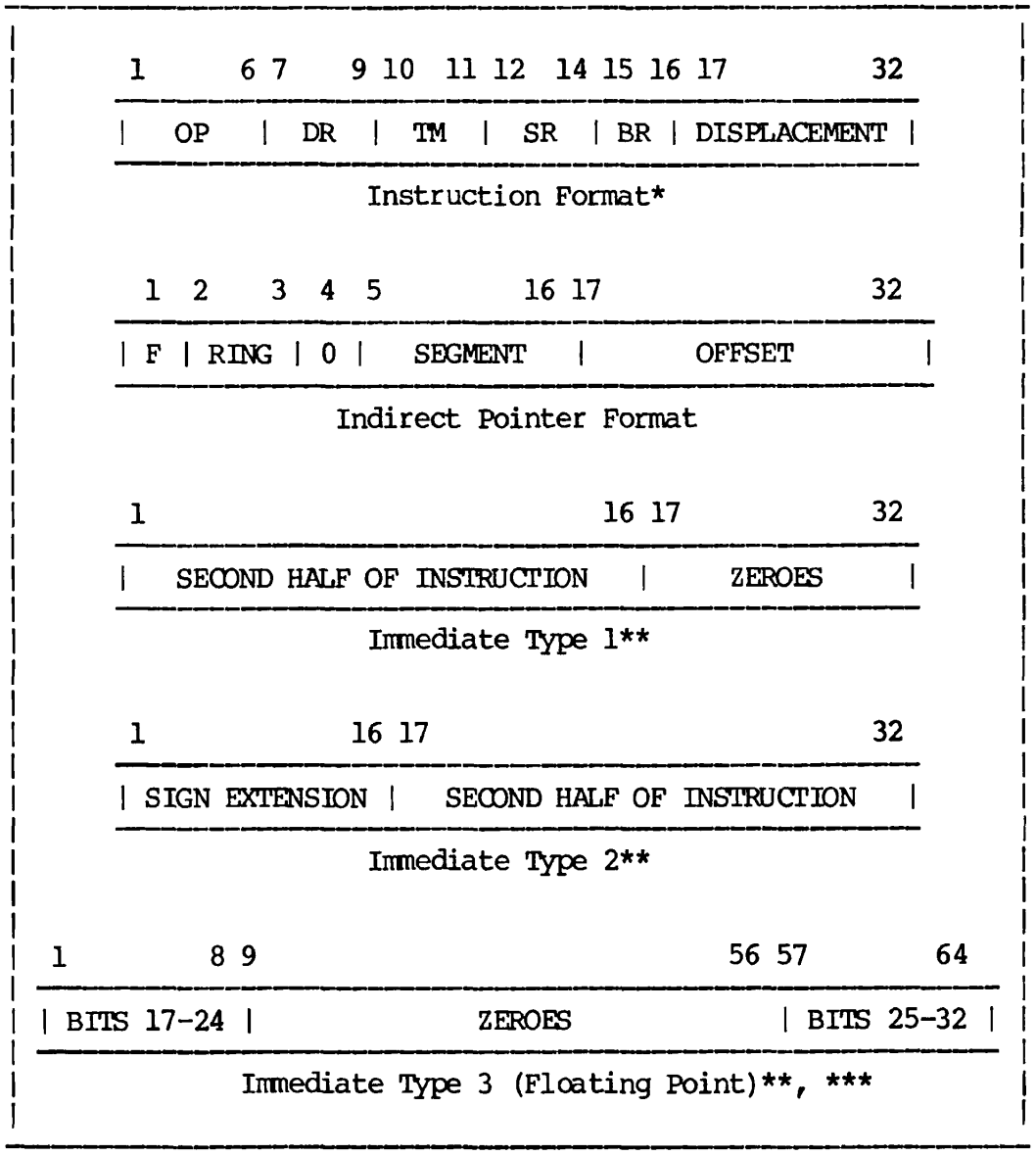
I	X	Y	400, 250-II, 550-II, 2250	750, 850	9950
0	0	0	*Direct	Direct	Direct
0	0	1	Index by Y	Direct	Direct
0	1	0	*Direct	Direct	Direct
0	1	1	I(A+X)	I(A)	Direct
1	0	0	I(A+Y)	I(A)	I(A)
1	0	1	*I(A)	I(A)	I(A)
1	1	0	I(A+X)	I(A)	I(A)
1	1	1	*I(A)	I(A)	I(A)

Notes to Table 3-4

* These modes should be used to ensure consistent behavior across processors.

The symbol $\underline{A}$ in Table 3-4 represents the value calculated from the base register (PB, SB, LB, or XB) and displacement in the instruction.

32I Mode



32I Mode Formats
Figure 3-6

Notes to Figure 3-6

* TM represents the tag modifier, which specifies the type of register to use.

** The instruction itself specifies the type of immediate to use. When the instruction executes, the processor forms the immediate in the appropriate form and stores it internally for use in the operation. The three formats

shown in Figure 3-6 represent the value that is stored internally.

*** Bits 1-8 of Immediate Type 3 are formed from bits 17-24 of the I mode instruction. Similarly, bits 57-64 are formed from bits 25-32 of the I mode instruction.

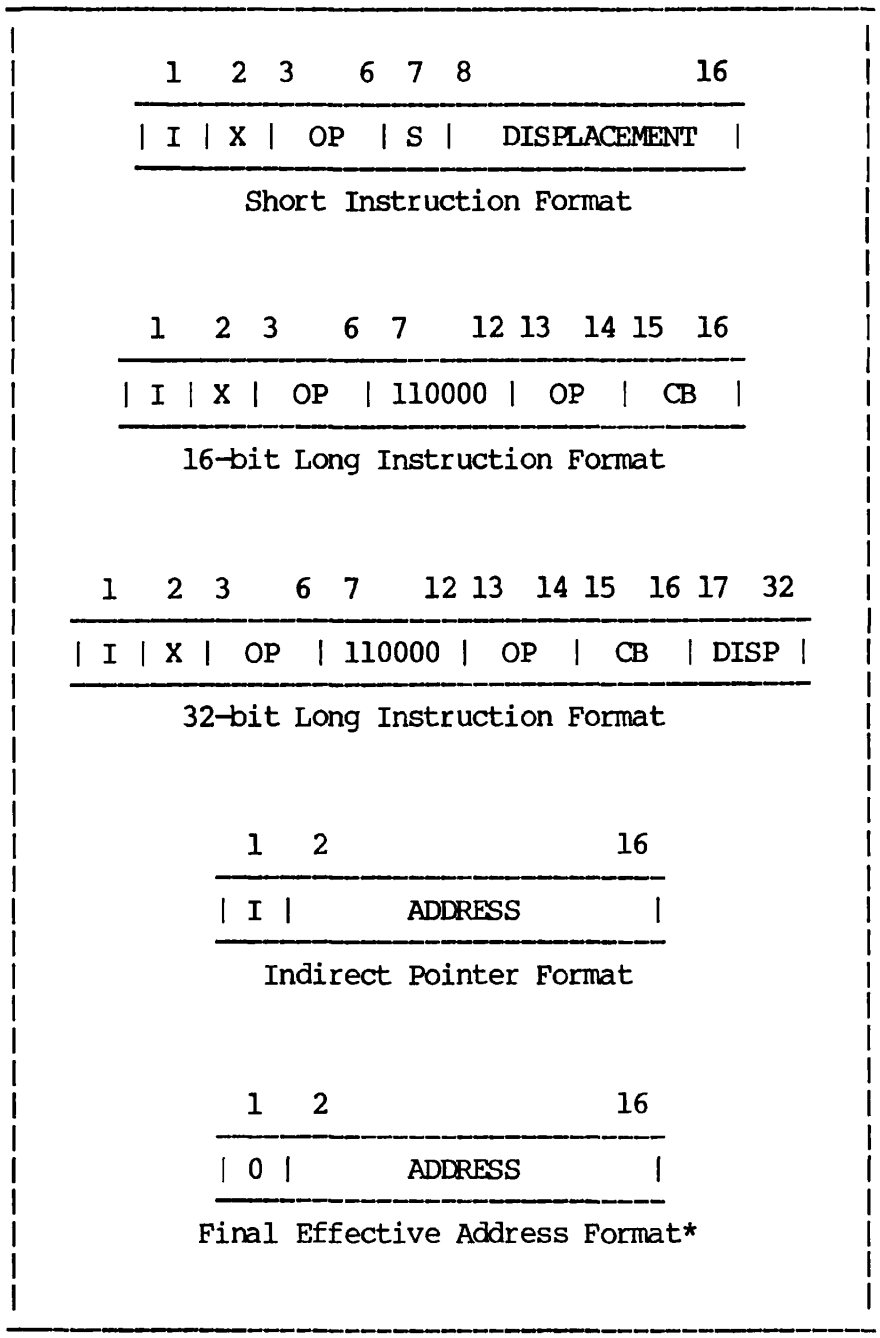
Table 3-5
32I Mode Summary

TM	SR	BR	Instruction Type	EA
3	0	-	Indirect	(D+B)*
3	>0	-	Indirect postindexed	(D+B)*+S
2	0	-	Indirect	(D+B)*
2	>0	-	Indirect preindexed	(D+B+S)*
1	0	-	Direct	D+B
1	>0	-	Indexed	D+B+S
0	0-7	0	Register-to-register	—
0	0	1	Immediate type 1	—
0	>0	1	Immediate type 2	—
0	0	2	Immediate type 3	—
0	1	2	Floating register source (FR0)	—
0	2	2	Undefined; generates UII (unimplemented instruction) fault	—
0	3	2	Floating register source (FR1)	—
0	4-7	2	Undefined; generates UII fault	—
0	—	3	Undefined; generates UII fault	—

Note to Table 3-5

Displacements are within the range 0 to '177777, inclusive.

32R Mode



32R Mode Formats
Figure 3-7

Table 3-6
32R Mode Summary

I	X	S	CB	Displacement	Instruction Type	Form of EA
0	0	0	—	0 to '777	Direct	0/D
0	1	0	—	0 to '777	Indexed	0/D+X
1	0	0	—	0 to '777	Indirect	I(0/D)
1	1	0	—	0 to '77	Indirect, preindexed	I(0/D+X)
1	1	0	—	'100 to'777	Indirect, postindexed	I(0/D)+X
0	0	1	—	'-360 to '+377	Direct	P+D
0	1	1	—	'-360 to '+377	Indexed	P+D+X
1	0	1	—	'-360 to '+377	Indirect	I(P+D)
1	1	1	—	'-360 to '+377	Indirect postindexed	I(P+D)+X
0	0	1	2	—	@Postincrement	SP
0	1	1	2	—	@Postincrement, indirect, postindexed	I(SP)+X
1	0	1	2	—	@Postincrement, indirect	I(SP)
0	0	1	3	—	#Predecrement	SP-1
0	1	1	3	—	#Predecrement, indirect, postindexed	I(SP-1)+X
1	0	1	3	—	#Predecrement, indirect	I(SP-1)
0	0	1	0	0 to '177777	*Direct, long reach	D
0	1	1	0	0 to '177777	*Indexed, long reach	D+X
1	0	1	0	0 to '177777	*Indirect, long reach	I(D)
1	1	1	0	0 to '177777	*Indirect, preindexed, long reach	I(D+X)
1	1	1	2	0 to '177777	*Indirect, postindexed, long reach	I(D)+X
0	0	1	1	0 to '177777	*Direct, stack relative	D+SP
0	1	1	1	0 to '177777	*Indexed, stack relative	D+SP+X
1	0	1	1	0 to '177777	*Indirect, stack relative	I(D+SP)
1	1	1	1	0 to '177777	*Indirect, preindexed stack relative	I(D+SP+X)
1	1	1	3	0 to '177777	*Indirect, postindexed stack relative	I(D+SP)+X

Note to Figure 3-7

The final form of an effective address in 32R mode is only 15 bits wide. Special hardware logic exists to truncate the effective address to this length. The program counter, however, is a full 16 bits wide. Multilevel indirection is a feature of 32R mode.

Notes to Table 3-6

* These instruction types use the 32-bit long format shown in Figure 3-7.

@ These instruction types use the 16-bit long format shown in Figure 3-7. They also increment the contents of SP by 1 during EA formation.

These instruction types use the 16-bit long format shown in Figure 3-7. They also decrement the contents of SP by 1 during EA formation.

For all instruction types listed above, address traps can occur when any part of the EA formation results in an address in the range 0-'7 (segmented mode) or 0-'37 (unsegmented mode). See the end of this chapter for more information.

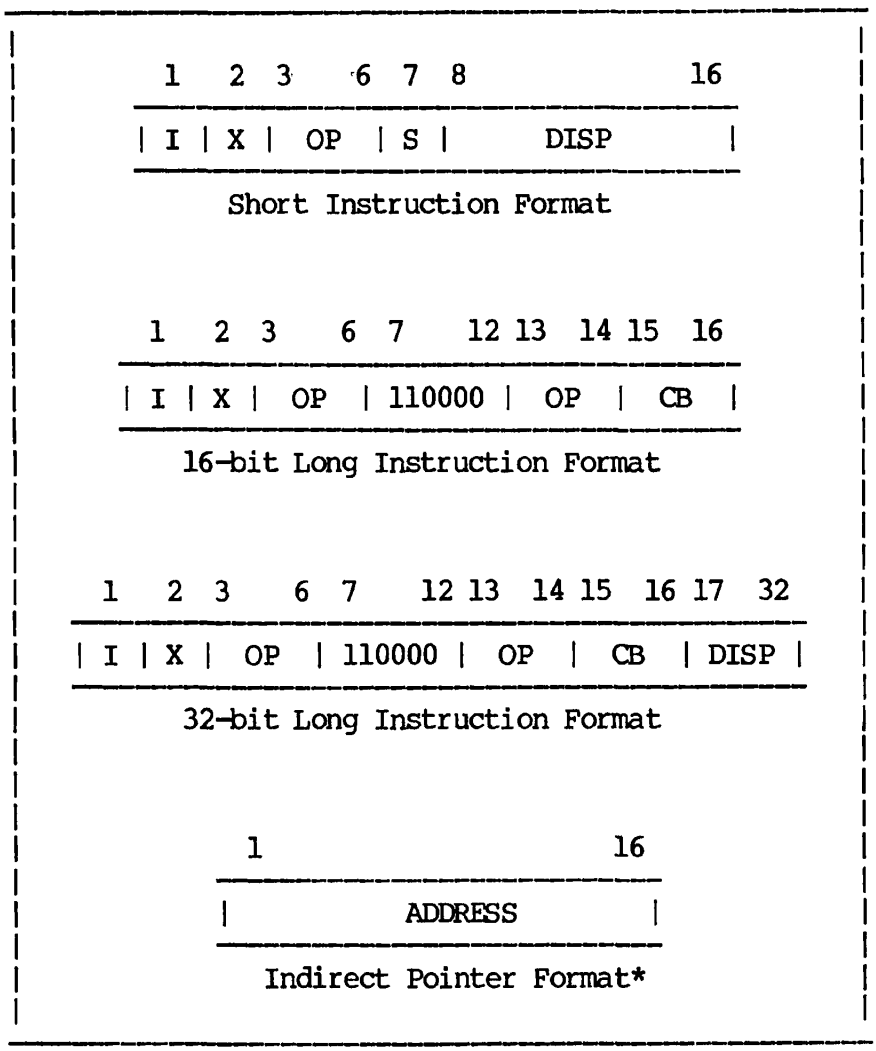
The processor performs one level of indexing and multiple levels of indirection.

0/D indicates that the displacement is within Sector 0; P+D, within the current sector.

CB represents the class bits of the instruction.

The instructions STX, FLX, JDX, JIX, LDX, and JSX do not do indexing. The processor treats the X bit as a 0 to determine what addressing mode to use. For example, if one of these instructions specifies I, X, S, and CB as 0113, the processor interprets it as 0013.

64R Mode



*Only a single level of indirection is possible in 64R mode.

64R Mode Formats
Figure 3-8

Table 3-7
64R Mode Summary

I	X	S	CB	Displacement	Instruction Type	Form of EA
0	0	0	—	0 to '777	Direct	0/D
0	1	0	—	0 to '777	Indexed	0/D+X
1	0	0	—	0 to '777	Indirect	I(0/D)
1	1	0	—	0 to '77	Indirect, preindexed	I(0/D+X)
1	1	0	—	'100 to'777	Indirect, postindexed	I(0/D)+X
0	0	1	—	'-360 to '+377	Direct	P+D
0	1	1	—	'-360 to '+377	Indexed	P+D+X
1	0	1	—	'-360 to '+377	Indirect	I(P+D)
1	1	1	—	'-360 to '+377	Indirect postindexed	I(P+D)+X
0	0	1	2	—	@Postincrement	SP
0	1	1	2	—	@Postincrement, indirect, postindexed	I(SP)+X
1	0	1	2	—	@Postincrement, indirect	I(SP)
0	0	1	3	—	#Predecrement	SP-1
0	1	1	3	—	#Predecrement, indirect, postindexed	I(SP-1)+X
1	0	1	3	—	#Predecrement, indirect	I(SP-1)
0	0	1	0	0 to '177777	*Direct, long reach	D
0	1	1	0	0 to '177777	*Indexed, long reach	D+X
1	0	1	0	0 to '177777	*Indirect, long reach	I(D)
1	1	1	0	0 to '177777	*Indirect, preindexed, long reach	I(D+X)
1	1	1	2	0 to '177777	*Indirect, postindexed, long reach	I(D)+X
0	0	1	1	0 to '177777	*Direct, stack relative	D+SP
0	1	1	1	0 to '177777	*Indexed, stack relative	D+SP+X
1	0	1	1	0 to '177777	*Indirect, stack relative	I(D+SP)
1	1	1	1	0 to '177777	*Indirect, preindexed stack relative	I(D+SP+X)
1	1	1	3	0 to '177777	*Indirect, postindexed stack relative	I(D+SP)+X

Notes to Table 3-7

For all the instruction types listed in Table 3-7, address traps can occur when any part of the EA formation results in an address in the range 0-'7 (segmented mode) or 0-'37 (unsegmented mode). See the end of this chapter for more information.

* These instruction types use the 32-bit long format shown in Figure 3-8.

@ These instruction types use the 16-bit long format shown in Figure 3-8. They also increment the contents of SP by 1 during EA formation.

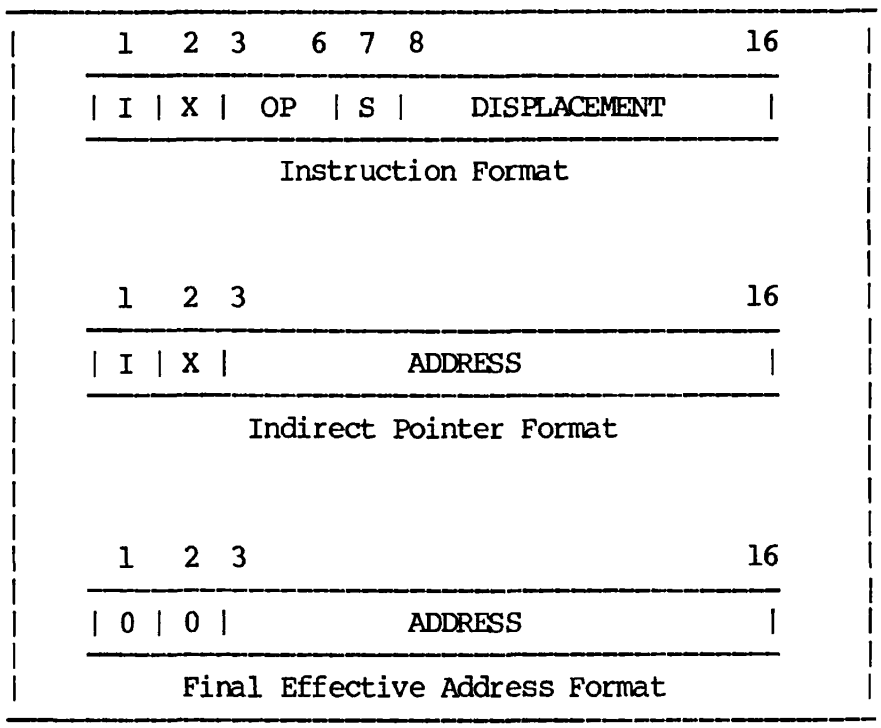
The processor performs one level of indexing and multiple levels of indirection.

0/D indicates that the displacement is within Sector 0; P+D, within the current sector.

CB represents the class bits of the instruction.

The instructions STX, FLX, JDX, JIX, LDX, and JSX do not do indexing. The processor treats the X bit as a 0 to determine what addressing mode to use. For example, if one of these instructions specifies I, X, S, and CB as 0113, the processor interprets it as 0013.

16S Mode



16S Mode Formats
Figure 3-9

Note to Figure 3-9

The final form of effective addresses in S mode are only 14 bits wide. Special hardware logic exists to truncate the effective address to this length. The program counter, however, is a full 16 bits wide.

Table 3-8
16S Mode Summary

I	X	S	Disp	Instruction Type	Example	EA Form
0	0	0	0-'777	Direct	LDA ADR	0/D
0	0	1	0-'777	Direct	LDA ADR	C/D
0	1	0	0-'777	Indexed	LDA ADR,1	0/D+X
0	1	1	0-'777	Indexed	LDA ADR,1	C/D+X
1	0	0	0-'777	Indirect	LDA ADR,*	I(0/D)
1	0	1	0-'777	Indirect	LDA ADR,*	I(C/D)
1	1	0	0-'777	Indirect preindexed	LDA ADR,1*	I(D+X)
1	1	1	0-'777	Indirect preindexed	LDA ADR,1*	I(D+X)

Notes to Table 3-8

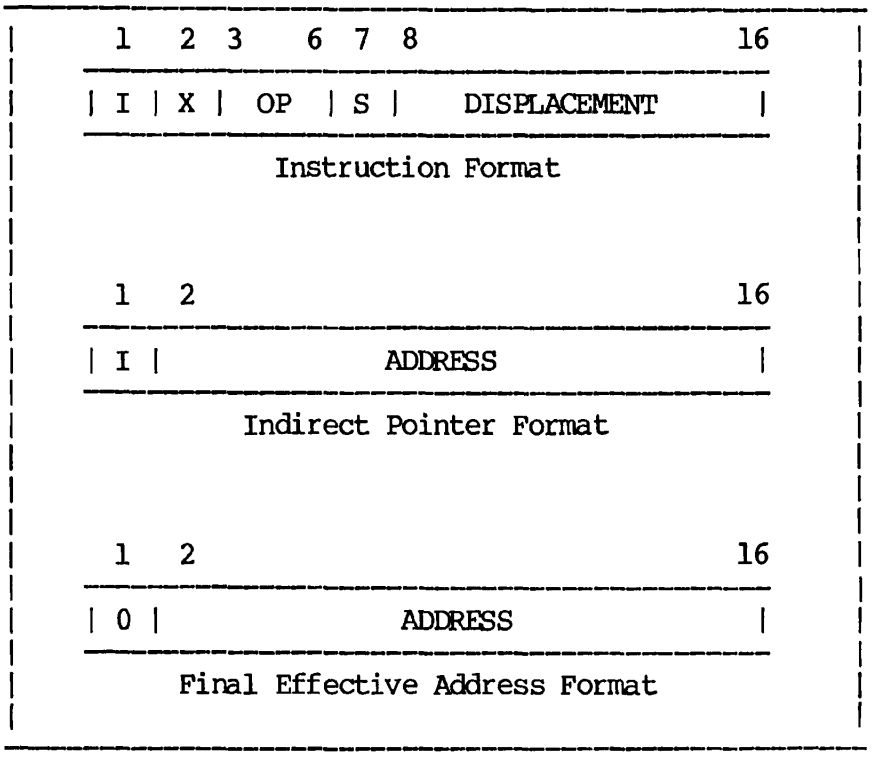
The processor performs indexing before resolving each level of indirection.

This mode allows multiple levels of both indexing and indirection.

The instructions, LDX and STX, cannot do indexing. The effective address is formed as if bit 2 = 0.

0/D indicates that the displacement is within Sector 0; C/D, within the current sector.

32S Mode



32S Mode Formats
Figure 3-10

Note to Figure 3-10

The final form of effective addresses in S mode are only 15 bits wide. Special hardware logic exists to truncate the effective address to this length. The program counter, however, is a full 16 bits wide.

Table 3-9
32S Mode Summary

I	X	S	Disp	Instruction Type	Example	EA Form
0	0	0	0-'777	Direct	LDA ADR	0/D
0	0	1	0-'777	Direct	LDA ADR	C/D
0	1	0	0-'777	Indexed	LDA ADR,1	0/D+X
0	1	1	0-'777	Indexed	LDA ADR,1	C/D+X
1	0	0	0-'777	Indirect	LDA ADR,*	I(0/D)
1	0	1	0-'777	Indirect	LDA ADR,*	I(C/D)
1	1	0	0-'77	Indirect preindexed	LDA ADR,1*	I(D+X)
1	1	0	'100-'777	Indirect postindexed	LDA ADR,*1	I(D)+X
1	1	1	0-'777	Indirect postindexed	LDA ADR,*1	I(D)+X

Notes to Table 3-9

The processor performs indexing before resolving each level of indirection.

This mode allows one level of indexing, and multiple levels of indirection.

The instructions, LDX and STX, cannot do indexing. The effective address is formed as if bit 2 = 0.

ADDRESS TRAPS

Several of the summaries in the last section specified special cases of EA formation when the address was within a particular range. This range of addresses corresponds to registers within the current user register set in the register file. (See Chapter 9.) In segmented mode, this range is '0 to '7; in nonsegmented mode, '0 to '37. Note that this range of addresses for segmented and nonsegmented modes is referred to as the ATR, or address trap range, throughout this section.

The registers within the user register set contain information, such as general, base, floating-point, and index registers, and system status and control information. Each time any part of the EA formation generates an address within the ATR, an address trap aborts any read or write to a memory location and instead references the specific register.

Table 3-10 summarizes when address traps occur for all modes of addressing and instruction types.

Table 3-10
Address Trap Information

Mode	Inst Type	Action
16S, 32S, 32R, 64R	Memory reference	Address trap occurs if the EA falls within the ATR. The format or length of the instruction has no bearing.
	Generic	Address traps never occur.
	Generic AP	Address traps do not occur when the processor is fetching the address pointer.
64V	Two-word memory reference	Address traps never occur.
	Short format	See Table 3-11.
	16-bit indirect	Address traps occur if the EA falls within the ATR.
	32-bit indirect	Address traps never occur.
32I	All types	Address traps never occur.

When bits 17-32 of the program counter contain a value within the ATR and the processor is reading an instruction, an address trap always occurs. The only exception to this is if the machine is operating in 32I mode.

When the processor executes short format instructions in 64V mode, address traps can occur during operand fetches or indirect fetches. Table 3-11 lists the conditions that must be present for an address trap to occur.

Table 3-11

Address Trap Action for Short Format Instructions, 64V Mode

I	X	S	D	Action
0	0	0	'0 to '7	Takes address trap.
0	0	0	'10 to '37	Takes address trap only if segmentation is off.
0	0	0	'40 to '377	Cannot take address trap.
0	0	1	-'340 to +'377	Takes address trap if EA (D+RP) is within the ATR.
0	1	0	'0 through ATR	Takes address trap if D+X is within the ATR. If D+X is outside the ATR, the EA is SB(seg #) | D+X (850, 750, 9950) or SB(seg #) | D+X+SB(word #) (all other V mode Machines).
			From ATR to '377	Cannot take address trap; EA is SB+D+X (P750, P850, 9950). All other machines take address trap if D+X is within the ATR.
			'400 to '777	Cannot take address trap.
0	1	1	-'340 to +'377	Takes address trap if EA (D+X+RP) is within the ATR.
1	0	0	'0 to '777	Takes address trap if D is within the ATR.*
1	0	1	-'340 to +'377	Takes address trap if EA ((RP+D)) is within the ATR.*
1	1	0	'0 to '777	Takes address trap if D<'100 and D+X is within the ATR.*
1	1	1	-'340 to +'377	Takes address trap if EA (D+RP) is within the ATR.*

Note to Table 3-11

The indirect address also takes an address trap if EA is within the ATR.

If an instruction specifies a write operation that could potentially cause an address trap, the instruction loads the data to be written into a temporary register. If a trap occurs, the routine aborts the memory write. It loads the specified register file location with the contents of the temporary register.

If the instruction specifies a read operation that causes an address trap, the trap routine aborts the memory read and fetches the contents of a register file location. It loads this value into the cache to save it. The trap routine loads it into the area specified in the instruction from the cache location.

Table 3-12 shows the address trap locations and the registers to which they correspond. For more information on the register file, see Chapter 9.

SUMMARY

The fields of a memory reference instruction specify information used to form an effective address. These fields specify which information is to be used in the formation, how the formation is to be done, and — in conjunction with the rest of the program — the addressing mode under which the address is to be formed. Depending on the segmentation mode and the EA formation, addresses can reference registers within the current user register file as well as memory locations.

Table 3-12
Address Trap/Register File Correspondence

AT	S, R Modes	V Mode
'0	X	X
'1	A	A, LH
'2	B	LL
'3	S	Y
'4	FAC bits 1-16	FAC bits 1-16
'5	FAC bits 17-32	FAC bits 17-32
'6	FP exponent	FP exponent
'7	PC, LSBs	PC, LSBs
'10*	DTAR3H	DTAR3H
'11*	FCODEH	FCODEH
'12*	FADDRL	FADDRL
'13*		
'14*		SBH
'15*		SBL
'16*		LBH
'17*		LBL
'20*	DMA cell '20H	DMA cell '20H
'21*	DMA cell '20L	DMA cell '20L
'22*	DMA cell '22H	DMA cell '22H
'23*	DMA cell '22L	DMA cell '22L
'24*	DMA cell '24H	DMA cell '24H
'25*	DMA cell '24L	DMA cell '24L
'26*	DMA cell '26H	DMA cell '26H
'27*	DMA cell '26L	DMA cell '26L
'30*	DMA cell '30H	DMA cell '30H
'31*	DMA cell '30L	DMA cell '30L
'32*	DMA cell '32H	DMA cell '32H
'33*	DMA cell '32L	DMA cell '32L
'34*	DMA cell '34H	DMA cell '34H
'35*	DMA cell '34L	DMA cell '34L
'36*	DMA cell '36H	DMA cell '36H
'37*	DMA cell '36L	DMA cell '36L

Note to Table 3-12

* These correspond to user register file locations only in nonsegmented mode.

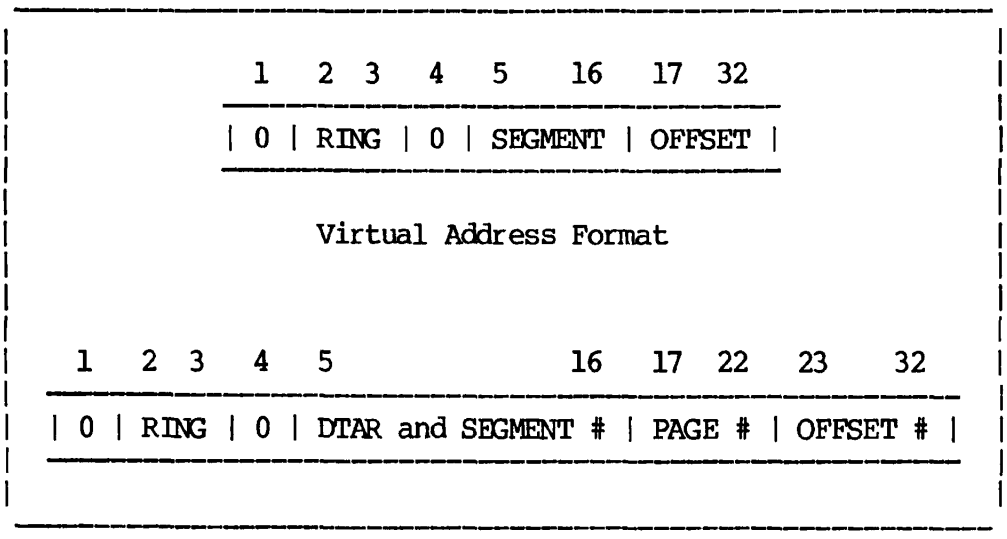
4

Memory Management

The last chapter showed how the 50 Series systems use information contained in an instruction to form a virtual address. This address specifies a location in the virtual address space, which may or may not correspond to a location currently loaded in physical memory. This means that the processor must find some way to convert the virtual address into something that can address a physical memory location, and must then search physical memory for that location. This chapter describes how the processor uses a virtual address to address memory, and describes the data structures (registers and tables) that facilitate the reference.

THE VIRTUAL ADDRESS

A virtual address is a reflection of the segmented virtual address space the user sees. A physical address, similarly, must reflect the pages that make up physical memory. How does the processor make the transition from a segment-oriented address to a page-oriented one? The virtual address (diagrammed in Figure 4-1) is the starting point. (As this figure shows, the page number and DTAR are generally transparent to the user. They are seen only by the mapping hardware.)



Virtual Address Format as Seen by
the Mapping Hardware

Figure 4-1

The steps the processor takes to convert this virtual address into a physical address are:

1. Check the STLB and the cache. If both of these contain the correct information, the reference can be completed. If not, go on to the next step.

2. Translate the virtual address into a physical address. During the translation, identify if the virtual page containing the information is currently loaded into main memory. If it is, load the physical page address (the result of the translation) into the STLB and retry the access. If main memory does not contain the page, go on to the next step.

3. Find the correct virtual page on disk and move it into main memory. After the virtual page is loaded into a physical page, the reference is retried.

The first task is completely performed in hardware; the second, by a microcode routine. A software page fault handler performs all aspects of paging.

MEMORY MANAGEMENT DATA STRUCTURES

All three of the steps in the memory reference operation use several data structures to maintain needed information:

- Segmentation table lookaside buffer (STLB)
- Cache
- Descriptor table address registers (DTARs)
- Segment descriptor tables (SDTs)
- Page map tables (PMTs), for 9950 only
- Hardware page map tables (HMAPs), for other 50 Series systems
- Logical page map tables (LMAPs), for other 50 Series systems
- Memory map table (MMAP)

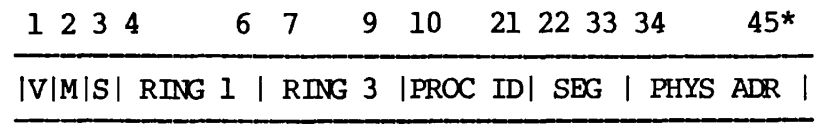
Table 4-1 shows the steps in which each structure is used.

Table 4-1
Use of Memory Management Data Structures

Structure	When Used
STLB	STLB/cache access, address translation
Cache	STLB/cache access, address translation
DTARs	STLB/cache access, address translation
SDWs	Address translation
PMTs	Address translation, paging (9950 only)
HMAPs	Address translation, paging (other 50 Series systems)
LMAPs	Address translation, paging (other 50 Series systems)
MMAP	Paging

The STLB

The STLB contains 128 entries on the 9950 and 64 entries on other 50 Series systems. Each STLB entry specifies one virtual address and one physical page address. Since each entry specifies a physical page address, each STLB entry is valid for a 2-Kbyte block (one page) of physical memory locations. Figure 4-2 describes the format of each STLB entry.



*46 for 9950, which has a 13-bit PHYS ADR field

Bits	Mnem	Description
1	V	Valid bit. Indicates if the STLB contains valid data.
2	M	Modified bit. Specifies if the physical page has been modified since its contents were loaded from disk.
3	S	Shared bit. Indicates if this entry represents a location in shared or unshared memory.
4-6	RING 1	Specifies the Ring 1 access rights that are to govern the reference.
7-9	RING 3	Specifies the Ring 3 access rights that are to govern the reference.
10-21	PROC ID	Specifies the process ID for the process making the reference to memory.
22-33	SEG	Specifies the segment number from the virtual address.
34-45*	PHYS ADR	Specifies the physical page address (from translation).

*34-46 for 9950

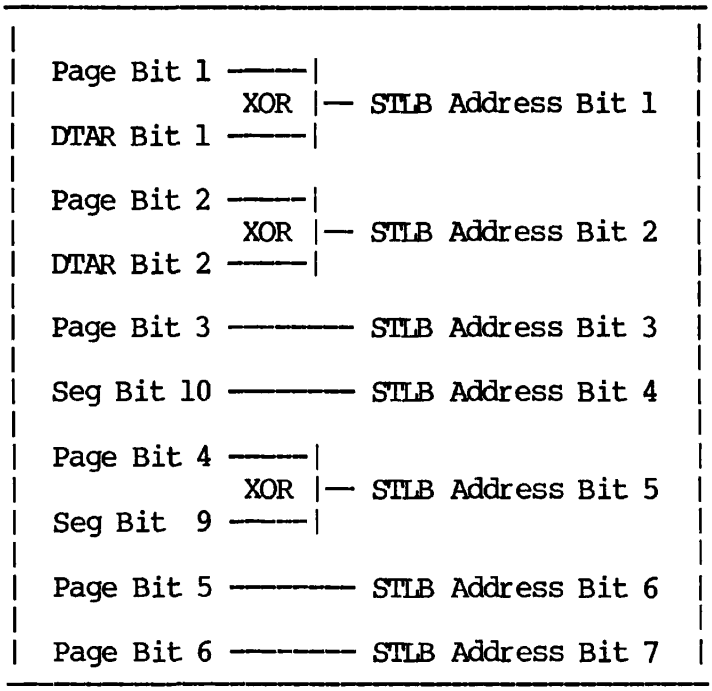
Figure 4-2
STLB Entry Format

The processor uses a hashing algorithm to access the STLB. Ten bits from the virtual address are used in the hashing algorithm, as shown in Table 4-2. This table also identifies the names that will be used for these bits in the explanation of the algorithm.

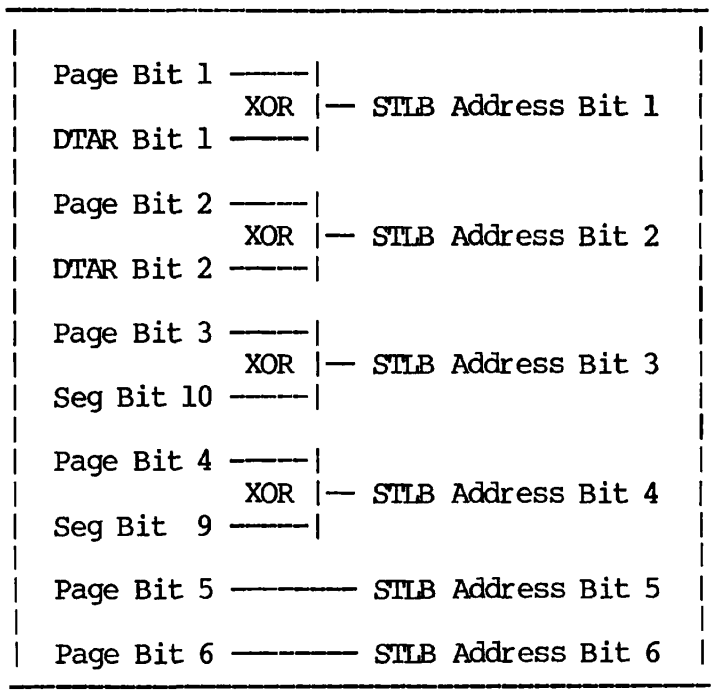
Table 4-2
Bits Used in the Hashing Algorithm

Bits	Name
Bits 5-6 of the virtual address. These specify one of the four DTARs.	DTAR Bit 1 and DTAR Bit 2
Bits 15-16 of the virtual address. These are the two least significant bits of the segment field.	Seg Bit 9 and Seg Bit 10
Bits 17-22 of the virtual address. These are all of the bits in the page field.	Page Bit 1 through Page Bit 6

The hashing algorithm exclusively ORs pairs of these bits to form a 6-bit or 7-bit address into the STLB (a 7-bit address for the 9950, a 6-bit address for other 50 Series systems). Figure 4-3 shows how the bits are ORed to form the address for the 9950 and for other 50 Series systems.



Hashing Algorithm for the 9950 STLB
Figure 4-3a

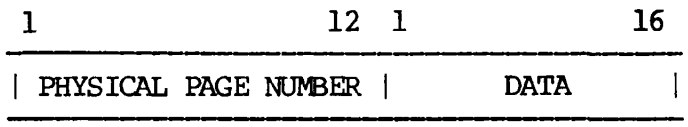


Hashing Algorithm for the STLB of Other
50 Series Machines

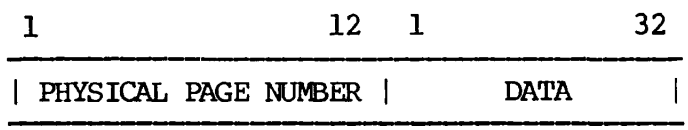
Figure 4-3b

Cache

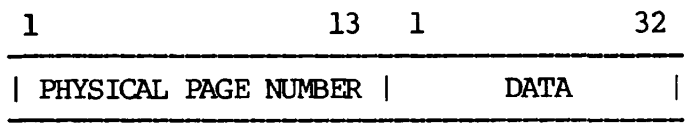
Like the STLB, the cache specifies the number of the physical page that contains the desired physical location. In addition, it contains the contents of that physical location. Figure 4-4 describes the format of each cache entry.



I450, 250-II, 550-II Cache Entry Format



750, 850, Cache Entry Format



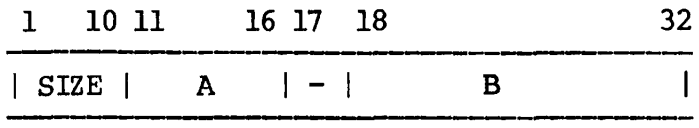
9950 Cache Entry Format

Bits	Mnem	Description
1-12 or 1-13	PHYSICAL PAGE NUMBER	Specifies the number of the physical page that contains the specified location. This is the cache index.
1-16 or 1-32	DATA	Contains a copy of the contents of a location in physical memory.

Cache Entry Format
Figure 4-4

DTARs

As described in Chapter 2, the 50 Series virtual address space is divided into four groups of 1024 segments each. Each group is referenced through a descriptor table address register (DTAR) associated with it. The public (shared) segments are referenced through DTAR0 and DTAR1; the private (unshared) segments, through DTAR2 and DTAR3. Figure 4-5 shows the format of the DTARs.

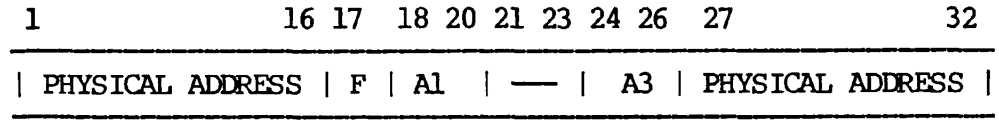


Bits	Mnem	Description
1-10	SIZE	Specifies 1024 minus the size of the segment table.
11-16	A	Bits 1-6 of the segment descriptor table physical address.
17	—	Must have the same value as bit 18.
18-32	B	Bits 7-21 of the segment descriptor table physical address.

DTAR Format
Figure 4-5

Segment Descriptor Tables

Each of the four DTARs described above points to a segment descriptor table (SDT). These SDTs contain from 0 to 1024 32-bit entries called segment descriptor words (SDWs). Each SDW describes one segment. The table must begin on an even word boundary, and must not cross a segment boundary. It must also be located in the first 8 Mbytes of physical memory, since the DTAR can specify only a 22-bit address. The format of the SDWs is shown in Figure 4-6.

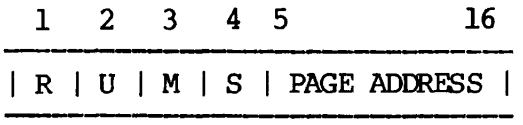


Bits	Mnem	Description
1-16	PHYSICAL ADDRESS	Bits 7-22 of an HMAP's physical starting address. Bits 17-22 of this address must be 0.
17	F	Fault bit.
18-20	A1	Specifies the access rights for Ring 1: 000 = no access 001 = gate 010 = read access 011 = read, write access 100 = reserved 101 = reserved 110 = read, execute access 111 = read, write, execute access
21-23	——	Reserved.
24-26	A3	Specifies the access rights for Ring 3. See bits 18-20 for a list of the available access codes.
27-32	PHYSICAL ADDRESS	Bits 1-6 of an HMAP's physical starting address.

Segment Descriptor Word Format
Figure 4-6

Hardware Page Map Tables

Bits 1-16 and 27-32 of each SDW contain the starting address of a hardware page map table (HMAP). Each table contains 64 16-bit entries, each of which contains information about one virtual page. An HMAP cannot cross a '200000 (65,536) boundary. Figure 4-7 shows the format of each HMAP entry.

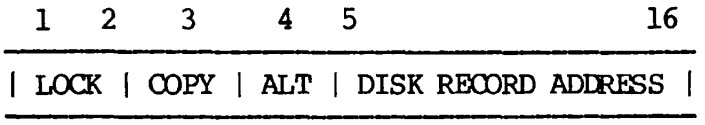


Bits	Mnem	Name	Description
1	R	Resident Bit	Indicates if the page resides in physical memory. 1 indicates residency.
2	U	Used Bit	Hardware sets U to 1 when a page is used.
3	M	Modified Bit	Hardware resets M to 0 when a page is modified.
4	S	Shared Bit	Inhibits use of cache.
5-16	PAGE ADDRESS	Page Address	Specifies high-order 12 bits of physical page address, or the address of an LMAP entry. (See Paging, below.)

Hardware Page Map Table Entry Format
Figure 4-7

Logical Page Map Tables

As mentioned earlier, each HMAP has a logical page map table (LMAP) associated with it. Each HMAP entry specifies either the physical address of a page in memory, or, if that page is not currently loaded in main memory, the address of an entry in the HMAP's associated LMAP. This LMAP entry specifies the disk address of the page. Figure 4-8 shows the format of each LMAP entry.



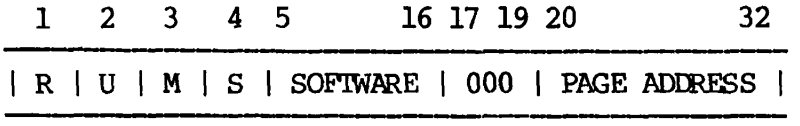
Bits	Mnem	Name	Description
1-2	LOCK	Locked Bits	If 00, the page is not locked into memory. If 01, the page is locked into memory.
3	COPY	Copy Bit	If 0, a copy of this page already exists on disk. If 1, no such copy exists.
4	ALT	Alternate Paging Device	Contains 1 if the page fault handler should use another paging device.
5-16	DISK RECORD ADDRESS	Disk Record Address	Specifies the address of the page on disk (to the nearest 8-page block).

LMAP Entry Format
Figure 4-8

Bits 1-2 of the LMAP entry are lock bits. These bits can be set to ensure that the associated page always remains in main memory and is not mapping tables, I/O buffers, or some of the data structures described in this chapter. By locking these contents into main memory, the processor can always be sure to access the correct data and complete vital operations when it cannot stop to handle a page fault.

Page Map Tables

Bits 1-16 and 27-32 of each SDW contain the starting address of a page map table (PMT). These tables contain 64 32-bit entries, each of which contains information about one page. A page map table cannot cross a '200000 (65,536) boundary. Figure 4-9 shows the format of each page map table entry.

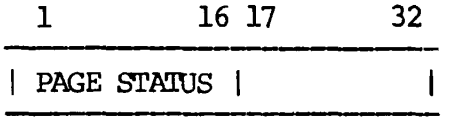


Bits	Mnem	Name	Description
1	R	Resident Bit	Indicates if the page resides in physical memory. 1 indicates residency.
2	U	Used bit	Hardware sets U to 1 when a page is used.
3	M	Modified Bit	Hardware resets M to 0 when a page is modified.
4	S	Shared Bit	Inhibits use of cache.
5-16	SOFTWARE	Software	Reserved for software use.
17-19	——	——	Must be zero.
20-32	PHYSICAL ADDRESS	Physical Address	Specifies high-order 13 bits of a physical page address.

PMT Entry Format (for 9950 only)
Figure 4-9

Memory Map

Each entry of the memory map table (MMAP) describes one physical page and tells whether it is already in use, is available for use, or does not exist. The first two descriptions, page in use and page is available, are self-explanatory. The last, page does not exist, indicates that the system is not currently accessing this page. This means that the system can still run even if part of physical memory has a problem or does not exist. Figure 4-10 shows the format of each MMAP entry.



Bits	Mnem	Name	Description
1-16	PAGE STATUS	Page Status	Contains a two's complement integer: n < 0: Page is not to be used (does not exist). n = 0: Page is available. n > 0: Page is in use; n is a pointer to the HMAP entry for that page.
17-32	——	Unused	Reserved for future use.

MMAP Entry Format
Figure 4-10

ACCESSING THE STLB AND CACHE

As described in Chapter 2, the STLB and the cache are high-speed buffers. If these buffers contain valid information for the process making a reference to a piece of data, the processor can access them in very little time instead of having to make a long memory access.

The hardware accesses both the STLB and the cache in parallel to speed up the reference. A slightly different set of actions is performed, depending on whether the operation is a read or a write. Refer to Figures 4-11 and 4-12 when reading the text in these sections.

Read Memory Access

As shown in Figure 4-11, the hardware performs three tasks in parallel: it references the STLB, references the cache, and validates the reference's access rights. The priority among these three tasks is also illustrated in the figure: the leftmost task (checking STLB entry) has a higher priority than the access check, and the access check has a higher priority than the cache entry stage. This means that if a problem arises in the STLB entry stage, that is solved first; then the whole access is retried from the beginning. The text in this section describes the access according to this priority.

Step 1. Accessing an STLB Entry

The hashing algorithm described above uses bits from the virtual address to choose an STLB entry. To make sure that this entry contains valid data, the hardware checks the entry's valid bit. If it contains 1, the entry is valid; 0, invalid. The hardware must also check that the process ID in the STLB entry is identical to that of the process making the reference. This is done only if the segment number specified in the virtual address is greater than or equal to '4000 — that is, if the segment specified is an unshared segment. If these conditions are met, the STLB entry contains valid data and can be used.

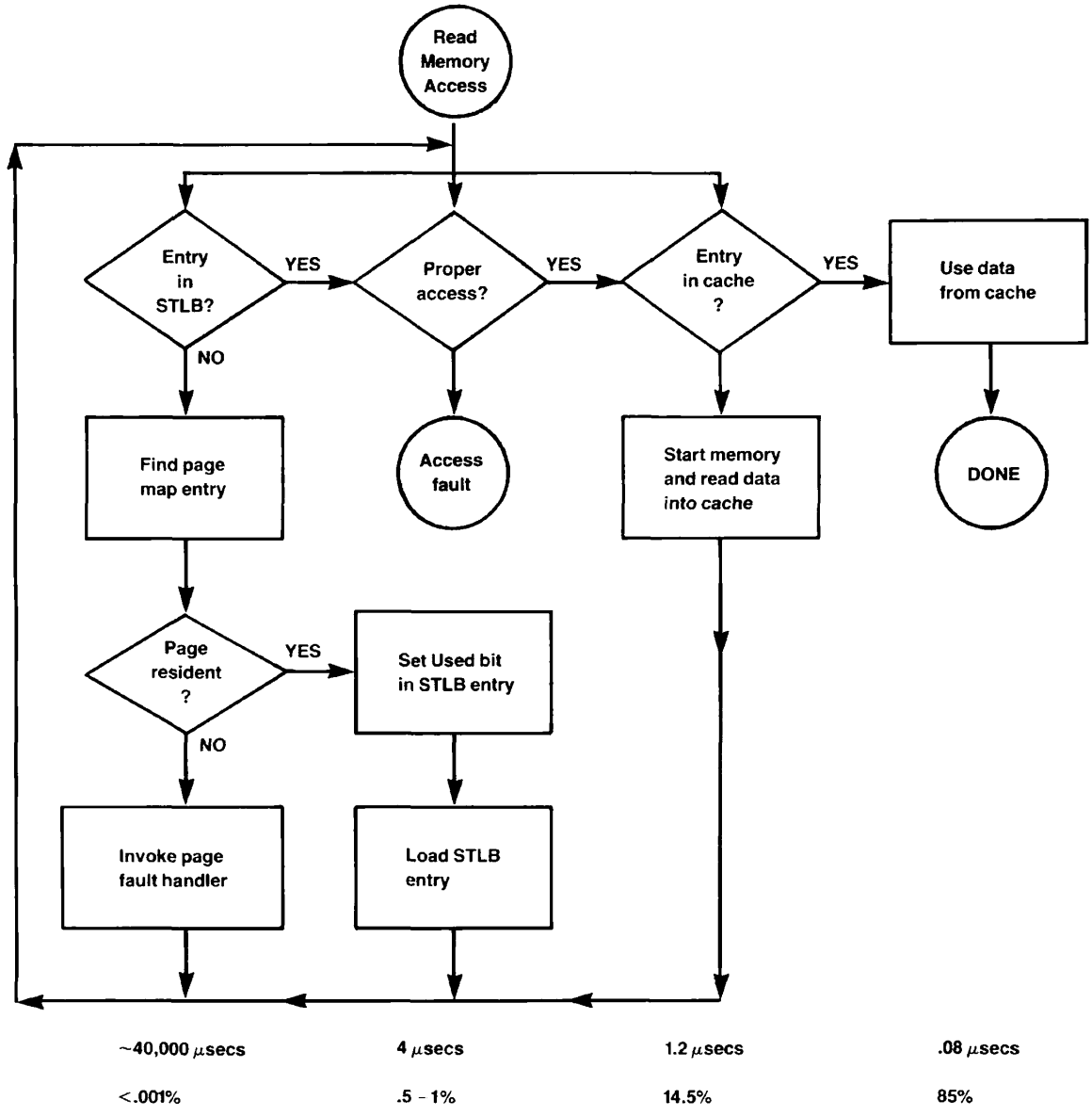
If the conditions are not met, the STLB needs to be loaded with the correct data. Therefore, the address translation microcode is invoked. (See Address Translation, below.) Assuming no page faults occur, the new translation is loaded into the STLB entry, and the used bit in that entry is set to 1. The reference is then retried from the beginning.

Step 2. Choosing an Access Field

If the STLB entry contains valid data, the hardware must determine what access rights should govern the reference. This requires two steps: first, isolating the ring number that specifies what access field to use; and second, using the access field contents to determine whether the reference is valid or not. Note that STLB entries for segment 0 have no ring field entry and can be accessed only by Ring 0.

To isolate the ring number, the processor weakens the ring number contained within the program counter by logically ORing it with the ring number contained in the effective address. This screens out all invalid references to lower-numbered rings (inward references), but allows references to higher-numbered rings (outward references) to be made.

This screening process makes sure that the access rights of the referencing procedure are weaker than those of the referenced procedure. If this were not done, then a Ring 3 procedure could call a



Read Memory Access
Figure 4-11

Ring 0 procedure, which in turn could call several procedures for which the Ring 3 procedure had no access rights. Screening out such references protects the integrity of the entire system.

Once the EA ring number has been weakened, the processor uses the weakened ring number to select an access field. If the ring number is 00, the hardware assumes that the reference has unlimited access and no further access checking is done. If the ring number is 01 or 11, the hardware uses the Ring 1 or Ring 3 access fields, respectively, in the STLB entry as the access field. If the ring number is 10, undefined results occur.

The access fields in the STLB entry specify the operations that references using this entry can legitimately perform. Table 4-3 lists the values these fields can contain and their meanings.

Table 4-3
Access Field Values and Their Meanings

Value	Description
000	No access
001	Gate (See Chapter 8.)
010	Read access
011	Read, write access
100	Reserved
101	Reserved
110	Read, execute access
111	Read, write, execute access

The hardware checks the operation specified in the instruction, making the reference against the selected access field to ensure that the operation is valid. For example, if the instruction specifies a read operation and the selected access field allows reads, then the read operation is valid. If, however, the instruction specifies a write and the access field allows only reads, then the operation is invalid. In the first case, the processor performs the valid operation and program execution continues. In the latter case, an access fault occurs and control transfers to the access fault handler. See Chapter 11 for more information about faults.

A reference must have read access to perform either a write or an execute operation. If an instruction specifies either a write or an execute and the access field does not allow reads, an access fault occurs.

Step 3. Accessing the Cache

If the access check is successful, the hardware references the cache. To do this, the hardware must form an address that references an entry in the cache index, which in turn specifies an entry in the cache data. The cache index address is formed in one of two ways, depending on the processor.

For the 2250, 250-II and earlier Prime systems, the hardware uses bits 23-32 in the virtual address as an address of an entry in the cache index. These bits are the 10-bit offset field.

For the I450 and the 550-II, the hardware uses bits 21-32 of the virtual address as an address of an entry in the cache index; the 750, 850, and 9950 use bits 20-32 of the virtual address. These are the least significant two or three bits of the page field and the 10-bit offset field. Note that the extra two or three bits create a virtually mapped cache. See Mapped I/O in Chapter 12 for information about how the MBIO bits in the IOTLB reconstruct this virtual mapping.

When the hardware has an address, it uses it to select an entry, j, in the cache index. Entry j contains a physical page address, which the hardware compares to the physical page address specified in the STLB entry. If the page numbers are the same, then the jth entry in the cache data area contains the contents of the desired physical location. These contents are used in the specified operation.

If the page numbers are not the same, the hardware must read the data in the specified physical location into the cache. It starts memory, reads the data into the cache, and then retries the access from the beginning.

Step 4. Timing Considerations

Figure 4-11 lists the time taken by each step of the read memory access. These figures are based on a 1 MIP machine. The figure also notes the percentage of times each step is likely to occur. As shown, the cache and STLB contain the needed information 85% of the time, and so the access requires only 80 nanoseconds. In addition, even though a page fault requires 40,000 microseconds it occurs very rarely (on the order of 10 per second). The other three steps occur the majority of the time, and give the system an average read memory access time of .24-.26 microseconds.

Write Memory Access

Figure 4-12 describes the general stages that occur in a write memory access. Note that the hardware references the STLB, validates the reference's access rights, and checks the STLB modified bit in parallel. However, the access validation takes precedence over

checking the modified bit, and the STLB entry access takes precedence over the access validation. This means that if problems occur in one of the stages with higher precedence, the problem is corrected and the access is retried from the beginning even if no problems occur with other stages.

Stage 1. Accessing the STLB

The hardware uses the hashing algorithm described above to select an STLB entry. The entry is validated in the same way as that described in the Read Memory Access section.

Stage 2. Checking the Access Rights

This stage is identical to that described in the Read Memory Access section above.

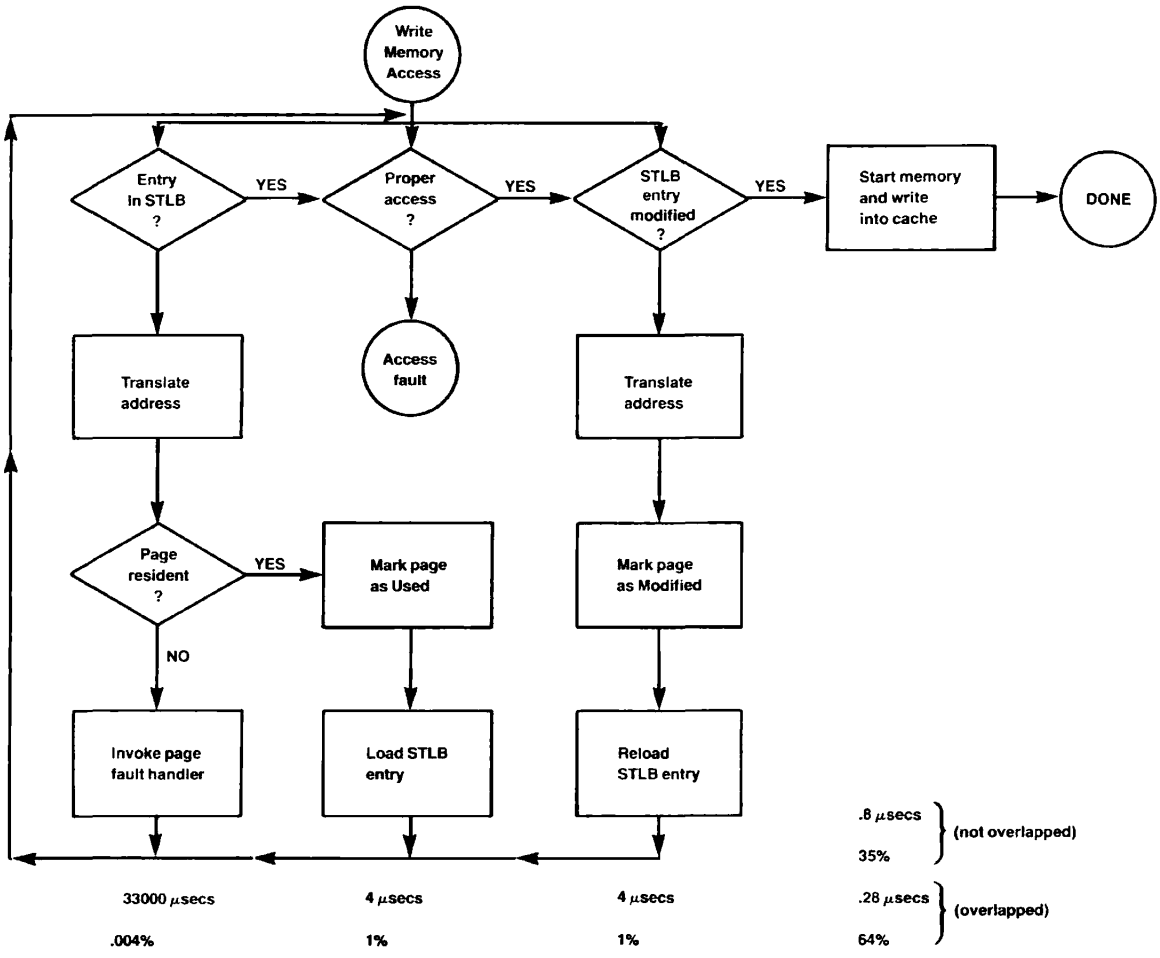
Stage 3. Checking the STLB Modified Bit

If the STLB entry is valid and if the reference has the proper access, the hardware checks the STLB entry's modified bit. If this bit contains 1, the page has been modified since this STLB entry was last used. This means that hardware must reload the STLB entry via the address translation mechanism. Once the new translation is loaded into the STLB entry, the reference is retried from the beginning.

If the STLB entry's modified bit is 0, then the entry contains valid data. The hardware forms the address of a cache entry (see Accessing the Cache, above), starts memory, and writes the contents of the referenced location into that cache entry.

Stage 4. Timing Considerations

Figure 4-12 lists the time each step of the write memory access takes. These figures are based on a 1 MIP machine. The figure also notes the percentage of times each step is likely to occur. As shown, the STLB contains the needed information 35-64% of the time, depending on whether the accesses are overlapped or not. In the case of overlapped transfers, the system's average write access time is about .22 microseconds; for transfers that are not overlapped, the average time is about .32 microseconds.



Write Memory Access
Figure 4-12

Address Translation

When the STLB does not contain information about the virtual-to-physical translation, a microcoded part of PRIMOS (called the address translation mechanism, or ATM) must perform the translation. The DTARs, the segment descriptor tables, and the hardware page map tables allow the ATM to make the correct reference.

When reading the detailed description of the translation process, refer to Figures 4-13a and 4-13b. Figure 4-13a depicts address translation on the 9950; Figure 4-13b, address translation on other 50 Series systems. The numbers labelling the discussion match the numbers on the diagram.

1. Interpreting the Virtual Address

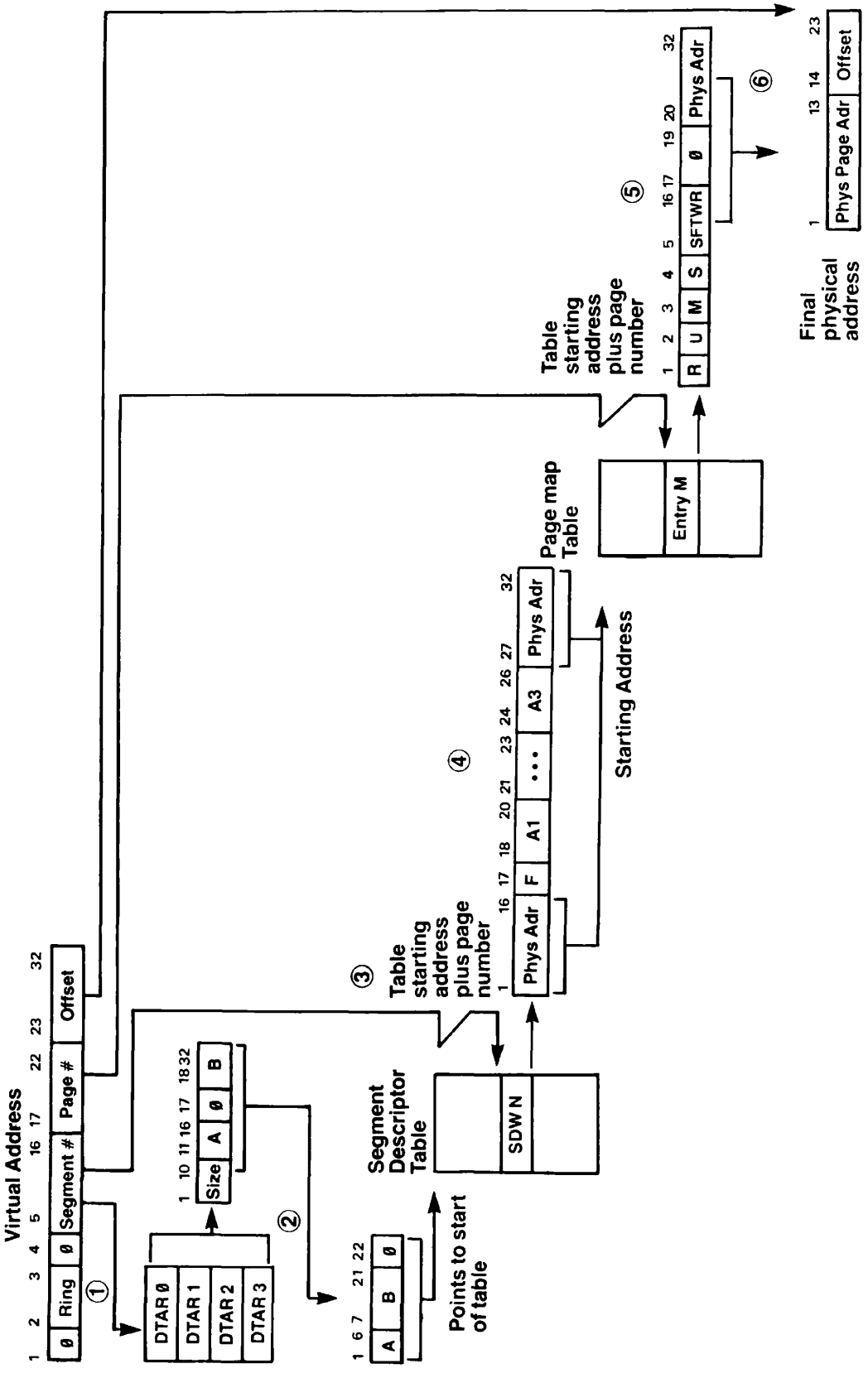
The virtual address derived from the information contained in an instruction is a 30-bit quantity. When the translation occurs, the virtual address is interpreted as shown in Figure 4-1. Bits 2-3 contain protection information and will be described in the next chapter. Bits 5-16 contain a segment number; bits 17-22, a page number; and bits 23-32, an offset. The ATM looks at bits 5-6 first, since they specify one of the four DTARs. The ATM references the specified DTAR.

2. Referencing the DTAR

The specified DTAR contains the address of a segment descriptor table, as well as the size of the table. The ATM uses the contents of bits 11-32 of the DTAR to form the starting address of the SDT.

3. Validating the Segment Number

After forming the table's starting address, the ATM uses bits 7-16 of the virtual address as an offset into the table. It first compares the segment number contained in these bits to bits 1-10 of the DTAR to check if the virtual address specifies an invalid segment. If the segment number is greater than the maximum allowable table size, the segment number is invalid and a segment fault occurs (segment number too large). If the segment number is less than or equal to the maximum allowable table size, the segment number is valid and the ATM adds the segment number to the starting address of the SDT. The sum specifies an entry, n, in the SDT.



Address Translation on the 9950
Figure 4-13a

4. Referencing the SDT

Entry n in the SDT contains a segment fault bit, access information (see next chapter), and the address of a hardware page map table (HMAP). The ATM checks bit 17, the fault bit, for an invalid segment. If F contains a 1, the segment is invalid or an HMAP is missing, and a segment fault occurs. If F contains a 0, the segment is valid and the ATM uses bits 1-16 and 27-32 of entry n as the starting address of an HMAP. The ATM adds the contents of bits 17-22 of the virtual address to the starting address in order to specify an entry (entry m) in the HMAP.

In the 9950, the ATM adds twice the value of bits 17-22 of the virtual address to reference the correct entry in a 640-element entry page map table.

5. Checking Page Status

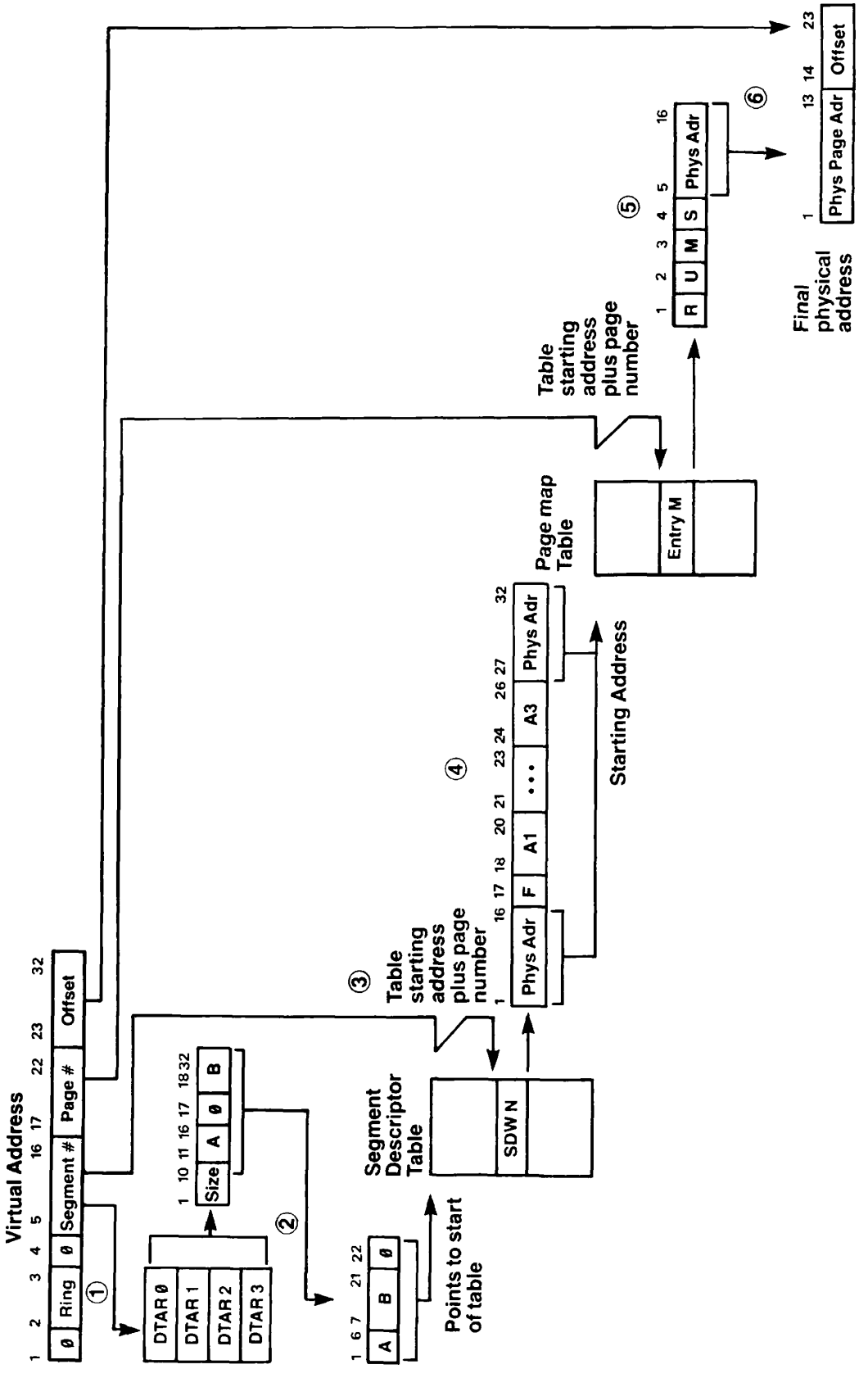
Bits 1-4 of entry m contain status information about a page of memory. When the entry is obtained from memory, the ATM examines the used (U) bit. If the content is 1, the page is assumed to be resident (R bit=1). If the U bit content is 0, the resident (R) bit is examined. If R contains 1, the page is resident but unused; the ATM sets the U bit in the PMT/HMAP entry and loads the translation into the STLB. If R contains 0, the page is not resident and a page fault occurs. (Chapter 11 contains more information about faults.) This ordering of the examination of the U and R bits maximizes the speed of the ATM.

Note

The combination of R=0 and U=1 is illegal and will cause undefined results.

6. Forming the Address and Loading the STLB

After determining that no page fault exists, the 9950 ATM combines the physical page address contained in bits 20-32 of the PMT entry with bits 23-32 of the virtual address to form a 23-bit physical address. The ATM for all other 50 Series systems combines the physical page address contained in bits 5-16 of the HMAP entry with bits 23-32 of the virtual address to form a 22-bit physical address. This is the final physical address. The ATM loads this address, plus its associated access information, into the STLB. The translation process for any address has to be done only the first time that location is referenced, because after that the STLB contains the translated value.



Address Translation on Other 50 Series Machines
Figure 4-13b

PAGING

When a requested page is not in main memory, a page fault occurs. Often, a page must be moved out of main memory and onto disk so that the new page can be loaded in. The software page fault handler uses three tables to move out a page, if necessary, and load in the requested page: the memory map table, and either the page map table (for 9950) or the hardware page map table and the logical page map table (for all other 50 Series systems).

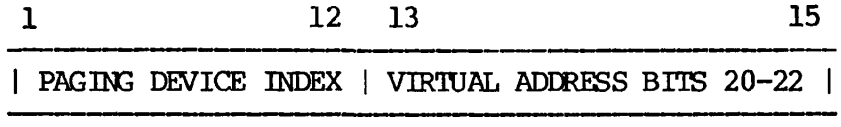
Refer to Figure 4-15 when reading the text in this section.

Step 1. Locating the Page on Disk

The first thing the page fault handler must do is locate the specified page on the disk. it uses the virtual address to reference an SDT entry (see Address Translation, above), which contains the starting address of a PMT/HMAP. This starting address and the contents of the virtual address page field allow the handler to reference an HMAP entry.

The HMAP specifies the starting address of an LMAP. The virtual page field contents are used as an offset into this table to reference an LMAP entry. Bits 5-16 of the LMAP entry specify a paging device index. This index points to a block of eight pages on the disk. (PMT bits 5-16 perform a similar function; they are reserved for software use.)

To choose one of the eight pages in this block, the handler uses the three least significant bits of the virtual address page field. The resulting address of the page on disk is shown in Figure 4-14.



Disk Page Address
Figure 4-14

Step 2. Allocating a Page in Physical Memory

Before the handler can load the page from disk into main memory, it must find a place to put it. It checks MMAP for an available page. If there are no available pages, it uses a first in, not used, first out algorithm to choose a page to move out of main memory. (With periodic polling, this algorithm gives an effective simulation of a first in, first out algorithm. It also provides many of the same benefits as a least recently used algorithm.)

Each PMT/HMAP entry contains a used bit (U) that specifies whether the page the entry identifies has been used since the last page fault occurred. If U contains 0, the page has not been used; if U contains 1, the page has been used. The page fault handler checks the used bits and identifies the first available page whose U is 0. This algorithm assures the handler that the page to be moved out of memory is not one currently being used by another process.

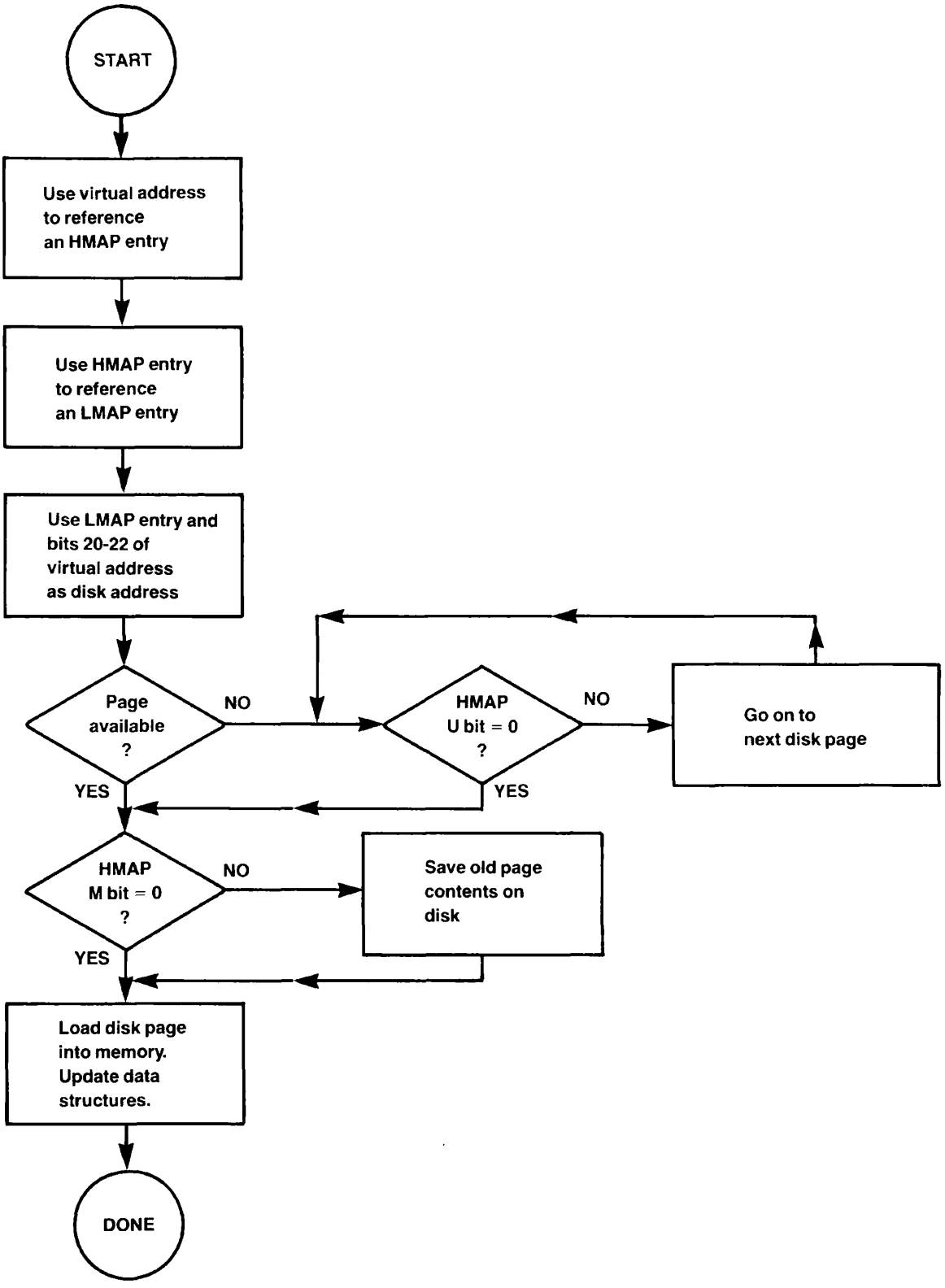
The handler can be configured to use the LRU algorithm to check for more than one currently available page. When this is done, the handler identifies several least recently used pages (default is 3) and prepares to move them out of main memory. This is called prepaging (also known as paging ahead or anticipatory clearance). It can speed up processor execution by paging out several pages during one page fault. When the next page fault occurs, the handler has only to load the disk page into main memory without first having to clear a space for the page.

Step 3. Saving the Old Page Contents

After the handler identifies available physical pages, it must choose a page to page out, and must determine whether or not it must store the old page contents on disk before loading in the new information. The PMT/HMAP entry aids in this task.

Bit 2 of the PMT/HMAP entry specifies whether this page has been used since this entry was last reset. If the page has not been used, then the handler can move it out without adversely affecting any running processes. If it has been used, the handler should locate another page to move out, since some process is likely to be using this page.

When the handler chooses a used page to page out, it checks bit 3 of the PMT/HMAP entry to determine if the page's contents have been modified since they were moved into main memory. If the old contents have not been modified (bit 3 contains 1), the handler can load in the new contents immediately. If the old contents have been modified (bit 3 contains 0), the handler must save a copy of them on disk before loading in the new data.



Paging
Figure 4-15

Like the LRU algorithm, the HMAP entries save the system processing time by limiting the number of disk accesses necessary to page in new information. By checking the entries periodically and tracking how they change, the handler determines the best page to swap out of main memory.

Step 4. Loading the Available Page

Once the handler has a physical page available, it loads a copy of the disk page into the physical page. After the copy, the handler updates the affected entries in the SDW, HMAP, LMAP, and MMAP.

SUMMARY

This chapter described how a 50 Series system uses a virtual address to locate information in physical memory. The cache and STLB provide rapid means of locating commonly referenced information without requiring memory access. When these buffers do not contain the desired information, PRIMOS can translate the user's virtual address into a physical one through the use of specialized data structures and algorithms. The software page fault handler ensures that information currently on disk is moved in a controlled fashion into main memory when it is needed.

5

Restricted Instructions and Control Information

The previous three chapters have described physical and virtual memory, how they are manipulated, and the data structures used in their manipulation. These data structures, like many parts of PRIMOS, are essential to system operation and so are protected against use by the casual user. However, a set of restricted instructions is available for situations that require manipulation of these and other system structures.

Restricted instructions can be executed in Ring 0, and many of them perform system functions, such as purging an STLB entry. Others manipulate some of the other system data structures, such as the keys register or the sense switches. This chapter describes some of these other data structures, especially the keys and modals, and lists the restricted instructions and describes what they do. For more detailed information about these instructions, refer to the appropriate entries in Chapters 13 and 14.

OTHER SYSTEM DATA STRUCTURES

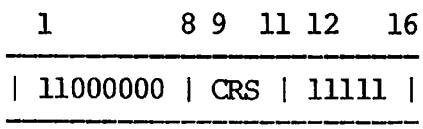
There are other data structures the system uses:

- Modals
- Keys
- CBIT, LINK, and condition code bits

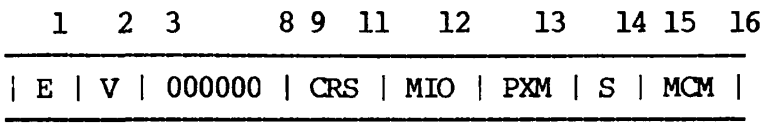
Modals

The 16-bit register called the modals contains information about the state of the processor. This register specifies information needed by the hardware and the operating system, such as the type of process control the system uses and which user register set is currently active. (See Chapter 10.) Note that this register is directly accessible only in V and I modes.

Figure 5-1 shows the normal setting of the modals that PRIMOS uses. Figure 5-2 shows the format of the modals. Table 5-1 lists the instructions that modify the modals.



Normal Modals Setting
Figure 5-1



Bits	Mnem	Description
1	E	Enable interrupts: 0 = interrupts disabled 1 = interrupts enabled
2	V	Vectored interrupt mode: 0 = standard interrupt mode 1 = vectored interrupt mode
3-8	——	Must be zero.
9-11	CRS	Specifies the current register set. Only the PXM can alter these bits. (See Chapter 8.)
12	MIO	Specifies the current mode of I/O: 0 = unmapped mode 1 = mapped mode
13	PXM	Process exchange enable/disable: 0 = process exchange disabled 1 = process exchange enabled
14	S	Specifies the mode of segmentation: 0 = no segmentation 1 = segmentation
15-16	MCM	Machine check mode: 00 = no reporting 01 = report only uncorrected memory parity errors 10 = report only unrecovered errors 11 = report all errors See Chapter 10 for more information.

Modals Format
(V and I Modes Only)

Figure 5-2

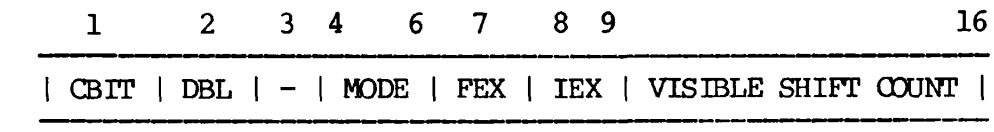
Table 5-1
Modals Instructions

Mnem	Name	Modes	Description
EMCM	Enter Machine Check Mode	S,R,V,I	Enters machine check mode.
ENB	Enable Interrupts	S,R,V,I	Sets bit 1 of the modals to 1.
ESIM	Enter Standard Interrupt Mode	S,R,V	Resets bit 2 of the modals to 0.
EVIM	Enter Vectored Interrupt Mode	S,R,V	Sets bit 2 of the modals to 1.
INH	Inhibit Interrupts	S,R,V,I	Resets bit 1 of the modals to 0.
LMCM	Leave Machine Check Mode	S,R,V,I	Leaves machine check mode.
LPSW	Load Program Status Word	V,I	Loads the PSW with the contents of a location in memory.
RMC	Reset Machine Check Flag to 0	S,R,V,I	Resets bits 15-16 of the modals to 0 and inhibits interrupts for the next instruction.

Keys

The other 16-bit register, the keys, describes the currently running process and the procedure that process is executing. The keys contain status information (such as the mode of addressing currently enabled) and specify fault handling information. Figure 5-3 shows the format of the keys for S and R modes; Figure 5-4 shows the format for V and I modes. Table 5-2 lists the instructions that modify the keys.

Never modify the keys or modals with the STLR instruction; use only the instructions listed in Tables 5-1 and 5-2. In addition, never use LPSW to change bits 15-16 of the keys or bits 9-11 of the modals. For more information, refer to individual instruction descriptions in Chapters 13 and 14.



Bits	Mnem	Description
1	CBIT	Reflects arithmetic conditions of some instructions.
2	DBL	Reflects arithmetic mode: 0 = single precision 1 = double precision
3	——	Reserved for future use.
4-6	MODE	Specifies the current mode of addressing: 000 = 16S 001 = 32S 010 = 64R 011 = 32R 100 = 32I 101 = unused 110 = 64V 111 = unused
7	FEX	Floating-point exception enable/disable: 0 = set CBIT to 1 and invoke fault handler on error 1 = set CBIT to 1 only on error
8	IEX	Integer exception enable/disable: 1 = set CBIT to 1 only on error 0 = set CBIT to 1 and invoke fault handler on error
9-16	VISIBLE SHIFT COUNT	Bottom half of the floating-point exponent.

Keys Format, S and R Modes
Figure 5-3

1	2	3	4	6	7	8	9	10	11	12	13	14	15	16
CBIT	0	LINK	MODE		FEX	IEX	LT	EQ	DEX	ASCII-8	RND	P850	IN	SD

Bits	Mnem	Description
1	CBIT	Reflects arithmetic conditions of some instructions.
2	——	Must be zero.
3	LINK	Reflects arithmetic conditions of some instructions.
4-6	MODE	Specifies the current mode of addressing: 000 = 16S 001 = 32S 010 = 64R 011 = 32R 100 = 32I 101 = unused 110 = 64V 111 = unused
7	FEX	Floating-point exception enable/disable: 0 = set CBIT to 1 and invoke fault handler on error 1 = set CBIT to 1 only on error
8	IEX	Integer exception enable/disable: 0 = set CBIT to 1 only on error 1 = set CBIT to 1 and invoke fault handler on error
9	LT	Less Than condition code: 1 reflects a less than 0 condition.
10	EQ	Equal To condition code: 1 reflects an equal to 0 condition.
11	DEX	Decimal exception enable/disable: 0 = set CBIT to 1 only on error 1 = set CBIT to 1 and invoke fault handler on error

Keys Format, V and I Modes
Figure 5-4

Bits	Mnem	Description
12	ASCII-8	ASCII character representation: specifies whether 7-bit or 8-bit ASCII characters are to be used. 0 = most significant bit of characters is 1 (8-bit format) 1 = most significant bit of characters is 0 (7-bit format) Used on 9950 only. Disregarded on other machines.
13	RND	Floating-point round: specifies the form of rounding to use in floating-point operations. 0 = no rounding 1 = rounding Used on 9950 only.
14	P850	P850 bit: used by the P850 processor.
15	IN	In dispatcher: specifies if the current process associated with the register is in the dispatcher. 0 = process is in the dispatcher 1 = process is not in the dispatcher Only the PXM alters this bit.
16	SD	Save done bit: specifies if PXM has saved values of current register set. 0 = save must be done before this register set can be used 1 = save has been done and this register set is available Only the PXM alters this bit.

Keys Format, V and I Modes
Figure 5-4 (continued)

Table 5-2
Keys Instructions

Mnem	Name	Modes	Description
DBL	Enter Double Precision Mode	S,R	Sets bit 2 in the keys to 1.
E16S	Enter 16S Mode	S,R,V,I	Sets bits 4-6 of the keys to 000.
E32I	Enter 32I Mode	S,R,V,I	Sets bits 4-6 of the keys to 100.
E32S	Enter 32S Mode	S,R,V,I	Sets bits 4-6 of the keys to 001.
E32R	Enter 32R Mode	S,R,V,I	Sets bits 4-6 of the keys to 011.
E64R	Enter 64R Mode	S,R,V,I	Sets bits 4-6 of the keys to 010.
E64V	Enter 64V Mode	S,R,V,I	Sets bits 4-6 of the keys to 110.
INK	Input Keys	S,R,I	Reads the keys into the specified register.
OTK	Output Keys Mode	S,R,I	Loads the keys with the contents of the specified register.
RCB	Reset CBIT	S,R,V,I	Sets the value of CBIT in the keys to 0.
SCA	Load Shift Count into A	S,R	Loads bits 9-16 of the keys into bits 9-16 of A.
SCB	Set CBIT	S,R,V,I	Sets the value of CBIT in the keys to 1.
SGL	Enter Single Precision Mode	S,R	Sets bit 2 in the keys to 0.
LPSW	Load PSW	V,I	Loads new data into the keys, modals, and program counter.
TAK	Transfer A to Keys	S,R,V	Transfers the contents of A into the keys.
TKA	Transfer Keys to A	S,R,V	Transfers the contents of the keys into A.

CBIT, LINK, and the Condition Codes

Some of the bits in the keys merit extra discussion. Bit 1, CBIT, and bit 2, LINK, are set by many instructions to indicate conditions under which the instruction completed execution. Several instructions performing arithmetic operations, for example, set CBIT to 1 to indicate that the operation has resulted in an overflow (result too large to fit in the specified number of bits). Others set LINK to 1 to reflect a carry out condition. Still others set CBIT to indicate a fault condition. The instruction entries in Chapters 13 and 14 state how each instruction affects the values of these bits.

Also note that bits 9-10 of the keys contain the condition codes. Many arithmetic, branch, skip, jump, and other instructions set these bits to indicate the result of a test (result is less than 0, for example), to indicate whether a value is positive or negative, and so on. Other instructions use the condition code values as Boolean values. The instruction entries in Chapters 13 and 14 also describe how an instruction affects the state of these bits.

EQ shows whether or not a 16- or 32-bit result is equal to 0. LT contains the extended sign for arithmetic and comparison operations. The extended sign is the sign of the result as if the operation had been done on a machine of infinite precision; thus, LT shows the correct sign of the result despite any overflow. For logic operations, LT reflects the sign of the result. Table 5-3 shows condition code interpretation for comparison, arithmetic, and logic operations.

Table 5-3
Interpretation of Condition Codes

LT, EQ Values	Comparison	Arithmetic	Logic
00	Register > 0 Register > EA Reg 1 > Reg 2	Signed result > 0 Unsigned result <> 0	Result <> 0, High-order bit = 0
01	Register = 0 Register = EA Reg 1 = Reg 2	Result = 0	Result = 0, High-order bit = 0
10	Register < 0 Register < EA Reg 1 < Reg 2	Result = 0	Result <> 0, High-order bit = 1
11	Not working	Possible if largest negative number is added to itself. (CBIT is set to 1 as well, to indicate overflow.)	Not working

RESTRICTED INSTRUCTIONS

Table 5-3 lists the restricted instructions and briefly describes their actions. Refer to Chapters 13 and 14 for more information about these instructions.

Table 5-3
Restricted Instructions

Mnem	Name	Modes	Description
CAI	Clear Active Interrupt	S,R,V,I	Clears the currently active interrupt.
EIO	Execute I/O	V,I	Executes an effective address as an I/O instruction.
EMCM	Enter Machine Check Mode	S,R,V,I	Enters machine check mode.
ENB	Enable Interrupts	S,R,V,I	Enables interrupts so that devices can request service.
ESIM	Enter Standard Interrupt Mode	S,R,V,I	All interrupts use location '63 to reach the interrupt handler.
EVIM	Enter Vectored Interrupt Mode	S,R,V,I	Services interrupts according to their priority on the I/O bus.
HLT	Halt	S,R,V,I	Halts the processor.
INA	Input to A	S,R	Loads data from the specified device into A.
INBC	Interrupt Notify	V,I	Notifies during the interrupt code. Uses LIFO queuing. Clears the currently active interrupt.
INBN	Interrupt Notify	V,I	Notifies during the interrupt code. Uses LIFO queuing. Does not clear the currently active interrupt.
INEC	Interrupt Notify	V,I	Notifies during the interrupt code. Uses FIFO queuing. Clears the currently active interrupt.
INEN	Interrupt Notify	V,I	Notifies during the interrupt code. Uses FIFO queuing. Does not clear the currently active interrupt.
INH	Inhibit Interrupts	S,R,V,I	Disables interrupts so that devices cannot request service.
IRTC	Interrupt Return	V,I	Returns control from an interrupt and clears the currently active interrupt.
IRTN	Interrupt Return	V,I	Returns control from an interrupt and does not clear the currently active interrupt.

Table 5-3 (continued)
Restricted Instructions

Mnem	Name	Modes	Description
ITLB	Invalidate STLB Entry	V,I	Invalidates the STLB entry specified by L.
LIOT	Load I/O TLB	V,I	Loads an entry in the IOTLB.
LMCM	Leave Machine Check Mode	S,R,V,I	Leaves machine check mode.
LPID	Load Process ID	V,I	Loads the process ID contained in A into RPID.
LPSW	Load PSW	V,I	Loads new values into the program counter, keys, and modals.
MDRS	Memory Diagnostic Read Syndrome Bits Pulse	V,I	Reads the memory syndrome bits.
MDWC	Memory Diagnostic Write Control Register	V,I	Writes the control register.
NFYE	Notify End of Queue	V,I	Notifies on the specified semaphore. Uses LIFO queuing. Does not clear the currently active interrupt.
NFYB	Notify Head of Queue	V,I	Notifies on the specified semaphore. Uses FIFO queuing. Does not clear the currently active interrupt.
OCP	Output Control	S,R	Sends a control pulse to a device.
OTA	Output from A	S,R	Transfers data from A to the specified device.
PTLB	Purge TLB	V,I	Purges either an entry or a page in the translation lookaside buffer.

Table 5-3 (continued)
Restricted Instructions

Mnem	Name	Modes	Description
RMC	Clear Machine Check	S,R	Clears the machine check flag.
RTS	Reset Time Slice	V,I	Resets the value of the interval timer.
SKS	Skip on Satisfied Condition	S,R	When the specified condition is satisfied, the specified device responds ready and the instruction skips the next word.
SNR	Skip on Sense Switch Reset	S,R,V	Skips the next word if the specified sense switch is off.
SNS	Skip on Sense Switch Set	S,R,V	Skips the next word if the specified sense switch is on.
SR1, SR2, SR3, SR4	Skip on Sense Switch Reset	S,R	Skips the next word if the specified sense switch is off.
SS1, SS2, SS3, SS4	Skip on Sense Switch Set	S,R	Skips the next word if the specified sense switch is on.
SSR	Skip on Any Sense Switch Reset	S,R,V	Skips the next word if any of the sense switches are off.
SSS	Skip on Any Sense Switch Set	S,R,V	Skips the next word if any of the sense switches are on.
STPM	Store Processor Model Number	V,I	Stores the CPU model number and microcode revision number into memory.
VIRY	Verify	S,R,V,I	Executes the verify routine.
WAIT	Wait	V,I	Waits until the specified semaphore is notified.

SUMMARY

This chapter described more of the system registers and data structures that aid in controlling system operation. The next chapter, Datatypes, presents the data representations and formats supported on the 50 Series processors. It also lists the instructions you can use to manipulate the various types of data.

6
Datatypes

The 50 Series systems support several data representations. These representations fall into the major groups:

- Fixed-point data
- Floating-point numbers
- Decimal integers
- Character strings
- Queues

This chapter describes each of these data representations, and the operations and instructions available to manipulate each type.

Throughout the rest of this book, R is used to indicate a 32-bit I mode general register, while r indicates bits 1-16 of a 32-bit I mode general register. In addition, A and B represent S and R mode 16-bit registers; L and E represent V mode 32-bit registers.

FIXED-POINT DATA

Fixed-point data can be a logical value, a signed or unsigned integer, or an address. Addresses are treated as unsigned integers.

Logical Values

A logical value is a 16- or 32-bit value that is interpreted as a string of bits. Table 6-1 lists the instructions that perform logical operations, such as OR and AND. Note that the 50 Series processors treats each bit in a bit string separately: the value of one bit does not affect the value of another.

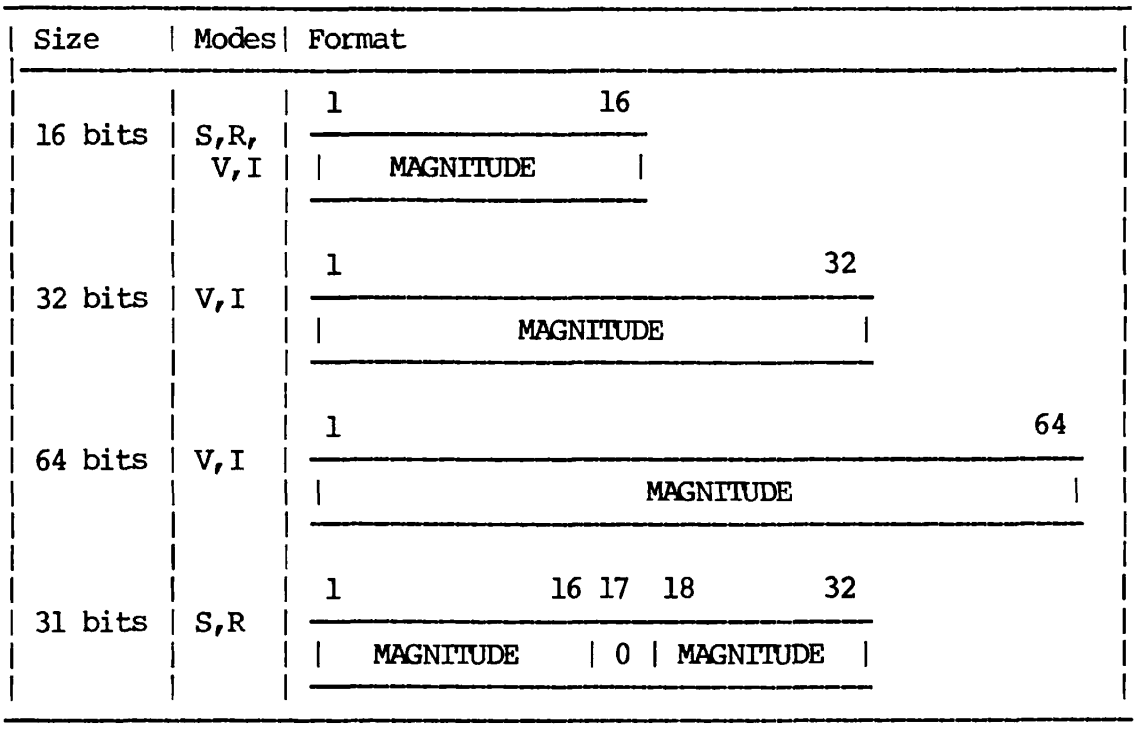
There are several instructions available that test logical values and perform an action depending on the result of the test. Chapter 7 discusses these instructions.

Table 6-1
Logic Instructions

Mnem	Name	Modes	Description
ANA	AND to A	S,R,V	Logically ANDs the contents of A and the contents of a memory location.
ANL	AND Long	V	Logically ANDs the contents of L and the contents of a memory location.
CMA	Complement A	S,R,V	Forms the one's complement of the contents of A.
CMH	Complement Halfword	I	Forms the one's complement of the contents of r.
CMR	Complement Fullword	I	Forms the one's complement of the contents of R.
ERA	Exclusive OR to A	S,R,V	Exclusively ORs the contents of A and the contents of a memory location.
ERL	Exclusive OR Long	V	Exclusively ORs the contents of L and the contents of a memory location.
N	AND Fullword	I	Logically ANDs the contents of R and the contents of a memory location.
NH	AND Halfword	I	Logically ANDs the contents of r and the contents of a memory location.
O	OR Fullword	I	Logically ORs the contents of R and the contents of a memory location.
OH	OR Halfword	I	Logically ORs the contents of r and the contents of a memory location.
ORA	Inclusive OR to A	V	Logically ORs the contents of A and the contents of a memory location.
X	Exclusive OR Fullword	I	Exclusively ORs the contents of R and the contents of a memory location.
XH	Exclusive OR Halfword	I	Exclusively ORs the contents of r and the contents of a memory location.

Signed Integers

Depending on the addressing mode, there are a variety of signed integer formats to use. Each is based on a magnitude field that represents a two's complement value. Figure 6-1 shows the formats and data sizes available for each addressing mode.



Signed Integer Formats
Figure 6-1

Unsigned Integers

Unsigned integers can be 16, 32, or 64 bits long. Regardless of length or addressing mode, all of the bits in the unsigned integer represent the magnitude of the number.

Most operations work for both signed and unsigned numbers. Special unsigned support is provided only for those magnitude branch instructions that allow results to be evaluated as unsigned integers. Multiply and divide instructions do not work correctly for unsigned integers.

Table 6-2 lists the instructions that operate on signed and unsigned integers.

Table 6-2
Integer Arithmetic Instructions

Mnem	Name	Modes	Description
A	Add Fullword	I	Adds the 32-bit contents of a memory location to the contents of R.
AlA	Add 1 to A	S,R,V	Adds one to the contents of A.
A2A	Add 2 to A	S,R,V	Adds two to the contents of A.
ACA	Add CBIT to A	S,R,V	Adds the value of CBIT to the contents of A.
ADD	Add	S,R,V	Adds the contents of a 16-bit memory location to the 16-bit contents of A.
ADL	Add Long	V	Adds the 32-bit contents of a memory location to the 32-bit contents of L.
ADLL	Add LINK to L	V	Adds the value of LINK to the contents of L.
ADLR	Add LINK to R	I	Adds the value of LINK to the contents of R.
AH	Add Halfword	I	Adds the 16-bit contents of a memory location to the contents of r.
C	Compare Fullword	I	Compares the contents of R to the contents of a memory location and sets the condition codes to reflect the result of the compare.
CH	Compare Halfword	I	Compares the contents of r to the contents of a memory location and sets the condition codes to reflect the result of the compare.
CHS	Change Sign	I	Complements bit 1 of R.
CHS	Change Sign	S,R,V	Complements bit 1 of A.
CSA	Copy Sign of A	S,R,V	Sets CBIT to the value of bit 1 in A, then sets bit 1 of A to 0.
CSR	Copy Sign		Copies bit 1 of R into CBIT and resets bit 1 of R to 0.
D	Divide Fullword	I	Divides the 64-bit contents of R and R+1 by the 32-bit contents of of a memory location.
DAD	Double Add	S,R	Adds the 31-bit contents of a memory location to the 31-bit contents of A and B.
DH	Divide Halfword	I	Divides the 32-bit contents of R by the 16-bit contents of a memory location.
DH1	Decrement r by 1	I	Decrements r by 1 and stores the results in r.
DH2	Decrement r by 2	I	Decrements r by 2 and stores the results in r.

Table 6-2 (continued)
Integer Arithmetic Instructions

Mnem	Name	Modes	Description
DIV	Divide	S,R	Divides the 31-bit contents of A and B by the 16-bit contents of a memory location.
DIV	Divide	V	Divides the 32-bit contents of L by the 16-bit contents of a memory location.
DM	Decrement Memory Fullword	I	Decrements the contents of the specified memory location by 1.
DMH	Decrement Memory Halfword	I	Decrements the contents of the specified memory location by 1.
DR1	Decrement R by 1	I	Decrements R by 1 and stores the result in r.
DR2	Decrement R by 2	I	Decrements R by 2 and stores the result in r.
DSB	Double Subtract	S,R	Subtracts the 31-bit contents of a memory location from the 31-bit contents of A and B.
DVL	Divide Long	V	Divides the 64-bit contents of E and L by the 32-bit contents of a memory location.
IH1	Increment r by 1	I	Increments r by 1 and stores the result in r.
IH2	Increment r by 2	I	Increments r by 2 and stores the result in r.
IM	Increment Memory Fullword	I	Increments the contents of the specified memory location by 1.
IMH	Increment Memory Halfword	I	Increments the contents of the specified memory location by 1.
IR1	Increment R by 1	I	Increments R by 1 and stores the result in R.
IR2	Increment R by 2	I	Increments R by 2 and stores the result in R.
M	Multiply Fullword	I	Multiplies the 32-bit contents of R by the 32-bit contents of a memory location to get a 64-bit result.
MH	Multiply Halfword	I	Multiplies the 16-bit contents of r by the 16-bit contents of a memory location to get a 32-bit result.

Table 6-2 (continued)
Integer Arithmetic Instructions

Mnem	Name	Modes	Description
MPL	Multiply Long	V	Multiplies the 32-bit contents of L by the 32-bit contents of a memory location to get a 64-bit result.
MPY	Multiply	S,R	Multiplies the 16-bit contents of A by the 16-bit contents of a memory location to get a 31-bit result.
MPY	Multiply	V	Multiplies the 16-bit contents of A by the 16-bit contents of a memory location to get a 32-bit result.
MPY	Multiply	I	Multiplies the 16-bit contents of r by the 16-bit contents of a memory location to get a 32-bit result.
NRM	Normalize	S,R	Normalizes the contents of A and B.
PID	Position for Integer Divide	S,R	Converts the 16-bit integer in A to to a 31-bit integer in A and B.
PID	Position for Integer Divide	I	Converts the 32-bit integer in R to to a 64-bit integer in R and R+1.
PIDA	Position for Integer Divide	V	Converts the 16-bit integer in A to to a 31-bit integer in L.
PIDH	Position for Integer Divide	I	Converts the 16-bit integer in r to a 32-bit integer in R.
PIDL	Position for Integer Divide	V	Converts the 32-bit integer in L to to a 64-bit integer in L and E.
PIM	Position After Integer Multiply	S,R	Converts the 31-bit integer in A and B to a 16-bit integer in A.
PIM	Position After Integer Multiply	I	Converts the 64-bit integer in R and R+1 to a 32-bit integer in R.

Table 6-2 (continued)
Integer Arithmetic Instructions

Mnem	Name	Modes	Description
PIMA	Position After Integer Multiply	V	Converts the 32-bit integer in L to a 16-bit integer in A.
PIMH	Position for Integer Multiply	I	Converts the 32-bit integer in R to a 16-bit integer in r.
PIML	Position After Integer Multiply Long	V	Converts the 64-bit integer in L and E to a 32-bit integer in L.
S1A	Subtract 1 From A	S,R,V	Subtracts 1 from the contents of A.
S2A	Subtract 2 From A	S,R,V	Subtracts 2 from the contents of A.
S	Subtract Fullword	I	Subtracts the 32-bit contents of a memory location from the 32-bit contents of R.
SBL	Subtract Long	V	Subtracts the 32-bit contents of a memory location from the 32-bit contents of L.
SH	Subtract Halfword	I	Subtracts the 16-bit contents of a memory location from the 16-bit contents of r.
SSM	Set Sign Minus	S,R,V	Sets bit 1 of A to 1.
SSM	Set Sign Minus	I	Sets bit 1 of R to 1.
SSP	Set Sign Plus	S,R,V	Sets bit 1 of A to 0.
SSP	Set Sign Plus	I	Sets bit 1 of R to 0.
SUB	Subtract	S,R,V	Subtracts the 16-bit contents of a memory location from the 16-bit contents of A.
TCA	Two's Complement A	S,R,V	Forms the two's complement of the contents of A.
TCL	Two's Complement L	V	Forms the two's complement of the contents of L.

Table 6-2 (continued)
Integer Arithmetic Instructions

Mnem	Name	Modes	Description
TC	Two's Complement R	I	Forms the two's complement of the contents of R.
TCH	Two's Complement r	I	Forms the two's complement of the contents of r.
TM	Test Memory Fullword	I	Tests the contents of a memory location and sets the condition codes to reflect the result of the test.
TMH	Test Memory Halfword	I	Tests the contents of a memory location and sets the condition codes to reflect the result of the test.

Addresses

The 50 Series processors manipulate addresses as if they were unsigned integers. Table 6-3 lists the instructions that handle addresses.

Table 6-3
Address Manipulation Instructions

Mnem	Name	Modes	Description
EAFA	EA to FAR	V,I	Calculates an effective address and loads it into the specified FAR.
FLX, DFLX QFLX	Load Floating Index	R,V	Loads X with a multiple of the contents of a memory location.
CEA	Compute EA	S,R	Uses the contents of A as an indirect address, calculates an effective address from the referenced location and loads the EA into A.
EAA	Effective Address to A	S,R,V	Loads an effective address into A.
EAL	Effective Address to L	S,R,V	Loads an effective address into L.
EALB	Effective Address to LB	V,I	Loads an effective address into LB.
EAR	Effective Address to R	I	Loads an effective address into R.
EAXB	Effective Address to XB	V,I	Loads an effective address into XB.

Fixed-point Operations

The 50 Series processors can perform several kinds of operations on fixed-point data. Some examples are setting or resetting a single bit in a logical value, or storing an unsigned integer into a memory location. Table 6-4 lists the instructions that move fixed-point data from one place to another. Table 6-5 describes a group of load/store instructions. Table 6-6 lists the instructions that shift the contents of a 16- or 32-bit register. Table 6-7 shows instructions that can be used to set or reset all or part of a piece of data.

Table 6-4
Data Movement Instructions

Mnem	Name	Modes	Description
DLD	Double Load	S,R	Loads A and B with the contents of two 16-bit memory locations.
DST	Double Store	S,R	Stores the contents of A and B into two 16-bit memory locations.
I	Interchange R and Memory Fullword	I	Interchanges the contents of R and a memory location.
IAB	Interchange A and B	S,R,V	Interchanges the values of A and B.
ICA	Interchange Characters in A	S,R,V	Interchanges the contents of the two bytes in A.
ICBL	Interchange and Clear Left	I	Interchanges the contents of the bytes in r, then loads zeroes into the leftmost byte of r.
ICBR	Interchange and Clear Right	I	Interchanges the contents of the bytes in r, then loads zeroes into the rightmost byte of r.
ICHL	Interchange Halfwords and Clear Left	I	Interchanges the contents of bits 1-16 and 17-31 of R, then load s bits 1-16 of R with zeroes.
ICHR	Interchange Halfwords and Clear Right	I	Interchanges the contents of bits 1-16 and 17-31 of R, then loads bits 17-31 of R with zeroes.
ICL	Interchange and Clear Left	S,R,V	Interchanges the contents of the bytes in A, then loads zeroes into the leftmost byte of A.
ICR	Interchange and Clear Right	S,R,V	Interchanges the contents of the bytes in A, then loads zeroes into the rightmost byte of A.

Table 6-4 (continued)
Data Movement Instructions

Mnem	Name	Modes	Description
IH	Interchange r and Memory	I	Interchanges the contents of r and a memory location.
ILE	Interchange E and L Halfword	V	Interchanges the contents of E and L.
IMA	Interchange A and Memory	S,R,V	Interchanges the contents of A and a memory location.
IRB	Interchange Register Bytes	I	Interchanges the contents of bits 1-8 and 9-16 of r.
IRH	Interchange Register Halves	I	Interchanges the contents of bits 1-16 and 17-32 of R.
L	Load Fullword	I	Loads the contents of a memory location into R.
LDA	Load A	S,R,V	Loads the contents of a memory location into A.
LDL	Load long	V	Loads the contents of a memory location into L.
LDX	Load X	S,R,V	Loads the contents of a memory location into X.
LDY	Load Y	V	Loads the contents of a memory location into Y.
LH	Load Halfword	I	Loads the contents of a memory location into r.
LHL1	Load Halfword Left Shifted By 1	I	Shifts the contents of a memory location left one bit and loads the result into r.
LHL2	Load Halfword Left Shifted By 2	I	Shifts the contents of a memory location left two bits and loads the result into r.
LHL3	Load Halfword Left Shifted By 3	I	Shifts the contents of a memory location left three bits and loads the result into r.
ST	Store Fullword	I	Stores the contents of R into a memory location.
STA	Store A	S,R,V	Stores the contents of A into memory.

Table 6-4 (continued)
Data Movement Instructions

Mnem	Name	Modes	Description
STAC	Store A Conditionally	V	Stores the contents of A into memory if the contents of the specified memory location equal the contents of B.
STCD	Store Conditional Fullword	I	Stores the contents of R into the location specified by EA if the contents of R+1 equal the contents of the location specified by EA.
STCH	Store Conditional Halfword	I	Stores the contents of r into the location specified by EA if the contents of bits 17-32 equal the contents of the location specified by EA.
STH	Store Halfword	I	Stores the contents of r into a memory location.
STL	Store long	V	Stores the contents of L into memory.
STLC	Store L Conditionally	V	Stores the contents of L into memory if the contents of the specified memory location equal the contents of E.
STX	Store X	S,R,V	Stores the contents of X into memory.
STY	Store Y	V	Stores the contents of Y into memory.
TAB	Transfer A to B	V	Transfers the contents of A into B.
TAX	Transfer A to X	V	Transfers the contents of A into X.
TAY	Transfer A to Y	V	Transfers the contents of A into Y.
TBA	Transfer B to A	V	Transfers the contents of B into A.
TXA	Transfer X to A	V	Transfers the contents of X into A.
TYA	Transfer Y to A	V	Transfers the contents of Y into A.
XCA	Exchange and Clear A	S,R,V	Exchanges the contents of A and B, then loads zeroes into A.
XCB	Exchange and Clear B	S,R,V	Exchanges the contents of B and A, then loads zeroes into B.

Table 6-5
Special Load/Store Instructions

Mnem	Name	Modes	Description
RSAV	Save Registers	V,I	Saves the contents of the general, floating, temporary, and base registers in a block of consecutive memory locations.
RRST	Restore Registers	V,I	Restores the values of the general, floating, temporary, and base registers with information contained in a block of consecutive memory locations.
LDAR	Load Addressed Register	V,I	Loads the contents of a register file location into R.
LDLR	Load L from Register File	V	Loads the contents of a register file location into L.
STAC	Store A Conditionally	V	Stores the contents of A at the specified address if the contents of the specified address are equal to the contents of B.
STAR	Store Addressed Register	I	Stores the contents of the specified R in a register file location.
STLC	Store L Conditionally	V	Stores the contents of L into the specified address if the contents of the specified address are equal to the contents of E.
STLR	Store L Into Register File	V	Loads the contents of L into a register file location.

Table 6-6
Shift Instructions

Mnem	Name	Modes	Description
ALL	A Left Logical	S,R,V	Shifts the contents of A left a specified number of bits.
ALR	A Left Rotate	S,R,V	Shifts the contents of A left a specified number of bits, rotating bit 1 into bit 16.
ALS	A Left Shift	S,R,V	Shifts the contents of A left a specified number of bits.
ARL	A Right Logical	S,R,V	Shifts the contents of A right a specified number of bits.
ARR	A Right Rotate	S,R,V	Shifts the contents of A right a specified number of bits, rotating bit 16 into bit 1.
ARS	A Right Shift	S,R,V	Shifts the contents of A right a specified number of bits.
LLL	L Left Logical	S,R,V	Shifts the contents of L left a specified number of bits.
LLR	L Left Rotate	S,R,V	Shifts the contents of L left a specified number of bits, rotating bit 1 into bit 16.
LLS	L Left Shift	S,R	Shifts the contents of A and B left a specified number of bits, bypassing bit 1 of B.
LLS	L Left Shift	V	Shifts the contents of L left a specified number of bits.
LRL	L Right Logical	S,R,V	Shifts the contents of L right a specified number of bits.
LRR	L Right Rotate	S,R,V	Shifts the contents of L right a specified number of bits, rotating bit 16 into bit 1.
LRS	L Right Shift	V	Shifts the contents of L right a specified number of bits.
LRS	L Right Shift	S,R	Shifts the contents of A and B right a specified number of bits, bypassing bit 1 of B.
ROT	Rotate	I	Rotates the contents of R a specified number of bits in a specified direction.
SHA	Arithmetic Shift	I	Shifts the contents of R a specified number of bits in a specified direction.
SHL	Logical Shift	I	Shifts the contents of R a specified number of bits in a specified direction.
SL1	Shift R Left 1	I	Shifts the contents of R left one bit.
SL2	Shift R Left 2	I	Shifts the contents of R left two bits.

Table 6-6 (continued)
Shift Instructions

Mnem	Name	Modes	Description
SR1	Shift R Right 1	I	Shifts the contents of R right one bit.
SR2	Shift R Right 2	I	Shifts the contents of R right two bits.
SHL1	Shift r Left 1	I	Shifts the contents of r left one bit.
SHL2	Shift r Left 2	I	Shifts the contents of r left two bits.
SHR1	Shift r Right 1	I	Shifts the contents of r right one bit.
SHR2	Shift r Right 2	I	Shifts the contents of r right two bits.

Note to Table 6-6

The instructions in Table 6-6 specify three types of shift operations. An instruction that performs a logical shift treats the data to be shifted as a logical string of bits, shifting zeroes into the vacated bits. The carry reflects the state of the last bit shifted out.

An instruction performing an arithmetic shift treats the data as a signed number. For a right arithmetic shift, the instruction shifts in copies of the sign bit into the vacated bits; CBIT reflects the state of the last bit shifted out. For a left arithmetic shift, the instruction shifts zeroes into the vacated bits. If there is a sign change in bit 1 (interpreted as an overflow condition), an integer exception occurs. (See Chapter 11.)

An instruction that performs a rotate shifts bits out of one side of the data word and loads them into vacated bits on the other side.

Table 6-7
Clear Register/Memory Instructions

Mnem	Name	Modes	Description
CAL	Clear A Left Byte	S,R,V	Sets bits 1-8 of A to 0.
CAR	Clear A Right Byte	S,R,V	Sets bits 9-16 of A to 0.
CR	Clear Register	I	Sets the specified register to 0.
CRA	Clear A	S,R,V	Resets the contents of A to 0.
CRB	Clear B	S,R,V	Resets the contents of B to 0.
CRBL	Clear High Byte 1 Left	I	Sets bits 1-8 of the specified register to 0.
CRBR	Clear High Byte 1 Right	I	Sets bits 9-16 of the specified register to 0.
CRE	Clear E	V	Resets the contents of E to 0.
CRHL	Clear Left Halfword	I	Sets bits 1-16 of the specified register to 0.
CRHR	Clear Right Halfword	I	Sets bits 17-32 of the specified register to 0.
CRL	Clear L	S,R,V	Resets the contents of L to 0.
CRLE	Clear L and E	V	Resets the contents of L and E to 0.
ZM	Zero Memory Fullword	I	Resets the 32-bit contents of the specified memory location to 0.
ZMH	Zero Memory Halfword	I	Resets the 16-bit contents of the specified memory location to 0.

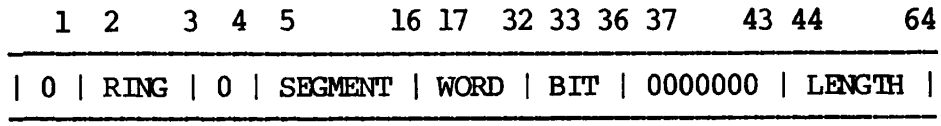
Field Operations

The 50 Series processors support a group of instructions that perform field operations. These instructions use the field address and length registers in their manipulations. These registers are abbreviated as FAR, for field address register, or FLR, for field length register; but both are specified in the same 64-bit register shown in Figure 6-2.

Note that the field address and length registers overlap the floating accumulators. The precise overlap varies from one Prime machine to another, as shown in Figures 6-2 and 6-3. Table 6-8 lists the field operation instructions.

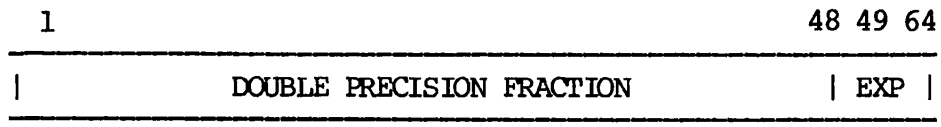
Table 6-8
Field Operation Instructions

Mnem	Name	Modes	Description
ALFA	Add Long to FAR	V	Adds the contents of L to the contents of the specified FAR.
ARFA	Add R to FAR	I	Adds the contents of the specified R to the contents of the specified FAR.
EAFA	EA to FAR	V,I	Calculates an effective address and loads it into the specified FAR.
LDC	Load Character	V,I	Calculates an effective address. Loads the character in the specified field into the addressed location.
LFLI	Load Immediate to FLR	V,I	Loads an immediate value into the specified FLR.
STFA	Store FAR	V,I	Calculates an effective address and stores the contents of the specified FAR into the addressed location.
STC	Store Character into Field	V,I	Stores the contents of a register into the specified field.
TFLL	Transfer Long from FLR	V	Transfers the contents of the specified FLR to L.
TLFL	Transfer Long to FLR	V	Transfers the contents of L into the specified FLR.
TFLR	Transfer FLR to R	I	Transfers the contents of the specified FLR to the specified R.
TRFL	Transfer R to FLR	I	Transfers the contents of the specified R to the specified FLR.



Bits	Mnem	Description
2-3	RING	Specifies the ring number of the field address.
5-16	SEGMENT	Specifies the segment number of the field address.
17-32	WORD	Specifies the word number of the field address.
33-36	BIT	Specifies the bit number of the field address.
37-43	---	Must be 0.
44-64	LENGTH	Specifies 21 bits of field length.

Format of Field Address and Length Register (FAR, FLR)
Figure 6-2



Bits	Mnem	Description
1-48	DOUBLE PRECISION FRACTION	Specifies the sign and magnitude of a floating-point number.
49-64	EXP	Specifies the exponent of a floating-point number.

Format of Floating Register (F)
Figure 6-3

FLOATING-POINT NUMBERS

Floating-point numbers are made up of two fields:

- A fraction containing the two's complement value of the number
- An exponent

Bits 1-24 (single precision), bits 1-48 (double precision), or bits 1-48 and 65-112 (quad precision, applicable only to 9950) contain the two's complement value representing the fraction of the number. Bit 1 indicates whether the number is positive (bit 1 contains 0) or negative (bit 1 contains 1). The binary point lies between bits 1 and 2.

Bits 25-32 (single precision) or bits 49-64 (double and quad precision) contain the exponent of the floating-point number. The exponent is the power of 2 that is to multiply the fraction in excess 128 form. The true value of the exponent is always 128 less than the value contained in the exponent field.

In other words:

```
Floating-point Number = (fraction) * (2**(exponent-128))
```

Figure 6-4 shows the format of single (SP), double (DP), and quad precision (QP) numbers. The abbreviated names of the SP, DP, and QP floating-point accumulators are FAC, DAC, and QAC, respectively. The number of floating accumulators for each mode and precision type appears in Table 6-9. These accumulators are overlapped, sharing the same storage.

Table 6-9
Number of Floating-point Accumulators

Name	R Mode	V Mode	I Mode
FAC	1	1	2
DAC	1	1	2
QAC	None	1	1

Location	Size	Format
Memory	Single Precision	1 … 24 25 … 32 FRACTION | EXPONENT
Memory	Double Precision	1 … 48 49 … 64 FRACTION | EXPONENT
Memory	Quad Precision	1 … 48 49 … 64 FRACTION | EXPONENT 65 … 112 113 … 128 FRACTION | UNUSED
Accumulator	Single Precision (2250*, 650, 550-II)	1 … 32 33 … 48 FRACTION | EXPONENT
Accumulator	Single Precision (750, 850, 9950)	1 … 48 49 … 64 FRACTION | EXPONENT
Accumulator	Double Precision	1 … 48 49 … 64 FRACTION | EXPONENT
Accumulator	Quad Precision	1 … 48 49 … 64 FRACTION | EXPONENT 65 … 112 113 … 128 FRACTION | UNUSED

*Throughout this section, points applying to a 2250 also relate to the 150, 250, 250-II, 350, 400, 450, I450, I500, 550, and I1000 systems.

Floating-point Formats
Figure 6-4

Floating Accumulators

In R and V modes, FAC or DAC occupies locations '12-'13 in the current register file set. I mode has two FAC or DAC accumulators labeled 0 and 1 that occupy locations '10-'13. For all modes, QAC combines floating accumulators 0 and 1 into one accumulator occupying locations '10-'13. Note that high-order fraction bits and exponent of a quad floating-point number are found in DAC1 in I mode.

The field address and length registers overlap the floating-point registers. Using FAR0, FLR0, and FAC0 instructions will not modify the contents of FAC1, FLR1, or FAC1, and vice versa. However, mixing FAR0 and FLR0 instructions with FAC0 (32I mode), or combining FAR1 or FLR1 instructions with FAC1 (32I mode or FAC 64V mode), produces variable results from machine to machine and attempt to attempt.

There is no particular implied overlap amongst LDLR and STLR intructions. Extracting the exponent can best be done with either an LDA 6 (address trap) or a DFST T followed by an LDA T+3.

Floating-point Operations

In R, V, and I modes, floating-point has instructions that operate from memory to register or on a register alone. I mode also has some floating-point instructions that operate in a register to register and immediate fashion. Table 6-10 lists all floating-point operations. Note that the first letter of a floating-point instruction shows its data type:

- F for single precision
- D for double precision
- Q for quad precision

Table 6-10
Floating-point Instructions

Mnem	Name	Modes
FAD, DFAD	Floating Add	R, V
QFAD		V
FA, DFA, QFA		I
FC, DFC, QFC	Floating Compare	I
FCM, DFCM	Floating Complement	R, V, I
QFCM		V, I
FCS, DFCS	Floating Compare	R, V
QFCS	and Skip	V
FDV, DFDV	Floating Divide	R, V
QFDV		V
FD, DFD, QFD		I
FLD, DFLD	Floating Load	R, V
QFLD		V
FL, DFL, QFL		I
FMP, DFMP	Floating Multiply	R, V
QFMP		V
FM, DFM, QFM		I
FSB, DFSB	Floating Subtract	R, V
QFSB		V
FS, DFS, QFS		I
FST, DFST	Floating Store	R, V, I
QFST		V, I

Manipulating Floating-point Numbers

The following topics are pertinent for many operations since they deal with some aspect of handling the accumulator results: overflow or underflow, normalization, and rounding.

Overflow and Underflow: Overflow occurs when the number of bits in the exponent of a result exceeds the capacity of its destination's exponent. Underflow happens when the exponent of a result is too small to be represented in a specified register or memory location. For all 50 Series systems, upon overflow or underflow, the fraction is incorrect and the exponent has the incorrect sign. Underflow can be distinguished from overflow by checking the sign of the exponent.

A floating-point exception occurs upon overflow or underflow. When this happens the processor checks the content of bit 7 of the keys for the prescribed action. If bit 7 contains 1, the processor merely sets CBIT to 1. If bit 7 contains 0, the processor sets CBIT to 1 and also loads the FADDR, FCODEH, and FCODEL registers of the user register file as described in Chapter 11.

Because the FAC has a much greater exponent range than the memory format, overflow in single precision is detected only when a store operation is performed. This situation produces a store exception. See Chapter 11 for more information.

Normalization: All numbers generated by arithmetic floating-point operations are normalized by the processor. A number is defined as being normalized either when bits 1 and 2 contain different values or when the number is a zero with both fraction and exponent equal to zero. If this is not the case when a result is first generated, the processor shifts the fraction to the left and adjusts the exponent appropriately until bits 1 and 2 do have different values.

The 9950 retains two extra least significant bits of precision, called guard bits, that are shifted into the right side of the fraction during the first two left bit shifts. If more bit shifts are needed, the processor shifts in zeroes.

Multiply instructions for the 550-II, 650, 750, and 850 also keep guard bits for normalization use. No guard bits are saved in any other instruction or for the 2250 as a whole; in these cases the processor shifts in only zeroes during normalization.

Rounding: Table 6-11 lists the prerequisites and procedures for rounding on all 50 Series systems. Note that rounding is done after the result is normalized; rounding in turn may produce a result that needs to be normalized again.

Table 6-11
Rounding Prerequisites and Procedures

Type	9950	550-II, 650 750 and 850	2250
SP	Add, subtract, multiply: In rounding mode (bit 13 of keys is 1), add guard bit 1 to FAC bit 48 and normalize. FRN may be done in rounding mode and a double round will not occur.	Add, subtract, multiply: FRN compiler option rounds result just before store. (See Store below.)	Add, subtract, multiply: FRN compiler option rounds result just before store. (See Store below.)
	Divide: Always rounds. 49 mantissa bits are generated for rounding to 48.	Divide: Always rounds. 33 mantissa bits are generated for rounding to 32.	Divide: Rounding never done.
	Store: In rounding mode, add 1 to FAC bit 25, normalize result, but leave original FAC mantissa unchanged.	Store: FRN rounds and normalizes just before store. If FAC bit 25 = 1, add 1 to bit 24, zero rest of FAC mantissa.	Store: FRN rounds and normalizes just before store. If FAC bit 25 = 1, add 1 to bit 24, zero rest of FAC mantissa.
	Compare and Skip: In rounding mode, add 1 to FAC bit 25, normalize result, store in temporary register for compare, but do not load back into FAC; original FAC mantissa left left unchanged.	Compare and Skip: Rounding never done.	Compare and Skip: Rounding never done.

Table 6-11 (continued)
Rounding Prerequisites and Procedures

Type	9950	550-II, 650 750 and 850	2250
DP	Arithmetic operations: Rounding is the same as in SP. Other instructions: Rounding never done.	Divide: 49 mantissa bits generated for rounding to 48. Other instructions: Rounding never done.	Rounding never done.
QP	Divide: Always rounds. 97 mantissa bits are generated for rounding to 96. Other instructions: Rounding never done.	n/a	n/a

Normalized Versus Unnormalized Operands

Floating-point operations in Prime processors always produce normalized results. Hence, an unnormalized number can only enter the system as an external input operand. Instructions assume normalized floating-point operands; however, no exception results from unnormalized operands apart from those in a divide. To ensure accurate floating-point results, use normalized numbers.

There are several ways of obtaining normalized numbers. FAD, DFAD, or QFAD instructions normalize an unnormalized memory argument when the other value is a floating-point zero (defined as having both mantissa and exponent equal to zero). The instruction sequence DFLD, DFCM, and DFCM also normalizes an operand. Data conversion instructions FLOT, FLT, FLTA, and FLTH convert integers to normalized floating-point numbers. Lastly, standard Prime compilers and assemblers produce normalized constants.

When floating-point instructions are performed on unnormalized numbers, the following guarantees apply. The instructions do not hang or deviate from the processor's normal flow of control. Add, subtract, complement, and compare and skip instructions produce approximately correct answers. Bit for bit identical values will compare equal or subtract to zero by using either a subtract instruction, or a complement instruction that is followed by an add. All floating load

and store instructions copy 32, 64, or 128 bit quantities from place to place as appropriate without faulting or normalizing unless single precision is used and rounding mode is enabled. Because single precision rounding mode rounds and normalizes on a compare and store, the single precision numbers will always be normalized before a store, causing a bit pattern change.

Using unnormalized numbers for some floating-point operations causes problems in the following cases. Compare and skip instructions fail on machines that look first at the sign, then the exponent, and finally the fraction for possible inequality. Divide produces indeterminate results on all processors but that of the 2250 when confronted with unnormalized numbers. Accuracy loss is probable for all other operations on all other systems.

Programming Notes: FORTRAN 66 programmers often use floating-point to store character strings. To the processor, these character strings are unnormalized floating-point values. REAL*8 values work for copy and identity comparison operations, but make sorted ordering impossible. REAL*4 values work in a similar fashion if rounding mode is not enabled. For storing character strings, use INTEGER*4 since they work faster and permit sorting.

Floating-point Accuracy and Precision

Table 6-12 shows the accuracy of floating-point arithmetic instructions as performed on normalized numbers. The number of guard bits preserved need be no greater than two to simulate infinite precision if normalized numbers are used and the algorithm is carefully designed.

Table 6-13 shows floating-point precision for all 50 Series systems when performed with normalized numbers. The degree of floating-point precision and accuracy varies among these systems due to their differences in implemention, as discussed in the following paragraphs.

Table 6-12
Floating-point Instruction Accuracy

Instruction	9950	750 and 850	550-II and 650	2250
FAD	48+#	48	32	32
DFAD	48+#	48	48	48
FSB	48+#	48	32	32
DFSB	48+#	48	48	48
FMP	48+#	48+	32+	29
DFMP	48+#	48+	48+	45
FDV	48+*	31*	31*	30
DFDV	48+*	47*	47*	46
QFAD	96	n/a	n/a	n/a
QFSB	96	n/a	n/a	n/a
QFMP	96	n/a	n/a	n/a
QFDV	96	n/a	n/a	n/a

+ means 2 extra guard bits are used.
# means rounding mode can be used.
* means rounding is always performed.

Table 6-13
Floating-point Precision for All 50 Series Systems

Precision	9950	750 and 850	550-II and 650	2250
Mantissa Bits:				
Memory	24/48/96	24/48/—	24/48/—	24/48/—
Accumulator	48/48/96	48/48/—	32/48/—	32/48/—
Exponent Bits:				
Memory	8/16/16	8/16/—	8/16/—	8/16/—
Accumulator	16/16/16	16/16/—	16/16/—	16/16/—
Guard Bits	2 for all, excepting quad	2 for multiply	2 for multiply	None
Rounds Automatically	For divide regardless of mode or precision. For rest of SP or DP instructions in rounding mode only.	For divide	For divide	No

The number of mantissa and exponent bits is shown in SP/DP/QP form.

9950 Systems: All 9950 SP and DP arithmetic operations generate at least 48 mantissa bits plus two guard bits to safeguard accuracy during normalization. If more than two bit shifts are needed during normalization, the processor shifts in zeroes. After normalization, the processor rounds if in rounding mode (as explained in Table 6-13), and then renormalizes the result.

To store the number in SP memory while in non-rounding mode, the processor truncates the result to 24 bits. In rounding mode, the processor rounds the stored value to 24 bits.

Quad precision divide instructions generate 97 mantissa bits for rounding to 96. All other operations produce 96 mantissa bits of mantissa; guard bits are not used.

The quad floating point accumulator and memory is 128 bits long. Bits 1-112 of this are used for calculations. Bits 113-128 are unused but are subject to the following restrictions. QFLD loads bits 1-112 into QAC and zeroes QAC bits 113-128, or QFLD loads 128 bits into QAC. QFLD followed by QFST does not reliably copy 128 bits of data. All arithmetic operations zero bits 113-128 on completion.

750 and 850 Systems: The 750 and 850 processors operate in DP even when executing SP instructions. Floating load instructions zero accumulator bits 25 through 48. SP add, subtract, and multiply instructions do not truncate accumulator mantissas to 32 bits, resulting in an additional 16 bits of precision. The multiply instruction keeps extra bits of precision that are used during normalization.

In an SP divide instruction, one mantissa is 48 bits and the other is 24 bits. This instruction generates 33 mantissa bits and rounds to 32 before placing the result in the SP accumulator. A DP divide instruction, however, generates 49 mantissa bits and rounds to 48.

550-II and 650 Systems: A 550-II or 650 system has a separate double-precision hardware floating-point unit. These systems insert zeroes in mantissa bits 25 through 48 of an SP memory argument before loading the accumulator. They also zero mantissa bits 33 through 48 for arguments from the SP accumulator. All arithmetic operations are then performed in DP.

Mantissas are truncated to 32 bits to place the results in the FAC, leaving the low order 16 bits alone in the overlapped DAC. Storing a number in SP memory truncates a number further to 24 bits. A multiply instruction alone preserves two extra bits of precision for use in normalization.

A divide instruction automatically generates an extra mantissa bit for rounding the result to 32 bits (SP) or 48 bits (DP).

A single precision floating load instruction always zeroes accumulator bits 25 through 48 before actually loading the number for systems with PRIMOS Rev. 18 or above.

2250 Systems: When an SP number is loaded from memory to the accumulator, zeroes are placed in FAC mantissa bits 25 through 32. After performing a floating-point operation, the FAC mantissa contains a 32-bit result. To store this result in SP memory, the processor truncates bits 25 through 32 but leaves bits 33 through 48 alone.

DP memory and accumulator mantissas both have a capacity of 48 bits, so no bits of precision disappear when transferring DP numbers from one place to the other.

A single precision floating load instruction always zeroes accumulator bits 25 through 48 before actually loading the number for system with PRIMOS Rev. 18 or above.

Converting Datatypes

Several 50 Series system instructions convert floating-point numbers to integers and vice versa. Table 6-14 lists these instructions and gives a brief description of each.

Table 6-14
Conversion Instructions

Mnem	Name	Modes	Description
DBLE	Convert Single to Double	I	Converts the single precision floating-point number to a double precision floating-point number.
DRN	Double Round from Quad	V,I	Converts a quad precision floating-point accumulator value to a double precision floating-point number.
DRNM	Double Round from Quad to Minus Infinity	V,I	Converts a quad precision floating-point accumulator value to a double precision floating-point number.
DRNP	Double Round from Quad to Plus Infinity	V,I	Converts a quad precision floating-point accumulator value to a double precision floating-point number.
DRNZ	Double Round from Quad to Zero	V,I	Converts a quad precision floating-point accumulator value to a double precision floating-point number.
FCDQ	Floating Convert Double to Quad	V,I	Converts a double precision floating-point accumulator number to a quad precision floating-point number.
FDBL	Floating Point Convert Single to Double	R,V	Converts a single precision floating-point accumulator number to a double precision floating-point number.
FLOT	Convert Integer to Floating Point	R	Converts the 31-bit contents of A and B to a normalized floating-point number and stores the 31-bit result in the floating accumulator.
FLT	Convert Integer to Floating Point	I	Converts the contents of the specified R to a normalized floating-point number and stores the result in the floating accumulator.
FLTA	Convert Integer to Floating Point	V	Converts the 16-bit contents of A to normalized floating-point number and stores the result in the floating accumulator.
FLTH	Convert Halfword Integer to Floating Point	I	Converts the 16-bit integer contained in the specified r to a normalized floating-point number and stores it in the floating accumulator.
FLTL	Convert Integer to Floating Point	V	Converts the 32-bit contents of L to a floating-point number and stores the result in the floating accumulator.

Table 6-14 (continued)
Conversion Instructions

Mnem	Name	Modes	Description
FRN	Floating Round	R,V,I	Rounds the mantissa of a floating-point accumulator number to the nearest 24-bit fraction.
FRNM	Floating Round from DP to Minus Infinity	V,I	Converts a double precision floating-point accumulator value to a single floating-point number.
FRNP	Floating Round from DP to Plus Infinity	V,I	Converts a double precision floating-point accumulator value to a single precision floating-point number.
FRNZ	Floating Round from DP to Zero	V,I	Converts a double precision floating-point accumulator value to a single precision floating-point number.
INT	Convert Floating Point to Integer	R	Converts the number in a floating accumulator to a 31-bit integer and stores it in A and B.
INT	Convert Floating Point to Integer	I	Converts the number in a floating accumulator to a 32-bit integer and stores it in GR2.
INTA	Convert Floating Point to Integer	V	Converts the number in a floating accumulator to a 16-bit integer and stores it in A.
INTH	Convert Floating Point to Halfword Integer	I	Converts the number in a floating accumulator to a 16-bit integer and stores it in r.
INTL	Convert Floating Point to Long Integer	V	Converts the number in the floating accumulator to a 32-bit number and stores it in L.
QINQ	Floating Convert Integer to Quad	V,I	Converts the truncated integer portion of the floating-point accumulator to a quad precision floating-point number.
QIQR	Floating Convert Integer to Quad Rounded	V,I	Converts the rounded integer portion of the floating-point accumulator to a quad precision floating-point number.

DECIMAL DATA

Decimal data can be represented in packed or unpacked forms.

Unpacked Decimal

There are four forms of unpacked decimal numbers, as shown in Figure 6-5.

Type	Format	Example
Leading Sign, not embedded	First byte contains sign only.	10101011 | 10110011 | 10110000 | 10110101 + | 3 | 0 | 5
Trailing Sign, not embedded	Last byte contains sign only.	10110010 | 10110110 | 10110001 | 10101101 2 | 6 | 1 | -
Leading Sign, embedded	First byte contains sign and first digit.	10110110 | 10110110 | 10111001 | 10111001 +6 (6) | 6 | 9 | 9
Trailing Sign, embedded	Last byte contains sign and last digit.	10110100 | 10110110 | 10111000 | 11001010 4 | 6 | 8 | -1 (J)

Unpacked Decimal Formats
Figure 6-5

In the first two cases listed in Figure 6-5, a plus sign represents a positive number, and a minus sign a negative number. You can use a space character to represent a positive sign, and the processor will interpret it correctly. Numerical operations, however, cannot produce positive numbers that contain a space character.

In the two cases where the sign is embedded, a single character represents the appropriate sign and digit. Table 6-15 shows the characters that you use to represent sign/digit combinations.

Table 6-15
Sign/Digit Representations for Unpacked Decimal

Digit	Positive Rep.	Negative Rep.
0	0, space, +, {	}, -
1	1, A	J
2	2, B	K
3	3, C	L
4	4, D	M
5	5, E	N
6	6, F	O
7	7, G	P
8	8, H	Q
9	9, I	R

There are several multiple representations listed above. The processor recognizes all of the representations, but it generates only the first character as the result of an operation. For example, the processor will generate a } to represent a negative zero with embedded sign.

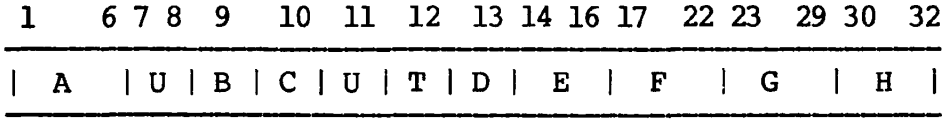
Packed Decimal

The fifth way to represent decimal numbers is called packed decimal. A number in this form uses four bits to represent each digit in the number; the last four bits of the number represent the sign. (Packed decimal numbers are always in trailing sign format.) A decimal number must contain an odd number of digits (excluding the sign digit). It must also begin on a byte boundary.

The sign digit of a decimal result contains a hex C if the sign is positive or a hex D if it is negative. The processor interprets the sign digits of a decimal operand as positive if it contains anything other than a hex C or D.

Control Word Format

Unlike the instructions already listed in this chapter, decimal arithmetic instructions require more information to execute than they can contain. They require a control word to specify the characteristics of the operations to be performed. When a decimal instruction is executing in V mode, L contains a copy of the control word; in I mode, General Register 2 contains the copy. Figure 6-6 shows the format of the control word. Within this figure, F1 and F2 stand for field 1 and field 2, respectively.



Field	Bits	Contents or Meaning
A	1-6	Number (0-'77) of digits in F1
U	7-8	Unused; must be zero
B	9	Sign of F1: B=1: sign of F1 is inverse of specified value 0: sign of F1 is as specified
C	10	Sign of F2: C=1: sign of F2 is inverse of specified value 0: sign of F2 is as specified
U	11	Unused; must be zero
T	12	Sign of result: T=1: result is forced positive 0: instruction operation dictates the sign
D	13	Round flag (used only by XMV)
E	14-16	Decimal data type of F1
F	17-22	Number (0'77) of digits in F2
G	23-29	Scale differential
H	30-32	Decimal data type of F2

Decimal Control Word Format
Figure 6-6

Most of the fields are self-explanatory. Fields D, E, G, H, and T, however, merit extra discussion.

Field D is used only by the XMV instruction. This field tells the processor whether to round the decimal number in F1 or not. If D contains a 0, no rounding occurs. If D contains 1, rounding occurs if the last digit of the F1 (adjusted as specified by field G) is greater than or equal to 5. The rounding occurs when XMV moves the contents of F1 into F2.

For this field to be effective when XMV uses it, make sure that the scale differential in field G is greater than or equal to 1.

Control word fields E and H specify the decimal data types of the operands. Table 6-16 lists the available data types and the codes used to represent them in the control word fields.

Table 6-16
Decimal Data Types

Code	Decimal Data Type
0	Leading separate
1	Trailing separate
3	Packed decimal
4	Leading embedded
5	Trailing embedded

Control word field G specifies the scale differential, the difference in magnitude between the operators of an instruction. This field contains a 7-bit, two's complement number with the value:

Fx = magnitude(F1) - magnitude(F2).

If Fx is positive, then F1 must be shifted right so that it aligns with F2; if negative, F1 must be shifted left to be aligned with F2.

For example, suppose F1 contains 999V99, and F2 contains 999. The scale differential for these operands would be +2, since F1 must be shifted to the right two digits to align with F2.

The T bit is used by the decimal instructions XAD, XDV, XMP, and XMV. For all these instructions, results are forced positive if the T bit contains 1.

The descriptions of the decimal instructions (see Chapters 13 and 14) list the control word fields required for instruction execution. Any unused fields must contain zeroes for proper execution to occur.

Decimal Operations

Decimal results are correct for all the digits shown in the result field. The processor calculates the result to all its bits of precision, then loads as many as can fit into the result field. If the portion stored does not contain the most significant bits of the result, an overflow occurs that causes a decimal exception. (See Chapter 12.)

Register Use

In general, all decimal instructions use GR0, GR1, GR3, GR4, and GR6 in both V and I mode. On the 9950, all decimal instructions use L (GR2 in I mode), FAR0, and FAR1. XDTB and XBTD do not use FAR1, but also use GR4.

Table 6-17 lists the decimal instructions.

Table 6-17
Decimal Instructions

Mnem	Name	Modes	Description
XAD	Decimal Add	V,I	Adds the contents of two decimal fields together and stores the result in the destination field.
XMV	Decimal Move	V,I	Moves the contents of the source field into the destination field.
XCM	Decimal Compare	V,I	Compares the contents of the source and destination fields and sets the condition codes depending on the outcome of the compare.
XMP	Decimal Multiply	V,I	Multiplies the contents of the source and destination fields and stores the result in the destination field.
XDV	Decimal Divide	V,I	Divides the contents of the destination field by the contents of the source field and stores the result and the remainder in the destination field.
XBTD	Binary to Decimal Conversion	V,I	Converts a binary number contained in a register to a decimal number and stores the result in a memory location.
XDTB	Decimal to Binary Conversion	V,I	Converts a decimal number in memory to a binary number and stores the result in a register.
XED	Decimal Edit	V,I	Edits a decimal string under control of an edit subprogram.

CHARACTER STRINGS

Character strings are made up of bytes, with each byte representing one ASCII character. A character string can contain from 1 to (2**17)-1 bytes. Table 6-18 lists the character instructions.

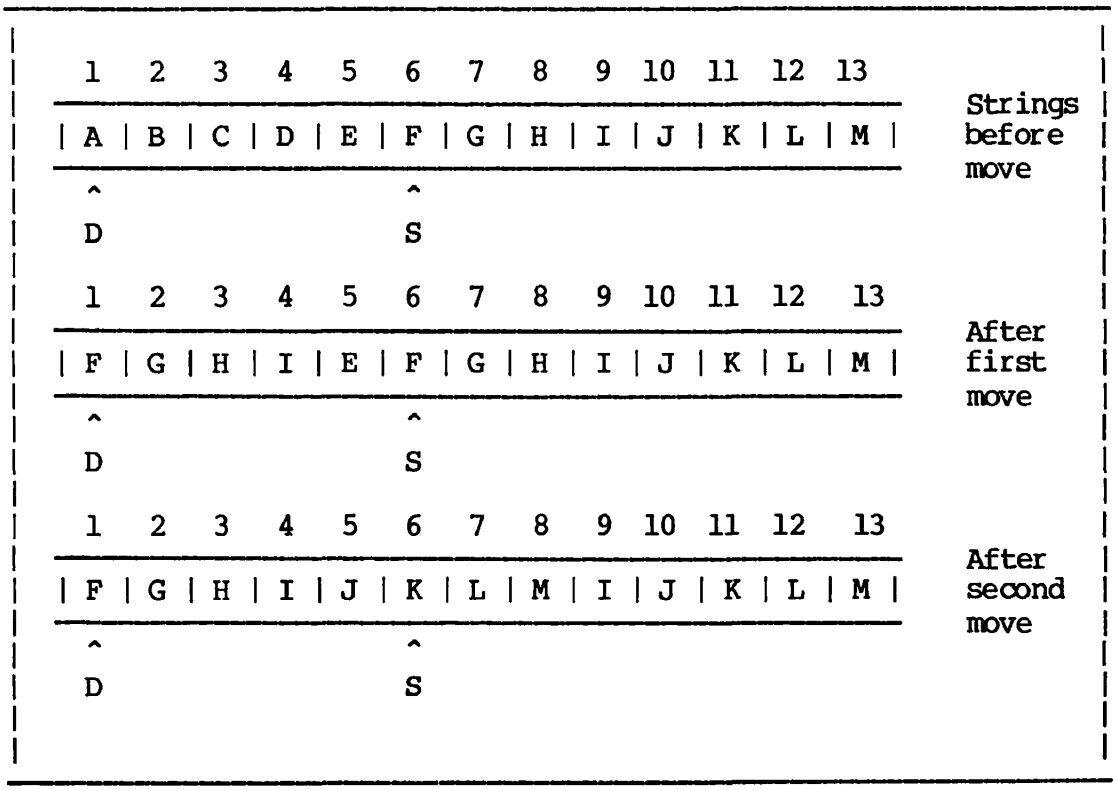
Table 6-18
Character Instructions

Mnem	Name	Modes	Description
LDC	Load Character	V,I	Calculates an effective address. Loads the character in the specified field into bits 9-16 of a register. Clears bits 1-8.
STC	Store Character	V,I	Stores the contents of bits 9-16 of A into the specified field.
ZCM	Compare Character Fields	V,I	Compares two character fields and sets the condition codes depending on the outcome of the compare.
ZED	Edit Character Fields	V,I	Moves characters from one field to another under control of an edit subprogram.
ZFIL	Fill Field	V,I	Stores a character into each byte of the specified field.
ZMV	Move Characters	V,I	Moves characters from one field to another.
ZMVD	Move Equal Length	V,I	Moves characters from one field to another of equal length.
ZTRN	Translate Character Field	V,I	Uses one field to reference a translation table and construct a second field.

The Z-prefix character instructions (that is, all character instructions except LDC and STC) move data in the source string starting from the lowest addressed byte (ascending order). Note that ZED and ZTRN move one byte at a time; ZCM, ZFIL, ZMV, and ZMVD always move four bytes at a time (unless there are fewer than six bytes to move and the source and destination are not aligned).

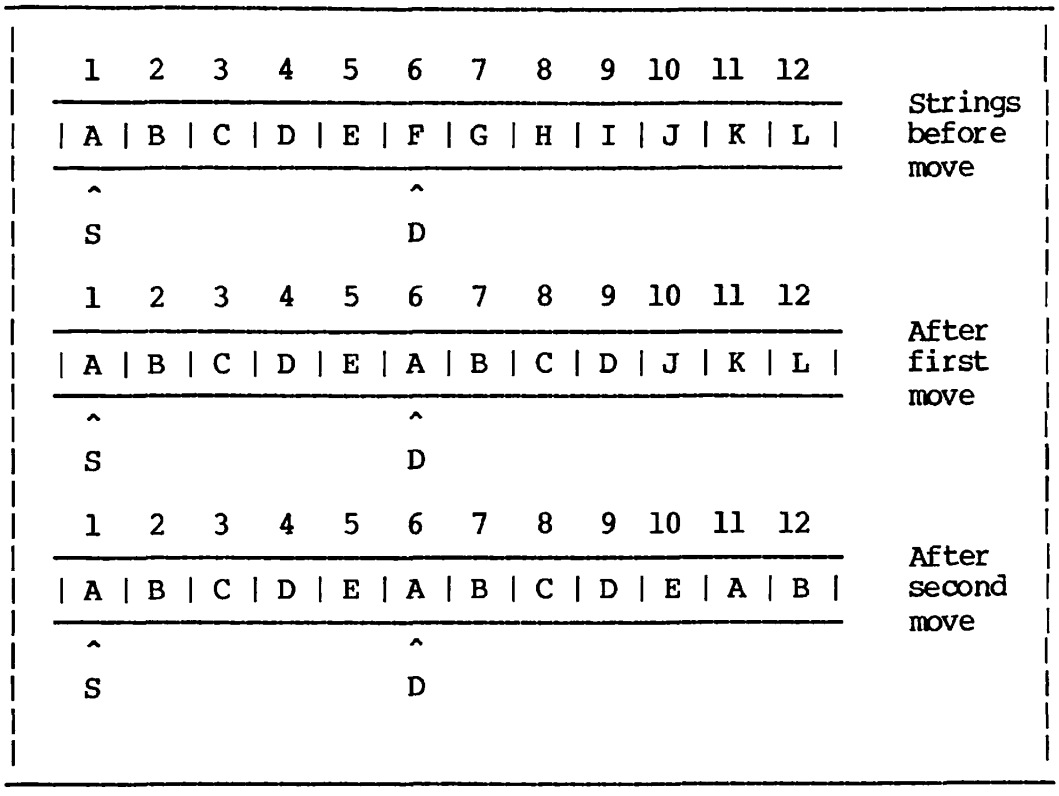
The Z-prefix character instructions may produce unexpected results if the source and destination strings overlap. For example, suppose ZMV is to move the contents of a large source string into a destination string. Figure 6-7 shows how the source and destination strings overlap; S represents the first byte in the source string (labelled 6) and D represents the first byte in the destination string (labelled 1).

After ZMV moves the first four characters, the strings are as shown in the second part of Figure 6-7. The last part shows how the second move affects the string. The third and subsequent moves would work in the same way. In this case ZMV simply moves all characters in the source string into the destination string straightforwardly, without deviation.



String Manipulation
Figure 6-7

Suppose, however, that the starting addresses of the two strings are switched. The first five bytes in the source string will be correctly moved, but the rest of the string will have been overwritten by copies of the first five bytes. These same five characters will propagate through the rest of the destination string, as shown in Figure 6-8.

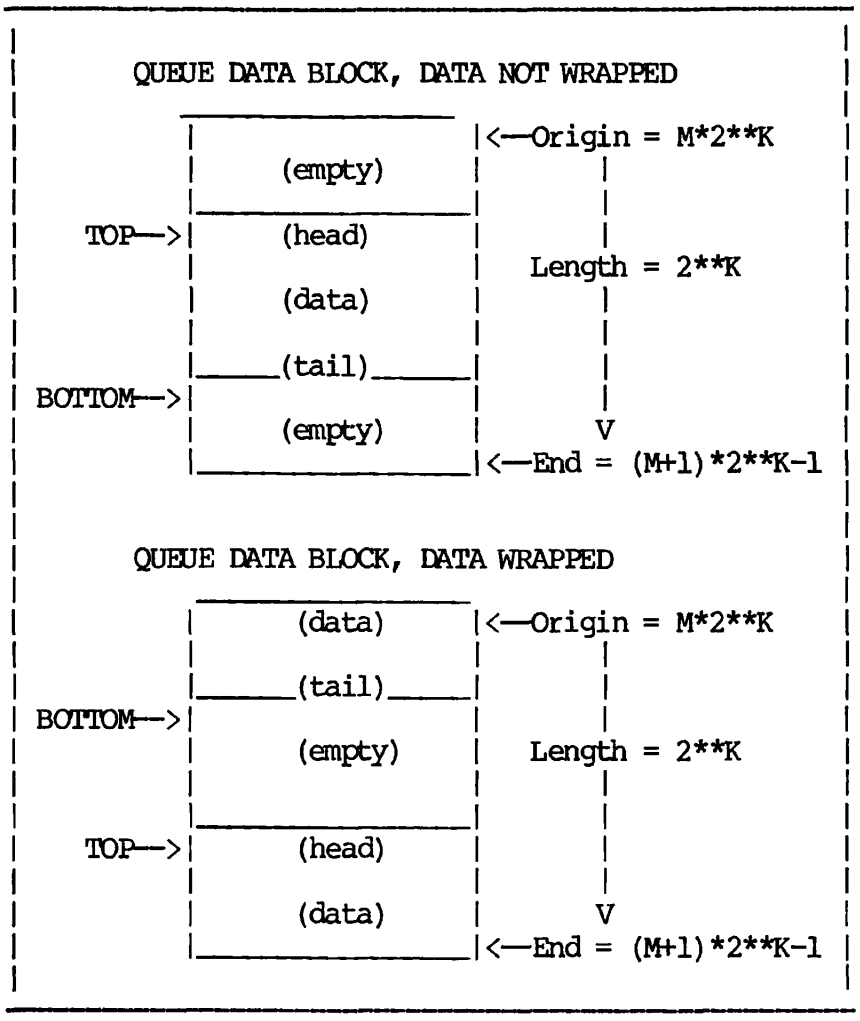


String Manipulation
Figure 6-8

While the move shown in Figure 6-8 is useful, it may not be the action that was intended. Overlapping strings produce arbitrarily different results for each Prime machine. For this reason, avoid using overlapping strings in any situation.

QUEUES

A queue is a fixed length, double-ended, circular word buffer. Figure 6-9 shows the format of a typical queue with wrapped and unwrapped data.



Queues With Wrapped and Unwrapped Data
Figure 6-9

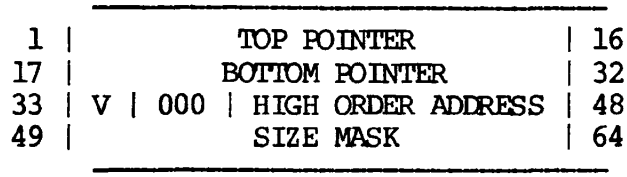
QCBs

Each queue in the system is controlled by a queue control block (QCB). This QCB contains information about the queue's location in memory, as well as data used to manipulate the elements. In addition, the QCB defines the queue's type. If the QCB has a physical address, the associated queue is called a physical queue. These types of queues are the only ones used for DMQ operations. If the QCB has a virtual

segment number and offset rather than a physical address, the queue is called a virtual queue. Queues of this type are never used for I/O operations.

Try to align QCBs on 8-byte boundaries. DMQ operations (discussed in Chapter 12) require this alignment. For program queue manipulation via the queue instructions, alignment is not necessary but does produce faster queue operations.

Figure 6-10 Shows the format of the QCB.



Bits	Name	Description
1-16	Top Pointer	Points to first filled location (the head) in the queue.
17-32	Bottom Pointer	Points to last filled location (the tail) in the queue.
33	V	Virtual/physical control bit: 0 = physical queue, 1 = virtual queue.
34-36	———	Reserved; must be 0.
37-48	High Order Address	Queue address (if V = 0), or segment number (if V = 1).
49-64	Size Mask	Mask; value = 2**(K-1).

Figure 6-10
Format of the QCB

When addressing a QCB, the ring number in the reference specifies the access privileges that will govern the reference. Physical queues can only be accessed from Ring 0.

Queue Specifications

A queue must be 2**K words long, where K is an integer between 4 and 16 inclusive. In addition, the queue's starting address must be M(2**K), where M is an integer value. These restrictions allow the firmware to easily identify and locate a queue. Note that two queues in the system do not have to have the same K in common.

The 50 Series processors use a mask word to add elements to or delete elements from a queue. This mask specifies the size of the queue, and is 16 bits wide. The least significant K bits contain 1 and all other bits contain 0. This means that the numerical value of the mask is (2**K)-1.

```
Suppose K = 5.
     mask = 0000000000011111
          = '37
          = 31 decimal
          = (2**5)-1, QED.
```

Calculating a Mask
Figure 6-11

The mask also makes it easy to determine the starting and ending addresses of the queue. If P is a pointer to some location within a queue, the address of the queue's origin is:

origin = P AND (NOT mask)

and the address of the queue's last location is:

end = P OR mask.

```
Suppose K = 5,  P = '204, and M = 4.
     mask = '37 and queue length = 2**5 = '37.

    origin = '204 AND (NOT '37)
           = 10000100 AND 1111111111100000
           = 10000000
           = '200
           = 128 decimal
           = 4(2**5), QED.

       end = '204 OR '37
           = 10000100 OR 11111
           = 10011111
           = '237
           = queue origin + queue length
           = '200 + '37, QED.
```

Calculating the Origin and End of a Queue
Figure 6-12

Queues operate under one final restriction. They are defined to be empty when the contents of the top pointer equal the contents of the bottom pointer. This means that the maximum number of elements in a queue is (2**K)-1.

Queue Algorithms

The 50 Series processors use four algorithms to insert or delete queue elements (depending on the specified operation). Table 6-19 shows the algorithms used for specific operations. The symbols T1-T5 represent temporary storage registers.

Table 6-19
Queue Algorithms

Inst	Algorithm
RTQ	T1 <- TOP T2 <- BOTTOM If T1 = T2 then A <- 0 CC <- EQ else T3 <- SEGMENT T4 <- MASK A <- SEGMENT | T1 (16 bits) TOP <- T1 AND NOT T4 OR (T1 + 1) AND T4
ABQ	T1 <- TOP T2 <- BOTTOM T3 <- SEGMENT T4 <- MASK T5 <- T2 AND NOT T4 OR (T2 + 1) AND T4 If T1 = T5 then CC <- EQ else location(SEGMENT | T2) <- A BOTTOM <- T5
ATQ	T1 <- TOP T2 <- BOTTOM T3 <- SEGMENT T4 <- MASK T1 <- T1 AND NOT T4 OR (T1 - 1) AND T4 If T1 = T2 then CC <- EQ else location(SEGMENT | T1) <- A TOP <- T1
RBQ	T1 <- TOP T2 <- BOTTOM If T1 = T2 then A <- 0 CC <- EQ else T3 <- SEGMENT T4 <- MASK T2 <- T2 AND NOT T4 OR (T2 - 1) AND T4 A <- SEGMENT | T2 (16 bits) BOTTOM <- T2

The instructions provided for programmed queue manipulation are shown in Table 6-20. The pointer in the instructions references the QCB for that queue. Note that an RTQ instruction is equivalent to a DMQ output operation, and an ABQ is equivalent to a DMQ input, as noted in Chapter 12, Input/Output.

Table 6-20
Queue Instructions; S, R, V Modes

Mnem	Name	Description
RTQ	Remove from Top of Queue	Removes a single word from the top of a queue and places it in A.
RBQ	Remove from Bottom of Queue	Removes a single word from the bottom of a queue and places it in A.
ABQ	Add to the Bottom of Queue	Adds the contents of A to the bottom of the specified queue.
ATQ	Add to the Top of Queue	Adds the contents of A to the top of the specified queue.
TSTQ	Test Queue	Sets A to the number of items in a specified queue and sets the condition codes depending on the new value of A.

SUMMARY OF DATATYPES AND APPLICABLE INSTRUCTIONS

Table 6-21 summarizes the different datatypes and lists the various operations available. The body of the table shows which instructions perform a specific operation on a specific datatype. For detailed information about each instruction, refer to the instruction dictionaries in Chapters 13 and 14.

When using Table 6-21, note that aa represents the set of arithmetic conditions [EQ, GE, GT, LE, LT, NE]. Also note that Table 6-21 does not include instructions that operate on CBIT, LINK, the condition codes, or queues.

Table 6-21
Summary of Datatypes and Applicable Instructions

Operation	Size of Datatype (in Bits)							
	16 (A)	31 (A/B)	32 (L)	64 (L/E)	32FP (FAC)	64FP (DAC)	128FP (QAC)	Dec (-)
Load from memory	LDA	DLD	LDL		FLD	DFLD	QFLD	XMV
Store to memory	STA	DST	STL		FST	DFST	QFST	
Add	ADD	DAD	ADL		FAD	DFAD	QFAD	XAD
Subtract	SUB	DSB	SBL		FSB	DFSB	QFSB	XAD
Multiply	MPY		MPL		FMP	DFMP	QFMP	XMP
Divide	DIV		DVL		FDV	DFDV	QFDV	XDV
Increment	IRS, A1A, A2A							
Decrement	S1A, S2A							
AND	ANA		ANL					
OR	ORA							
XOR	ERA		ERL					
Complement	CMA							
Compare	CAS, CAZ		CLS		FCS	DFCS	QFC, QFCS	XCM
Logical test	Laa		LLaa		LFaa	LFaa		
Branch	Baa		BLaa		BFaa	BFaa		
Logical left shift	ALL		LLL					
Logical right shift	ARL		LRL					
Arithmetic left shift	ALS	LLS	LLS					
Arithmetic right shift	ARS	LRS	LRS					
Rotate left shift	ALR		LLR					

Table 6-21 (continued)
Summary of Datatypes and Applicable Instructions

Operation	Size of Datatype (in Bits)							
	16 (A)	31 (A/B)	32 (L)	64 (L/E)	32FP (FAC)	64FP (DAC)	128FP (QAC)	Dec (-)
Rotate right shift	ARR		LRR					
Clear	CRA	CRL	CRL	CRLE				
Clear left	CAL	CRA	CRA	CRL				
Clear right	CAR	CRB	CRB	CRE				
Interchange halves	ICA	IAB	IAB	ILE				
Interchange and clear left	ICL	XCA	XCA					
Interchange and clear right	ICR	XCB	XCB					
Two's complement	TCA		TCL		FCM	DFCM	QFCM	
Set sign	SSM	SSM	SSM					
Clear sign	SSP	SSP	SSP					
Change sign	CHS		CHS					
Convert datatypes:								
Integer to floating point	FLTA	FLOT	FLTL					
Floating point to integer	INTA	INT	INTL				QINQ QIQR	
Binary to decimal	XBTD		XBTD	XBTD				
Decimal to binary	XDTB		XDTB	XDTB				
Position for integer divide	PIDA	PID	PIDL	PIDL				
Position after multiply	PIMA	PIM	PIML	PIML				
Skips	Saa				FSaa	FSaa		

SUMMARY

This chapter has introduced the datatypes supported on the 50 Series processors and has listed the instructions you can use to manipulate them. The next chapter, Altering Sequential Flow, lists instructions that allow you to test for a condition and perform actions depending on the outcome of the test.

7
Altering Sequential Flow

So far this document has confined its discussions mostly to arithmetic operations. This chapter describes instructions that can alter the normally sequential flow of control within a program.

BRANCH AND SKIP INSTRUCTIONS

The simplest way to change the flow of control in a program is to use a branch or a skip instruction. These instructions may directly load a new value into the program counter, or they may first test some value and then load the program counter according to the outcome of the test. Note that branch and skip instructions always load the program counter with an address contained within the current segment. (To transfer control to an address outside the current segment, use a jump instruction, explained in the second half of this chapter.)

Table 7-1 lists the branch instructions. Table 7-2 lists the logic test instructions. Table 7-3 contains information about the conditional skip instructions. Table 7-4 describes the floating-point skip instructions.

Table 7-1
Branch Instructions

Mnem	Name	Modes	Description
BEQ, BGE, BGT, BLE, BLT, BNE	Branch on A Set With Respect to 0	V	Branches if the contents of A meet the specified condition with respect to 0.
BCEQ, BCGE, BCGT, BCLE, BCLT, BCNE	Branch on CC Set With Respect to 0	V,I	Branches if the condition code reflects the specified condition with respect to 0.
BFEQ, BFGE, BFGT, BFLE, BFLT, BFNE	Branch on FA With Respect to 0	V,I	Branches if the contents of the floating accumulator reflect the specified condition with respect to 0.
BHEQ, BHGE, BHGT, BHLE, BHLT, BHNE	Branch on r With Respect to 0	I	Branches if the contents of the specified r meet the specified condition with respect to 0.
BLEQ, BLGE BLGT, BLLE, BLLT, BLNE	Branch on L With Respect to 0	V	Branches if the contents of L meet the specified condition with respect to 0.
BMEQ, BMGE, BMGT, BMLE, BMLT, BMNE	Branch on Magnitude Condition Set With Respect to 0	V,I	Branches if LINK and the condition codes meet the the specified condition with respect to 0.
BREQ, BRGE, BRGT, BRLE, BRLT, BRNE	Branch on R Set With Respect to 0	I	Branches if the contents of the specified R meet the specified condition with respect to 0.
BRBR	Branch on R Bit Reset	I	Branches if the specified bit in R is 0.
BRBS	Branch on R Bit Set	I	Branches if the specified bit in R is 1.
BHD1, BHD2, BHD4	Branch on r Decremented by Value	I	Decrements r by the specified value and branches if the value is not equal to 0.
BHI1, BHI2, BHI4	Branch on r Incremented by Value	I	Increments r by the specified value and branches if the values is not equal to 0.

Table 7-1 (continued)
Branch Instructions

Mnem	Name	Modes	Description
BRD1, BRD2, BRD4	Branch on R Decremented by Value	I	Decrements R by the specified value and branches if the value is not equal to 0.
BRI1, BRI2, BRI4	Branch on R Incremented by Value	I	Increments R by the specified value and branches if the values is not equal to 0.
BCS	Branch if CBIT is Set	V,I	Branches if the value of CBIT is 1.
BCR	Branch if CBIT is Reset	V,I	Branches if the value of CBIT is 0.
BLS	Branch if LINK is Set	V,I	Branches if the value of LINK is 1.
BLR	Branch if LINK is Reset	V,I	Branches if the value of LINK is 0.
BDX	Branch on Decremented X	V	Decrements the contents of X by 1 and branches if the decremented value equals 0.
BDY	Branch on Decremented Y	V	Decrements the contents of Y by 1 and branches if the decremented value equals 0.
BIX	Branch on Incremented X	V	Increments the contents of X by 1 and branches if the incremented value equals 0.
BIY	Branch on Incremented Y	V	Increments the contents of Y by 1 and branches if the incremented value equals 0.
CGT	Computed GOTO	V,I	Branches if the contents of A are greater than 1 and less than a specified integer; otherwise, executes the next instruction.

Table 7-2
Logic Test Instructions

Mnem	Name	Modes	Description
LEQ, LGE, LGT, LLE, LLT, LNE	Load on Register With Respect to 0	S,R,V,I	Loads a register with a 1 if the register reflects the specified condition with respect to 0; otherwise, clears the register to 0.
LCEQ, LCGE, LCGT, LCLE, LCLT, LCNE	Load Register on Condition Codes Set With Respect to 0	S,R,V,I	Loads a register with a 1 if the condition codes reflect the specified condition with respect to 0; otherwise, clears the register to 0.
LFEQ, LFGE, LFGT, LFLE, LFLT, LFNE	Load Register on FAC With Respect to 0	S,R,V,I	Loads a register with a 1 if the contents of the floating accumulator reflect the specified condition with respect to 0; otherwise, clears the register to 0.
LHEQ, LHGE, LHGT, LHLE, LHLT, LHNE	Load R on r With Respect to 0	I	Loads R with a 1 if the contents of r reflect the specified condition with respect to 0, or with a 0 if another condition exists.
LLEQ, LLGE, LLGT, LLLE, LLLT, LLNE	Load A on L With Respect to 0	S,R,V	Loads A with a 1 if the contents of L reflect the specified condition with respect to 0, or with a 0 if another condition exists.
LT	Load True	S,R,V,I	Loads a register with a 1.
LF	Load False		Loads a register with a 0.

Table 7-3
Conditional Skip Instructions

Mnem	Name	Modes	Description
CAS	Compare A and Skip	S,R,V	Compares the contents of A to the the contents of a memory location and skips depending on the result of the compare.
CAZ	Compare A to 0	S,R,V	Compares the contents of A to 0 and skips depending on the outcome of the test.
CLS	Compare L and Skip	V	Compares the contents of L to the contents of a memory location and skips depending on the outcome of the compare.
DRX	Decrement and Replace X	S,R,V	Decrements the contents of X by 1 and skips the next word if the decremented value is 0.
IRS	Increment and Replace Memory	S,R,V	Increments the contents of a memory location and skips the next word if the incremented value is 0.
IRX	Increment and Replace X	S,R,V	Increments the contents of X and skips the next word if the incremented value is 0.
SAR	Skip on A Register Bit 0	S,R,V	Skips the next word if the specified bit in A contains 0.
SAS	Skip on A Register Bit 1	S,R,V	Skips the next word if the specified bit in A contains 1.
SGT	Skip on A Greater than 0	S,R,V	Skips the next word if the contents of A are greater than 0.
SLE	Skip on A Less Than 0	S,R,V	Skips the next word if the contents of A are less than 0.
SNR	Skip on Sense Switch Reset to 0	S,R	Skips if the contents of the specified sense switch are are equal to 0.
SNS	Skip on Sense Switch Set to 1	S,R	Skips if the contents of the specified sense switch are equal to 1.

Table 7-4
Floating-point Skip Instructions

Mnem	Name	Modes	Description
FSGT	Floating Skip If Greater Than 0	R,V	Skips the next location if the contents of the floating accumulator are greater than 0.
FSLE	Floating Skip If Less Than or Equal to 0	R,V	Skips the next location if the contents of the floating accumulator are less than or equal to 0.
FSMI	Floating Skip If Minus	R,V	Skips the next location if the contents of the floating accumulator are less than 0.
FSNZ	Floating Skip If Not Zero	R,V	Skips the next location if the contents of the floating accumulator are not equal to 0.
FSPL	Floating Skip If Plus	R,V	Skips the next location if the contents of the floating accumulator are greater than 0.
FSZE	Floating Skip If Zero	R,V	Skips the next location if the contents of the floating accumulator are equal to 0.

JUMP INSTRUCTIONS

Like the instructions listed in the tables above, jump instructions can load new addresses into the program counter. The difference is that jump instructions can transfer control to addresses outside the current segment of execution. Table 7-5 lists these instructions.

SUMMARY

The 50 Series supports branch, skip, and jump instructions that you can use to transfer control from one part of your program to another. The next chapter begins the discussion of more complex methods of control transfers.

Table 7-5
Jump Instructions

Mnem	Name	Modes	Description
JDX	Jump on Decremented X	S,R,V	Decrements the contents of X by 1 and jumps if the decremented value is 0.
JEQ	Jump on A Equal to 0	S,R,V	Jumps if the contents of A equal 0.
JGE	Jump on A Greater Than or Equal to 0	S,R,V	Jumps if the contents of A are greater than or equal to 0.
JGT	Jump on A Greater Than 0	S,R,V	Jumps if the contents of A are greater than 0.
JIX	Jump on Incremented X	S,R,V	Increments the contents of X by 1 and jumps if the incremented value is 0.
JLE	Jump on A Less Than or Equal to 0	R	Jumps if the contents of A are less than or equal to 0.
JLT	Jump on A Less Than 0	R	Jumps if the contents of A are less than 0.
JMP	Unconditional Jump	S,R,V,I	Jumps to the specified effective address.
JNE	Jump on A Not Equal to 0	R	Jumps if the contents of A are not equal to 0.
JSR	Jump to Subroutine	I	Jumps to the specified effective address and saves the return address in r.
JST	Jump and Store	S,R,V	Stores the current contents of the program counter into memory and jumps to the specified effective address.
JSX	Jump and Save in X	S,R,V	Increments the contents of the program counter by 1 and stores the result in X, then jumps to the specified effective address.
JSXB	Jump and Save in XB	V,I	Stores the current contents of the program counter in XB and jumps to the specified effective address.
JSY	Jump and Save in Y	V	Increments the contents of the program counter by 1 and stores the result in Y, then jumps to the specified effective address.

8

Stacks and Procedure Calls

This chapter describes how to transfer control from one procedure to another. This type of control transfer, the procedure call, can:

- Call inward rings from outward rings.
- Invoke reentrant procedures.
- Invoke recursive procedures.
- Use an embedded operating system.

Before describing how procedure calls work, however, this chapter defines several key terms. It also describes the stack, the data blocks that contain information about a call, and the special access rights that govern a call.

DEFINITION OF TERMS

Note the difference between the terms process and procedure. A procedure is a set of instructions, such as the body of a text editor or diagnostic program. A process is the execution of a procedure, such as the process that the system assigns to a user. A process may execute several procedures throughout its life.

A procedure may call other procedures by using the Procedure Call (PCL) instruction. A processor may exchange one process for another by invoking the process exchange mechanism (PXM). For information about the PXM, refer to Chapters 9 and 10.

Note also the use of the terms caller, callee, calling procedure, and called procedure. The procedure making the call is the calling procedure, or caller. The procedure answering the call is the called procedure, or callee. These terms are used throughout this and future chapters.

STACKS AND STACK MANAGEMENT

The more sophisticated methods of altering sequential program flow use stacks as temporary storage areas. Procedure calls use the stack to save the state of the machine before altering program flow and to contain the parameters of the call. When the specified operation is complete, information in the stack is used to restore the machine state to what it was before the procedure call took place.

A stack is a group of one or more segments. Since a 50 Series processor can support more than one stack at a time, the segment number of the first segment in each stack (the stack root) serves as a unique identifier. Stack segments following the stack root segment are called stack extension segments. A stack can contain many stack extension segments.

Stack Header

The first four locations of the stack root segment contain the stack header. These locations contain information needed by the processor to manage the stack. Table 8-1 shows the format of these locations.

Each stack extension segment also has a header. Words 0-1 of each extension segment must contain 0. Words 2-3 contain an extension pointer that references the next stack extension segment. This pointer contains 0 if this segment is the last stack extension segment.

Table 8-1
Stack Header Format for the Initial Stack Segment

Word	Name	Description
0,1	Free Pointer	Pointer to first word of next free space in the current stack segment (segment number/ word number). This value must be even.
2,3	Stack Extension Pointer	Pointer to first location of extension segment, if one has been allocated. If there is not enough room to allocate a new frame in the current segment referenced by the free pointer, the processor uses the extension pointer to reference the next segment. If the extension pointer contains 0, no extension segment has been allocated and a stack overflow fault occurs.

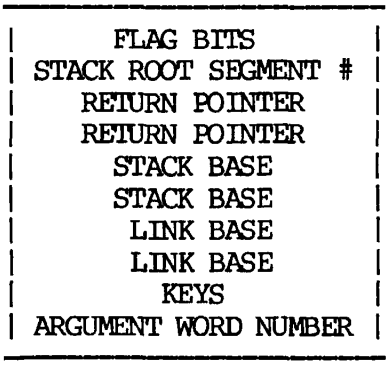
Stack Frames

The 50 Series processors store information on the stack in blocks called stack frames. They allocate the frames in a last in first out (LIFO) manner. Each time the PCL instruction executes, a new frame is allocated; a PRTN instruction deallocates the frame when the procedure specified by PCL completes execution. Note that an unextended frame cannot cross a segment boundary. (See STEX in Chapters 13 and 14.)

The stack frames allocated at any time are backward threaded only. This means that each frame points back to the frame of the procedure that previously used this stack.

The information contained in a frame header defines the state of the machine that was in effect when the calling procedure executed the PCL instruction. This arrangement permits calls to or returns from a procedure without having to reference the frame of the calling procedure.

Figure 8-1 shows the format of the stack frame header. Note that all procedures in the same ring can use the same stack for storage. Different processes, however, usually do not share stack segments.



Words in Frame	Contents	Description
1	Flag Bits	PCL always sets these bits to 0.
2	Stack Root Segment #	Address of the free pointer.
3,4	Return Pointer	Pointer to return location (that following the last argument template of the PCL instruction that created this frame).
5,6	Stack Base	Contents of caller's SB (pointer to previous frame).
7,8	Link Base	Contents of caller's LB.
9	Keys	Contents of caller's keys.
10	Argument Word #	Word number of the location following the PCL that created this frame.

Stack Frame Format
Figure 8-1

ENTRY CONTROL BLOCKS

The entry control block (ECB) identifies a procedure. When PCL executes, it forms the effective address of the called procedure's ECB, not of the procedure itself. The ECB contains information about the called procedure, as well as about the expected parameters (such as number of expected arguments, size of stack frame, and so on). Figure 8-2 shows the format and contents of the ECB.

1	16 17	32
ECB.PBH	ECB.PBL	
ECB.SFSIZE	ECB.ROOTSN	
ECB.ARGDISP	ECB.NARGS	
ECB.LBH	ECB.LBL	
ECB.KEYSH	0	
0	0	
0	0	
0	0	

Word in Block	Name	Description
0-1	ECB.PB	Pointer (ring, segment, word number) to the first executable instruction of the called procedure.
2	ECB.SFSIZE	Stack frame size to create (in words). Must be even.
3	ECB.ROOTSN	Stack root segment number. If zero, keep same stack.
4	ECB.ARGDISP	Displacement in new frame of where to build argument list.
5	ECB.NARGS	Number of arguments expected.
6-7	ECB.LB	Pointer (ring, segment, word) to be loaded as called procedure's linkage base (location of called procedure's linkage frame less '400).
8	ECB.KEYS	Keys desired by called procedure.
9-15		Reserved, must be zero.

Entry Control Block Format
Figure 8-2

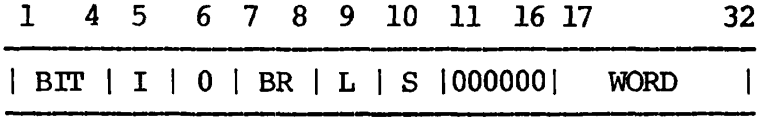
INDIRECT POINTERS

If the callee expects arguments, several pointers to the arguments should follow the PCL instruction. These pointers are called argmuent templates (or argument pointers). They contain directions that PCL uses to form indirect pointers to the actual arguments. Indirect pointers are saved in a stack frame that the callee uses to reference the arguments.

Several templates may be used in succession to form one indirect pointer. One template may specify a level of indirection; the next, a base register. Each template contains an S bit that determines if that template is the last one to be used to form a single indirect pointer. If this S bit contains 1, then the argument is the last one to be used for this indirect pointer, and the processor should store it into the current stack frame. If the S bit contains 0, then the indirect pointer requires more templates.

Each template also contains an L bit to indicate if it is the last one for the last indirect pointer. When L and S are both 1, then this argument is the last one for the last pointer. When L is 0, other arguments follow it. When L is 1 and S is 0, the processor forms dummy indirect pointers. See Storing Indirect Pointers, below, for information about these dummy indirect pointers.

Figure 8-3 shows the format of all argument templates. Figure 3-3 in Chapter 3 shows the format of 32-bit and 48-bit indirect pointers.



Bits	Mnem	Contents
1-4	BIT	Bit number.
5	I	Indirect.
6	—	Reserved; must be 0.
7-8	BR	Base register.
9	L	Last template for this call.
10	S	Last template for this argument; store argument address.
11-16	—	Reserved; must be zero.
17-32	WORD	Word number.

Argument Template Format
Figure 8-3

GATE ACCESS

There are some Ring 0 or 1 procedures that procedures in higher-numbered rings will want to call. Since normal read, write, and execute access rights will not allow such inward references, these Ring 0 or 1 procedures must specify a special access right called gate access. Gate access allows a Ring 3 procedure to safely use a specific set of Ring 0 and 1 procedures without harming the rest of the system.

For identification, the ECBs of the procedures that allow gate accesses are grouped in a special gate access segment. These ECBs must all have starting addresses of 0(mod16) in this segment. If a procedure references an improperly aligned ECB, an access fault occurs.

To call any of the procedures allowing gate accesses, the caller must execute a PCL instruction that points to an ECB in the gate access segment. There is no other way to call these procedures.

MAKING A PROCEDURE CALL

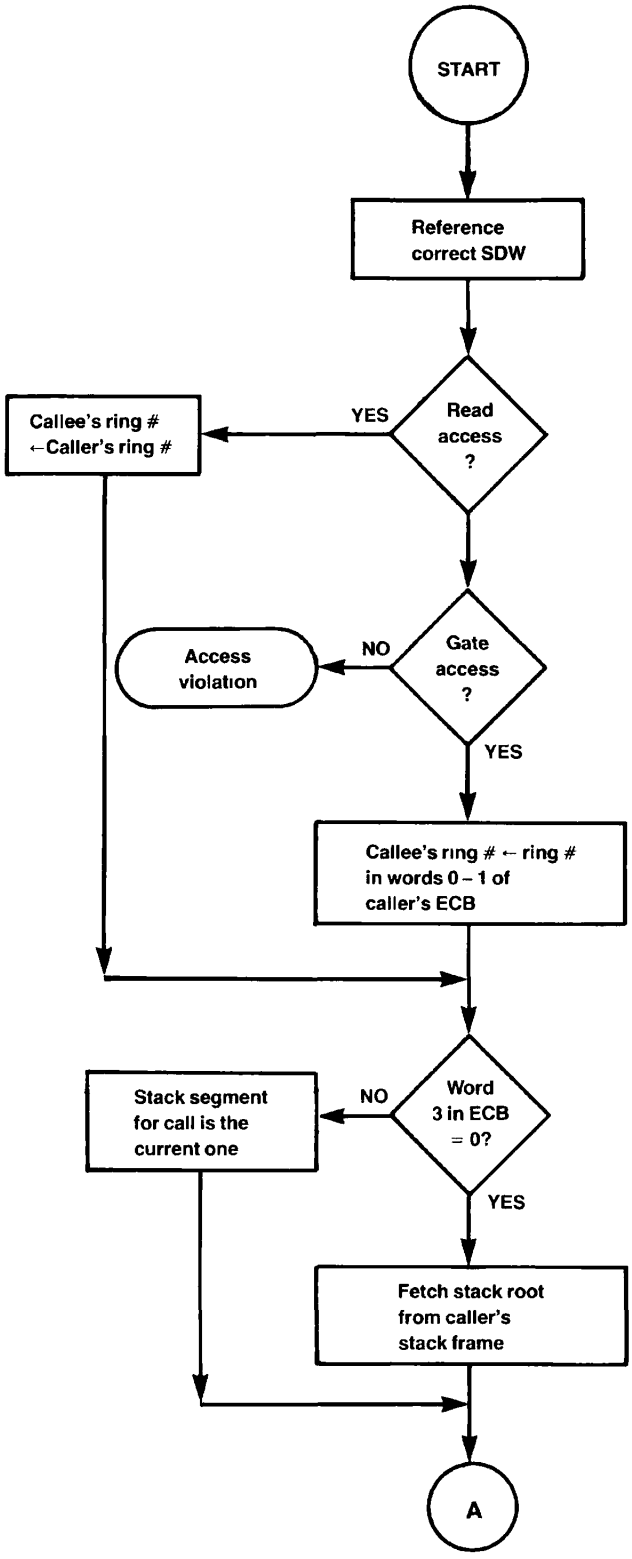
When PCL executes, it:

- Calculates the callee's ring number.
- Allocates a new stack frame for the callee.
- Saves the caller's state.
- Loads the callee's state.
- Calculates and stores indirect pointers for the callee's use.

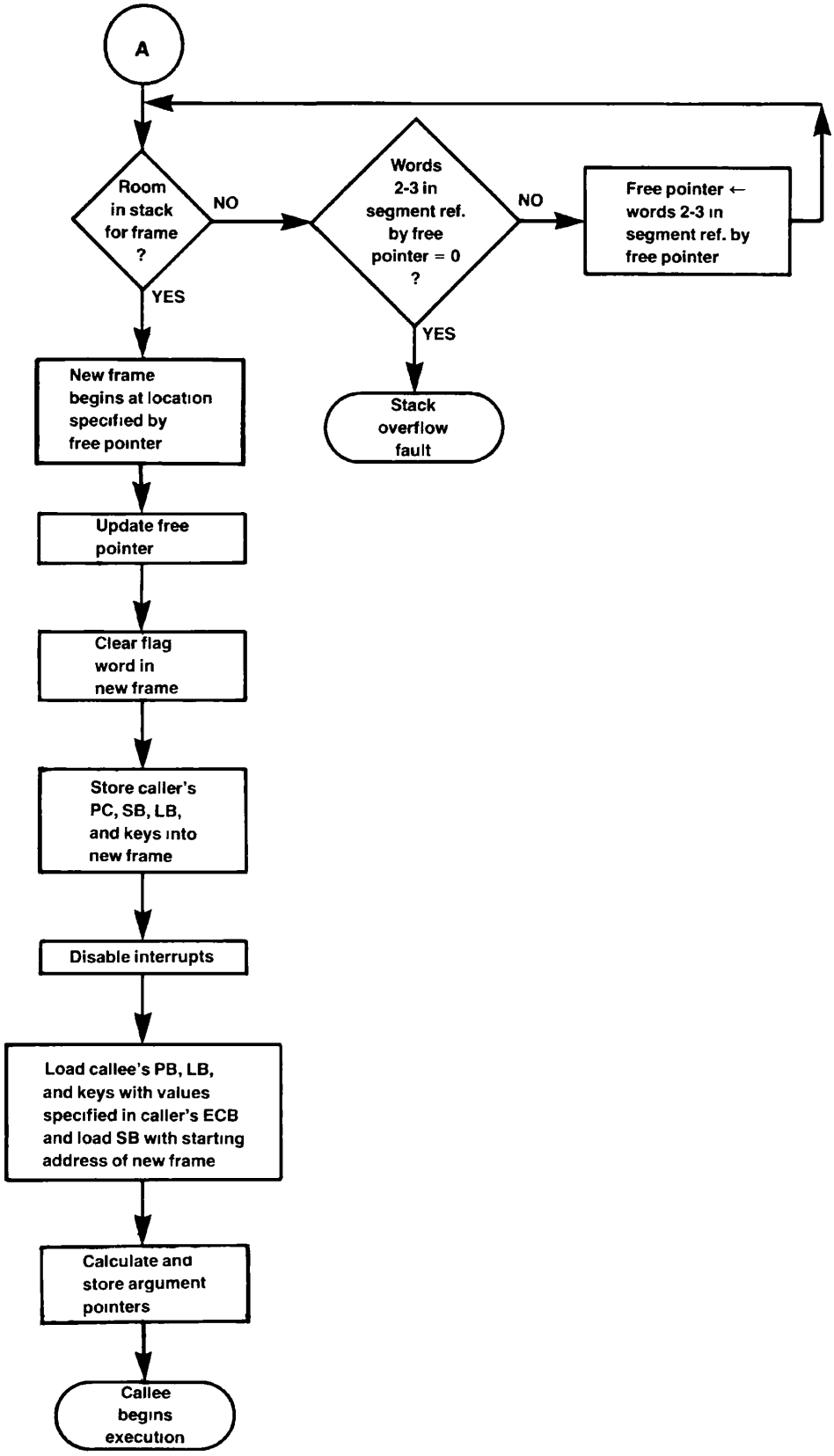
This sequence of events is summarized in Figure 8-4 and described below.

Calculating a Ring Number

When PCL begins execution, it calculates the ring number of the call. PCL looks at the appropriate STLB entry, since it contains access rights for the calling procedure. PCL uses these access rights to determine if the caller has access to the callee's ECB. If the STLB specifies read access, PCL weakens the ring number contained in the callee's ring field to that of the caller. If the callee's ECB is in a gate segment, PCL uses the ring field contained in words 0-1 of the callee's ECB as the ring number.



Actions of PCL, Part 1
Figure 8-4a



Actions of PCL, Part 2
Figure 8-4b

Allocating a Stack

PCL looks at the contents of ECB.ROOTSN (word 3 of the ECB) to determine the stack root segment. If ECB.ROOTSN contains zeroes, the processor fetches the stack root number from the stack frame of the caller. (Gate ECB's must have a nonzero stack root segment indicated in ECB.ROOTSN.) The first two words of the stack root segment contain the free pointer; PCL compares the number of available locations in the segment to the contents of ECB.SFSIZE (the number of words contained in a frame). Stack frame sizes and free pointers are always rounded upwards to form an even value.

If the frame will fit into the locations remaining in the stack segment, PCL starts the new frame at the location specified by the free pointer. It also updates the contents of the free pointer so that they point past the new frame.

If the new frame is too large to fit in the current segment, PCL examines the contents of words 2-3 in the segment referenced by the free pointer. If words 2-3 contain 0, a stack overflow fault occurs.

If words 2-3 contain a nonzero value, this value becomes the new free pointer. PCL rechecks for available segment locations as it did the first segment. If this segment cannot contain the whole frame, a stack fault occurs. If there are enough available locations, PCL starts the frame at the first available location.

Saving the Caller's State

The processor clears the flag word of the new frame and stores the contents of the caller's program counter, stack base and link base registers, and keys into the new frame. Note that the contents of the saved program counter specify the ring and segment of the caller, and that these saved contents point to the location immediately following PCL.

Loading the Callee's State

PCL disables interrupts so that page faults, STLB faults, or interrupts cannot disrupt the system while it is loading the callee's state. After these are disabled, PCL loads the program counter with the contents of ECB.PB and LB with the contents of ECB.LB. The keys are loaded with the contents of ECB.KEYS; note, however, that bits 15-16 of the keys are set to 0. PCL also loads the address of the new frame into SB.

Calculating Indirect Pointers

Figure 8-5 shows how the indirect pointers are formed. The text that follows elaborates on this figure.

To form an indirect pointer, PCL first forms the ring field. It compares the contents of the program counter's ring field and that of the base register specified in the caller. The larger value of these two fields becomes the ring field of the indirect pointer.

The contents of the segment field of the caller's specified base register become the contents of the indirect pointer's segment field. The contents of the base register's word and bit fields are added to those of the word and bit fields specified in the argument template to form the appropriate fields for the indirect pointer. When XB is the base register, bits 1-4 of X (GR7 in I mode) contain the bit field, if there is one.

When the argument template bit field is added to the XB field, the carry out goes into the word number part of the address. Note that the argument template bit field is used only when the argument template indirect bit is 0.

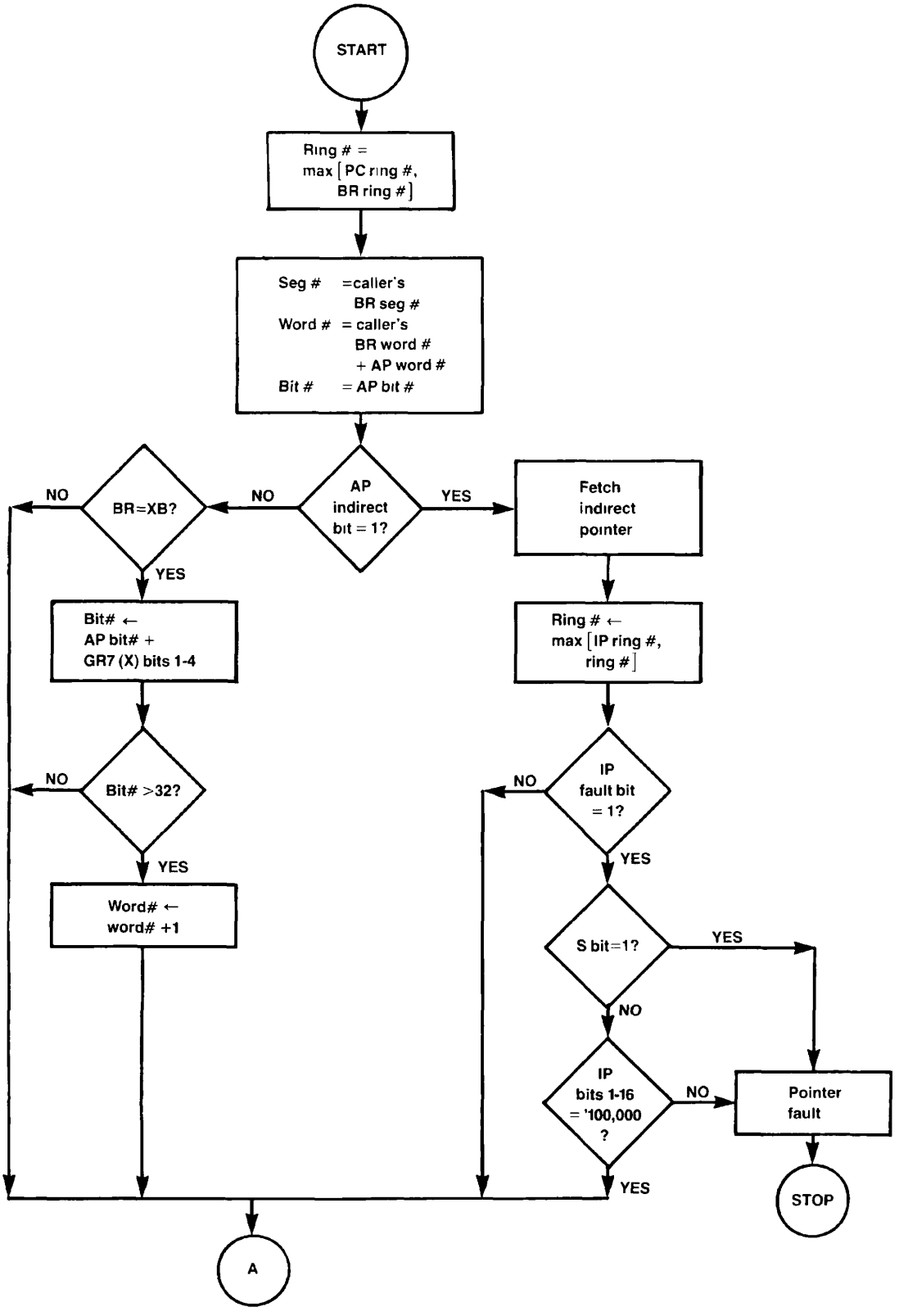
If the argument template indirect bit is 0, the value just calculated is the final value.

If the argument template indirect bit is 1, the value just calculated is not the final value. PCL uses this calculated value to fetch the indirect pointer. PCL compares the calculated value's ring field to the caller's ring field (found in the program counter) and takes the larger of the two as the new ring field. The contents of the segment, word, and bit fields are the same as the contents of those just calculated.

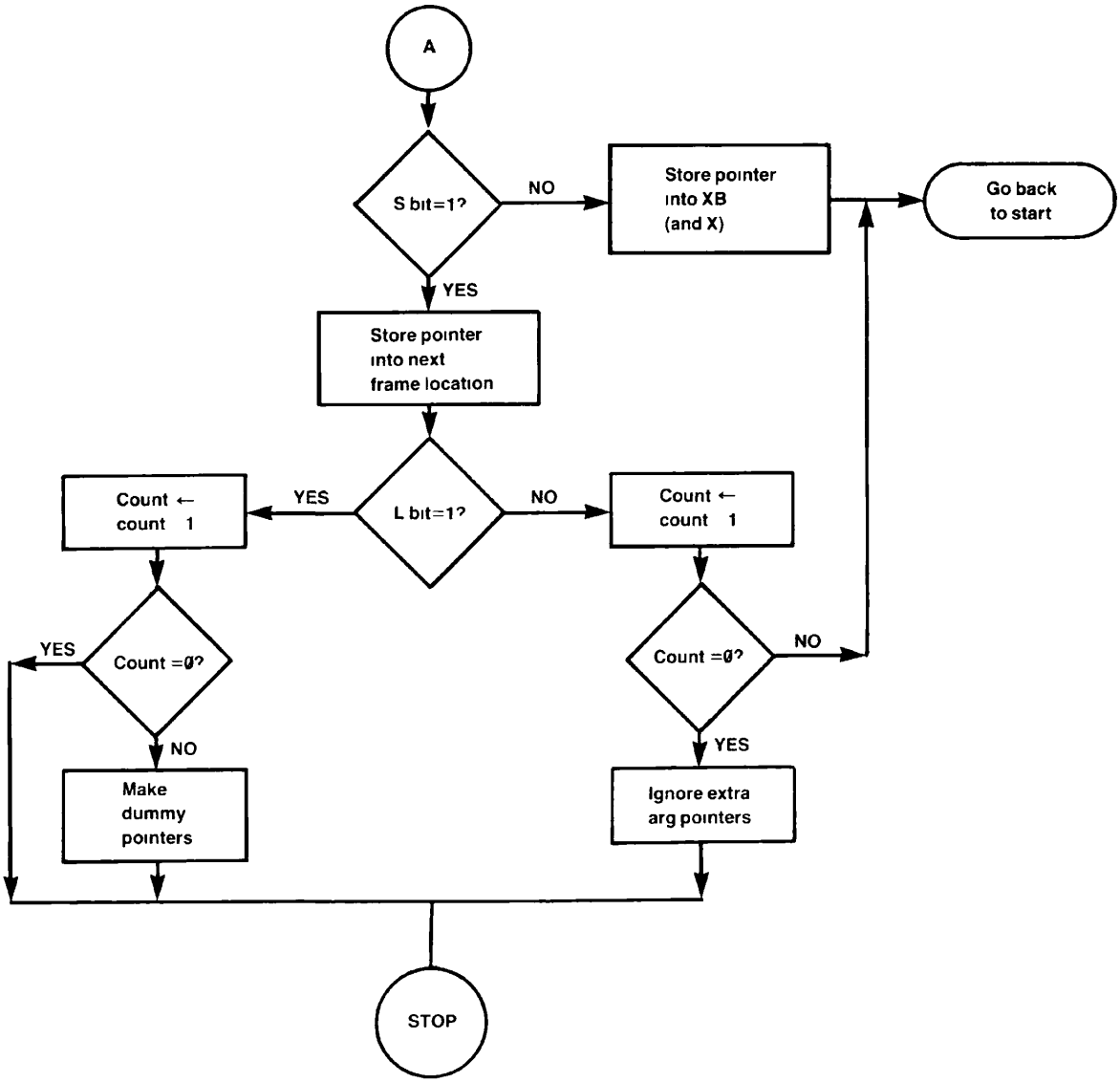
When an indirect pointer's fault bit contains a 1, the contents of the argument template S bit and the pointer's first word determine the action to be taken. If the S bit contains a 1 and the pointer's first word is '100000, the indirect pointer is loaded onto the callee's stack frame; all other cases result in a pointer fault.

Once PCL finds the final value generated by the template, it examines the S bit to determine if it should store the pointer in the stack frame as an indirect pointer, or if it should store the pointer in XB.

If S contains a 0, PCL must use at least one more template to complete the formation of the pointer. The value calculated so far is stored in XB. (If there is a bit field, the value is also stored in X. Bit 4 of XB, the E bit, contains 1 when X is used.) The value calculated for the next template is stored in XB and X again. This continues until the S bit of one of the templates contains a 1.



Calculating and Storing Argument Pointers, Part 1
Figure 8-5a



Calculating and Storing Argument Pointers, Part 2
Figure 8-5b

Storing Indirect Pointers

If S contains a 1, PCL stores the calculated indirect pointer in the next stack frame location. If L also contains a 1, then there are no more indirect pointers to be calculated. A 0 in L indicates that there are more arguments to follow, so PCL proceeds with the next one.

If the number of indirect pointers produced is greater than the number the callee expects, PCL ignores the extras.

If the number of indirect pointers produced is less than the number the callee expects, PCL creates dummy indirect pointers and stores them in the current frame. The format of these dummy pointers is '100000, where bit 1 = 1 indicates a pointer fault (omitted argument pointer). PCL stores one dummy pointer for each omitted one.

Note that the callee can reference omitted indirect pointers only to pass them on to other new procedures; if such a reference occurs, the new procedure will see such indirect pointers as omitted. Any use of an omitted indirect pointer other than to pass it on causes a pointer fault.

PCL always allocates three words in the current stack frame to store each indirect pointer. An indirect pointer occupies all three words, however, only if it has a nonzero bit field. If this is the case, PCL sets the E bit for that indirect pointer to 1. If an indirect pointer has a bit field containing 0, PCL sets the argument's E bit to 0 and loads the indirect pointer into the first two allocated locations; when PCL loads the next indirect pointer, it skips the third location.

THE ARGT INSTRUCTION

PCL is resumable if any interruption occurs while it is transferring arguments. When such an interruption occurs, the program counter in the return block contains the address of the first instruction in the callee. If the callee does not expect arguments, its first instruction can be anything. If arguments are expected, however, the first instruction of the callee must always be the Argument Transfer (ARGT) instruction. After the processor services the interrupt, control returns to ARGT, which identifies how many indirect pointers have yet to be transferred, and begins the transfer anew at that point.

Note that ARGT transfers arguments only if an interrupt occurs during PCL's execution. If this happens, ARGT completes the transfer that PCL began. If no interrupt occurs, ARGT is not executed.

THE PRTN INSTRUCTION

After all arguments are transferred, control transfers to the called procedure. The last instruction of the called procedure must be a procedure return instruction, PRTN. When this called procedure completes execution, PRTN transfers control back to the calling procedure. The calling procedure picks up execution at the instruction immediately following PCL and its arguments.

PRTN also deallocates the stack frame created when the procedure call was first made. To deallocate the frame, the instruction stores the current value of the stack base register into the free pointer. It then restores the caller's state by loading the caller's stack base and link base registers with the values contained in the frame being deallocated. The keys are similarly loaded, but bits 15-16 of the keys are set to 0. PRTN also loads the program counter with the appropriate address contained in the frame, but loads the program counter's ring field with the logical OR (weaker) of the saved program counter ring number and the current ring number. This prevents inward returns, yet allows returns from gated calls to work properly.

Programming Notes

When making a procedure call, make sure that the caller, callee, and associated ECB all contain consistent information about arguments. If the ECB specifies no arguments, then no argument templates should follow PCL, nor should the callee begin with ARGT. Similarly, if the ECB specifies arguments, the associated callee must begin with ARGT, and PCL should be followed by the correct number of argument templates (or fewer).

Also note that PCL without argument pointers does not change the contents of any general registers or XB. PCL with argument pointers may alter the contents of some general registers, so do not rely on them to be the same as they were before PCL executed. Specifically, when calling an inner-ring procedure, do not use an indexed or an XB-relative PCL instruction. If an asynchronous interrupt condition occurs, the software restarts the interrupted call at the location specified by the calling PCL. Since neither XB nor the general registers were saved during the first try of PCL, the processor may calculate an invalid effective address.

In addition, do not specify an XB-relative argument template unless it is immediately preceded by at least one other template whose S bit is 0. The previous template's S bit tells the processor that another template is to follow, and to save the current template in XB, not to store it in memory. The processor reads in the XB-relative template, and uses the saved contents of XB in the manipulation. If the XB-relative template were not immediately preceded by another template whose S bit is 0 and if the processor were to retry PCL, XB would not contain valid contents; the calculated template would be invalid.

9

Process Exchange on Single-stream Processors

INTRODUCTION

The previous chapter described how to transfer control from one procedure to another. This chapter and the next discuss the process exchange mechanism (PXM) and how it transfers control from one process to another. This chapter describes the PXM implemented on the single-stream processors: the 2750, 2250, 250-II, 550-II, and 750. The next chapter describes the PXM implemented on the dual-stream 850.

As defined in the previous chapter, a process is a dynamic state of execution, such as a user in a time-sharing system. To quickly service as many processes as possible (up to approximately 1000 at once), the 50 Series PXM executes one process for a given length of time. If a resource is not available or time for this process is up, the PXM exchanges this process for another, and so on. This allows many processes to work towards completion at the same time.

ELEMENTS OF THE PXM

The main elements of the process exchange mechanism (PXM) are:

- Three data structures:

  Process control blocks
  Ready list
  Wait lists

- Two PXM instructions:

  WAIT
  NOTIFY

- The dispatcher

In addition to these elements, the PXM manipulates the register file and the process interval timer during process exchange.

PROCESS CONTROL BLOCKS

Each process has a process control block (PCB) that describes it. Each PCB contains a minimum of 64 words and completely specfies its process from a hardware point of view. Table 9-1 shows the format of the PCB.

A single dedicated segment contains the PCBs of all processes running throughout the system. Bits 1-16 of word 25 in the current register set specify the number of this segment, OWNERH. (See Table 9-6 later in this chapter for the format of the current register set.) All pointers and addresses in a PCB (except fault vectors and wait list pointers) are 16 bits long and are assumed to be relative to OWNERH. (For more information on OWNERH, see the section on User Register Files, later in this chapter.)

PCBs generally start on 0(mod64) boundaries, but must start on at least 0(mod32) boundaries.

READY LIST

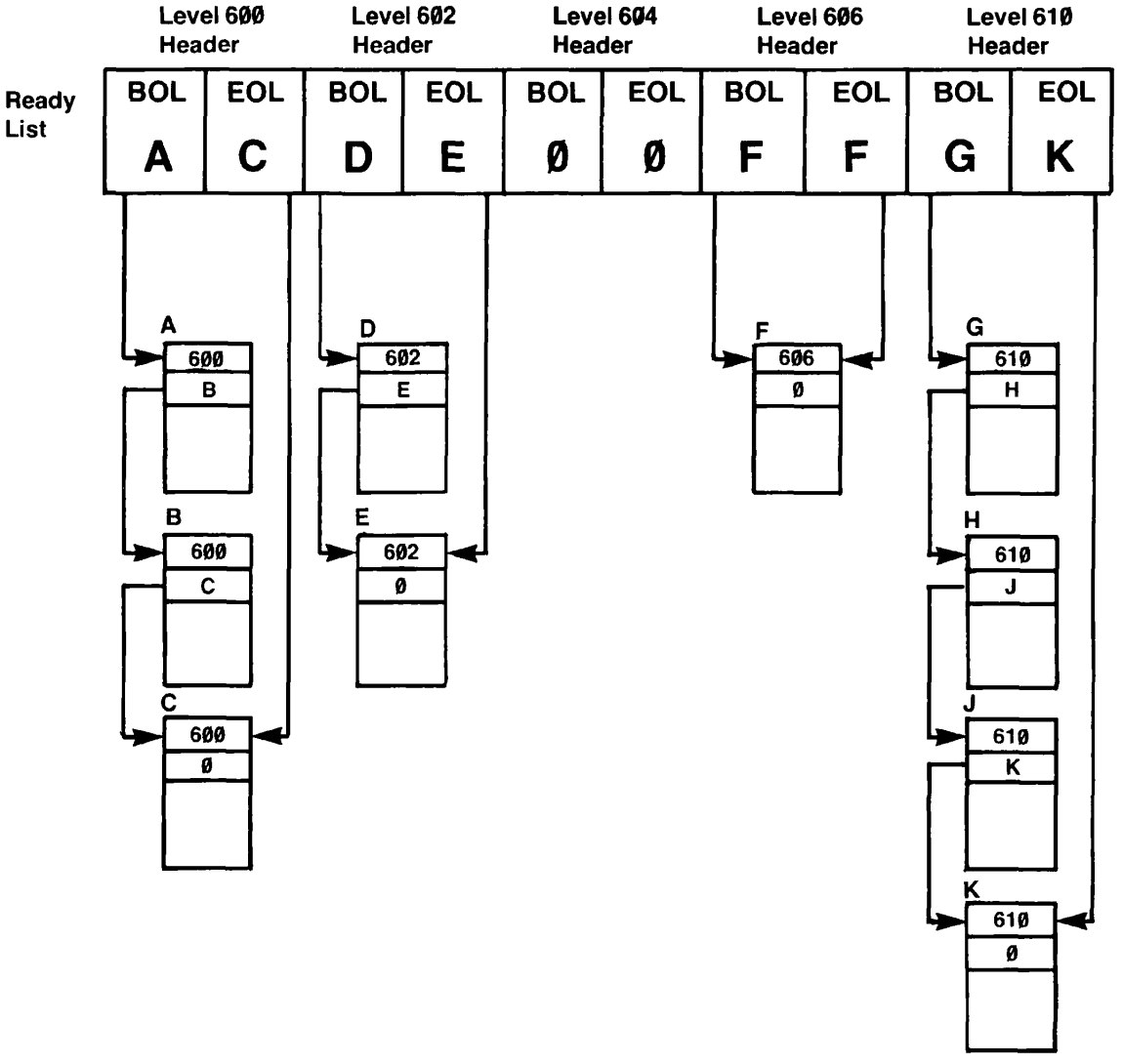
The PXM uses the ready list to indicate priorities and dispatch processes. The elements of the ready list are:

- A series of headers that make up the actual ready list.
- A data base made up of PCBs.
- Two 32-bit registers, PPA and PPB.

Figure 9-1 and the text in the following section show the relationships between the ready list elements.

Table 9-1
PCB Format

Section	Word # (octal)	Contents
Control	0	Level pointer to BOL in ready list.
	1	Link pointer to next PCB, or 0.
	2-3	Segment #/word # of the semaphore whose wait list is currently pointing to this PCB. A segment # of 0 indicates that this PCB is on the ready list.
	4	Abort flags used to generate a process fault when this PCB is dispatched. Bits 1-15: set by the software Bit 16: process interval timer overflow
	5	Pointer to the register set this process used last.
	6-7	Reserved for future use.
Process State	10-11	Process elapsed timers. These must be maintained by the software that resets the live interval timer.
	12-15	DTAR2 and DTAR3. These are never saved, only restored.
	16	Interval timer, bits 1-16.
	17	Interval timer, bits 17-32.
	20	Save mask. PXM uses this to avoid saving or restoring registers containing zeroes. Format of the word is: 1-8: GR0-GR7 (8 32-bit registers) 9-12: FP0-FP1 (4 32-bit registers) 13-16: base registers (4 32-bit registers; PB, SB, LB, XB)
	21	Keys.
	22-61	Storage for nonzero registers. (See Save mask, above.)
Fault	62-63	Fault vector. Segment #/offset to fault table for Ring 0.
	64-65	Fault vector. Segment #/offset to fault table for Ring 1.
	66-67	Reserved for future use.
	70-71	Fault vector. Segment #/offset to fault table for Ring 3.
	72-73	Fault vector. Segment #/offset to fault table for page fault.
	74-76	Concealed fault stack header (FIRST, NEXT, and LAST pointers).
	77	Reserved.
	100-137	Concealed stack. These words can go anywhere in segment OWNERH; they need not start at location '100. The concealed stack can contain as many frames as desired.



Ready List and Associated PCB Lists
Figure 9-1

Headers

The ready list itself is made up of headers, one header for each level of priority. These headers are allocated in contiguous memory locations, with the highest priority header contained in the lowest numbered memory location. Each header, in turn, is made up of two 16-bit pointers. The pointers are called the beginning of list (BOL) pointer and the end of list (EOL) pointer, and each contains the address of a PCB in segment OWNERH.

The PCB referenced by a BOL pointer is associated with the first process having a particular priority. The EOL pointer points to the PCB of the last process with that particular priority.

A BOL pointer containing a 1 signals the end of the ready list, since PCB addresses must be even. A BOL pointer containing a 0 signals an empty level.

Ready List Data Base

The ready list data base is made up of linked lists of PCBs whose associated processes are ready to execute. There is one list defined for each level of priority; all PCBs contained in that list have the same level of priority. A list can contain as many processes as can exist in the system at a time.

The first location in each PCB specifies the process' priority level by pointing to one of the BOL pointers in the ready list. The second location contains a forward link to the next PCB in the linked list. For the last PCB in the linked list (that is, the last PCB in the ready list with this level of priority), the second location contains 0.

PPA and PPB Registers

The PXM uses the pointer to process A (PPA) and pointer to process B (PPB) registers to locate the next process to dispatch. Both registers are 32 bits wide.

PPA always contains information about the currently active process. Bits 17-32 contain PCBA, the address of the process' PCB. Bits 1-16 contain the level of priority, called Level A. Level A always specifies the system's highest priority level that has an associated PCB ready to run. This is because the system's currently running process is always the highest priority process that is capable of running.

PPB contains Level B and PCBB, which specify the priority level and the PCB address, respectively, of the next process to run when execution of the current process terminates.

Using PPA, PPB, and the Ready List

To show how PPA and PPB are used, suppose Process H is running when Process J, whose priority is higher than that of Process H, needs to be serviced. This means that Process J preempts Process H. The PXM suspends Process H, saves the contents of PPA (which reference Process H) in PPB, and then services Process J. When Process J completes, the PXM checks PPB to see what process to run next. PPB identifies Process H, and so the PXM resumes execution of Process H.

Except when bringing the system up from a cold start, software should never alter the contents of PPA or PPB. This holds even if PCBA or PCBB contains 0, indicating invalid register contents. Even if PCBA is invalid, Level A specifies the highest level of priority that was executing in the system, and this determines the starting point of a scan to find the next process to run. When PCBA is invalid, PCBB is guaranteed to be invalid. Note that PCBB is also invalid when the system is idle.

Upon cold start, the cold start software loads the PLA register with the highest level of priority in the ready list. At all other times, however, Level A specifies the highest level of priority that was last known to contain a process. All scans of the ready list can begin at this last known level. Whenever the PXM needs to run a process of higher priority than that specified in Level A, the PXM loads PPA with that higher level.

The PXM does not maintain a pointer to the highest priority level of the ready list. The ready list allocator that starts the PXM, however, knows the starting address of the ready list. In addition, Level A always points to either the highest priority level currently in the system, or the last known highest level. This means that Level A can be a pointer into the ready list.

If PCBB is valid, Level B points to the next process to be executed when the current process completes. Note that the priority level of this next process is lower than or equal to that of the currently executing process. If PCBB is invalid, the contents of Level B are unpredictable.

WAIT LISTS

Wait lists specify a group of processes that are waiting for an event to occur. There are two major elements of each wait list:

- A semaphore
- A data base made up of PCBs

Figure 9-2 and the text in the following section describe the relationship between the semaphore and the wait list PCBs.

Semaphores

Semaphores define an event, such as the completion of a task. The definition of the semaphore is known by at least two processes, or by one process and phantom interrupt code. Upon completion of the event, a NOTIFY instruction changes the value of the semaphore. This change in value may cause the PXM to run a new process.

A semaphore consists of two sequential 16-bit memory locations. The first location contains a WAIT counter, C. If C is greater than zero, then it specifies the number of PCBs on the associated wait list. If C is negative, it specifies the number of times the event has occurred without running a process.

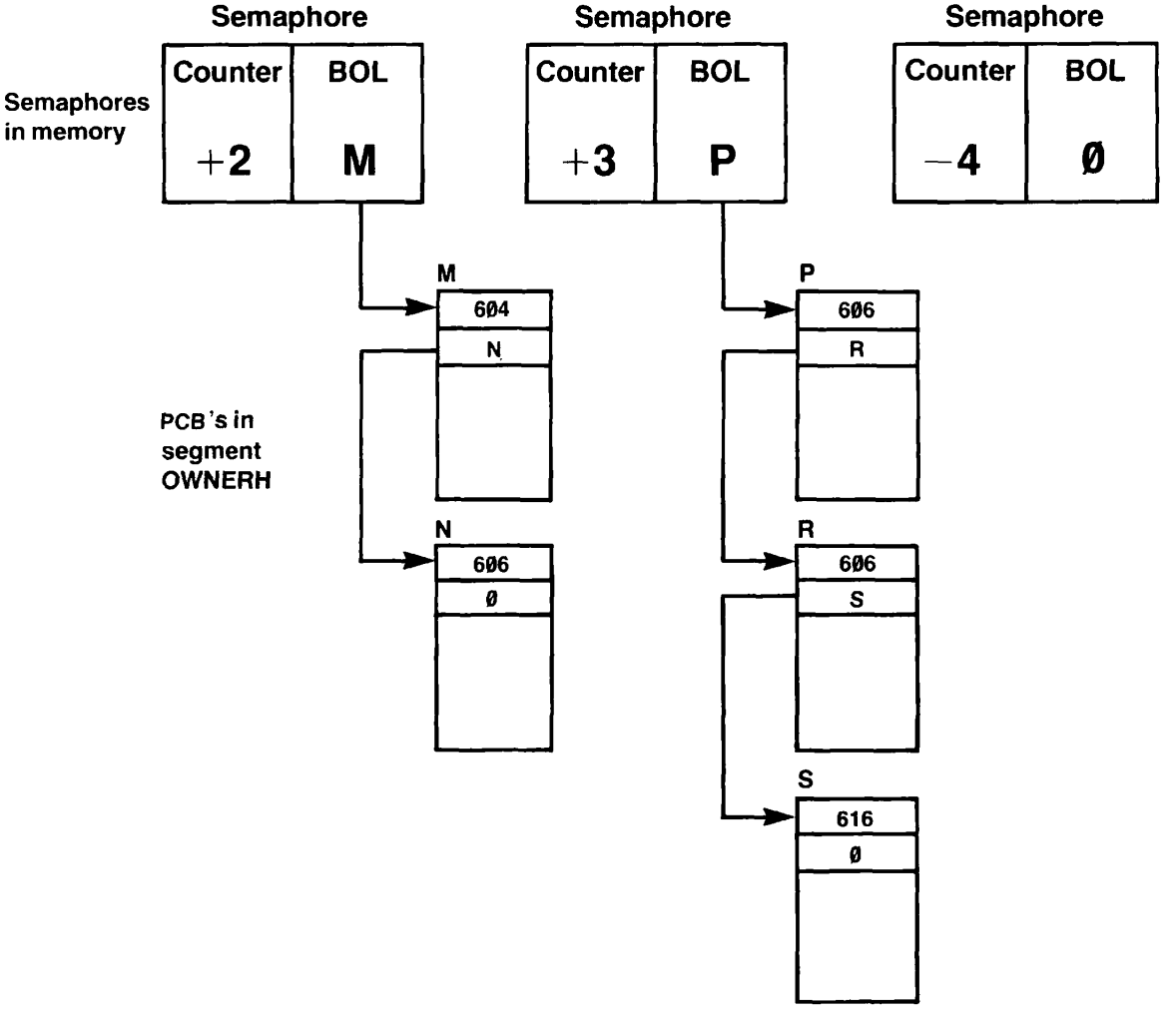
The second location contains the address of the first PCB awaiting completion of the specified event. Since all PCBs are contained in segment OWNERH, a 16-bit pointer is all that is needed to identify a specific PCB.

A semaphore can reside anywhere in memory but segment 0. It does not usually reside in segment OWNERH.

Wait List Data Base

Each wait list has associated with it a linked list of PCBs. The processes represented by the PCBs all share the same semaphore; this means that they are all waiting for the same event to occur.

Note that the PCBs in a wait list need not have the same level of priority, since the wait list uses a priority-based queuing algorithm. This means that processes with higher priorities are queued ahead of those with lower priorities.



Wait List and Associated PCB Lists
Figure 9-2

PXM INSTRUCTIONS

The two notify instructions, NFYE and NFYB, and the wait instruction, WAIT, are restricted instructions. Therefore, they must be executed in Ring 0. All three instructions are 48 bits long: bits 1-16 contain an instruction code, and bits 17-48 contain a 32-bit address pointer to a semaphore.

The WAIT Instruction

Figures 9-3 and 9-4 show the actions of the WAIT instruction.

As the name indicates, WAIT signals the PXM to wait for an event before executing any more of the currently running process. When WAIT executes, the processor uses the address pointer contained in the instruction to reference a semaphore. The processor increments the counter contained in the addressed semaphore, then looks at the result.

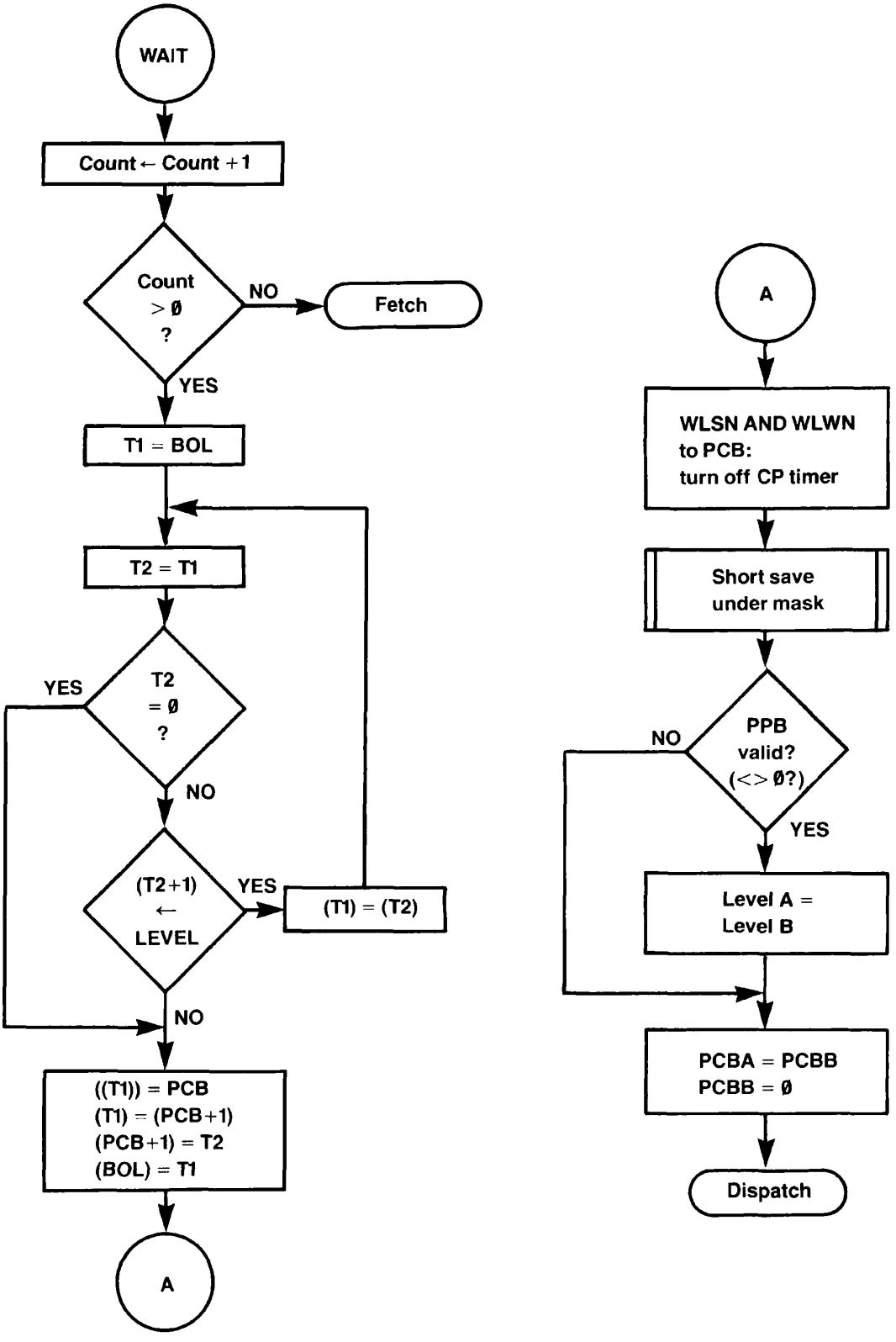
If the result is less than or equal to 0, there are no other processes waiting for the event defined by the semaphore. In this case, the currently executing process can continue.

If the result is greater than 0, either the expected result has not occurred, or the desired resource is not available. The processor stops executing the current process, removes the associated PCB from the ready list, and places the PCB on the wait list associated with the semaphore. The PCB's priority level dictates where on the wait list the PCB should go. If the wait list already contains PCBs with the same priority level, the new PCB is placed after the ones already there.

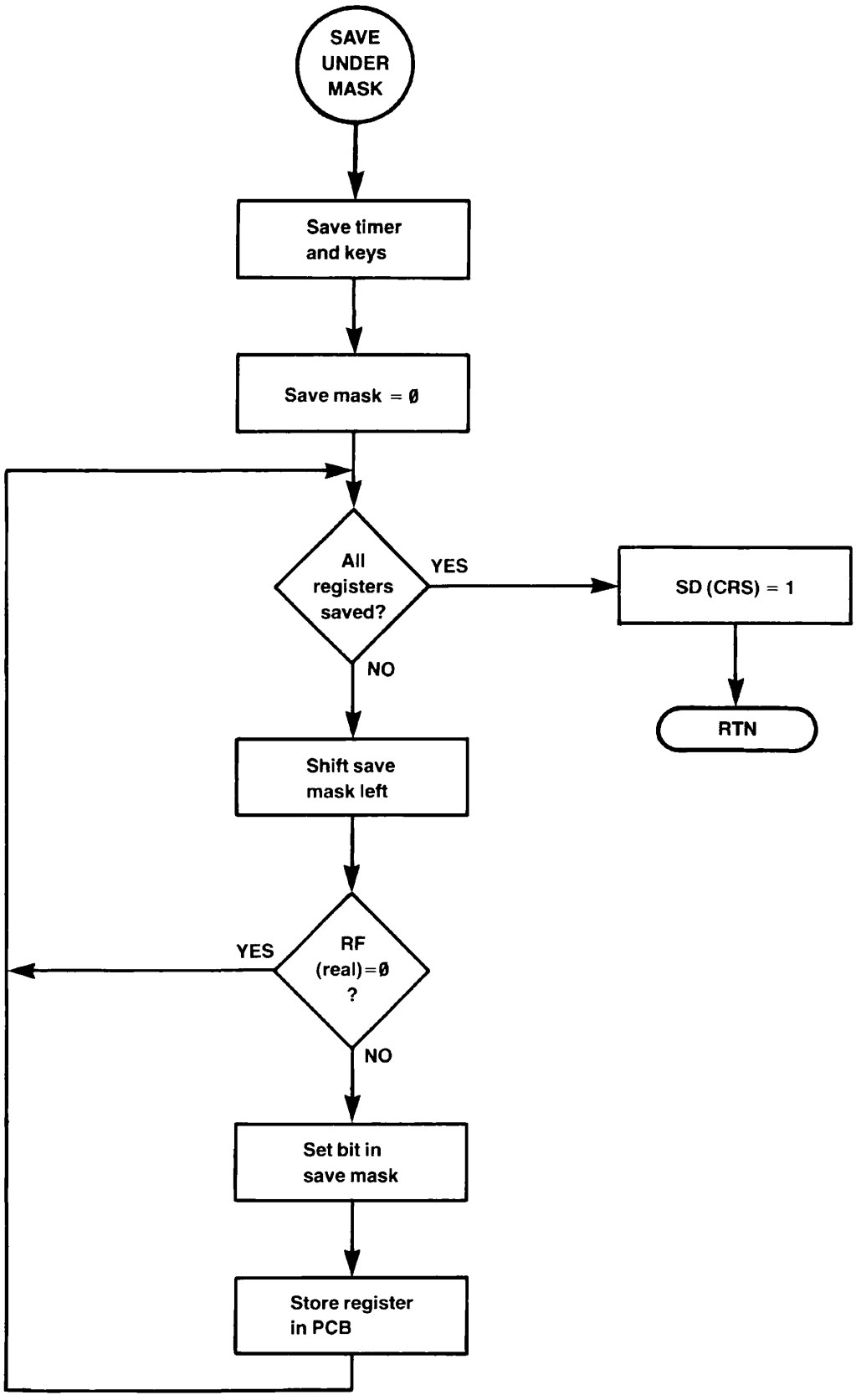
If the result is greater than +32767, a semaphore overflow fault occurs.

Note

The processor saves only the contents of the keys, base registers, and program counter when it adds a PCB to the wait list. It does not save the contents of the general registers or floating registers. After this short save the processor makes the register set used by the exchanged process available to the next process to run. For this reason, never assume that the contents of the general registers after a WAIT instruction executes are the same as they were before WAIT executed.



WAIT Instruction
Figure 9-3



Save Under Mask Algorithm
Figure 9-4

The NOTIFY Instruction

Figure 9-5 shows the actions of NOTIFY.

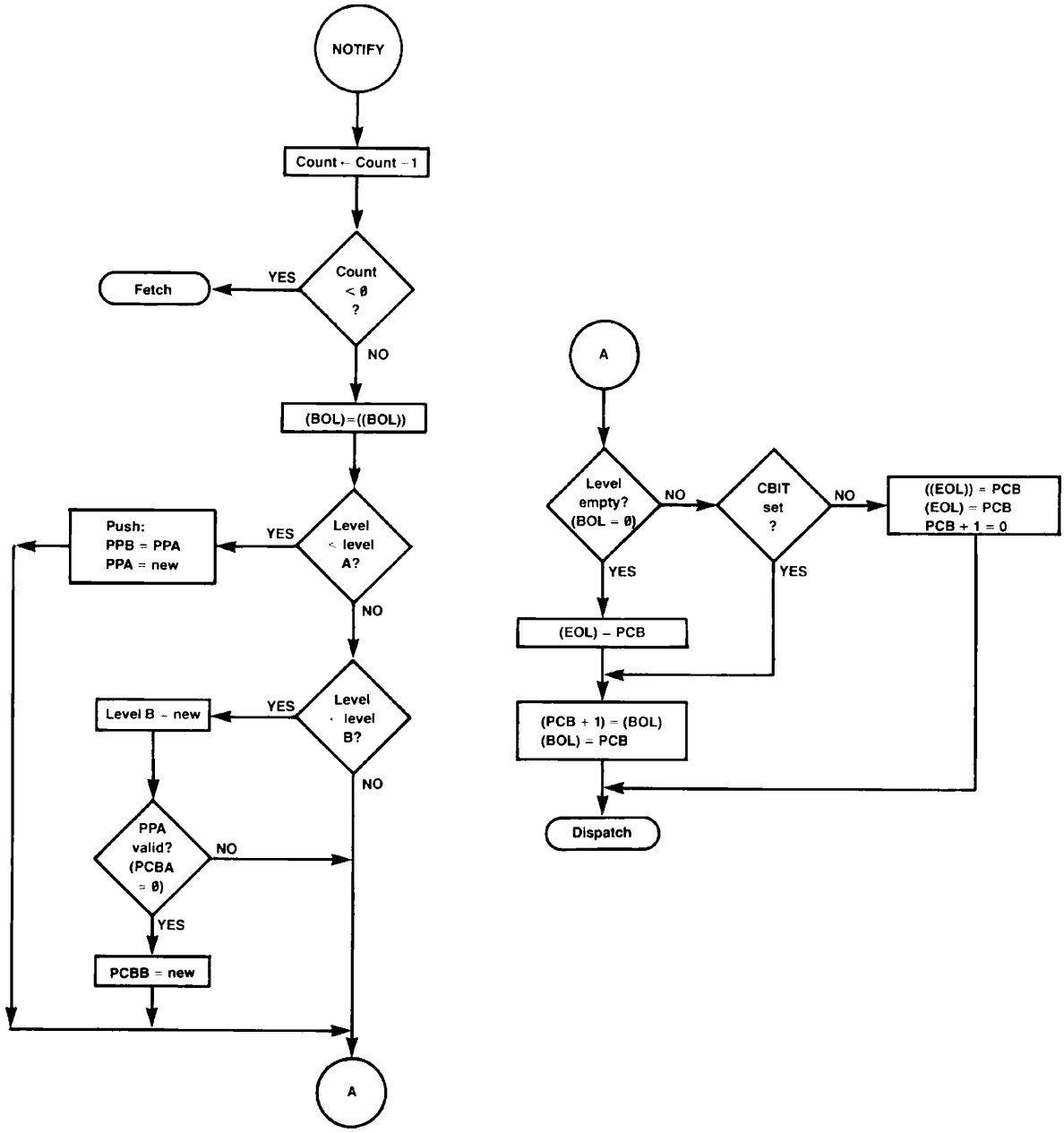
The two notify instructions, NFYE and NFYB, perform the same sequence of events. They differ only in the queuing algorithm used: NFYE queues PCBs at the end of the appropriate ready list priority level, while NFYB queues PCBs at the beginning of the appropriate priority level. In the discussion that follows, NOTIFY encompasses the operation of both instructions.

NOTIFY signals the PXM that some awaited event has occurred. When NOTIFY executes, the processor uses the address pointer contained in the instruction to reference a semaphore. The processor decrements the counter contained in the semaphore by 1 and checks the result.

If the result is less than 0, no process is waiting for this event, so the processor continues the currently executing process. (If the result is less than -32768, a semaphore undeflow fault occurs.)

If the result is greater than or equal to 0, the processor removes the PCB at the head of the specified wait list and places it on the ready list. If the process associated with the PCB moved to the ready list has a higher level of priority than that of the currently executing process, the processor will preempt the current one. However, it does not remove the current process' PCB from the ready list. In addition, the processor saves the contents of the preempted process' registers before starting to execute the new process.

As the above explanation shows, NOTIFY does not always interrupt the currently executing procedure. However, it does always make a change in the specified semaphore.



NOTIFY Instructions
Figure 9-5

DISPATCHER

The operations performed by the PXM are mostly governed by the dispatcher. This microcoded routine is responsible for:

- Deciding which process to run next.
- Assigning that process a register set.
- Managing the register file, including saves and restores.
- Turning the process timer on and off.

The section Dispatcher Operation below, describes the details of the dispatcher's actions.

REGISTER FILES

The 9950 processor contains eight distinct register files. All other 50 Series processors have four register files. Each register file contains 32 32-bit registers that each have a high half and a low half. Table 9-2 shows the allocation of the register files and the absolute memory locations each occupies.

Table 9-2
Register File Allocation

Register File	Absolute Locs	Use
RF0	'0-'37	Microcode scratch and system registers (set 1 for 9950)
RF1	'40-'77	32 DMA channels
RF2	'100-'137	User register set 1
RF3	'140-'177	User register set 2
RF4*	'200- 237	User register set 3
RF5*	'240-'277	User register set 4
RF6*	'300-'337	Microcode scratch and system registers (set 2)
RF7*	'340-'377	Spare register set

*For the 9950 processor only.

Microcode Register Files

RF0 and RF6 are reserved for microcode use. These registers can hold temporary data, control information, or other such items for the microcode to use. Some locations are defined for microdiagnostic use. Table 9-3 defines the locations in the microcode register file.

Table 9-3a
Microcode Register File Set 1, RF0, for the 9950

Loc	Contents	Loc	Contents
0	TR0	20	RMASAVE
1	TR1	21	—
2	TR2	22	PARREG1
3	TR3	23	PARREG2
4	TR4	24	PARREG3
5	TR5	25	PBSAVE
6	TR6	26	SYSREG1
7	TR7	27	DSWPARITY
10	FR032, TR8	30	PSWPB
11	TR9	31	PSWKEYS
12	FR132, TR10	32	PLA, PPA
13	TR11	33	PLB, PPB
14	REOIV, UCSADDR	34	DSWRMA
15	RDSAVE	35	DSWSTAT
16	CFF00, C00FF	36	DSWPB
17	RATMP	37	RSAVPTR

Table 9-3b
Microcode Register File Set 2, RF6, for the 9950

Loc	Contents	Loc	Contents
300	DGR0 (STLBRF1)	320	MINUS1
301	DGR1 (STLBRF2)	321	ONE32
302	DGR2 (RDMX1)	322	KMASK, IUART
303	DGR3	323	C3FF, C3F
304	DGR4	324	C8000
305	DGR5	325	C0D0D, CB0B0L
306	DGR6	326	C9C00, C0080
307	DGR7	327	CB1E0, ---
310	DGR10	330	C6666
311	DGR11	331	C10K, ACK2
312	DGR12	332	FERRET6
313	DGR13	333	FERRET5
314	DGR14	334	FERRET4
315	DGR15	335	FERRET3
316	DGR16	336	FERRET2
317	DGR17	337	FERRET1

Table 9-3c
Microcode Register File, RF0, for All Other 50 Series Systems

Loc	Contents	Loc	Contents
0	TR0	20	ZERO, ONE
1	TR1	21	PBSAVE
2	TR2	22	RDMX3
3	TR3	23	RDMX4
4	TR4	24	C377
5	TR5	25	MINUS1, MINUS2
6	TR6	26	WWADTR
7	TR7	27	DSWPARITY
10	RDMX1	30	PSWPB
11	RDMX2	31	PSWKEYS
12	USCADDR*,REOIV#	32	PPA, PCBA
13	RSGT1	33	PPB, PCBB
14	RSGT2	34	DSWRMA
15	RECC1	35	DSWSTAT
16	RECC2	36	DSWPB
17	——, RATMPL#	37	RSAVPTR

* Used only for the 750 and 850 systems.

The locations for REOIV and RATMPL are switched on the 2250, 250, 400, and 550-II.

DMA Channel Register File

The DMA register file, RF1, contains 32 channel registers. Table 9-4 shows the format of this register file.

Table 9-4
DMA Register File (RF1) Format

Loc	Contents	Loc	Contents
40	DMA cell 00	60	DMA cell 20
41	DMA cell 01	61	DMA cell 21
42	DMA cell 02	62	DMA cell 22
43	DMA cell 03	63	DMA cell 23
44	DMA cell 04	64	DMA cell 24
45	DMA cell 05	65	DMA cell 25
46	DMA cell 06	66	DMA cell 26
47	DMA cell 07	67	DMA cell 27
50	DMA cell 10	70	DMA cell 30
51	DMA cell 11	71	DMA cell 31
52	DMA cell 12	72	DMA cell 32
53	DMA cell 13	73	DMA cell 33
54	DMA cell 14	74	DMA cell 34
55	DMA cell 15	75	DMA cell 35
56	DMA cell 16	76	DMA cell 36
57	DMA cell 17	77	DMA cell 37

User Register Files

Table 9-6 shows the format of the user register files, RF2 through RF5, for V, I, R, and S modes. Table 9-5 defines the terms used in Table 9-6.

Table 9-5
Definition of Register File Terms

Name	Contents	Name	Contents
GR0	General register 0	FACML	Floating accumulator, mantissa low
GR1	General register 1		
GR2	General register 2	FAC	Floating accumulator
GR3	General register 3	PB	Procedure base
GR4	General register 4	SB	Stack base
GR5	General register 5	LB	Linkage base
GR6	General register 6	XB	Temporary base
GR7	General register 7	DTAR3	Descriptor table address, segments 3072-4095
A	Accumulator		
B	Double-precision and long accumulator extension	DTAR2	Segments 2048-3071
		DTAR1	Segments 1024-2047
E	Accumulator extension for MPL,DVL	DTAR0	Segments 0-1023
		KEYS	Keys
S	Stack, alternate index	MODALS	Modals
		OWNER	PCB address of the process that owns the register contents
X	Index		
FAR0	Field address register 0		
FLR0	Field length register 0	FCODE	Fault code
		FADDR	Fault address
FAR1	Field address register 1	FAW #	Fault address word number
FLR1	Field length register 1	CPUT	Process 1024-usec timer
FACMH	Floating accumulator, mantissa high	CLKB	Uses timer uses bits 1-9
FACMM	Floating accumulator, mantissa middle	FACE	Floating accumulator, exponent
L	Double-precision accumulator	Y	Index register

Table 9-6
User Register Files (RF2 through RF5)

Location RF2, RF3, RF4, RF5	V Mode	I Mode	S, R Modes
100, 140, 200, 240	—	GR0	—
101, 141, 201, 241	—	GR1	—
102, 142, 202, 242	L,A,B	GR2	A, B (1,2)
103, 143, 203, 243	E	GR3	—
104, 144, 204, 244	—	GR4	—
105, 145, 205, 245	S,Y	GR5	S (3)
106, 146, 206, 246	—	GR6	—
107, 147, 207, 247	X	GR7	X (0)
110, 150, 210, 250	FAR0	FAR0	(13)
111, 151, 211, 251	FLR0	FLR0	—
112, 152, 212, 252	FAR1, FACMH and FACMM	FAR1, FACMH and FACMM	FAC0 (4,5)
113, 153, 213, 253	FLR1, FACE and FACML	FLR1, FACE and FACML	FAC1 (6)
114, 154, 214, 254	PB	PB	PB
115, 155, 215, 255	SB	SB	SB (14,15)
116, 156, 216, 256	LB	LB	LB (16,17)
117, 157, 217, 257	XB	XB	XB
120, 160, 220, 260	DTAR3	DTAR3	DTAR3 (10)
121, 161, 221, 261	DTAR2	DTAR2	DTAR2
122, 162, 222, 262	DTAR1	DTAR1	DTAR1
123, 163, 223, 263	DTAR0	DTAR0	DTAR0
124, 164, 224, 264	KEYS/MODALS	KEYS/MODALS	KEYS/MODALS
125, 165, 225, 265	OWNER	OWNER	OWNER
126, 166, 226, 266	FCODE	FCODE	FCODE (11)
127, 167, 227, 267	FADDR, FAW#	FADDR	FADDR (12)
130, 170, 230, 270	TIMER	TIMER	TIMER
131, 171, 231, 271	—	—	—
132, 172, 232, 272	—	—	—
133, 173, 233, 273	—	—	—
134, 174, 234, 274	—	—	—
135, 175, 235, 275	—	—	—
136, 176, 236, 276	—	—	—
137, 177, 237, 277	—	—	—

Note

User register sets RF4 and RF5 are for the 9950 only.

The twenty-fifth location in each user register set specifies OWNER, the address of the PCB associated with the process that owns the register set. Note that bits 1-16 of OWNER specify OWNERH, the number of the segment containing the ready list and the PCBs. Make sure that

OWNERH contains the proper value in both user register sets BEFORE entering process exchange mode.

Directly Addressing A Register Set

To address the register file directly, you must use the LDLR/STLR instructions. For more information, refer to the descriptions of LDLR and STLR in Chapters 13 and 14. Some register set locations can be addressed as memory locations in some addressing modes as well. See the Address Traps section in Chapter 3 for more information on this topic.

PROCESS INTERVAL TIMER

The process interval timer is a 48-bit number that represents the time that has passed since this process began executing (or, for system processes, the time since cold start). The timer represents time in units of 1.024 milliseconds. Bits 1-42 of the timer represent the time; bits 43-48 are reserved for future use.

Four PCB locations and two register file locations contain timer information. Table 9-7 describes these locations and their contents.

Table 9-7
Timer Control Words

PCB Loc	Name	Contents
10-11	Elapsed Timer (ETH, ETL)	Total time used by this process in units of 1.024 msecs.
16	Interval Timer High (ITH)	Copy of TIMERH from location 30 in the current register set. This value is the two's complement of the number of 1.024 msec intervals left before the end of the timeslice.
17	Interval Timer Low (ITL)	Bits 1-10 contain a copy of TIMERL from location 30 in the current register set. This value is the amount of process time used in units of one usec. Bits 11-16 are reserved.

The 550-II, the I450, the 850, and the 9950 use a timer accurate to the microsecond. The 2250, the 250-II, and the 750 process interval timer is accurate to the millisecond.

The process timer represents the amount of time that has passed in the current timeslice. The interval timer contained in the register file locations represents the amount of time remaining in this timeslice. Figure 9-6 shows how to use these two values to calculate the time that has passed since the last reset.

```
LDL   ET       /* load L with value of ET
STL   SET      /* save the current value of ET at location SET
LDA   RESET    /* load A with the reset value
RTS            /* reset the timeslice
IMA   CURRTS   /* save the reset value in CURRTS, load A with
               /*    previous reset value
SUB   RESET    /* find difference between new, old reset values
TCA            /* form 2's comp of contents of A
PIDA           /* position for addition
ADL   ET       /* add difference of reset values to contents of ET
SBL   SET      /* subtract old value of ET from contents of L
               /* L now specifies the time that has passed since
               /*   the last timer reset.
```

Timer Example for I450, P850, and 9950
Figure 9-6

The I450, the 550-II, the 850, and the 9950 support two instructons that manipulate the process timer. Table 9-8 describes these instructions.

Table 9-8
Process Timer Instructions

Mnem	Name	Modes	Description
RTS	Reset Timeslice	V	Adds the contents of A, the interval timer, and the elapsed timer and stores the result in the elapsed timer. Loads the contents of A into the interval timer.
STTM	Store Process Time	V	Stores the contents of the process timer into memory.

DISPATCHER OPERATION

As mentioned earlier, the dispatcher governs most of the actions of the PXM. These can be divided into the following steps:

1. Turning off the process interval timer.
2. Choosing a process to run.
3. Selecting a user register set for that process.
4. Turning the process interval timer back on.

The paragraphs below elaborate on each of these steps.

Step 1. Turning off the Process Interval Timer

As soon as the dispatcher begins to execute, it turns off the process interval timer. This timer is located in bits 1-16 (2250, 250-II, 550-II, and 750) or bits 1-26 (I450, 550-II, 850 and 9950) of location '30 in the current register set. It contains a negative number specifying the amount of time left in the current timeslice. On each tick, this negative value is incremented by 1; when the incremented value reaches 0, the dispatcher sets bit 16 in the PCB abort flags to cause a process fault, signalling the end of this timeslice.

Step 2. Choosing the Next Process to Run

PCBA, contained in PPA, holds information about which process the dispatcher should dispatch next. When the dispatcher is first activated, it checks PCBA; if PCBA contains a nonzero value, it specifies a valid PCB and the dispatcher will dispatch the associated process.

If PCBA contains zero, it is invalid and the dispatcher checks PPB for a nonzero value. If PPB is valid, the dispatcher will dispatch that associated process.

If PPB is invalid, the dispatcher must scan the ready list for the PCB of the next process to dispatch. The scan begins at the level specified by Level A in PPA. If the dispatcher finds a PCB, it changes Level A to reflect the level of the found PCB and dispatches that process next. If it finds no PCB, the ready list is empty and the dispatcher idles.

Step 3. Manipulating User Register Sets

Once the dispatcher has identified the next process to dispatch, it must allocate a user register set to the process. Since there are only a finite number of register sets, the dispatcher may have to swap one register set for another; the new process will require a register set other than that used by the last process. Figure 9-7 shows a flowchart of the allocation algorithm the dispatcher uses. The text in this section elaborates on the figure.

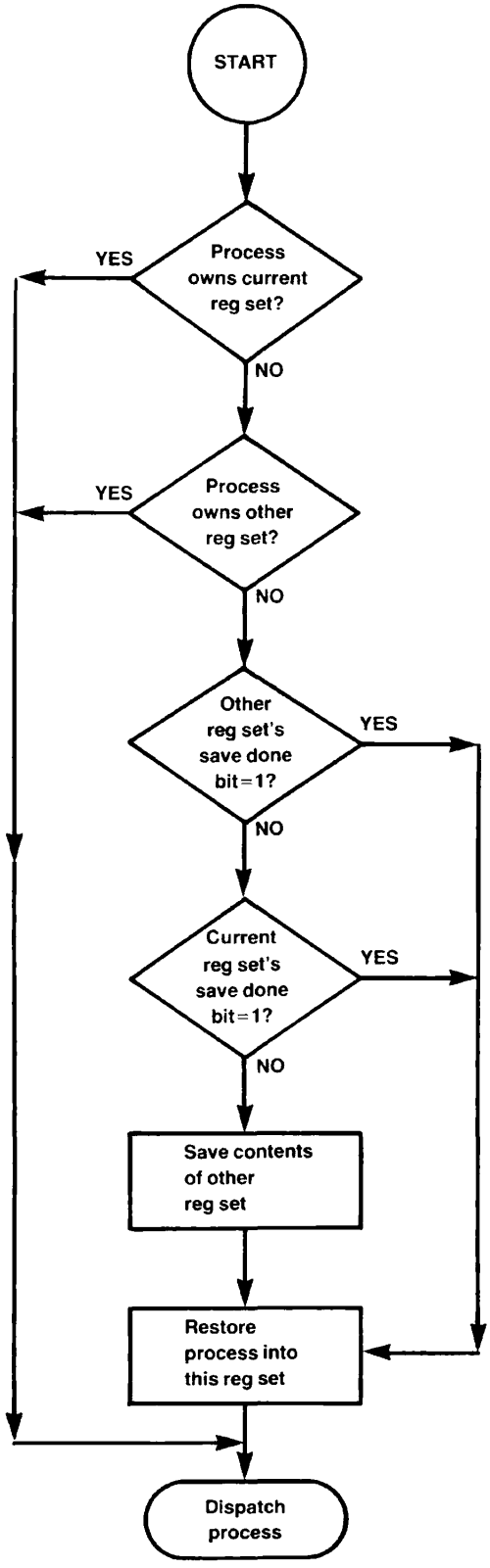
The dispatcher first checks whether the process to be dispatched owns the current register set. It looks at the contents of bits 17-32 of OWNER (location 25 in the current register file). These specify the address of the PCB whose associated process owns that register set. If OWNERL specifies the address of the PCB associated with the next process to run, then this process owns the current register set. The dispatcher makes no changes in the current register set before dispatching the next process.

If OWNERL specifies the address of some other PCB, the next process to be dispatched does not own the current register set. For 50 Series systems, dispatcher makes the other user register set the current register set.

For the 9950, the dispatcher reads the contents of word 5 in the PCB associated with the next process to run to find the number of the register set this process used last. The dispatcher checks OWNER in the register set specified by word 5 to see if the next process to run owns this register set. If it does, the dispatcher must make this register set the current one. Figure 9-8 shows register set allocation on the 9950.

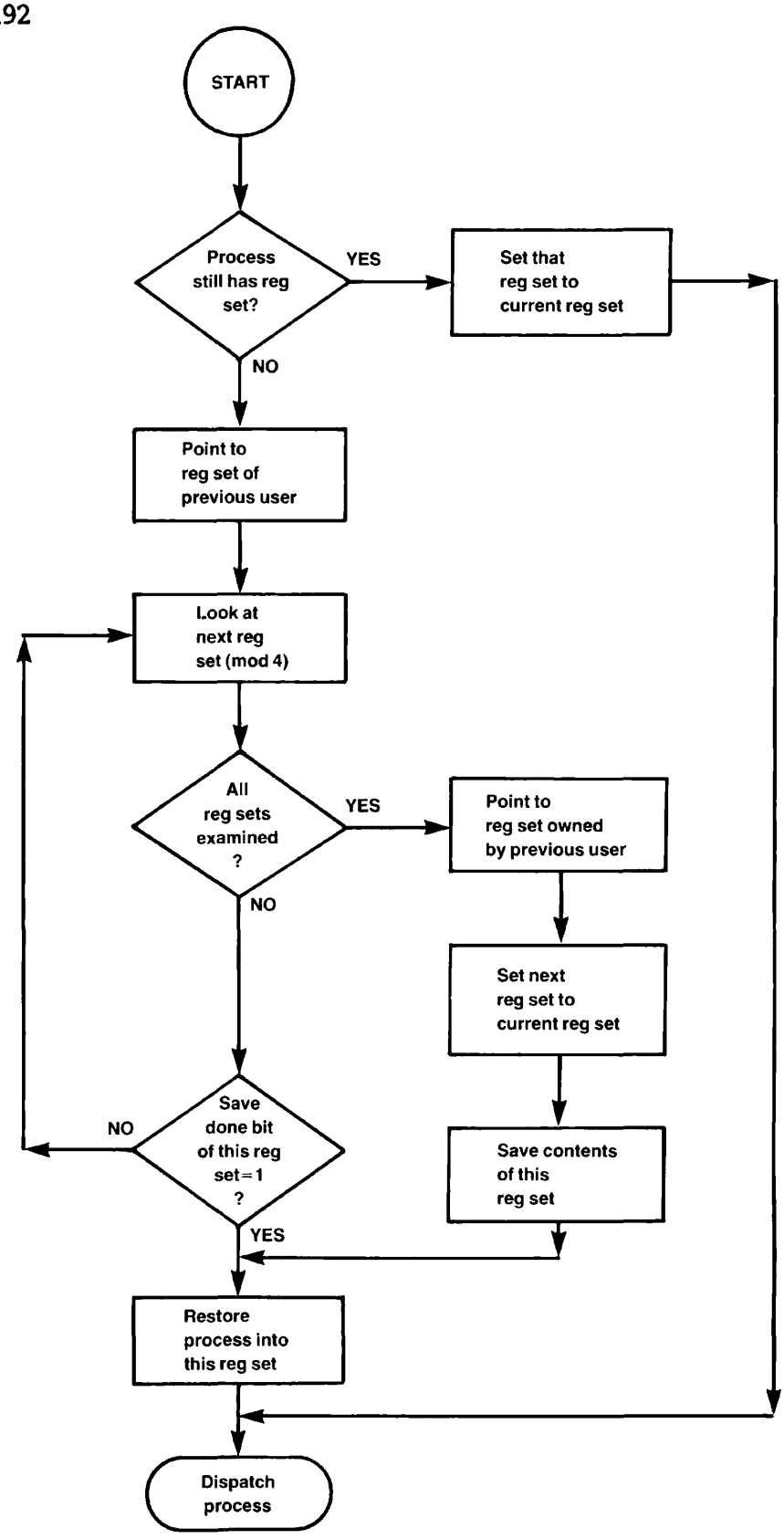
If the next process to run does not own the last register set it used, the dispatcher must choose one for it. It increments the number of the current register set by 1 (modulo 4) to form the number of the new register set, then makes this register set the current one.

3A. The Save Done Bit: In the case where the process does not own the current register set, the dispatcher must load the values of the new process' registers into the current register set. Before it can do this, it must determine whether it must save the old contents of the current register set. Bit 16 of the keys contains the Save Done bit. If this bit contains a 0, the dispatcher must save the old contents of the current register set before restoring the new process to run. After the save, the dispatcher loads the new data into the current register set, resets bits 15-16 of the keys (the In Dispatcher bit and the Save Done bit) to 0, and loads the program counter with the contents of PB.



Register Set Allocation Algorithm
(All Processors Except 9950)

Figure 9-7



Register Set Allocation Algorithm for the 9950

Figure 9-8

If the Save Done bit contains a 1, the old contents of the current register set have been saved in the PCB and register file memory locations, so no further save needs to be done before the new data is loaded. After loading the registers, the dispatcher resets bits 15-16 of the keys and loads the program counter from PB.

3B. Saving the Current Register Set: When the dispatcher must save the current register set before loading in new data, it saves only the registers that contain nonzero values. The contents of these nonzero registers are packed together and loaded into the save area. The save mask determines which registers have had their contents saved and the exact location of those contents in the PCB.

Only the currently active register set contains valid information in the modals field. Whenever the processor switches register sets, the microcode automatically copies the contents of the current modals field into the new register set.

Step 4. Turning On the Process Interval Timer

The last thing the dispatcher must do before dispatching a process is to turn on the process interval timer. The dispatched process begins execution immediately after.

FETCH CYCLE TRAPS

At various points during dispatcher execution, the processor checks for fetch cycle traps, to allow the system to handle external interrupts. For more information about this topic, refer to Chapter 11, Interrupts, Faults, Checks, and Traps.

SUMMARY

This chapter described the actions that occur during process exchange for all single-stream processors. The next chapter describes how the PXM is implemented on the dual-stream 850.

10

Process Exchange on the 850

The previous chapter described process exchange for the single-stream members of the 50 Series family. On the dual-stream 850, however, process exchange is more complex because:

- There are two processing units, the ISUs.
- Two processes can execute at once (one per ISU).
- The two ISUs share one set of PCBs, one ready list, and one set of wait lists.

This chapter elaborates on each of these points. It also describes the elements of the 850 PXM, and describes the actions of the 850 dispatcher.

INSTRUCTION STREAM UNITS

Before beginning this discussion, note the use of two terms. This ISU refers to the ISU on which a process of interest is currently executing. The Other ISU designates the second system ISU. Throughout this discussion, This ISU is assumed to be the master ISU; The Other ISU, the slave ISU.

As mentioned in Chapter 1, the 850 contains two instruction stream units, or ISUs, each of which is equivalent to a 750 CPU. The ISUs operate independent of each other and are capable of performing any task any 750 processor can perform. The one exception is that only one

ISU performs I/O. This means that if This ISU is currently incapable of performing I/O, any process running on it that wants to request I/O service is moved to The Other ISU.

Two Executing Processes

Since there are two ISUs per system, two independent processes can be executing at the same time. These two processes are always the two having the highest level of priority in the entire system. Ensuring that the processes with the highest priority are the ones that are selected to execute makes dual-stream process exchange more complicated than its single-stream complement. It is further complicated by the fact that a process can be locked to one ISU, which means that it can only execute on a particular ISU (such as the backstop or supervisor). See the section, The PX Lock, below, for more information about this topic.

One Set of Process Exchange Data Structures

To aid the ISUs in selecting the highest priority processes, the 850 uses one ready list, one group of wait lists, and one group of PCBs for both ISUs. This means that an ISU has to scan only one list to determine the processes available to execute. It also means the system has to maintain only one set of information, eliminating the need to check and update any duplicates. In addition, it means that a process not locked to one ISU may execute faster, since whichever ISU becomes available first can execute it.

850 PROCESS EXCHANGE ELEMENTS

The data structures of the 850 PXM include:

- PCBs
- Ready list
- Wait lists
- WAIT and NOTIFY instructions
- Dispatcher

Like its single-stream counterpart, the 850 PXM also manipulates the register file and the process interval timer. In addition, the 850 PXM uses the value CPUNUM and the PX lock to facilitate its operations.

The CPUNUM

CPUNUM is a 16-bit number stored in bits 1-16 of location '33 of the current register set. This number distinguishes the two ISUs. CPUNUM contains '41004 to represent the This ISU and '102010 to represent The Other ISU.

The PX Lock

The PX lock ensures that only one ISU at a time has access to and can modify the contents of the process exchange data structures. This lock is a 16-bit number. When the lock contains 0, then either ISU can claim the right to access the structures. When it does not contain 0, the lock contains the same value as CPUNUM; i.e., the ID for one of the ISUs. Only the ISU specified by the lock can access the structures; the second ISU must wait until the first ISU is through its current task before gaining access.

PCBs

The process control block format for the 850 is nearly identical to that of the single-stream PCBs. Only a few locations contain added information, as shown in Table 10-1.

OWNERH (bits 1-16 of location 25 in the current register set) specifies the segment containing all the PCBs. Each PCB contains at least 64 locations and must be aligned on a 128-byte boundary. The starting address of the PCB is also the process ID.

No PCB (or any other data structure the PXM uses) should be contained in locations 0-'37 of a segment. Each addressing mode handles address traps differently; avoiding these locations ensures that all addressing modes handle process exchange in the same way.

Table 10-1
PCB Format for the 850

Section	Word # (octal)	Contents
Control	0	Level pointer to BOL in ready list.
	1	Link pointer to next PCB, or 0.
	2-3	Segment #/offset of the semaphore on whose wait list this process is currently. A segment # of 0 indicates that this PCB is on the ready list.
	4	Abort flags used to generate a process fault when this PCB is dispatched. Bits 1-15: Set by the software. Bit 16: Process interval timer overflow.
	5	Bits 1-4: Temporarily restrict process from running on one of the ISUs: 0000 = no restrictions 0100 = bar from This ISU 1000 = bar from The Other ISU Bit 5: Reserved for future use. Bits 6-7: If 01, this process last ran on This ISU; if 10, The Other ISU. Bit 8: If 0, the registers for this process has not been saved in the PCB. If 1, the registers have been saved in the PCB. Bits 9-11: Indicates which register set this process used last. Uses the same format as the modals CRS field. Bit 12: Reserved for future use. Bit 13-16: Process is locked to: 0000 = neither ISU 0100 = This ISU 1000 = The Other ISU
	6-7	Reserved for future use.
Process State	10-11	Process elapsed timers. This value is added to contents of PCB location '16 to give the number of msec this process has run. RTS can alter this location.
	12-15	DTAR2 and DTAR3. These are never saved, only restored.

Table 10-1 (continued)
PCB Format for the 850

Section	Word # (octal)	Contents
	16	Interval timer, bits 1-16.
	17	Interval timer, bits 17-32.
	20	Save mask. PXM uses this to avoid saving or restoring registers containing zeroes. Format of the word is: 1-8: GR0-GR7 (8 32-bit registers) 9-12: FP0-FP1 (4 32-bit registers) 13-16: base registers (4 32-bit registers PB, SB, LB, XB)
	21	Keys.
	22-61	Storage for nonzero registers. (See mask, above.)
Fault	62-63	Fault vector. Segment #/offset to fault table for Ring 0.
	64-65	Fault vector. Segment #/offset to fault table for Ring 1.
	66-67	Reserved for future use.
	70-71	Fault vector. Segment #/offset to fault table for Ring 3.
	72,73	Fault vector. Segment #/offset to fault table for page fault.
	74-76	Concealed fault stack header (FIRST, NEXT, and LAST pointers).
	77	Reserved.
	100-137	Concealed stack. These words can go anywhere in segment OWNERH; i.e., they do not have to start at location '100. The concealed stack can contain as many frames as desired.

Ready List and Wait Lists

The wait lists used in the 850 are identical to those found in the other 50 Series processors. The ready list is also identical except for the process exchange registers it uses.

Each ISU contains four process exchange registers. Two specify information about the currently running processes, and two specify information about the next processes to run. All four are 32 bits wide.

MY_PPA and OTHER_PPA define either the currently running process, or the process that is about to run. MY_PPA represents this process for This ISU; OTHER_PPA, for The Other ISU. Bits 1-16 of each register contain the process' level of priority; bits 17-32, the starting address of that process' PCB. Bits 1-16 of each register are guaranteed to always point to the ready list priority level that contains the highest priority process that is able to execute for the appropriate ISU.

The MY_PPNEXT register specifies the next process to run on This ISU; OTHER_PPNEXT, for The Other ISU. Like their single-stream counterpart (PPB), bits 1-16 specify the priority level of the next process to run, and bits 17-32 identify the PCB of this process. A nonzero value in bits 1-16 indicates valid contents.

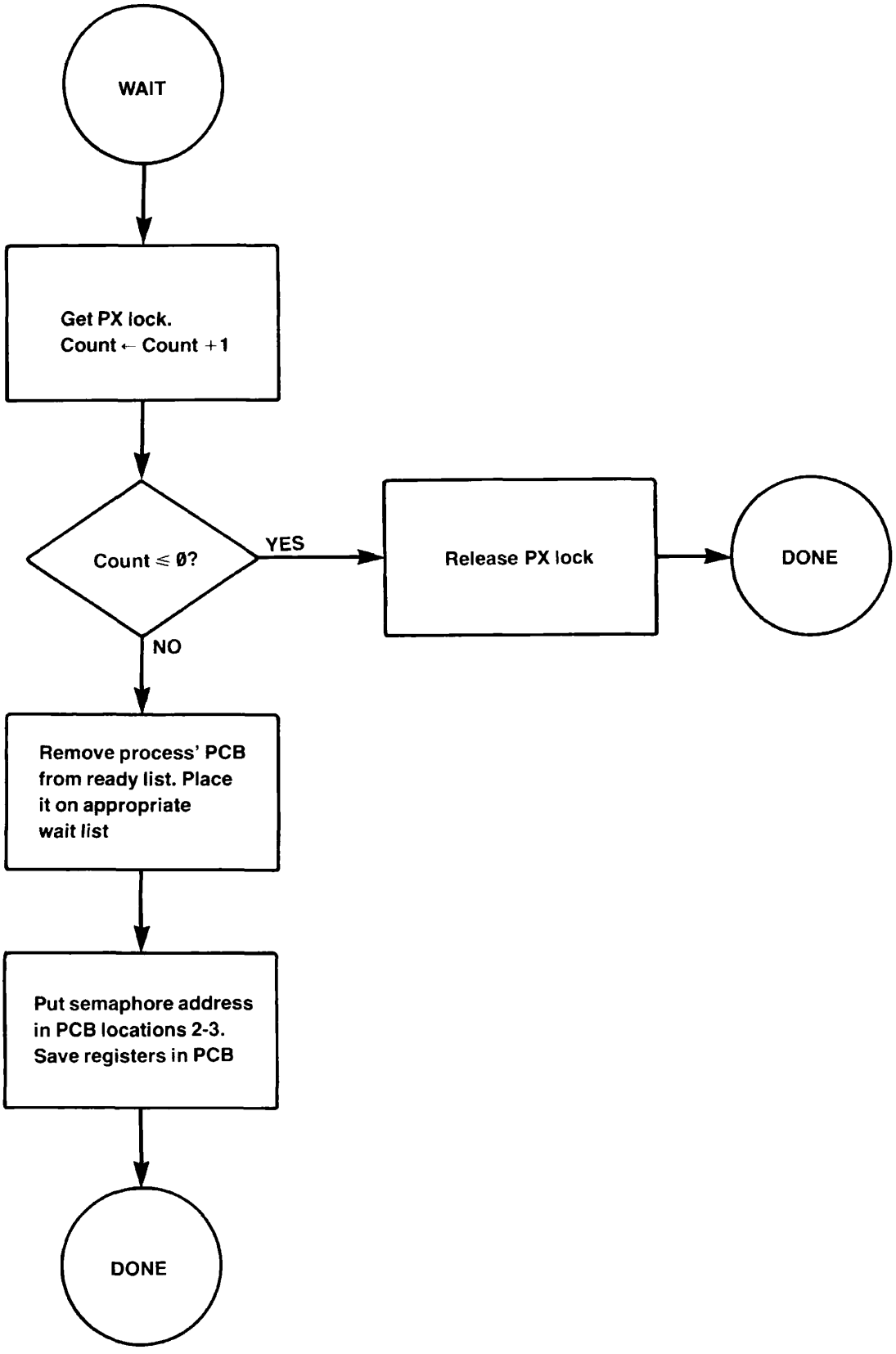
WAIT and NOTIFY Instructions

These instructions perform the same basic functions as their single-stream counterparts. However, their tasks also include obtaining the PX lock and loading the PXM registers with the correct information so that each ISU can correctly determine its own state and that of the second ISU. Figures 10-1 and 10-2, together with the text in this section, give simplified versions of how the 850 WAIT and NOTIFY instructions work.

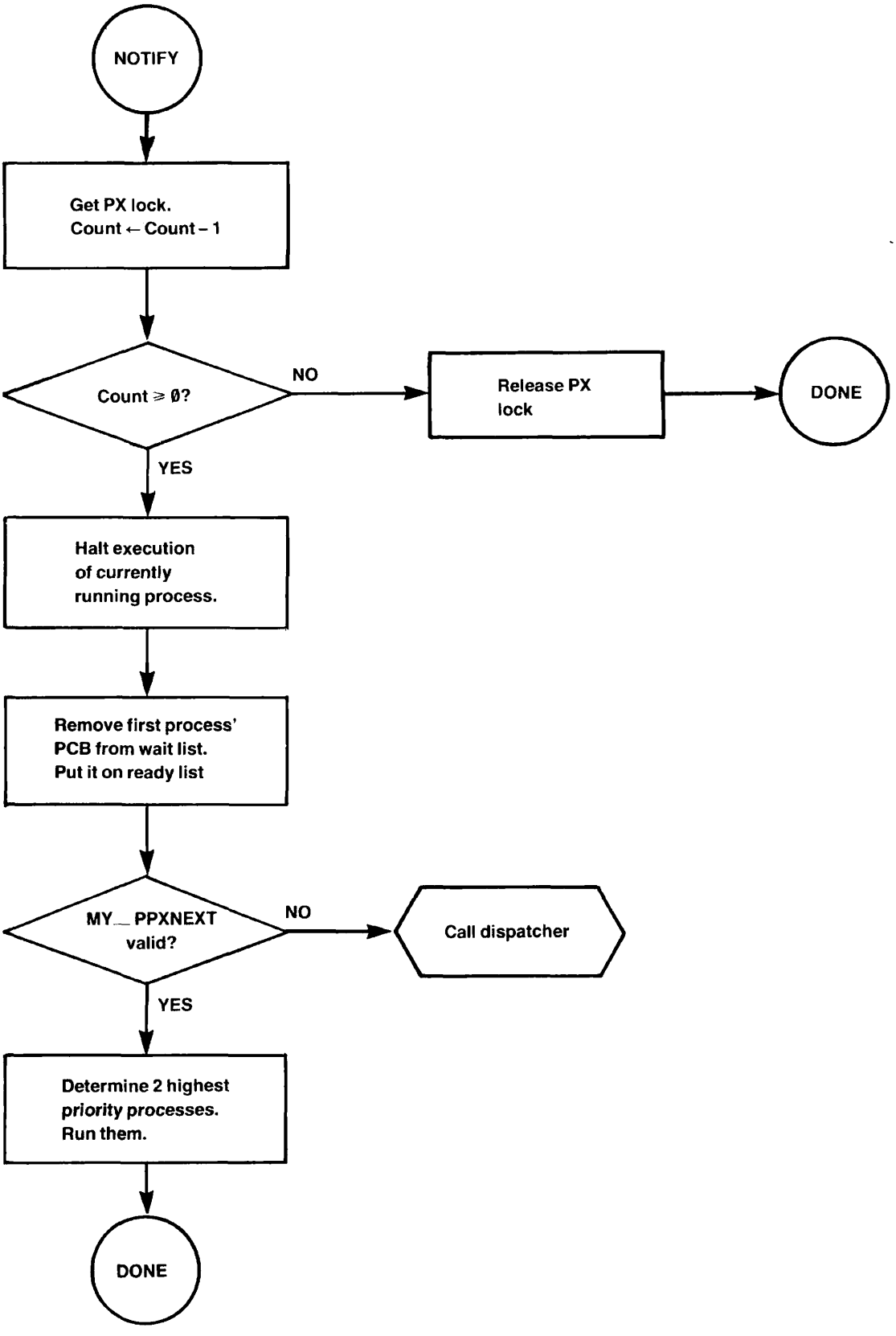
WAIT: WAIT tells the PXM to wait for an event to occur before executing more of the currently active process. The address pointer contained in WAIT specifies a semaphore on which the process is to wait. WAIT obtains the PX lock, then increments the semaphore count by 1.

If the incremented value is less than or equal to 0, WAIT releases the PX lock and performs no other actions. If the incremented value is greater than 0, WAIT removes the process' PCB from the ready list and places it on the appropriate wait list acording to the process' priority. WAIT loads locations 2-3 of the process' PCB with the semaphore address and saves the process' base registers into its PCB.

After the short save, WAIT either runs the next process, if it knows it, or invokes the dispatcher to choose a new process to run.



850 WAIT Instruction
Figure 10-1



The 850 NOTIFY Instruction
Figure 10-2

NOTIFY: The 850 NOTIFY is significantly more complex than the single-stream WAIT. Its purpose is deceptively easy to state: NOTIFY ensures that the two currently running processes in the system are the two highest priority processes that are able to run. To do this, NOTIFY notifies the process that is at the top of the associated semaphore's wait list, then compares the priority level of this process with those of the two processes currently running.

Step 1. Finding a Process to Notify: When it executes a NOTIFY instruction, the PXM first acquires the PX lock. It then uses the pointer contained in the NOTIFY to reference a semaphore and decrement the semaphore count by 1. If the decremented value is less than 0, the PXM releases the PX lock and the NOTIFY is done.

If the decremented value is greater than or equal to 0, then the PXM must notify a process. It ceases to execute the current process and removes the first PCB on the semaphore's wait list. It places the PCB at the beginning or end of the appropriate level of the ready list as indicated by the NOTIFY.

Step 2. Choosing a Process to Run: The PXM must now choose a new process to run. If the contents of MY_PPNEXT are invalid, control transfers to the dispatcher, which determines the next process to run.

If the contents of MY_PPA are valid, the PXM must decide if the process it just notified is of higher priority than either of the processes currently executing. Six cases exist:

A<C and B<C
C<B<A
C<A<B
C<A=B
A<C<B
B<C<A

where A is the process currently running on This ISU, B is the process currently running on The Other ISU, and C is the process that was just notified.

These cases can become quite involved, depending on where each of the three processes can run, and depending on what actions the PXM has taken previously. This discussion will explain two simple examples.

Suppose the first case were true. This means that C has the lowest priority of the three processes and will not be run. All the PXM needs to do is to decide on which ISU C is to be run.

If C can be run on only one ISU and has a higher priority than the process that ISU is to execute next (as specified in that ISU's MY_PPNEXT register), the PXM updates that ISU's MY_PPNEXT register so that it points to C. Therefore, that ISU will execute C next.

If C can be run on either ISU, the PXM updates MY_PPNEXT and OTHER_PPNEXT on both ISUs so that C will execute as soon as either ISU is free.

As another example, suppose case 2, C<B<A, were true. Here C has the highest priority of all, and should run on This ISU, if possible. A simplified algorithm for this case is shown in Figure 10-3.

```
     If C can run on This ISU
     then if A can run on The Other ISU
          then if OTHER_PPNEXT is of lower priority than A
               then invalidate OTHER_PPNEXT
                    set MY_PPA to A
     set PPA to C and go to the dispatcher.

     If C can run on The Other ISU,
***  then if The Other ISU has received the most recent scan
          then send this scan message. Scan identifies C as the
               process to consider running;
     else if priority of process in last scan is greater than C's
          then return;
          If priority of process in last scan is less than C's
          then go to *** above;
          If priority of process in last scan equals C's
          then call dispatcher to scan ready list to
               pick up the process queued first and return.
```

Sample NOTIFY Algorithm
Figure 10-3

Dispatcher

Like its single-stream counterpart, the 850 dispatcher selects the next process to run and sets up the registers and conditions that process needs to run. The section, Dispatcher Operation, below, explains its actions.

Register Sets

Each ISU contains a register file identical to the single-stream register file. Each contains two user register sets designated as the current register set (CRS) and the other register set (ORS). Both of these have the same format as the user register sets on the single-stream processors.

Microsecond Timer

The 850 process timer is accurate to the microsecond. It is contained in two registers, TIMERH and TIMERL. TIMERH contains the two's complement of the millisecond portion of the clock. Bits 1-10 of TIMERL contain the microsecond part. Bits 11-16 of TIMERL are never changed.

Every 1.024 milliseconds the microsecond time overflows, causing a fetch cycle trap. The contents of TIMERH are incremented; when the contents of TIMERH overflow, a process abort fault occurs and stops the current process from being executed.

DISPATCHER OPERATION

When a process completes execution or is aborted, the dispatcher begins to execute to select the next process to run. This discussion assumes that the PX lock contains the number of This ISU, so This ISU has the right to access the PXM data structures.

Step 1. Finding a Process to Run

The dispatcher first checks the contents of MY_PPA. If bits 17-32 are 0, the contents are invalid, as are the contents of PPNEXT. To find the next process to run, the dispatcher scans the ready list beginning at the level specified in bits 1-16 of MY_PPA.

The dispatcher scans the ready list until it finds the first process that is neither locked from This ISU, nor currently running on The Other ISU. Any processes the dispatcher finds during the scan that are temporarily locked from This ISU are unlocked by setting the lock field in the process' PCB location 5 to 0. If the ready list contains no suitable process, the dispatcher releases the PX lock.

If the dispatcher finds a process on the ready list to run, it next checks for two things:

- Does the OTHER_PPNEXT point to this process?
- Has This ISU sent a scan message to The Other ISU suggesting that The Other ISU run this process?

If the OTHER_PPNEXT points to this process, This ISU will not run this process. It will be run at a later date on The Other ISU.

If a scan message was sent, the dispatcher invalidates the message so that The Other ISU will not run this process. Once the message is invalidated, or if no such message was sent, the dispatcher loads MY_PPA with the level and PCB starting address of this process.

Step 2. Locating Register Values and a Register Set

Once MY_PPA contains valid information, the dispatcher must locate the register values this process needs for execution, and must find a register set to contain them. The values can be in one of three places:

- In a register set on This ISU
- In a register set on the Other ISU
- In the process' PCB

The dispatcher checks the CRS field in the process' PCB to see if either of This ISU's register sets or either of The Other ISU's register sets already contain the process' register values. If either of This ISU's register sets do, the dispatcher makes that set the CRS.

If either of The Other ISU's register sets contain the process' values, the dispatcher sends a message to The Other ISU telling it to save the contents of that register set into the process' PCB. The dispatcher then releases the PX lock so that The Other ISU can save the values. After a short time, This ISU regains the PX lock and tries to choose a register set from the beginning.

If none of the register sets on either ISU already contains the process' register values, the dispatcher must load them from the process' PCB. The dispatcher chooses a register set on This ISU by checking the Save Done bit of both the CRS and ORS.

If the Save Done Bit of the CRS contains a 1, the CRS is available. The dispatcher loads the process' values from the PCB into the CRS.

If the CRS is not available, the dispatcher checks the Save Done bit of the ORS. If the Save Done bit contains a 1, the dispatcher makes ORS the CRS, then loads in the process' register values.

If neither the CRS nor the ORS is available (both Save Done bits contain 0), the dispatcher saves the contents of the ORS into the appropriate PCB, makes ORS the CRS, then loads the process' register values into it.

Step 3. Updating Information and Running the New Process

After choosing and loading (if necessary) a register set, the dispatcher loads location 5 of the process' PCB with the ID of the ISU on which the process most recently ran. It also loads the PCB with the location of the process' register values, and sets bits 15-16 of the keys to 0. The dispatcher then releases the PX lock and enables the microsecond timer. The new process begins to execute.

SUMMARY

This chapter explained how the dual-stream 850 processor performs process exchange. The next chapter describes how all members of the 50 Series family handle interrupts, faults, checks, and traps.

11

Interrupts, Faults, Checks, and Traps

Most of the time, the processor executes instructions contained in one process, then goes on to those contained in another. At some point, however, another part of the system may require service; when this happens, the processor has to break the flow of control within the currently running process and service whatever has interrupted. This chapter describes the types of breaks that can occur, and how the 50 Series processors service them.

BREAKS

Breaks in execution can be caused by four events:

- An interrupt
- A fault
- A check
- A trap

The first three types of events are breaks in software execution. The last, the trap, is a break in microcode execution.

The way in which the processor services a break depends on its type and on the current process exchange mode of the machine. When the PXM is disabled, the processor handles all software breaks in the same way. Interrupts, checks, and faults all vector through a dedicated Sector 0 location to reach their handlers.

When the PXM is enabled, the processor handles each software break with a different protocol. Table 11-1 defines the software breaks and briefly describes the protocols that the machine uses to service them.

Microcode breaks are handled differently. When a trap occurs, it may cause a software break, which the processor services to clear the microcode break. If no software break is necessary, the processor handles the microcode break in a fashion transparent to the currently executing process.

Table 11-1
Summary of Software Breaks

Break	Definition	How Serviced
Interrupt	The processor receives a signal from an external device requiring service.	The currently executing software does not usually cause an interrupt. Code especially designed for the purpose services the interrupt outside the context of the currently executing process.
Fault	The currently executing software requires software intervention.	The currently executing software usually handles a fault by mirroring a procedure call to fault code. This code services the fault within the context of the current process.
Check	The processor detects an internal consistency problem requiring software intervention, such as an integrity violation, a reference to a nonexistent memory module, or a power failure.	As with interrupts, code designed especially for the purpose services the check outside the context of the currently executing process.

INTERRUPTS

Interrupts take one of two forms: external interrupts, or memory increment interrupts. (Memory increment interrupts are not supported on the 9950.) As mentioned above, actions depend on whether the PXM is enabled or disabled.

External Interrupts, PXM Disabled

If an external interrupt occurs when the PXM is disabled, the processor uses either the address specified by the controller (vectored interrupt mode) or the contents of location '63 (standard interrupt mode) to reference a vector in segment 0. This vector points to the interrupt response code (IRC), which is also located in segment 0.

To reach the IRC, the processor jumps indirectly through the vector, as if it had executed a JST instruction in 64R mode. The JST saves the current contents of the program counter in the location specified by the vector contents before transferring control to the IRC. IRC execution begins at the location specified by the vector plus 1.

Interrupts are disabled when the IRC begins execution, but all other keys and modals remain unchanged. In vectored mode, the IRC must clear the active interrupt before reenabling interrupts. After the clear, the IRC reenables interrupts, saves the current contents of any register it intends to use, and completes the rest of its operation. When it is done, it transfers control back to the location whose address is contained in the first IRC location.

In standard interrupt mode, only one IRC can execute at a time, so the IRC has nothing to clear or save (other than the contents of any registers it intends to use) before reenabling interrupts. As in vectored mode, the IRC completes the rest of its operation and transfers control back to the location whose address is contained in the first IRC location.

External Interrupts, PXM Enabled

If an external interrupt occurs when the PXM is enabled, the processor uses the address sent by the controller as a 16-bit offset into the interrupt segment, segment 4. The microcode saves the current value of PB and the keys in the phantom code scratch registers — PSWPB and PSWKEYS, respectively — turns off the microsecond timer, and inhibits interrupts. The address mode is then set to 64V, Ring 0 is entered, and interrupts are disabled — i.e., bits 1-16 of PB are set to 4, the keys are set to '14000, and the E (Enable Interrupt) bit of the modals is reset to 0. The IRC in segment 4 (called the immediate IRC, or phantom interrupt code) begins to execute.

Phantom Interrupt Code: Phantom interrupt code gives the processor a chance either to perform a trivial task to service the interrupt, or, as happens most often, merely to notify the real interrupt handler. It is usually only a few instructions long. An example of what the phantom interrupt code might look like is shown in Figure 11-1.

Code Purpose	Code Sequence	Comments
Perform trivial task	STA address	Save A register.
	EIO address	Read a 16-bit quantity from a device.
	ABQ address	Add entry to the bottom of a queue.
	LDA address	Restore A register.
	IRTC	Clear interrupt from I/O bus, enable interrupts, and return to normal execution.
Notify	INBC address	Notify a process, clean up the I/O bus, and enable interrupts.

Sample Phantom Interrupt Code Sequences
Figure 11-1

Some restrictions govern phantom interrupt code. Since it has no PCB that PPA can reference, it does not belong to a process. Also, phantom interrupt code saves only PB and the keys. If another interrupt were to occur before the phantom interrupt code completed service to a previous interrupt, the contents of PSWPB and PSWKEYS would be overwritten, destroying information about the first interrupt. Therefore, interrupts must remain inhibited until the phantom interrupt code completes.

Because of these restrictions, the phantom interrupt code can completely service only very simple interrupts. If more complete service is required, the phantom interrupt code only turns off the controller's interrupt mask, clears the currently active interrupt, and notifies the real interrupt handler.

Returning From an External Interrupt: When the IRC completes, it issues either an interrupt return (IRTN, IRTC) if completely finished or an interrupt notify (INEN, INEC, INBN, INBC) to notify the real interrupt handler. The IRTN restores the keys and PB with the saved contents of PSWKEYS and PSWPB, respectively, and enables interrupts, leaving the machine state as it was before the interrupt. (Restoring

the keys also restores the addressing mode to what it was before the interrupt.) The interrupt notify (INOTIFY) instructions put the machine back to the pre-interrupt state by reloading PB and the keys from PSWPB and PSWKEYS, enabling interrupts, and executing the appropriate notify instruction. This allows the process exchange mechanism to work as if the phantom interrupt code did not happen, returning to the code originally interrupted.

All phantom interrupt code sequences must clean up the I/O bus by issuing a CAI signal before interrupts are reenabled. This can be done by using IRTC, INEC, or INBC instructions as appropriate. Alternatively, a CAI can be issued in the IRC before exiting phantom interrupt code through an IRTN, INEN, or INBN instruction as appropriate.

Memory Increment Interrupt

Service for this interrupt is always the same, regardless of the process exchange mode. The processor uses the address supplied by the controller as a 17-bit offset into either of the I/O segments, 0 or 1 (if in mapped I/O). This offset addresses a halfword whose contents the processor increments by 1. If the incremented value does not equal 0, the processor does nothing more and returns.

If the incremented value does equal 0, the processor generates an end-of-range (EOR) signal on the I/O bus and returns. The requesting device typically generates an external interrupt when the EOR is generated.

Unlike the external interrupt, the memory increment interrupt cannot be masked out and can occur at any fetch cycle break.

Note that memory increment interrupts are not supported on the 9950.

Returning From a Memory Increment Interrupt: While the PXM mode does not affect service of this interrupt, it does determine where the processor returns to upon service completion. When the PXM is enabled, the processor returns to the fetch cycle or the dispatcher, depending on where the interrupt occurred. In the case of the dispatcher, the processor always returns to the top of the dispatcher and does not change the PB or KEYS.

When the PXM is disabled, the processor always returns to the fetch cycle.

FAULTS

Faults occur when software tries to perform an action that cannot complete without special help. Examples of faults are page faults (where a reference is made to a page not currently loaded in physical memory) and stack overflow or underflow. In all, there are eleven classes of faults that can occur. Table 11-2 summarizes these classes and their subdivisions, and shows the corresponding fault that occurs on the Prime 300.

Table 11-2
Fault Classes

Fault	50 Series Systems	Prime 300
RXM	Restrict mode violation	Same
Process	Abort flags word does not equal 0 in PCB on dispatch	n/a
Page	Page fault (page not in memory)	Same
SVC	Supervisor call (superceded by direct entry calls)	Supervisor call
UII	Unimplemented instruction	Same
ILL	Illegal instruction	Same
Semaphore (9950 only)	Semaphore overflow or underflow	n/a
Access	Violation of segment access rights	Page write violation
Arithmetic	All FLEX, DEX, and IEX (arithmetic exceptions)	FLEX
Stack	Stack overflow/underflow	Procedure stack (S reg) underflow
Segment	1: Segment # too big (SDT too short)	n/a
	2: Missing segment (SDW fault bit set)	n/a
Pointer	Fault bit in pointer set	n/a

Fault Handler

The software routine that services faults is called the fault handler. It is made up of two parts: a group of entrances (one entrance for each type of fault) and a common fault routine. When a fault occurs, execution begins at the entrance for that fault type. The entrance microcode sets up conditions applicable to the fault, then transfers control to the common handler. This arrangement provides service for several types of faults while avoiding the expense of many different handlers.

There are four elements in the fault mechanism:

- Four fault vectors
- Four fault tables
- The Call Fault Handler (CALF) instruction
- The concealed stack

The microcode routine uses these four elements to convert faults into procedure calls to the various service routines.

Fault Vectors

The fault vectors occupy locations '62-'65 and '70-'73 in the PCB. Each vector contains the address of a fault table. (See Fault Tables, below.) The format of the vectors is identical to that of a 32-bit indirect pointer, as shown in Figure 3-3 in Chapter 3.

The vectors provide a choice of how to handle a particular fault. For example, one process may need to have Ring 0 service a pointer fault, while another process defines its own routines in the current ring to do the service. Since the vectors are located in the process' PCB, different vectors can be specified for processes that need different service. Table 11-3 describes the PCB locations that contain the vectors.

Table 11-3
PCB Fault Vector Locations

PCB Loc	Contents
'62-'63	Ring 0 fault vector
'64-'65	Ring 1 fault vector
'70-'71	Ring 3 fault vector
'72-'73	Page fault fault vector

A separate vector is devoted to page faults, even though page faults require Ring 0 service. This allows a system to specify a universal page fault handler to handle all page faults that occur within the system. If a system uses a universal page fault routine, make sure that all page fault vectors for processes currently within the system contain the address of this universal routine, rather than some other Ring 0 routine.

When a fault occurs, the program counter is loaded with the fault vector in the PCB, including the ring number. This means that fault code is not automatically executed in either Ring 0 or the current ring: the code in the fault tables may either weaken the ring or go through a gate to strengthen the ring.

Fault Tables

Each fault vector points to a fault table. Each table contains 11 8-byte entries, each entry corresponding to one of the types of faults. Table 11-4 lists information about the fault table.

The fault table for page faults must always be located in physical memory. A page fault must never result in an unresolved chain of page faults. For these reasons, the fault table for Ring 0 must exist in a defined segment. If it does not, it is possible to have an infinite number of segment faults occurring recursively, since the Ring 0 fault table for each fault never references a valid segment.

Table 11-4
Fault Information

Fault	Num	Offset	Vector Loc	FCODEH	FADDR	Ring	Saved PB
RXM	0	0	'62	0	Adr	Cur	Backed
Process	1	4	'63	ABFLAGS	—	0	Cur
Page	2	'10	'64	0	Adr	0	Backed
SVC	3	'14	'65	0	—	Cur	Cur
UII	4	'20	'66	Cur RPL	Adr	Cur	Backed
Semaphore (for 9950 only)	5	'24	'67	Underflow $0; Overflow $1	Adr of Semaphore	0	Backed
ILL	10	'40	'72	Cur RPL	Adr	Cur	Backed
Access	11	'44	'73	0	Adr	0	Backed
Arith.	12	'50	'74	See Table 11-10	Adr	Cur	Cur
Stack	13	'54	'75	0	Adr	Cur	Backed
Segment	14	'60	'76	DTAR: 1; SDW: 2	Adr	0	Backed
Pointer	15	'64	'77	PCL: '100000; Else the indirect adr of faulting pointer	Pt Adr	Cur	Backed

The CALF Instruction

Each entry in the fault table can contain any type of instruction, but usually the instruction is either a HLT or a CALF instruction. When the entry contains a HLT, the machine stops every time the fault corresponding to that entry occurs.

When the fault table entry contains a CALF instruction, the format of the entry is as shown in Table 11-5. Note that bytes 3-6 of CALF contain a pointer to the ECB of a fault routine. CALF uses this pointer to transfer control to the fault routine as if the transfer were a normal procedure call. The advantages of this are described in Servicing a Fault, below.

Table 11-5
Format of Fault Table Entries

Byte	Contents
1-6	CALF instruction. Bytes 3-6 contain a pointer to the ECB of a software fault handler.
7-8	Reserved.

CALF performs a normal procedure call where no arguments are expected by the callee. If the callee's ECB specifies arguments, then dummy arguments are substituted and loaded into the stack frame. See Chapter 8 for information about dummy arguments.

The rest of this section describes how a fault is handled if the associated fault vector contains a CALF instruction.

The Concealed Stack

When a fault occurs, the state of the system at the time of the fault must be saved before the fault can be serviced. The processor uses the concealed stack to save information about the system state at the time of a fault.

Information is stored in the concealed stack in frames. Each frame contains 12 bytes of information, as shown in Table 11-6.

Table 11-6
Concealed Stack Frame Format

Word	Contents
0-1	Program counter (segment #/offset)
2	Keys
3	Fault code
4-5	Fault address (segment #/offset)

Six bytes of the PCB keep track of the concealed stack frames. These bytes contain the addresses of the first, last and next available frames in the concealed stack. Table 11-7 describes these locations.

Table 11-7
Contents of PCB Concealed Stack Locations

Loc	Name	Description
'74	FIRST	Pointer to the first frame in the concealed stack.
'75	NEXT	Pointer to the next frame to be used.
'76	LAST	Pointer to the last frame in the concealed stack.
'77+	——	Up to 6 6-word concealed stack frames.

The processor uses a separate stack for faults in order to simplify handling chains of faults. Frequently the CALF instruction for one fault can generate another fault, such as a segment fault, when it tries to call the fault handler. The CALF for this fault may in turn cause another fault, and so on. Instead of using the current segment's stack to contain the information about all of these faults, the concealed stack is used. Since the concealed stack is located in the PCB, the fault handler can easily access it, and there is no danger of using data from anything other than a fault frame.

Note that if a chain of faults occurs, the processor services them in reverse order: the last fault to occur is the first to be serviced.

The concealed stack can accommodate a chain of up to n faults, (n = 6 in PRIMOS), one fault per concealed stack frame. Make sure that the concealed stack contains enough frames to allow for the longest chain of faults that can occur. Since the concealed stack is circular, if one more fault occurs than there are concealed stack frames, the frame for the latest fault will overwrite that of the first fault. For example, suppose the concealed stack contains only four frames, and the chain of faults that occurs is:

```
pointer (link) fault->segment fault->stack fault->segment->page fault
```

The frame for the page fault overwrites that of the link fault frame. The concealed stack no longer contains the proper information about the link fault frame, so the link fault will never be serviced.

Servicing a Fault

As with interrupts, the type of fault service that the processor performs depends on whether the PXM is enabled or not. If the PXM is disabled, it handles all faults in the same way. It saves the contents of the program counter, disables interrupts for one instruction (if the fault is ultimately to be serviced by a Ring 0 handler only) and jumps indirectly (JST) through a fault vector to the appropriate handler.

If the PXM is enabled, the processor must perform a more complex routine:

1. Set up a concealed stack frame.
2. Change the addressing mode to 64V.
3. Select a fault vector.
4. Set PB so that it points to the proper fault table entry.

When a fault occurs, the processor identifies the fault's type by indexing into the fault table. (See Table 11-4.) After identifying the type of fault, the processor uses NEXT to load the next available concealed stack frame with information about the fault. It updates NEXT to point to the next available frame, then sets the machine addressing mode to 64V (if necessary), and references the appropriate fault vector.

The fault vector contains the starting address of a fault table. The processor adds the offset corresponding to the type of fault to this starting address to form the address of a table entry. This entry contains a CALF instruction that points to the ECB of a fault routine. If the fault is ultimately to be serviced by a Ring 0 fault handler, interrupts are disabled for one instruction to allow the CALF instruction to execute. If a handler in another ring is to service the fault, no such interrupt disable occurs.

When the CALF instruction begins to execute, it allocates a stack frame on the current segment's stack and loads it with the information shown in Table 11-8. Note that CALF gets much of this information from the current concealed stack frame.

After loading the concealed stack frame into the current segment's new procedure stack, CALF pops the most recent frame from the concealed stack and sets the flag word to 1. Control is transferred to the entrance specified in the ECB.

Table 11-8
Format of CALF Stack Frame

Word	Contents
0	Flag bits. CALF sets this word to 1.
1	Stack root segment number.
2-3	Return pointer. This is the value of PB found in the current concealed stack frame.
4-5	SB. This value is unchanged.
6-7	LB. This value is unchanged.
8	Keys. This is the value of the keys found in the current concealed stack frame.
9	Address of the location following the call.
10	Fault code.
11-12	Fault address.
13-15	Reserved.

When the handler completes, the PRTN instruction transfers control to the location specified in words 2-3 in the current segment stack frame. Note that words 2-3 contain the saved PB value shown in Table 11-4. This value and the type of fault that occurred determine the actions of the processor after it completes fault service. (For example, the processor might retry the instruction that caused the fault.)

The ECB specified by the stack frame in the current segment's stack must not specify any arguments. It can be a gate or not.

Summary of Fault Classes

Table 11-4 listed the eleven types of faults. Table 11-9 briefly describes what causes each type.

Table 11-9
Summary of Fault Classes

Fault	Cause	Source of Fault
RXM	Non-ring 0 process tries to execute a restricted instruction when restricted mode is enabled.	Hardware; from microcode independent action code.
Process	Word 4 in the PCB does not contain 0 upon dispatch.	Dispatch microcode test.
Page	Reference made to page with missing bit reset to 0. This usually indicates that the page is not in physical memory.	STLB update microcode test.
UII	Processor tries to execute an instruction that is not implemented on this machine.	Decode net or microcode branch.
Semaphore (9950 only)	A semaphore has either overflowed due to too many notifies, or has underflowed due to too many waits.	NOTIFY or WAIT microcode.
ILL	Processor tries to execute an illegal instruction.	Decode net or microcode branch.
Access	Reference made to a segment without the proper access rights.	STLB update microcode test.
Arithmetic	Integer, decimal, or floating-point exceptions.	If IEX, hardware; if not, explicit microcode test.
Stack	Stack overflow or underflow has occurred.	PCL microcode.
Segment	Either the specified segment number is too big, or the segment is missing.	STLB update microcode test.
Pointer	The fault bit in the specified pointer is 1 indicating an invalid pointer.	IP processing in ARGT fetch microcode.

Arithmetic Exceptions

The arithmetic exceptions (integer, floating-point, and decimal) require more explanation than is given in Table 11-9. These three exceptions determine what type of action occurs when an arithmetic overflow, divide by zero, or other such condition exists. Three bits in the keys select what action should occur:

- Bit 7 in the keys specifies the action that is to occur if a floating-point exception occurs.
- Bit 8 determines the action that should follow an integer exception.
- Bit 11 determines the action that should follow a decimal exception.

When any of these exceptions occur, the processor checks the value of the corresponding bit in the keys. In the case of integer and decimal exceptions, a 0 in the corresponding bit tells the processor only to set CBIT to 1. When the corresponding bit in the keys contains a 1, the processor not only sets CBIT to 1, but also loads three registers — FCODEL, FCODEH, and FADDR — with appropriate values, and services the fault.

The processor takes the same actions when a floating-point exception occurs, except that when bit 7 in the keys contains a 1, the processor only sets CBIT to 1. When bit 7 contains a 0, the processor both sets CBIT to 1 and services the exception.

FADDR, FCODEH, and FCODEL are located in the user register file. When the processor loads these registers, FCODEL always contains a '50, which indicates that an arithmetic fault has occurred. FCODEH contains a code that identifies the specific exception that has occurred. FADDR contains a pointer to the instruction that caused the exception, a pointer to the address used by the faulting instruction, or 0. Table 11-10 lists the codes and the faults they indicate.

On all 50 Series processors except the 9950, 850, and 750, when an integer overflow exception occurs, the resulting fault takes place before the next instruction is started. The program counter points to the next instruction suitable for execution. If, however, an ECCC check becomes pending at the same time as the integer exception, that integer exception will be lost. On a 750 or 850, from 1 to 4 instructions are executed before the integer overflow occurs, except in the case of divide by zero, which always points to the next instruction.

Table 11-10
Arithmetic Exception Codes

Data Type	Exception Type	FCODEH	FADDR
Single precision floating-point	Exponent overflow	$100	Address of faulting instruction
	Divide by 0	$101	Address of faulting instruction
	Store exception on FST instruction	$102	Memory address used by FST
	INT exception	$103	Address of faulting instruction
	Intrinsic function exception	$500	Address of faulting instruction
Double precision floating-point	Overflow or underflow	$200	Address of faulting instruction
	Divide by 0	$201	Address of faulting instruction
	Intrinsic function exception	$600	Address of faulting instruction
Integer	Integer overflow	$300	0
	Divide by 0	$301	Address of faulting instruction
Decimal	Decimal overflow	$700 $704*	Address of faulting instruction
	Divide by 0	$701	Address of faulting instruction
	Conversion exception	$702	Address of faulting instruction

*For 2250, 250, 150, 250-II, 450, and I450-II.

CHECKS

The last section described how problems in a process or procedure cause faults. When problems arise with the state of the system itself, a check occurs. These problems may not be visible to the currently executing procedure, or they may be serious enough to terminate the entire system's operation. There are four types of checks:

- Power failure
- Environment (9950 only)
- Memory parity error
- Machine check
- Missing memory module

The power supply for the system initiates a power failure check when AC power fails. The check indicates that 20 milliseconds of DC power remains before all power is gone.

Environmental checks were described in Chapter 1 and apply only to the 9950. They include UPS battery, cabinet or processor board overtemperature, and air flow failure. Environmental checks use the same check vector and DSWSTAT as power failures. In addition, each environmental check has its own check code:

Processor board temperature	$01
Cabinet temperature	$02
Air flow	$04
UPS battery	$08

An environmental check code is stored in register 26L (CHKREG). This register is valid only after a processor check has been issued. (All other checks store 0 in this register.)

The memory error checking logic issues a memory parity error check when it detects a memory parity error or an uncorrected error correction code (ECCU) error. The CPU issues a machine check when it detects an internal parity error. The MCU initiates a missing memory module check when a program tries to access nonexistent physical memory.

Check Handler

Like the fault handler, the check handler is made up of a group of entrances, one for each type of check, and a common check routine. To service checks, it uses a check header, check vectors, a diagnostic status word, and the MCM field of the modals.

Check Header: The 50 Series processors use an eight-word save area in memory to contain information about the system. This check header is located in segment 4 (the interrupt segment). Table 11-11 shows the format of the check block.

Table 11-11
Check Header Format

Word	Contents
0-1	PBH, PBL
2-3	KEYSH, KEYSL (modals)
4-7	Software code (possibly a JST instruction)

Check Vectors: Segment 0 locations starting at '200 can contain the four check vectors. Check vectors are 16-bit indirect pointers with the format shown in Figure 3-4. The 50 Series processors use these vectors in check handling only when PXM mode is disabled.

Diagnostic Status Word: The 50 Series processors also use a group of 32-bit registers collectively called the diagnostic status word (DSW). The check handler uses the DSW as a source of information about the system as it was when the check occurred. The format of the each DSW register is shown in the following tables:

DSW Register	System	Table
DSWPARITY	9950	11-12
DSWPARITY	750 and 850	11-13
DSWSTAT	9950	11-14
DSWSTAT	Rest of 50 Series	11-15
DSWRMA	All 50 Series	11-16
DSWPB	All 50 Series	11-16

Note

DSWPARITY is used only with 9950, 750, and 850 systems.

Table 11-12
Format of DSWPARITY Register for 9950

Bits	Name	Description
1	RCCPER	If 1, the control store detected an RCC parity error. Sets bits 3-8 of DSWPARITY to reflect the state of parity error: Bits 3-5: encoding of RCC parity error bits 1-8 Bit 6: logical OR of RCC parity bits 1-8 Bit 7: RCC parity error bit 9 Bit 8: 0
2	IOPER	If 1, the control store detected an I/O parity error. Sets bits 3-8 of DSWPARITY to reflect the state of the I/O parity error: Bit 3: error is in left byte of either BPA or BPD Bit 4: error is in right byte of either BPA or BPD Bit 5: CPU detected a parity error on BPD Bit 6: CPU detected a parity error on BPA Bit 7: controller detected a parity error on BPD Bit 8: controller detected a parity error on BPA
3-8	Parity Error Code	Specifies information about the RCC or or I/O parity error that occurred. See bits 1 and 2 above for specifics.
9	—	Currently unused.
10	BBH Left Byte Parity Error	If 1, the E1 board detected a parity error on BBH, left byte.
11	BBH right Byte Parity Error	If 1, the E1 board detected a parity error on BBH, right byte.
12	BBL Left Byte Parity Error	If 1, the E1 board detected a parity error on BBL, left byte.
13	BBL Right Byte Parity Error	If 1, the E1 board detected a parity error on BBL, right byte.
14	BAH Parity Error	If 1, the E1 board detected a parity error on BAH.
15	BAL Parity Error	If 1, the E1 board detected a parity error on BAL.
16	BAE Parity Error	If 1, the E1 board detected a parity error on BAE.

Table 11-12 (continued)
Format of DSWPARITY Register for 9950

Bits	Name	Description
17	BD Parity Error	If 1, the memory control unit detected a parity error on BD. Sets bits 20-23 to reflect the error's location. Bit 20: BDH, left byte Bit 21: BDH, right byte Bit 22: BDL, left byte Bit 23: BDL, right byte
18	Memory Data Parity Error	If 1, the memory control unit detected a latched memory data error. Sets bits 20-23 of DSWPARITY to reflect the error's location: Bit 20: LMDH, left byte Bit 21: LMDH, right byte Bit 22: LMDL, left byte Bit 23: LMDL, right byte
19	Memory Address Parity Error	If 1, the memory control unit detected a latched memory address error. Sets bits 20-23 of DSWPARITY to reflect the error's location: Bit 20: MCADDR, high byte Bit 21: MCADDR, low byte Bit 22: MCADDR, extended byte Bit 23: unused
24	MC ECCU Error	If 1, the memory control unit detected an ECC uncorrectable error.
25	I Unit Error	If 1, the I unit detected an error. Bits 26-28 describe the error:
26-28	I Unit Error Bits	000: no error 001: currently unused 010: currently unused 011: decode net, right byte 100: decode net, left byte 101: base register file high 110: base register file low 111: index register file
29	S Unit Error	If 1, the S unit detected an error. Bits 30-32 describe the error:
30-32	S Unit Error Bits	000: PID or STLB control bits 001: LBPA out of STLB in error 010: cache index, right 16 bits 011: cache index, left 16 bits 100: cache data high side 101: cache data low side 110: LBVA out of STLB in error 111: branch cache parity error

Table 11-13
Format of DSWPARITY Register for 750/850 Processors

Bits	Name	Description
1	RPA Parity Error, Type 1	If 1, the control store has detected a parity error as follows: DMx input E6: BPD or Burst- R0, R2 DMx input E5: BPD or Burst- R0, R1, R2, R3 DMx output: BMD
2	RPA Parity Error, Type 2	If 1, DMx input E6: BPD or Burst- R1, R3 DMx input E5: BPD DMx output: BMA
3	Burst-mode DMx Parity Error	If 1, the control store detected a DMx burst mode parity error.
4	DMx I/O Parity Error	Setting specifies that the control store detected a DMx parity error as follows: 0: DMx input 1: DMx output
5-7	J Board Parity Errors	The J board detected a parity error as follows: 000: peripheral reports BPD error (output) 001: base register file high 010: memory reports BMD error (write) 011: prefetch buffer address 100: peripheral reports BPA error (output) 101: base register file low 110: memory reports BMA error 111: prefetch buffer instruction.
8	RCM Parity Error	If 1 and no board reported an error, then an RCM parity error has been detected.
9	ECCC Error	If 1, memory detected an ECC uncorrectable error on read.
10	Prefetch Board Parity Error	If 1, prefetch board parity error.
11	BPA Input Parity Error	If 1, BPA input parity error (DMx or interrupt).
12	RDX Parity Error	If 1, RDX parity error when most recently closed.
13	Register File Parity Error	If 1, register file parity error.
14	REA Parity Error	If 1, REAH or REAL parity error.

Table 11-13 (continued)
Format of DSWPARITY Register for 750/850 Processors

Bits	Name	Description
15	DMx Cycle Parity Error	If 1, parity error occurred during DMx cycle.
16	AP Board Parity Error	If 1, AP board detected parity error.
17	C Board Parity Error	If 1, C board detected parity error.
18	BMD Input Even Parity Error	If 1, BMD input even word parity error.
19	BMD Input Odd Parity Error	If 1, BMD input odd word parity error.
20	Missing Memory Module	If 1, missing memory module at cache miss.
21	BMA Parity Error	If 1, memory detected BMA parity error at cache miss.
22	RMA Increment	If 1, RMA was incremented at time of parity error (cache miss).
23	BMA15 Indicator	Setting of BMA15 indicator at time of parity error (cache miss).
24	BMA16 Indicator	Setting of BMA16 indicator at time of parity error (cache miss).
25	ECCU Error	If 1, memory reports an ECC uncorrectable error on a cache miss.
26	ECCC Error	If 1, memory reports an ECC correctable error on a cache miss.
27	Cache Index Parity Error	If 1, cache index parity error on cache read.
28	Cache Data Odd Word Parity Error	If 1, cache data odd word parity error on cache read.
29	Cache Data Even Word Parity Error	If 1, cache data even word parity error on cache read.
30	Cache Cycle Purpose	Specifies the purpose of the cache cycle at the time of the error: 0: prefetch 1: execute
31-32	—	Currently unused.

Table 11-14
Format of DSWSTAT Register for 9950 Processor

Bits	Name	Description
1	Check Immediate	If 1, the check was taken immediately.
2	Machine Check	If 1, a machine check occurred.
3	Memory Parity	If 1, a memory parity error caused the check.
4	Missing Memory Module	If 1, a missing memory module caused caused the check.
5	E1 Unit	If 1, the E1 board reported a parity error.
6	S Unit	If 1, the S unit reported a parity error.
7	I Unit	If 1, the I unit reported a parity error.
8	MC Unit	If 1, the memory controller unit reported a parity error.
9	ECCU	If bits 3 and 9 are both 1, the memory parity error was ECC uncorrectable.
10	ECCC	If bits 3 and 10 are both 1, the memory parity error was ECC correctable.
11	CS Unit	If 1, the control store board reported a parity error.
12	RCM Parity	If 1, an RCM parity error was detected by the control store board.
13-14	RPBU	Specifies the RP backup count at the time of the error.
15	DMx Operation	If 1, a DMx transfer was in progress when the error occurred.
16	I/O Operation	If 1, an I/O operation was in progress when the error occurred.
17-23	ECC Syndrome Bits	If a memory parity error occurred, these bits describe the error. See Table 11-18.
24	Memory Module Number	If a memory error occurred, this bit identifies the interleaved memory module that contained the error (bit 15 of address at time of error).
25	RMA Invalid	If 1, the contents of DSWRMA are invalid.
26-32	—	Currently unused.

Table 11-15
Format of DSWSTAT Register for Rest of 50 Series Systems

Bits	Name	Description
1	Check Immediate	If 1, the check was taken immediately.
2	Machine Check	If 1, a machine check occurred.
3	Memory Parity	If 1, a memory parity error caused the check.
4	Missing Memory Module	If 1, a missing memory module error caused the check.
5-7	Machine Check Code	The hardware detected the cause of the trap as follows: 000: peripheral reports BPD error (output) 001: base register file high 010: memory reports BMD error (write) 011: prefetch buffer address 100: peripheral reports BPA error (output) 101: base register file low 110: memory reports BMA error 111: prefetch buffer instruction
8	RCM	Control unit memory — this bit is reset when an error is detected.
9	ECCU	If bits 3 and 9 are both 1, the memory parity error was ECC uncorrectable.
10	ECCC	If bits 3 and 10 are both 1, the memory parity error was ECC correctable.
11	BUNV	If 1, the RP backup count in bits 12-13 is not valid.
12-14	RPBUP	Specifies the RP backup count, which is the amount DSWPB was incremented in the current instruction.
15	DMx Operation	If 1, a DMx transfer was in progress when the error occurred.
16	I/O Operation	If 1, an I/O operation was in progress when the error occurred.
17-22	ECC Syndrome Bits	If a memory parity error occurred, these bits describe the error. See Table 11-18.
23	—	Currently unused.
24	Memory Module Number	If a memory error occurred, this bit identifies the interleaved memory module that contained the error (bit 15 of address at time of error).
25	RMA Invalid	If 1, the contents of DSWRMA are invalid.
26*	U-verify Pass	U-verify pass number as follows: 0: first pass (check mode off) 1: second pass (check mode on)
27-32*	U-verify Test Failure	If set, contains the number of a failed u-verify test.

Note to Table 11-15

* Valid for 750 and 850. For rest of 50 Series: bit 26 is unused; bit 27 is the u-verify pass number; and bits 28-32 are the u-verify test failure number.

Table 11-16
Format of the DSWRMA and DSWPB Registers for All 50 Series Systems

Bits	Name	Description
1-32	DSWRMA (Memory Address Register)	Contains bits 1-13 of a 23-bit physical address at the time of the error. Valid: If an ECCC, ECCU, or missing memory module check occurred. Invalid: If any other checks occurred, or if no check occurred. In the event of multiple checks, DSWRMA is the RMA of the missing memory module check, if there is one. If not, it is the RMA of the machine or ECC-uncorrected check (they are mutually exclusive) if there is one. If not, it is the RMA of the ECC-corrected check.
1-32	DSWPB (Extended Program Counter — ring, segment, word)	Always valid. In the event of multiple checks, DSWPB is the program counter of the missing memory module check, if there is one. If not, it is the program counter of the machine or ECC-uncorrected check (which are mutually exclusive) if there is one. If not, it is the program counter of the ECC-corrected check.

Each time the processor performs a check (except for power failure) it sets particular register file locations to reflect the contents of the DSW, as shown in Table 11-17.

Table 11-17
DSW Value After Checks

RF Loc	Contents
'27*	DSWPARITY (750, 850 and 9950 only)
'34*	DSWRMA
'35*	DSWSTAT
'36*	DSWPB

* These are absolute locations in the register file.

MCM Field: The 50 Series processors use the MCM field (bits 15-16) of the modals to determine what kind of check reporting to do. Table 11-18 shows the possible modes of reporting.

Table 11-18
Modes of Check Reporting

Modals 15-16	Reporting Mode
00	No reporting.
01	Report uncorrected memory errors (ECCU) only.
10	Report fatal (ECCU, machine checks, and missing memory module) errors only. (This state is called quiet mode.)
11	Report all errors.

Check Handler Operation

As with faults, the type of check service provided depends on whether the PXM is enabled or not when the check occurs. If the PXM is disabled, the processor sets the MCM field in the modals to 0, then jumps indirectly (JST) through the appropriate check vector to the check routine.

If the PXM is enabled, the processor:

1. Sets up a check header.
2. Inhibits the machine.
3. Switches to 64V mode.
4. Sets the MCM field to 0 (2 if ECCU).
5. Transfers control to the check handler.

The software must clear the DSW after each check. This ensures that the processor does not use old data when servicing future checks.

The DSW is large enough to contain data about one of each type of check before the handler takes control. However, the values of RMA and PB for the last check only are saved.

To determine which check stored RMA and PB, use DSWSTAT to determine which checks have occurred:

- If a missing memory module check, machine check, or ECCU memory check occurred, RMA and PB reflect values stored by that check. These three checks are mutually exclusive, and are guaranteed to be the most recent check that occurred.
- If any other check occurred, RMA and PB reflect values stored by the ECCC check that occurred most recently.

Table 11-19 summarizes some information about each check.

Table 11-19
Types of Checks

Type of Check	Header Loc*	Handler Loc*	Effect on DSW
Power failure	'200	'204	Does not set DSW.
Environment (9950 only)	'200	'204	Does not set DSW.
Memory parity	'270	'274	Sets DSW.
Machine check	'300	'304	Sets DSW.
Missing memory module	'310	'314	Sets DSW.

* These are locations in Segment 4.

Check Trap

Some checks cause a microcode trap when they occur. When this happens, the action taken depends on the type of microcode that was trapped. Table 11-20 shows the checks that can cause traps and the actions that occur.

Note that the first and second categories listed in Table 11-20 always leave the I/O bus clean.

Table 11-20
Check-produced Traps and Their Actions

Event	Actions
Missing Memory Module, ECC Uncorrectable, or Machine Check during I/O (DMx, PIO, interrupt processing, excepting machine check for ROM parity)	9950: Error is ignored until all current requests for DMx and I/O are processed, and then the check is taken immediately. Rest of 50 Series: Sets end-of-instruction flag to 1; sets REOIV to the proper offset or vector; sets MCM to 00; executes microcode return to the trapped microcode step. Note that correctable memory errors are ignored during I/O.
ECC Correctable Error (not during I/O)	9950: Action is deferred until the next fetch cycle, and then a check is taken. Rest of 50 Series: Sets end-of-instruction flag to 1; sets REOIV to the proper offset or vector; sets MCM to 2; executes microcode return to the trapped microcode step.
Power Failure; Environment (9950 only)	All 50 Series: Action is deferred until the next fetch cycle, and then a check is taken.
All other checks	All 50 Series: Software check occurs immediately.

TRAPS

Traps are breaks in microcode execution. When a trap occurs, the processor takes the current microcode location where the trap occurred and goes to the predetermined microcode location that handles traps. The processor handles the trap, then retries the microcode location where the trap originally occurred.

Traps are separated into two groups. G1 traps occur during references to parts of memory. G2 traps are hardware related and encompass several subgroups. Table 11-21 lists the traps in both groups and the further subgroups.

The traps are listed in order of priority, from highest to lowest. G2 traps always have higher priority than G1 traps do. Within the G2 group, missing memory module traps have the highest priority.

Table 11-21
Types of Traps and Their Priorities

Type	Individual Trap	Causes and Actions
G1	32-bit or 16-bit read address trap	If a memory reference instruction forms an EA between 0-'7 (V mode) or 0-'37 (S and R mode), the addressed location is in the current user register file, not memory. When such an address is calculated, this trap aborts the memory read and loads a cache entry with the contents of the addressed register. The cache is marked invalid but the cache's use once bit is set to 1 so that a cache hit occurs when the microstep is retried. A cache miss occurs on the next reference to this cache entry.
	STLB miss	This trap aborts the step. The STLB miss translates the virtual address to a physical one, then puts the translation into the STLB. The step is retried after the translation is loaded into the STLB.
	Access violation	A procedure tries to reference a memory location for which it has insufficient access rights. This trap causes an access violation fault.
	Page modified trap	This trap occurs during each step that writes into a physical page whose modified bit (in the page's STLB entry) contains 0. This trap sets the modified bit to 1 so that future writes to this page do not cause other traps.
G2	Missing memory module	If no memory board responds to a memory read or write request, this trap occurs. Actions taken as a result of this trap depend on the operating system.

Table 11-21 (continued)
Types of Traps and Their Priorities

Type	Individual Trap	Causes and Actions
	Machine check	Indicates a parity error or an ECCU. DSWPARITY indicates the type of parity error that occurred. See Checks in this chapter for more information. This is a fatal trap.
	Write address trap	Specifying an address within the range 0-'7 (V Mode) or 0-'37 (S or R mode) as a write address causes this trap. This trap aborts the write to memory but otherwise allows the operation to to complete.
	Integer exception	The current instruction caused an integer exception. This trap causes an integer exception fault.
	Branch cache problem	A branch cache hit occurs during execution of something other than a branch instruction.
	DMx requests	If a controller wants to request a DMX transfer, this trap transfers control to the DMx microcode.
	Fetch cycle traps:	Allows the processor to perform several steps between microsteps.
	CPU timer overflow	The microsecond timer overflows. This trap increments the contents of TIMER by 1. If the incremented value of TIMER does not overflow, execution continues. If it does overflow, this trap sets the process abort flag in the process' PCB to 1.
	Diagnostic processor interrupts*	The diagnostic processor sends a command to the processor. The microcode reads the command and decodes it.

* 9950 and 2250 only

Table 11-21 (continued)
Types of Traps and Their Priorities

Type	Individual Trap	Causes and Actions
	ECCC	Error correcting codes on the memory boards note when single bit errors in MOS memory occur. This trap notes the address where such an error occurred and the value of that address' syndrome bits. The syndrome bits show which bit in that location is in error. See Table 11-18 for information about syndrome bit values for single bit errors.
	External interrupts	Point where a device requested service. This trap causes an external interrupt. (See Interrupts in this chapter for more information.)
	Memory increment interrupts	Point where a controller requested service. This trap causes a memory increment interrupt. (See Interrupts in this chapter for more information.) Not used for 9950.
	Program interval timer	If the timer is enabled, it causes an interrupt. The timer places a vector on the address bus for PRIMOS.
	End-of-instruction trap	A parity error occurred on an I/O transfer. Not used for 9950.
	Power failure	AC power failed. This trap causes a power failure check. (See Checks in this chapter for more information.)
	Restricted instruction trap	This trap causes a fault when a process tries to execute a restricted instruction in a ring other than Ring 0.

Read Address Trap

If the effective address calculated by a memory reference instruction is within the range 0-'7 (V mode) or 0-'37 (S and R mode), inclusive, the addressed location is in the current user register file, not in memory. When such an address is calculated, this trap aborts the memory read and loads a cache cell with the contents of the addressed register. The cache is marked invalid but the cache's use once bit is set to 1 so that a cache hit occurs when the microstep is retried. The cache miss occurs on the next reference to this cache cell.

STLB Miss

When an STLB miss occurs, this trap aborts the step. The STLB miss translates the virtual address to a physical one, then puts the translation into the STLB. The step is retried after the translation is loaded into the STLB.

Access Violation

If one procedure tries to call another and an access violation occurs, this trap causes an access violation fault.

Page Modified

This trap occurs during each step that writes into a physical page whose modified bit (in the page's STLB entry) contains a 0. This trap sets the modified bit to 1 to indicate the presence of information that must be saved.

Missing Memory Module

If no memory board responds to a memory read or write request, this trap occurs. A missing memory module check alerts the operating system to this trap's occurrence; resulting actions depend on the operating system.

Error Correcting Code

Error correcting codes on the memory boards note when single bit errors in MOS memory occur. This trap notes the address where such an error occurred and the value of that address' syndrome bits. The syndrome bits show which bit in that location is in error. Tables 11-22 and 11-23 show the values of the syndrome bits and the single bit errors they indicate for the 9950 and the rest of the 50 Series systems, respectively.

Table 11-22
Syndrome Bits for 9950 Processor

Check Bits 6543210	Bit in Error	Check Bits 6543210	Bit in Error
0000000	No error	0101100	Word bit 13
0000001	Check bit 0	1101101	Word bit 14
0000010	Check bit 1	1101110	Word bit 15
0000100	Check bit 2	0101111	Word bit 16
0001000	Check bit 3	1110000	Word bit 17
0010000	Check bit 4	0110001	Word bit 18
0100000	Check bit 5	0110010	Word bit 19
1000000	Check bit 6	1110011	Word bit 20
0000111	Word bit 01	0110100	Word bit 21
1100001	Word bit 02	1110101	Word bit 22
1100010	Word bit 03	1110110	Word bit 23
0100011	Word bit 04	0110111	Word bit 24
1100100	Word bit 05	0111000	Word bit 25
0100101	Word bit 06	1111001	Word bit 26
0100110	Word bit 07	1111010	Word bit 27
1100111	Word bit 08	0111011	Word bit 28
1101000	Word bit 09	1111100	Word bit 29
0101001	Word bit 10	0111101	Word bit 30
0101010	Word bit 11	0111110	Word bit 31
1101011	Word bit 12	1111111	Word bit 32

Table 11-23
Syndrome Bits for the Rest of the 50 Series

Check Bits 123456	Bit in Error	Check Bits 123456	Bit in Error
00000X	Multiple bits	10000X	Multiple bits
00001X	Multiple bits	100011	Word bit 07
00010X	Multiple bits	10010X	Multiple bits
000111	Word bit 15	100111	Word bit 03
00100X	Multiple bits	10100X	Multiple bits
001011	Word bit 14	101011	Word bit 02
001101	Word bit 13	101101	Word bit 01
001111	Word bit 09	101111	Check bit 2
01000X	Multiple bits	110001	Word bit 08
01001X	Multiple bits	110011	Word bit 06
01010X	Multiple bits	110101	Word bit 05
010111	Word bit 12	110111	Check bit 5
011001	Word bit 16	111001	Word bit 04
011011	Word bit 11	111011	Check bit 4
011101	Word bit 10	111101	Check bit 3
011111	Right parity/ check bit 1	111111	Overall parity
		111110	No error

Note to Table 11-23

X means undefined.

Machine Check

This trap, like that for missing memory module, indicates a serious problem with the system. It may indicate faulty components, noise, or a timing problem. This is a fatal trap.

Write Address Trap

Specifying an address within the range 0-'7 (V mode) or 0-'37 (S or R mode) as a write address causes this trap. This trap aborts the write to memory but otherwise allows the operation to complete.

DMx

If a controller wants to request a DMx transfer, this trap transfers control to the DMx microcode.

Fetch Cycle Traps

Fetch cycle traps occur only at the end of the first microstep of a Prime assembly language instruction. They are caused by a program interval timer overflow, external interrupts, and power failures.

These traps occur only after the first step of an assembly language instruction has completed. This guarantees that the previous assembly language instruction has completed execution.

Restricted Instruction

This trap causes a fault when a process tries to execute a restricted instruction in a ring other than Ring 0.

Summary of Software Breaks Caused by Traps

As mentioned above, some of the traps listed in Table 11-21 cause software breaks. Table 11-24 shows which traps cause additional breaks and the types of breaks that can occur.

Table 11-24
Software Breaks Caused By Traps

Traps	Additional Software Break
Missing memory module; ECCU, machine check, and other parity errors	No additional break occurs. These traps are reported to the operating system via a check; the operating system takes an appropriate action.
External interrupt, memory increment interrupt, program interval timer interrupt	Interrupt occurs.
Integer exception, access violation, restrict mode violation	Fault occurs.
STLB miss	Page fault, segment fault may occur.
Power failure, ECCC	Check occurs.
All other traps	No additional action occurs.

INTERVAL CLOCK

The 2250 uses a 500 Hz interval clock to drive the PRIMOS clock process. The clock generates a timing pulse every 2 milliseconds. The 9950 uses a 250 Hz interval clock.

If the interval clock is enabled, a fetch cycle trap occurs when a timing pulse occurs. The fetch cycle trap causes an external interrupt. If interrupts are enabled on the machine, the processor services the interrupt and updates the pointers in the clock process. If interrupts are disabled, the interrupt is ignored.

Table 11-25 lists the instructions that control the interval clock.

Table 11-25
Instructions Affecting the Interval Timer

Mnem	Name	Modes	Description
INA '1120	Input to A	S,R,V	Loads the ID of the controller into A.
INA '1320	Input to A	S,R,V	Loads the contents of the interrupt vector into A.
OCP '0020	Output Control Pulse	S,R,V	Starts the interval timer.
OCP '0220	Output Control Pulse	S,R,V	Stops the interval timer.
OTA '0720	Output from A	S,R,V	Transfers data from A into the control register.
OTA '1320	Output from A	S,R,V	Transfers data from A into the interrupt vector.
SKS '0020	Skip on Condition Met	S,R,V	Skips if the interval timer is not interrupting.

SUMMARY

This chapter described four kinds of breaks in execution that can occur, and how the 50 Series processors handle them. Traps are breaks in microcode execution. Checks indicate hardware consistency problems; faults indicate software exception conditions. External devices issue interrupts when they desire service. The next chapter, Input/Output, shows how external devices issue interrupts, and how the 50 Series processors handle these requests for service.

12
Input/Output

The previous chapter discussed the various types of breaks that can occur in program execution. The I/O system is closely related to these breaks, since data transfers between the processor and other parts of the system usually include some type of break. Depending on the type of transfers and the device, the I/O system can perform a wide variety of functions applicable to many situations.

I/O on the 50 Series processors is divided into three types:

- Programmed I/O (PIO)
- Direct memory (DMx)
- Interrupts

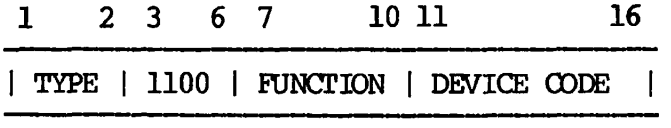
These three types of I/O differ in what initiates the action. For PIO, the processor issues a command to a device, which performs the desired action. For DMx transfers, a controller requests service from the processor, which provides the service on a priority basis. For interrupts, the controller again alerts the processor to a situation that requires the processor's attention. Chapter 11 discussed interrupts and how the processor deals with them. This chapter describes PIO and DMx.

PROGRAMMED I/O

PIO is I/O performed by a program. This means that the instruments used to perform PIO are instructions. These instructions:

- Send control information to a peripheral device.
- Test devices for skip conditions.
- Move data between a device and the CPU.

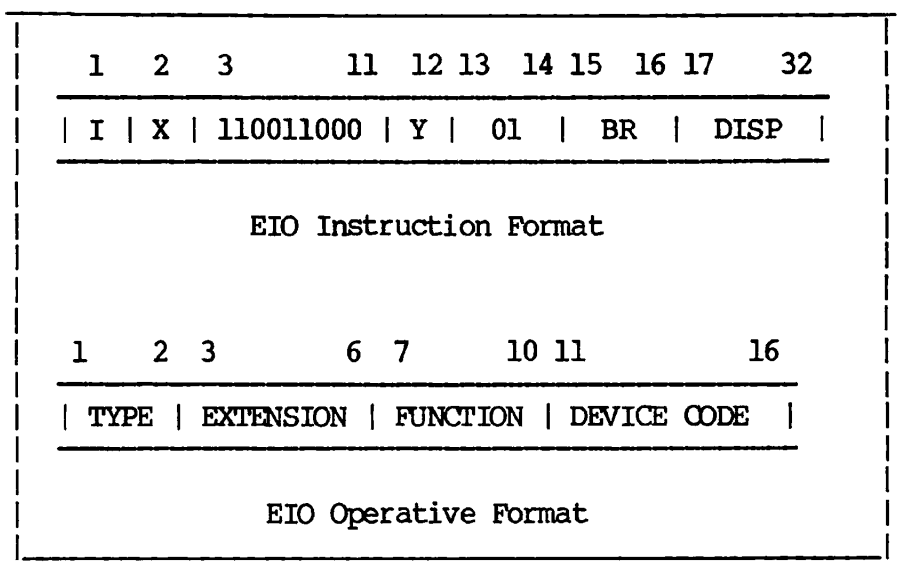
The PIO instructions use one of two formats. In S and R mode, four PIO instructions (OCP, SKS, INA, OTA) are available for use. They have the format shown in Figure 12-1.



INA, OCP, OTA, SKS Operative Format
Figure 12-1

For these four instructions, the operative (the part that the processor actually executes to perform the desired action) is the instruction itself. A different arrangement exists for V mode PIO.

In V mode, the processor cannot directly execute INA, OCP, OTA, or SKS. Instead, it must use an EIO instruction, which forms an effective address. The processor executes bits 1-16 of this effective address as a PIO instruction. These bits (the operative of the EIO instruction) should specify one of the four PIO instructions described above. The upper drawing in Figure 12-2 shows the format of the EIO instruction; the lower drawing shows the format the processor uses to interpret the EIO operative.



EIO Formats
Figure 12-2

Note that both the S and R mode PIO operatives and the V mode operative have the same basic format. In both cases, bits 1-6 specify the operation the processor is to perform. Bits 1-2 always identify the basic type of operation to be performed, as shown in Table 12-1.

Table 12-1
Basic I/O Operations

Type	Inst	Name
00	OCP	Output control pulse
01	SKS	Skip if condition satisfied
10	INA	Input to A
11	OTA	Output from A

Bits 3-6 have different meanings in different modes. In S and R modes, these bits are set to 1100. In V mode, bits 3-6 specify an extension to the type field. The processor may use this field to distinguish between controllers that require different types of service, or between different software implementations. This feature has yet to be completely defined.

In all three modes, the function field (bits 7-10) specifies one of 16 device-dependent commands. Each controller defines the function codes that it uses for each of the four basic PIO operations. For example, the controller for one device might use INA with a function field of 0 to load data into A, INA with a function field of 1 to load a device ID into A, and INA with a function field of 3 to load status into A.

In all three modes, bits 11-16 specify a device address that identifies a controller and its implementation. Tables 12-5 and 12-6, at the end of this chapter, show these device addresses.

PIO Operative Actions

The processor performs the same actions for each identical PIO operative, regardless of the mode of the machine. This means that the INA operation specified by EIO in V mode results in the same actions as does the INA directly executed in S and R modes. After performing the operation, however, the processor indicates the success or failure of the operation in different ways, depending on the mode. The descriptions below explain the actions of each operation, as well as how the processor indicates success or failure for each mode.

INA: INA is enabled over BPA. If the specified device is not ready and does not have device address '20, the instruction ends. If the device is ready, or has device address '20, the device responds ready and data is read over BPD. In V mode, the condition codes reflect success or failure as shown in Table 12-2.

In S and R mode when the device address is not '20, the processor indicates success by incrementing the contents of the program counter by 1; when the device address is '20, no increment occurs.

Note that for device address '20 the data can have bad parity. INAs to device address '20 ignore the data parity and generate their own correct parity. INAs to device addresses other than '20, however, do check the data parity and indicate a BPD parity error if the parity is incorrect.

Table 12-2
Effect of EIO on Condition Codes

CC	Meaning
EQ	Successful INA, OTA, or SKS instruction
NE	Unsuccessful INA, OTA, OR SKS; any OCP

OTA: OTA is enabled over BPA. If the specified device is not ready and does not have device address '20, the instruction ends. If the device is ready, or has device address '20, the device responds ready and data in A is sent over BPD to the device. In V mode, the condition codes reflect success or failure as shown in Table 12-2.

In S and R mode when the device address is not '20, the processor indicates success by incrementing the contents of the program counter by 1; when the device address is '20, no increment occurs.

SKS: SKS is enabled over BPA. If the specified device is not ready, the instruction ends. If the device is ready, the processor indicates success in V mode by setting the condition codes, as shown in Table 12-2, regardless of the device address.

If the device is ready, the processor indicates success in S and R mode by incrementing the contents of the program counter by 1, regardless of the device address.

OCP: OCP is enabled over BPA. The specified device performs the specified command and the instruction ends. Note that OCP never indicates success or failure.

DMX

While PIO operations are suitable to use when only small amounts of data need to be transferred, they are typically not suitable for multiple word transfers. Each time PIO transfers data, the processor must execute several instructions for each transferred word. This ratio of control instructions to transferred data makes the transfer of blocks of data rather slow. DMx operations allow devices to access memory directly, rather than by using software. This cuts down on the amount of processor time required to perform the transfer, and allows the transfer to occur without specific software attention.

DMx Transfers

There are four types of DMx transfers:

- DMA, or direct memory access
- DMC, or direct memory channel
- DMT, or direct memory transfer
- DMQ, or direct memory queue

All of these transfers occur in three phases. The request to transfer occurs during the request phase. The CPU receives the transfer address during the address phase. The data is transferred during the data phase.

To make any DMx request, the controller desiring the transfer sends a DMx request to the processor. This request will be serviced when:

- The processor issues a DMx request enable.
- There are no other DMx requests pending from devices with a higher priority (a lower slot number).

If the request from this controller has the highest priority, the processor recognizes it. The controller sends an address on BPA and control information on the mode lines. The mode lines request the specific type of DMx transfer, which in turn defines how the address line information is to be interpreted.

After receiving the control information, the processor strobes the data as appropriate over BPD. The processor sends an end-of-range (EOR) signal, if appropriate, at the end of the block transfer.

The length of time between when a device requests service and when the processor responds depends on two things:

- How many requests of higher priority are already pending.
- What the processor is doing when the device makes its request.

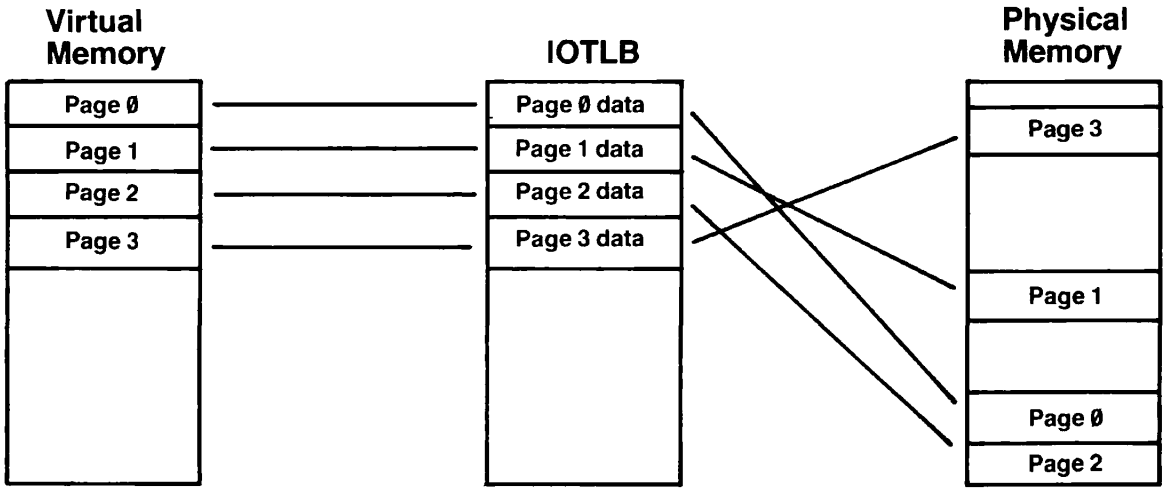
A device must wait until the processor services all requests of higher priority. This means that the device with the highest priority in the system can preempt service to any other device, and may completely occupy the processor if it transfers data at the maximum rate.

Though the processor can pause between instructions or at selected points within instruction execution, it cannot stop immediately each time a request for a transfer occurs. Also, the processor cannot service requests when servicing interrupts or phantom interrupt code. This means that even the highest priority device in the system may have to wait less than 7 microseconds if the processor is busy. Once the processor transfers the first word, however, it transfers the rest of the words in the block as fast as possible. At the maximum speed, note that the processor cannot process anything else at the same time.

Mapped I/O

When a controller specifies the transfer starting address, it can specify a virtual address or a physical one. The processor is using absolute I/O when the address specified is a physical one. When the controller specifies a virtual address, the processor is using mapped I/O.

Mapped I/O allows the limited addressing range of DMx transfers to address all of physical memory. It is especially useful when transferring several contiguous pages in virtual memory to physical locations that may not be contiguous. For example, suppose the processor wants to transfer four contiguous pages of data in virtual memory to a device. As shown in Figure 12-3, mapped I/O allows the system to map the four pages to any four available pages, instead of requiring one four-page block.



Mapped I/O
Figure 12-3

The IOTLB contains the information needed to map the transfer addresses to physical memory locations. The IOTLB forms half of the virtual-to-physical address mapping hardware. (The STLB is the other half.) It contains 64 entries (128 for the 9950). Table 12-3 shows the contents of each IOTLB entry.

Table 12-3
IOTLB Entry Format

Number of Bits				
9950	750, 850	2250, I450, 550-II	Contents	Description
13	12	12	Physical page number	Specifies a physical page in either of the I/O segments.
1	1	1	Valid bit	Indicates if this entry contains old data.
3	3	2 (0 for 2250)	MBIO bits	Specifies the cache leaf to invalidate when writing to memory.

Note that 3 MBIO bits are used for an 8-leaf cache; 2 MBIO bits for a 4-leaf cache. These MBIO bits determine which eighth or fourth of the cache, respectively, to invalidate after a memory write. Since the cache of the 9950, 750, or 850 contains 16K bytes, it can contain mapping information about 8 entries of physical memory, each having the same page offset. The cache of the I450 and 550-II has 8K bytes and can hold mapping information for 4 entries of physical memory. The MBIO bits allow the information for only the modified entry to be invalidated after a memory write, rather than each of the 4 or 8 possible places.

Each IOTLB entry contains mapping information for one page of the I/O segments. Entry 0 contains mapping information about page 0 of segment 0; entry 64, about page 0 of segment 1.

The IOTLB allows the I/O address translation during DMx to be done swiftly because information about the translation is always guaranteed to be in the IOTLB. If the processor were to rely on the STLB, an STLB miss could occur and the transfer would fail. Preloading the IOTLB is, therefore, essential before initiating I/O.

The LIOT instruction loads the IOTLB entries with transfer information. On all machines except the 2250, this instruction must be used before any transfer occurs so that the processor maps virtual pages to the desired physical ones. The 2250 has no leaf bits because the cache is exactly the size of a page. On the 2250, loading the IOTLB is done by accessing the appropriate page in segment 0 by an instruction, such as LDA, before any transfer.

DMA

DMA is useful for bulk data transfers when speed is important. Maximum rates of transfer for DMA and the other DMx transfers for the 9950 are shown in Table 12-4; maximum transfer rates for all other processors are shown in Table 12-5.

The register file contains the DMA register set occupying locations '40-'77. These locations contain direct memory channels 0-'37, respectively, that allow devices to access memory with a minimum of processor intervention.

Making a DMA Request: To perform a DMA transfer, a program must:

1. Set up a DMA cell.
2. Tell the controller to perform the transfer.

A DMA cell is one 32-bit location in the register file. Bits 1-12 of this location contain the two's complement of the total number of halfwords to be transferred. This means that the largest block of halfwords that can be transferred on a single channel is 4096; to transfer more requires more than one channel.

The use of bits 13-32 depends on the machine and on whether mapped I/O mode or physical I/O mode is being used. In physical I/O mode, bits 13 and 14 are reserved; bits 15-32 supply the physical address of the first location to transfer.

Table 12-4
DMx Transfer Rates on the 9950

Type	Transfer	Maximum Speed
DMA	Input	2.4 Mbytes/sec
	Output	2.0 Mbytes/sec
DMC	Input	1.2 Mbytes/sec*
	Output	1.1 Mbytes/sec*
DMT	Input	2.8 Mbytes/sec*
	Output	2.2 Mbytes/sec*
DMQ	Input	300 Kbytes/sec*
	Output	300 Kbytes/sec*
Burst mode	Input	9.4 Mbytes/sec
	Output	6.0 Mbytes/sec

* This is an approximate value.

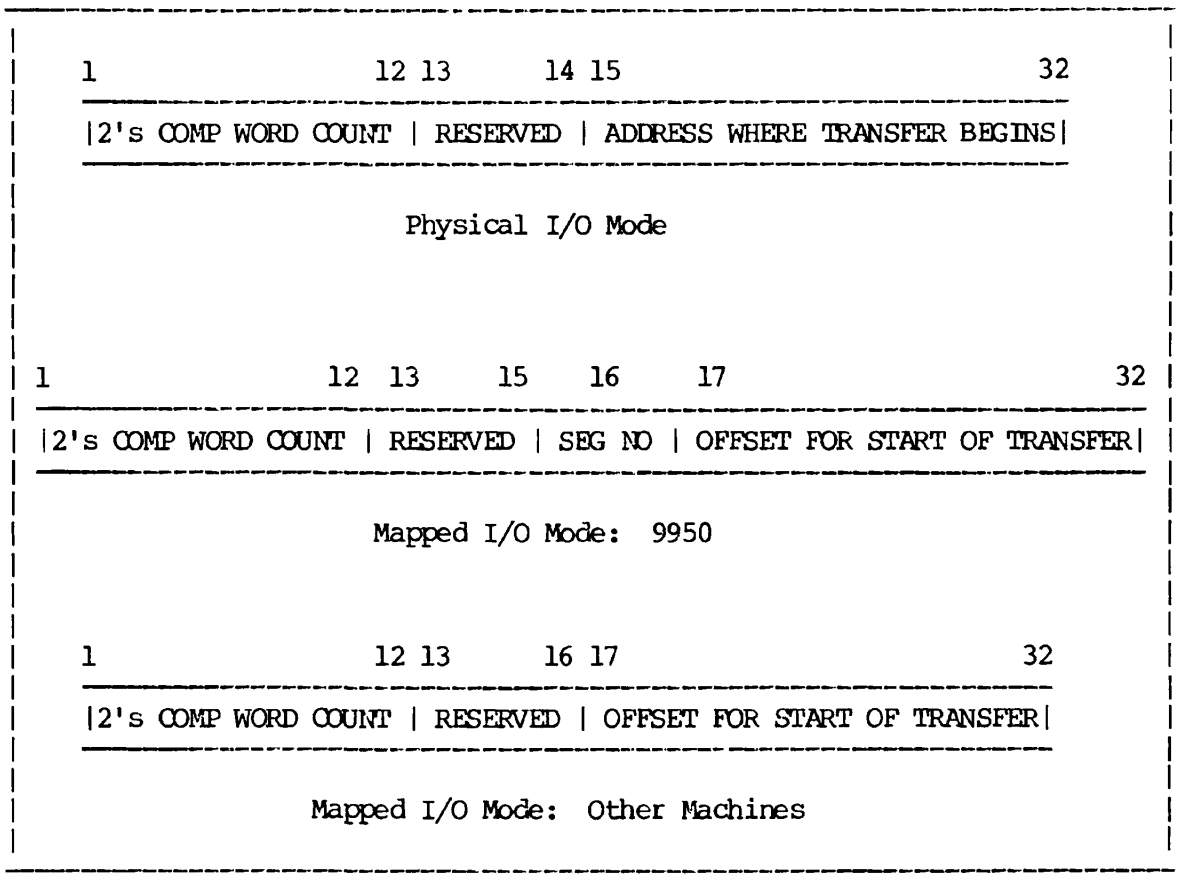
Table 12-5
DMx Transfer Rates for Rest of 50 Series

Type	Transfer	Maximum Speed
DMA	Input	2.5 Mbytes/sec
	Output	2.5 Mbytes/sec
DMC	Input	1.0 Mbytes/sec*
	Output	1.0 Mbytes/sec*
DMT	Input	2.5 Mbytes/sec*
	Output	2.5 Mbytes/sec*
DMQ	Input	280 Kbytes/sec*
	Output	280 Kbytes/sec*
Burst mode	Input	8.0 Mbytes/sec
	Output	5.6 Mbytes/sec

*This is an approximate value.

When mapped I/O mode is being used on the 9950, bit 16 selects I/O segment 0 or 1; bits 17-32 specify the offset within the segment at which the transfer is to begin. Bits 13-15 are reserved.

When mapped I/O mode is used on other machines, bits 13-16 are reserved; bits 17-32 designate the offset within segment 0 of the first (or next) location to transfer.



Format of DMA Control Word
Figure 12-4

Servicing a DMA Request: When a controller wants to make a transfer, it signals the processor over BPA for memory access via a requested channel. Normally, the processor acts on the request one microstep after the request arrives. If only one request is pending, the processor services it immediately. If more than one is pending, the processor services the request from the device mounted in the lowest numbered I/O slot first, then it services the request from the device in the next lowest slot, and so on.

Note that when the processor pauses to service a request, it services all pending requests before resuming instruction execution or servicing an interrupt.

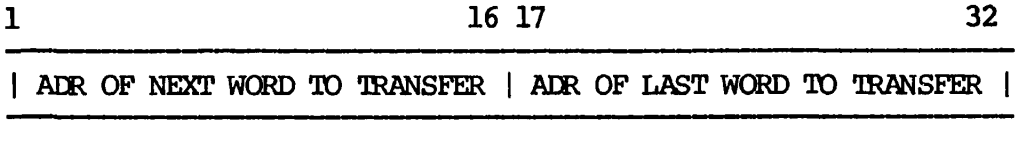
Once the processor has selected a request for service, it fetches the contents of the word to transfer and either sends them over the bus, or stores them at the address specified by the channel. It then increments the values of the word count and transfer address by 1. If the incremented value of the word count is 0, the processor issues an EOR signal. A word count of any other value means that there are more words to transfer.

At the end of each request, the word count specifies the number of words left to transfer and the transfer address specifies the address of the next word to transfer. At the normal end of the transfer, the word count contains a 0 and the transfer address specifies the address of the last word transferred plus 1.

DMC

DMC operates in much the same way as DMA does. The differences are that DMC provides a total of 32,768 channels rather than just 32, and that data blocks can contain up to 64K words. Also, the DMC transfer rate is much slower than that for DMA since DMC performs three memory operations per transfer versus one for DMA.

DMC operations require a control word just as DMA operations do. The DMC control word, however, is not contained in the current register file, but in a 32-bit memory location in the range of 0-'177776. Bits 1-16 of the control word contain the 16-bit address of the next word to be transferred; bits 17-32, the 16-bit address of the last word to be transferred. (See Figure 12-5.) The DMC control word must be aligned on an even word boundary.



Format of DMC Control Word
Figure 12-5

As in DMA service, a controller uses BPA to request the processor for memory access via a specified channel. When the processor can break its execution, it services any pending requests. If more than one request is pending, the processor services the request of the device mounted in the lowest numbered slot first, then others in order of their priority.

Once it has selected a request to service, the processor either reads or writes the contents of the location specified in bits 1-16 of that channel's control word. After the read or write, the processor increments the contents of bits 1-16 by 1. If the value before the increment equals the contents of bits 17-32 in the control word, the processor issues an EOR signal. If the two values are not equal, then there are more words to transfer.

At the end of each request, bits 1-16 of the control word point to the next word to transfer. At the normal end of the transfer, bits 1-16 point to the last transferred location plus 1.

DMT

DMT transfers are used by controllers that do not need an external control word stored in memory or in the register file. Since the controller specifies all the information necessary to perform the transfer, all channel control functions can overlap with processor and memory functions at a speed equivalent to that of DMA transfers. DMT transfers are useful when manipulating tumble tables and channel programs.

When a controller wants to request a DMT transfer, it uses BPA to ask the processor for memory access. When the processor can service the request, it transfers words to or from the controller. The address specified by the controller is either the source or the destination of the data to transfer, depending on the transfer direction.

DMQ

Chapter 6, Datatypes, defined queues, their parameters, and how they are manipulated. As noted there, one of their uses is as a storage device. DMQ operations use physical memory queues to hold data traveling between device and processor.

To make a DMQ request, the controller uses BPA to ask the processor for queue access via a selected QCB. (The QCB address specified over BPA must be aligned on a four-word boundary.) For an input operation, the processor adds the contents of the word at the specified address to the bottom of the queue (equivalent to an ABQ), if there is room. If there is no room, the processor sends an EOR signal to the controller.

For an output operation, the processor removes the word from the top of the queue (equivalent to an RTQ) and transfers it to the specified address. If the queue contains no words, the processor issues an EOR to the controller, as well as a word of zeroes. Note, however, that if the processor removes the last word from a queue, it does not signal the controller when it removes the word.

DMQ is fully interlocked with the queue manipulation instructions shown in Chapter 6.

Burst Mode I/O

Burst mode is used only with 750, 850, and 9950 systems.

Burst mode operations are similar to DMA transfers because they are both set up the same way. Like DMA, burst mode sets up a DMA cell and tells the controller what to transfer. The difference is that burst mode transfers four 16-bit quantities of data in each transfer, rather than just one. This makes burst mode efficient for transferring large blocks of data. The DMA range count and address are both incremented by 4 each transfer.

The data to be transferred can be arbitrarily aligned in memory. However, burst mode will operate as ordinary DMA at ordinary DMA rates unless the data is aligned on 64-bit boundaries and there are at least 64 bits left in the range.

The controllers do not request burst mode transfer unless they have 64 bits or more of data to transfer. If the controllers have been doing a burst mode transfer but have, for example, 32 bits left, they request two ordinary DMA transfers to the same channel.

Table 12-5
Device Address Assignments

Device Address	Controller Model	Device Description
'00	——	——
'01	3000	Paper tape reader
'02	3000	Paper tape punch
'03	3100	URC #1 (unit record controller): line printer, card reader, card punch
'04	3000	System terminal
'05	3100	URC #2: line printer, card reader, card punch
'06	7000	IPC (interprocessor communications board)
'07	7040	Primenet node controller #1
'10	——	ICS2 #1 or ICS1 (See Table 12-6 for number.) (intelligent communications subsystem)
'11	——	ICS2 #2 or ICS1 (See Table 12-6 for number.)
'12	4300	Floppy disk
'13	4020	Magnetic tape #2
'14	4020	Magnetic tape #1
'15	5000	AMLC #5 (asynchronous multiline controller) or ICS1 (See Table 12-6 for number.)
'16	5000	AMLC #6 or ICS1 (See Table 12-6 for number.)
'17	5000	AMLC #7 or ICS1 (See Table 12-6 for number.)
'20	3000	Control panel, RTC (realtime clock), SOC (system option controller)
'21	4002	Disk option B'
'22	4004/5/6	Disk controller #3
'23	4004/5/6	Disk controller #4
'24	2076 & 461	Writeable control store
'25	4000	Disk option B
'26	4004/5/6	Disk controller #1
'27	4004/5/6	Disk controller #2
'30	3007	Buffered parallel I/O channel #1
'31	3025	Buffered parallel I/O channel #2
'32	5000	AMLC #8 or ICS1 (See Table 12-6 for number.)
'33	3009/3008	Versatec/Gould printer plotter
'34	3009/3008	Versatec/Gould printer plotter
'35	5000	AMLC #4 or ICS1 (See Table 12-6 for number.)
'36	——	ICS1 #1
'37	——	ICS1 #2
'40	6000 & 6005	PRIMAD (AIS, analog input system)
'41	6020	Digital input #1 (DIS)
'42	6020	Digital input #2
'43	6040	Digital output #2 (DOS)
'44	6040	Digital output #2
'45	6060	Digital to analog (AOS, analog output system)
'46	6080	Computer products interface
'47	7040	Primenet node controller #2

Table 12-5 (continued)
Device Address Assignments

Device Address	Controller Model	Device Description
'50	5300	HSSMLC #1 (high speed synchronous multiline controller) or MDLC (multiple data link controller)
'51	5300	HSSMLC #2 or MDLC
'52	5000	AMLC #3 or ICS1 (See Table 12-6 for number.)
'53	5000	AMLC #2
'54	5000	AMLC #1
'55	5400	Multiple autocall
'56	5200	SMLC (synchronous multiline controller)
'57	——	——
'60	7000	General purpose interface board
'61	7000	General purpose interface board
'62	7000	General purpose interface board
'63	7000	General purpose interface board
'64	7000	General purpose interface board
'65	7000	General purpose interface board
'66	7000	General purpose interface board
'67	7000	General purpose interface board
'70	——	Reserved for specials
'71	——	Reserved for specials
'72	——	Reserved for specials
'73	——	Reserved for specials
'74	——	Reserved for specials
'75	——	Reserved for controllers using T$GPPI
'76	——	Reserved for controllers using T$GPPI
'77	——	I/O bus tester

Table 12-6
ICS1 Number and Address Assignments

ICS1 Number	Device Address (Dependent on Number of ICS2s Configured)		
	No ICS2s	1 ICS2	2 ICS2s
1	'36	'36	'36
2	'37	'37	'37
3	'10	'11	'32
4	'11	'32	'17
5	'32	'17	'16
6	'17	'16	'15
7	'16	'15	'35
8	'15	'35	'52

13

S, R, and V Mode Instruction Dictionary

INTRODUCTION

This chapter contains descriptions for all 50 Series instructions used in S, R, and V modes. In the description of each instruction, you will find:

- The instruction mnemonic followed by any arguments.
- The name of the instruction.
- The bit format of the instruction.
- The modes for which the instruction is valid.
- Detailed information describing the instruction's action.
- Information about how the instruction affects LINK, CBIT, and the condition codes.

Notation Conventions

Several abbreviations and symbols are used throughout this dictionary. Table 13-1 defines the dictionary notation.

Table 13-1
Dictionary Notation

Symbol	Meaning
A	The A register.
ADDRESS	Encompasses all the elements needed to specify an effective address. This term is used because various types of addressing require you to specify the elements in different orders (such as indirect or pre- and post-indexing).
AP	Address pointer.
B	The B register.
BR	Base register.
CB	Class bits.
CBIT	Bit 1 of the keys.
DAC	The double precision floating-point accumulator with 48 bits of mantissa and 16 bits of exponent.
E	The E register.
EA	Effective address.
F	Floating-point accumulator.
FAC	The single precision floating-point accumulator with 48 bits of mantissa and 16 bits of exponent.
FAR	Field address register.
FLR	Field length register.
I	Indirect bit.
L	The 32 bit L register.
LINK	Bit 3 of the keys.
QAC	The quad precision floating-point accumulator with 96 bits of mantissa and 16 bits of exponent.
skip	Skip the next 16-bit word before continuing execution.
X	The X register (indexing).

Table 13-1 (continued)
Dictionary Notation

Symbol	Meaning
XB	Auxiliary base register.
Y	The Y register (indexing).
m\n	Specifies the number of bits, n, occupied by a field, m.
[ ]	Specifies an optional argument.

Resumable Instructions

Some assembly language instructions are resumable. When an interrupt occurs during instruction execution, the processor usually services the interrupt, then restarts the interrupted instruction from the beginning. Some instructions, however, are too long or too complex for this to be desirable. When an interrupt occurs during one of these resumable instructions, the processor preserves the state of the interrupted instruction, handles the interrupt, then resumes the instruction at the point where the interrupt occurred. Table 13-2 lists the resumable assembly language instructions.

Table 13-2
Resumable Instructions

Instructions			
ARGT	XAD	XBTD	XCM
XDTB	XDV	XED	XMP
XMV	ZCM	ZED	ZFIL
ZMV	ZMVD	ZTRN	STEX

Storing Data into the 9950 Instruction Stream

After any instruction that stores data into memory, you must wait five instructions before executing data. If in doubt about the next five instructions (temporally) to be executed, a mode change instruction to the current addressing mode, such as E64V, allows the stored data to be executed.

Instruction Formats

All S, R, and V mode instructions belong to one of the following instruction types:

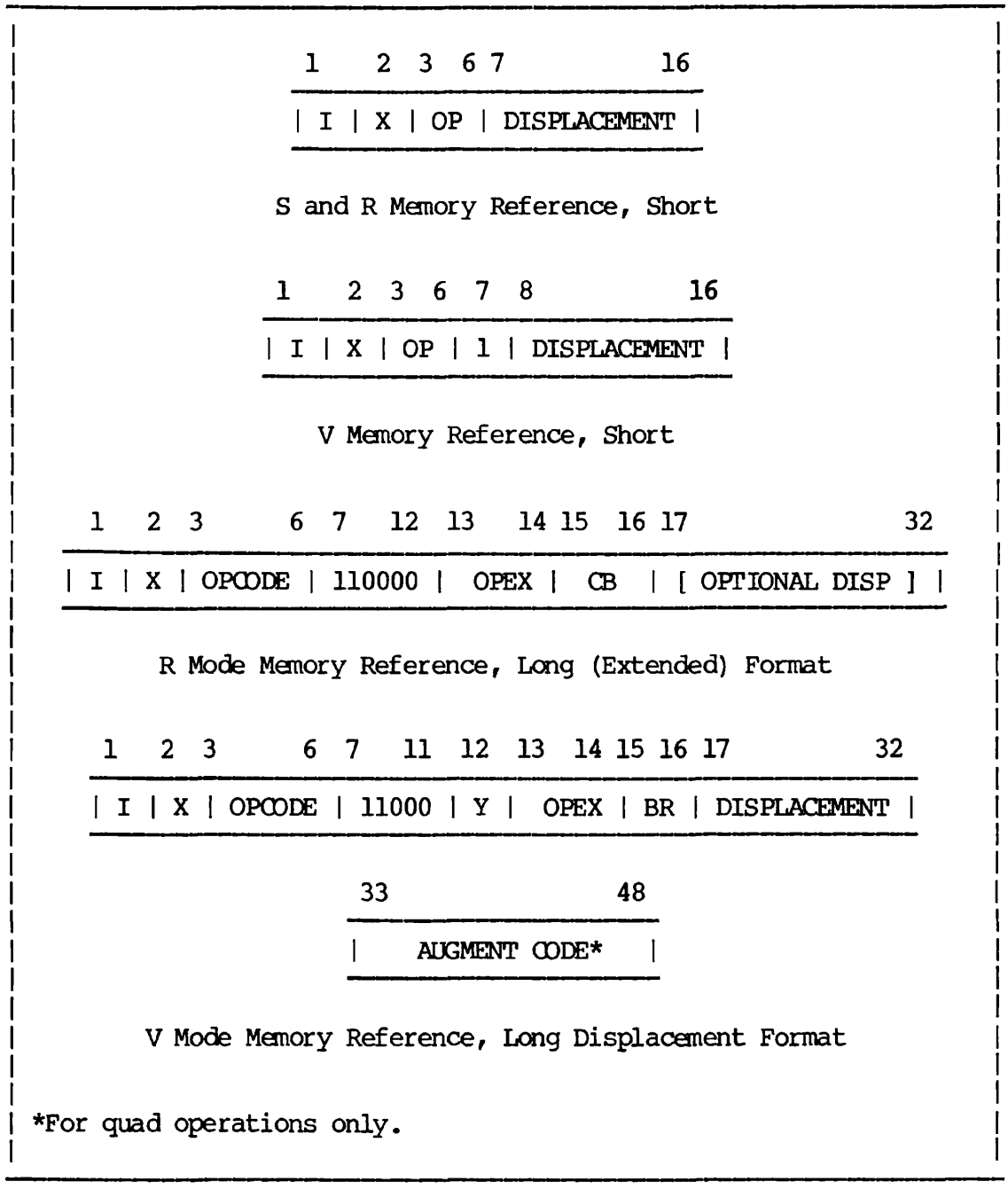
- S and R Mode Memory Reference, Short
- V Mode Memory Reference, Short
- R Mode Memory Reference, Long
- V Mode Memory Reference, Long
- V Mode Generic AP (Address Pointer)
- S, R, and V Mode Generic Type A
- S, R, and V Mode Generic Type B
- S, R, and V Mode Shift
- S, R, and V Mode Skip

The format of each instruction type is shown in Figure 13-1.

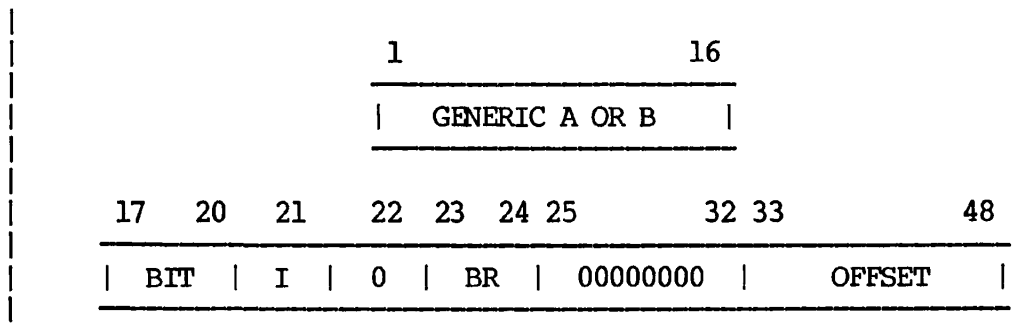
Short and long memory reference instructions have an opcode in bits 3-6. The value of this opcode ranges from 1 to '17, inclusive, with the exception of '14, which is reserved for I/O. For opcode '15, the X bit is part of the opcode.

In addition, long memory reference instructions have an opcode extention contained in bits 13-14. Generic AP instructions have a generic A or B format (where bits 7-16 contain the opcode extension) followed by a 32-bit address pointer.

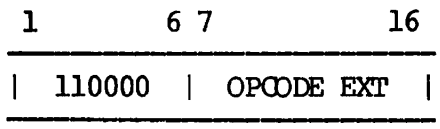
Generic A and B, shift, and skip instructions are 16 bits long, all of which form an opcode. The values of bits 1 and 2 determine the basic instruction type: 11 for Generic A, 00 for Generic B, 01 for shifts, and 10 for skips. Bits 3-6 contain 0. Bits 7-16 contain an opcode extension. For shifts, bits 10-16 of the opcode extension contain the two's complement of the number of shifts to perform.



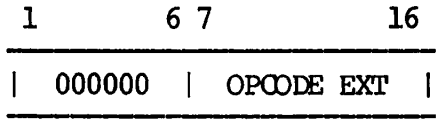
S, R, V Mode Instruction Formats
Figure 13-1



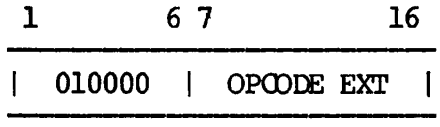
Generic AP Format



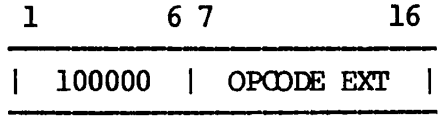
S, R, V Modes Generic A Format



S, R, V Modes Generic B Format



S, R, V Mode Shift Format



S, R, V Mode Skip Format

S, R, and V Mode Instruction Formats
Figure 13-1 (continued)

INSTRUCTIONS

▶ AlA
Add 1 to A
1 1 0 0 0 0 1 0 1 0 0 0 0 1 1 0 (S, R, V mode form)

Adds 1 to the contents of A and stores the result in A. If A initially contains (2**15)-1, an integer exception occurs and the instruction loads -(2**15) into A. If no integer exception occurs, the instruction resets CBIT to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ A2A
Add 2 to A
1 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 (S, R, V mode form)

Adds 2 to the contents of A and stores the result in A. If A initially contains (2**15)-1 or (2**15-2), an integer exception occurs and the instruction loads -(2**15) or -(2**15)+1, respectively, into A. If no exception occurs, the instruction resets CBIT to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ ABQ address
Add Entry to Bottom of Queue
1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 (V mode form)
AP\32

Adds the entry contained in A to the bottom of the queue referenced by the AP. (AP points to the queue's QCB.) Sets the condition codes to reflect EQ if the queue is full, or to NE if not full. Leaves the values of CBIT and LINK unchanged. See Chapters 5 and 12 for more information about queues and queue operations.

▶ ACA
Add CBIT to A
1 1 0 0 0 0 1 0 1 0 0 0 1 1 1 0 (S, R, V mode form)

Adds the value of CBIT to the contents of A and stores the result in A. If the initial value of A is (2**15)-1 and CBIT is 1, the instruction loads -(2**15) into A and an integer exception occurs. If no integer exception occurs, the instruction resets CBIT to 0. The condition codes reflect the result of the operation. (See Table 5-3.) LINK contains the carry-out bit.

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

This instruction adds CBIT to bit 16 of A.

▶ ADD address
Add
I X 0 1 1 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 1 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Fetches the 16-bit contents of the location specified by EA and adds them to the contents of A. Stores the results in A.

If the resulting sum is less than or equal to (2**15)-1 and greater than -(2**15), the instruction resets CBIT to 0. If the sum is greater than or equal to 2**15, an integer exception occurs and the instruction loads -(2**15)-1 into A. If the sum is less than or equal to -(2**15)-1, an integer exception occurs and the instruction loads +(2**15)-1 into A.

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

At the end of the operation, LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ ADL address
Add Long
I X 0 1 1 0 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Fetches the 32-bit contents of the location specified by EA and adds them to the contents of L. Stores the results in L.

If the resulting sum is less than or equal to (2**31)-1 and greater than or equal to -(2**31), the instruction resets CBIT to 0. If the sum is greater than or equal to (2**31), an integer exception occurs and the instruction loads -(2**31)-1 into L. If the sum is less than or equal to -(2**31)-1, an integer exception occurs and the instruction loads +(2**31)-1 into L.

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

At the end of the operation, LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ ADLL
Add LINK to L
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 (V mode form)

Adds the contents of LINK to the contents of L and stores the result in L. If the initial value of L is (2**31)-1 and LINK is 1, an integer exception occurs and the instruction loads -(2**31) into L. If no integer exception occurs, the instruction resets CBIT to 0. The condition codes reflect the result of the operation. (See Table 5-3.) LINK contains the carry-out bit.

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

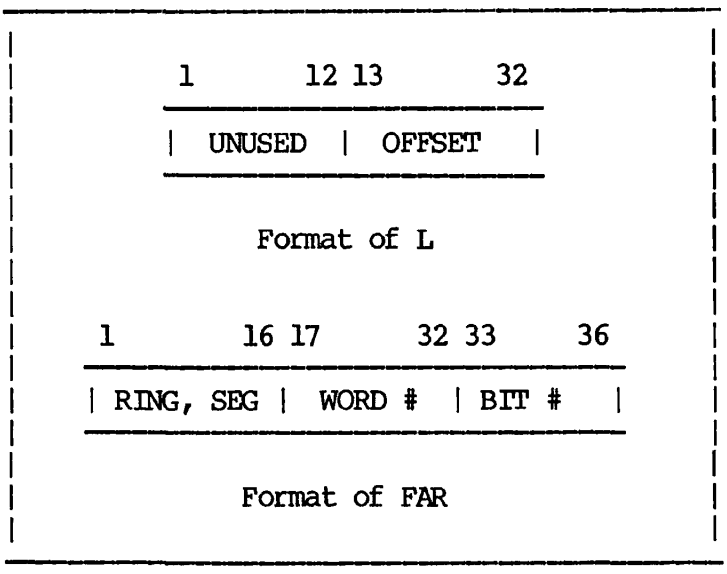
Note

This instruction adds the value of LINK to bit 32 of L.

▶ ALFA far
Add L to FAR
0 0 0 0 0 0 1 0 1 1 0 0 F 0 0 1 (V mode format)

Adds the offset contained in L to the word and bit number fields of FAR and stores the result in the specified FAR. The values of the condition codes remain unchanged. Leaves the values of LINK and CBIT indeterminate.

Figure 13-2 shows the format of L and the specified FAR for this instruction.



L and FAR Format for ALFA
Figure 13-2

▶ ALL n
A Left Logical
0 1 0 0 0 0 1 1 0 0 N\6 (S, R, V mode form)

Shifts the contents of A left the appropriate number of bits, bringing 0s into bit 16. CBIT contains the value of the last bit shifted out; the values of the other bits shifted out are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged. See Chapter 5 for more information about shifts.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ ALR n
A Left Rotate
0 1 0 0 0 0 1 1 1 0 N\6 (S, R, V mode form)

Shifts the contents of A to the left, rotating bit 1 into bit 16. Stores the result in A. CBIT contains the value of the last bit rotated into bit 16. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged. See Chapter 5 for more information about shifts.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ ALS n
A Arithmetic Left Shift
0 1 0 0 0 0 1 1 0 1 N\6 (S, R, V mode form)

Shifts the contents of A to the left, bringing 0s in on the right. Stores the result in A. If bit 1, the sign bit, changes state, the shift has resulted in a loss of significance and produces an integer exception. If no integer exception occurs, the instruction resets CBIT to 0. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged. See Chapter 5 for more information about shifts.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ ANA address
AND to A
I X 0 0 1 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 0 1 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 0 1 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Logically ANDs the 16-bit contents of the location specified by EA with the contents of A and stores the result in A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ANL address
And to A Long
I X 0 0 1 1 1 1 0 0 0 Y 1 1 BR\2 (V mode form)

Calculates a 32-bit effective address, EA. Logically ANDs the 32-bit contents of the location specified by EA with the contents of L and stores the result in L. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ARGT
Argument Transfer
0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 (V mode form)

Transfers arguments from a source procedure to a destination procedure. ARGT is fetched and executed only when the argument transfer phase of a procedure call (PCL) instruction is interrupted or faulted.

To perform a procedure call and argument transfer, the source procedure must contain the PCL instruction followed by a number of argument templates. The destination procedure must begin with the ARGT instruction. When the PCL instruction is executed, control transfers to the destination procedure, and the ARGT instruction uses the templates to form the actual arguments. The arguments are stored in the new stack frame as they are computed. At the end of the ARGT instruction, the values of LINK, CBIT, and the condition codes are indeterminate.

Note that ARGT must be the first executable instruction in any destination procedure that will use arguments. For those procedures whose entry control blocks specify zero arguments, omit ARGT.

For more information about argument transfers, refer to the section on procedure calls in Chapter 8.

▶ ARL n
A Right Logical
0 1 0 0 0 0 0 1 0 0 N\6 (S, R, V mode form)

Shifts the contents of A right the appropriate number of bits, bringing 0s into bit 1. CBIT contains the value of the last bit shifted out; the values of the other bits shifted out are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ ARR n
A Right with Rotate
0 1 0 0 0 0 0 1 1 0 N\6 (S, R, V mode form)

Shifts the contents of A to the right, rotating bit 16 into bit 1. CBIT contains the value of the last bit rotated into bit 1. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ ARS n
A Right Shift
0 1 0 0 0 0 0 1 0 1 N\6 (S, R, V mode form)

Shifts the contents of A to the right arithmetically, shifting copies of bit 1, the sign bit, into the vacated bits. CBIT contains the value of the last bit shifted out; the values of the other bits shifted out are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ ATQ address
Add Entry to Top of Queue
1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 1 (V mode form)
AP\32

Adds the entry contained in A to the top of the queue referenced by the AP. (AP points to the queue's QCB.) Sets the condition codes to reflect EQ if the queue is full, or to NE if not full. Leaves the values of CBIT and LINK unchanged. For more information about queues and queue manipulation, see Chapters 5 and 12.

▶ BCEQ address
Branch on Condition Code EQ
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 (V mode form)
ADDRESS\16

If the condition codes reflect equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCGE address
Branch on Condition Code GE
1 1 0 0 0 0 1 1 1 0 0 0 0 1 0 1 (V mode form)
ADDRESS\16

If the condition codes reflect greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCGT address
Branch on Condition Code GT
1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1 (V mode form)
ADDRESS\16

If the condition codes reflect greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCLE address
Branch on Condition Code LE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 (V mode form)
ADDRESS\16

If the condition codes reflect less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCLT address
Branch on Condition Code LT
1 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0 (V mode form)
ADDRESS\16

If the condition codes reflect less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCNE address
Branch on Condition Code NE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 (V mode form)
ADDRESS\16

If the condition codes reflect not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCR address
Branch on CBIT Reset to 0
1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 (V mode form)
ADDRESS\16

If CBIT has the value 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If CBIT has the value 1, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCS address
Branch on CBIT Set to 1
1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 0 (V mode form)
ADDRESS\16

If CBIT has the value 1, the instruction loads the specified address into the program counter. This address must be within the current segment. If CBIT has the value 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BDX address
Branch on Decremented X
1 1 0 0 0 0 0 1 1 1 0 1 1 1 0 0 (V mode form)
ADDRESS\16

Decrements the contents of X by one and stores the result in X. If the decremented value is not equal to 0, loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BDY address
Branch on Decremented Y
1 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 (V mode form)
ADDRESS\16

Decrements the contents of Y by one and stores the result in Y. If the decremented value is not equal to 0, loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BEQ address
Branch on A Equal to 0
1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 0 (V mode form)
ADDRESS\16

If the contents of A are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are not equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFEQ address
Branch on Floating Accumulator Equal to 0
1 1 0 0 0 0 1 1 1 0 0 0 1 0 1 0 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are not equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFGE address
Branch on Floating Accumulator Greater Than or Equal to 0
1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 1 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are less than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFGT address
Branch on Floating Accumulator Greater Than 0
1 1 0 0 0 0 1 1 1 0 0 0 1 0 0 1 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are less than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFLE address
Branch on Floating Accumulator Less Than or Equal to 0
1 1 0 0 0 0 1 1 1 0 0 0 1 0 0 0 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are greater than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFLT address
Branch on Floating Accumulator Less Than 0
1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are greater than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFNE address
Branch on Floating Accumulator Not Equal to 0
1 1 0 0 0 0 1 1 1 0 0 0 1 0 1 1 (V mode form)
ADDRESS\16

If the contents of the floating accumulator are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BGE address
Branch on A Greater Than or Equal to 0
1 1 0 0 0 0 0 1 1 0 0 0 1 1 0 1 (V mode form)
ADDRESS\16

If the contents of A are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are less than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BGT address
Branch on A Greater Than 0
1 1 0 0 0 0 0 1 1 0 0 0 1 0 0 1 (V mode form)
ADDRESS\16

If the contents of A are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are less than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BIX address
Branch on Incremented X
1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 (V mode form)
ADDRESS\16

Increments the contents of X by one and stores the result in X. If the incremented value is not equal to 0, loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BIY address
Branch on Incremented Y
1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 (V mode form)
ADDRESS\16

Increments the contents of Y by one and stores the result in Y. If the incremented value is not equal to 0, loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BLE address
Branch on A Less Than or Equal to 0
1 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 (V mode form)
ADDRESS\16

If the contents of A are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are greater than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLEQ address
Branch on L Equal to 0
1 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 (V mode form)
ADDRESS\16

If the contents of L are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are not equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLGE address
Branch on L Greater Than or Equal to 0
1 1 0 0 0 0 0 1 1 0 0 0 1 1 0 1 (V mode form)
ADDRESS\16

If the contents of L are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are less than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLGT address
Branch on L Greater Than 0
1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 (V mode form)
ADDRESS\16

If the contents of L are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are less than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLLE address
Branch on L Less Than or Equal to 0
1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 (V mode form)
ADDRESS\16

If the contents of L are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are greater than 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLLT address
Branch on L Less Than 0
1 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0 (V mode form)
ADDRESS\16

If the contents of L are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are greater than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLNE address
Branch on L Not Equal to 0
1 1 0 0 0 0 0 1 1 1 0 0 0 0 1 1 (V mode form)
ADDRESS\16

If the contents of L are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the L contents are equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BLR address
Branch on LINK Reset to 0
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 (V mode form)
ADDRESS\16

If LINK has the value 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If LINK has the value 1, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BLS address
Branch on LINK Set to 1
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 0 (V mode form)
ADDRESS\16

If LINK has the value 1, the instruction loads the specified address into the program counter. This address must be within the current segment. If LINK has the value 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BLT address
Branch on A Less Than 0
1 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0 (V mode form)
ADDRESS\16

If the contents of A are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are greater than or equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BMEQ address
Branch on Magnitude Condition EQ
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 (V mode form)
ADDRESS\16

Performs the same operation as the BCEQ instruction, except that it allows the result to be evaluated as unsigned.

▶ BMGE address
Branch on Magnitude Condition GE
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 0 (V mode form)
ADDRESS\16

Performs the same function as the BLS instruction does, except that it allows the result to be evaluated as unsigned.

▶ BMGT address
Branch on Magnitude Condition GT
1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 0 (V mode form)
ADDRESS\16

If LINK is 1 and the condition codes reflect not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If some other condition exists, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BMLE address
Branch on Magnitude Condition LE
1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 1 (V mode form)
ADDRESS\16

If LINK is 0 or the condition codes reflect equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If some other condition exists, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BMLT address
Branch on Magnitude Condition LT
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 (V mode form)
ADDRESS\16

Performs the same function as the BLR instruction does, except that it allows the result to be evaluated as unsigned.

▶ BMNE address
Branch on Magnitude Condition NE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 (V mode form)
ADDRESS\16

Performs the same function as the BLNE instruction does, except that it allows the result to be evaluated as unsigned.

▶ BNE address
Branch on A Not Equal to 0
1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 (V mode form)
ADDRESS\16

If the contents of A are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the A contents are equal to 0, execution continues with the next instruction. The condition codes contain the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ CAI
Clear Active Interrupt
0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 (S, R, V mode form)

Clears the current active interrupt. Effective only in vectored interrupt mode. Inhibits interrupts for one instruction. Leaves the values of LINK, CBIT,and the condition codes unchanged.

Note

This is a restricted instruction.

▶ CAL
Clear A Left Byte
1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 (S, R, V mode form)

Clears the left byte of A to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CALF address
Call Fault Handler
0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 (V mode form)
AP\32

The address pointer in this instruction is to the ECB of a fault routine. The instruction uses this pointer to transfer control to the fault routine as if the transfer were a normal procedure call. The values of CBIT, LINK, and the condition codes are indeterminate. See Chapter 11 for more information.

▶ CAR
Clear A Right Byte
1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 (S, R, V mode form)

Clears the right byte of A to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CAS address
Compare A and Skip
I X 1 0 0 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 0 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 0 0 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Compares the contents of the A register to the contents of the location specified by EA and skips as follows:

Condition	Skip
Contents of A > contents of EA.	No skip.
Contents of A = contents of EA.	Skip one word.
Contents of A < contents of EA.	Skip two words.

LINK contains the carry-out bit. The value of CBIT is unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ CAZ
Compare A With 0
1 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 (S, R, V mode form)

Compares the contents of A with 0. Skips as follows:

Condition	Skip
Contents of A > 0.	No skip.
Contents of A = 0.	Skip one word.
Contents of A < 0.	Skip two words.

LINK contains the carry-out bit. The value of CBIT remains unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ CEA
Compute Effective Address
0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 (S, R mode form)

Interprets the contents of A as a 16-bit indirect address in the current addressing mode. Calculates an effective address, EA, from the indirect address and loads the final address into A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CGT
Computed GOTO
0 0 0 0 0 0 1 0 1 1 0 0 1 1 0 0 (V mode form)
INTEGER N\16
BRANCH ADDRESS 1\16
..
BRANCH ADDRESS (N-1)\16

If the contents of A are greater than or equal to 1 and less than the specified integer N that follows the opcode, the instruction adds the contents of A to the contents of the program counter to form an address. (The program counter points to the integer N following the opcode.) Loads the contents of the location specified by this address into the program counter. If the contents of A are not within this range, the instruction adds integer N to the contents of the program counter and stores the result in the program counter. The values of CBIT, LINK, and the condition codes are indeterminate.

Note

Each of the branch addresses following the instruction specifies a location within the current procedure segment.

▶ CHS
Change Sign
1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 (S, R, V mode form)

Complements bit 1 of A. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CLS
Compare Long and Skip
I X 1 0 0 1 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Compares the contents of L to the contents of the 32-bit location specified by EA and skips as follows:

Condition	Skip
Contents of LINK > contents of EA.	No skip.
Contents of LINK = contents of EA.	Skip one word.
Contents of LINK < contents of EA.	Skip two words.

LINK contains the carry-out bit. The value of CBIT is unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ CMA
Complement A
1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 (S, R, V mode form)

Forms the one's complement of the contents of A by inverting the value of each bit, and stores the result in A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CRA
Clear A to 0
1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 (S, R, V mode form)

Clears the contents of A to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CRB
Clear B to 0
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 (S, R, V mode form)
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0

Clears the contents of B to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

Opcode 140014 executes both a CRB and a FDBL. This is a conversion aid for P300 programs. This opcode should not be used; it is implemented for compatibility's sake only.

▶ CRE
Clear E to 0
1 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 (V mode form)

Clears the contents of E to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CREP address
Call Recursive Entry Procedure
I X 1 0 0 0 1 1 0 0 0 0 1 0 CB\2 (R mode form)
[DISPLACEMENT\16]

Increments the contents of the program counter and loads the result into the location following the one specified by the current value of the R mode stack pointer. Calculates an effective address, EA, and loads it into the program counter. Execution continues with the location specified by the new value of the program counter.

This instruction performs subroutine linkage for reentrant or recursive procedures. CREP stores the return address in the second word of a stack frame created by the ENTR instruction, rather than in the destination address as JST does.

▶ CRL
Clear L to 0
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 (S, R, V mode form)

Clears the contents of L to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CRLE
Clear L and E to 0
1 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 (V mode form)

Clears the contents of E and L to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CSA
Copy Sign of A
1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 (S, R, V mode form)

Sets CBIT equal to the value of bit 1 of A and clears bit 1 of A to 0. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ DAD address
Double Add
I X 0 1 1 0 1 1 0 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]
I X 0 1 1 0 DISPLACEMENT\10 (S, R mode form)

Calculates an effective address, EA. Fetches the 31-bit contents of the location specified by EA and adds them to the 31-bit contents of A and B. Stores the result in A and B.

If the result is greater than or equal to 2**30, an integer exception occurs and the instruction loads bit 1 of A with a 1, and bits 2-16 of A and bits 2-16 of B with (result - (2**30)). Bit 1 of B contains 0.

If the result is less than -(2**30), an integer exception occurs and the instruction loads bit 1 of A with a 0 and bits 2-16 of A and bits 2-16 of B with the negative of (result + (2**30)). Bit 1 of B contains 0. If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

At the end of the instruction, LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

Notes

1. Bit 17 of each 31-bit integer must be 0. If nonzero, unpredictable results will occur.

2. This instruction executes in double precision mode only.

▶ DBL
Enter Double Precision Mode
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 (S, R mode form)

Enters double precision mode by setting bit 2 of the keys to 1. Subsequent LDA, STA, ADD, and SUB instructions manipulate 31-bit integers and are interpreted as DLD, DST, DAD, and DSB, respectively. Leaves the values of LINK, CBIT, and the condition codes unchanged. In V or I mode, bit 2 of the keys has no effect.

▶ DFAD address
Double Precision Floating Add
I X 0 1 1 0 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 0 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the double precision number in the location specified by EA to the 64-bit contents of the DAC. (See Chapter 6 for more information.) Normalizes the result and loads it into the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

For 750 and 850 processors, exponent underflow is detected, but exponent overflow is not.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ DFCM
Double Precision Floating Complement
1 1 0 0 0 0 0 1 0 1 1 1 1 1 0 0 (R, V mode form)

Forms the two's complement of the double precision number in the DAC and normalizes it if necessary. (See Chapter 6.) Stores the result in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ DFCS address
Double Precision Floating Point Compare and Skip
I X 1 0 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 0 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the contents of the DAC (explained in Chapter 6) to the contents of the 64-bit location specified by EA and skips as follows:

Condition	Skip
DAC contents > EA contents.	No skip.
DAC contents = EA contents.	Skip one word.
DAC contents < EA contents.	Skip two words.

The values of CBIT, LINK, and the condition codes are indeterminate.

▶ DFDV address
Double Precision Floating Point Divide
I X 1 1 1 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the contents of the DAC by the contents of the location specified by EA. (See Chapter 6.) Normalizes the result and stores the whole quotient in the DAC. An overflow or a divide by 0 causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ DFLD address
Double Precision Floating Point Load
I X 0 0 1 0 1 1 0 0 0 Y 1 0 BR\2 (V mode long form)
DISPLACEMENT\16

I X 0 0 1 0 1 1 0 0 0 0 1 0 CB\2 (R mode long form)
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the 64-bit contents of the location specified by EA into the DAC. (See Chapter 6.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction does not normalize the result before loading it into the DAC.

▶ DFLX address
Double Precision Floating Point Load Index
I 0 1 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I 0 1 1 0 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the index register, X, with four times the 16-bit contents of the location specified by EA. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction cannot do indexing.

▶ DFMP address
Double Precision Floating Point Multiply
I X 1 1 1 0 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 0 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the contents of the DAC by the 64-bit contents of the location specified by EA. (See Chapter 6.) Normalizes the result, if necessary, and stores it in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ DFSB address
Double Precision Floating Point Subtract
I X 0 1 1 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 64-bit contents of the locations specified by EA from the contents of the DAC. (See Chapter 6.) Loads the result in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

For 750 and 850 processors, exponent underflow is detected, but exponent overflow is not.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DFST address
Double Precision Floating Point Store
I X 0 1 0 0 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 0 0 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the DAC into the location specified by EA. (See Chapter 6.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction does not normalize the result before loading it into the specified memory location.

▶ DIV address
Divide
I X 1 1 1 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 1 1 1 DISPLACEMENT\10 (S mode; R mode short)

Calculates an effective address, EA. Divides the 31-bit contents of A and B by the 16-bit contents of the location specified by EA. Stores the 16-bit quotient in A and the 16-bit remainder in B. The sign of the remainder equals the sign of the dividend.

Overflow occurs when the quotient is less than -(2**15) or greater than (2**15)-1. An overflow or a divide by 0 causes an integer exception. If no integer exception occurs, CBIT is reset to 0. This instruction leaves the values of LINK and the condition codes indeterminate.

If an integer exception occurs when bit 8 of the keys contains a 1, the instruction sets CBIT to 1. If bit 8 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DIV address
Divide
I X 1 1 1 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 1 DISPLACEMENT\10 (V mode short)

Calculates an effective address, EA. Divides the contents of L by the 16-bit contents of the location specified by EA. Stores the 16-bit quotient in A and the 16-bit remainder in B. The sign of the remainder equals the sign of the dividend.

When the quotient is less than -(2**15) or greater than (2**15)-1, an overflow occurs, causing an integer exception. A divide by 0 also causes an integer exception. If no integer exception occurs, CBIT is reset to 0. This instruction leaves the values of LINK and the condition codes indeterminate.

If the integer exception occurs when bit 8 of the keys is 0, the instruction sets CBIT to 1. If bit 8 is 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DLD
Double Load
I X 0 0 1 0 1 1 0 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 0 1 0 DISPLACEMENT\10 (S mode; R mode short)

Calculates an effective address, EA. Loads the 16-bit contents of the location specified by EA into A, and the 16-bit contents of the location specified by EA+1 into B. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction executes only in double precision mode.

▶ DRN
Double Round from Quad
0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 (V mode form)

Converts the value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 50-96 of QAC are not 0, or bit 48 of QAC contains 1, the instruction adds the value of bit 49 to that of bit 48 (unbiased round) and clears bits 49-96 of QAC to 0. If any other condition exists, no unbiased rounding occurs but bits 49-96 of QAC are still cleared to 0. After any rounding and clearing occurs, the instruction normalizes the result and loads it into bits 1-64 of QAC.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNM
Double Round from Quad towards Negative Infinity
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 1 (V mode form)

Converts the value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 49-96 of QAC contain 0s, the instruction ends. In any other case, the instruction clears bits 49-96 to 0, normalizes the result, and places it in bits 1-64 of QAC.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNP
Double Round from Quad towards Positive Infinity
0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 (V mode form)

Converts the value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 49-96 of QAC contain 0s, the instruction ends. In any other case, the instruction adds 1 to the value contained in bit 48 of QAC, clears bits 49-96 to 0, the instruction normalizes the result and places it in bits 1-64 of QAC.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNZ
Double Round from Quad towards Zero
0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 0 (V mode form)

Converts the value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 49-96 of QAC contain 0s and bit 1 contains 1, the instruction adds 1 to the value contained in bit 48 of QAC, clears bits 49-96 to 0, normalizes the result and places it in bits 1-64 of QAC. If any other condition exists, no rounding occurs.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRX
Decrement and Replace X
1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 (S, R, V mode form)

Decrements the contents of X by 1 and stores the result in X. Skips the next memory location if the decremented value is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ DSB address
Double Subtract
I X 0 1 1 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 1 1 DISPLACEMENT\10 (S mode; R mode short)

Calculates an effective address, EA. Fetches the 31-bit integer contained in the locations specified by EA and EA+1 and subtracts it from the 31-bit integer contained in A and B. Stores the result in in A and B.

If the result is greater than or equal to 2**30, an integer exception occurs and the instruction loads bit 1 of A with 1, and bits 2-16 of A and 2-16 of B with the absolute value of (result - (2**30)). Bit 1 of B must be 0.

If the result is less than -(2**30), an integer exception occurs and the instruction loads bit 1 of A with a 0, and bits 2-16 of A and bits 2-16 of B with the negative of (result + (2**30)). Bit 1 of B must be 0.

If no integer exception occurs, CBIT is reset to 0. At the end of the instruction, LINK contains the borrow bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Notes

1. Bit 17 of each 31-bit integer must be 0 or indeterminate results occur.

2. This instruction executes in double precision mode only.

3. To negate a 31-bit integer, subtract it from 0.

▶ DST address
Double Store
I X 0 1 0 0 1 1 0 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 0 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Stores the contents of A at the location specified by EA, and the contents of B at the location specified by EA+1. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction executes only in double precision mode.

▶ DVL address
Divide Long
I X 1 1 1 1 1 1 0 0 0 Y 1 1 BR\2 (V mode long)
DISPLACEMENT\16

Calculates an effective address, EA. Divides the 64-bit contents of L and E by the 32-bit contents of the location specified by EA. Stores the quotient in L and the remainder in E. An overflow or divide by 0 causes an integer exception. If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ E16S
Enter 16S Mode
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 (S, R, V mode form)

Sets bits 4-6 of the keys to 000. Subsequent S mode instructions may now be interpreted, and 16S address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32I
Enter 32I Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 (S, R, V mode form)

Sets bits 4-6 of the keys to 100. Subsequent I mode instructions may now be interpreted, and 32I address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32R
Enter 32R Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1 (S, R, V mode form)

Sets bits 4-6 of the keys to 011. Subsequent R mode instructions may now be interpreted, and 32R address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32S
Enter 32S Mode
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 (S, R, V mode form)

Sets bits 4-6 of the keys to 001. Subsequent S mode instructions may now be interpreted, and 32S address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E64R
Enter 64R Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 (S, R, V mode form)

Sets bits 4-6 of the keys to 010. Subsequent R mode instructions may now be interpreted, and 64R address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E64V
Enter 64V Mode
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 (S, R, V mode form)

Sets bits 4-6 of the keys to 110. Subsequent V mode instructions may now be interpreted, and 64V address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

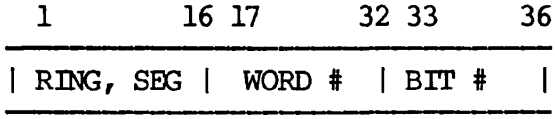
▶ EAA address
Effective Address to A
I X 0 0 0 1 1 1 0 0 0 0 0 1 CB\2 (R mode form)

Calculates an effective address, EA, and loads it into A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EAFA far,address
Effective Address to FAR
0 0 0 0 0 0 1 0 1 1 0 0 FAR 0 0 0 (V mode form)
AP\32

Builds a 36-bit EA from the 32-bit address pointer contained in the instruction and loads it into the specified FAR. Note that the AP bit field is processed and loaded into the bit portion of the FAR for direct access;; indirection uses the bit field in the indirect pointer (if any). Leaves the values of CBIT, LINK, and the condition codes unchanged.

Figure 13-3 shows the format of the EA loaded into the specified FAR.



EA Format for EAFA
Figure 13-3

▶ EAL address
Effective Address to L
I X 0 0 0 1 1 1 0 0 0 Y 0 1 BR\2 (V mode form)

Calculates an effective address, EA, and loads it into L. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EALB address
Effective Address to LB
I X 1 0 1 1 1 1 0 0 0 Y 1 0 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA, and loads it into LB. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ EAXB address
Load XB with Effective Address
I X 1 0 1 0 1 1 0 0 0 Y 1 0 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA, and loads it into XB. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EIO address
Execute I/O
I 0 1 1 0 0 1 1 0 0 0 Y 0 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Executes bits 17-32 of EA as if the bits were an extended PIO instruction. If execution is successful, the instruction sets the condition codes as follows:

CC	Meaning
EQ	Successful INA, OTA, or SKS instruction
NE	Unsuccessful INA, OTA, OR SKS; all OCP

Leaves the values of LINK and CBIT unchanged. See Chapter 12 for more information.

Note

This is a restricted instruction.

▶ EMCM
Enter Machine Check Mode
0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 (S, R, V mode form)

Enters machine check mode 3 by loading 3 into modal bits 15-16. This mode enables the reporting of all errors. The actions taken upon an error depend on whether the machine was in process exchange mode or not.

The instruction inhibits interrupts during execution of the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged. See Chapter 11 for more information about checks.

If an error occurs in process exchange mode, the microcode stores the machine state in the appropriate check vector and transfers control to that vector, automatically dropping back to machine check mode 0.

If an error occurs when the machine is not in process exchange mode, the following actions occur. If the appropriate check vector contains a nonzero value, the processor jumps indirectly through this vector to the check routine. If the check vector location contains 0, the machine halts.

Note

This is a restricted instruction.

▶ ENB
Enable Interrupts
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 (S, R, V mode form)

Enables interrupts by setting bit 1 of the modals to 1. Interrupts remain inhibited for the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENBL
Enable Interrupts (Local)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 (S, R, V mode form)

This 850 instruction performs the same actions as ENB except that it is performed specifically for the local processor. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

ENBL is a restricted instruction.

▶ ENBM
Enable Interrupts (Mutual)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 (S, R, V mode form)

For the 850, a processer checks the availability of the mutual exclusion lock. If available, the processor sets this lock and enables interrupts. Otherwise, it waits for the lock to be released by the other processor and then sets the lock and enables interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENBP
Enable Interrupts (Process)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 (S, R, V mode form)

For the 850, a processor checks the availability of the process exchange lock. If available, the processor sets this lock and enables interrupts. Otherwise, it waits for this lock to be released by the other processor and then sets the lock and enables interrupts. Leves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENTR n
Enter R Mode Recursive Procedure Stack
I X 0 0 0 1 1 1 0 0 0 0 1 1 CB\2 (R mode long form)
[DISPLACEMENT\16]

Creates a save area n words long and saves the current value of the R mode stack pointer in the first word of the save area. The starting address of the save area is:

(contents of R mode stack pointer) - n

This means that the instruction creates a stack frame containing n locations, and that the first location points to the previous frame.

The ENTR instruction leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ ERA address
Exclusive OR to A
I X 0 1 0 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 0 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 0 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Exclusively ORs the contents of the location specified by EA and the contents of A. Stores the results in A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ERL
Exclusive Or to L
I X 0 1 0 1 1 1 0 0 0 Y 1 1 BR\2 (V mode long)
DISPLACEMENT\16

Calculates an effective address, EA. Exclusively ORs the contents of L with the contents of the 32-bit location specified by EA. Stores the results in L. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ ESIM
Enter Standard Interrupt Mode
0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 (S, R, V mode form)

Enters standard interrupt mode by resetting bit 2 of the modals to 0. Inhibits interrupts for one instruction. ESIM is meaningless when in the system is in process exchange mode (that is, the value of modal bit 13 is 1). All interrupts use location '63. The processor services interrupts according to their relative positions on the I/O bus. Lower devices have higher priority. Inhibits interrupts during execution of the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged. Refer to Chapter 11 for more information about interrupts.

Note

This is a restricted instruction.

▶ EVIM
Enter Vectored Interrupt Mode
0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 (S, R, V mode form)

Enters vectored interrupt mode by setting bit 2 of the modals to 1. EVIM is meaningless when in the system is in process exchange mode (that is, the value of modal bit 13 is 1). The processor services interrupts according to their relative positions on the I/O bus. Lower devices have higher priority. Interrupts occur through a location specified by the interrupting device. Inhibits interrupts during execution of the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged. Refer to Chapter 11 for more information about interrupts.

Note

This is a restricted instruction.

▶ FAD address
Floating Add
I X 0 1 1 0 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 0 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the contents of the location specified by EA to the contents of the FAC. (See Chapter 6.) Stores the result in the FAC and normalizes it if necessary. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FCDQ
Floating Point Convert Double to Quad
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 1 (V mode form)

Clears FAC0 to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ FCM
Floating Point Complement
1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0 (R, V mode form)

Forms the two's complement of the FAC mantissa and normalizes the result if necessary. (See Chapter 6.) Stores the result in the FAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FCS address
Floating Compare and Skip
I X 1 0 0 1 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 0 1 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. In rounding mode, the instruction rounds the contents of DAC, then compares the rounded value to the contents of the memory location specified by EA. In normal mode, no rounding occurs before the compare. (See Chapter 6 for more information.) The compare results in a skip as follows:

Condition	Skip
FAC contents > EA contents.	No skip.
FAC contents = EA contents.	Skip one word.
FAC contents < EA contents.	Skip two words.

The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FDBL
Floating Point Convert Single to Double
1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 (V mode form)

Converts the single precision floating-point number in the floating accumulator to a double precision floating-point number by loading 0s into bits 25-48 of the floating accumulator. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ FDV address
Floating Point Divide
I X 1 1 1 1 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 1 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the contents of the FAC by the contents of the location specified by EA. (See Chapter 6.) Normalizes the result if necessary and stores it in the FAC. A divide by 0 or an overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

The location specified by EA must contain a normalized floating-point number. An unnormalized divisor can cause an error.

▶ FLD address
Floating Point Load
I X 0 0 1 0 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 0 1 0 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates a 32-bit effective address, EA. Loads the 32-bit contents in the location specified by EA into the FAC. (See Chapter 6.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ FLOT
Convert Integer to Floating Point
1 1 0 0 0 0 0 1 0 1 1 0 1 0 0 0 (R mode form)

Converts the 31-bit integer contained in A and B to a normalized floating-point number and stores the result in the floating accumulator. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FLTA
Convert Integer to Float
1 1 0 0 0 0 0 1 0 1 0 1 1 0 1 0 (V mode form)

Converts the 16-bit integer in A to a floating-point number and stores the result in the floating accumulator. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FLTL
Convert Long Integer to Float
1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 1 (V mode form)

Converts the 32-bit integer in L to a floating-point number and stores the result in the floating accumulator. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FLX address
Floating Load Index
I 0 1 1 0 1 1 1 0 0 0 0 0 1 CB\2 (R, V mode form)

Calculates an effective address, EA. Loads the index register, X, with two times the 16-bit contents of the location specified by EA. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction cannot do indexing.

▶ FMP address
Floating Multiply
I X 1 1 1 0 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 0 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the contents of the FAC by the contents of the location specified by EA. (See Chapter 6.) Normalizes the result if necessary and stores it in the FAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRN
Floating Round
1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 0 (R, V mode form)

This instruction operates on and stores all results in the floating accumulator.

For the 9950, the following actions occur. If bits 1-48 contain 0, then bits 49-64 are cleared to 0. If bits 24 and 25 both contain 1, then 1 is added to bit 24, bits 25-48 are cleared to 0, and the result is normalized. If bit 25 contains 1 and bits 26-48 are not equal to 0, then 1 is added to bit 24, bits 25-48 are cleared, and the result is normalized.

For the rest of the 50 series, the following actions occur. If bits 1-48 contain 0, then bits 49-64 are cleared to 0. Otherwise, bit 25 is added to bit 24, bits 25-48 are cleared to 0, and the result is normalized.

For all systems, if no floating point exception occurs, sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating point exception fault. See Chapter 11 for more information.

▶ FRNM
FLoating Point Round towards Negative Infinity
0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 (V mode form)

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, the instruction ends. If bits 25-48 of DAC contain 0s, the instruction ends. In any other case, the instruction clears bits 25-48 to 0, normalizes the result, and places it in DAC.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRNP
Floating Point Round towards Positive Infinity
0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 (V mode form)

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, the instruction ends. If bits 25-48 of DAC contain 0s, the instruction ends. In any other case, the instruction adds 1 to the value contained in bit 24 of DAC, clears bits 25-48 to 0, normalizes the result, and places it in DAC.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRNZ
Floating Point Round towards Zero
0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 1 (V mode form)

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, the instruction ends. If bits 25-48 of DAC are not 0s and bit 1 contains 1, the instruction adds 1 to the value contained in bit 24 of DAC, clears bits 25-48 to 0, normalizes the result, and places it in DAC. If any other condition exists, no rounding occurs.

If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FSB address
Floating Subtract
I X 0 1 1 1 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 1 1 1 0 0 0 0 0 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 32-bit contents of the locations specified by EA from the contents of the FAC. (See Chapter 6.) Normalizes the result if necessary and stores it in the FAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FSGT
Floating Skip on F Greater Than 0
1 1 0 0 0 0 0 1 0 1 0 0 1 1 0 1 (R, V mode form)

Skips the next word if the contents of the floating accumulator are greater than 0. Leaves the value of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ FSLE
Floating Skip on F Less Than or Equal To 0
1 1 0 0 0 0 0 1 0 1 0 0 1 1 0 0 (R, V mode form)

Skips the next word if the contents of the floating accumulator are less than or equal to 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ FSMI
Floating Skip on F Minus
1 1 0 0 0 0 0 1 0 1 0 0 1 0 1 0 (R, V mode form)

Skips the next word if the contents of the floating accumulator are less than 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ FSNZ
Floating Skip on F Not 0
1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 (R, V mode form)

Skips the next word if the contents of the floating accumulator are less than or equal to 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ FSPL
Floating Skip on FAC Plus
1 1 0 0 0 0 0 1 0 1 0 0 1 0 1 1 (R, V mode form)

Skips the next word if the contents of the floating accumulator are greater than or equal to 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ FST address
Floating Store
I X 0 1 0 0 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 0 0 1 1 0 0 0 0 0 1 BR\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the FAC into the 32-bit location specified by EA. (See Chapter 6.) If the exponent contained in the FAC is too large to be expressed in 8 bits, a floating-point exception (store exception) occurs. If no floating-point exception occurs, the instruction resets CBIT to 0. At the end of the instruction, the values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information. In either case, a floating-point exception leaves the contents of the memory location in an indeterminate state.

Note

This instruction does not normalize the result before loading it into the specified memory location.

▶ FSZE
Floating Skip on F Equal to 0
1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 (R, V mode form)

Skips the next word if the contents of the floating accumulator equal 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ HLT
Halt
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (S, R, V mode form)

Halts computer operation. The program counter points to the instruction that would have been executed if execution had not been stopped. The supervisor terminal indicates a halt. Leaves the values of LINK, CBIT, and the condition codes unchanged.

This instruction saves the contents of registers in a memory location specified by the RSAVPTR. The contents of RSAVPTR can be accessed by an LDLR/STLR instruction with address '40037. The registers are saved in their physical order. (See Chapter 9 for the format of these register files.) The saved register file order is shown in Table 13-3.

Table 13-3
Order of Saved Registers after HLT

9950	850	Rest of 50 Series
User Reg Set 1	ISP #1:	User Reg Set 1
User Reg Set 2	User Reg Set 1	User Reg Set 2
User Reg Set 3	User Reg Set 2	DMx Reg File
User Reg Set 4	DMx Reg File	Microcode Reg File
Microcode Reg File, Set 2	Microcode Reg File	
Indirect Reg Set	ISP #2:	
Microcode Reg File, Set 1	User Reg Set 1	
	User Reg Set 2	
DMx Reg File	DMx Reg File	
	Microcode Reg File	

Note

This is a restricted instruction.

▶ IAB
Interchange A and B
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 (S, R, V mode form)

Interchanges the contents of A and B. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICA
Interchange Bytes of A Register
1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 (S, R, V mode form)

Interchanges the bytes of A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICL
Interchange Bytes and Clear Left
1 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 (S, R, V mode form)

Interchanges the bytes of A, then clears the left byte to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICR
Interchange Bytes and Clear Right
1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 (S, R, V mode form)

Interchanges the bytes of A, then clears the right byte to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ILE
Interchange E and L
1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 (S, R, V mode form)

Interchanges the values of E and L. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ IMA address
Interchange A and Memory
I X 1 0 1 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 1 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 0 1 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Interchanges the contents of A and the contents of the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ INA function,device
Input to A
1 0 1 1 0 0 FUNCTION\4 DEVICE\6
Valid for modes S, R.

Loads data from the specified device into A. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INBC address
Interrupt Notify Beginning, Clear Active Interrupt
0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1 (V mode form)
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Places the notified process at the beginning of the appropriate priority level queue. Issues a CAI pulse to clear the currently active interrupt.

Note that a process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK, and the condition codes are indeterminate.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INBN address
Interrupt Notify Beginning
0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 1 (V mode form)
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Places the notified process at the beginning of the appropriate priority level queue. Does not issue a CAI pulse to clear the currently active interrupt.

Note that a process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK, and the condition codes are indeterminate.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INEC address
Interrupt Notify End, Clear Active Interrupt
0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0 (V mode form)
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Issues a CAI pulse to clear the currently active interrupt. Places the notified process at the end of the appropriate priority level queue.

Note that a process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK and the condition codes are indeterminate.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INEN address
Interrupt Notify End
0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 (V mode form)
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Does not issue a CAI pulse to clear the currently active interrupt. Places the notified process at the end of the appropriate priority level queue.

Note that a process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK, and the condition codes are indeterminate.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INH
Inhibit Interrupts
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 (S, R, V mode form)

Inhibits interrupts by setting bit 1 of the modals to 0. Inhibits interrupts for one instruction. The processor ignores any interrupt requests that are made over the I/O bus. Note that this instruction takes effect immediately. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHL
Inhibit Interrupts (Local)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 (S, R, V mode form)

This 850 instruction performs the same actions as INH does. Leaves the values of LINK, CBIT,and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHM
Inhibit Interrupts (Mutual)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 (S, R, V mode form)

For the 850, a processor checks the availability of the mutual exclusion lock. If available, the processor sets this lock and inhibits interrupts. Otherwise, it waits for the lock to be released by the other processor and then sets the lock and inhibits interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHP
Inhibit Interrupts (Process)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 (S, R, V mode form)

For the 850, a processor checks the availability of the process exchange lock. If available, the processor sets it and inhibits interrupts. Otherwise, it waits for the lock to be released by the other processor and then sets the lock and inhibits interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INK
Input Keys
0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 (S, R mode form)

Loads the contents of the R mode keys into A. Reads the low-order 8 bits of the S register along with the high-order 8 bits of the keys register. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ INT
Convert Floating Point to Integer
1 1 0 0 0 0 0 1 0 1 1 0 1 1 0 0 (S,R mode form)

Converts the double precision floating-point number contained in the floating accumulator to a 31-bit integer and stores the result in A and bits 2-16 of B. Bit 1 of B (bit 17 of the result) is forced to 0. Ignores the fractional portion of the floating-point number. Overflow occurs if the value in the floating accumulator is less than -2**30 or

greater than (2**30)-1. If overflow occurs, a floating-point exception occurs. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ INTA
Convert Floating Point to Integer
1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 1 (V mode form)

Converts the double precision number contained in the floating accumulator to a 16-bit integer and stores the result in A. Ignores the fractional portion of the floating-point number. Overflow occurs if the value in the floating accumulator is less than -2**15 or greater than (2**15)-1. If overflow occurs, a floating-point exception occurs. At the end of this instruction, the B register contents are indeterminate. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ INTL
Convert Floating Point to Long Integer
1 1 0 0 0 0 0 1 0 1 0 1 1 0 1 1 (V mode form)

Converts the double precision floating-point number contained in the floating accumulator to a 32-bit integer and stores the result in L. Ignores the fractional portion of the floating-point number contained in the floating accumulator. Overflow occurs if the floating-point number is less than -2**31 or greater than (2**31)-1. If overflow occurs, a floating-point exception occurs. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ IRS address
Increment and Replace Memory
I X 1 0 1 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 1 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 0 1 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Fetches the contents of the location specified by EA, adds 1, and stores the result back in the location specified by EA. Skips the next location if the incremented value is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ IRTC
Interrupt Return, Clear Active Interrupt
0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 (V mode form)

Returns from an interrupt. Issues a CAI pulse to clear the currently active interrupt. Restores the state existing before the interrupt by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from the values saved in microcode temporary registers PSWPB and PSWKEYS.

Note

This is a restricted instruction.

▶ IRTN
Interrupt Return
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 (V mode form)

Returns from an interrupt. Does not issue a CAI pulse to clear the currently active interrupt. Restores the state existing before the interrupt by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from the values saved in microcode temporary registers PSWPB and PSWKEYS.

Note

This is a restricted instruction.

▶ IRX
Increment and Replace X
1 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 (S, R, V mode form)

Increments the contents of X by 1 and stores the result in X. Skips the next word if the incremented value is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ITLB
Invalidate STLB Entry
0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1 (V mode form)

Invalidates the STLB entry that corresponds to the virtual address contained in L. The values of CBIT, LINK, and the condition codes are indeterminate. You must execute this instruction whenever you change the page table entry for the given address.

If you change an SDW or DTAR, (explained in Chapter 4), you usually have to invalidate the entire STLB by issuing the instruction PTLB. A 0 in the segment number portion of L invalidates the IOTLB entry corresponding to the address specified by L.

Note

This is a restricted instruction.

▶ JDX address
Jump and Decrement X
I 0 1 1 0 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts 1 from the contents of the index register, X. If the decremented value does not equal 0, the instruction loads EA into the program counter. If the decremented value is equal to 0, execution continues with the next sequential instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction cannot do indexing. See Chapter 2 for more information.

▶ JEQ address
Jump on A Equal to 0
I X 0 0 1 0 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads EA into the program counter if the contents of A are equal to 0. If the contents of A are not equal to 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ JGE address
Jump on A Greater Than or Equal to 0
I X 0 1 1 1 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. If the contents of A are greater than or equal to 0, the instruction loads EA into the program counter. If the contents of A are less than 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

▶ JGT address
Jump on A Greater Than 0
I X 0 1 0 1 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. If the contents of A are greater than 0, the instruction loads EA into the program counter. If the contents of A are less than or equal to 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

▶ JIX address
Increment X and Jump if 0
I 0 1 1 0 1 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds 1 to the contents of X, the index register. If the incremented value does not equal 0, the instruction loads EA into the program counter. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

Note

This instruction cannot do indexing.

▶ JLE address
Jump on A Less Than or Equal to 0
I X 0 1 0 0 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. If the contents of A are less than or equal to 0, the instruction loads EA into the program counter. If the contents of A are greater than 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

▶ JLT address
Jump on A Less Than 0
I X 0 1 1 0 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. If the contents of A are less than 0, the instruction loads EA into the program counter. If the contents of A are greater than 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

▶ JMP address
Jump
I X 0 0 0 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 0 0 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 0 0 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Loads EA into the program counter. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ JNE address
Jump on A Not Equal to 0
I X 0 0 1 1 1 1 0 0 0 0 1 1 CB\2 (R mode form)
[DISPLACEMENT\16]

Calculates an effective address, EA. If the contents of A do not equal 0, the instruction loads EA into the program counter. If the contents of A are equal to 0, execution continues with the next instruction. Leaves the contents of LINK, CBIT, and the condition codes unchanged.

▶ JST address
Jump and Store
I X 1 0 0 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 0 0 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 0 0 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Stores the contents of the program counter in the location specified by EA. Execution continues at the location EA+1.

The JST instruction truncates the return address according to the addressing mode before storing it. The high-order bits of the memory location are not affected by the store. This allows you to preset the I or X bits in some modes as follows:

Mode	Allowed Presets
16S	I, X
32S, 32R	I
64R, 64V	none

Note

This instruction cannot be used in shared code. Inhibits interrupts for one instruction in Ring Only.

In Ring 0, this instruction inhibits interrupts during execution of the next instruction.

▶ JSX address
Jump and Save in X
I 1 1 1 0 1 1 1 0 0 0 0 1 1 BR\2 (V mode long)
DISPLACEMENT\16

I 1 1 1 0 1 1 1 0 0 0 0 1 1 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Increments the contents of the program counter by 1 and loads the result into X. Loads EA into the program counter. For the 750 and 850, if the value of CB is 2 or 3, then the next 16 bits are skipped. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction cannot do indexing.

▶ JSXB address
Jump and Set XB
I X 1 1 0 0 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 0 0 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the contents of the program counter into XB. Loads EA into the program counter. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction can make subroutine calls outside the current segment as well as within.

▶ JSY address
Jump and Save in Y
I X 1 1 0 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 0 0 DISPLACEMENT\10 (V mode short)

Calculates an effective address, EA. Loads Y with the location number of the program counter. Loads EA into the program counter. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction may call only those subroutines residing in the same procedure segment only, since only the word number field of the program counter is saved.

▶ LCEQ
Load A on EQ
1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 (V mode form)

If the condition codes reflect an equal to condition, the instruction loads A with a 1. If the condition codes reflect a not equal condition, the instruction loads A with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCGE
Load A on GE
1 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 (V mode form)

If the condition codes reflect a greater than or equal to condition, the instruction loads A with a 1. If the condition codes reflect a less than condtion, the instruction loads A with a 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ LCGT
Load A on GT
1 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 (V mode form)

If the condition codes reflect a greater than condition, the instruction loads with a 1. If the condition codes reflect a less than or equal to condition, the instruction loads A with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCLE
Load A on LE
1 1 0 0 0 0 1 1 0 1 0 0 0 0 0 1 (V mode form)

If the condition codes reflect a less than or equal to condition, the instruction loads A with a 1. If the condition codes reflect a greater than condition, the instruction loads A with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCLT
Load A on LT
1 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 (V mode form)

If the condition codes reflect a less than condition, the instruction loads A with a 1. If the condition codes reflect a greater than or equal to condition, the instruction loads A with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCNE
Load A on NE
1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 (V mode form)

If the condition codes reflect a not equal condition, the instruction loads A with a 1. If the condition codes reflect an equal condition, the instruction loads A with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LDA address
Load A
I X 0 0 1 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 0 1 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 0 1 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Loads the contents of the location specified by EA into A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LDC flr
Load Character
0 0 0 0 0 0 1 0 1 1 0 0 FLR 0 1 0 (V mode form)

If the contents of the specified FLR are nonzero, the instruction loads the single character pointed to by the appropriate FAR into bits 9-16 of A and loads 0s into bits 1-8. Updates the contents of the appropriate FAR by 8 so that they point to the next character. Decrements the contents of the specified FLR by 1. Sets the condition codes to NE.

If the contents of the specified FLR are 0, the instruction sets the condition codes to EQ.

The instruction leaves the values of CBIT and LINK unchanged.

Note

This instruction uses FAR0 when FLR0 is specified, and FAR1 when FLR1 is specified.

▶ LDL address
Long Load
I X 0 0 1 0 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates a long effective address, EA. Loads the 32-bit contents of the location specified by EA into L. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LDLR address
Load L from Addressed Register
I X 0 1 0 1 1 1 0 0 0 Y 0 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates a doubleword effective address, EA. Loads L with the contents of the register file location specified by the word portion of EA. Bit 2 and bit 12 of the word portion of EA determine the actions of this instruction:

Bit 2	Bit 12	Action
1*	——	Ignore bits 1 and 3-9. The word portion of EA specifies an absolute register number from 0-'377.
0*	0	Bits 13-16 of the word portion of EA specify one of the registers '20-'37 in the current register set.
0	0	Bits 13-16 of the word portion of EA specify one of the registers 0-'17 in the current register set.

*This is a restricted instruction.

Leaves the values of CBIT and LINK unchanged; the values of the condition codes are indeterminate. See Chapter 9 for more information on register sets.

▶ LDX address
Load X
I 1 1 1 0 1 DISPLACEMENT\10 (S, R, V mode short form)

I 1 1 1 0 1 1 1 0 0 0 0 1 1 CB\2 (R mode long)
[DISPLACEMENT\16]

I 1 1 1 0 1 1 1 0 0 0 0 1 1 BR\2 (V mode long)
DISPLACEMENT\16

Calculates an effective address, EA. Loads X, the index register, with the contents of the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged. For 750 and 850 processors in R mode only, if CB contains 2 or 3, the first 16 bits of the next instruction will be skipped.

Note

This instruction cannot specify indexing, though an address calculated in the indirect chain may do so in 16S mode. See Chapter 2 for more information.

▶ LDY address
Load Y
I 1 1 1 0 1 1 1 0 0 0 Y 0 1 BR\2 (V mode form)
DISPALCEMENT\16

Calculates an effective address, EA. Loads Y with the contents of the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction cannot do indexing. See Chapter 2 for more information.

▶ LEQ
Load A on A Equal to 0
1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 1 (S, R, V mode form)

If the contents of A are equal to 0, the instruction loads A with a 1. If the contents of A are not equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LF
Load False
1 1 0 0 0 0 0 1 0 0 0 0 1 1 1 0 (S, R, V mode form)

Loads A with a 0. Leaves the values of LINK and CBIT unchanged. Sets the condition codes to reflect the result of the comparison. (See Table 5-3.)

▶ LFEQ
Load A on F Equal to 0
1 1 0 0 0 0 1 0 0 1 0 0 1 0 1 1 (V mode form)

If the contents of the floating accumulator are equal to 0, the instruction loads A with a 1. If the F contents are not equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFGE
Load A on Floating Accumulator Greater Than or Equal to 0
1 1 0 0 0 0 1 0 0 1 0 0 1 1 0 0 (V mode form)

If the contents of the floating accumulator are greater than or equal to 0, the instruction loads A with a 1. If the F contents are less than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFGT
Load A on Floating Accumulator Greater Than 0
1 1 0 0 0 0 1 0 0 1 0 0 1 1 0 1 (V mode form)

If the contents of the floating accumulator are greater than 0, the instruction loads A with a 1. If the F contents are less than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFLE
Load A on Floating Accumulator Less Than or Equal to 0
1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 1 (V mode form)

If the contents of the floating accumulator are less than or equal to 0, the instruction loads A with a 1. If the F contents are greater than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFLI flr,data
Load FLR Immediate
0 0 0 0 0 0 1 0 1 1 0 0 FLR 0 1 1 (V mode form)
INTEGER\16

Loads the 16-bit, unsigned integer contained in the second word of the instruction into the specified FLR. Clears the upper bits of the FLR. Leaves the values of CBIT, LINK, the condition codes, and the associated FAR unchanged.

▶ LFLT
Load A on Floating Accumulator Less Than 0
1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 (V mode form)

If the contents of the floating accumulator are less than 0, the instruction loads A with a 1. If the F contents are greater than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFNE
Load A on Floating Accumulator Not Equal to 0
1 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0 (V mode form)

If the contents of the floating accumulator are not equal to 0, the instruction loads A with a 1. If the F contents are equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LGE
Load A on Greater Than or Equal to 0
1 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0 (S, R, V mode form)

If the contents of A are greater than or equal to 0, the instruction loads A with a 1. If the A contents are less than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LGT
Load A on Greater Than 0
1 1 0 0 0 0 0 1 0 0 0 0 1 1 0 1 (S, R, V mode form)

If the contents of A are greater than 0, the instruction loads A with a 1. If the A contents are less than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LIOT address
Load I/O TLB
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 (V mode form)
AP\32

Loads a specified IOTLB entry. The following list shows the contents of the LIOT entry and the origin of the information.

Origin	Description
AP in LIOT	Virtual address in segment 0 (calculated from the EA).
Page table	Physical address (translation of the virtual address) obtained from segment 0. Note that if the fault bit is set to 1, a page fault occurs.
L register	Target virtual address. This is the segment number and page number of the virtual address that will be used by procedures accessing this information. This is used to help invalidate the proper locations in the cache. This is provided in L as a virtual address. The low-order 10 bits (word number in the page) and the segment number are ignored.

The values of CBIT, LINK, and the condition codes are indeterminate.

Note

This is a restricted instruction.

▶ LLE
Load on A Less Than or Equal to 0
1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 (S, R, V mode form)

If the contents of A are less than or equal to 0, the instruction loads A with a 1. If the A contents are greater than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLEQ
Load A on L Equal to 0
1 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 (V mode form)

If the contents of L are equal to 0, the instruction loads A with a 1. If the L contents are not equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLGE
Load A on L Greater Than or Equal to 0
1 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0 (V mode form)

If the contents of L are greater than or equal to 0, the instruction loads A with a 1. If the L contents are less than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLGT
Load A on L Greater Than 0
1 1 0 0 0 0 1 1 0 1 0 0 1 1 0 1 (V mode form)

If the contents of L are greater than 0, the instruction loads A with a 1. If the L contents are less than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLL n
Long Left Logical
0 1 0 0 0 0 1 0 0 0 N\6 (S, R, V mode form)

Shifts the contents of A and B to the left, bringing 0s into bit 16 of B. Shifts bits out of bit 1 of B into bit 16 of A. CBIT contains the value of last bit shifted out of A; the values of all other bits shifted out of A are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LLLE
Load A on L Less Than or Equal to 0
1 1 0 0 0 0 1 1 0 1 0 0 1 0 0 1 (V mode form)

If the contents of L are less than or equal to 0, the instruction loads A with a 1. If the L contents are greater than 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLLT
Load A on L Less Than 0
1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 (V mode form)

If the contents of L are less than 0, the instruction loads A with a If the L contents are greater than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLNE
Load A on L Not Equal to 0
1 1 0 0 0 0 1 1 0 1 0 0 1 0 1 0 (V mode form)

If the contents of L are not equal to 0, the instruction loads A with a 1. If the L contents are equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LLR n
Long Left Rotate
0 1 0 0 0 0 1 0 1 0 N\6 (S, R, V mode form)

Shifts the contents of A and B left, rotating bit 1 of A into bit 16 of B. Bit 1 of B shifts into bit 16 of A. CBIT contains a copy of the last bit rotated into bit 16 of B. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LLS n
Long Left Shift
0 1 0 0 0 0 1 0 0 1 N\6 (V mode form)

Shifts the 32-bit integer in L left arithmetically, bringing 0s into bit 32. Bits shifted out of bit 1 are lost. If bit 1 changes state, it is interpreted as an overflow and causes an integer exception. If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ LLS n
Long Left Shift
0 1 0 0 0 0 1 0 0 1 N\6 (S, R mode form)

Shifts the 31-bit integer contained in A and B left arithmetically, bringing 0s into bit 16 of B. Bit 1 of B does not take part in the shift; bit 2 of B is shifted into bit 16 of A. Bits shifted out of bit 1 of A are lost. If bit 1 of A changes state, it is interpreted as an overflow and causes an integer exception. If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ LLT
Load on A Less Than 0
1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 (S, R, V mode form)

If the contents of A are less than 0, the instruction loads A with a 1. If the A contents are greater than or equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LMCM
Leave Machine Check Mode
0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 (S, R, V mode form)

Leaves machine check mode by setting bits 15-16 of the modals to 00. If a machine parity error occurs in this mode, the hardware sets the machine check flag but no check (V mode) or interrupt (S, R modes) occurs. Inhibits the machine for one instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ LNE
Load on A Not Equal to 0
1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 (S, R, V mode form)

If the contents of A are not equal to 0, the instruction loads A with a 1. If the A contents are equal to 0, the instruction loads A with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes contain the result of the comparison. (See Table 5-3.)

▶ LPID
Load Process ID
0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 (V mode form)

Loads the process ID from bits 1-12 of A into RPID (the process ID register). This contains the 10 most significant bits of the user's address space. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ LPSW address
Load PSW
0 0 0 0 0 0 0 1 1 1 0 0 1 0 0 1 (V mode form)
AP\32

Changes the status of the processor by loading new values into the program counter, keys, and modals. Inhibits interrupts for one instruction.

Addresses a 4-word block at the specified location. The block has the following:

Word in Block	Contents
1-2	New program counter (ring, segment, word numbers)
3	New keys
4	New modals

Loads the program counter and keys of the currently running process with the contents of the first three words, then loads the processor modals with the contents of the fourth word.

The new value of bit 15 in the keys, the in-dispatch bit, can temporarily halt execution of the current process. This bit is altered by software only during a cold or warm start. If bit 15 is 0, the currently executing process will continue to execute, but at a location defined by the new value of the program counter. If bit 15 is 1, the processor enters the dispatcher and dispatches the ready process with the highest priority. When execution resumes for the process that was temporarily halted, note that execution resumes at the point defined by the value of the new program counter.

Regardless of the value of bit 15, the new value of the modals takes effect immediately, since the modals are associated with the processor, not the process.

Note that this instruction loads the four words of the register set that the STLR instruction cannot correctly load. STLR does not update the separate hardware registers the processor uses to maintain duplicate information for optimization.

Never use this instruction to change bits 9-11 of the modals. These bits specify the current user register set. This means that if you do not know the current value of these bits, you must do the following each time you want to execute an LPSW:

1. Inhibit interrupts.

2. Read the current values of modal bits 9-11 (use LDLR).

3. Mask the old values of the modal bits into the new information.

4. Load the new information into the modals with an LPSW.

For the two common uses of LPSW, you do not have to perform this sequence, since the values of modal bits 9-11 are predictable. When you use LPSW after a Master Clear to turn on processor exchange mode, bits 9-11 are 010 because the processor is always initialized to register set 2. When you use LPSW to return from a fault, check, or interrupt, simply reload the values stored by the break because these values are still correct.

Also note that you should not use LPSW to set bits 16 (the save done bit) or 15 (the in-dispatcher bit) of the keys, unless you are merely loading status following a fault, check, or interrupt. When issuing LPSW after a Master Clear, make sure you load 0s into both of these bits.

Note

This is a restricted instruction. This instruction inhibits interrupts during execution of the next instruction.

▶ LRL n
Long Right Logical
0 1 0 0 0 0 0 0 0 0 N\6 (S, R, V mode form)

Shifts the contents of A and B right, bringing 0s into bit 1 of A. Shifts bit 16 of A into bit 1 of B. CBIT contains the value of the last bit shifted out of B; the values of all other bits shifted out of B are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LRR n
Long Right Rotate
0 1 0 0 0 0 0 0 1 0 N\6 (S, R, V mode form)

Shifts the contents of A and B right, rotating bit 16 of B into bit 1 of A. Shifts bit 16 of A into bit 1 of B. CBIT contains a copy of the last bit rotated from B to A. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LRS n
Long Right Shift
0 1 0 0 0 0 0 0 0 1 N\6 (V mode form)

Shifts the 32-bit integer contained in L right arithmetically. Shifts copies of bit 1, the sign bit, into each of the vacated bits. CBIT contains the value of the last bit shifted out of L; the values of all other bits shifted out are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LRS n
Long Right Shift
0 1 0 0 0 0 0 0 0 1 N\6 (S, R mode form)

Shifts right arithmetically the 31-bit integer contained in A and B, leaving bit 1 of A unaffected. Bit 1 of B does not take part in the shift; bit 16 of A is shifted into bit 2 of B. Shifts copies of bit 1 of A into each of the vacated bits. CBIT contains the value of the last bit shifted out of B; the values of all other bits shifted out of B are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

N contains the two's complement of the number of shifts to perform. If N contains 0, the instruction performs 64 shifts.

▶ LT
Load True
1 1 0 0 0 0 0 1 0 0 0 0 1 1 1 1 (S, R, V mode form)

Loads A with a 1. Leaves the values of LINK and CBIT unchanged. Sets the condition codes to reflect the outcome of the operation. (See Table 5-3.)

▶ MDRS
Memory Diagnostic Read Syndrome Bits
0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 0 (V mode form)

Reads memory syndrome bits. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ MDWC
Memory Diagnostic Load Write Control Register
0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1 (V mode form)

Writes memory control register. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ MIA
Microcode Execute A
I X 1 0 1 0 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

This instruction currently causes a UII fault. If implemented, this instruction is for user-written microcode. For more information about UII, refer to Chapter 11.

▶ MIB
Microcode Execute B
I X 1 0 1 1 1 1 1 0 0 0 Y 0 1 BR\2 (V mode long)
DISPLACEMENT\16

This instruction currently causes a UII fault. If implemented, this instruction is for user-written microcode. For more information about UII, refer to Chapter 11.

▶ MPL address
Multiply Long
I X 1 1 1 0 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Multiplies the 32-bit integer in L by the 32-bit integer in the location specified by EA. Stores the 63-bit result in L and E. Resets CBIT to 0. Leaves the value of LINK unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

Note

This instruction cannot cause overflow.

▶ MPY address
Multiply
I X 1 1 1 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 1 1 1 0 DISPLACEMENT\10 (V mode short)

Calculates an effective address, EA. Multiplies the 16-bit integer in A by the 16-bit integer in the location specified by EA. Stores the 32-bit result in A and B. Resets CBIT to 0. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

Note

This instruction cannot cause overflow.

▶ MPY address
Multiply
I X 1 1 1 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 1 1 1 0 DISPLACEMENT\10 (S mode; R mode short)

Calculates an effective address, EA. Multiplies the 16-bit integer in A by the 16-bit integer in the location specified by EA. Loads the 31-bit result in A and B. If the multiplier and multiplicand are both -(2**15), an integer exception occurs. If no integer exception occurs, CBIT is reset to 0. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ NFYB
Notify
0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 1 (V mode form)
AP\32

Notifies the semaphore at address specified by the address pointer in the instruction. Uses LIFO queueing. Does not clear the currently active interrupt. The values of LINK, CBIT, and the condition codes are indeterminate.

Note

This is a restricted instruction.

▶ NFYE
Notify
0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 (V mode form)
AP\32

Notifies the semaphore at address specified by the address pointer in the instruction. Uses FIFO queueing. Does not clear the currently active interrupt. The values of LINK, CBIT, and the condition codes are indeterminate.

Note

This is a restricted instruction.

▶ NOP
No Operation
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 (S, R, V mode form)

Does nothing. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ NRM
Normalize
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 (S, R mode form)

Shifts the 31-bit integer in A and B to the left arithmetically, shifting in 0s into bit 16 of B. The shift does not affect bit 1 of B or bit 1 of A. The instruction shifts bits out of bit 2 in A until the value of bit 2 is opposite the value of bit 1 in A. Loads bits 9-16 of the S and R mode keys with the number of shifts performed. Leaves the values of CBIT and the condition codes unchanged; the value of LINK is indeterminate.

Note

Since the bits shifted out of bit 2 in A contain copies of the sign of the 31-bit number, the shift results in no loss of information.

▶ OCP function,device
Output Control Pulse
1 0 1 1 0 0 FUNCTION\4 DEVICE\6 (S, R mode form)

Sends a control pulse to perform the specified function to the specified device. This instruction never skips. Leaves the values of CBIT, LINK, and the condition codes unchanged. See Chapter 12 for more information.

Note

This is a restricted instruction.

▶ ORA
Inclusive OR
I X 0 0 1 1 1 1 0 0 0 Y 1 0 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Logically ORs the contents of the location specified by EA and the contents of A and stores the result in A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ OTA function,device
Output from A
1 1 1 1 0 0 FUNCTION\4 DEVICE\6 (S, R mode form)

Transfers data from A to the specified device. Leaves the values of CBIT, LINK, and the condition codes unchanged. See Chapter 12 for more information.

Note

This is a restricted instruction.

▶ OTK
Output Keys
0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 (S, R mode form)

Stores the contents of A in the keys. Loads CBIT, LINK, and the condition codes as a result of the operation. If this instruction is executed in Ring 0, it inhibits interrupt during execution of the next instruction. Loads the low-order 8 bits of the S register with the low-order 8 bits of A.

▶ PCL
Procedure Call
I X 1 0 0 0 1 1 0 0 0 0 Y 1 0 BR\2 (V mode form)
DISPLACEMENT\16

Sets CBIT, LINK, and the condtion codes to the values contained in the ECB. See Chapter 8 for a complete description of this instruction.

Note

When arguments are to be transferred to the called procedure, this instruction uses X, Y, and XB, destroying the previous contents of these registers. The contents of X, Y, and XB remain unchanged if no arguments are transferred. The contents of the condition codes, CBIT, and LINK are not correctly saved in the ECB along with the rest of the caller's keys.

▶ PID
Position for Integer Divide
0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 (S, R mode form)

Moves the contents of bits 2-16 of A into bits 2-16 of B. Clears bit 1 or register B to 0 and extends the sign contained in bit 1 of A into bits 2-16 of A. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ PIDA
Position for Integer Divide
0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 (V mode form)

Moves the contents of bits 1-16 of A into bits 17-32 of L. Extends the sign contained in bit 1 of A into bits 2-16 of A. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ PIDL
Position for Integer Divide Long
0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 (V mode form)

Moves the contents of L into E and extends the sign contained in bit 1 of L into bits 2-32 of L. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ PIM
Position After Multiply
0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 (S, R mode form)

Moves bits 2-16 of B into bits 2-16 of A. This converts a 31-bit integer to a 16-bit integer. Leaves the values of LINK, CBIT, and the condition codes unchanged. Note that loss of precision does not cause an integer exception.

▶ PIMA
Position after Multiply
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 (V mode form)

Moves bits 17-32 of L into bits 1-16 of A. This converts a 32-bit integer to a 16-bit integer. An integer exception occurs if there is a loss in precision. (This occurs if Bits 1-16 of A contain a value other than all 0s or all ones before the move.) If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

To position bits 17-32 of L in A, PIMA swaps the two words of L. Since A and B overlap L, this swap means that B contains whatever was in bits 1-16 of L. This value is not always 0, even when no integer exception occurs. (B may contain 111111, for example.)

▶ PIML
Position Following Integer Multiply Long
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 (V mode form)

Moves the contents of bits 1-32 of E into bits 1-32 of L. This converts a 64-bit integer to a 32-bit integer. Any loss of precision causes an integer exception. If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ PRTN
Procedure Return
0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 1 (V mode form)

Deallocates the stack frame created for the executing procedure and returns to the environment of the procedure that called it.

To deallocate the frame, the instruction stores the current value of the stack base register into the free pointer. It then restores the caller's state by loading the caller's program counter, stack base register, linkage base register, and keys with the values contained in the frame being deallocated. Sets bits 15-16 of the keys to 0.

Loads the ring number in the program counter with the current ring number to allow outward returns but prevent inward returns.

▶ PTLB
Purge TLB
0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 (V mode form)

L contains the address of a physical page, right justified. Based on the value of L bit 1, PTLB purges either the first 128 locations or a single location. If L bit 1 contains a 1, the instruction performs a complete purge. If L bit 1 contains a 0, the instruction purges the page specified by L. Leaves the values of CBIT, LINK, and the condition codes indeterminate. See Chapters 1, 3, and 12 for more information about the STLB and IOTLB.

Note

This is a restricted instruction.

On the 750, 850, or 9950, insert a CRE (Clear E) instruction before PTLB. Since PTLB uses E as a pointer, the CRE 0s E before PTLB manipulates it. If an interrupt occurs during PTLB's execution, E points to the location PTLB is currently purging. PTLB leaves the contents of E in an undefined state at the end of its execution.

▶ QFAD address
Quad Precision Floating Add
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

Calculates an effective address, EA. Adds the 112-bit, quad precision number contained in the locations specified by EA to the contents of QAC. (See Chapter 6.) Normalizes the result and loads it into QAC. An overflow or underflow causes a floating-point exception. If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFCM
Quad Precision Floating Complement
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 0 (V mode form)

Forms the two's complement of the value contained in QAC and normalizes it if necessary. (See Chapter 6.) Stores the result in QAC. An underflow or overflow causes a floating-point exception. If no floating-point exception occurs, resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFCS address
Quad Precision Floating Point Compare and Skip
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0

Calculates an effective address, EA. Compares the contents of QAC (see Chapter 6) to the 128-bit contents of the location specified by EA and skips according to the list below.

Condition	Skip
QAC > EA contents.	No skip.
QAC = EA contents.	Skip one word.
QAC < EA contents.	Skip two words.

The values of CBIT, LINK, and the condition codes are indeterminate.

Note

Be sure to use normalized numbers for correct results.

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFDV address
Quad Precision Floating Point Divide
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1

Calculates an effective address, EA. Divides the contents of QAC by the 112-bit contents of the location specified by EA. Normalizes the result and stores the whole quotient into QAC. An overflow, underflow, or divide by 0 causes a floating-point exception. If there is no floating-point exception, resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If the QFDV instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFLD address
Quad Precision Floating Point Load
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Calculates an extended, augmented effective address, EA. Performs one of the following actions with the value contained in the location specified by EA. Loads bits 1-112 into QAC and zeros QAC bits 113-128, or loads 128 bits into QAC. (See Chapter 6 for more information.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFLX address
Quad Precision Floating Point Load Index
I 0 1 1 0 1 1 1 0 0 0 Y 1 1 BR\2 (V mode long)
DISPLACEMENT\16

Calculates an effective address, EA. Shifts the 16-bit contents of the location specified by EA to the left three times to multiply the contents by eight. Shifts in 0s on the right and shifts data out on the left first through bit 2 and then bit 1. Leaves the values of CBIT, LINK, and and the condition codes unchanged.

Note

This instruction cannot do indexing.

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFMP address
Quad Precision Floating Point Multiply
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

Calculates an effective address, EA. Multiplies the contents of QAC by the 112-bit contents of the location specified by EA. (See Chapter 6.) Normalizes the result if necessary and stores it into QAC. An overflow or underflow causes a floating-point exception. If there is no floating-point exception, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFSB address
Quad Precision Floating Point Subtract
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

Calculates an effective address, EA. Subtracts the contents of the locations specified by EA from the 112-bit contents of QAC. (See Chapter 6.) Normalizes the result if necessary and loads it into QAC. An overflow or underflow causes a floating-point exception. If there is no floating-point exception, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFST address
Quad Precision Floating Point Store
I X 0 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Calculates an effective address, EA. Stores the 128-bit contents of QAC into the 128 bits of memory specified by EA. (See Chapter 6.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction does not normalize the result before storing it into the specified memory location.

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QINQ
Quad to Integer, in Quad Convert
1 1 0 0 0 0 0 1 0 1 1 1 1 0 1 0 (V mode form)

Strips the fractional portion of QAC as described in Table 13-4:

Table 13-4
QINQ Actions

Exponent Value	Action
'340 <= Exp	A conversion fault occurs.
'200 < Exp < '340	If sign >= 0, strip fractional part of QAC for result. If sign < 0 and fractional part = 0, strip fractional part of QAC and increment result by 1. If sign < 0 and fractional part <> 0, strip fractional part for result.
'200 = Exp	If sign >= 0, result = 0. If sign < 0 and bits 2-96 = 0 result = -1. If sign < 0 and bits 2-96 <> 0 result = 0.
'200 > Exp	Result = 0.

The QINQ instruction can cause a floating-point exception; an exception does not alter the contents of QAC. If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QIQR
Quad to Integer, in Quad Convert Rounded
1 1 0 0 0 0 0 1 0 1 1 1 1 0 1 1 (V mode form)

Strips the fractional portion of QAC as described in Table 13-5.

Table 13-5
QIQR Actions

Exponent Value	Action
'340 < Exp	A conversion fault occurs.
Exp = '340	Set the most significant bit of the fractional part of QAC to 0.
'200 < Exp < '340	If sign >= 0, strip fractional part of QAC for result. If sign < 0 and fractional part <>0, strip fractional part of QAC for result. If sign <> 0 and fractional part = 0, strip the fractional part and increment the integer part by 1. In any case, increment the integer part by 1 if it exists and the most significant bit of the fractional part of QAC is 1.
Exp = '200	If sign >= 0, result = 0. If sign < 0 and bits 2-96 = 0 result = -1. If sign < 0 and bits 2-96 <> 0 result = 0. For all cases increment integer part by 1 if it exists and the most significant bit of QAC = 1.
Exp < '200	The result is 0.

The QIQR instruction can cause a floating-point exception; an exception does not alter the contents of QAC. If no floating-point exception occurs, the instruction resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ RBQ address
Remove Entry from Bottom of Queue
1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 1 (V mode form)
AP\32

The address pointer in this instruction points to the QCB for a queue. The instruction removes the entry from the bottom of the referenced queue and loads it into A. If the queue is not empty, sets the condition codes to NE; if empty, resets A to 0 and sets the condition codes to EQ. Leaves the values of CBIT and LINK unchanged.

▶ RCB
Reset CBIT Bit to 0
1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 (S, R, V mode form)

Resets CBIT to 0. Leaves the values of the condition codes and LINK unchanged.

▶ RMC
Reset Machine Check Flag to 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 (S, R, V mode form)

Resets the MCM flag (bits 15-16 of the modals) to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged. Inhibits interrupts during execution of the next instruction.

Note

This is a restricted instruction.

▶ RRST address
Restore Registers
0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 1 (V mode form)
AP\32

Calculates an effective address, EA, from the 32-bit address pointer in the instruction. This specifies the starting address of a save area for the general, floating, and XB registers. The save area format is shown in Table 13-6. Restores the contents of the general, floating, and XB registers from this save area. Bits 1-16 of the save area are a save mask, whose format appears in Figure 13-4. A mask bit value of 1 means that the corresponding register had nonzero contents that have been saved in the save area; a mask bit value of 0 means that the corresponding register's contents were 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Table 13-6
RRST Save Area Format

Word #	Contents
1	Save mask
2-5	FR1 (F)
6-9	FR0
10-11	X, GR7
12-13	GR6
14-15	Y, S, GR5
15-17	GR4
18-19	E, GR3
20-21	A, B, L, GR2
22-23	GR1
24-25	GR0
26-27	XB

1 4	5 6 7	8 9	10	11	12 13	14	15 16
0000	FR1	FR0	X	-	Y	-	E	L,B,A	—

Save Mask Format, RRST and RSAV Instructions
Figure 13-4

▶ RSAV
Save Registers
0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 1 (V mode form)
AP\32

Calculates an effective address, EA, from the 32-bit address pointer in the instruction. This specifies the starting address of a save area for the general, floating, and XB registers. The save area format is shown in Table 13-7. Bits 1-16 of the save area are a save mask, whose format appears in Figure 13-5. This instruction sets the mask bit of each register as follows: to 1 if the register's contents have a nonzero value; to 0 if a 0 value. Saves the nonzero contents of the general, floating, and XB registers in the save area. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Table 13-7
RSAV Save Area Format

Word #	Contents
1	Save mask
2-5	FR1 (F)
6-9	FR0
10-11	X, GR7
12-13	GR6
14-15	Y, S, GR5
15-17	GR4
18-19	E, GR3
20-21	A, B, L, GR2
22-23	GR1
24-25	GR0
26-27	XB

1 4	5 6 7	8 9	10 11	12 13	14	15 16
0000	FR1	FR0	X -	Y -	E	L,B,A ——

Save Mask Format, RRST and RSAV Instructions
Figure 13-5

▶ RTN
Return
0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 (R mode form)

Returns control from a P300 recursive procedure to the calling routine. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction reverses the actions done by CREP and ENTR.

▶ RTQ address
Remove Entry from Top of Queue
1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 (V mode form)
AP\32

The address pointer in this instruction is to the QCB for a queue. The instruction removes the entry from the top of the referenced queue, and loads it into A. If the queue is empty, the instruction resets A to 0 and the condition codes to EQ; if not empty, sets the condition codes to NE. Leaves the values of CBIT and LINK unchanged.

▶ RTS
Reset Time Slice
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 (V mode form)
Valid for the 550-II, 750, 850, I450, and new processors.

A contains a negative value representing the number of milliseconds in the new time slice.

Adds the current value of the interval timer (locations 16-17 of the PCB) to the contents of the elapsed timer (locations 10-11 of the PCB), then subtracts the contents of A from the sum of the timers. Stores the result in the elapsed timer. Loads the contents of A into the interval timer. Leaves the contents of A unchanged. The values of CBIT, LINK, and the condition codes are unchanged.

The addition performed by this instruction is equivalent to the following series of instructions:

```
LDA   ITH   /* load A with the contents of ITH
SUB   RV    /* subtract reset value (in RV) from contents of A
PIDA        /* sign extend the contents of A into bits 17-32 of L
SRC         /* skip next word if CBIT is 0 (no overflow)
CMA         /* complement A
ADL   ET    /* add contents of L and contents of ET
STL   ET    /* store contents of L in ET
LDA   RV    /* load A with reset value
STA   ITH   /* store the reset value into ITH
```

Note

This instruction can be executed in Ring 0 only.

▶ S1A
Subtract 1 from A
1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 (S, R, V mode form)

Subtracts 1 from the contents of A and stores the result in A. If the number to be decremented is -(2**15), an integer exception occurs, and the instruction loads (2**15)-1 into A. If no overflow occurs, the instruction resets CBIT to 0. LINK reflects the state of CBIT. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ S2A
Subtract 2 from A
1 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 (S, R, V mode form)

Subtracts 2 from the contents of A and stores the result in A. If the number to be decremented is -(2**15-1) or -2**15, an integer exception occurs and the instruction loads (2**15)-1 into A. If no overflow occurs, the instruction resets CBIT to 0. LINK reflects the state of the CBIT. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ SAR n
Skip on A Register Bit Reset to 0
1 0 0 0 0 0 0 0 1 0 1 1 N\4 (S, R, V mode form)

Skips the next word if bit n in register A contains 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

N specifies the bit to test. A value of 0 indicates bit 1; 1, bit 2; and so on.

Note

The assembler converts n to the octal equivalent of bit number minus 1.

▶ SAS n
Skip on A Register Bit Set to 1
1 0 0 0 0 0 1 0 1 0 1 1 N\4 (S, R, V mode form)

Skips the next word if bit n in register A contains 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

N specifies the bit to test. A value of 0 indicates bit 0, and so on.

Note

The assembler converts n to the octal equivalent of bit number minus 1.

▶ SBL address
Subtract Long
I X 0 1 1 1 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Subtracts the 32-bit integer in the location specified by EA from the contents of L. Stores the results in L. If the result is greater than 2**31, an integer exception occurs and the instruction loads bit 1 of L with a 1 and bits 2-32 with (result - (2**31)).

If the result is less than -(2**31), an integer exception occurs and the instruction loads bit 1 of L with a 0 and bits 2-32 with the negative of (result + (2**31)).

If no overflow occurs, the instruction resets CBIT to 0. The instruction loads LINK with the borrow bit. The condition codes reflect the outcome of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains a 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ SCA
Load Shift Count into A
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 (S, R mode form)

Loads the contents of bits 9-16 of the keys into bits 9-16 of A. Clears bits 1-8 of A to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

The SCA instruction is used with NRM.

▶ SCB
Set CBIT Bit to 1
1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 (S, R, V mode form)

Sets the value of CBIT to 1. Leaves the values of the condition codes unchanged. The value of LINK is indeterminate.

▶ SGL
Enable Single Precision Mode
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 (S, R mode form)

Enters single precision mode by resetting bit 2 of the keys to 0. Subsequent LDA, STA, ADD, and SUB instructions manipulate 16-bit integers. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SGT
Skip on A Greater than 0
1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A are greater than 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SKP n
Skip
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the specified condition is met. Leaves the values of LINK, CBIT, and the condition codes unchanged.

This instruction allows you to test for several conditions. The table below shows the conditions available to test and information about the associated instruction.

Table 13-8
SKP Conditions

Mnem	Opcode	Condition
NOP	101000	No operation.
SKP	100000	Unconditional skip.
SMI	101400	Skip on bit 1 of A equal to 1.
SPL	100400	Skip on bit 1 of A equal to 0.
SLN	101100	Skip on bit 16 of A equal to 1.
SLZ	100100	Skip on bit 16 of A equal to 0.
SNE	101040	Skip on A not equal to 0.
SZE	100040	Skip on A equal to 0.
SS1*	101020	Skip on sense switch 1 set to 1.
SR1*	100020	Skip on sense switch 1 reset to 0.
SS2*	101010	Skip on sense switch 2 set to 1.
SR2*	100010	Skip on sense switch 2 reset to 0.
SS3*	101004	Skip on sense switch 3 set to 1.
SR3*	100004	Skip on sense switch 3 reset to 0.
SS4*	101002	Skip on sense switch 4 set to 1.
SR4*	100002	Skip on sense switch 4 reset to 0.
SSS*	101036	Skip on all sense switches set to 1.
SSR*	100036	Skip on any sense switch reset to 0.
SSC	101001	Set CBIT to 1.
SRC	100001	Reset CBIT to 0.

Note

These are restricted instructions.

Note that you do not have to specify the unique mnemonic to test a particular condition; you can specify the SKP mnemonic and give the correct bit configuration for bits 7-16 of the desired test. Make sure that you set bit 7 of the SKP instruction properly: if it contains a 1, the skip occurs only if all the specified conditions are true; if it contains a 0, the skip occurs if any of the specified conditions are true.

▶ SKS function,device
Skip on Condition Met
0 1 1 1 0 0 FUNCTION\4 DEVICE\6 (S, R mode form)

Tests for the condition specified in the function field of the instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged. See Chapter 12 for more information.

Note

SKS is a restricted instruction.

▶ SLE
Skip if A Less than or Equal to 0
1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A are less than or equal to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SLN
Skip on LSB of A Nonzero
1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if bit 16 of A is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SLZ
Skip on LSB of A Zero
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the bit 16 in A equals 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SMCR
Skip on Machine Check Reset to 0
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 (S, R, V mode form)

Skips the next word if the machine check flag is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

If the processer is operating in machine check mode, this instruction has no meaning; it executes as an unconditional skip.

▶ SMCS
Skip on Machine Check Set to 1
1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 (S, R, V mode form)

Skips the next word if the machine check flag is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

If the processer is operating in machine check mode, this instruction has no meaning; it executes as a NOP.

▶ SMI
Skip on A Minus
1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A are less than 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SNR n
Skip on Sense Switch N Reset to 0
1 0 0 0 0 0 0 0 1 0 1 0 N\4 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of sense switch N are 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

N specifies the sense switch to test.

Note

This is a restricted instruction.

▶ SNS
Skip on Sence Switch N Set to 1
1 0 0 0 0 0 1 0 1 0 1 0 N\4 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch N is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

N specifies the sense switch to test.

Note

SNS is a restricted instruction.

▶ SNZ
Skip on A Nonzero
1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A are not equal to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SPL
Skip on A Plus
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A are greater than or equal to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SR1
Skip on Sense Switch 1 Reset to 0
1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 1 is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SR2
Skip on Sense Switch 2 Reset to 0
1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 2 is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SR3
Skip on Sense Switch 3 Reset to 0
1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 3 is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SR4
Skip on Sense Switch 4 Reset to 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 4 is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is is a restricted instruction.

▶ SRC
Skip on CBIT Reset to 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of CBIT is 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SS1
Skip on Sense Switch 1 Set to 1
1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 1 is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SS2
Skip on Sense Switch 2 Set to 1
1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 2 is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SS3
Skip on Sense Switch 3 Set to 1
1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 3 is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SS4
Skip on Sense Switch 4 Set to 1
1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of sense switch 4 is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SSC
Skip on CBIT bit Set to 1
1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 (S, R, V mode form)

Skips the next sequential 16-bit word if the value of CBIT is 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ SSM
Set the Sign of A Minus
1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 (S, R, V mode form)

Sets bit 1 of A to 1. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ SSP
Set the Sign of A Plus
1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 (S, R, V mode form)

Sets bit 1 of A to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ SSR
Skip on All Sense Switches Reset to 0
1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the values of sense switches 1, 2, 3, and 4 are all 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ SSS
Skip on All Sense Switches Set to 1
1 0 0 0 0 0 1 0 0 0 0 1 1 1 1 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the values of sense switches 1, 2, 3, and 4 are all 1. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ STA address
Store A into Memory
I X 0 1 0 0 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 0 0 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 0 0 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Stores the contents of the A register in the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ STAC address
Store A Conditionally
0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 (V mode form)
AP\32

Compares the contents of B with the contents of the location referenced by the specified address pointer. If the two values are equal, the instruction stores the contents of A into that referenced location. If the two values are not equal, execution continues with the next instruction. Leaves the values of CBIT and LINK unchanged. Sets the condition codes to EQ if the store occurs and to NE if not.

Note that the comparison and store will not be separated by execution of other instructions. This means that no instruction can alter the contents of the specified memory location between the compare and the store.

Note

This instruction is useful when two cooperating, sequential processes are manipulating shared data. It is interlocked against direct memory I/O; this means you can use it to interlock a process with a DMA, DMC, or DMQ channel, as well as to interlock a memory location that is possibly accessed by I/O.

▶ STC flr
Store Character
0 0 0 0 0 0 1 0 1 1 0 1 FLR 0 1 0 (V mode form)

If the contents of the specified FLR are nonzero, the instruction stores the contents of bits 9-16 of A into the character byte pointed to by the appropriate FAR. Updates the contents of the appropriate FAR so that they point to the next character. Decrements the contents of the specified FLR by 1. Sets the condition code NE.

If the contents of the specified FLR are 0, the instruction sets the condition code EQ and does not store a character.

The instruction leaves the values of LINK and CBIT unchanged.

Note

When the instruction specifies FLR0, FAR0 is used; FLR1, FAR1.

▶ STEX
Stack Extend
0 0 0 0 0 0 1 0 1 1 0 0 1 1 0 1 (V mode form)

Extends the length of the procedure stack.

A and B contain a 32-bit number specifying the word size of the extension.

The firmware rounds up the number specified by A and B to an even number of words. The instruction uses this value to allocate a block of memory to the procedure stack. Note that the extension and the initial stack do not have to be contiguous, since there may not have been enough room left in the initial stack to contain a complete frame.

The instruction returns a segment number/word number in A and B that specifies the starting address of the extension.

The extension is automatically deallocated when the current procedure completes execution. There is no limit on the number of extensions you can make.

A stack fault occurs if there is no room for the extension. The values of LINK and the condition codes are indeterminate. See Chapters 8 and 11 for more information about this instruction, stacks, and stack faults.

▶ STFA far,address
Store FAR
0 0 0 0 0 0 1 0 1 1 0 1 FAR 0 0 0 (V mode form)
AP\32

Stores the specified FAR contents as a hardware recognizable indirect pointer at the memory location referenced by the specified address pointer. If the bit number field of the specified FAR contains 0, the instruction stores the first two words of the pointer and clears the pointer's extend bit to 0. If the bit number field of the specified FAR does not contain 0, the instruction saves all three words of the pointer and sets the pointer's extend bit to 1. Leaves the values of CBIT, LINK, and the condition codes indeterminate.

▶ STL address
Store Long
I X 0 1 0 0 1 1 0 0 0 Y 1 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Stores the contents of L in the 32-bit location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ STLC
Store L Conditionally
0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 (V mode form)
AP\32

Calculates an effective address, EA. Stores the contents of L into the 32-bit location specified by EA if and only if the contents of the specified location equal the contents of E. Leaves the values of CBIT and LINK unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

Note

This instruction is useful when two cooperating, sequential processes are manipulating shared data. It is interlocked against direct memory I/O; this means you can use it to interlock a process with a DMA, DMC, or DMQ channel, as well as to interlock a memory location that is possibly accessed by I/O.

▶ STLR address
Store L into Addressed Register
I X 0 0 1 1 1 1 0 0 0 Y 0 1 BR\2 (V mode form)
DISPLACEMENT\16

Calculates a doubleword effective address, EA. Stores the contents of L into the register location specified by the word portion of EA. Bit 2 and bit 12 of the word portion of EA determine the actions of this instruction:

Bit 2	Bit 12	Action
1*	——	Ignore bits 1 and 3-9. The word portion of EA specified an absolute register number from 0-'377.
0*	1	Bits 13-16 of the word portion of EA specify one of the registers '20-'37 in the current register set.
0	0	Bits 13-16 of the word portion of EA specify one of the registers 0-'17 in the current register set.

*This is a restricted instruction.

Leaves the values of CBIT and LINK unchanged; the values of the condition codes are indeterminate. See Chapter 9 for more information about register sets.

Note

Do not use this instruction to write into the keys or modals. You can use LPSW or a mode control operation to change either of these registers. Under no circumstances should you try to change the value of the current register set bits contained in the modals.

In addition, do not change the contents of the procedure base register (PB) with this instruction. Use either LPSW or a control transfer. Loading any value other than 0 into PBL will change future effective address calculations for the currently running process.

▶ STPM
Store Processor Model Number
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 (V mode form)

Stores the CPU model number and microcode revision number in an 8-word field. XB contains a pointer to the field. The format of the field is shown in Table 13-9.

Table 13-9
STPM Memory Field Format

Word	Name	Description
1-2	Processor Model Number	Contains a code specifying the machine: 0L - 400/500, no Rev B microcode 1L - 400, Rev. B microcode 2L - Reserved 3L - 350 4L - 450/550 5L - 750 6L - 650 7L - 250 8L - 850 9L - 250-II 10L - 550-II 11L - 2250 15L - 9950
3-4	Microcode Revision	Word 1: Bits 1-8 reserved Bits 9-16 manufacturing microcode revision number Word 2: Bits 1-16 engineering microcode revision number
5	Processor Line	Specifies options enabled for this machine: Bits 1-15 reserved; must be 0 Bit 16 marketing segment specification bit
6	Extended Microcode ID	To be implemented
7-8	——	Reserved for future use

This instruction leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ STTM
Store Process Timer
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 (V mode form)

Valid for the 550-II, 750, 850, I450, and 9950.

XB contains the address of a 3-word memory block. Stores the 48-bit process timer in the block referenced by XB. Returns ETH, ETL+ITH, ITL as the current process time. Bits 1-10 of ITL contain the microsecond count.

The addition performed by this instruction is a 48-bit operation. It is equivalent to the following series of instructions:

```
LDA    ITH    /* Load A with the contents of ITH.
PIDA          /* Sign extend the contents of A into L.
ADL    ET     /* Add ET and L (containing ITH).
```

▶ STX address
Store X
I 0 1 1 0 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I 0 1 1 0 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I 0 1 1 0 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Stores the contents of X at the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction cannot directly specify indexing, though an address in the indirection chain may do so in 16S mode. See Chapter 2 for more information.

▶ STY
Store Y
I 1 1 1 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode form)
DISPLACEMENT\16

Calculates an effective address, EA. Stores the contents of Y at the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

The STY instruction cannot do indexing. See Chapter 2 for more information.

▶ SUB address
Subtract
I X 0 1 1 1 1 1 0 0 0 Y 0 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 1 1 1 1 1 0 0 0 0 0 0 CB\2 (R mode long)
[DISPLACEMENT\16]

I X 0 1 1 1 DISPLACEMENT\10 (S mode; R, V mode short)

Calculates an effective address, EA. Fetches the 16-bit integer contained in the location specified by EA and subtracts them from the contents of A. Stores the results in A.

If the result is greater than 2**15, an integer exception occurs and the instruction sets CBIT to 1 and loads bit 1 of A with a 1 and bits 2-16 with (result minus (2**15)).

If the result is less than -2**15, an integer exception occurs and the instruction loads bit 1 of A with 0 and bits 2-16 with the negative of (result + (2**15)).

If no overflow occurs, the instruction resets CBIT to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ SVC
Supervisor Call
0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 (S, R, V mode form)

Supervisor call. Generates a directed fault. Leaves the values of LINK, CBIT, and the condition codes unchanged.

This instruction allows you to make an operating system request that is addressing mode independent. An operation code followed by argument pointers is sent to the operating system. For more information, refer to Chapter 11.

▶ SZE
Skip on A Zero
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 (S, R, V mode form)

Skips the next sequential 16-bit word if the contents of A equal 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TAB
Transfer A to B
1 1 0 0 0 0 0 0 1 1 0 0 1 1 0 0 (V mode form)

Transfers the contents of A into B. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TAK
Transfer A to Keys
0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 (V mode form)

Moves a copy of the contents of A into the keys. Loads CBIT, LINK, and the condition codes as a result of the operation. Resets bits 15-16 of the keys to 0.

Note

If the new contents of the keys specifies a new addressing mode, the new mode takes effect with the instruction immediately following TAK.

▶ TAX
Transfer A to X
1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 (V mode form)

Loads X with a copy of the contents of A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TAY
Transfer A to Y
1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 (V mode form)

Loads Y with a copy of the contents of A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TBA
Transfer B to A
1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 (V mode form)

Transfers a copy of the contents of B to A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TCA
Two's Complement A
1 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1 (S, R, V mode form)

Forms the two's complement of the contents of A and stores the result in A. If the number to be complemented is -2**15, an integer exception occurs and the instruction loads -2**15 into A. If no integer exception occurs, the instruction resets CBIT to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ TCL
Two's Complement Long
1 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 (V mode form)

Forms the two's complement of the contents of L and stores the result in L. If the number to be complemented is -2**31, an integer exception occurs and the instruction loads -2**31 into L. If no integer exception occurs, the instruction resets CBIT to 0. The condition codes reflect the result of the operation. (See Table 5-3.) LINK contains the carry-out bit.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ TFLL flr
Transfer FLR to L
0 0 0 0 0 0 1 0 1 1 0 1 FLR 0 1 1 (V mode form)

Transfers the contents of the specified FLR into L as an unsigned, 32-bit integer. Clears bits 1-11 of L to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ TKA
Transfer Keys into A
0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 (V mode form)

Moves a copy of the keys into A. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ TLFL flr
Transfer L to FLR
0 0 0 0 0 0 1 0 1 1 0 1 FLR 0 0 1 (V mode form)

Transfers the 32-bit unsigned integer contained in L into the specified FLR. Clears bits 1-11 of L to 0 so that bits 1-6 of the specified FLR will be 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction allows you to load the specified FLR with a value computed at execution time. The maximum allowable integer you can load is 2**20. This number is 21 bits wide and equals the number of bits in a 64K segment.

▶ TSTQ address
Test Queue
1 1 0 0 0 0 1 1 1 1 1 0 1 1 1 1 (V mode form)
AP\32

The address pointer in this instruction is to the QCB of a queue. This instruction tests the referenced queue and sets A to equal the number of items in the queue. Sets the condition codes to EQ when the queue is empty. If the queue is not empty, sets the condition codes to NE. Leaves the values of CBIT and LINK unchanged.

▶ TXA
Transfer X to A
1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 0 (V mode form)

Transfers a copy of the contents of X to A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TYA
Transfer Y to A
1 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 (V mode form)

Transfers a copy of the contents of Y to A. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ WAIT
Wait
0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 (V mode form)
AP\32

The address pointer in this instruction is to a 16-bit semaphore counter, C. The instruction increments C. If C is greater than 0, either the resource is not available, or the event has not occurred. The instruction removes the PCB from the ready list and adds it to the wait list associated with the semaphore. It then makes the register set available and turns off the process timer.

If C is less than or equal to 0, the currently executing process continues.

If the instruction places the PCB on the wait list, no general registers are saved. This means that a process cannot depend on these registers to be intact after this instruction occurs. This instruction potentially clears the general, floating, and XB registers.

Leaves LINK, CBIT, and the condition codes unchanged.

For more information about semaphores, PCBs, and wait lists, refer to Chapter 8.

Note

This is a restricted instruction.

▶ XAD
Decimal Add
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 (V mode form)

Performs a decimal arithmetic operation under control of FAR0, FAR1, and L.

FAR0 contains the address of field 1. FAR1 contains the address of field 2. L contains the control word; fields B and C of the control word specify the decimal operation to be performed, as shown in Table 13-10.

Table 13-10
XAD Decimal Operations

B	C	Operation	Destination
0	0	+F1+F2	F2
0	1	+F1-F2	F2
1	0	-F1+F2	F2
1	1	-F1-F2	F2

The scale differential field in the control word specifies the difference in the decimal point alignment between F1 and F2:

SD	Relation of F1 and F2
SD>0	F1 > F2
SD=0	F1 = F2
SD<0	F1 < F2

If the T bit contains a 1, the results will be forced positive. For more information about decimal arithmetic, refer to Chapter 6.

If the add operation results in an overflow, a decimal exception occurs. If no overflow occurs, the instruction sets CBIT to 0 to indicate success.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers is indeterminate. The value of LINK is indeterminate. The condition codes reflect the state of F2 after the decimal operation. (See Table 5-3.)

▶ XBTD
Binary to Decimal Conversion
0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 (V mode form)

Converts a binary number to a decimal number. FAR0 contains the decimal field address. L contains the control word.

This instruction uses fields A, E, and H in the control word. H specifies the length of the binary number and its location:

H	Length	Location
0	16 bits	EH register
1	32 bits	E register
2	64 bits	FP1 register

Converts the specified binary integer to a decimal integer and stores the result in the location specified by FAR0. Leaves the values of LINK unchanged. Overflow results in a decimal exception. If no overflow occurs, the instruction sets CBIT to 0. The values of the condition codes are indeterminate.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, and FLR0. At the end of the instruction, the contents of these registers are indeterminate.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

Note

This instruction does not use or modify FAR1, FLR1, or FAC1.

▶ XCA
Exchange and Clear A
1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 (S, R, V mode form)

Interchanges the contents of registers A and B, then clears A to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ XCB
Exchange and Clear B
1 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 (S, R, V mode form)

Interchanges the values of A and B and then clears B to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ XCM
Decimal Compare
0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 (V mode form)

Compares two decimal numbers and sets the condition codes depending on the result of the compare.

FAR0 contains the address of field 1 (F1). FAR1 contains the address of field 2 (F2). L contains the control word. This instruction uses fields A, B, C, E, F, G, and H of the control word.

Compares the two specified numbers. The instruction uses the G field of the control field to adjust the two numbers before the compare:

G	Decision
>0	Low-order digits of F1 only affect the initial borrow from the low-order digit of F2.
<0	Assume F1 is zero-extended with low 0s.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate. Leaves the value of LINK indeterminate. The condition codes reflect the result of the compare:

CC	Test Result
GT	F2 > F1
EQ	F2 = F1
LT	F2 < F1

▶ XDTB
Decimal to Binary Conversion
0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 (V mode form)

Converts a decimal string to a binary string.

FAR0 contains the address of the decimal string. L contains the control word; this instruction uses the A, E, and H fields. Field H specifies the length of the binary string and its location:

H	Length	Destination Register
00	16 bits	A register
01	32 bits	L register
10	64 bits	L|E

Converts the decimal string to a binary string of the specified type and stores it in the specified register. A conversion error causes a decimal exception. Leaves the value of LINK unchanged. The values of the condition codes are indeterminate.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, and FLR0. At the end of this instruction the contents of these registers are indeterminate.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

Note

This instruction does not use or modify FAR1, FLR1, or FAC1.

▶ XDV
Decimal Divide
0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1 (V mode form)

Divides a decimal number, D2, by another, D1, and stores the quotient and remainder in the location of D2.

FAR0 contains the address of D1. FAR1 contains the address of D2. L contains the control word; this instruction uses fields A, B, C, E, F, H, and T.

Both decimal numbers must be in trailing sign embedded format. In addition, D2 must contain a number of leading 0s equal to the length of D1.

The instruction divides the two numbers. After the divide, the location of D2 contains the quotient of length (D2 length - D1 length) followed by the remainder of length (D1 length). Since D2 had leading 0s, no overflow can occur.

If the T bit contains a 1, the results will be forced positive. For more information about decimal arithmetic, refer to Chapter 6.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

At the end of the instruction, the condition codes and LINK contain undefined results. If no overflow occurs, CBIT is reset to 0.

If D1 is 0, overflow occurs which causes a decimal exception. Decimal exceptions also occur if D1 or D2 have the incorrect data type or if the length of D2 is less than that of D1. If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

▶ XEC address
Execute
I X 0 0 0 1 1 1 0 0 0 Y 1 0 BR\2 (V mode long)
DISPLACEMENT\16

I X 0 0 0 1 1 1 0 0 0 0 1 0 CB\2 (R mode long)
[DISPLACEMENT\16]

Calculates an effective address, EA. Executes the instruction found at EA, but does not transfer control to that location. Leaves the values of LINK, CBIT, and the condition codes modified as specified by the executed instruction.

The XEC instruction has limited application since all instructions cannot be executed in this way. The XEC instruction is useful for 16-bit register generic instructions such as shifts, rotates, clears, interchanges, and NOPs.

The following instruction types should not be used with XEC since they may not execute properly or will produce undefined results: instructions that change the address mode, program counter, or instruction stream; instructions that cause arithmetic faults; decimal or character instructions; and generic skips.

▶ XED
Numeric Edit
0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 (V mode form)

Edits the contents of a string under control of a subprogram.

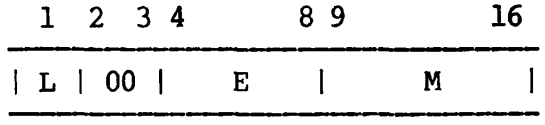
The registers used are GR2 (L), XB, FAR0, FAR1, and FLR0. At the end of the instruction, the contents of these registers are indeterminate.

FAR0 contains the address of the source string. The source string must be leading separate sign type and must have at least the same number of decimal digits and the decimal point alignment as called for in the edit subprogram.

FAR1 contains the address of the destination string. Bits 1-8 of A contain the floating character; bits 9-16, the status register. Bits 1-8 of B contain the number of remaining bytes to be processed (used if a fault or interrupt occurs). Bits 9-16 of B contain the suppression character whose initial value is determined by bit 12 of the keys ('240 if bit 1 contains 0; '40 if bit 12 contains 1). XB contains the address of the edit subprogram.

The instruction uses an edit subprogram to alter a source string and store the edit result in a destination location(s). To set up, perform a decimal move to correct the correct the type, alignment, and length of the number to be edited. Next, use a LCEQ instruction to set up the initial contents of the register.

Each word in the edit subprogram has the format shown in Figure 13-6.



Edit Subprogram Word Format
Figure 13-6

where L is 1 if this word is the last word in the subprogram,
0 if it is not the last word;
E is a suboperator;
M is a suboperator modifier.

This instruction uses several variables internally to control the edit subprogram. These are shown in Table 13-11.

Table 13-11
XED Internal Variables

Var	Definition
SC	Zero suppression character; contained in B. Initial value is the space character ('240 or '40, depending on whether bit 12 of the keys contains 0 or 1.
FC	Floating edit character; contained in A. Initial value is not defined.
SIGN	Sign of the source field. The first character fetch sets up the value of this variable.
SIG	End zero suppression flag.

There are 17 edit suboperators, shown in Table 13-12.

Table 13-12
XED Suboperators

Subop	Mnem	Name and Description
00	ZS	Zero Suppress. Fetches M digits from the source field consecutively, each time checking SIG. If SIG is 1, copies the digit into the destination string. If SIG is 0 and the digit is not 0, inserts the floating character (if defined) and copies the digit into the destination field. If SIG is 0, the digit is not 0, and the floating character is not defined, sets the SIG flag and copies the digit into the destination. If SIG and the digit are both 0, substitutes SC for the 0 digit in the destination field.
01	IL	Insert Literal. Copies M into the destination string. Increments XB and FAR1 by 1.
02	SS	Set Suppress Character. Sets SC to M and increments XB by 1.
03	ICS	Insert Character. If SIG is 1, copies M into the destination string. If SIG is 0, copies SC into the destination string. Increments XB and FAR1 by 1.
04	ID	Insert Digits. If SIG is 0, and FC is defined, copies FC and M digits into the destination field then sets SIG to 1. Increments XB by 1, FAR0 by M, and FAR1 by M+1. If SIG is 0 and FC is not defined, sets SIG to 1 and copies M digits from the source to the destination; increments XB by 1 and both FAR0 and FAR1 by M. If SIG is 1, copies M digits from the source to the destination and increments XB by 1 and both FAR0 and FAR1 by M.

Table 13-12
XED Suboperators (continued)

Subop	Mnem	Name and Description
05	ICM	Insert Character if Minus. If SIGN = 1, copies M into the destination string. If SIGN = 1, copies SC into the destination string. Increments both SB and FAR1 by 1.
06	ICP	Insert Character if Plus. If SIGN = 0, copies M into the destination string. If SIGN = 1, copies SC into the destination string. Increments both SB and FAR1 by 1.
07	SFC	Set Floating Character. Sets FC to M and increments XB by 1.
10	SFP	Set Floating if Plus. If SIGN = 0, sets FC to M. If SIGN = 1, sets FC to SC. Increments XB by 1.
11	SFM	Set Floating if Minus. If SIGN = 1, sets FC to M. If SIGN = 0, sets FC to SC. Increments XB by 1.
12	SFS	Set Floating to SIGN. If SIGN = 0, sets FC to '253. If SIGN = 1, sets FC to '255. Increments XB by 1.
13	JZ	Jump if Zero. If the condition flag in A = 0, increments XB by 1. If the condition flag in A = 1, adds M to XB and then increments XB by 1.
14	FS	Fill with Suppression Characters. Copies SC M times into the destination string. Increments XB by 1 and FAR1 by M.
15	SF	Set Significance. If SIG = 0 and FC <> 0, inserts FC into the destination string, sets SIG to 1, and increments XB and FAR1 by 1. If SIG = 0 and FC = 0, sets SIG to 1 and increments XB and FAR1 by 1. If SIG = 1, increments XB by 1.
16	IS	Insert Sign. If SIGN = 0, copies '253 into the destination string. If SIGN = 1, copies '255 into the destination string. Increments XB by 1.
17	SD	Suppress Digits. Fetches M digits from the source string and checks if they are '260. If the source digit = '260, inserts SC into the destination string. If the source digit <> '260, copies the source digit into the destination string. Increments XB by 1 and both FAR0 and FAR1 by M.
20	EBS	Embed Sign. Fetches M digits from the source string. If SIGN = 0, copies each digit into the destination string. If SIGN = 1, embeds a minus sign into each digit before copying it into the destination string. Table 5-14 shows the characters used to represent the sign/digit combinations. Note that } represents negative 0.

▶ XMP
Decimal Multiply
0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 (V mode form)

Multiplies one decimal number, D2, by another, D1, and stores the result in D2's location in memory.

FAR0 contains the address of D1. FAR1 contains the address of D2. L contains the control word; this instruction uses fields A, B, C, E, F, G, H, and T. Note that field G, the scale differential, must contain the number of decimal digits in the multiplier (M). This value is not the same as the length of the D2.

For correct results, D2 must contain a number of leading 0s equal to or greater than the length of D1.

The instruction multiplies D2 by D1 and stores the result in the location specified by FAR1. The result of the multiply is:

D1 x D2 + partial product field

The partial product field is equal to:

length(D2) - M.

The partial product field is left justified in D2's location. The maximum partial product added in per traverse of the multiplicand is:

source digits + multiplier digits processed

Note that there is also an implied weighting of the partial product field. The weighting is:

10 ** multiplier digits

If the T bit is set to 1, the results are forced positive. See Chapter 6 for more information about decimal arithmetic.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, and XB. At the end of this instruction, the contents of these registers are indeterminate. At the end of the instruction, the condition codes reflect the state of the result. (See Table 5-3.) Overflow causes a decimal exception; if no overflow occurs, resets CBIT to 0. LINK contains undefined results.

A decimal exception occurs if there are more potential or actual product digits than there is space in D2.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

▶ XMV
Decimal Move
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 (V mode form)

Moves a string of characters from one location to another.

FAR0 contains the address of the source string. FAR1 contains the address of the destination string. L contains the control word; this instruction uses fields A, B, D, E, F, G, H and T.

The instruction moves the contents of the source field into the destination field from right to left. If the B field in the control word is 1, changes the the sign of the source field during the move. If the D field in the control word is 1 and the scale differential is greater than 0, the instruction rounds the source field during the move. If the scale differential (from the H field) is less than 0, the instruction pads the source field with SD trailing zeroes before transferring.

Note that since the T bit is used by all systems for this instruction, the result is forced positive if this bit is set to 1.

The registers used are GR0, GR1, GR2 (L), GR3 (E), GR4, GR6, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

A decimal exception occurs if there are more non-zero source digits than there is room in the destination, after any padding. If there is no decimal exception, CBIT is reset to 0. Leaves the value of LINK indeterminate. The values of the condition codes reflect the state of the destination field after the move. (See Table 5-3.)

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. If no exception occurs, the instruction sets CBIT to 0. See Chapter 11 for more information about decimal exceptions.

Note

The source and destination strings may not overlap in memory.

▶ ZCM
Compare Character Field
0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 (V mode form)

Compares two fields and sets the condition codes depending on the result of the compare.

FAR0 contains the address of field 1 (F1). FLR0 contains an integer specifying the length of F1. FAR1 contains the address of field 2 (F2). FLR1 contains an integer specifying the length of F2.

The instruction compares the contents of F1 and F2 on a byte by byte basis. If the fields are not of equal length, the instruction automatically extends the shorter string with space characters. A space character is '240 or '40 when bit 12 of the keys contains 0 or 1, respectively. Sets the condition codes as a result of the compare:

Result of Compare	Set Condition Codes
F1 > F2	GT
F1 = F2	EQ
F1 < F2	LT

The registers used are GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1; at the end of this instruction, the contents of these registers are indeterminate.

When the instruction completes execution, the values of CBIT and LINK are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZCM does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

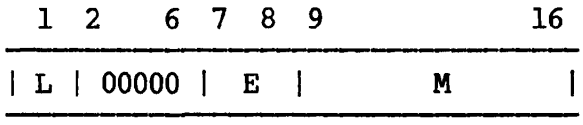
▶ ZED
Character Field Edit
0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 (V mode form)

Controls an edit subprogram.

Uses registers FAR0, FAR1, FLR0, and XB. At the end of this instruction the contents of these registers are indeterminate.

FAR0 contains the address of the source string. FLR0 specifies the length of the source string. FAR1 contains the address of the destination string. XB contains the address of the edit subprogram.

The instruction uses the edit subprogram to alter the source string, then loads the edited result into the destination string. The subprogram, addressed by the contents of XB, contains a list of commands, each with the format shown in Figure 13-7:



ZED Subprogram Word Format
Figure 13-7

where L is 1 if this command is the last command in the subprogram,
0 if it is not;
E is the edit opcode;
M is the edit modifier.

Note that bits 2-6 must be 0.

M, the operator modifier, specifies information E uses when editing the source string. (See Table 13-13.)

E, the edit suboperator, specifies the operation to be performed on the source string. Available values for E are shown in Table 13-13.

Table 13-13
ZED Suboperators

Subop	Value	Action
CPC	00	Copies characters from the source string into the destination string. If the length of the source string is greater than the contents of the M field, then CPC moves a total of M source characters into the destination string, increments FAR0 and FAR1 by M, increments XB by 1, and decrements FLR0 by M. If the length of the source string is less than the contents of the M field, then CPC moves the rest of the source string into the destination string, and then pads the remaining space to be filled with spaces. (See note.) Increments FAR0 by FLR0, increments FAR1 by M, increments XB by 1, and decrements FLR0 by FLR0 (so FLR0 - 0).
INL	01	Inserts M into the destination string and increments XB and FAR1 by 1.
SKC	10	Skips characters in the source string. If the remaining length of the source string is greater than or equal to the contents of the M field, SKC skips over the next M characters of the source field by incrementing FAR0 by M and decrementing FLR0 by M. If the remaining length of the source string is less than the contents of the M field, SKC skips over FLR0 characters in the source string by incrementing FAR0 by FLR0 and decrementing FLR0 by FLR0 (FLR0 = 0). In either case, SKC increments XB by 1.
BLK	11	Inserts M spaces into the destination string, increments FAR1 by M, and increments XB by 1. A space is '240 or '40, depending on whether bit 12 of the keys is 0 or 1.

Note

Leaves the values of CBIT, LINK, and the condition codes indeterminate. This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZED does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

▶ ZFIL
Fill Field with Character
0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 (V mode form)

Stores a character into a series of destination bytes.

Bits 9-16 of A contain the character to be stored. FAR0 contains the starting address of the destination field (byte aligned). FLR1 contains an integer specifying the length of the destination field (in bytes).

The instruction stores the character specified in A in each byte of the destination field. If FLR1 contains 0, no operation takes place. Leaves the values of CBIT, LINK, and the condition codes indeterminate.

The registers used are GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1; at the end of this instruction, the contents of these registers are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZFIL does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

▶ ZMV
Move Character Field
0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 0 (V mode form)

Moves a character field from one location to another.

FAR0 contains the address of the source string (byte aligned). FLR0 specifies the length in bytes, N, of the source string. FAR1 contains the address of the destination string (byte aligned). FLR1 specifies the length in bytes, M, of the destination string.

Compares N and M. If N is less than M, the instruction moves the contents of the source string into the destination string followed by M-N space characters. (A space character is '240 or '40 when bit 12 of the keys is 0 or 1, respectively.) If the destination string is shorter, the instruction moves the first M characters of the source string into the destination string.

When the instruction completes, the values of FAR0, FAR1, FLR0, FLR1, CBIT, LINK, and the condition codes are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZMV does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time. This instruction does not work with overlapping strings. See Chapter 6 for more information.

▶ ZMVD
Move Characters Between Equal Length Strings
0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 (V mode form)

Moves characters from one string to another of equal length.

FAR0 contains the address of the source string. FAR1 contains the address of the destination string. FLR1 contains the number of characters to move, N.

The instruction moves N characters from the source string to the destination string. Characters are moved from lower addresses to higher addresses.

The registers used are GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1; at the end of this instruction, the contents of these registers are indeterminate. The values of CBIT, LINK, and the condition codes are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZMVD does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time. This instruction does not work with overlapping strings. See Chapter 6 for more information.

▶ ZTRN
Character String Translate
0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 (V mode form)

Translates a string of characters and stores the translations in the specified destination.

FAR0 contains the address of the source string (byte aligned). FAR1 contains the address of the destination string (byte aligned). FLR1 specifies the length of the source and destination strings. XB contains the starting address of a translation table. Each byte in the 256-byte table contains an alphabetic character.

The ZTRN instruction uses the address in FAR0 to reference a character. It interprets this character as an integer, adding it to the contents of XB to form an address into the translation table. The instruction takes the referenced character in the translation table and writes it into the location specified by FAR1. After storing the character, the instruction increments the contents of FAR0 and FAR1 by 1, decrements the contents of FLR0 by 1, and repeats the operation until FLR1 contains 0.

At the end of the instruction, FAR0 and FAR1 point to the last byte in the source and destination strings, respectively. FLR1 contains 0. Leaves the values of XB, CBIT, LINK, and the condition codes unchanged.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZTRN does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

14
I Mode Instruction Dictionary

INTRODUCTION

This chapter contains descriptions for all 50 Series instructions used in I mode. In the description of each instruction, you will find:

- The instruction mnemonic followed by any arguments.
- The name of the instruction.
- The bit format of the instruction.
- Detailed information describing the instruction's action.
- Information about the how the instruction affects LINK, CBIT, and the condition codes.

Notation Conventions

Several abbreviations and symbols are used throughout this dictionary. Table 14-1 defines the dictionary notation.

Table 14-1
Dictionary Notation

Symbol	Meaning
A	The 16-bit A register.
ADDRESS	Encompasses all the elements needed to specify an effective address. This term is used because various types of addressing require you to specify the elements in different orders (such as indirect or pre- and post-indexing).
AP	Address pointer.
B	The 16-bit B register.
BR	Base register.
CBIT	Bit 1 of the keys.
DAC	The double precision floating-point accumulator.
DR	Destination register (normal register specifier).
E	The 32-bit E register.
EA	Effective address.
F	Floating accumulator.
FAC	The single precision floating-point accumulator.
FAR	Field address register.
FLR	Field length register.
GR2	General register 2.
I	Indirect bit.
L	The 32-bit L register.
LINK	Bit 3 of the keys.
QAC	The quad precision floating-point accumulator.
PB	The procedure base register.
R	A 32-bit general register.

Table 14-1 (continued)
Dictionary Notation

Symbol	Meaning
r	Bits 1-16 of a general register.
skip	Skip the next 16-bit word before continuing execution.
SR	Source register (or index if memory reference).
TM	Tag modifier. Bits used in I mode effective address calculation to specify indirection, indexing, etc.
X	The X register (indexing).
XB	Auxiliary base register.
Y	The Y register (indexing).
m\n	Specifies the number of bits, n, occupied by field m.
[ ]	Specifies an optional argument.

Resumable Instructions

Some assembly language instructions are resumable. When a fault or interrupt occurs during instruction execution, the processor usually services it, then restarts the instruction from the beginning. Some instructions, however, are too long or too complex for this to be desirable. When a fault or interrupt occurs during one of the resumable instructions, the processor preserves the state of the instruction, handles the fault or interrupt, and then resumes the instruction at the point where the interrupt occurred. Table 14-2 lists the resumable assembly language instructions.

Table 14-2
Resumable Instructions

Instructions			
ARGT	XAD	XBTD	XCM
XDTB	XDV	XED	XMP
XMV	ZCM	ZED	ZFIL
ZMV	ZMVD	ZTRN	STEX

Storing Data Into the 9950 Instruction Stream

After any instruction that stores data into memory, you must wait five instructions before executing data. If in doubt about the next five instructions (temporally) to be executed, a mode change instruction to the current addressing mode, such as E32I, allows the stored data to be executed.

Instruction Formats

All I mode instructions belong to one of the following instruction types:

- I Mode Memory Reference
- I Mode Special Memory Reference
- I Mode Generic AP (Address Pointer)
- I Mode Register Generic
- I Mode Register Generic Branch
- Generic A and B (see below)

The format of each instruction type is shown in Figure 14-1.

Memory reference instructions have the opcode in bits 1-6. Special memory reference instructions (for floating point) have the opcode in bits 2, 3, 7, and 9; bit 8 specifies the floating accumulator. Some memory reference and special memory reference instructions have register-to-register and/or immediate forms. Such instructions are so identified in this I Mode Instruction Dictionary.

The immediate form of a memory reference instruction has a 16-bit literal in bits 17-32 instead of a 16-bit displacement. Register-to-register forms are 16 bits long, since they have no displacement. Bits 7-9 specify the destination register and bits 12-14 specify the source register.

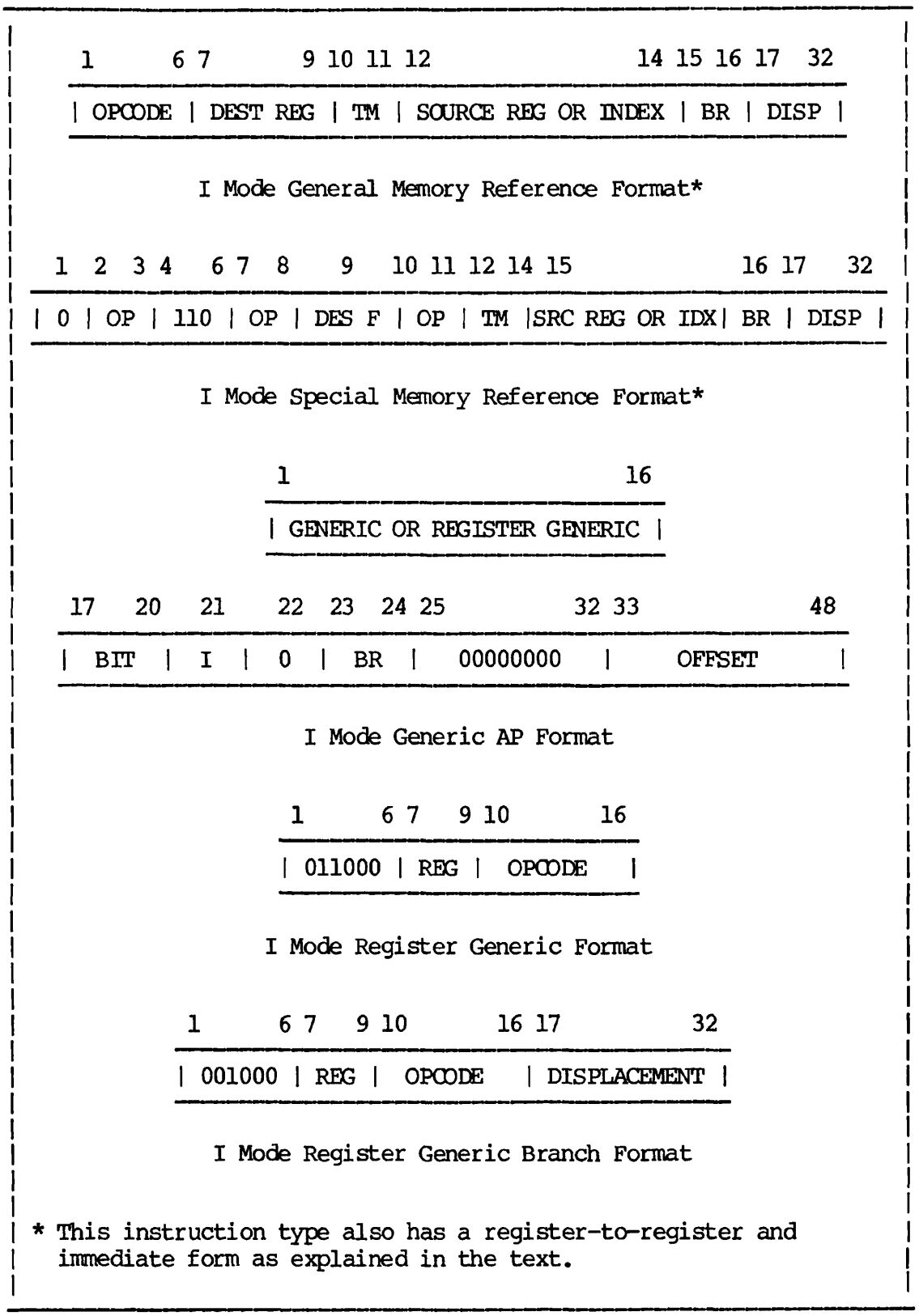
The immediate form of a special memory reference instruction has a 16-bit encoding in bits 17-32 instead of a 16-bit displacement. The register-to-register form is 16 bits long, since it has no displacement. Bit 8 specifies the floating-point destination accumulator and bits 12-14 specify the index register or the floating-point source register (in bit 13).

Generic AP instructions have a generic format (where bits 10-16 contain the opcode extension) followed by a 32-bit address pointer.

Register generic instructions are 16 bits long and have an opcode in bits 10-16. The value of bits 1-6 is 011000; bits 7-9 specify a general register.

Register generic branch instructions are 32 bits long and have an opcode in bits 10-16. The value of bits 1-6 is 00100; bits 7-9 specify a general register. Bits 17-32 contain a displacement.

Generic A and B instructions that do not reference the A, B, E, or L registers are also used in I Mode. See Chapter 13, Figure 13-1 for the format of these instructions. Instructions defined in I mode for this class are included in this instruction dictionary.



I Mode Instruction Formats
Figure 14-1

INSTRUCTIONS

▶ A R,address
Add Fullword
0 0 0 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the value contained in the specified R to the 32-bit value contained in the location specified by EA. Stores the result in the specified R. An overflow produces an integer exception. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ ABQ address
Add Entry to Bottom of Queue
0 1 1 0 0 0 R\3 1 0 1 1 1 0 0
AP\32

Adds the entry contained in the specified R to the bottom of the queue referenced by the AP. (AP points to the queue's QCB.) Sets the condition codes to reflect EQ if the queue was full, or to NE if not full. Leaves the values of CBIT and LINK unchanged.

▶ ADLR R
Add LINK to Register
0 1 1 0 0 0 R\3 0 0 0 1 1 0 0

Adds the contents of LINK to the contents of R and stores the result in R. If there is an overflow, an integer exception occurs. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ AH r,address
Add Halfword
0 0 1 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the value contained in the specified r to the 16-bit value contained in the location specified by EA. Stores the result in r. An overflow produces an integer exception. If no integer exception occurs, CBIT is reset to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ ARFA far,R
Add Register to FAR
0 1 1 0 0 0 R\3 1 1 1 FAR 0 0 1

Adds the bit address in the specified R to the contents of the specified FAR. Stores the result in the FAR. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ ARGT
Argument Transfer
0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1

Transfers arguments from a source procedure to a destination procedure. ARGT is fetched and executed only when the argument transfer phase of a procedure call (PCL) instruction is interrupted or faulted.

To perform a procedure call and argument transfer, the source procedure must contain the PCL instruction followed by a number of argument templates. The destination procedure must begin with the ARGT instruction. When the PCL instruction is executed, control transfers to the destination procedure, and the ARGT instruction uses the templates to form the actual arguments. The arguments are stored in the new stack frame as they are computed. At the end of the ARGT instruction, the values of LINK, CBIT, and the condition codes are indeterminate.

ARGT must be the first executable instruction in any destination procedure that will use arguments. For those procedures whose entry control blocks specify zero arguments, omit ARGT.

For more information about argument transfers, refer to the section on procedure calls in Chapter 8.

▶ ATQ address
Add Entry to Top of Queue
0 1 1 0 0 0 R\3 1 0 1 1 1 0 1
AP\32

Adds the entry contained in the specified R to the top of the queue referenced by the AP. (AP points to the queue's QCB.) Sets the condition codes to reflect EQ if the queue was full, or to NE if not full. Leaves the values of CBIT and LINK unchanged.

▶ BCEQ address
Branch on Condition Code EQ
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0
ADDRESS\16

If the condition codes reflect equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCGE address
Branch on Condition Code GE
1 1 0 0 0 0 1 1 1 0 0 0 0 1 0 1
ADDRESS\16

If the condition codes reflect greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCGT address
Branch on Condition Code GT
1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1
ADDRESS\16

If the condition codes reflect greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCLE address
Branch on Condition Code LE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0
ADDRESS\16

If the condition codes reflect less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCLT address
Branch on Condition Code LT
1 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0
ADDRESS\16

If the condition codes reflect less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCNE address
Branch on Condition Code NE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1
ADDRESS\16

If the condition codes reflect not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the condition codes reflect some other condition, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCR address
Branch on CBIT Reset to 0
1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1
ADDRESS\16

If CBIT has the value 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If CBIT has the value 1, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BCS address
Branch on CBIT Set to 1
1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 0
ADDRESS\16

If CBIT has the value 1, the instruction loads the specified address into the program counter. This address must be within the current segment. If CBIT has the value 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BFEQ f,address
Branch on Floating Accumulator Equal to 0
0 0 1 0 0 0 0 F 0 1 0 1 0 0 1 0
ADDRESS\16

If the contents of the specified floating-point accumulator are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are not equal to 0, execution continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFGE f,address
Branch on Floating Accumulator Greater Than or Equal to 0
0 0 1 0 0 0 0 F 0 1 0 1 0 1 0 1
ADDRESS\16

If the contents of the specified floating-point accumulator are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are less than 0, execution continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFGT f,address
Branch on Floating Accumulator Greater Than 0
0 0 1 0 0 0 0 F 0 1 0 1 0 0 0 1
ADDRESS\16

If the contents of the specified floating-point accumulator are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are less than or equal to 0, execution continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFLE f,address
Branch on Floating Accumulator Less Than or Equal to 0
0 0 1 0 0 0 0 F 0 1 0 1 0 0 0 0
ADDRESS\16

If the contents of the specified floating-point accumulator are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are greater than 0, execution

continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFLT f,address
Branch on Floating Accumulator Less Than 0
0 0 1 0 0 0 0 F 0 1 0 1 0 0 1 0
ADDRESS\16

If the contents of the specified floating-point accumulator are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accuumulator contents are greater than or equal to 0, execution continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BFNE f,address
Branch on Floating Accumulator Not Equal to 0
0 0 1 0 0 0 0 F 0 1 0 1 0 0 1 1
ADDRESS\16

If the contents of the specified floating-point accumulator are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the floating accumulator contents are equal to 0, execution continues with the next instruction. The condition codes reflect the result of the comparison. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ BHDl r,address
Branch on Half Register Decremented by 1
0 0 1 0 0 0 R\3 1 1 0 0 1 0 0
ADDRESS\16

Decrements the contents of the specified r by 1 and stores the result in the specified r. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHD2 r,address
Branch on Half Register Decremented By 2
0 0 1 0 0 0 R\3 1 1 0 0 1 0 1
ADDRESS\16

Decrements the contents of the specified r by 2 and stores the result in the specified r. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHD4 r,address
Branch on Half Register Decremented By 4
0 0 1 0 0 0 R\3 1 1 0 0 1 1 0
ADDRESS\16

Decrements the contents of the specified r by 4 and stores the result in the specified r. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHEQ r,address
Branch on Half Register Equal To 0
0 0 1 0 0 0 R\3 1 0 0 1 0 1 0
ADDRESS\16

If the contents of the specified r are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are not equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BHGE r,address
Branch on Half Register Greater Than or Equal To 0
0 0 1 0 0 0 R\3 1 0 1 0 1 0 1
ADDRESS\16

If the contents of the specified r are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are less than 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BHGT r,address
Branch on Half Register Greater Than 0
0 0 1 0 0 0 R\3 1 0 0 1 0 0 1
ADDRESS\16

If the contents of the specified r are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are less than or equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BHI1 r,address
Branch on Half Register Incremented By 1
0 0 1 0 0 0 R\3 1 1 0 0 0 0 0
ADDRESS\16

Increments the contents of the specified r by 1 and stores the result in the specified r. If the incremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHI2 r,address
Branch on Half Register Incremented By 2
0 0 1 0 0 0 R\3 1 1 0 0 0 0 1
ADDRESS\16

Increments the contents of the specified r by 2 and stores the result in the specified r. If the incremented value is not equal to 0, the instruction loads the the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHI4 r,address
Branch on Half Register Incremented By 4
0 0 1 0 0 0 R\3 1 1 0 0 0 1 0
ADDRESS\16

Increments the contents of the specified r by 4 and stores the result in the specified r. If the incremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BHLE r,address
Branch on Half Register Less Than or Equal to 0
0 0 1 0 0 0 R\3 1 0 0 1 0 0 0
ADDRESS\16

If the contents of the specified r are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are greater than 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BHLT r,address
Branch on Half Register Less Than 0
0 0 1 0 0 0 R\3 1 0 0 1 1 0 0
ADDRESS\16

If the contents of the specified r are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are greater than or equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BHNE r,address
Branch on Half Register Not Equal To 0
0 0 1 0 0 0 R\3 1 0 0 1 0 1 1
ADDRESS\16

If the contents of the specified r are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the contents of r are equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BLR address
Branch on LINK Reset to 0
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1
ADDRESS\16

If LINK has the value 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If LINK has the value 1, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BLS address
Branch on LINK Set to 1
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 0
ADDRESS\16

If LINK has the value 1, the instruction loads the specified address into the program counter. This address must be within the current segment. If LINK has the value 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BMEQ address
Branch on Magnitude Condition EQ
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0
ADDRESS\16

Performs the same operation as the BCEQ instruction, except that it allows the result to be evaluated as unsigned.

▶ BMGE address
Branch on Magnitude Condition GE
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 0
ADDRESS\16

Performs the same function as the BLS instruction, except that it allows the result to be evaluated as unsigned.

▶ BMGT address
Branch on Magnitude Condition GT
1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 0
ADDRESS\16

If LINK is 1 and the condition codes reflect not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If some other condition exists, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BMLE address
Branch on Magnitude Condition LE
1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 1
ADDRESS\16

If LINK is 0 or the condition codes reflect equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If some other condition exists, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BMLT address
Branch on Magnitude Condition LT
1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1
ADDRESS\16

Performs the same function as the BLR instruction, except that it allows the result to be evaluated as unsigned.

▶ BMNE address
Branch on Magnitude Condition NE
1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1
ADDRESS\16

Performs the same function as the BLNE instruction, except that it allows the result to be evaluated as unsigned.

▶ BRBR R,bit #,address
Branch on Register Bit Reset
0 0 1 0 0 0 R\3 0 1 BIT\5
ADDRESS\16

Bits 12-16 of the instruction contain a value between '00 and '37. This value specifies the bit position in the register to be tested. A value of '00 corresponds to bit 1; '01, bit 2; and so on.

If the specified bit position contains 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the specified bit position contains 1, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BRBS R,bit #,address
Branch on Register Bit Set
0 0 1 0 0 0 R\3 0 0 BIT\5
ADDRESS\16

Bits 12-16 of the instruction contain a value between '00 and '37. This value specifies the bit position in the register to be tested. A value of '00 corresponds to bit 1; '01, bit 2; and so on.

If the specified bit position contains 1, the instruction loads the specified address into the program counter. This address must be within the current segment. If the specified bit position contains 0, execution continues with the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ BRD1 R,address
Branch on Register Decremented By 1
0 0 1 0 0 0 R\3 1 0 1 1 1 0 0
ADDRESS\16

Decrements the contents of the specified R by 1 and stores the result in the specified R. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BRD2 R,address
Branch on Register Decremented By 2
0 0 1 0 0 0 R\3 1 0 1 1 1 0 1
ADDRESS\16

Decrements the contents of the specified R by 2 and stores the result in the specified R. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BRD4 R,address
Branch on Register Decremented By 4
0 0 1 0 0 0 R\3 1 0 1 1 1 1 0
ADDRESS\16

Decrements the contents of the specified R by 4 and stores the result in the specified R. If the decremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the decremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BREQ R,address
Branch on Register Equal To 0
0 0 1 0 0 0 R\3 1 0 0 0 0 1 0
ADDRESS\16

If the contents of the specified R are equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are not equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BRGE R,address
Branch on Register Greater Than or Equal To 0
0 0 1 0 0 0 R\3 1 0 1 0 1 0 1
ADDRESS\16

If the contents of the specified R are greater than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are less than 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BRGT R,address
Branch on Register Greater Than 0
0 0 1 0 0 0 R\3 1 0 0 0 0 0 1
ADDRESS\16

If the contents of the specified R are greater than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are less than or equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BRI1 R,address
Branch on Register Incremented By 1
0 0 1 0 0 0 R\3 1 0 1 1 0 0 0
ADDRESS\16

Increments the contents of the specified R by 1 and stores the result in the specified R. If the incremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BRI2 R,address
Branch on Register Incremented By 2
0 0 1 0 0 0 R\3 1 0 1 1 0 0 1
ADDRESS\16

Increments the contents of the specified R by 2 and stores the result in the specified R. If the incremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BRI4 R,address
Branch on Register Incremented By 4
0 0 1 0 0 0 R\3 1 0 1 1 0 1 0
ADDRESS\16

Increments the contents of the specified R by 4 and stores the result in the specified R. If the incremented value is not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the incremented value is equal to 0, execution continues with the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ BRLE R,address
Branch on Register Less Than or Equal to 0
0 0 1 0 0 0 R\3 1 0 0 0 0 0 0
ADDRESS\16

If the contents of the specified R are less than or equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are greater than 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BRLT R,address
Branch on Register Less Than 0
0 0 1 0 0 0 R\3 1 0 0 0 1 0 0
ADDRESS\16

If the contents of the specified R are less than 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are greater than or equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ BRNE R,address
Branch on Register Not Equal To 0
0 0 1 0 0 0 R\3 1 0 0 0 0 1 1
ADDRESS\16

If the contents of the specified R are not equal to 0, the instruction loads the specified address into the program counter. This address must be within the current segment. If the R contents are equal to 0, execution continues with the next instruction. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the values of CBIT and LINK unchanged.

▶ C R,address
Compare Fullword
1 1 0 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the 32-bit value contained in the specified R to the 32-bit value contained in the location specified by EA. The comparison is done by subtracting the contents of the the memory location from the contents of the register. Sets the condition codes to the result of the comparison. (See Table 5-3.) Leaves the value of CBIT unchanged. LINK contains the carry-out bit.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ CAI
Clear Active Interrupt
0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1

Clears the current active interrupt and inhibits interrupts for the next instruction. Effective only in vectored interrupt mode. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ CALF
Call Fault Handler
0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1
AP\32

The address pointer in this instruction points to the ECB of a fault routine. CALF uses this pointer to transfer control to the fault routine as if the transfer were a normal procedure call. The values of CBIT, LINK, and the condition codes are indeterminate. See Chapter 11 for more information.

▶ CGT R
Computed GOTO
0 1 1 0 0 0 R\3 0 0 1 0 1 1 0
INTEGER N\16
BRANCH ADDRESS 1\16
...
BRANCH ADDRESS N-1\16

If the contents of the specified R are greater than or equal to 1 and less than the specified integer N that follows the opcode, the instruction adds the contents of R to the contents of the program counter to form an address. (The program counter points to the integer N following the opcode.) Loads the contents of the location specified by this address into the program counter. If the contents of R are not within this range, the instruction adds integer N to the contents of the program counter and stores the result in the program counter. Each of the branch addresses following the instruction specifies a location within the current procedure segment. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ CH r,address
Compare Halfword
1 1 1 0 0 1 DR\3 TM\2 SR\2 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the value contained in the specified r to the 16-bit value contained in the location specified by EA. Leaves the value of CBIT unchanged. LINK contains the carry-out bit. The condition codes reflect the result of the comparison. (See Table 5-3.)

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ CHS R
Change Sign
0 1 1 0 0 0 R\3 0 1 0 0 0 0 0

Complements bit 1 of the specified R. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CMH r
Complement r
0 1 1 0 0 0 R\3 0 1 0 0 1 0 1

Forms the one's complement of the contents of the specified r by inverting the value of each bit and stores the result in r. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CMR R
Complement R
0 1 1 0 0 0 R\3 0 1 0 0 1 0 0

Forms the one's complement of the contents of the specified R by inverting the value of each bit and stores the result in R. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CR R
Clear R
0 1 1 0 0 0 R\3 0 1 0 1 1 1 0

Clears the specified R to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CRBL R
Clear R High Byte 1 Left
0 1 1 0 0 0 R\3 0 1 1 0 0 1 0

Loads 0 into bits 1-8 of the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CRBR R
Clear R High Byte 2 Right
0 1 1 0 0 0 R\3 0 1 1 0 0 1 1

Loads 0 into bits 9-16 of the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ CRHL R
Clear R Left Halfword
0 1 1 0 0 0 R\3 0 1 0 1 1 0 0

Clears bits 1-16 of the specified R to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CRHR R
Clear R Right Halfword
0 1 1 0 0 0 R\3 0 1 0 1 1 0 1

Clears bits 17-32 of the specified R to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ CSR R
Copy and Save Sign
0 1 1 0 0 0 R\3 0 1 0 0 0 0 1

Copies the value of bit 1 of the specified R into CBIT, and then loads 0 into bit 1 of R. Leaves the values of LINK and the condition codes unchanged.

▶ D R,address
Divide Fullword
1 1 0 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the 64-bit value contained in the specified R and R+1 by the 32-bit value contained in the location specified by EA. Stores the quotient in the specified R and the remainder in R+1. Overflow may occur if the quotient is less than -(2**31) or greater than (2**31)-1. Overflow and divide by 0 cause an integer exception.

If no integer exception occurs, CBIT is reset to 0. The instruction leaves the values of LINK and the condition codes indeterminate.

If an integer exception occurs and bit 8 in the keys contains 0, the instruction sets CBIT to 1; if bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. For more information, see Chapter 11.

Note

R must specify an even register.

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DBLE f
Convert Single to Double Floating Point
0 1 1 0 0 0 0 F 0 1 0 0 0 1 1 0

Converts the single precision number contained in the specified floating-point accumulator to a double precision number by toing bits 25-48 of the floating-point accumulator. Stores the result in the floating-point accumulator. Leaves the values of CBIT, LINK, and the condition codes unchanged.

For 750 and 850 processors, exponent underflow is detected, but exponent overflow is not.

▶ DFA f,address
Double Floating Add
0 0 1 1 1 0 1 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the contents of the specified DAC to the contents of the location specified by EA. Stores the result in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFC f,address
Double Floating Compare
0 0 0 1 1 0 1 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the contents of the specified DAC to the contents of the location specified by EA. Leaves the values of CBIT and LINK unchanged. Sets the condition codes to the outcome of the comparison.

Condition	CC
Contents of DAC > contents of location specified by EA.	GT
Contents of DAC = contents of location specified by EA.	EQ
Contents of DAC < contents of location specified by EA.	LT

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFCM f
Double Precision Floating Complement
0 1 1 0 0 0 0 F 0 1 1 0 0 1 0 0

Forms the two's complement of the double precision, floating-point number contained in the specified DAC and normalizes it if necessary. Stores the result in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

▶ DFD f,address
Double Floating Divide
0 1 1 1 1 0 0 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the contents of the specified DAC by the contents of the location specified by EA. Normalizes the quotient if necessary. Stores the result in the DAC. An overflow or divide by to causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFL f,address
Double Floating Load
0 0 0 1 1 0 0 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the 64-bit contents of the location specified by EA into the specified DAC. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

The DFL instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFM f,address
Double Floating Multiply
0 1 0 1 1 0 1 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the 64-bit contents of the location specified by EA by the contents of the specified DAC. Normalizes the result if necessary. Stores the result in the DAC. An overflow causes a -point exception. If no floating floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFS f,address
Double Floating Subtract
0 1 0 1 1 0 0 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 64-bit contents of the location specified by EA from the contents of the specified DAC. Stores the result in the DAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

For 750 and 850 processors, exponent underflow is detected, but exponent overflow is not.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

Note

The DFS instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DFST f,address
Double Precision Floating Point Store
0 0 1 1 1 0 0 F 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the specified DAC into the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ DH R,address
Divide Halfword
1 1 1 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the 32-bit dividend contained in the specified R by the 16-bit value contained in the location specified by EA. Stores the quotient in bits 1-16 of R and the remainder in bits 17-32 of R. The sign of the remainder equals the sign of the dividend. If the quotient is less than -(2**15) or greater than (2**15)-1, an overflow occurs and causes an integer exception. The values of LINK and the condition codes are indeterminate. If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 in the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. For more information, see Chapter 11.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ DH1 r
Decrement r By 1
0 1 1 0 0 0 R\3 1 0 1 1 0 0 0

Decrements the contents of r by 1 and stores the result in r. If an overflow occurs, an integer exception occurs. LINK reflects the value of the carry. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DH2 r
Decrement r By 2
0 1 1 0 0 0 R\3 1 0 1 1 0 0 0

Decrements the contents of r by 2 and stores the result in r. If an overflow occurs, an integer exception occurs. LINK reflects the value of the carry. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DM address
Decrement Memory Fullword
1 1 0 1 1 0 0 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Subtracts 1 from the 32-bit integer contained in the specified location and stores the result back in the specified location. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ DMH address
Decrement Memory Halfword
1 1 1 1 1 0 0 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Subtracts 1 from the 16-bit integer contained in the specified location and stores the result back in the specified location. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ DR1 R
Decrement Register By 1
0 1 1 0 0 0 R\3 1 0 1 0 1 0 0

Decrements the contents of R by 1 and stores the result in R. An overflow causes an integer exception. LINK contains the value of the borrow bit. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DR2 R
Decrement Register By 2
0 1 1 0 0 0 R\3 1 0 1 0 1 0 1

Decrements the contents of the specified R by 2 and stores the result in R. An overflow causes an integer exception. LINK contains the value of the borrow bit. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ DRN
Double Round from Quad
0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0

Converts the 112-bit value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 50-96 of QAC are not to, or bit 48 of QAC contains 1, the instruction adds the value of bit 49 to that of bit 48 (unbiased round) and clears bits 49-96 of QAC to 0. If any other condition exists, no unbiased rounding occurs, but bits 49-96 of QAC are still cleared to 0. After any rounding and clearing occurs, the instruction normalizes the result and loads it into bits 1-64 of QAC.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If the DRN instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNM
Double Round from Quad towards Negative Infinity
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 1

Converts the 112-bit value in QAC to a double precision floating-point number. If QAC contains 0, or if bits 49-96 of QAC contain zeroes, the instruction ends. In any other case, the instruction clears bits 49-96 to 0, normalizes the result, and places it in bits 1-64 of QAC.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNP
Double Round from Quad towards Positive Infinity
0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1

Converts the 112-bit value in QAC to a double precision floating point number. If QAC contains 0, or if bits 49-96 of QAC contain zeroes, the instruction ends. In any other case, the instruction adds 1 to the value contained in bit 48 of QAC, clears bits 49-96 to 0, normalizes the result, and places it in bits 1-64 of QAC.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If the DRNP instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ DRNZ
Double Round from Quad towards Zero
0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 0

Converts the 112-bit value in QAC to a double precision floating-point number. If QAC contains 0, the instruction ends. If bits 49-96 of QAC contain zeroes and bit 1 contains 1, the instruction adds 1 to the value contained in bit 48 of QAC, clears bits 49-96 to 0, normalizes the result, and places it in bits 1-64 of QAC. If any other condition exists, no rounding occurs.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ E16S
Enter 16S Mode
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

Sets bits 4-6 of the keys to 000. Subsequent S mode instructions may now be interpreted, and 16S address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32I
Enter 32I Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0

Sets bits 4-6 of the keys to 100. Subsequent I mode instructions may now be interpreted, and 32I address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32R
Enter 32R Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1

Sets bits 4-6 of the keys to 011. Subsequent R mode instructions may now be interpreted, and 32R address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E32S
Enter 32S Mode
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1

Sets bits 4-6 of the keys to 001. Subsequent S mode instructions may now be interpreted, and 32S address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ E64R
Enter 64R Mode
0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1

Sets bits 4-6 of the keys to 010. Subsequent R mode instructions may now be interpreted, and 64R address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

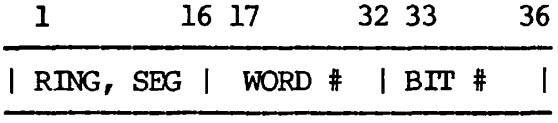
▶ E64V
Enter 64V Mode
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

Sets bits 4-6 of the keys to 110. Subsequent V mode instructions may now be interpreted, and 64V address calculations may be used to form effective addresses. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EAFA far,address
Effective Address to FAR
0 0 0 0 0 0 1 0 1 1 0 0 F 0 0 0
AP\32

Builds a 36-bit EA from the 32-bit address pointer contained in the instruction and loads it into the specified FAR. The AP bit field is processed and loaded into the bit portion of the FAR, for direct access. Indirection uses the bit field in the indirect pointer (if any). Leaves the values of CBIT, LINK, and the condition codes unchanged.

Figure 14-2 shows the format of the EA loaded into the specified FAR.



EA Format for EAFA
Figure 14-2

▶ EALB address
Effective Address to LB
1 0 0 1 1 0 0 1 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA, and loads it into LB. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ EAR R,address
Effective Address to Register
1 1 0 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the 32-bit EA into the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EAXB address
Load XB with Effective Address
1 0 1 1 1 0 0 1 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA, and loads it into XB. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ EIO address
Execute I/O
0 1 1 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Executes bits 17-32 of EA as if they were a PIO instruction. If execution is successful, the instruction sets the condition codes as follows:

CC	Meaning
EQ	Successful INA, OTA, or SKS instruction
NE	Unsuccessful INA, OTA, OR SKS; any OCP

Leaves the values of LINK and CBIT unchanged. For more information about I/O operations, see Chapter 12.

Note

This is a restricted instruction.

▶ EMCM
Enter Machine Check Mode
0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1

Enters machine check mode 3 by loading 3 into modal bits 15-16. This mode enables the reporting of all errors. The actions taken upon an error depend on whether or not the machine was in process exchange mode.

The instruction inhibits interrupts during execution of the next instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged. See Chapter 11 for more information about checks.

If an error occurs in process exchange mode, the microcode stores the machine state in the appropriate check vector and transfers control to that vector, automatically dropping back to machine check mode.

If an error occurs when the machine is not in process exchange mode, the following actions occur. If the appropriate check vector contains a nonzero value, the processor jumps indirectly through this vector to the check routine. If the check vector location contains 0, the machine halts.

Note

This is a restricted instruction.

▶ ENB
Enable Interrupts
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1

Enables interrupts by setting bit 1 of the modals to 1. Inhibits interrupts for one instruction. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENBL
Enable Interrupts (Local)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1

This 850 instruction performs the same actions as ENB, except that it is performed specifically for the local processor. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENBM
Enable Interrupts (Mutual)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

For the 850, a processor checks the availability of the mutual exclusion lock. If available, the processor sets this lock and enables interrupts. Otherwise, it waits for the lock to be released by the other processor and then sets the lock and enables interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ENBP
Enable Interrupts (Process)
0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0

For the 850, a processor checks the availability of the process exchange lock. If available, the process sets this lock and enables interrupts. Otherwise, it waits for this lock to be released by the other processor, and then sets the lock and enables interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ ESIM
Enter Standard Interrupt Mode
0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1

Enters standard interrupt mode by resetting bit 2 of the modals to 0. ESIM is meaningless when the system is in process exchange mode (that is, the value of modal bit 13 is 1). All interrupts use location '63. The processor services interrupts according to their relative positions on the I/O bus. Lower devices have higher priority. Inhibits interrupts during execution of the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged. Refer to Chapter 11 for more information about interrupts.

Note

This is a restricted instruction.

▶ EVIM
Enter Vectored Interrupt Mode
0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1

Enters vectored interrupt mode by setting bit 2 of the modals to 1. EVIM is meaningless when in the system is in process exchange mode (that is, the value of modal bit 13 is 1). The processor services interrupts according to their relative positions on the I/O bus. Lower devices have higher priority. Interrupts occur through a location specified by the interrupting device. Inhibits interrupts during execution of the next instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged. Refer to Chapter 11 for more information about interrupts.

Note

This is a restricted instruction.

▶ FA f,address
Floating Add
0 0 1 1 1 0 1 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the contents of the specified FAC to the 32-bit contents of the location specified by EA. (See Chapter 6.) Stores the result in the FAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FC f,address
Floating Compare
0 0 0 1 1 0 1 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the contents of the specified FAC to the contents of the location specified by EA. Leaves the values of LINK and CBIT unchanged. Sets the condition codes to reflect the outcome of the comparison:

Condition	CC
Contents of FAC > contents of location specified by EA.	GT
Contents of FAC = contents of location specified by EA.	EQ
Contents of FAC < contents of location specified by EA.	LT

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FCDQ
Floating Point Convert Double to Quad
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 1

Clears FAC0 to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ FCM f
Floating Point Complement
0 1 1 0 0 0 0 F 0 1 0 0 0 0 0 0

Forms the two's complement of the contents of the FAC and normalizes the result if necessary. (See Chapter 6.) Stores the result in the FAC. An overflow causes a floating-point exception. The values of LINK and the condition codes are indeterminate. If no floating-point exception occurs, CBIT is reset to 0.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

▶ FD f,address
Floating Divide
0 1 1 1 1 0 0 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the contents of the specified FAC by the contents of the location specified by EA. (See Chapter 6.) Stores the result in the FAC and normalizes if necessary. A divide by to or an overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

Note

The FD instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FL f,address
Floating Load
0 0 0 1 1 0 0 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Converts the single precision operand to double precision and loads the result into the specified FAC. Leaves the contents of CBIT, LINK, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FLT f,R
Convert Integer to Floating Point
0 1 1 0 0 0 R\3 1 0 0 F 1 0 1

Converts the integer contained in R to a floating-point number and stores the result in the specified FAC. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FLTH f,r
Convert Halfword Integer to Floating Point
0 1 1 0 0 0 R\3 1 0 0 F 0 1 0

Converts the halfword integer contained in r to a floating-point number and stores the result in the specified FAC. The values of CBIT, LINK, and the condition codes are indeterminate.

▶ FM f,address
Floating Multiply
0 1 0 1 1 0 1 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the 32-bit contents of the location specified by EA by the contents of the specified FAC. (See Chapter 6.) Normalizes the result, if necessary, and stores it in the FAC. An exponent overflow causes a floating-point exception. If

no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FRN
Floating Round
1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 0 (R, V mode form)

This instruction operates on and stores all results in the floating accumulator.

For the 9950, the following actions occur. If bits 1-48 contain 0, then bits 49-64 are cleared to 0. If bits 24 and 25 both contain 1, then 1 is added to bit 24, bits 25-48 are cleared to 0, and the result is normalized. If bit 25 contains 1 and bits 26-48 are not equal to 0, then 1 is added to bit 24, bits 25-48 are cleared, and the result is normalized.

For the rest of the 50 series, the following actions occur. If bits 1-48 contain 0, then bits 49-64 are cleared to 0. Otherwise, bit 25 is added to bit 24, bits 25-48 are cleared to 0, and the result is normalized.

For all systems, if no floating-point exception occurs, sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRNM f
FLoating Point Round towards Negative Infinity
0 1 1 0 0 0 0 F 0 1 1 0 0 1 1 0

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, or if bits 25-48 of DAC contain zeroes, the instruction ends. In any other case, the instruction clears bits 25-48 to 0, normalizes the result, and places it in DAC.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRNP f
Floating Point Round towards Positive Infinity
0 1 1 0 0 0 0 F 0 1 1 0 0 1 0 1

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, or if bits 25-48 of DAC contain zeroes, the instruction ends. In any other case, the instruction adds 1 to the value contained in bit 24 of DAC, clears bits 25-48 to 0, normalizes the result, and places it in DAC.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FRNZ f
Floating Point Round towards Zero
0 1 1 0 0 0 0 F 0 1 1 0 0 1 1 1

Converts the 64-bit value in DAC to a single precision floating-point number. If DAC contains 0, the instruction ends. If bits 25-48 of DAC are not zeroes and bit 1 contains 1, the instruction adds 1 to the value contained in bit 24 of DAC, clears bits 25-48 to 0, normalizes the result, and places it in DAC. If any other condition exists, no rounding occurs.

If no floating-point exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ FS f,address
Floating Subtract
0 1 0 1 1 0 0 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 32-bit contents of the location specified by EA from the contents of the specified FAC. (See Chapter 6.) Normalizes the result, if necessary, and stores it in the FAC. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ FST f,address
Floating Store
0 0 1 1 1 0 0 F 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the specified FAC into the 32-bit location specified by EA. (See Chapter 6.) If the exponent contained in the FAC is too large to be expressed in 8 bits, a floating-point exception (store exception) occurs. If no exception occurs, the instruction sets CBIT to 0. At the end of the instruction, the values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information. In either case, a floating-point exception leaves the contents of the memory location in an indeterminate state.

▶ HLT
Halt
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Halts computer operation. The program counter points to the instruction that would have been executed if execution had not been stopped. The supervisor terminal indicates a halt. Leaves the values of LINK, CBIT, and the condition codes unchanged.

This instruction saves the contents of registers in a memory location specified by the RSAVPTR. The contents of RSAVPTR can be accessed by an LDLR/STLR instruction with address '40037. The registers are saved in their physical order. (See Chapter 9 for the format of these register files.) The saved register file order is shown in Table 14-3.

Table 14-3
Order of Saved Registers After HLT

9950	850	Rest of 50 Series
User Reg Set 1	ISP #1:	User Reg Set 1
User Reg Set 2	User Reg Set 1	User Reg Set 2
User Reg Set 3	User Reg Set 2	DMx Reg File
User Reg Set 4	DMx Reg File	Microcode Reg File
Microcode Reg File, Set 2	Microcode Reg File	
Indirect Reg Set	ISP #2:	
Microcode Reg File, Set 1	User Reg Set 1	
DMx Reg File	User Reg Set 2	
	DMx Reg File	
	Microcode Reg File	

Note

This is a restricted instruction.

▶ I R,address
Interchange Register and Memory Fullword
1 0 0 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Interchanges the 32-bit value contained in the specified R with the 32-bit value contained in the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register form. See Chapter 3 for more information.

▶ ICBL r
Interchange Bytes and Clear Left
0 1 1 0 0 0 R\3 0 1 1 0 1 0 1

Interchanges bits 1-8 and bits 9-16 of the specified r, then loads 0 into bits 1-8 of r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICBR r
Interchange Bytes and Clear Right
0 1 1 0 0 0 R\3 0 1 1 0 1 1 0

Interchanges bits 1-8 and bits 9-16 of the specified r, then loads 0 into bits 9-16 of r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICHL r
Interchange Halfwords and Clear Left
0 1 1 0 0 0 R\3 0 1 1 0 0 0 0

Interchanges the contents of bits 1-16 and bits 17-32 of the specified R, then loads zeroes into bits 1-16 of R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ICHR r
Interchange Halfwords and Clear Right
0 1 1 0 0 0 R\3 0 1 1 0 0 0 1

Interchanges the contents of bits 1-16 and bits 17-32 of the specified R, then loads zeroes into bits 17-32 of R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ IH r,address
Interchange r and Memory Halfword
1 0 1 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Interchanges the value contained in the specified r with the 16-bit value contained in the location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register form. See Chapter 3 for more information.

▶ IH1 r
Increment r By 1
0 1 1 0 0 0 R\3 1 0 1 0 1 1 0

Increments the contents of the specified r by 1 and stores the result in r. An overflow causes an integer exception. LINK reflects the state of the carry. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ IH2 r
Increment r By 2
0 1 1 0 0 0 R\3 1 0 1 0 1 1 1

Increments the contents of the specified r by 2 and stores the result in r. An overflow causes an integer exception to occur. LINK reflects the state of the carry. The condition codes reflect the result of the operation. (See Table 5-3.) If no integer exception occurs, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ IM address
Increment Memory Fullword
1 0 0 1 1 0 0 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Adds 1 to the 32-bit integer contained in the specified location and stores the result back in the specified location. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ IMH address
Increment Memory Halfword
1 0 1 1 1 0 0 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Adds 1 to the 16-bit integer contained in the specified location and stores the result back in the specified location. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the operation. (See Table 5-3.)

▶ INBC address
Interrupt Notify Beginning, Clear Active Interrput
0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Places the notified process at the beginning of the appropriate priority level queue. Issues a CAI pulse to clear the currently active interrupt.

A process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK, and the condition codes are indeterminate. See Chapter 11 for more information.

Note

INBC is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INBN address
Interrupt Notify Beginning
0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 1
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Places the notified process at the beginning of the appropriate priority level queue. Does not issue a CAI pulse to clear the currently active interrupt.

A process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK, and the condition codes are indeterminate. See Chapter 11 for more information.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INEC address
Interrupt Notify End, Clear Active Interrupt
0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Issues a CAI pulse to clear the currently active interrupt. Places the notified process at the end of the appropriate priority level queue.

A process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK and the condition codes are indeterminate. See Chapter 11 for more information.

Note

INEC is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INEN address
Interrupt Notify End
0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0
AP\32

Notifies a semaphore at the specified address during phantom interrupt code. Restores the state of the interrupted process by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from microcode temporary registers PSWPB and PSWKEYS. Does not issue a CAI pulse to clear the currently active interrupt. Places the notified process at the end of the appropriate priority level queue.

A process exchange will occur if the notified process is of a higher priority than the interrupted process. The values of CBIT, LINK and the condition codes are indeterminate. See Chapter 11 for more information.

Note

This is a restricted instruction.

This instruction is normally used to transfer from phantom interrupt code to an interrupt process.

▶ INH
Inhibit Interrupts
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1

Inhibits interrupts by setting bit 1 of the modals to 0. The processor ignores any interrupt requests that are made over the I/O bus. This instruction takes effect immediately. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHL
Inhibit Interrupts (Local)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1

This 850 instruction performs the same actions as INH does. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHM
Inhibit Interrupts (Mutual)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

For the 850, a processor checks the availability of the mutual exclusion lock. If available, the processor sets this lock and inhibits interrupts. Otherwise, it waits for the lock to be released by the other processor and then sets the lock and inhibits interrupts. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INHP
Inhibit Interrupts (Process)
0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0

For the 850, a processor checks the availability of the process exchange lock. If available, the processor sets it and inhibits interrupts. Otherwise, it waits for the lock to be released by the other processor, and then sets the lock and inhibits interrupts. It also inhibits interrupts in the local processor. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ INK r
Input Keys
0 1 1 0 0 0 R\3 0 1 1 1 0 0 0

Loads the contents of the R-mode keys into the specified r. Leaves the values of LINK, CBIT, and the condition codes unchanged. Reads the low-order 8 bits of the S register along with the high-order 8 bits of the keys register.

▶ INT f,R
Convert Floating Point to Integer
0 1 1 0 0 0 R\3 1 0 0 F 0 1 1

Converts the double precision floating-point number contained in the specified floating accumulator to a 32-bit integer and stores the result in R. Ignores the fractional part of the floating-point number. Overflow occurs if the value in the floating accumulator is less than -2**31 or greater than (2**31)-1. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

▶ INTH f,r
Convert Floating Point Number to Halfword Integer
0 1 1 0 0 0 R\3 1 0 0 F 0 0 1

Converts the double precision floating-point number contained in the specified floating accumulator to an integer and stores the result in r. Overflow occurs if the value in the floating accumulator is less than -2**15 or greater than (2**15)-1. An overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0.

At the end of this instruction, the contents of R bits 17-32 are indeterminate. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 in the keys contains 1, the instruction sets CBIT to 1. If bit 7 contains 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. For more information, see Chapter 11.

▶ IR1 R
Increment Register By 1
0 1 1 0 0 0 R\3 1 0 1 0 0 1 0

Increments the contents of the specified R by 1 and stores the result in R. An overflow causes an integer exception fault. If no integer exception occurs, CBIT is reset to 0. LINK contains the carry-out bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 in the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. (See Chapter 11.)

▶ IR2 R
Increment Register By 2
0 1 1 0 0 0 R\3 1 0 1 0 0 1 1

Increments the contents of the specified R by 2 and stores the result in R. An overflow causes an integer exception fault. If no integer exception occurs, CBIT is reset to 0. LINK contains the carry-out bit. The condition code contain the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 in the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. For more information, see Chapter 11.

▶ IRB r
Interchange r Bytes
0 1 1 0 0 0 R\3 0 1 1 0 1 0 0

Interchanges bits 1-8 and bits 9-16 of the specified r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ IRH R
Interchange Register Halves
0 1 1 0 0 0 R\3 0 1 0 1 1 1 1

Interchanges the contents of bits 1-16 and bits 17-32 of the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ IRTC
Interrupt Return, Clear Active Interrupt
0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1

Returns from an interrupt. Issues a CAI pulse to clear the currently active interrupt. Restores the state existing before the interrupt by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from the values saved in microcode temporary registers PSWPB and PSWKEYS.

Note

This is a restricted instruction.

▶ IRTN
Interrupt Return
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1

Returns from an interrupt. Does not issue a CAI pulse to clear the currently active interrupt. Restores the state existing before the interrupt by loading bits 1-16 of PB, bits 17-32 of the program counter, and the keys from the values saved in microcode temporary registers PSWPB and PSWKEYS.

Note

This is a restricted instruction.

▶ ITLB
Invalidate STLB Entry
0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1

Invalidates the STLB entry that corresponds to the virtual address contained in GR2. The values of CBIT, LINK, and the condition codes are indeterminate. You must execute this instruction whenever you change the page table entry for the given address.

If you change a SDW or DTAR (explained in Chapter 4), you usually have to invalidate the entire STLB by issuing the instruction PTLB. A 0 in the segment number portion of GR2 invalidates the IOTLB entry corresponding to the address specified by GR2.

Note

This is a restricted instruction.

▶ JMP address
Jump
1 0 1 1 1 0 0 0 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA, and loads it into the program counter. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ JSR R,address
Jump to Subroutine
1 1 1 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Saves the 16-bit word number position of the return address in the specified R. Loads the program counter with the current segment location specified by bits 17-32 of the EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction is useful for calling routines within the current segment only.

▶ JSXB address
Jump and Set XB
1 1 0 1 1 0 0 0 1 TM\2 SR\3 BR\2

Calculates an effective address, EA. Loads the contents of the program counter into XB. Loads EA into the program counter. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This instruction can make subroutine calls outside the current segment as well as within.

▶ L R,address
Load Full Word
0 0 0 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads EA into the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ LCEQ R
Load Register on EQ
0 1 1 0 0 0 R\3 1 1 0 1 0 1 1

If the condition codes reflect an equal to 0 condition, the instruction loads the specified R with a 1. If the condition codes reflect a not equal to 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCGE R
Load Register on GE
0 1 1 0 0 0 R\3 1 1 0 1 1 0 0

If the condition codes reflect a greater than or equal to 0 condition, the instruction loads the specified R with a 1. If they reflect a less than 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCGT R
Load Register on GT
0 1 1 0 0 0 R\3 1 1 0 1 1 0 1

If the condition codes reflect a greater than 0 condition, the instruction loads the specified R with a 1. If they reflect a less than or equal to 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCLE R
Load Register on LE
0 1 1 0 0 0 R\3 1 1 0 1 0 0 1

If the condition codes reflect a less than or equal to 0 condition, the instruction loads the specified R with a 1. If they reflect a greater than 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCLT R
Load Register on LT
0 1 1 0 0 0 R\3 1 1 0 1 0 0 0

If the condition codes reflect a less than 0 condition, the instruction loads the specified R with a 1. If they reflect a greater than or equal to 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LCNE R
Load Register on NE
0 1 1 0 0 0 R\3 1 1 0 1 0 1 0

If the condition codes reflect a not equal to 0 condition, the instruction loads the specified R with a 1. If they reflect an equal to 0 condition, the instruction loads R with a 0. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ LDAR R,address
Load Addressed Register
1 0 0 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates a doubleword effective address, EA. Loads the specified R with the contents of the register file location specified by the word portion of EA. Bit 2 and bit 12 of the word portion of EA determine the actions of this instruction.

Bit 2	Bit 12	Action
1*	——	Ignore bits 1 and 3-9. The word portion of EA specifies an absolute register number from 0-'377.
0*	1	Bits 13-16 of the word portion of EA specify one of the registers '20-'37 in the current register set.
0	0	Bits 13-16 of the word portion of EA specify one of the registers 0-'17 in the current register set.

*This is a restricted instruction.

Leaves the values of CBIT and LINK unchanged; the values of the condition codes are indeterminate. See Chapter 9 for more information about register sets.

Note

If the current ring is not 0 and EA is outside the range of 0-17, inclusive, any access causes an RXM violation.

▶ LDC flr,r
Load Character
0 1 1 0 0 0 R\3 1 1 1 FLR 0 1 0

If the contents of the specified FLR are nonzero, the instruction loads the single character pointed to by the specified FAR into bits 9-16 of r and loads zeroes into bits 1-8. Updates the contents of the FAR by 8 (one byte) so that they point to the next character. Decrements the contents of the specified FLR by 1. Sets the condition codes to NE. Leaves the values of CBIT and LINK unchanged.

If the contents of the specified FLR are 0, the instruction sets the condition codes to EQ.

Note

This instruction uses FAR0 when FLR0 is specified, and FAR1 when FLR1 is specified.

▶ LEQ R
Load Register on Equal to 0
0 1 1 0 0 0 R\3 0 0 0 0 0 1 1

If the contents of the specified R are equal to 0, the instruction loads R with a 1. If not equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LF R
Logic Set False
0 1 1 0 0 0 R\3 0 1 0 1 1 0 0

Loads the specified R with 0. Leaves the values of LINK and CBIT unchanged. Sets the condition codes to the outcome of the comparison. (See Table 5-3.)

▶ LFEQ f,R
Load Register on Floating Accumulator Equal to 0
0 1 1 0 0 0 R\3 0 0 1 F 0 0 1

If the contents of the specified floating accumulator are equal to 0, the instruction loads the specified R with a 1; if not equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFGE f,R
Load Register on Floating Accumulator Greater Than or Equal to 0
0 1 1 0 0 0 R\3 0 0 1 F 1 0 0

If the contents of the specified floating accumulator are greater than or equal to 0, the instruction loads the specified R with a 1; if less than 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFGT f,R
Load Register on Floating Accumulator Greater Than 0
0 1 1 0 0 0 R\3 0 0 1 F 1 0 1

If the contents of the specified floating accumulator are greater than 0, the instruction loads the specified R with a 1; if less than or equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFLE f,R
Load Register on Floating Accumulator Less Than or Equal to 0
0 1 1 0 0 0 R\3 0 0 1 F 0 0 1

If the contents of the specified floating accumulator are less than or equal to 0, the instruction loads the specified R with a 1; if greater than 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFLI flr,data
Load FLR Immediate
0 0 0 0 0 0 1 0 1 1 0 0 F 0 0 1
INTEGER\16

Loads the 16-bit, unsigned integer contained in the second word of the instruction into the specified FLR. Clears the upper bits of the FLR. Leaves the values of CBIT, LINK, the condition codes, and the associated FAR unchanged.

▶ LFLT f,R
Load Register on Floating Accumulator Less Than 0
0 1 1 0 0 0 R\3 0 0 1 F 0 0 0

If the contents of the specified floating accumulator are less than 0, the instruction loads the specified R with a 1; if greater than or equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LFNE f,R
Load Register on Floating Accumulator Not Equal to 0
0 1 1 0 0 0 R\3 0 0 1 F 0 1 0

If the contents of the specified floating accumulator are not equal to 0, the instruction loads the specified R with a 1; if equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LGE R
Load Register on Greater Than or Equal to 0
0 1 1 0 0 0 R\3 0 0 0 0 1 0 0

If the contents of the specified R are greater than or equal to 0, the instruction loads R with a 1; if less than 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LGT R
Load Register on Greater Than 0
0 1 1 0 0 0 R\3 0 0 0 0 1 0 1

If the contents of the specified R are greater than 0, the instruction loads R with a 1; if less than or equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LH r,address
Load Halfword
0 0 1 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads the 16-bit contents contained in the location specified by EA into r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ LHEQ r
Load r on EQ
0 1 1 0 0 0 R\3 0 0 0 1 0 1 1

If the contents of the specified r are equal to 0, the instruction loads r with a 1; if not equal to 1, the instruction loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LHGE r
Load r on GE
0 1 1 0 0 0 R\3 0 0 0 0 1 0 0

If the contents of the specified r are greater than or equal to 0, the instruction loads r with a 1; if less than 0, the instruction loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LHGT r
Load r on GT
0 1 1 0 0 0 R\3 0 0 0 1 1 0 1

If the contents of the specified r are greater than 0, the instruction loads r with a 1; if less than or equal to 0, the instruction loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LHL1 r,address
Load Halfword Shifted Left by 1
0 0 0 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Shifts the contents of the location specified by EA left one bit and stores the result in the specified r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

LHL2 also has a register-to-register form. See Chapter 3 for more information.

▶ LHL2 r,address
Load Halfword Shifted Left by 2
0 0 1 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Shifts the 16-bit contents of the location specified by EA left two bits and stores the result in the specified r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

LHL2 also has a register-to-register form. See Chapter 3 for more information.

▶ LHL3 r,address
Load Halfword Shifted Left by 3
0 1 1 1 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Shifts the 16-bit contents of the location specified by EA left three bits and stores the result in the specified r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register form. See Chapter 3 for more information.

▶ LHLE r
Load r on LE
0 1 1 0 0 0 R\3 0 0 0 1 0 0 1

If the contents of the specified r are less than or equal to 0, the instruction loads r with a 1; if greater than 0, the instruction loads r with 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LHLT r
Load r on LT
0 1 1 0 0 0 R\3 0 0 0 0 0 0 0

If the contents of the specified r are less than 0, the instruction loads r with a 1; if greater than or equal to 0, loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LHNE r
Load r on NE
0 1 1 0 0 0 R\3 0 0 0 1 0 1 0

If the contents of the specified r are not equal to 0, the instruction loads r with a 1; if equal to 0, the instruction loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LIOT address
Load I/O TLB
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
AP\32

Loads a specified IOTLB entry. Table 14-4 shows the contents of the LIOT entry and the origin of the information. The values of CBIT, LINK, and the condition codes are indeterminate.

Table 14-4
LIOT Data

Origin	Description
AP in LIOT	Virtual address in segment 0 (calculated from the EA).
Page table	Physical address (translation of the virtual address) obtained from segment 0. If the fault bit is set to 1, a page fault occurs.
L register	Target virtual address. This is the segment number and page number of the virtual address that will be used by procedures accessing this information. This is used to help invalidate the proper locations in the cache. This is provided in L as a virtual address. the low-order 10 bits (word number in the page) and the segment number are ignored.

Note

This is a restricted instruction.

▶ LLE R
Load Register on Less Than or Equal to 0
0 1 1 0 0 0 R\3 0 0 0 0 0 0 1

If the contents of the specified R are less than or equal to 0, the instruction loads R with a 1; if greater than 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LLT R
Load Register on Less Than 0
0 1 1 0 0 0 R\3 0 0 0 0 0 0 0

If the contents of the specified R are less than 0, the instruction loads R with a 1; if greater than or equal to 0, the instruction loads R with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LMCM
Leave Machine Check Mode
0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1

Leaves machine check mode by resetting bits 15-16 of the modals to 00. If a machine parity error occurs in this mode, the hardware sets the machine check flag but no check (V mode) or interrupt (S, R modes) occurs. Inhibits the machine for one instruction. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ LNE R
Load Register on Not Equal to 0
0 1 1 0 0 0 R\3 0 0 0 0 0 1 0

If the contents of the specified R are not equal to 0, the instruction loads r with a 1; if equal to 0, the instruction loads r with a 0. Leaves the values of LINK and CBIT unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

▶ LPID
Load Process ID
0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1

Loads the process ID from bits 1-12 of A into RPID (the process ID register, which contains the 10 most significant bits of the user's address space). Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ LPSW address
Load PSW
0 0 0 0 0 0 0 1 1 1 0 0 1 0 0 1

Changes the status of the processor by loading new values into the program counter, keys, and modals. Inhibits interrupts for one instruction.

Addresses a four-word block at the specified location. The block has the format:

Word in Block	Contents
1-2	New program counter (ring, segment, word numbers)
3	New keys
4	New modals

Loads the program counter and keys of the currently running process with the contents of the first three words, then loads the processor modals with the contents of the fourth word.

The new value of bit 15 in the keys, the in-dispatch bit, can temporarily halt execution of the current process. This bit is altered by software only during a cold or a warm start. If bit 15 is 0, the currently executing process will continue to execute, but at a location defined by the new value of the program counter. If bit 15 is 1, the processor enters the dispatcher and dispatches the ready process with the highest priority. When execution resumes for the process that was temporarily halted, execution resumes at the point defined by the value of the new program counter.

Regardless of the value of bit 15, the new value of the modals takes effect immediately, since the modals are associated with the processor, not the process.

The LPSW instruction loads the four words of the register set that the STLR instruction cannot correctly load. STLR does not update the separate hardware registers the processor uses to maintain duplicate information for optimization.

Never use this instruction to change bits 9-11 of the modals. These bits specify the current user register set. This means that if you do not know the current value of these bits, you must do the following each time you want to execute an LPSW:

1. Inhibit interrupts.
2. Read the current values of modal bits 9-11 with an LDLR '24 instruction.
3. Mask the old values of the modal bits into the new information.
4. Load the new information into the modals with an LPSW.

For the two common uses of LPSW, you do not have to perform this sequence, since the values of modal bits 9-11 are predictable. When you use LPSW after a Master Clear to turn on processor exchange mode, bits 9-11 are 010 because the processor is always initialized to register set 2. When you use LPSW to return from a fault, check, or interrupt, simply reload the values stored by the break because these values are still correct.

You should not use LPSW to set bits 16 (the save-done bit) or 15 (the in-dispatcher bit) of the keys, unless you are merely loading status following a fault, check, or interrupt. When issuing LPSW after a Master Clear, make sure you load zeroes into both of these bits.

Note

This is a restricted instruction.

▶ LT R
Logic Set True
0 1 1 0 0 0 R\3 0 0 0 1 1 1 1

Loads the specified R with 1. Leaves the values of LINK and CBIT unchanged. Sets the condition codes to the outcome of the operation. (See Table 5-3.)

▶ M R,address
Multiply Fullword
1 0 0 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the 32-bit value contained in the location specified by EA by the 32-bit value contained in the specified R. Stores the 64-bit result in the specified R and R+1. The least significant bit of the result is contained in bit 32 of R+1. Resets CBIT to 0. The values of LINK and the condition codes are indeterminate.

Note

R must be an even numbered register.

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ MDEI
Memory Diagnostic Enable Interleave
0 0 0 0 0 0 1 0 1 1 0 0 0 1 0 0

Enables the memory interleave facility. Leaves the values of LINK, CBIT, and the condition codes unchanged. This instruction is not implemented on the 9950.

Note

This is a restricted instruction.

▶ MDII
Memory Diagnostic Inhibit Interleave
0 0 0 0 0 0 1 0 1 1 0 0 0 1 0 1

Inhibits the memory interleave facility. Leaves the values of LINK, CBIT, and the condition codes unchanged. This instruction is not implemented on 9950.

Note

This is a restricted instruction.

▶ MDIW
Memory Diagnostic Write Interleaved
0 0 0 0 0 0 1 0 1 1 0 1 0 1 0 0

Writes interleaved memory. Leaves the values of LINK, CBIT, and the condition codes unchanged. This instruction is not implemented on the 9950.

Note

This is a restricted instruction.

▶ MDRS
Memory Diagnostic Read Syndrome Bits
0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 0

Reads memory syndrome bits. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ MDWC
Memory Diagnostic Load Write Control Register
0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1

Writes memory control register. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This is a restricted instruction.

▶ MH r,address
Multiply Halfword
1 0 1 0 1 0 R\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the 16-bit value contained in the location specified by EA by the 16-bit value contained in the specified r. Stores the 32-bit result in R. Bit 32 of R contains the least significant bit of the result. Resets CBIT to 0. The value of LINK is indeterminate. The condition codes reflect the result of the operation. (See Table 5-3.)

Note

MH r also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ MIA
Microcode Execute A
1 1 0 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

This instruction currently causes a UII fault. If implemented, this instruction is for user-written microcode. For more information about UII, refer to Chapter 11.

▶ MIB
Microcode Execute B
1 1 1 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

This instruction currently causes a UII fault. If implemented, this instruction is for user-written microcode. For more information about UII, refer to Chapter 11.

▶ N R,address
AND Fullword
0 0 0 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Logically ANDs the value contained in the specified R with the 32-bit value contained in the location specified by EA. Stores the result in the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ NFYB
Notify
0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 1
AP\32

Notifies on semaphore at address specified in second and third words of the instruction. Uses LIFO queueing. Does not clear the currently active interrupt. The values of LINK, CBIT, and the condition codes are indeterminate.

Note

This is a restricted instruction.

▶ NFYE
Notify
0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0
AP\32

Notifies on semaphore at address specified in second and third words of the instruction. Uses FIFO queueing. Does not clear the currently active interrupt. The values of LINK, CBIT, and the condition codes are indeterminate.

Note

This is a restricted instruction.

▶ NH r,address
AND Halfword
0 0 1 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Logically ANDs the value contained in the specified r with the 16-bit value contained in the location specified by EA. Stores the result in r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ NOP
No Operation
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Does nothing. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ O R,address
OR Fullword
0 1 0 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Logically ORs the value contained in the specified R with the 32-bit value contained in the location specified by EA. Stores the result in the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ OH r,address
OR Halfword
0 1 1 0 1 1 R\3 TM\2 SR\2 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Logically ORs the value contained in the specified r with the 16-bit value contained in the location specified by EA. Stores the result in r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ OTK r
Output Keys
0 1 1 0 0 0 R\3 0 1 1 1 0 0 1

Stores the contents of the specified r in the keys. Resets bits 15-16 of the keys to 0. Loads CBIT, LINK, and the condition codes from the specified r as a result of the operation.

▶ PCL
Procedure Call
1 0 0 1 1 0 0 0 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

See Chapter 8 for a complete description of this instruction. Sets LINK, CBIT, and the condition codes to the values contained in the ECB.

Note

When arguments are to be transferred to the called procedure, this instruction uses GR5, GR7, and XB, destroying the previous contents of these registers. The contents of GR5, GR7, and XB remain unchanged if no arguments are transferred. The contents of the condition codes, CBIT, and LINK are not correctly saved in the ECB, along with the rest of the caller's keys.

▶ PID R
Position for Integer Divide
0 1 1 0 0 0 R\3 0 1 0 1 0 1 0

Positions a register for integer divide. Loads the contents of the specified R into R+1. Extends the sign of R (bit 1) into bits 2-32 of R. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

R must be a even numbered register.

▶ PIDH r
Position r for Integer Divide
0 1 1 0 0 0 R\3 0 1 0 1 0 1 1

Moves the contents of the specified r (bits 1-16 of R) into bits 17-32 of R. Extends the contents of bit 1 of r into bits 2-16 of R. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ PIM R
Position After Multiply
0 1 1 0 0 0 R\3 0 1 0 1 0 0 0

Checks bit 1 of R+1 to see if it is the same as all the bits in the specified R, and then moves the contents of R+1 into R. If bit 1 of R+1 was not the same as all the bits in R, an overflow occurs which causes an integer exception.

If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 in the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains 1, the instruction sets CBIT to 1 and causes an integer exception fault. For more information, see Chapter 11.

Note

R must be an even numbered register.

▶ PIMH r
Position r after Multiply
1 0 1 1 0 0 0 R\3 0 1 0 1 0 0 1

Checks the contents of bit 17 of the specified R to see if it has the same value as do all of bits 1-16 of R, and then moves the contents of bits 17-32 into bits 1-16. If bit 17 was different from all of bits 1-16, an integer exception occurs. If no integer exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ PRTN
Procedure Return
0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 1

Deallocates the stack frame created for the executing procedure and returns to the environment of the procedure that called it.

To deallocate the frame, the instruction stores the current value of the stack base register into the free pointer. It then restores the caller's state by loading the caller's program counter, stack base register, linkage base register, and keys with the values contained in the frame being deallocated. Sets bits 15-16 of the keys to 0.

Loads the ring number in the program counter with the logical OR (weaker) of the saved program counter ring and the current ring number. This process prevents inward returns but also allows returns from gated calls to work properly.

▶ PTLB
Purge TLB
0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0

GR2 contains the address of a physical page, right justified. Based on the value of GR2 bit 1, PTLB purges either the first 128 locations of the STLB (i.e., not the IOTLB), or a specified physical page. If GR2 bit 1 contains a 1, the instruction performs a complete purge. If GR2 bit 1 contains a 0, the instruction purges the page specified by GR2. Leaves the values of CBIT, LINK, and the condition codes indeterminate. See Chapters 1, 3, and 12 for more information about the STLB and IOTLB.

Note

This is a restricted instruction.

On the 750, 850, or 9950,, insert a CRE (Clear E) instruction before PTLB. Since PTLB uses E (GR3 in I mode) as a pointer, the CRE zeroes GR3 before PTLB manipulates it. If an interrupt occurs during PTLB's execution, GR3 points to the location PTLB is currently purging. PTLB leaves the contents of GR3 in an undefined state at the end of its execution.

▶ QFAD address
Quad Precision Floating Add
0 1 1 1 1 0 1 1 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Adds the 112-bit, quad precision number contained in the locations specified by EA to the contents of QAC. (See Chapter 6.) Normalizes the result, if necessary, and loads it into QAC. An overflow or underflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFCM
Quad Precision Floating Complement
1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 0

Forms the two's complement of the value contained in QAC. (See Chapter 6.) Normalizes the result, if necessary, and stores it in QAC. An underflow or overflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFC address
Quad Precision Floating Point Compare and Skip
1 0 0 1 1 0 1 1 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Compares the contents of QAC (explained in Chapter 6) to the 112-bit contents of the location specified by EA and skips according to the list below.

Condition	Skip
QAC > contents of loc spec by EA.	No skip.
QAC = contents of loc spec by EA.	Skip one word.
QAC < contents of loc spec by EA.	Skip two words.

Sets the condition codes to reflect the state of the comparison. (See Table 5-3.) The values of CBIT and LINK are indeterminate.

Note

Be sure to use normalized numbers for correct results.

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFDV address
Quad Precision Floating Point Divide
1 0 0 1 1 0 1 1 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Divides the contents of QAC by the 112-bit contents of the location specified by EA. (See Chapter 6.) Normalizes the result, if necessary, and stores the whole quotient into QAC. An overflow, underflow, or divide by to causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If QFDV is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFLD address
Quad Precision Floating Load
0 1 1 1 1 0 1 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an extended, augmented effective address, EA. Performs one of the following actions with the value contained in the location specified by EA. Loads bits 1-112 into QAC and zeros QAC bits 113-128, or loads 128 bits into QAC. (See Chapter 6 for more information.) Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFMP address
Quad Precision Floating Point Multiply
1 0 0 1 1 0 1 0 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Multiplies the contents of QAC by the 112-bit contents of the location specified by EA. (See Chapter 6.) Normalizes the result, if necessary, and stores it into QAC. An overflow or underflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFSB address
Quad Precision Floating Point Subtract
0 1 1 1 1 0 1 1 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 112-bit contents of the locations specified by EA from the contents of QAC. (See Chapter 6.) Normalizes the result, if necessary, and loads it into QAC. An overflow or underflow causes a floating-point exception. If no floating-point exception occurs, CBIT is reset to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QFST address
Quad Precision Floating Store
0 1 1 1 1 0 1 0 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of QAC into the 128 bits of memory specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction does not normalize the result before storing it into the specified memory location.

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QINQ
Quad to Integer, in Quad Convert
1 1 0 0 0 0 0 1 0 1 1 1 1 0 1 0

Strips the fractional portion of QAC as described in Table 14-5.

Table 14-5
QINQ Actions

Exponent Value	Action
'340 <= Exp	A conversion fault occurs.
'200 < Exp < '340	If sign >= 0, strip fractional part of QAC for result. If sign < 0 and fractional part = 0, strip fractional part of QAC and increment result by 1. If sign < 0 and fractional part <> 0, strip fractional part for result.
'200 = Exp	If sign >= 0, result = 0. If sign < 0 and bits 2-96 = 0 result = -1. If sign < 0 and bits 2-96 <> 0 result = 0.
'200 > Exp	Result = 0.

QINQ can cause a floating-point exception. This exception does not alter the contents of QAC. If no exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ QIQR
Quad to Integer, in Quad Convert Rounded
1 1 0 0 0 0 0 1 0 1 1 1 1 0 1 1

Strips the fractional portion of QAC as described in Table 14-6.

Table 14-6
QIQR Actions

Exponent Value	Action
'340 < Exp	A conversion fault occurs.
Exp = '340	Set the most significant bit of the fractional part of QAC to 0.
'200 < Exp < '340	If sign >= 0, strip fractional part of QAC for result. If sign < 0 and fractional part <>0, strip fractional part of QAC for result. If sign <> 0 and fractional part = 0, strip the fractional part and increment the integer part by 1. In any case, increment the integer part by 1 if it exists and the most significant bit of the fractional part of QAC is 1.
Exp = '200	If sign >= 0, result = 0. If sign < 0 and bits 2-96 = 0 result = -1. If sign < 0 and bits 2-96 <> 0 result = 0. For all cases increment integer part by 1 if it exists and the most significant bit of QAC = 1.
Exp < '200	The result is 0.

QIQR can cause a floating-point exception. This exception does not alter the contents of QAC. If no exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

If a floating-point exception occurs and bit 7 of the keys contains a 1, the instruction sets CBIT to 1. If bit 7 contains a 0, the instruction sets CBIT to 1 and causes a floating-point exception fault. See Chapter 11 for more information.

Note

If this instruction is used for any system but the 9950, an unimplemented instruction (UII) fault occurs. (See Chapter 11.)

▶ RBQ address
Remove Entry from Bottom of Queue
0 1 1 0 0 0 R\3 1 0 1 1 0 1 1
AP\32

The address pointer in this instruction points to the QCB for a queue. The instruction removes the entry from the bottom of the referenced queue and loads it into the specified R. If the queue was not empty, this instruction sets the condition codes to reflect not equal to to. If the queue was empty, resets R to 0 and sets the condition codes to reflect equal to to. Leaves the values of CBIT and LINK unchanged.

▶ RCB
Reset CBIT Bit to 0
1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0

Resets CBIT to 0. Leaves the values of LINK and the condition codes unchanged.

▶ RMC
Reset Machine Check Flag to 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0

Resets the machine check mode (bits 15-16 of the modals) to 0. Leaves the values of LINK, CBIT, and the condition codes unchanged. Inhibits interrupts for the next instruction.

Note

This is a restricted instruction.

▶ ROT address
Rotating Shift
0 1 0 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Interprets bits 1-16 of EA as a shift command, as shown in Table 14-7.

Table 14-7
EA Format for ROT Shift Command

Bit	Value	Interpretation
1	0	Shift left.
	1	Shift right.
2	0	Word shift (32 bits).
	1	Halfword shift (16 bits).
3-10	—	Ignored.
11-16	—	Values specify the two's complement of the number of bits to shift. A value of 0 indicates a shift of 64 places; of -1, 1 place; of -63, 63 places; and so on.

Uses EA to perform a rotating shift on the contents of the specified R. Stores the shifted result in R. CBIT contains the value of the last bit shifted out. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ RRST address
Restore Registers
0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 1
AP\32

Calculates an effective address, EA, from the 32-bit address pointer in the instruction. This specifies the starting address of a save area for the general, floating, and XB registers. Restores the contents of these registers from this save area.

The save area format is shown in Table 14-8. Bits 1-16 of the save area are a save mask, whose format appears in Figure 14-3. A mask bit value of 1 means that the corresponding register had nonzero contents that have been saved in the save area; a mask bit value of 0 means that the corresponding register's contents were 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

Table 14-8
RRST and RSAV Save Area Format

Word #	Contents
1	Save mask
2-5	FR1
6-9	FR0
10-11	GR7, X
12-13	GR6
14-15	GR5, Y, S
16-17	GR4
18-19	GR3, E
20-21	GR2, A, B, L
22-23	GR1
24-25	GR0
26-27	XB

1 4	5	6 7	8	9 10	11	12	13	14	15	16
0000	FR1	FR0	GR7	GR6	GR5	GR4	GR3	GR2	GR1	GR0

Save Mask Format, RRST and RSAV Instructions
Figure 14-3

▶ RSAV
Save Registers
0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 1
AP\32

Calculates an effective address, EA, from the 32-bit address pointer in the instruciton. This specifies the starting address of a save area for the general, floating, and XB registers. Saves the nonzero contents of these registers in the save area.

The save area format is shown in Table 14-8. Bits 1-16 of the save area are a save mask, whose format appears in Figure 14-3. This instruction sets the mask bit of each register as follows: to 1 if the register's contents have a nonzero value; to 0 if a 0 value. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ RTQ address
Remove Entry from Top of Queue
0 1 1 0 0 0 R\3 1 0 1 1 0 1 0
AP\32

The address pointer in this instruction is to the QCB for a queue. The instruction removes the entry from the top of the referenced queue, and loads it into the specified R. If the queue was not empty, the instruction sets the condition codes to reflect not equal to 0. If the queue was empty, resets R to 0 and sets the condition codes to reflect equal to to. Leaves the values of CBIT and LINK unchanged.

▶ RTS
Reset Time Slice
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1

Valid for the 550-II, 750, 850, I450, and new processors.

GR2 contains a negative value representing the number of milliseconds in the new time slice.

Adds the current value of the interval timer (locations 16-17 of the PCB) to the contents of the elapsed timer (locations 10-11 of the PCB), then subtracts the contents of GR2 from the sum of the timers. Stores the result in the elapsed timer. Loads the contents of GR2 into the interval timer. Leaves the contents of GR2 unchanged. The values of CBIT, LINK, and the condition codes are unchanged.

The addition performed by this instruction is equivalent to the following series of instructions:

```
LH    0,ITH /* Load GR0 with contents of ITH.
SH    0,RV  /* Subtract reset value (in RV)
            /*   from contents of GR0.
PIDH  0     /* Sign extend the contents of
            /*   GR0 into bits 17-32 of GR0.
SRC         /* Skip next word if CBIT is 0.
CMR   0     /* Complement GR0.
LH    1,ET  /* Load GR1 with contents of ET.
A     0,1   /* Add ITH and ET.
ST    0,ET  /* Store result in ET.
LH    0,RV  /* Load reset value into GR0.
SH    0,ITH /* Store GR0 contents in ITH.
```

Note

This is a restricted instruction.

▶ S R,address
Subtract Fullword
0 1 0 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 32-bit value contained in the location specified by EA from the value contained in the specified R. Stores the result in the specified R. If overflow occurs, an integer exception results. If no integer exception occurs, CBIT is reset to 0. LINK contains the borrow bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ SCB
Set CBIT Bit to 1
1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0

Sets the value of CBIT to 1. Leaves the values of the condition codes unchanged. The value of LINK is indeterminate.

▶ SH r,address
Subtract Halfword
0 1 1 0 1 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Subtracts the 16-bit value contained in the location specified by EA from the value contained in the specified r and stores the result in R. An overflow causes an integer exception. If no integer exception occurs, CBIT is reset to 0. LINK contains the borrow bit. The condition codes reflect the result of the operation. (See Table 5-3.)

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

Note

The SH instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ SHA R,address
Arithmetic Shift
0 0 1 1 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Interprets bits 1-16 of EA as a shift command, as shown in Table 14-9.

Table 14-9
EA Format for SHA Shift Command

Bit	Value	Interpretation
1	0	Shift left.
	1	Shift right.
2	0	Word shift (32 bits).
	1	Halfword shift (16 bits).
3-10	—	Ignored.
11-16	—	Values specify the two's complement of the number of bits to shift. A value of 0 indicates a shift of 64 places; of -1, 1 place; of -63, 63 places; and so on.

Uses EA to perform an arithmetic shift on the contents of the specified R, and stores the result of the shift in R.

For a right shift, CBIT contains the value of the last bit shifted out. The values of all other shifted-out bits are lost.

For a left shift, an overflow causes an integer exception. If there is no integer exception, CBIT is reset to 0.

All shifts leave the value of LINK indeterminate and the values of the condition codes unchanged.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ SHL R,address
Logical Shift
0 0 0 1 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Interprets bits 1-16 of EA as a shift command, as shown in Table 14-10.

Table 14-10
EA Format for SHL Shift Command

Bit	Value	Interpretation
1	0	Shift left.
	1	Shift right.
2	0	Word shift (32 bits).
	1	Halfword shift (16 bits).
3-10	—	Ignored.
11-16	—	Values specify the two's complement of the number of bits to shift. A value of 0 indicates a shift of 64 places; of -1, 1 place; of -63, 63 places; and so on.

Uses EA to perform a logical shift on the contents of the specified R. Stores the shifted result in R. CBIT contains the value of the last bit shifted out. The values of all other shifted-out bits are lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SHL1 r
Shift r Left 1
0 1 1 0 0 0 R\3 0 1 1 1 1 1 0

Shifts the contents of the specified r to the left one bit and stores the result in r. CBIT contains the value of the bit shifted out. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SHL2 r
Shift r Left 2
0 1 1 0 0 0 R\3 0 1 1 1 1 1 1

Shifts the contents of the specified r to the left two bits and stores the result in r. CBIT contains the value of the last bit shifted out. The value of the first bit shifted out is lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SHR1 r
Shift r Right 1
0 1 1 0 0 0 R\3 1 0 1 0 0 0 0

Shifts the contents of the specified r to the right one bit and stores the result in r. CBIT contains the value of the bit shifted out. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SHR2 r
Shift r Right 2
0 1 1 0 0 0 R\3 1 0 1 0 0 0 1

Shifts the contents of the specified r to the right two bits and stores the result in r. CBIT contains the value of the last bit shifted out. The value of the first bit shifted out is lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SL1 R
Shift Register Left 1
0 1 1 0 0 0 R\3 0 1 1 1 0 1 0

Shifts the contents of the specified R to the left one bit and stores the result in R. CBIT contains the value of the bit shifted out. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SL2 R
Shift Register Left 2
0 1 1 0 0 0 R\3 0 1 1 1 0 1 1

Shifts the contents of the specified R to the left two bits and stores the result in R. CBIT contains the value of the last bit shifted out; the value of the first bit shifted out is lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SR1 R
Shift Register Right 1
0 1 1 0 0 0 R\3 0 1 1 1 1 0 0

Shifts the contents of the specified R to the right one bit and stores the result in R. CBIT contains the value of the bit shifted out. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SR2 R
Shift Register Right 2
0 1 1 0 0 0 R\3 0 1 1 1 1 0 1

Shifts the contents of the specified R to the right two bits and stores the result in R. CBIT contains the value of the last bit shifted out; the value of the first bit shifted out is lost. The value of LINK is indeterminate. Leaves the values of the condition codes unchanged.

▶ SSM R
Set Sign Minus
0 1 1 0 0 0 R\3 0 1 0 0 0 1 0

Sets bit 1 of the specified R to 1. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ SSP R
Set Sign Plus
0 1 1 0 0 0 R\3 0 1 0 0 0 1 1

Sets bit 1 of the specified R to 0. Leaves the values of CBIT, LINK, and the condition codes unchanged.

▶ ST R,address
Store Fullword
0 1 0 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the specified R into the location specified by EA.

▶ STAR R,address
Store Addressed Register
1 0 1 1 0 0 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates a doubleword effective address, EA. Stores the contents of the specified R into the register location specified by the word portion of EA. Bit 2 and bit 12 of the word portion of EA determine the actions of this instruction, as shown in Table 14-11.

Table 14-11
STAR Actions

Bit 2	Bit 12	Action
1*	——	Ignore bits 1 and 3-9. The word portion of EA specifies an absolute register number from 0-'377.
0*	1	Bits 13-16 of the word portion of EA specify one of the registers '20-'37 in the current register set.
0	0	Bits 13-16 of the word portion of EA specify one of the registers 0-'17 in the current register set.

*This is a restricted instruction.

Leaves the values of CBIT and LINK unchanged. The values of the condition codes are indeterminate. See Chapter 9 for more information about register sets.

Note

Do not use this instruction to write into the procedure base, keys, or modals. You can use LPSW to change any of these three registers. In addition, you can use a control transfer to change the procedure base, or a mode control operation to change the keys or modals. Under no circumstances should you try to change the value of the current register set bits contained in the modals.

If the current ring is not 0 and EA is outside the range of 0-17 inclusive, any access causes an RXM violation.

▶ STC flr,r
Store Character
0 1 1 0 0 0 R\3 1 1 1 FLR 1 1 0

If the contents of the specified FLR are nonzero, the instruction stores the contents of bits 9-16 of the specified r into the character byte address contained in the associated FAR. Updates the contents of the appropriate FAR so that they point to the next character. Decrements the contents of the specified FLR by 1. Sets the condition codes to NE.

If the contents of the specified FLR are 0, the instruction sets the condition codes to EQ and does not store a character.

The instruction leaves the values of LINK and CBIT unchanged.

Note

When the instruction specifies FLR0, FAR0 is used. When the instruction specifies FLR1, FAR1 is used.

▶ STCD R,address
Store Conditional Fullword
0 1 1 0 0 0 R\3 1 0 1 1 1 1 1
AP\32

Compares the contents of R+1 and the contents of the 32-bit location referenced by the specified address pointer. If the two values are equal, the instruction stores the contents of R in that referenced location. If the two values are not equal, execution continues with the next instruction. STCD is an interlocked operation, guaranteed to work in a multiprocessor.

Leaves the values of CBIT and LINK unchanged. The condition codes reflect the result of the comparison. (See Table 5-3.)

Note

R must be an even numbered register.

▶ STCH r,address
Store Conditional Halfword
0 1 1 0 0 0 R\3 1 0 1 1 1 1 0
AP\32

Compares the contents of bits 17-32 of the specified R with the contents of the location referenced by the specified address pointer. If the two values are equal, the instruction stores the contents of r into that referenced location. If the two values are not equal, execution continues with the next instruction. Leaves the values of CBIT and LINK unchanged. Sets the condition codes to EQ if the store occurs and to NE if not.

The comparison and store will not be separated by execution of other instructions. Therefore, no instruction can alter the contents of the specified memory location between the compare and the store.

Note

This instruction is useful when two cooperating, sequential processes are manipulating shared data. It is interlocked against direct memory I/O. This means you can use it to interlock a process with a DMA, DMC, or DMQ channel, as well as to interlock a memory location that is possibly accessed by I/O.

▶ STEX R
Stack Extend
0 1 1 0 0 0 R\3 0 0 1 0 1 1 1

Extends the length of the procedure stack. The designated R contains a 32-bit number that specifies the word size of the extension.

The firmware rounds up the number contained in the specified R to an even number of words. The instruction uses this value to allocate a block of memory to the procedure stack. The extension and the initial stack segment do not have to be contiguous, since there may not have been enough room left in the initial stack to contain a complete frame.

Returns a segment number/word number in the specified R that specifies the starting address of the extension. The extension is automatically deallocated when the current procedure completes execution. There is no limit on the number of extensions you can make.

A stack fault occurs if there is no room for the extension. The values of LINK and the condition codes are indeterminate. See Chapters 8 and 11 for more information about this instruction, stacks, and stack faults.

▶ STFA far,address
Store FAR
0 0 0 0 0 0 1 0 1 1 0 1 FAR 0 0 0
AP\32

Stores the specified FAR contents as a hardware recognizable indirect pointer at the memory location referenced by the specified address pointer. If the bit number field of the specified FAR contains 0, the instruction stores the first two words of the pointer and clears the pointer's extend bit to 0. If the bit number field of the specified FAR does not contain 0, the instruction saves all three words of the pointer and sets the pointer's extend bit to 1. Leaves the values of CBIT and LINK unchanged. The values of the condition codes are indeterminate.

▶ STH r,address
Store Halfword
0 1 1 0 0 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Stores the contents of the specified r into the 16-bit location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ STPM
Store Processor Model Number
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0

Stores the CPU model number and microcode revision number in an 8-word field. XB contains a pointer to the field in memory. Table 14-12 shows the format of the field.

Table 14-12
STPM Memory Field Format

Word	Name	Description
1-2	Processor Model Number	Contains a code specifying the machine: 0L - 400/500, no Rev B microcode 1L - 400, Rev. B microcode 2L - Reserved 3L - 350 4L - 450/550 5L - 750 6L - 650 7L - 250 8L - 850 9L - 250-II 10L - 550-II 11L - 2250 15L - 9950
3-4	Microcode Revision	Word 1: Bits 1-8 reserved Bits 9-16 manufacturing microcode revision number Word 2: Bits 1-16 engineering microcode revision number
5	Processor Line	Specifies options enabled for this machine: Bits 1-15 reserved; must be zero; Bit 16 marketing segment specification bit.
6	Extended Microcode ID	To be implemented.
7-8	——	Reserved for future use.

The STPM instruction leaves the values of CBIT, LINK, and the condition codes unchanged.

Note

STPM is a restricted instruction.

▶ STTM
Store Process Timer
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0

Valid for the 550-II, 750, 850, I450, and 9950.

Stores the 48-bit process timer in the block referenced by XB. XB contains the address of a three-word memory block. Returns ETH, ETL+ITH, ITL as the current process time. Bits 1-10 of ITL contain the microsecond count.

The addition performed by this instruction is a 48-bit operation. It is equivalent to the following series of instructions:

```
LH    0,ITH   /* Load GR0 with the contents of ITH.
PIDH  0       /* Sign extend GR0 bits 1-16.
LH    1,ET    /* Load GR1 with the contents of ET.
A     0,1     /* Adds contents of GR0 AND GR1.
```

▶ TC R
Two's Complement Register
0 1 1 0 0 0 R\3 0 1 0 0 1 1 0

Forms the two's complement of the contents of the specified R and stores the result in R. An overflow causes an integer exception. The condition codes reflect the result of the operation. (See Table 5-3.) The value of LINK is indeterminate. If there is no integer exception, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ TCH r
Two's Complement r
0 1 1 0 0 0 R\3 0 1 0 0 1 1 1

Forms the two's complement of the contents of the specified r and stores the result in r. An overflow causes an integer exception. The condition codes reflect the result of the operation. (See Table 5-3.) The value of LINK is indeterminate. If there is no integer exception, CBIT is reset to 0.

If an integer exception occurs and bit 8 of the keys contains 0, the instruction sets CBIT to 1. If bit 8 contains a 1, the instruction sets CBIT to 1 and causes an integer exception fault. See Chapter 11 for more information.

▶ TFLR flr,R
Transfer FLR to Register
0 1 1 0 0 0 R\3 1 1 1 FLR 0 1 1

Transfers the contents of the specified FLR into the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TM address
Test Memory Fullword
1 0 0 1 1 0 1 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Sets the condition codes according to the numerical value of the 32-bit contents of the location specified by EA. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ TMH address
Test Memory Halfword
1 0 1 1 1 0 1 0 0 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Sets the condition codes according to the numerical value of the contents of bits 1-16 of the location specified by EA. (See Table 5-3.) Leaves the values of LINK and CBIT unchanged.

▶ TRFL flr,R
Transfer Register to FLR
0 1 1 0 0 0 R\3 1 1 1 FLR 1 0 1

Transfers the contents of R into the specified FLR. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ TSTQ R,address
Test Queue
0 1 1 0 0 0 R\3 1 0 0 0 1 0 0
AP\32

The address pointer in this instruction points to the QCB of a queue. This instruction tests the referenced queue and sets R to equal the number of items in the queue. Sets the condition codes to EQ when the queue is empty. If the queue is not empty, the instruction sets the condition codes to NE. Leaves the values of CBIT and LINK unchanged.

▶ WAIT
Wait
0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1
AP\32

The address pointer in this instruction points to a 16-bit semaphore counter, C. The instruction increments C. If C is greater than 0, either the resource is not available, or the event has not occurred. Removes the PCB from the ready list and adds it to the wait list associated with the semaphore. It then makes the register set available and turns off the process timer.

If C is less than or equal to 0, the currently executing process continues.

If the instruction places the PCB on the wait list, no general registers are saved. This means that a process cannot depend on these registers to be intact after this instruction occurs. This instruction potentially clears the general, floating, and XB registers.

Leaves LINK, CBIT, and the condition codes unchanged.

For more information about semaphores, PCBs, and wait lists, refer to Chapter 8.

Note

This is a restricted instruction.

▶ X r,address
Exclusive OR Fullword
1 0 0 1 1 0 0 1 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Performs an exclusive OR of the contents of the specified R with the 32-bit value contained in the location specified by EA. Stores the result in the specified R. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ XAD
Decimal Add
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0

Performs a decimal arithmetic operation under control of FAR0, FAR1, and GR2.

FAR0 contains the address of field 1. FAR1 contains the address of field 2. GR2 contains the control word; fields B and C of the control word specify the decimal operation to be performed, as shown in Table 14-13.

Table 14-13
XAD Decimal Operations

B	CB	Operation	Destination
0	0	+F1+F2	F2
0	1	+F1-F2	F2
1	0	-F1+F2	F2
1	1	-F1-F2	F2

The scale differential field in the control word specifies the difference in the decimal point alignment between F1 and F2, as follows:

SD	Relation of F1 and F2
SD>0	F1 > F2
SD=0	F1 = F2
SD<0	F1 < F2

If the T bit is set to 1, the results are forced positive. If the add operation results in an overflow, a decimal exception occurs. If no overflow occurs, the instruction sets CBIT to 0 to indicate success.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, FLR0, and FLR1. At the end of the instruction, the contents of these registers are indeterminate. The value of LINK is also indeterminate. The condition codes reflect the state of F2 after the decimal operation. (See Table 5-3.)

▶ XBTD
Binary to Decimal Conversion
0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1

Converts a binary number to a decimal number. FAR0 contains the decimal field address. GR2 contains the control word. This instruction uses fields A, E, and H of the control word. H specifies the length of the binary number and its location, as follows:

H	Length	Location
0	16 bits	GR3 register, high side
1	32 bits	GR3 register
2	64 bits	FP1 register

Converts the specified binary integer to a decimal integer and stores the result in the location specified by FAR0. Leaves the values of LINK unchanged. Overflow results in a decimal exception. If no overflow occurs, sets CBIT to 0. The values of the condition codes are indeterminate.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, and FLR0. At the end of the instruction, the contents of these registers are indeterminate.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

Note

The XBTD instruction does not use or modify FAR1, FLR1, or FACC1.

▶ XCM
Decimal Compare
0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0

Compares two decimal numbers and sets the condition codes depending on the result of the compare. Uses the G field of the control field to adjust the two numbers before the compare, as follows:

G Field	Decision
>0	Low-order digits of F1 only affect the initial borrow from the low-order digit of F2.
<0	Assume F1 is zero-extended with low zeroes.

FAR0 contains the address of field 1 (F1). FAR1 contains the address of field 2 (F2). GR2 contains the control word. This instruction uses fields A, B, C, E, F, G, and H of the control word.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate. Leaves the value of LINK in an undefined state. The condition codes reflect the result of the comparison, as follows:

CC	Test Result
GT	F2 > F1
EQ	F2 = F1
LT	F2 < F1

▶ XDTB
Decimal to Binary Conversion
0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0

Converts a decimal string to a binary string. FAR0 contains the address of the decimal string. GR2 contains the control word.

This instruction uses the A, E, and H fields. Field H specifies the length of the binary string and its location, as shown below.

H	Length	Destination Register
00	16 bits	GR2H
01	32 bits	GR2
10	64 bits	GR2/GR3

Converts the decimal string to a binary string of the specified type and stores it in the specified register. A conversion error causes a decimal exception. If no decimal exception occurs, the instruction sets CBIT to 0. The values of LINK and the condition codes are indeterminate.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, and FLR0. At the end of the instruction, the contents of these registers are indeterminate.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

Note

This instruction does not use or modify FAR1, FLR1, or FAC1.

▶ XDV
Decimal Divide
0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1

Divides a decimal number, D2, by another, D1, and stores the quotient and remainder in the location of D2.

FAR0 contains the address of D1. FAR1 contains the address of D2. L contains the control word. This instruction uses fields A, B, C, E, F, and H.

Both decimal numbers must be in trailing sign embedded format. In addition, D2 must contain a number of leading zeroes equal to the length of D1.

The XDV instruction divides the two numbers. After the divide, the location of D2 contains the quotient of length (D2 length - D1 length) followed by the remainder of length (D1 length). Since D2 had leading zeroes, no overflow can occur.

If the T bit contains a 1, the results will be forced positive. For more information about decimal arithmetic, refer to Chapter 6.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, FLR0, and FLR1. At the end of the instructions, the contents of these registers are indeterminate.

At the end of the instruction, the condition codes, LINK, FAR0, and FAR1 contain undefined results. If no overflow occurs, CBIT is reset to 0.

If D1 is 0, overflow occurs and causes a decimal exception. Decimal exceptions also occur if D1 or D2 has the incorrect data type or if the length of D2 is less than that of D1. If no decimal exception occurs, the instruction sets CBIT to 0.

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

▶ XED
Numeric Edit
0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0

Edits the contents of a string under control of a subprogram.

The registers used are GR2 (L), XB, FAR0, FAR1, and FLR0. At the end of the instruction, the contents of these registers are indeterminate.

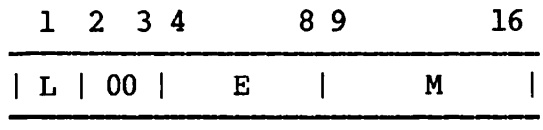
FAR0 contains the address of the source string. The source string must be leading separate sign type and must have at least the same number of decimal digits and the decimal point alignment as called for in the edit subprogram.

FAR1 contains the address of the destination string. Bits 1-8 of A contain the floating character; bits 9-16, the status register. Bits 1-8 of B contain the number of remaining bytes to be processed (used if a fault or interrupt occurs). Bits 9-16 of B contain the suppression character whose initial value is determined by bit 12 of the keys ('240 if bit 12 contains 0; '40 if bit 12 contains 1). XB contains the address of the edit subprogram.

The instruction uses an edit subprogram to alter a source string and store the edit result in a destination location(s). To set up, perform a decimal move to correct the type, alignment, and length of the number to be edited. Next, use a LCEQ instruction to set up the initial contents of the register.

Each word in the edit subprogram has the format shown in Figure 14-4, where:

L is 1 if this word is the last word in the subprogram,
0 if it is not the last word;
E is a suboperator;
M is a suboperator modifier.



Edit Subprogram Word Format
Figure 14-4

The XED instruction uses several variables internally to control the edit subprogram. These are shown in Table 14-14.

Table 14-14
XED Internal Variables

Var	Definition
SC	Zero suppression character; contained in B. Initial value is the space character ('240 or '40 if bit 12 of the keys contains 0 or 1, respectively).
FC	Floating edit character; contained in A. Initial value is not defined.
SIGN	Sign of the source field. The first character fetch sets up the value of this variable.
SIG	End zero suppression flag.

There are 17 edit suboperators, shown in Table 14-15.

Table 14-15
XED Suboperators

Subop	Mnem	Name and Description
00	ZS	Zero Suppress. Fetches M digits from the source field consecutively, each time checking SIG. If SIG is 1, copies the digit into the destination string. If SIG is 0 and the digit is not 0, inserts the floating character (if defined) and copies the digit into the destination field. If SIG is 0, the digit is not 0, and the floating character is not defined, sets the SIG flag and copies the digit into the destination. If SIG and the digit are both 0, substitutes SC for the 0 digit in the destination field.
01	IL	Insert Literal. Copies M into the destination string. Increments XB and FAR1 by 1.
02	SS	Set Suppress Character. Sets SC to M and increments XB by 1.
03	ICS	Insert Character. If SIG is 1, copies M into the destination string. If SIG is 0, copies SC into the destination string. Increments XB and FAR1 by 1.
04	ID	Insert Digits. If SIG is 0, and FC is defined, copies FC and M digits into the destination field then sets SIG to 1. Increments XB by 1, FAR0 by M, and FAR1 by M+1. If SIG is 0 and FC is not defined, sets SIG to 1 and copies M digits from the source to the destination. Increments XB by 1 and both FAR0 and FAR1 by M. If SIG is 1, copies M digits from the source to the destination and increments XB by 1 and both FAR0 and FAR1 by M.
05	ICM	Insert Character if Minus. If SIGN = 0, copies M into the destination string. If SIGN = 1, copies SC into the destination string. Increments both SB and FAR1 by 1.
06	ICP	Insert Character if Plus. If SIGN = 0, copies M into the destination string. If SIGN = 1, copies SC into the destination string. Increments both SB and FAR1 by 1.
07	SFC	Set Floating Character. Sets FC to M and increments XB by 1.
10	SFP	Set Floating if Plus. If SIGN = 0, sets FC to M. If SIGN = 1, FC to SC. Increments XB by 1.
11	SFM	Set Floating if Minus. If SIGN = 1, sets FC to M. If SIGN = 0, sets FC to SC. Increments XB by 1.
12	SFS	Set Floating to SIGN. If SIGN = 0, sets FC to '253. If SIGN = 1, sets FC to '255. Increments XB by 1.

Table 14-15 (continued)
XED Suboperators

Subop	Mnem	Name and Description
13	JZ	Jump if Zero. If the condition flag in A = 0, increments XB by 1. If the condition flag in A = 1, adds M to XB and then increments XB by 1.
14	FS	Fill with Suppression Characters. Copies SC M times into the destination string. Increments XB by 1 and FAR1 by M.
15	SF	Set Significance. If SIG = 0 and FC <> 0, inserts FC into the destination string, sets SIG to 1, and increments XB and FAR1 by 1. If SIG = 0 and FC = 0, sets SIG to 1 and increments XB and FAR1 by 1. If SIG = 1, increments XB by 1.
16	IS	Insert Sign. If SIGN = 0, copies '253 into the destination string. If SIGN = 1, copies '255 into the destination string. Increments XB by 1.
17	SD	Suppress Digits. Fetches M digits from the source string and checks if they are '260. If the source digit = '260, inserts SC into the destination string. If the source digit <> '260, copies the source digit into the destination string. Increments XB by 1 and both FAR0 and FAR1 by M.
20	EBS	Embed Sign. Fetches M digits from the source string. If SIGN = 0, copies each digit into the destination string. If SIGN = 1, embeds a minus sign into each digit before copying it into the destination string. Table 6-16 shows the characters used to represent the sign/digit combinations. Note that } represents negative 0.

▶ XH r,address
Exclusive OR Halfword
1 0 1 1 1 0 0 1 1 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Performs an exclusive OR of the contents of the specified r with the 16-bit value contained in the location specified by EA. Stores the result in r. Leaves the values of LINK, CBIT, and the condition codes unchanged.

Note

This instruction also has a register-to-register and an immediate form. See Chapter 3 for more information.

▶ XMP
Decimal Multiply
0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0

Multiplies one decimal number, D2, by another, D1, and stores the result in D2's location in memory.

FAR0 contains the address of D1. FAR1 contains the address of D2. L contains the control word. This instruction uses fields A, B, C, E, F, G, H, and T. Note that field G, the scale differential, must contain the number of decimal digits in the multiplier (M). This value is not the same as the length of the D2.

For correct results, D2 must contain a number of leading zeroes equal to or greater than the length of D1.

The instruction multiplies D2 by D1 and stores the result in the location specified by FAR1. The result of the multiply is:

D1 x D2 + partial product field

The partial product field is equal to:

length(D2) - M.

The partial product field is left justified in D2's location. The maximum partial product added in per traverse of the multiplicand is:

source digits + multiplier digits processed

There is also an implied weighting of the partial product field. The weighting is:

10 ** multiplier digits

If the T bit contains a 1, the results are forced positive.

The registers used are GR0, GR1, GR3 (E), GR4, GR6, FAR0, FAR1, and XB. At the end of this instruction, the contents of these registers are indeterminate. At the end of the instruction, the condition codes reflect the state of the result. (See Table 5-3.) Overflow causes a decimal exception. If no overflow occurs, resets CBIT to 0. LINK contains undefined results.

A decimal exception occurs if there are more potential or actual product digits than there is space in D2. If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information.

▶ XMV
Decimal Move
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1

Moves a string of characters from one location to another.

FAR0 contains the address of the source string. FAR1 contains the address of the destination string. GR2 contains the control word. This instruction uses fields A, B, D, E, F, G, H and T.

The instruction moves the contents of the source field into the destination field from right to left. If the B field in the control word is 1, the instruction changes the sign of the source field during the move. If the D field in the control word is 1 and the scale differential is greater than 0, the instruction rounds the source field during the move. If the scale differential (from the H field) is less than 0, the instruction pads the source field with SD trailing zeroes before transferring.

If the T bit is set to 1, the result will be forced positive.

An overflow causes a decimal exception. If no decimal exception occurs, the instruction resets CBIT to 0. At the end of the instruction, LINK, FAR0, and FAR1 contain undefined results. The values of the condition codes reflect the state of the destination field after the move. (See Table 5-3.)

If a decimal exception occurs and bit 11 of the keys contains a 0, the instruction sets CBIT to 1. If bit 11 contains a 1, the instruction sets CBIT to 1 and causes a decimal exception fault. See Chapter 11 for more information about decimal exceptions.

Note

The source and destination strings may not overlap in memory.

▶ ZCM
Compare Character Field
0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1

Compares two fields and sets the condition codes depending on the result of the compare. Uses registers GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

FAR0 contains the address of field 1 (F1). FLR0 contains an integer specifying the length of F1. FAR1 contains the address of field 2 (F2). FLR1 contains an integer specifying the length of F2.

The instruction compares the contents of F1 and F2 on a byte by byte basis. If the fields are not of equal length, the instruction automatically extends the shorter string with space characters. Sets the condition codes as a result of the comparison, as follows:

Result of Compare	Set Condition Codes
F1 > F2	GT
F1 = F2	EQ
F1 < F2	LT

When the instruction completes execution, the values of CBIT and LINK are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZCM does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

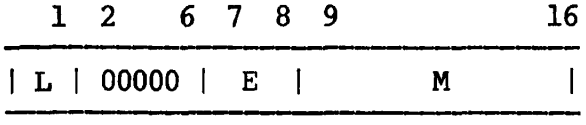
▶ ZED
Character Field Edit
0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1

Controls an edit subprogram.

FAR0 contains the address of the source string. FLR0 specifies the length of the source string. FAR1 contains the address of the destination string. XB contains the address of the edit subprogram.

The ZED instruction uses the edit subprogram to alter the source string, then loads the edited result into the destination string. The subprogram, addressed by the contents of XB, contains a list of commands, each with the format shown in Figure 14-5, where:

L is 1 if this command is the last command in the subprogram,
0 if it is not;
E is the edit opcode;
M is the edit modifier.



ZED Subprogram Word Format
Figure 14-5

Bits 2-6 must be 0.

M, the operator modifier, specifies information E uses when editing the source string. (See Table 14-16.)

E, the edit suboperator, specifies the operation to be performed on the source string. Table 14-16 shows the available values for E.

Table 14-16
ZED Suboperators

Subop	Value	Action
CPC	00	Copies characters from the source string into the destination string. If the length of the source string is greater than the contents of the M field, then CPC moves a total of M source characters into the destination string, increments FAR0 and FAR1 by M, increments XB by 1, and decrements FLR0 by M. If the length of the source string is less than the contents of the M field, then CPC moves the rest of the source string into the destination string, and then pads the remaining space to be filled with spaces. Increments FAR0 by FLR0, FAR1 by M, increments XB by 1, and decrements FLR0 by FLR0 (so FLR0 = 0).
INL	01	Inserts M into the destination string and increments XB and FAR1 by 1.
SKC	10	Skips characters in the source string. If the remaining length of the source string is greater than or equal to the contents of the M field, SKC skips over the next M characters of the source field by incrementing FAR0 by M and decrementing FLR0 by M. If the remaining length of the source string is less than the contents of the M field, SKC skips over FLR0 characters in the source string by incrementing FAR0 by FLR0 and decrementing FLR0 by FLR0 (FLR0 = 0). In either case, SKC increments XB by 1.
BLK	11	Inserts M spaces into the destination string, increments FAR1 by M, and increments XB by 1. A space is '240 or '40, depending on whether bit 12 of the keys contains 0 or 1.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZED does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

▶ ZFIL
Fill Field with Character
0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0

Stores a character into a series of destination bytes. Uses registers GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

Bits 9-16 of GR3 contain the character to be stored. FAR0 contains the starting address of the destination field (byte aligned). FLR1 contains an integer specifying the length of the destination field (in bytes).

The instruction stores the character specified in GR3 in each byte of the destination field. If FLR1 contains 0, no operation takes place. Leaves the values of CBIT, LINK, and the condition codes indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZFIL does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

▶ ZM address
Zero Memory Fullword
1 0 0 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads 0 into the 32-bit location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ZMH address
Zero Memory Halfword
1 0 1 0 1 1 DR\3 TM\2 SR\3 BR\2
[DISPLACEMENT\16]

Calculates an effective address, EA. Loads 0 into the 16-bit location specified by EA. Leaves the values of LINK, CBIT, and the condition codes unchanged.

▶ ZMV
Move Character Field
0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 0

Moves a character field from one location to another. Uses registers GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

FAR0 contains the address of the source string (byte aligned). FLR0 specifies the length in bytes, N, of the source string. FAR1 contains the address of the destination string (byte aligned). FLR1 specifies the length in bytes, M, of the destination string.

Compares N and M. If N is less than M, the instruction moves the contents of the source string into the destination string followed by M-N space characters. A space character is '240 or '40 when bit 12 of the keys is 0 or 1, respectively. If the destination string is shorter, the instruction moves the first M characters of the source string into the destination string.

When the instruction completes, the values of FAR0, FAR1, FLR0, FLR1, CBIT, LINK, and the condition codes are indeterminate.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZMV does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

This instruction does not work with overlapping strings. See Chapter 6 for more information.

▶ ZMVD
Move Characters Between Equal Length Strings
0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1

Moves characters from one string to another of equal length. Uses registers GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

FAR0 contains the address of the source string. FAR1 contains the address of the destination string. FLR1 contains the number of characters to move, N.

The instruction moves N characters from the source string to the destination string. Characters are moved from lower addresses to higher addresses.

When the ZMVD instruction completes, the values of FAR0, FAR1, FLR0, FLR1, CBIT, LINK, and the condition codes are indeterminate.

Note

The ZMVD instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZMVD does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

This instruction does not work with overlapping strings. See Chapter 6 for more information.

▶ ZTRN
Character String Translate
0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0

Translates a string of characters and stores the translations in the specified destination. Uses registers GR3 (E), GR4, FAR0, FAR1, FLR0, and FLR1. At the end of this instruction, the contents of these registers are indeterminate.

FAR0 contains the address of the source string (byte aligned). FAR1 contains the address of the destination string (byte aligned). FLR1 specifies the length of the source and destination strings. XB contains the address of a translation table. Each byte in the 256-byte table contains an alphabetic character.

The instruction uses the address in FAR0 to reference a character. It interprets this character as an integer, adding it to the contents of XB to form an address into the translation table. The instruction takes the referenced character in the translation table and writes it into the location specified by FAR1. After storing the character, the instruction increments the contents of FAR0 and FAR1 by 1, decrements the contents of FLR0 by 1, and repeats the operation until FLR1 contains 0.

At the end of the instruction, FAR0 and FAR1 point to the last byte in the source and destination strings, respectively. FLR1 contains 0. Leaves the values of XB, CBIT, LINK, and the condition codes unchanged.

Note

This instruction uses GR3, GR4, the FARs, and the FLRs during its operation. Since ZTRN does not save the contents of these registers before using them, any data contained in them is overwritten when this instruction executes, unless you save it ahead of time.

APPENDIXES

A
Power-up

POWER-UP AND SYSTEM INITIALIZATION

All 50 Series processors perform the following steps in the sequence shown for power-up and system initialization.

1. Power becomes valid.
2. VCP (Virtual Control Panel) or maintenance processor conducts self tests.
3. CPU micro-diagnostics perform processor validation.
4. CPU initializes to the state shown in Table A-1.

Note

The failure of step 2, 3, or 4 stops the entire process and causes an error message to be displayed.

Table A-1
CPU Initialization Values

Element Initialized	Initialized Value
CRS (current register set)	0 (specifies RF2, the first user register set)
Registers in CRS	0, generally
All DMA (direct memory access) I/O registers but 6	Undefined
DMA register 6	0 (or 3)/'1000 (manufacturing test equipment)
Keys	0 (addressing mode now 16S)
Modals	0
Program counter	Ring 0, segment 0, offset '1000
RSAVPTR (register save pointer)	0

INDEX

Index

T

SURVEY

READER RESPONSE FORM

DOC3060-192 System Architecture Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC3060-192 System Architecture Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC3060-192 System Architecture Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC3060-192 System Architecture Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC3060-192 System Architecture Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

   _____

   _____

   _____

4. What faults or errors gave you problems? _____

   _____

   _____

   _____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760