NonStop™ II System Description Manual

TANDEM

TANDEM

Tandem NonStop™ II System Description Manual

TANDEM NonStop II (TM)

SYSTEM DESCRIPTION MANUAL

Second Edition

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, California 95014

P/N 82077 B00

April 1982
Printed in U.S.A.

Summary of Changes in This Revision

This manual is the second edition of the NonStop II System Description Manual. It includes the following changes to the first edition:

- The instruction set definitions have been updated to reflect the new memory management algorithm for choosing pages to swap out, which results in the deletion of the PHYREF table and the FLRU, SLRU, and UREF instructions, plus microcode changes in the MAPS and UMPS instructions. Microcode changes in the LCKX, BNDW, XSTR, and XSTP instructions have also been recorded.
- The introductory description of the processor hardware has been expanded to include a brief discussion of the memory control unit, control panel, loadable control store, clock generator, PMI, and DDT.
- Some of the instruction definitions in Section 3 have been rewritten for greater clarity, and more information on overflow conditions has been added.
- Appendixes A and C have been combined into a single appendix (B), in order to bring the symbol definitions next to the table that uses the symbols. Old Appendix B has been renumbered to Appendix A.
- Minor technical and typographical errors have been corrected.

PREFACE

This manual provides a conceptual and functional description of the Tandem NonStop II (TM) system, presented as follows:

- Section 1 summarizes the factors involved in NonStop computer operation, and tells how this type of operation is achieved in the Tandem system. This section also stresses the close interrelationship between the system's hardware and software, and illustrates how these two aspects of the system interact to make NonStop performance possible.

- Section 2 describes the principles on which the system hardware operates, and shows how the hardware supports NonStop operation. Specifically, it discusses such factors as: hardware system structure, fundamental NonStop functions, processor module organization, program execution from the hardware standpoint, data formats and number representation, logical memory organization, the interrupt system, interprocessor buses and input/output channels, and physical memory mapping.

- Section 3 defines the instruction set for the Tandem system, in text form with illustrations.

- Appendixes A and B consist of reference tables pertaining to the instruction set.

- An index is provided to assist the reader in locating specific topics in this manual.

This manual was written for potential and present Tandem customers seeking a functional description of the hardware and instruction set, for Tandem field analysts and service engineers, and for enrollees in various courses provided by Tandem.

Before using this manual, one should read Introduction to Tandem Computer Systems for a more general overview of the system. This introductory manual explains the basic concepts and purposes behind the system architecture described in this manual and its counterpart for Tandem NonStop systems, the NonStop System Description Manual. Ideally, the reader should also have some working experience with the Tandem system.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCING THE TANDEM NonStop II (TM) COMPUTER SYSTEM

During the recent past, computer systems have evolved from the massive, unreliable vacuum tube machines of yesteryear to the compact, dependable systems of today. Early computers were very restrictive and limited; they required programmers to run their programs in a stand-alone environment (as shown in Figure 1-1).

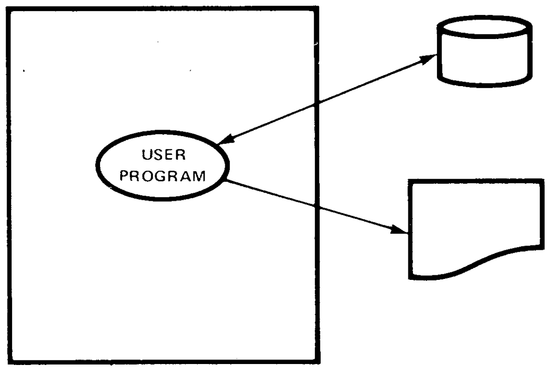


Figure 1-1. Stand-Alone Computer

These stand-alone programs were written in machine language and consisted of long lists of numbers. They required painstaking care to create. In fact, the programmer's responsibility included not only coding the application but implementing the details of physical input/output as well. In its stand-alone operating environment, a running program preempted all hardware resources of the entire machine--but seldom actually used them all.

Eventually, the primitive stand-alone environment gave way to one where the machine's hardware resources were managed by a control-oriented software package called an operating system. This simplified and generalized access to peripheral input/output devices. Building upon this idea, software designers extended operating systems to allow several user programs to share the limited processor and memory resources of the machine in a multiprogramming environment (Figure 1-2).

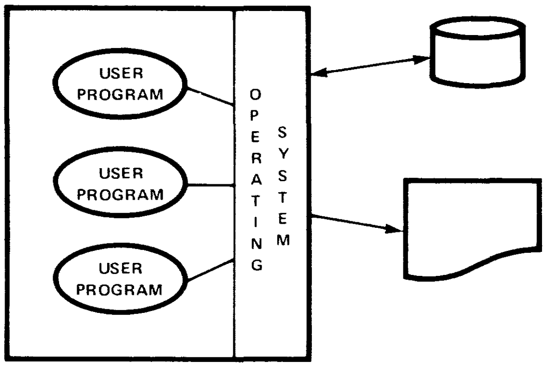


Figure 1-2. Multiprogramming Environment

Further developments led to operating systems that managed programming environments spread over several processors (Figure 1-3). These multiple processor configurations offered an additional advantage: they allowed a customer to increase the overall power of his system just by adding more processors to it.

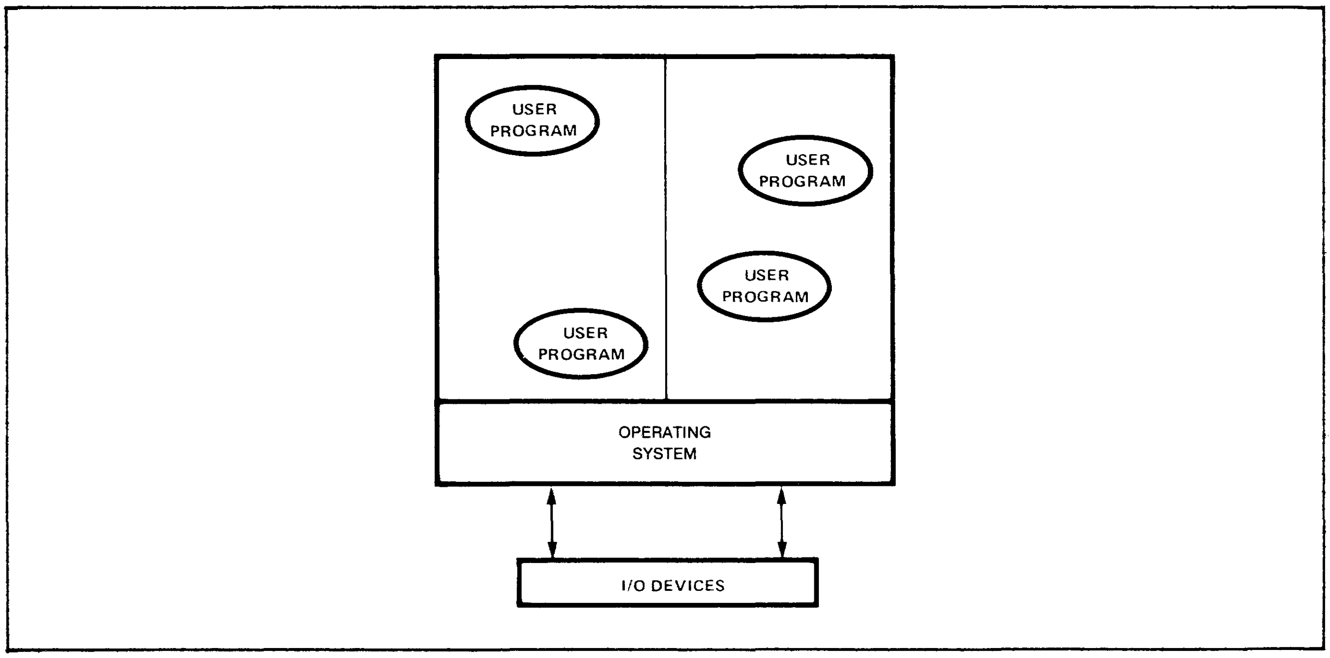


Figure 1-3. Multiple-Processor Environment

Finally, designers further extended the power of the computer by joining several groups of processors into networks of systems connected by long-distance communication lines (Figure 1-4). This approach to distributed computing power matched the natural organization of offices and plants found in many businesses and permitted them to establish and manage geographically-independent data bases.

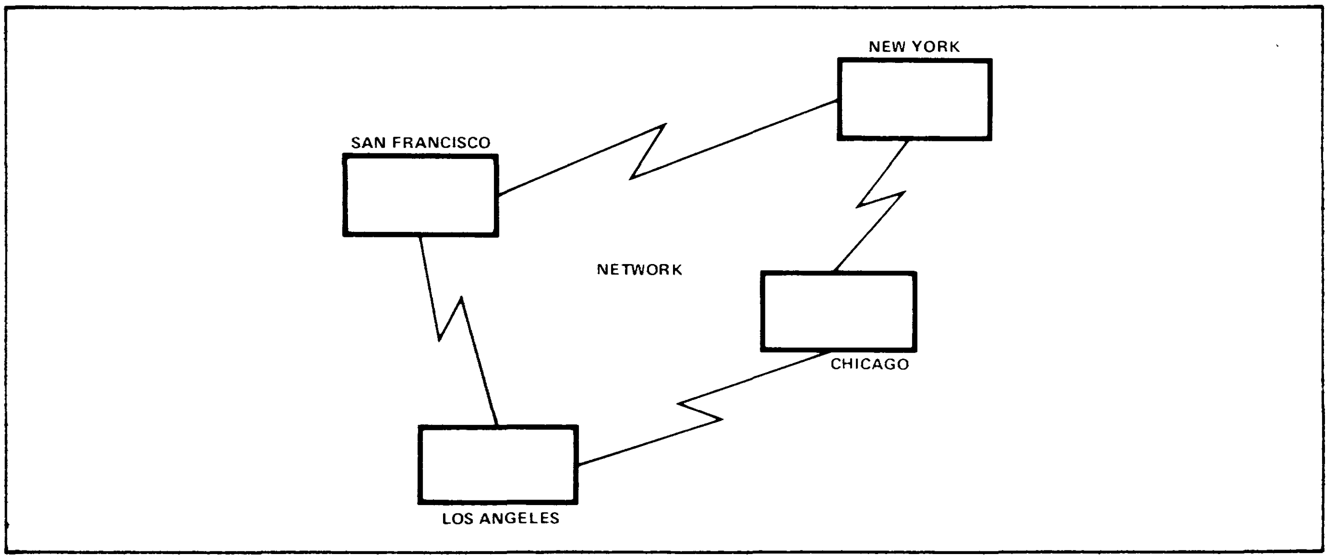


Figure 1-4. Network-Based Environment

INTRODUCING TANDEM'S NonStop AND NonStop II SYSTEMS

The Tandem NonStop and NonStop II systems incorporate all of the above technological advances: they are multiprogramming, multiple-processor, network-oriented systems. But beyond this, Tandem's primary design goal was to make these computers "NonStop," easily-expandable systems. Where the overall design required trade-offs between reliability and other factors, reliability always came first.

At the heart of NonStop operation are three interrelated factors: fault tolerance, on-line repair, and modular design. FAULT TOLERANCE implies that the system is able to continue operation even if a particular component fails. ON-LINE REPAIR means that field engineers can repair or replace faulty cpu's, power supplies, input/output controllers, or buses while the rest of the system continues to operate. And once an item is repaired, it can be reintegrated into the system without interrupting the on-line application work in progress. Both of these features are related to the MODULAR SYSTEM DESIGN, where system components are constructed to allow flexible system configuration and simplified maintenance.

The expandability feature that allows customers to incrementally extend the size and power of their systems also arises from the system's modular design. This feature lets customers upgrade system performance just by adding more cpu's, memory, or peripheral devices. Conventional systems, typically, cannot be easily expanded to add more cpu capability; as a result, they cannot grow with a customer's application or evolve to fit a wide range of computing needs.

The NonStop and NonStop II systems perform many different kinds of operations to make processing easier for their users. As some of their major functions, these systems:

- Prepare program files for execution as processes (running programs) in a virtual environment.

- Schedule cpu time among multiple processes according to their assigned priorities and their time of entry into an executable state.

- Provide the virtual memory function by automatically bringing absent memory pages in from disc when needed.

- Allow processes to communicate with each other regardless of the cpu's on which they are running.

- Permit logical, file-oriented access to all physical devices regardless of the cpu's to which these devices are attached.

- Allocate resources among running processes so that each process appears to have all resources in the system available to it.

HARDWARE AND SOFTWARE INTEGRATION

Ultimately, all of the major functions listed above depend on fundamental services provided by the basic software for the computer --the GUARDIAN operating system. Many of these functions are performed so often, however, that the designers could greatly increase overall system efficiency by closely integrating various software operations with those of the hardware components. In fact, certain critical procedures (originally part of the operating system) have been partially or entirely reimplemented in the hardware microcode. Now, these procedures are invoked just by executing a single hardware instruction.

A good example of how the hardware and software interact to increase system efficiency is provided by an instruction which queues a process for execution--the MRL (Merge Ready List) instruction. This instruction takes a pointer to a system table entry representing a process, searches a list of similar entries arranged by execution priority, and merges the entry into the list. If the priority in the new process entry exceeds that of the currently-executing process, the instruction notifies the operating system by interrupt. By removing this function from the software and placing it in the microcode, system designers have reduced to ONE the number of instruction fetches needed to do the operation. This, of course, dramatically increased the speed of the function. As the system software evolved, this type of hardware/software integration at the instruction set level increased. This, in turn, both simplified the GUARDIAN software and made the total computer system much more efficient.

In the NonStop II system, Loadable Control Store (LCS) has been added for system microcode and diagnostics. This allows Tandem to supply new versions of the microcode to customers on tape, to be loaded into the processors by system utility programs whenever new versions of the software are installed. Loadable Control Store thus provides a simple means for Tandem to add further system improvements at the microcode level.

While hardware/software cooperation is desirable for overall system efficiency, it is ABSOLUTELY NECESSARY to ensure such NonStop features as a failure-tolerant input/output system. As an example, consider a system that includes a device controller with two ports, each connected to a different cpu. In this system, the ownership of the device is agreed upon by the controller hardware AND by operating system software in each cpu. In this example, suppose that the controller is presently being serviced by CPU 1 (Figure 1-5).

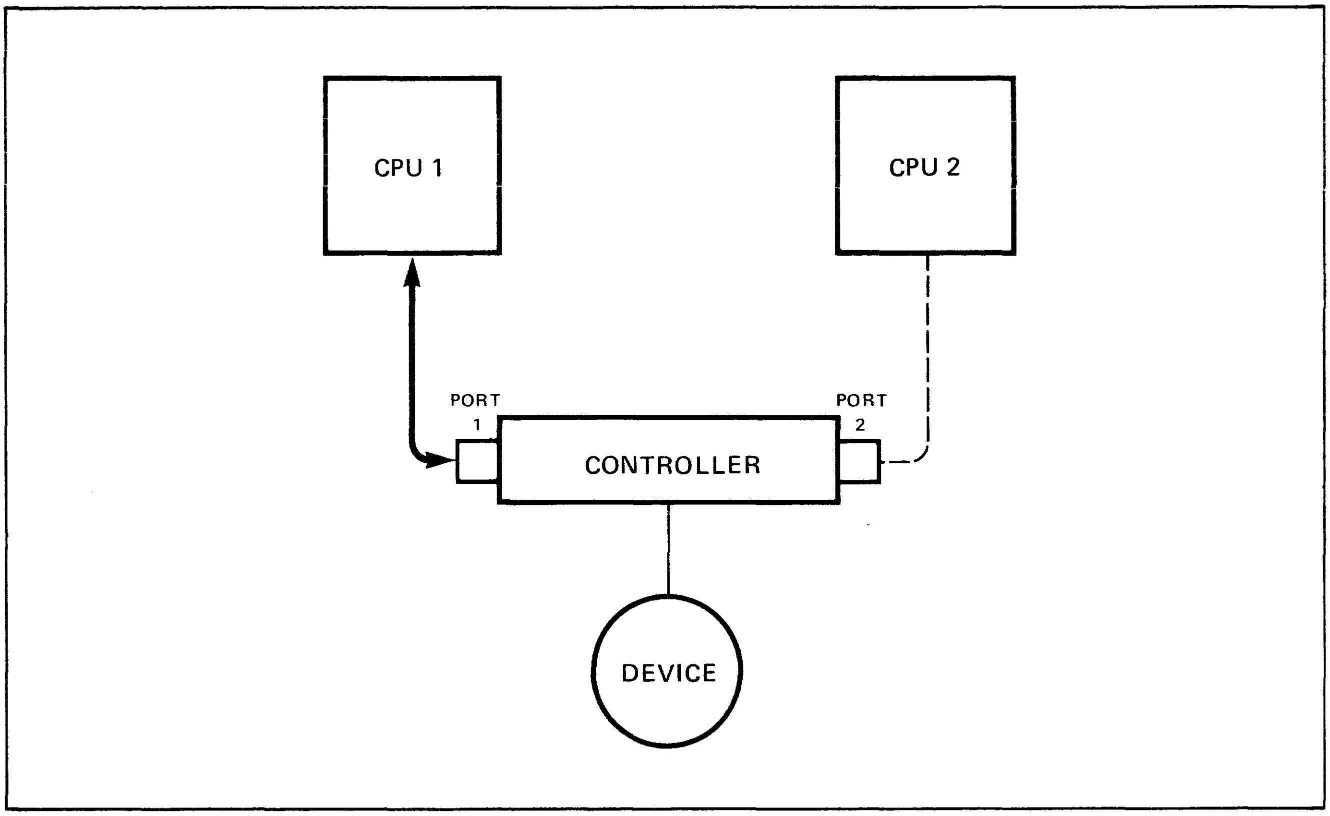


Figure 1-5. Fault-Tolerant Device Management in a NonStop System

The importance of joint hardware/software interaction in this system is underscored by considering what happens when certain kinds of errors occur. For example, suppose that the logic in Port 1 of the controller fails (Figure 1-6).

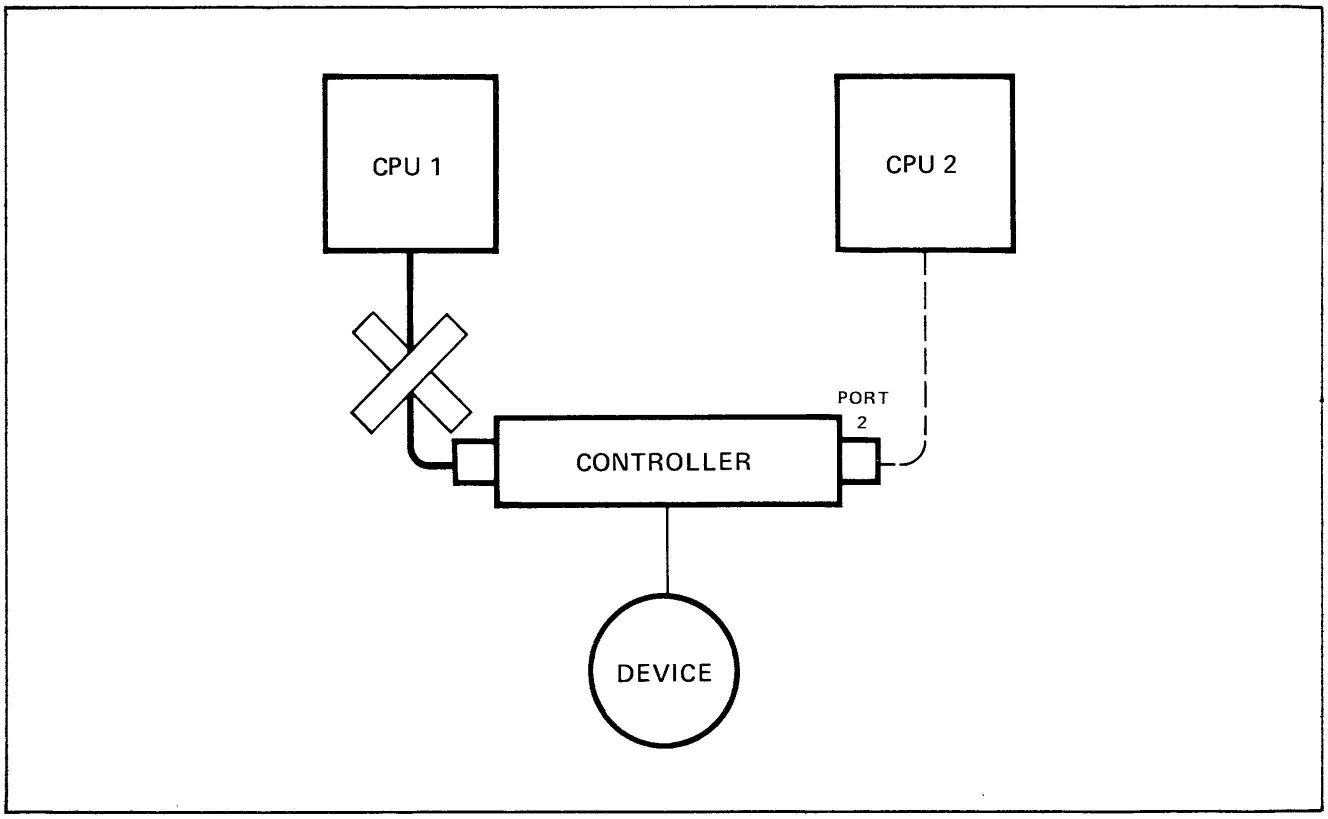


Figure 1-6. Controller Port Logic Failure

If this hardware failure results in a constant flow of interrupts, they will be detected by the GUARDIAN software in CPU 1. Now interrupts are not, of course, abnormal in the system. But when they occur with too great a frequency as in this case, the operating system assumes that an abnormal situation exists and executes a hardware instruction to disable Port 1 of the controller. This completely stops the flow of interrupts from the faulty port. At this point, software ownership of the controller may be switched to CPU 2, which in turn switches hardware controller ownership to the remaining operational port, and NonStop operation continues (Figure 1-7).

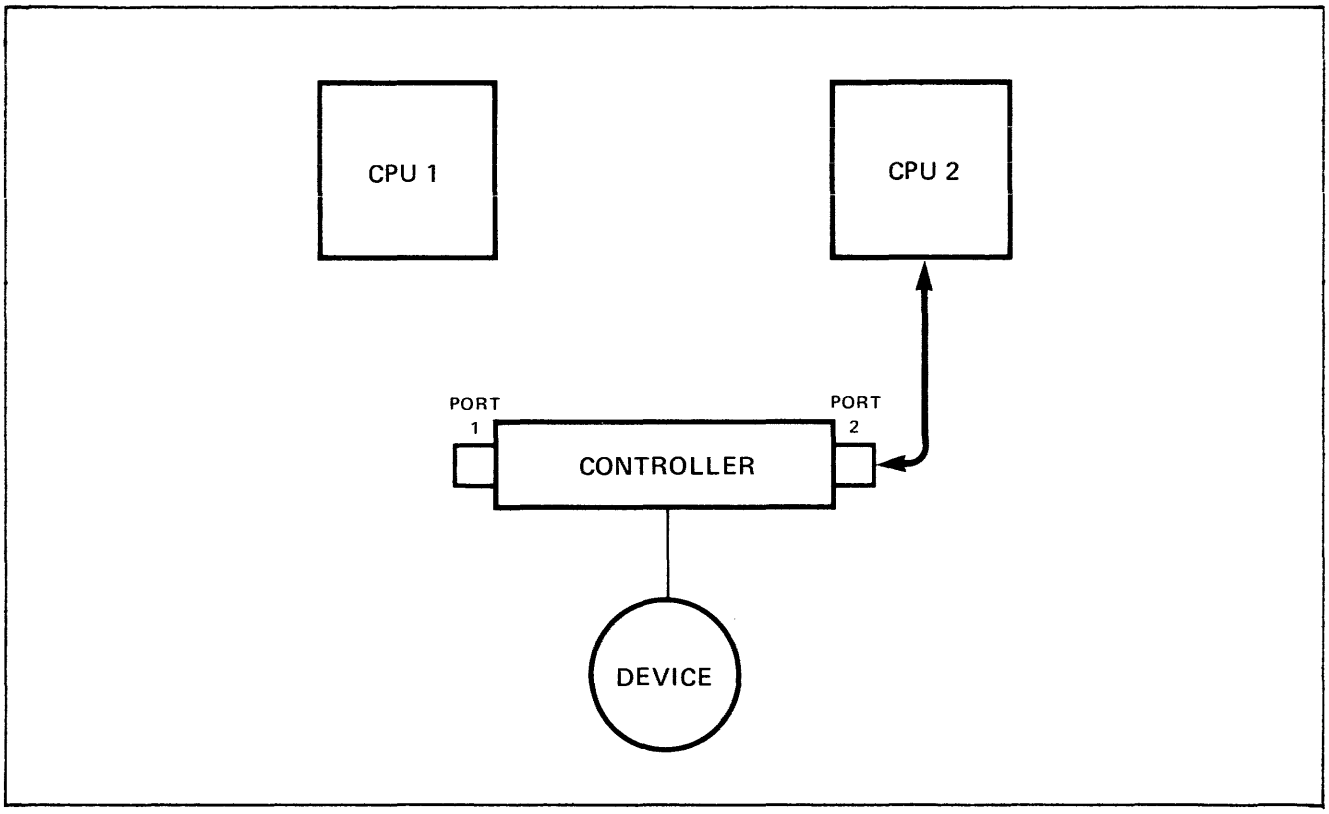


Figure 1-7. Switching Controller Ownership

This kind of joint hardware/software cooperation is necessary for any system that must function in a failure-tolerant way--the total burden of reliability must be carried by both the hardware and software. Without this mutual support, such a system would be impossible. And to implement such a system, a fully-unified overall design is required that carefully integrates the hardware and software with one another. The Tandem system is based on this kind of design.

SECTION 2

HARDWARE PRINCIPLES OF OPERATION

SYSTEM STRUCTURE

Hardware components of a NonStop system must be designed to allow continued execution of processes and access to data bases even if a single component fails. These design goals are illustrated in diagram 1 of Figure 2-1.

From a software point of view, failure tolerance for the user's process is accomplished by executing a secondary (or "backup") process in another processor, so programmed to require only periodic checkpoint messages to keep up to date on the current state of the primary process. Upon any failure of the processor that is executing the primary process, the backup process can resume execution of the work from the point of the last valid checkpoint. The backup process, instead of the primary process, will then be accessing the data base on disc. As indicated in the diagram, dual data paths are desired in order to assure communication of the checkpoint messages.

From a hardware point of view, failure tolerance for the user's data base is accomplished by the use of dual-ported controllers and, optionally, by maintaining duplicate data on two separate disc volumes ("mirrored" volumes). For mirrored volumes, all data written out to the user's files is automatically written into both disc volumes. Thus, whenever data is read from the files, either volume may be accessed, since they contain identical information. Like the interprocessor communications, two data paths to the disc volumes are desirable.

The various hardware features that accomplish these two major goals work together as an effective total solution. But for illustrative purposes, each feature is considered as a separate entity in the following discussions--illustrated by the remaining six diagrams in Figures 2-1 and 2-2.

It should be noted in considering the following information that, although the mechanics of instant on-line reconfigurability reside in the hardware, the control of such actions is a function of the GUARDIAN operating system.

Independent Multiple Processors

The NonStop II system consists of two to sixteen processor modules. A processor module is sometimes referred to as a central processing unit, or cpu, for convenience, although in a Tandem system, no one processor is more "central" than any other. Each processor (cpu) contains the functions that normally comprise a complete computer system: instruction processing unit (IPU), memory, and input/output channel. In addition, each module contains logic for a fourth main function: the interprocessor bus interface through which the processors communicate with each other. Furthermore, each module is associated with its own separate power supply. (See diagram 2 in Figure 2-1.) Therefore, each processor module is capable of operating independently of, and simultaneously with, all other processor modules in the system.

This fundamental design feature means that each processor is totally self-sufficient. An IPU failure, for example, cannot prevent another processor from functioning, since there are no shared elements, such as memory. A failing IPU cannot contaminate any memory data outside of its own module.

Dual-Bus Data Paths

Each processor module is connected to all other processor modules via redundant high-speed interprocessor buses, each controlled by its own separate bus controller. See diagram 3 in Figure 2-1. Programs running in one processor module communicate with programs running in other processor modules by means of these buses. Each interprocessor bus is fully autonomous, operating independently of (but simultaneously with) the other bus.

The use of two buses assures that two paths exist between all processor modules in the system. If one bus fails, all interprocessor communication is automatically routed over the remaining bus. The use of bus controllers that are separate and independent of the logic circuits within the modules assures that no failure of a processor module will cut off bus transmission.

The interprocessor bus interface in each module is capable of accepting transmissions from either bus, under control of the operating system.

Dual-Port Device Controllers

Data is transferred between an input/output device (i.e., disc, terminal, line printer, etc.) and a processor module by means of an input/output channel. Each processor module has one i/o channel that is capable of communicating with up to 256 i/o devices. See diagram 4 in Figure 2-1.

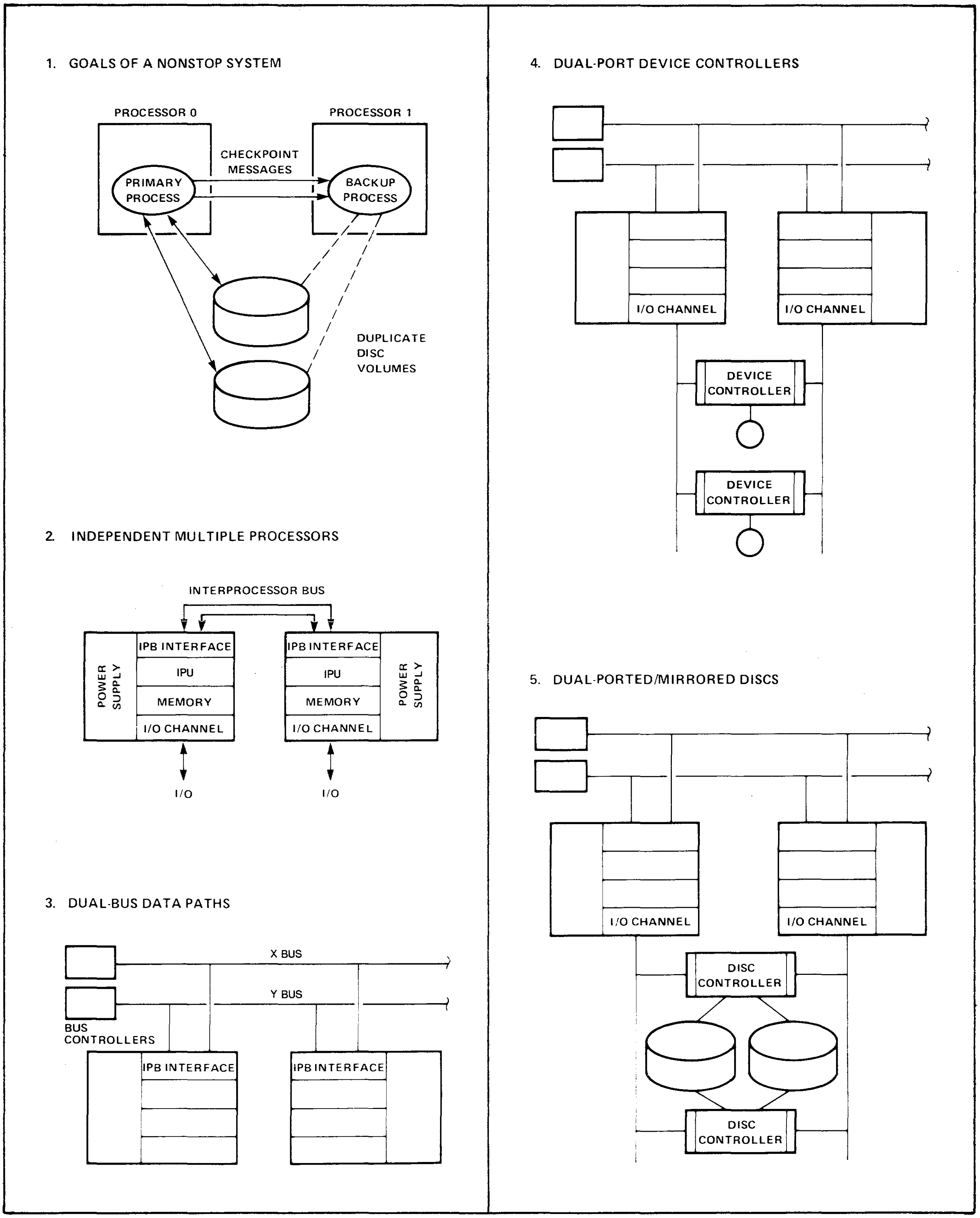


Figure 2-1. Elements of Hardware System Structure

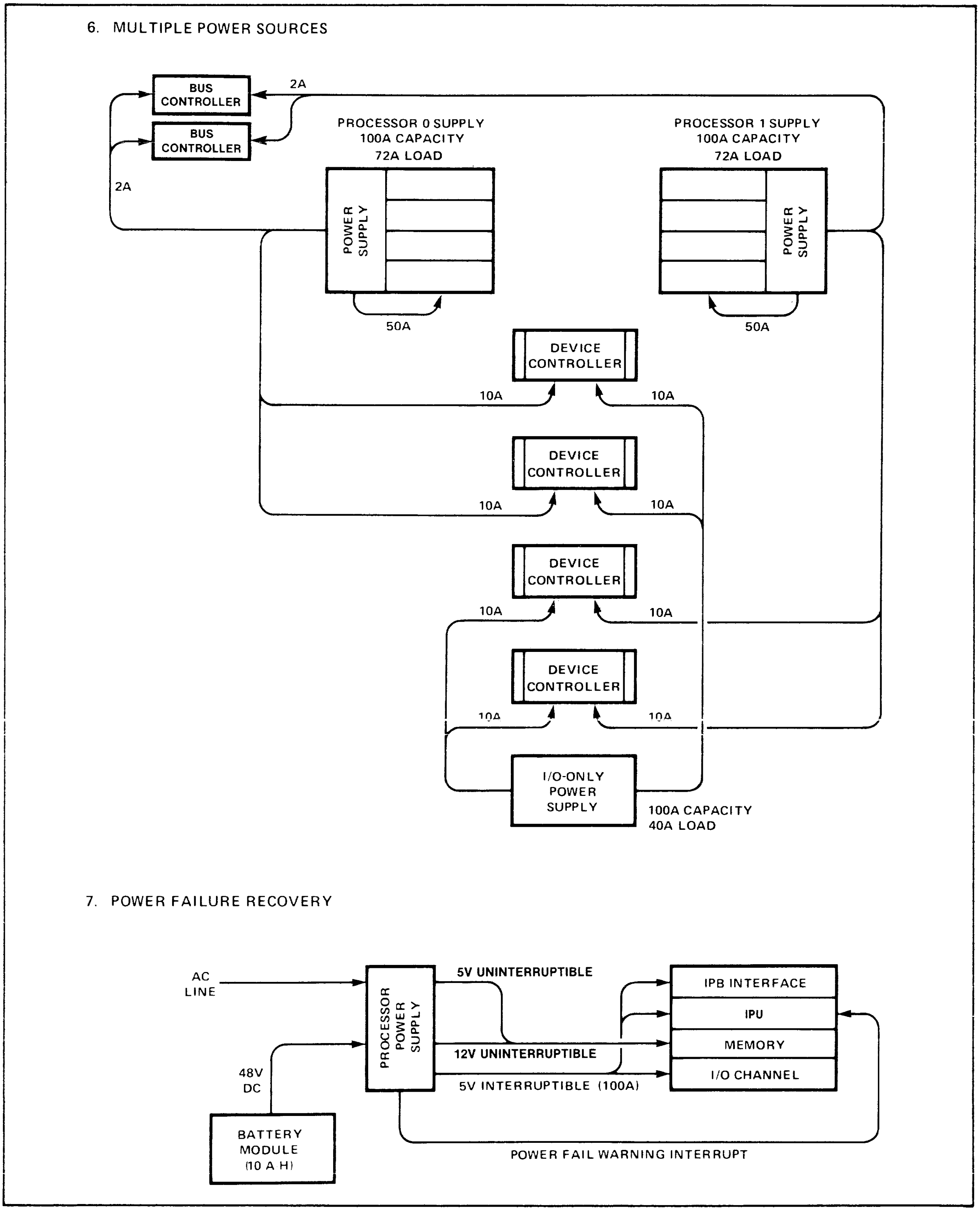


Figure 2-2. Power Distribution in the NonStop II System

I/O devices are interfaced to the i/o channels by dual-port controllers. Each dual-port controller is connected to the i/o channels of any two processor modules. Therefore, each i/o device can be controlled by either of two processor modules. However, in operation, an i/o device is controlled exclusively by one processor module until a failure occurs such that the processor module can no longer communicate with the i/o device. If such a failure occurs, the other processor module takes control of the i/o device.

Dual-Ported/Mirrored Discs

Because discs represent the most critical class of i/o devices, disc drives can also have dual ports. In combination with the dual ports on the disc controller, various configurations are possible, to meet any desired degree of failure tolerance. For example, connecting the dual ports of the controller to separate i/o channels provides for failure tolerance of the i/o channels. Connecting dual ports of a disc drive to separate controllers provides for failure tolerance of the disc controllers. Diagram 5 of Figure 2-1 shows an example of a fully mirrored, fully dual-ported configuration.

Multiple Power Sources

Power is distributed in the system in such a manner that each dual-port controller receives power from two sources. If a supply fails, causing a processor module to become inoperative, the alternate power supply can assume the full load.

As mentioned previously, there is a power supply associated with each processor, supplying power to that module. The processor consumes approximately half the power available from its supply; the remainder is available to help power the device controllers. In some cases, the power available from these supplies is sufficient to power all the device controllers; in other cases, a supplementary power supply for i/o only is necessary.

Diagram 6 in Figure 2-2 shows, in simplified form, the way in which power is distributed in the NonStop II system in order to achieve reliable power backup. The current values shown are mostly illustrative only; device controllers, for example, generally take much less than the 20 amperes assumed in this figure. Exact values and the adjustments required to achieve good power distribution are evaluated for each particular system by Tandem when the system is configured.

As shown, the two bus controllers require a total of about 4 amperes, 2 amperes each from the supplies associated with processor 0 and processor 1. (Bus controller power is always taken from the supplies for these particular cpu's.) The processor modules are assumed to require 50 amperes each; this depends on memory size and configuration. The output current capacity of the supplies is 100 amperes each (for the 5-volt interruptible supply, discussed later). Note that each device controller nominally receives one-half of its requirements (10 amperes) from each of two different power supplies. (In actuality, adjustments are made so that the cpu supply provides somewhat less than half the needed power, and the i/o supply provides slightly more than half.) Under the assumed conditions, then, each processor's power supply is loaded to 72 amperes, and the i/o-only supply is loaded to 40 amperes.

Now assume a failure in the processor 0 power supply. The processor 0 module goes down, but none of the device controllers or bus controllers is affected. The processor 1 power supply now delivers the full 4 amperes needed by the bus controllers (increasing its load to 74 amperes), and the i/o-only power supply delivers the full 20 amperes to each of the uppermost two device controllers (increasing its load to 60 amperes).

Likewise, if the i/o-only power supply should fail, the load on each processor's power supply would increase by 20 amperes (to 92), still within the 100-ampere capacity. Thus any single power supply failure can be compensated by increased loading on the remaining supplies. However, the failure of any two supplies cannot always be accommodated by the remaining ones.

Power Failure Recovery

Diagram 7 in Figure 2-2 illustrates the power failure recovery features that are incorporated into the internal circuits of each processor module. Note that memory is powered separately from the rest of the module, with its own 5-volt and 12-volt supplies; these are termed uninterruptible supplies, since they are maintained by battery power if an AC line failure occurs. Battery power then allows memory to retain its contents for 1.5 hours or more, depending on memory size and the charge state of the battery.

The interruptible 5-volt supply powers the remainder of the module. In order to allow the operating system to bring the central processing unit to an orderly halt, the power supply issues a special signal (power fail warning interrupt) when AC power is lost for more than 24 milliseconds. This signal gives a minimum of 5 milliseconds warning (depending on loading of the supply) that the 5-volt supply will be going down.

The system automatically restarts upon restoration of power, resuming execution of the processes that were in progress at the time of the power failure.

Other Failure-Tolerant Features

The ability of the Tandem computer system to provide an environment where applications can continue to run regardless of a module failure is due primarily to its unique NonStop features, described above. In addition to those unique features, the Tandem system also incorporates various other reliability features and certain standard design features currently found other systems. These include the following:

- The GUARDIAN operating system in each processor module saves the current operating state of its module in memory when a system-wide power failure occurs. For system power failures, the operating system automatically resumes all operations (including application programs) when power is restored.
- If an uncorrectable error occurs in memory, the operating system determines if the associated area is critical to system operation. If it is not, the area is flagged as bad and not used again until the memory is repaired. (Typically, the memory would be repaired during system preventive maintenance. However, the associated processor module could be taken off line to repair the memory, leaving the remainder of the system operable.) If the area is critical, the operating system halts execution in its processor.
- Critical portions of the operating system are main-memory resident; this assures their availability in the event that a virtual memory (disc) failure occurs.
- The cooling system for the computer is designed so that if a single failure occurs, ample cooling is still available.
- Any module in the system (i.e., processor, i/o controller, power supply, fan, etc.) can be removed from the system and replaced on-line without stopping operation of other system modules.
- Routing, sequence, and checksum words are generated by the transmitting processor module and checked by the receiving processor for every packet of 13 data words transferred over the interprocessor buses.
- A parity bit is associated with each 16-bit word transmitted over the i/o channels.
- An interval timer is provided; the operating system and the File System use the timers to notify the application program in the event a data transfer does not complete.
- Six error correction bits are generated and stored with each 16-bit word in the semiconductor memory; circuitry is provided to correct all single-bit errors and detect all double-bit errors.
- The addressing and count information associated with i/o transfers are kept in the controlling processor module. This prevents a controller from contaminating more than one processor module

because of a failure of an address or word count register.

- The File System protects against a failing input/output controller erroneously writing into memory (in the IOC table, either the device's count field is set to zero or its write-only bit is set).

- The memory mapping scheme provides separate system/user maps. Operating system data areas can be accessed only by operating system programs; application programs cannot inadvertently destroy the operating system.

- Two hardware modes of processor operation are provided: privileged and nonprivileged. Certain critical operations (such as accessing system tables from application programs or initiating input/output transfers) can be performed only while in privileged mode. Typically, only the GUARDIAN operating system runs in privileged mode; privileged operations are performed on behalf of application programs through calls to operating system procedures. Application programs running in nonprivileged mode are prevented from becoming privileged.

FUNDAMENTAL NonStop OPERATIONS

Hardware View of the Operating System

The GUARDIAN operating system oversees system operation. The operating system provides the multiprocessing (concurrent processing in separate processor modules) and multiprogramming (interleaved processing in one processor module) capabilities, and exercises control over the NonStop features of the Tandem system. A copy of the GUARDIAN operating system resides in each processor module (with the exception of system i/o processes, which only reside where they are needed).

The operating system automatically schedules application programs for execution according to an application-assigned priority, provides memory management functions (automatic overlaying, swapping to disc, and so on), and gives application programs the capability to start other programs executing in any processor module from any processor module.

Four major components of the GUARDIAN operating system that particularly relate to hardware operation are the Kernel, the Message System, system processes, and the File System. These are briefly discussed in the following paragraphs, in order to show the close interrelationship between the hardware and the software and to provide an understandable basis for the hardware functions described in this section of the manual.

KERNEL. The Kernel provides the capability for multiple processes to execute in parallel in a single processor module (the term "process" denotes an executing program). Among the Kernel's functions are scheduling processes for execution based on a run-time assigned execution priority and resolving system resource allocation conflicts.

MESSAGE SYSTEM. The Message System is actually part of the Kernel, but is listed separately here to emphasize its importance. It provides the means for processes (i.e., running programs) to communicate with each other. If two communicating processes are executing in different processor modules, the Message System automatically routes the communication over an interprocessor bus. The Message System makes use of both buses and guarantees delivery of a message even if one bus fails.

SYSTEM PROCESSES. The system processes are running programs that perform operating system related functions. These functions include loading programs into memory for execution, supporting virtual memory, and providing physical control of i/o devices.

FILE SYSTEM. Application processes do not interface directly with the Kernel, Message System, or system processes. Rather, they make use of the File System to communicate with other processes and with i/o devices. The File System provides a single interface between a user process and the outside world. Other processes and all i/o devices are accessed as "files" through a single set of system calls. Processes and i/o devices are referenced by means of preassigned, symbolic file names. The physical locations of i/o devices and of the processor modules where processes are executing are transparent to application programs.

Primary and Alternate I/O Paths

The use of dual-port controllers guarantees that a communication path exists to each i/o device even if a failure occurs. Each device has a "primary" path over which communication normally occurs. In addition, assuming the system is so configured, there exists an "alternate" path. See Figure 2-3.

If a failure occurs in a primary path, whether by cpu failure or i/o channel failure, the File System can reroute communication to the affected i/o device via the alternate path. Figure 2-3 assumes an i/o channel failure, requiring a switch from the primary device i/o process to the backup device i/o process.

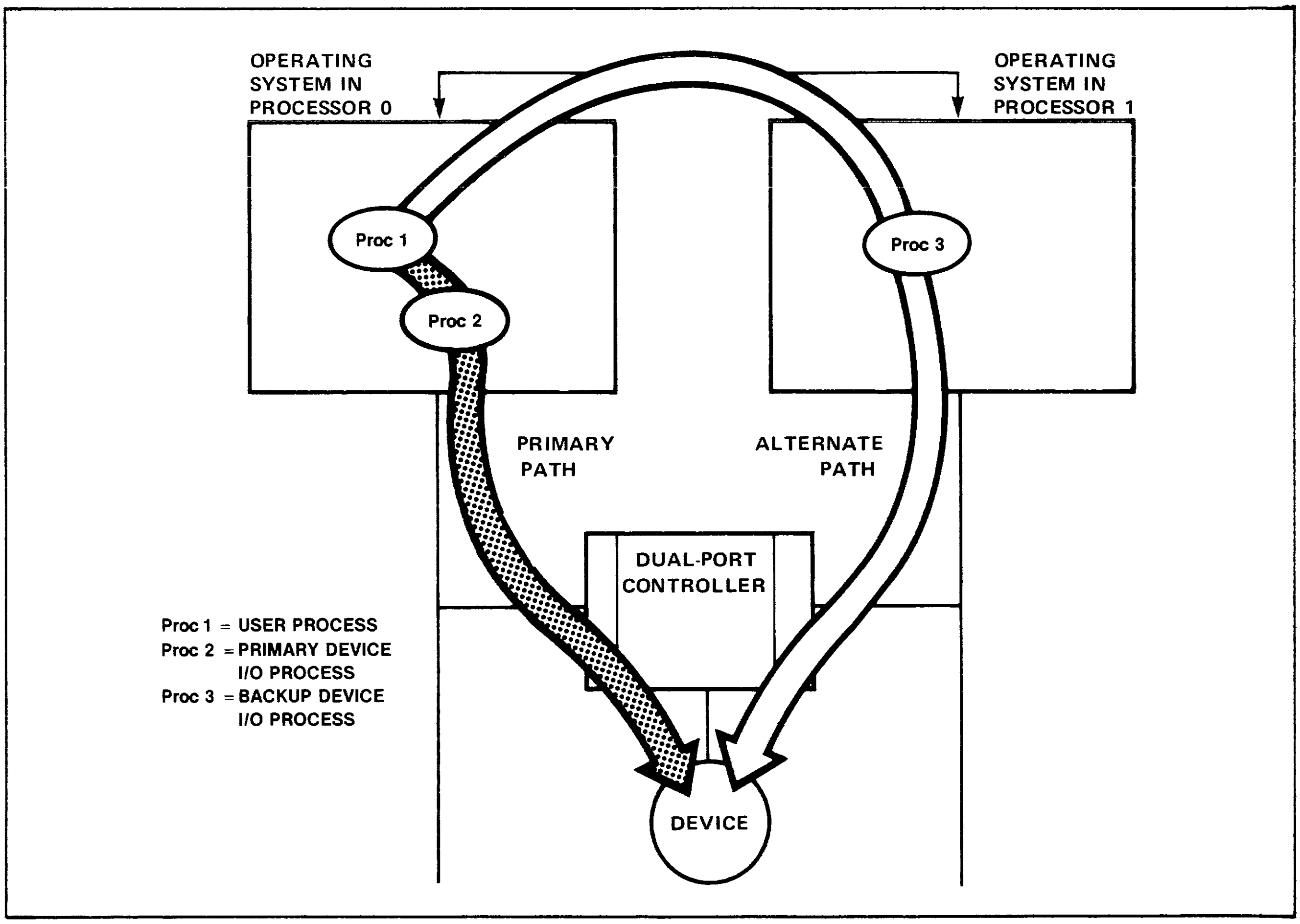


Figure 2-3. I/O Data Paths

Once the alternate path is put into use, all subsequent access to the i/o device is via that path. When the original primary path is restored, it may either become the alternate path or be restored as the primary path, depending upon system configuration choices.

The File System enables processes running in the same processor or in separate and redundant processor modules to communicate with each other and with any i/o device connected to the system. The hardware provides at least two paths to each processor module and to each i/o device. The operating system then guarantees that if at least a single path is available, communication will occur.

Processor Module Checking

The GUARDIAN operating system provides an additional function. Concurrent with application program execution, the Message System part of the GUARDIAN software in each processor module periodically transmits an "I'M ALIVE" message to all other processor modules in the system. (See Figure 2-4.) The Message System in each processor module, in turn, periodically checks for receipt of an "I'M ALIVE" message from every other processor module.

If the GUARDIAN operating system finds that more than one of these messages have not been received as expected (see Figure 2-5), it assumes that the nontransmitting processor module has failed. The operating system then sends a "CPU DOWN" message to interested system and application processes in its processor module. (This action occurs in every operational processor module.)

A NonStop Application

To show how the NonStop II system provides the means for creating a NonStop application, the following example is given. The example is illustrated in Figures 2-6 and 2-7.

The NonStop application consists of a "primary" application process running in processor module 0 (the primary process is designated A) and its "backup" process running in processor module 1 (the backup process is designated A'). The coded instructions for A and A' are identical. With the aid of the GUARDIAN software, each can determine whether it is the primary or the backup process, then perform its proper role.

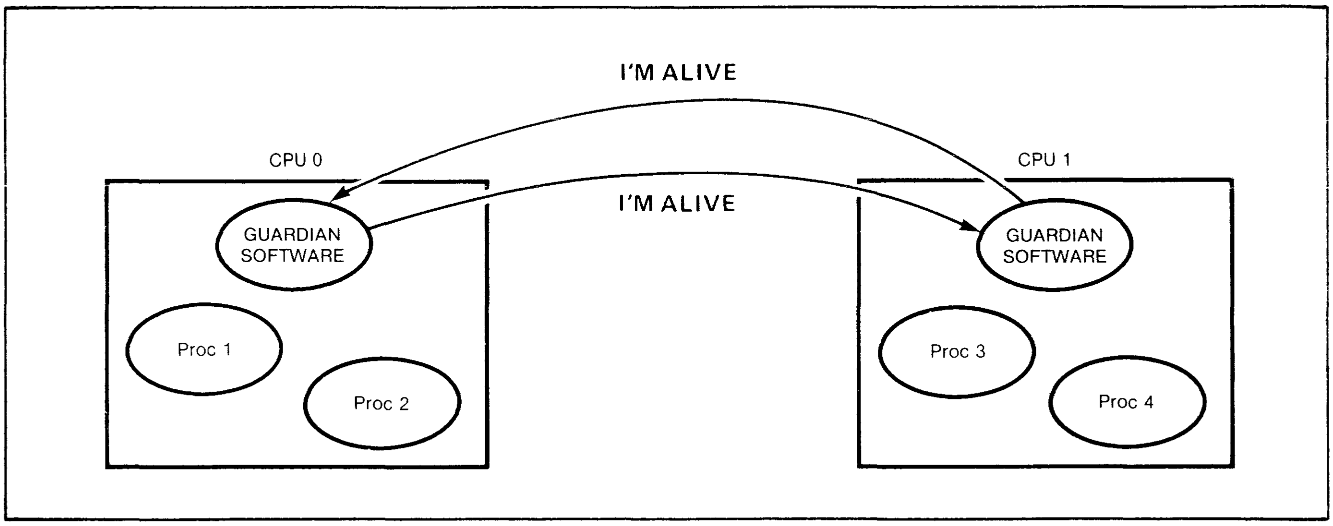


Figure 2-4. Processor Module Checking

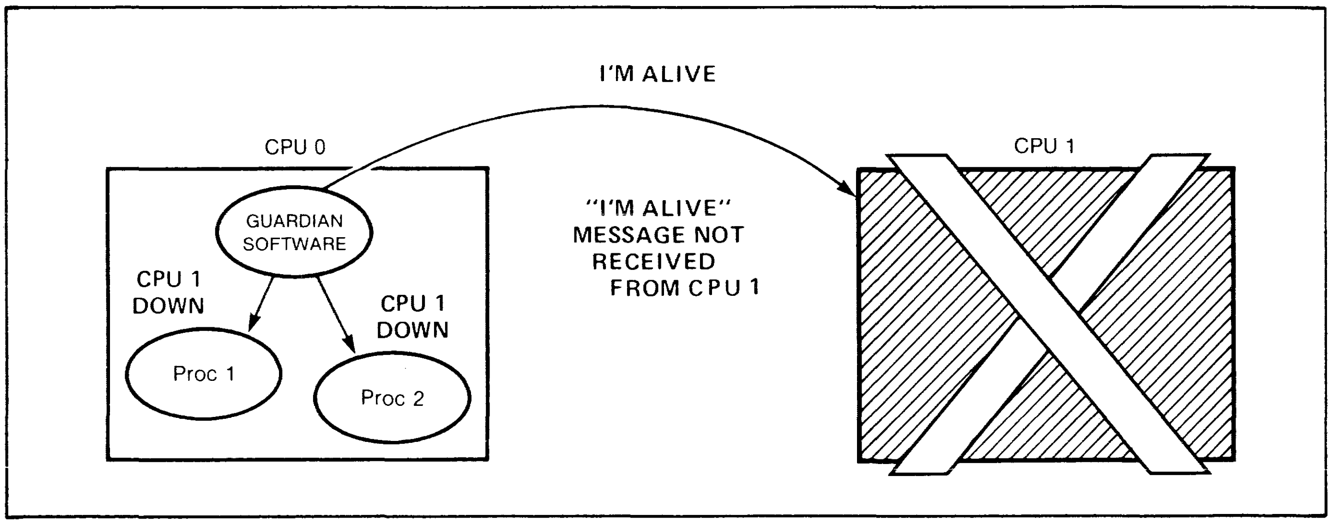
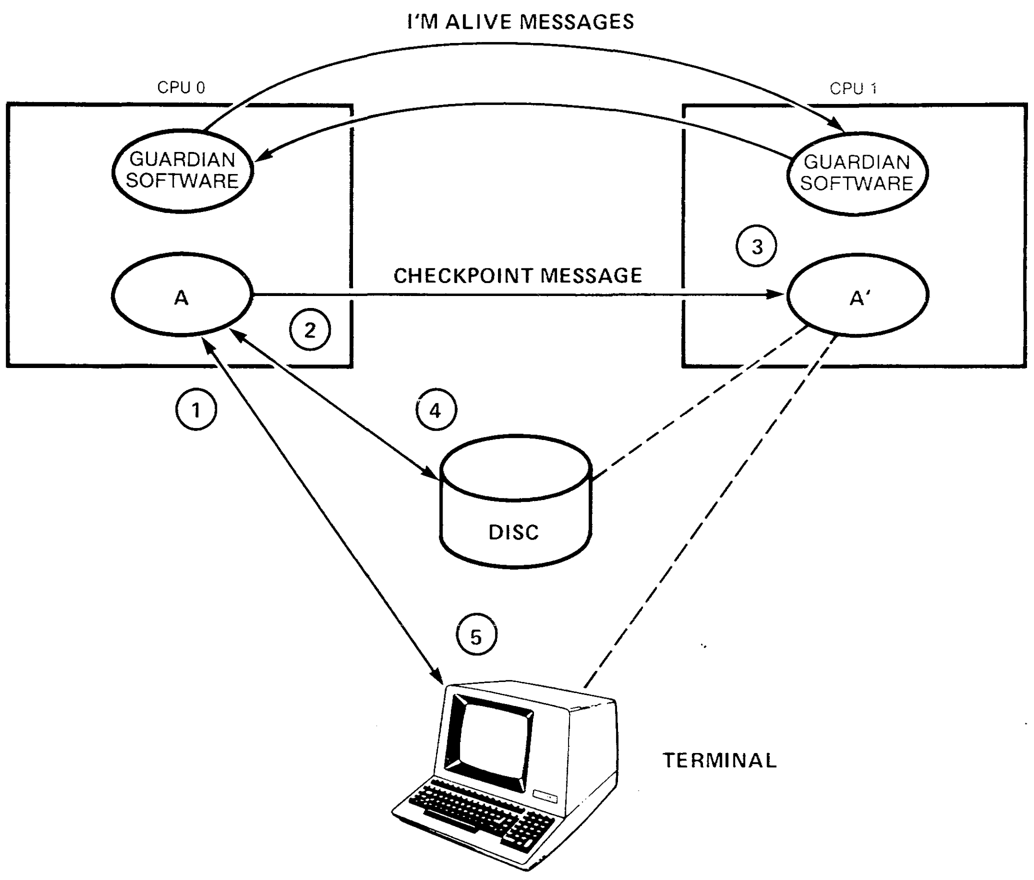


Figure 2-5. CPU Down Message



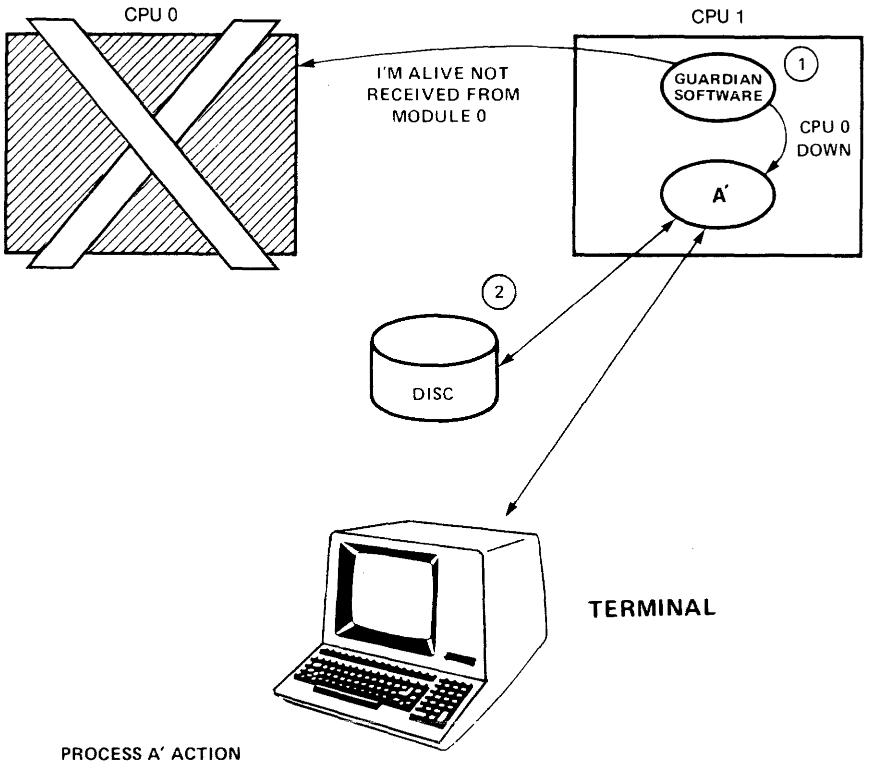
THE PROCESS: A

1. READ (a record from the terminal)
2. READ (a record from the disc)
3. WRITE (the updated disc record to A') Checkpoint
4. WRITE (the updated record to disc)
5. WRITE (the result on the terminal)

THE PROCESS: A'

READ (the checkpoint message from A)

Figure 2-6. NonStop Application



PROCESS A′ ACTION

(1) READ (the cpu 0 down message)

(2) WRITE (to the disc using the last checkpoint message to ensure update of the record)

Then continue with the same program as A.

```
READ (a record from the terminal)
READ (a record from the disc)
  .
  .
  .
  .
```

Except that there is no backup for A′ at this time, so no checkpoint message is sent.

Figure 2-7. Application Takeover by Backup

The "primary" process, while operable, performs ALL of the application's work. At critical points during each transaction cycle (such as prior to altering the contents of a disc file), the primary process sends a message, via the File System, to its backup process. These messages contain "checkpointing" information (such as an updated disc record) and keep the backup process up-to-date on the state of the application. All such messages are the result of checkpointing code that the programmer inserts in the application programs.

The "backup" process's responsibility, while the primary is operable, is to accept and process the checkpointing messages and be ready to take over the application if the primary process becomes inoperable.

If processor module 0 fails (see Figure 2-7), the GUARDIAN operating system in processor module 1 sends a "CPU 0 DOWN" message to the backup process A'. This is the signal for the backup process to take over the application's work. First, the backup process uses the latest checkpointing message (e.g., an updated disc record) to complete the transaction that the primary started just prior to its failure, leaving the application's data in the same state as if the primary had completed its last transaction successfully. At that point, the backup becomes the primary and continues with the application's work. (Note that there is no "backup" process at this time, therefore no checkpointing messages are sent).

When processor module 0 is reloaded, the GUARDIAN operating system sends a "CPU 0 UP" message to the current primary process (formerly the backup process). The primary process (through use of the GUARDIAN software) may then start a new backup process running in processor module 0. The primary also begins sending checkpointing information to the backup process. The application is now fully fault-tolerant once again.

PROCESSOR MODULE ORGANIZATION

Instruction Processing Unit

The instruction processing unit (IPU) has three functions: 1) to execute machine instructions, 2) to provide for the orderly interruption of a running process, and 3) to transfer data from the interprocessor buses into memory (this last item is invisible to the executing process and is handled entirely by the IPU's microprocessor).

A program's instructions reside in memory. In order to execute an instruction, it is first fetched from a location in memory determined by the address held in an IPU register; the register into which it is fetched is another IPU register. The instruction is decoded by the hardware to determine what sequence of microinstructions must be used to execute the instruction. During execution of the instruction, one or more memory transfers may occur, the IPU's scratchpad registers may be used to hold intermediate computations, and operands may be added to or deleted from the IPU's Register Stack.

While the current instruction is being executed, the next instruction in sequence is fetched from memory.

The instruction processing unit's microinstruction cycle time is 100 nanoseconds; microinstructions are 32 bits in length.

An IPU's basic instruction set consists of approximately 230 instructions. These include arithmetic operations (add, subtract, etc.), logical operations (and, or, exclusive or), bit shift and deposit, block (multiple-element) moves/compares/scans, procedure call and exit, interprocessor bus send, and the input/output instructions. All instructions are 16 bits in length.

Processor modules equipped with the Decimal Arithmetic option have an additional 14 instructions (six decimal arithmetic instructions are standard in all processors). These instructions operate on four-word operands and include add, subtract, multiply, divide, etc. (See Decimal Arithmetic Option headings in Section 3, "Instruction Set".) Modules equipped with the Floating Point option have an additional 41 instructions for doubleword and quadrupleword (extended) floating-point arithmetic and related operations. (See "Floating-Point Arithmetic" and "Extended Floating-Point Arithmetic" headings in Section 3.) With these options, a module has a total of approximately 280 instructions.

Two modes of process execution are provided: privileged and nonprivileged. A process executing in nonprivileged mode is not permitted to execute the instructions designated as privileged. Privileged instructions are associated with operations that, if performed incorrectly or inadvertently, could have an adverse affect on other processes or the operating system. These "privileged" operations include: interprocessor bus send, input/output, changes to map registers, execution of privileged procedures, and access to the system data segment. Normally, only the GUARDIAN operating system executes in privileged mode; application (user) processes execute in nonprivileged mode. Privileged operations are performed for nonprivileged processes through calls to operating system procedures. An attempt by a nonprivileged process to execute a privileged instruction causes the process to be trapped (interrupted).

The interrupt function provides for the orderly transfer of IPU control from an executing process to one of several routines in the operating system called interrupt handlers. This transfer of control is called an interrupt. Interrupts occur for several reasons. Among them are: data received over the interprocessor bus, completion of an i/o transfer, memory error, memory page absent, instruction failure (e.g., attempt by a nonprivileged process to execute a privileged instruction), and power failure.

Memory

Data is stored in memory in the form of 16-bit words. The maximum amount of memory addressable in a NonStop II system is sixteen megabytes (eight megawords). The maximum memory available for each processor is two megabytes. All accesses to memory are on word boundaries, even though the hardware provides element access to bytes, doublewords, and quadruplewords.

Addressing of processor memory is defined by two terms: logical addresses, which are relative to the start of code space or data space used by a single process; and physical addresses, the absolute addresses that define particular cells in physical memory.

A logical address most commonly consists of 16 bits; 16-bit addresses are capable of addressing a maximum of 65,536 words, which is defined as a "segment" of memory. Because a program consists of independently addressable areas (one or two code segments and one standard data segment), and each area can consist of 65,536 words, a single process can access up to 196,608 words (three segments) without using extended addressing. Extended addressing, which opens up the entire range of virtual memory, is considered at length under the heading "Memory Access".

A physical address consists of 23 bits; 23-bit addresses are capable of referencing any location in physical memory, and thus have a possible addressing range of sixteen megabytes. The conversion of the 16-bit logical address to a 23-bit physical address is accomplished through a mapping scheme. Sixteen maps are provided; each map consists of 64 entries, and is capable of completely defining one memory segment. Each map entry can be assigned to point to the start of a block of 1024 words of memory (called a page of memory).

The sixteen maps provide separate addressing of user code, user data, system code, system data, i/o buffers, and tables used in the implementation of the virtual memory addressing scheme. Some map entries are also used as IPU scratchpad registers and as a map entry cache to support virtual memory.

Several application processes and parts of the operating system can reside in memory concurrently. As each process is granted execution time in the processor, its logical memory space becomes part of the currently accessible portion of physical memory--that is, the process's segments become "mapped."

The data path between memory and other processor module functions is 16 bits wide. All data is verified for accuracy when it is read from memory. Six error correction bits are appended to each 16-bit word when it is stored. The use of the six error correction bits in the semiconductor memory permits the hardware to automatically correct all single-bit errors and to detect all double-bit errors. The detection of a memory error (whether correctable or uncorrectable) causes an interrupt to an operating system interrupt handler, which takes appropriate action.

Input/Output Channel

Each processor module has its own i/o channel that is capable of transferring data between i/o devices and memory at full memory speed. I/O operations, which are controlled by the operating system, are initiated by setting up an entry in a table in memory and then executing an EIO instruction. Once initiated, data transfer occurs concurrently with software process execution. The only time the software process is affected is when both the i/o channel and the IPU need to access memory at the same instant. If this occurs, the process's memory access is momentarily deferred while the i/o data is transferred between memory and the i/o channel (the action is invisible to the executing process). When the i/o operation completes, the currently executing process is interrupted, and control of the IPU is transferred to an operating system interrupt handler.

Each channel is capable of addressing 256 i/o devices, addressing each as a separate "subchannel." A single i/o operation is capable of transferring data in blocks of from one to 64k-1 bytes.

The table to control i/o transfers is called the I/O Control Table (IOC). Each processor module has its own IOC. (See Figure 2-8.) The IOC is known to the microcode and maintained by the operating system. The IOC table contains up to 256 entries, corresponding to the 256 possible devices (subchannels) on that processor's channel; each entry contains a buffer address (in one of the i/o buffer segments) and a count of the number of bytes to be transferred. The use of the IOC permits an i/o channel to run any number of devices (up to 256) concurrently while maintaining control on a device-by-device basis. When the number of bytes indicated in the IOC have been transferred, the device interrupts the currently executing process.

Data is buffered by each controller so that data is transferred in bursts through the channel at memory speed (the number of bytes in a "burst" depends upon the type of controller). Controllers are designed so that they signal the channel prior to actually emptying their buffers (during a write operation) or filling their buffers (during a read operation). This gives the channel ample time to respond, thereby providing a means to avoid data overrun. All 256 devices can be transferring simultaneously, with "bursts" from one device being interleaved with "bursts" from others, subject to i/o data rate configuration limits.

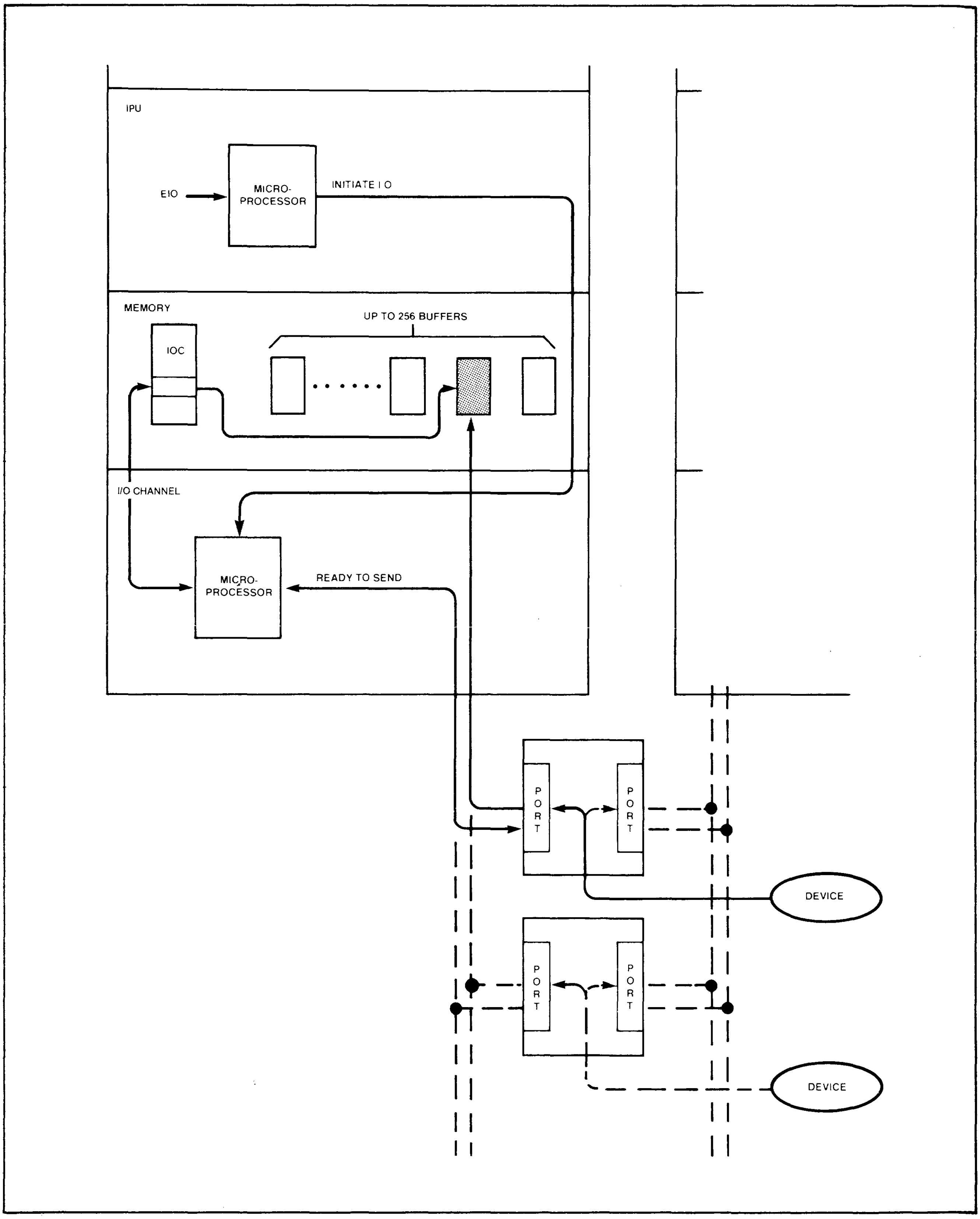


Figure 2-8. Input/Output Channel

Interprocessor Bus Interface

The NonStop II system has two interprocessor buses. Each bus functions independently of the other, transferring data from one processor module's memory to another processor module's memory. Both buses can be in use simultaneously. See Figure 2-9.

Data is transferred over each interprocessor bus at a rate of 13.33 megabytes per second. Each bus is capable of transferring data among all processor modules concurrently on a packet-multiplexed basis.

An interprocessor bus transfer involves two processor modules: the sender module and the receiver module. The transfer is initiated by the sender when a SEND instruction is executed. The receiver module checks the incoming packet for correct transmission, and directs the incoming data to a main memory buffer indicated by a firmware-known, software-maintained table.

The SEND instruction can transmit blocks of 1 to 64k-1 bytes to a designated processor module over one of the buses. Data is actually sent across a bus in "packets" of 16 words (a routing word, a sequence word, 13 data words, and a checksum word); each processor module contains two high-speed 16-word buffers (one for each bus) for receiving the incoming information. These buffers are designated INQ X (for the X bus) and INQ Y (for the Y bus). Transfers into the buffers occur simultaneously with IPU microprogram execution; when a buffer fills, the IPU microprogram is interrupted and a special microroutine moves the contents of the buffer into memory.

Each processor module's main memory contains a table called the Bus Receive Table (BRT). The BRT's are known by the firmware and are maintained by the operating system. They are used to direct the incoming bus data to a specified location in a processor module's memory. Each BRT contains 16 entries (corresponding to the 16 possible processor modules in a system); each entry specifies an expected packet sequence number, a buffer address where the incoming data is to be stored, and the number of bytes expected. When the expected number of bytes has been received, the currently executing process is interrupted, and the process for which the message is intended is notified.

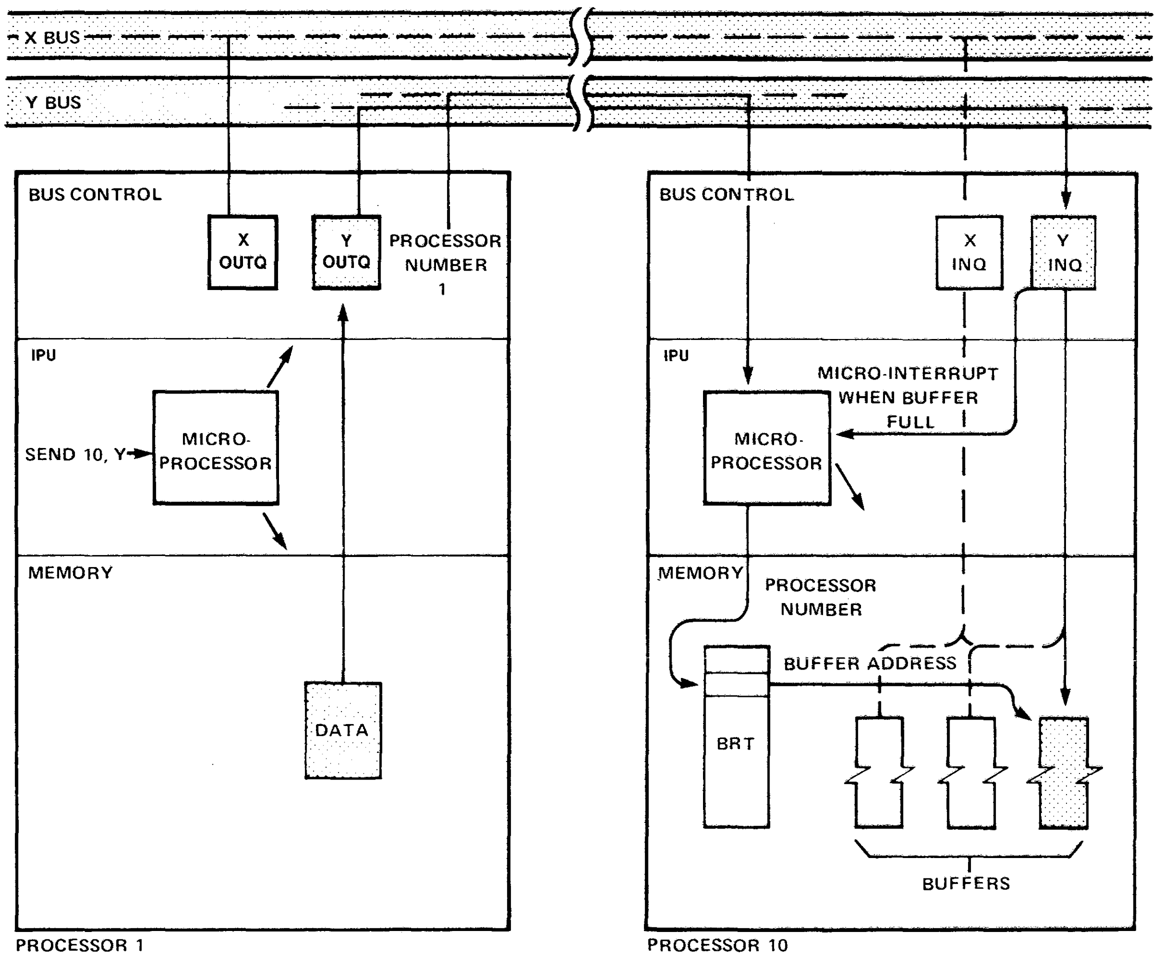


Figure 2-9. Interprocessor Bus Interface

Other Processor Components

In addition to the four main processor components just described--the IPU, memory, i/o channel, and interprocessor bus interface--each processor in a NonStop II system contains several other important components. These are discussed briefly in the following paragraphs. Figure 2-10 illustrates these components, showing their relationships to each other and to the four major components already discussed.

CLOCK GENERATOR. The clock generator is the main processor clock. It provides the synchronization of all hardware functions within the processor. The clock has a full cycle time of 100 nanoseconds, and a half-cycle time of 50 nanoseconds. Some clocking functions are performed on the half-cycle transition of the clock.

LOADABLE CONTROL STORE. The Loadable Control Store (LCS) contains microinstructions for use by the IPU. Each machine instruction causes the IPU to execute a specific set of microinstructions to implement the functions of that machine instruction. The Loadable Control Store cannot be written to by user programs, but it may be loaded with new versions of the system microcode and microcode options as they are purchased from or supplied by Tandem.

CONTROL PANEL. The control panel allows operators and maintenance personnel to interact directly with each NonStop II processor. The control panel can be used to reset a processor, cold load a processor, ready a processor for reload, and give visual indications of a processor's status. It also can be used to initiate some micro-diagnostics.

MEMORY CONTROL UNIT. The Memory Control Unit (MCU) provides access to memory for both the i/o channel and the IPU. The Memory Control Unit prioritizes memory requests; provides overlapped access, mapping of logical to physical memory, error control, and error reporting; and provides semiconductor memory refresh timing capability.

DIAGNOSTIC DATA TRANSCEIVER. The Diagnostic Data Transceiver (DDT) provides a communication path between a NonStop II processor and the Operations and Service Processor (OSP). Connected to the OSP through the Processor Maintenance Interface (PMI), it communicates at two distinct levels, as directed by the microprogram in the Loadable Control Store or by a running process. It can accept commands from the OSP to communicate with the operating system and diagnostics for operations or fault isolation. It can also report status conditions of the IPU, Memory Control Unit, i/o channel, and Loadable Control Store to the OSP.

PROCESSOR MAINTENANCE INTERFACE. The Processor Maintenance Interface (PMI) provides a common interface point for up to four processors to communicate with the Operations and Service Processor (OSP). If there are more than four processors in the system, additional PMI units are added, and the PMI's are connected together.

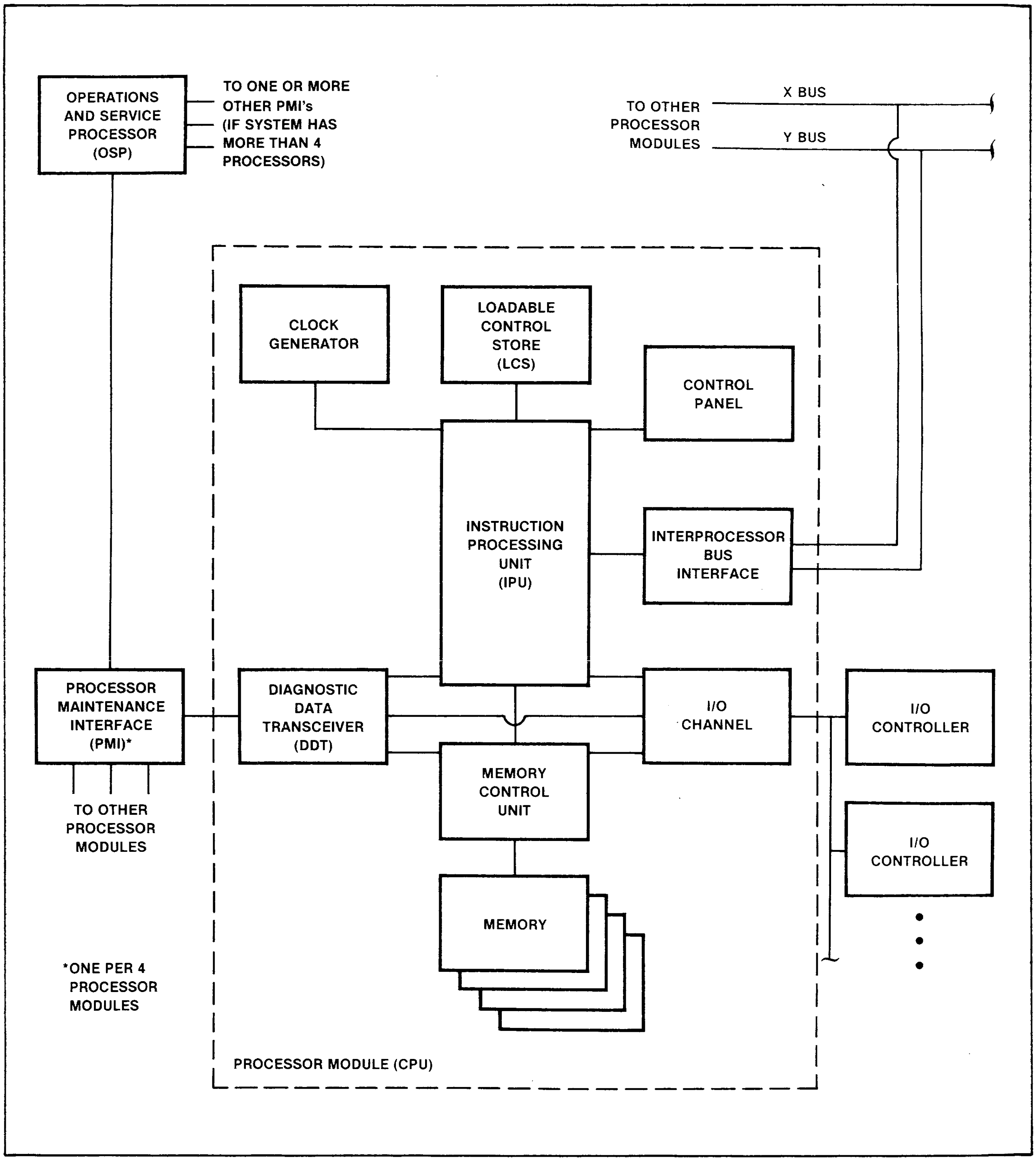

OPERATIONS AND SERVICE PROCESSOR (OSP)
TO ONE OR MORE OTHER PMI's (IF SYSTEM HAS MORE THAN 4 PROCESSORS)
TO OTHER PROCESSOR MODULES
X BUS
Y BUS
CLOCK GENERATOR
LOADABLE CONTROL STORE (LCS)
CONTROL PANEL
INSTRUCTION PROCESSING UNIT (IPU)
INTERPROCESSOR BUS INTERFACE
PROCESSOR MAINTENANCE INTERFACE (PMI)*
DIAGNOSTIC DATA TRANSCEIVER (DDT)
I/O CHANNEL
I/O CONTROLLER
I/O CONTROLLER
MEMORY CONTROL UNIT
MEMORY
TO OTHER PROCESSOR MODULES
*ONE PER 4 PROCESSOR MODULES
PROCESSOR MODULE (CPU)

Figure 2-10. Block Diagram of NonStop II Processor Hardware

The PMI provides switch functions and indicator lights showing processor and DDT status. In addition, it provides signal level conversion; it connects to the processors through differential signals, which it passes on to the OSP. The PMI may be used in a loopback mode to test the functionality of each processor's DDT. Finally, the PMI notifies the DDT of the speed at which the local or remote OSP is operating.

OPERATIONS AND SERVICE PROCESSOR (OSP)

The Operations and Service Processor (OSP) is the control center for the NonStop II system. Through the OSP, operators and maintenance personnel can communicate easily and flexibly with many low-level system functions, including all the essential functions of the control panel for each processor. Thus it enhances fault detection and isolation.

The OSP provides both local and remote operations and maintenance capabilities. As previously described, it is connected to each processor through the PMI and the DDT.

The OSP subsystem is made up of six components:

- Processor--The processor is the central part of the OSP subsystem. Most of the OSP functions are controlled by the processor. It provides intelligence and coordination of the subsystem. (The OSP processor is not to be confused with a processor module, or cpu.)

- Floppy Discs--The floppy discs are used to load the OSP operating system and diagnostics into the OSP processor. Two floppy discs are provided for failure tolerance.

- Switches and Indicators--The OSP switches and indicators provide access control and OSP functional indications.

- OSP Terminal--The OSP terminal, normally a 6520 terminal, provides an easy, flexible operations and maintenance interface with the OSP and the NonStop II system. Function keys are provided to allow fast interaction with the OSP.

- Modem--The modem included in the OSP subsystem allows communication with remote OSP's, remote terminals, and remote NonStop or NonStop II systems. Maintenance may be performed from all of these devices. Operations may be performed from a remote OSP or a remote 6520 terminal.

- Hard-Copy Printer--The optional 5508 hard-copy printer is provided for hard-copy logging of system console activity.

HOW THE HARDWARE EXECUTES PROGRAMS

Code and Data Separation

Programs executing as processes in memory are physically separated into two areas: code segments containing machine instructions and program constants, and data segments containing program variables. See Figure 2-11. The code segments of a process can be thought of as read-only storage, since no machine instructions can write into them.

Since code segments cannot be modified, they can be shared by a number of processes. In particular, operating system routines are shared by all application processes running in a given processor module (i.e., only one copy resides in memory).

Procedures

Programs are functionally separated into blocks of machine instructions called procedures. A procedure, like a program, has its own "local" data area (in the process's data segment). A procedure (i.e., the block of instructions that a procedure represents) is called into execution when a PCAL (procedure call) instruction is executed. The PCAL instruction saves the caller's environment and transfers control to the entry point instruction of the procedure.

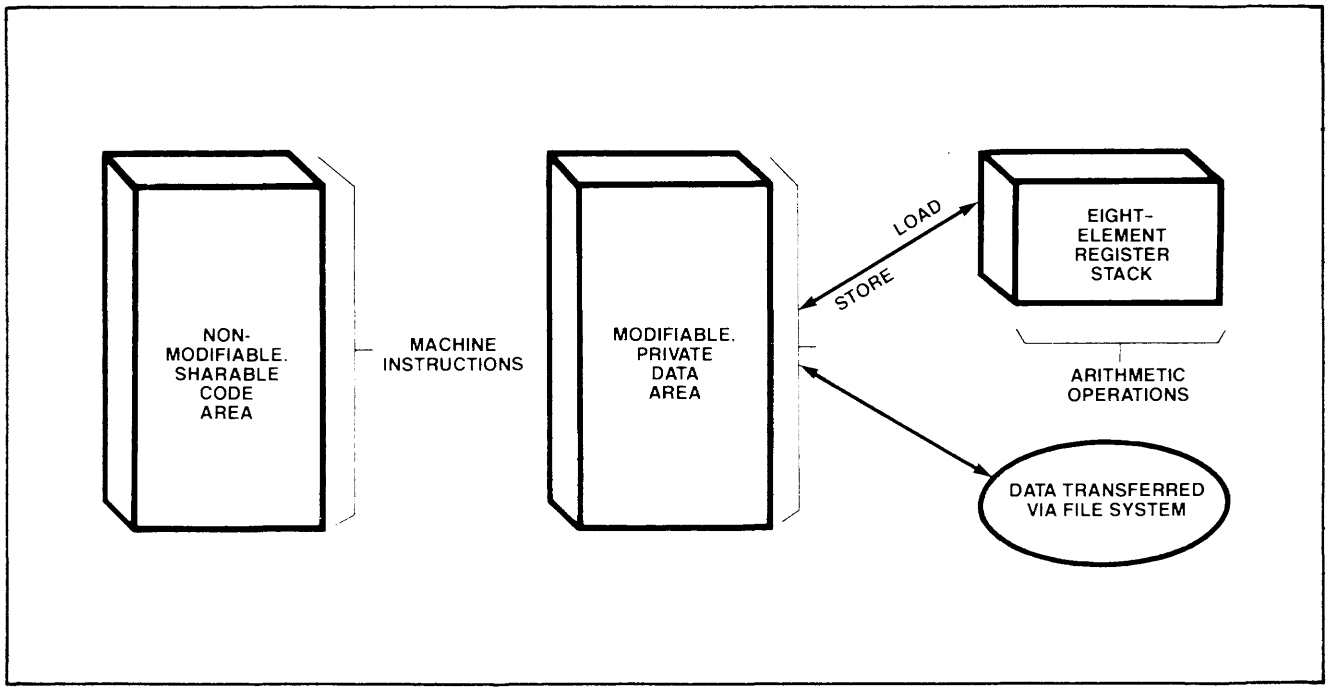

NON-MODIFIABLE. SHARABLE CODE AREA
MACHINE INSTRUCTIONS
MODIFIABLE, PRIVATE DATA AREA
LOAD
STORE
EIGHT-ELEMENT REGISTER STACK
ARITHMETIC OPERATIONS
DATA TRANSFERRED VIA FILE SYSTEM

Figure 2-11. Code and Data Separation

The procedure's instructions are then executed. The last instruction that a procedure executes is an EXIT instruction. The EXIT instruction restores the caller's environment and transfers control back to the caller's next instruction.

A procedure, while it executes, has its own local data area. This area is allocated for a procedure each time the procedure is called and is deallocated when the procedure exits (see "Memory Stack"). It can also access a shared global data area, which is accessible to all procedures of the process. The global data area and all the memory used for procedure local data areas are contained in the process's data segment.

Procedures can be written so that they can receive parameter information (arguments), perform computations using the parameters, then return results to the caller. (The machine instructions for passing parameters and returning results are generated automatically by compilers.)

Operating system functions (e.g., File System functions) are performed by calling procedures that are part of the operating system. A system procedure is called when an XCAL (external procedure call) instruction is executed. This is discussed later in this section under the heading "Calling External Procedures".

Memory Stack

Process segments are organized in main memory as stacks. A stack is a storage allocation method in which the last item (or block of items) added is the first item removed--like a stack of dishes. The "local" areas for procedures are blocks of data items in the memory stack. A procedure's local data is allocated in the memory stack only while it executes; after a procedure returns to the point where it was called, its data area is deallocated and may be used by another procedure called later. Therefore, the total amount of memory space required by a program is kept to a minimum.

Figure 2-12 illustrates the memory stack manipulations ("Data Area") during a sequence of procedure calls ("Code Area"). Sequence number (1) shows the memory stack when procedure A starts executing. At (2), a call to procedure C pushes C's parameters onto the stack (3), along with the link back to A. At (4), C begins to execute, using the stack for its local variables (5). Then a call to B (6, 7, 8) pushes B's parameters onto the stack, along with the link back to C, and B uses the stack for its local variables (9). Then, when B completes, it executes a return (10) back to C, deallocating its local variables, calling parameters, and return link from the stack. Procedure C, in turn, runs to completion and executes a return (11) back to A, deallocating its unneeded information from the stack. Procedure A continues its execution (12), with the stack back to the condition it was in prior to the calls; no unneeded data from these manipulations remains behind to waste memory.

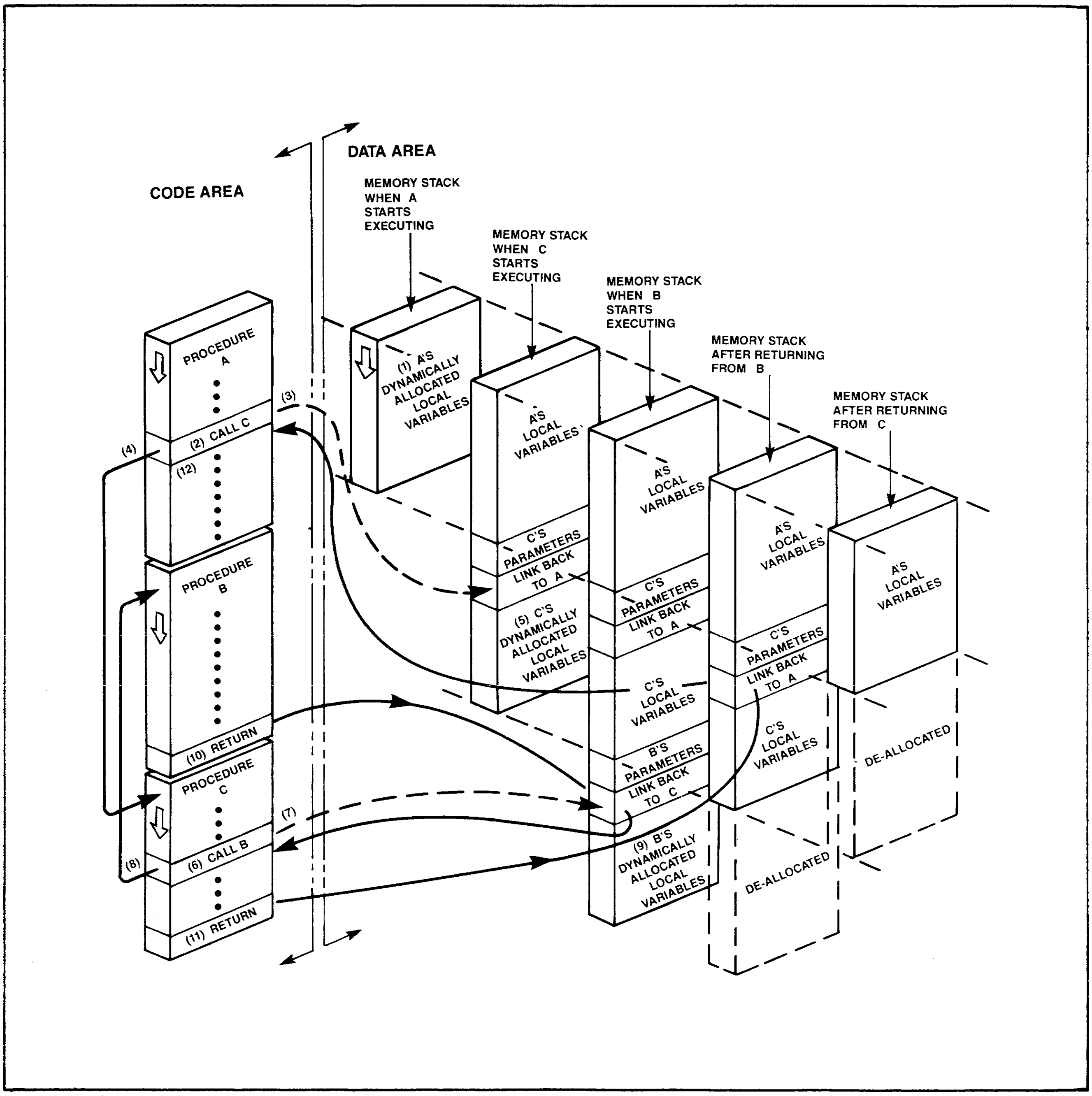

CODE AREA
DATA AREA
MEMORY STACK WHEN A STARTS EXECUTING
MEMORY STACK WHEN C STARTS EXECUTING
MEMORY STACK WHEN B STARTS EXECUTING
MEMORY STACK AFTER RETURNING FROM B
MEMORY STACK AFTER RETURNING FROM C
PROCEDURE A
(2) CALL C
(3)
(4)
(12)
PROCEDURE B
(10) RETURN
PROCEDURE C
(7)
(6) CALL B
(8)
(11) RETURN
(1) A'S DYNAMICALLY ALLOCATED LOCAL VARIABLES
A'S LOCAL VARIABLES
C'S PARAMETERS
LINK BACK TO A
(5) C'S DYNAMICALLY ALLOCATED LOCAL VARIABLES
C'S LOCAL VARIABLES
B'S PARAMETERS
LINK BACK TO C
(9) B'S DYNAMICALLY ALLOCATED LOCAL VARIABLES
DE-ALLOCATED

Figure 2-12. Memory Stack Operation

Register Stack

Each instruction processing unit contains a Register Stack consisting of eight separate registers. Each register stores one 16-bit word. The Register Stack provides a highly efficient means of executing arithmetic operations; operands are loaded onto the stack, arithmetic operations are performed, the operands are deleted, and a result is left on the stack. An add of two 16-bit numbers is illustrated in Figure 2-13.

The use of the Register Stack is transparent to programmers using Tandem-supplied languages. The language compilers automatically generate the machine instructions for efficiently using the Register Stack. The Transaction Application Language (TAL), however, does provide the capability of using the Register Stack explicitly.

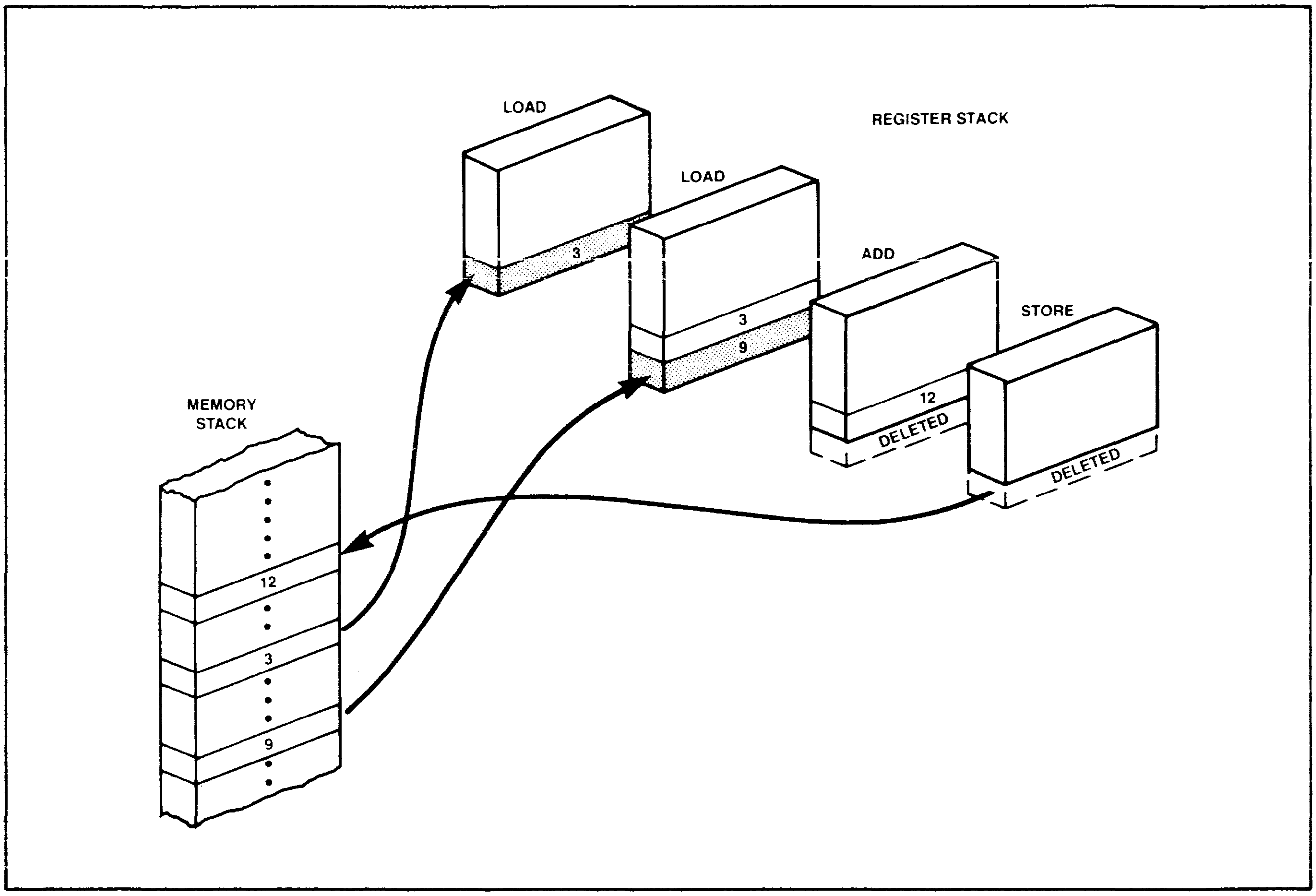


Figure 2-13. Register Stack Operation

DATA FORMATS

The basic unit of information in the NonStop II system is the 16-bit word. Individual access to and operations on single or multiple bits (bit fields) in a word, 8-bit bytes, 16-bit words, 32-bit doublewords, and 64-bit quadruplewords are possible. See Figure 2-14.

In this manual, a number surrounded by brackets is used to denote an individual element (i.e., word, doubleword, byte, or quadrupleword) in a block of elements:

```
block [element]
```

For example, to indicate the fourth element in a word block (beginning with element 0), the following notation is used:

```
WORD [3]
```

When referencing a block of words (or any elements), the first element is indicated by the element number that is the lowest numerically; the last element has the highest element number. The following notation is used to denote a block of elements:

```
block [first element:last element]
```

For example, to indicate the second through twentieth words in a block, the following notation is used:

```
WORD [1:19]
```

Words

The 16-bit word defines the machine instruction length and logical addressing range for the NonStop II system. The 16-bit word is the basic addressable unit stored in memory. The first word in each segment (i.e., code, data) of logical memory is addressed as WORD[0], the last addressable location is WORD[65,535]. This is shown in Figure 2-15.

The following instructions are provided for referencing words in logical memory:

LOAD:	Load word into Register Stack from data segment
STOR:	Store word from Register Stack into data segment
LWP:	Load Word into Register Stack from Program (code segment)
NSTO:	Non-destructive Store word from Register Stack into data segment
ADM:	Add word from Register Stack to word in Memory (data segment)
LDX:	Load Index Register from data segment

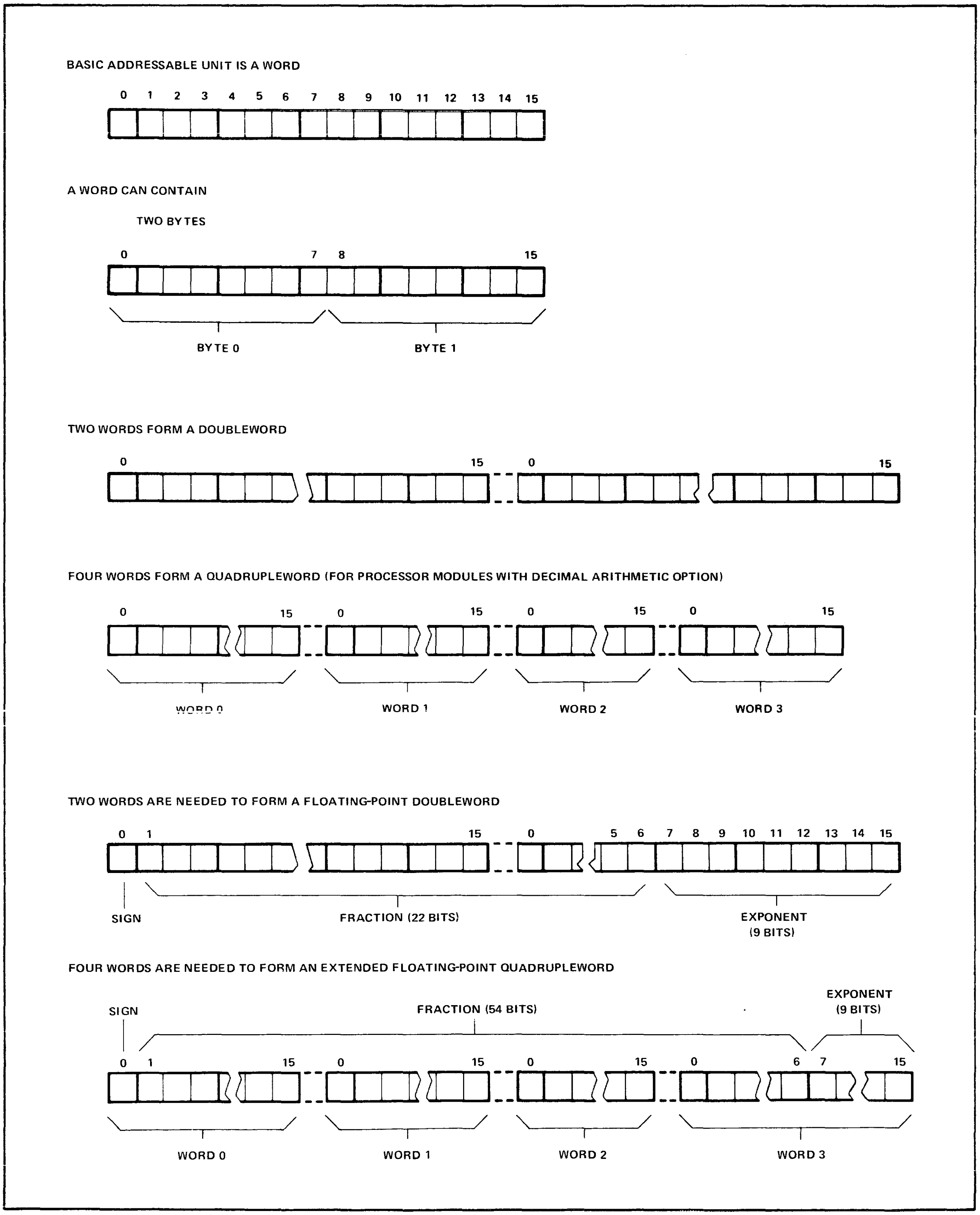


Figure 2-14. Data Formats

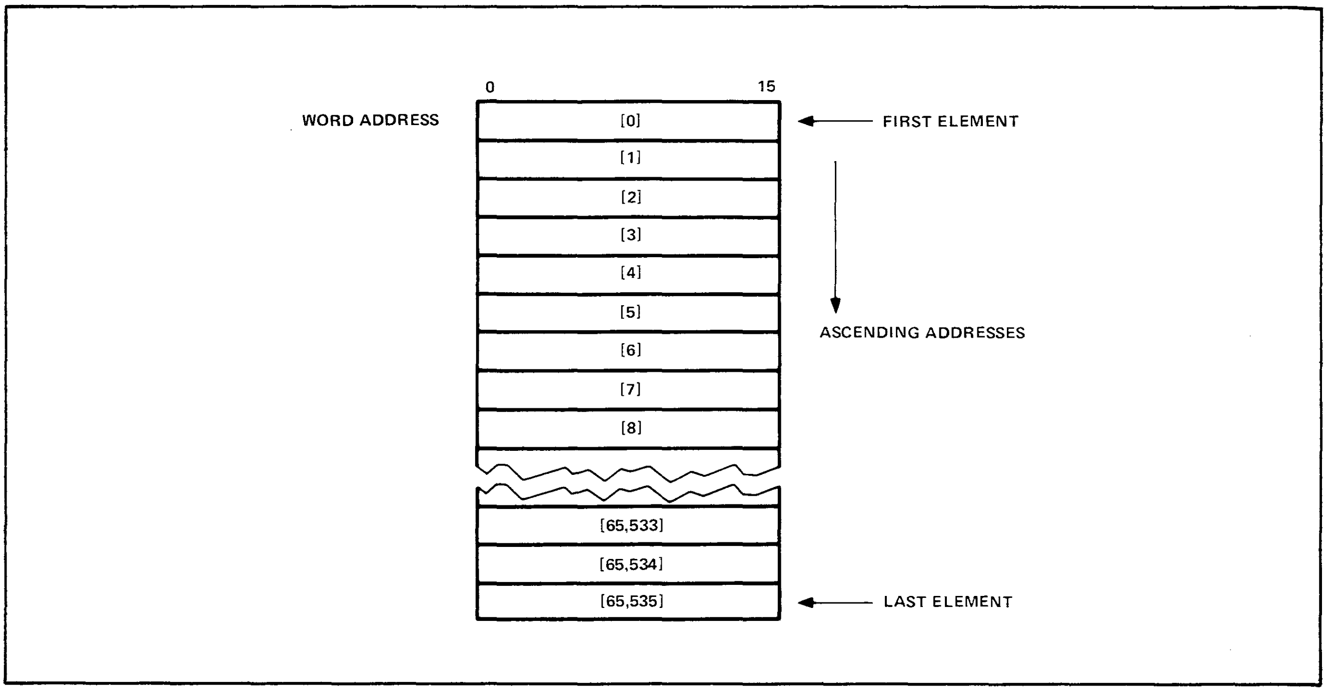


Figure 2-15. Word Addressing

Two instructions operate on blocks of words:

MOVW: Move Words from one memory location to another
COMW: Compare Words in one memory location with another

Bits

The individual bits in a word are numbered from zero (0) through fifteen (15), from left to right:

```
                          1 1 1 1 1 1
  WORD: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

The following notation is used in this manual (and in the TAL language) to describe bit fields:

WORD.<left bit:right bit>

For example, to indicate a field starting with bit four and extending through bit 15, the following notation would be used:

WORD.<4:15>

Or to indicate just bit 0 (zero) the following is used:

WORD.<0>

Bytes

The 16-bit word has the capability to store two bytes. The most significant byte in a word occupies WORD.<0:7> (left half); the least significant byte occupies WORD.<8:15>. The 16-bit address provides for element addressing of 65,536 bytes.

In the data segment, byte-addressable locations start at BYTE[0] and extend through BYTE[65,535]. Two bytes are stored per word; therefore the first 32,768 words of the data area (WORD[0:32,767]) can store 65,536 bytes. The upper half of the data segment, WORD[32,768:65,535], is not byte-addressable without the use of extended addressing.

In the code segment, byte addresses are computed by the hardware relative to whether the current setting of the P (for Program counter) Register is in the lower or the upper half of the code segment. Therefore, the entire code segment (WORD[0:65,535]) is byte-addressable, as explained in the description of the LBP instruction in Section 3.

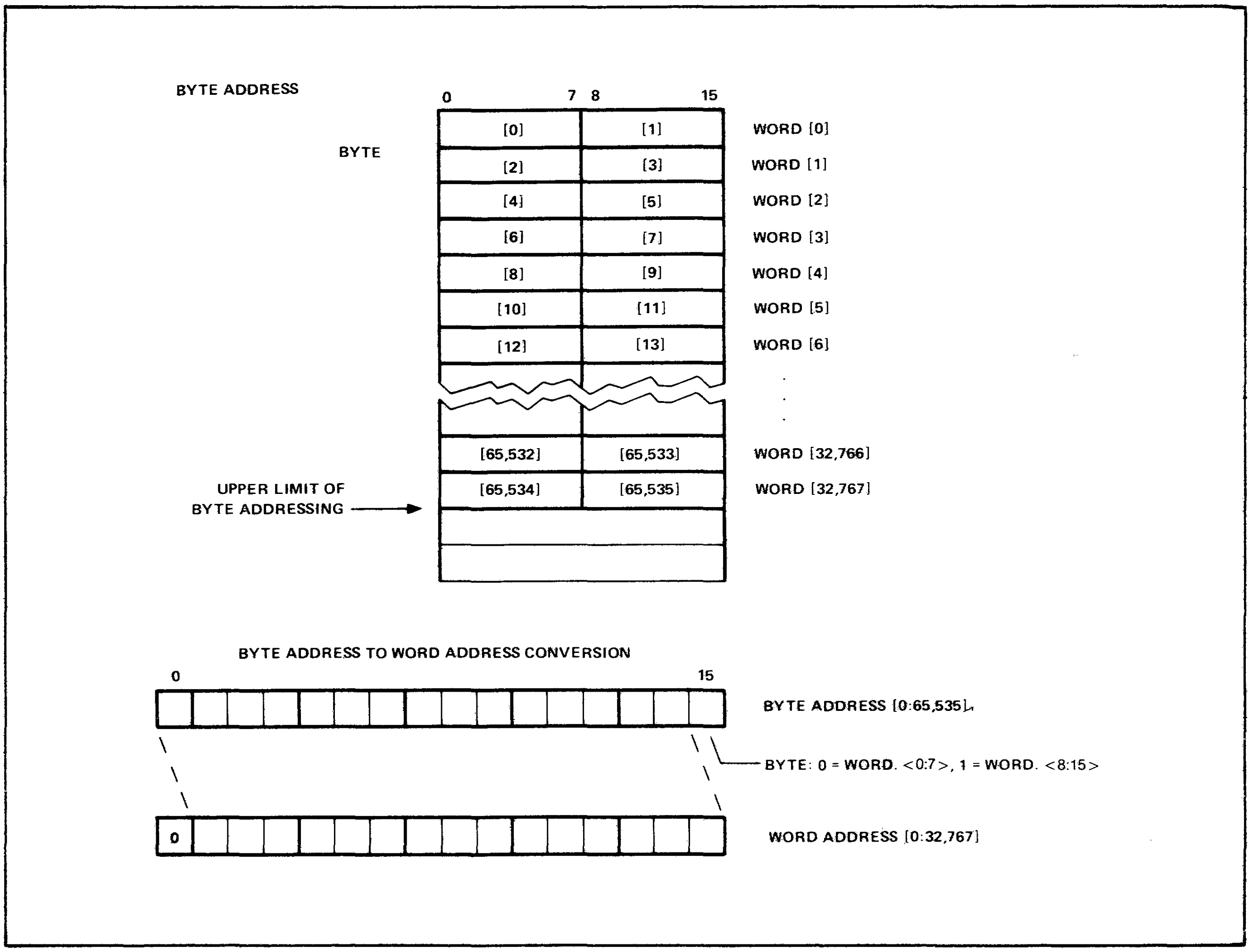


Figure 2-16. Byte Addressing

Data Formats

The IPU converts a byte address to a word address and bit field in that word as shown in Figure 2-16. That is, bit 15 of the byte address is extracted and used to specify left (0) or right (1) byte; the remaining 15 bits are logically shifted right by one bit to form the word address. In addressing a byte in the code segment, bit 0 of the word address is copied from bit 0 of the P Register.

The following instructions are provided for referencing bytes in logical memory:

LDB:	Load Byte into Register Stack from data segment
STB:	Store Byte from Register Stack into data segment
LBP:	Load Byte into Register Stack from Program (code segment)

Four instructions operate on blocks of bytes:

MOVB:	Move Bytes from one memory location to another
COMB:	Compare Bytes in one memory location with another
SBW:	Scan a block of Bytes While a test character is encountered
SBU:	Scan a block of Bytes Until a test character is encountered

Doublewords

Two 16-bit words can be accessed as a single 32-bit element. The hardware provides element access to doublewords in the data area (the software simulates doubleword access of elements in the code area). Doubleword elements are addressed on word boundaries; therefore doubleword addressing is permitted in all of the data area.

Two instructions are provided for referencing doublewords in logical memory:

LDD:	Load Doubleword into Register Stack from data segment
STD:	Store Doubleword from Register Stack into data segment

Quadruplewords

Four 16-bit words can be accessed as a single 64-bit element. The hardware provides element access to quadruplewords in the data segment (the software simulates quadrupleword access of elements in the code segment). Quadrupleword elements are addressed on word boundaries; therefore quadrupleword addressing is permitted in all of the data segment.

Two instructions are provided for referencing quadruplewords in the data segment:

QLD:	Quadrupleword Load into Register Stack from data segment
QST:	Quadrupleword Store from Register Stack into data segment

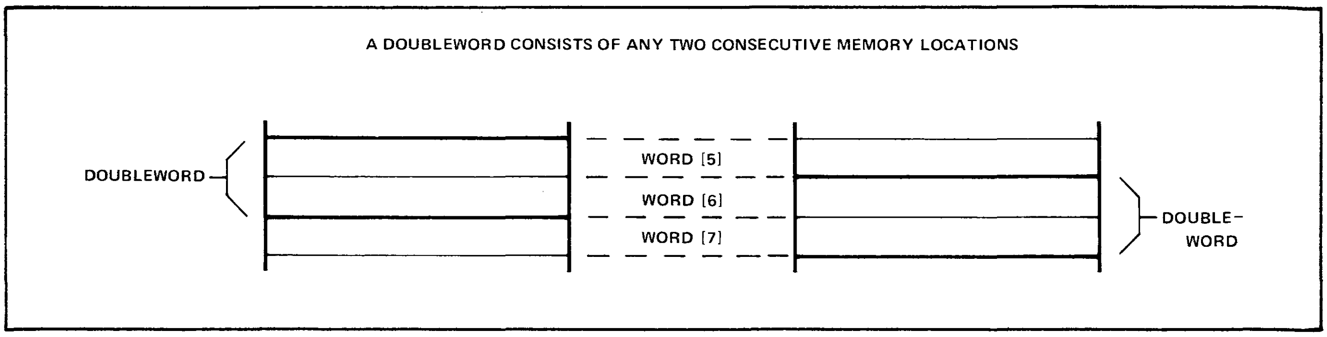

A DOUBLEWORD CONSISTS OF ANY TWO CONSECUTIVE MEMORY LOCATIONS
DOUBLEWORD
WORD [5]
WORD [6]
WORD [7]
DOUBLE-WORD

Figure 2-17. Doubleword Addressing

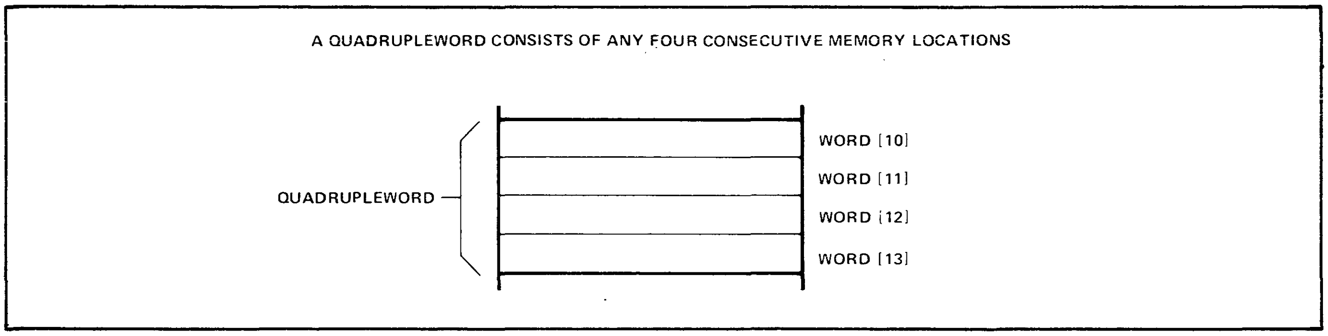

A QUADRUPLEWORD CONSISTS OF ANY FOUR CONSECUTIVE MEMORY LOCATIONS
QUADRUPLEWORD
WORD [10]
WORD [11]
WORD [12]
WORD [13]

Figure 2-18. Quadrupleword Addressing

NUMBER REPRESENTATION

The system hardware provides arithmetic on both signed and unsigned numbers. Signed numbers are characterized by being able to represent both positive and negative values; unsigned numbers represent only positive values. Signed numbers are represented in 16 bits (a word), 32 bits (doubleword), or 64 bits (quadrupleword). Representation of unsigned numbers is restricted to 8- and 16-bit quantities.

Positive values are represented in true binary notation. Negative values are represented in two's-complement notation with the sign bit of the most significant word set to one (i.e., WORD[0].<0>). The two's complement of a number is obtained by inverting each bit position in the number then adding a one. For example, in 16 bits, the number 2 is represented:

```
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

and the number -2 is represented:

```
        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

The representable range of numbers is determined by the sizes of operands (i.e., word, doubleword, and quadrupleword).

Single Word

Single-word operands can represent signed numbers in the range of

-32,768 to +32,767

and unsigned numbers in the range of

0 to +65,535.

Whether a word operand is treated as a signed or an unsigned value is determined by the instruction used when a calculation is performed. Signed arithmetic is indicated by the execution of "integer" instructions. The integer instructions are:

IADD:	Integer Add
ISUB:	Integer Subtract
IMPY:	Integer Multiply
IDIV:	Integer Divide
INEG:	Integer Negate (two's complement)
ICMP:	Integer Compare
ADDI:	(integer) Add Immediate
CMPI:	(integer) Compare Immediate
ADM:	(integer) Add to Memory

Unsigned arithmetic is indicated by the execution of "logical" instructions. The logical instructions are:

LADD:	Logical Add
LSUB:	Logical Subtract
LMPY:	Logical Multiply (returns doubleword product)
LDIV:	Logical Divide (returns 2-word quotient/remainder)
LNEG:	Logical Negate (one's complement)
LCMP:	Logical Compare
LADI:	Logical Add Immediate

Doubleword

Doubleword operands can represent signed numbers in the range of

-2,147,483,648 to +2,147,483,647.

Ten instructions perform integer arithmetic on doubleword operands. They are:

DADD:	Doubleword Add
DSUB:	Doubleword Subtract
DMPY:	Doubleword Multiply
DDIV:	Doubleword Divide
DNEG:	Doubleword Negate (two's complement)
DCMP:	Doubleword Compare
DTST:	Doubleword Test
MOND:	(load) Minus One in Doubleword form
ZERD:	(load) Zero in Doubleword form
ONED:	(load) One in Doubleword form

Byte

Byte operands represent unsigned values in the range of

0 to +255

This, of course, includes the ASCII character set. Byte operands are treated as the right half of word operands (i.e., WORD.<8:15>) when arithmetic is performed (the left half of the word is assumed to be zero).

There is one instruction for testing the class (i.e., ASCII alpha, ASCII numeric, and ASCII special) of a byte operand. It is:

BTST:	Byte Test

Quadrupleword (Decimal Arithmetic Option)

Quadrupleword operands for decimal arithmetic can represent 19-digit numbers in the range of

-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

Six instructions perform integer arithmetic on quadrupleword operands:

QADD:	Quadrupleword Add
QSUB:	Quadrupleword Subtract
*QMPY:	Quadrupleword Multiply
*QDIV:	Quadrupleword Divide
*QNEG:	Quadrupleword Negate
*QCMP:	Quadrupleword Compare

Three instructions are provided for scaling (i.e, normalizing) and rounding quadrupleword operands:

QUP:	Quadrupleword Scale Up
QDWN:	Quadrupleword Scale Down
*QRND:	Quadrupleword Round

Nine instructions are provided for converting operands between quadrupleword and other data formats:

*CQI:	Convert Quadrupleword to Singleword Integer
*CQL:	Convert Quadrupleword to Singleword Logical
*CQD:	Convert Quadrupleword to Doubleword
*CQA:	Convert Quadrupleword to ASCII
*CIQ:	Convert Singleword Integer to Quadrupleword
*CLQ:	Convert Singleword Logical to Quadrupleword
*CDQ:	Convert Doubleword to Quadrupleword
*CAQ:	Convert ASCII to Quadrupleword
*CAQV:	Convert ASCII to Quadrupleword with Initial Value

The asterisk indicates "optional instruction." Quadrupleword instructions not marked with an asterisk are part of the basic instruction set.

Floating-Point and Extended Floating-Point

The fraction of the floating-point numbers is always normalized, to be greater than or equal to 1 and less than 2. The high-order integer bit is therefore dropped and assumed to have the value of 1. For all calculations the sign is moved and the bit inserted. The integer plus 22 fraction bits of a floating-point number are equivalent to 6.9 decimal digits; the 55 bits for an extended floating-point number is equivalent to 16.5 decimal digits. If the value of the number to be represented is zero, the sign is 0, the fraction is 0, and the exponent is 0.

The fraction of the floating-point number is a binary number with the binary point always between the assumed integer bit and the high-order fraction bit. The exponent part of the number, bits 7 through 15 of the low-order word (see Figure 2-14), indicates the power of 2 multiplied by 1 + the fraction. This field may contain values from 0 to 511. In order to express numbers of both large and small absolute magnitude, the exponent is expressed as an excess-256 value. That is, 256 is added to the actual exponent of the number before it is stored. The exponent range is therefore actually -256 through +255.

The sign of the floating-point number is explicitly stated in the high-order bit (i.e., signed magnitude representation). A 0 is positive and a 1 is negative.

The absolute-value range of floating-point numbers is:

$$\left\{\begin{array}{c} \pm 2^{-256} \\ (\text{approx. } \pm 8.62 * 10^{-78}) \end{array}\right\} \text{ to } \left\{\begin{array}{c} \pm (1 - 2^{-23}) * 2^{256} \\ (\text{approx. } \pm 1.16 * 10^{77}) \end{array}\right\}$$

For extended floating-point numbers, the range is the same; only the precision is increased:

$$\begin{array}{c} \pm 2^{-256} \\ (\text{approx. } \pm 8.62 * 10^{-78}) \end{array} \text{ to } \begin{array}{c} \pm (1 - 2^{-55}) * 2^{256} \\ (\text{approx. } \pm 1.16 * 10^{77}) \end{array}$$

Arithmetic

The result of integer arithmetic (IADD, ISUB, IMPY, DADD, DSUB, DMPY, QADD, QSUB) must be representable within the number of bits comprising the operand minus the sign bit (e.g., 15 bits for a word operand, 31 bits for a doubleword operand). If the result cannot be represented, an arithmetic overflow condition occurs, and no part of the results on the stack can be assumed valid. When an overflow occurs, the hardware Overflow indicator sets and (if enabled) an interrupt to the operating system Overflow interrupt handler occurs. An overflow condition also occurs if a divide operation is attempted with a divisor of zero.

The results obtained from a logical add and subtract (LADD and LSUB) are identical to that obtained from integer add and subtract except that logical add and subtract do not set the Overflow indicator. The 16-bit result, the condition code setting, and the Carry indicator setting are the same. Logical divide (LDIV), however, sets the Overflow indicator if the quotient cannot be represented in 16 bits.

In addition to the Overflow indicator, two other hardware indicators are subject to change as the result of an arithmetic operation. They are:

- Condition Code (CC)--generally, indicates if the result of a computation was a negative value, zero, or a positive value. (The condition code can be tested by one of the branch-on-condition-code instructions and program execution sequence altered accordingly.)
- Carry--indicates that a carry out of the high-order bit position occurred.

For floating-point and extended floating-point arithmetic, the Overflow indicator is set if the exponent becomes either greater than +255 (exponent overflow) or less than -256 (exponent underflow) in trying to represent the normalized result of some operation. If the divisor in a divide operation is zero, the Overflow indicator is also set. If any conversion instruction causes a numeric overflow ("illegal conversion"), the Overflow indicator is set and the result (including Condition Code) is undefined. If the result of some operation has a zero fraction and nonzero exponent or sign, the value is forced to zero.

Table 2-1 defines termination conditions for various floating-point arithmetic errors. (For further explanation of the condition code CC, refer to the "Environment Register" section later in this manual.)

Table 2-1. Floating-Point Error Terminations

Condition	Overflow	CC	Result
Exponent Overflow	1	00	Calculated result with error truncated
Exponent Underflow	1	10	Calculated result with error truncated
Divide by Zero	1	01	Calculated result with error truncated
Illegal Conversion	1	xx	Undefined

PROGRAM ENVIRONMENT

A program executing as a process in a processor module consists of instruction codes in a CODE SEGMENT in memory that manipulate variable data in a separate DATA SEGMENT in memory. The IPU's eight-element REGISTER STACK is used to perform arithmetic operations and memory indexing. The instruction-to-instruction environment of a program is maintained in the IPU's ENVIRONMENT REGISTER. Programs themselves are separated into functional blocks of instructions called PROCEDURES.

These fundamental elements of the program environment are illustrated in Figure 2-19 and are discussed under separate subheadings below.

Code Segment

Information in a code segment consists of instruction codes and program constants. Although it is possible to address the code segments (via extended addressing or the LBP, LWP, or LWUC instruction), only read access is permitted; a write access attempt results in an address trap. Therefore the code segments cannot be modified during execution.

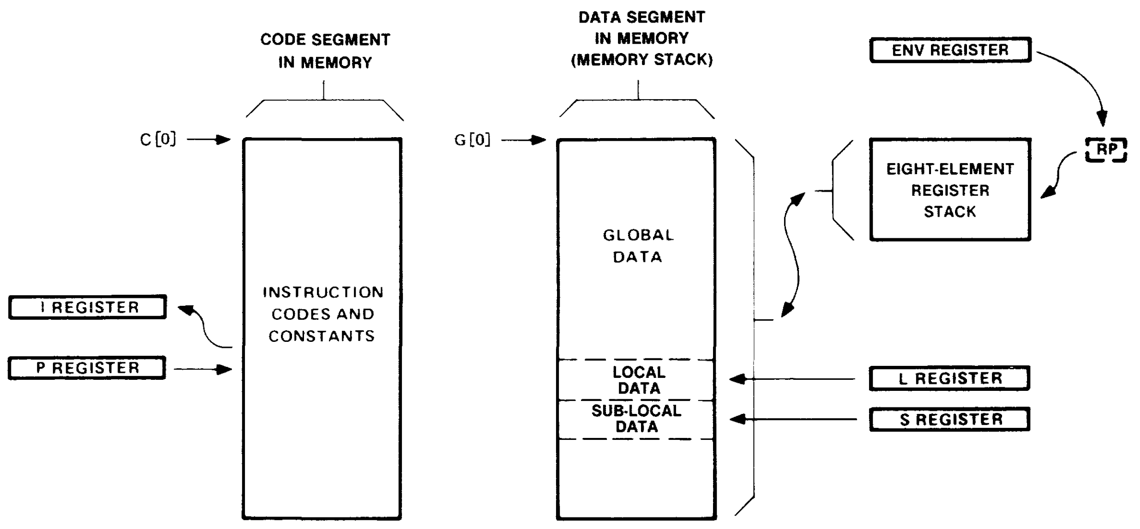
A given process may have two code segments: the User Code segment (standard for every process), and the User Library Code segment (optionally requested during compilation or at run time). External procedure calls allow the process to execute in either segment.

A code segment consists of up to 65,536 16-bit words. Words in a code segment are numbered consecutively from C[0] (code, element 0) through C[65,535]. This is illustrated in Figure 2-20.

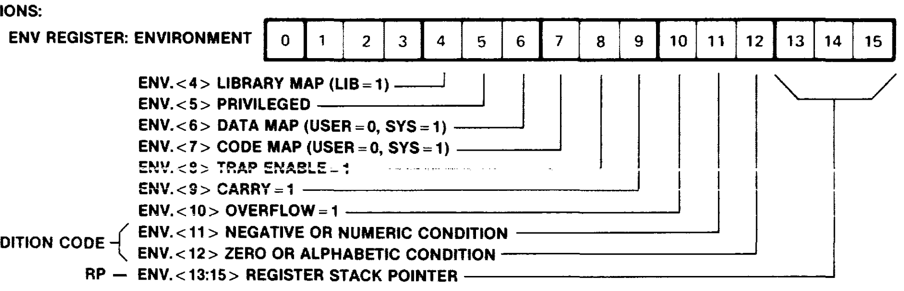
Two registers are associated with code segments. These are described in the following paragraphs.

P REGISTER. The P (for program) Register is the program counter. It contains the 16-bit C[0]-relative address of the current instruction plus one. The contents of the P Register are incremented by one at the beginning of instruction execution so that, nominally, instructions are fetched (and executed) from ascending memory locations. (See top diagram of Figure 2-21.)

When a program branch is taken, a procedure or subprocedure is called, or an interrupt occurs, the C[0]-relative address of the next instruction to be executed is placed in the P Register. (See bottom diagram of Figure 2-21.)



DEFINITIONS:



ENV.<4> LIBRARY MAP (LIB=1)
ENV.<5> PRIVILEGED
ENV.<6> DATA MAP (USER=0, SYS=1)
ENV.<7> CODE MAP (USER=0, SYS=1)
ENV.<8> TRAP ENABLE=1
ENV.<9> CARRY=1
ENV.<10> OVERFLOW=1
CONDITION CODE { ENV.<11> NEGATIVE OR NUMERIC CONDITION
ENV.<12> ZERO OR ALPHABETIC CONDITION
RP — ENV.<13:15> REGISTER STACK POINTER

I REGISTER: CURRENT INSTRUCTION REGISTER
P REGISTER: PROGRAM COUNTER; ADDRESS OF CURRENT INSTRUCTION +1 (RELATIVE TO C[0])
C[0]: FIRST ELEMENT IN THE CODE SEGMENT
G[0]: FIRST ELEMENT IN THE DATA SEGMENT
GLOBAL DATA: DATA AREA ACCESSIBLE FROM ANY POINT IN A PROGRAM
LOCAL DATA: DATA AREA ACCESSIBLE ONLY FROM CURRENTLY EXECUTING PROCEDURE
SUB-LOCAL DATA: DATA AREA ACCESSIBLE ONLY FROM CURRENTLY EXISTING SUBPROCEDURE
L REGISTER: LOCAL DATA POINTER: G[0] RELATIVE ADDRESS OF FIRST ELEMENT IN THE LOCAL DATA AREA. ALSO INDICATES THE LOCATION IN THE MEMORY STACK OF THE LINK (i.e., STACK MARKER) BACK TO THE CALLING PROCEDURE
S REGISTER: TOP OF STACK: G[0] RELATIVE ADDRESS OF THE LAST ACTIVE ELEMENT IN THE MEMORY STACK
REGISTER STACK: EIGHT-ELEMENT REGISTER STACK WHERE ARITHMETIC OPERATIONS ARE PERFORMED. THREE ELEMENTS CAN ALSO BE USED FOR INDEXING
RP: REGISTER STACK POINTER: INDICATES THE TOP ELEMENT IN THE REGISTER STACK

Figure 2-19. Elements of the Program Environment

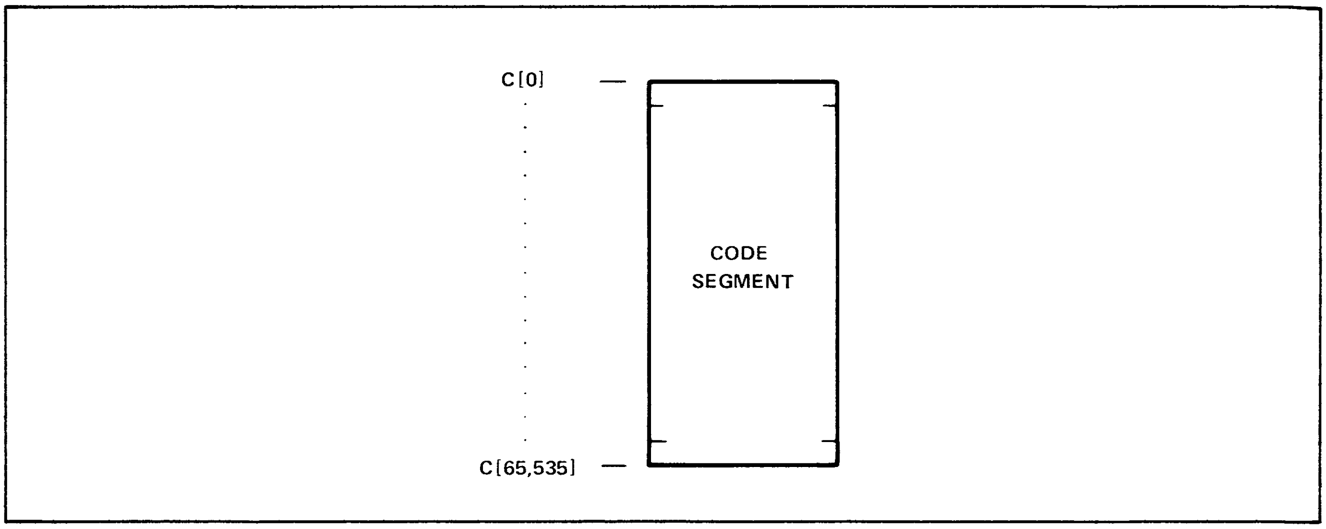


Figure 2-20. Code Segment Addressing Range

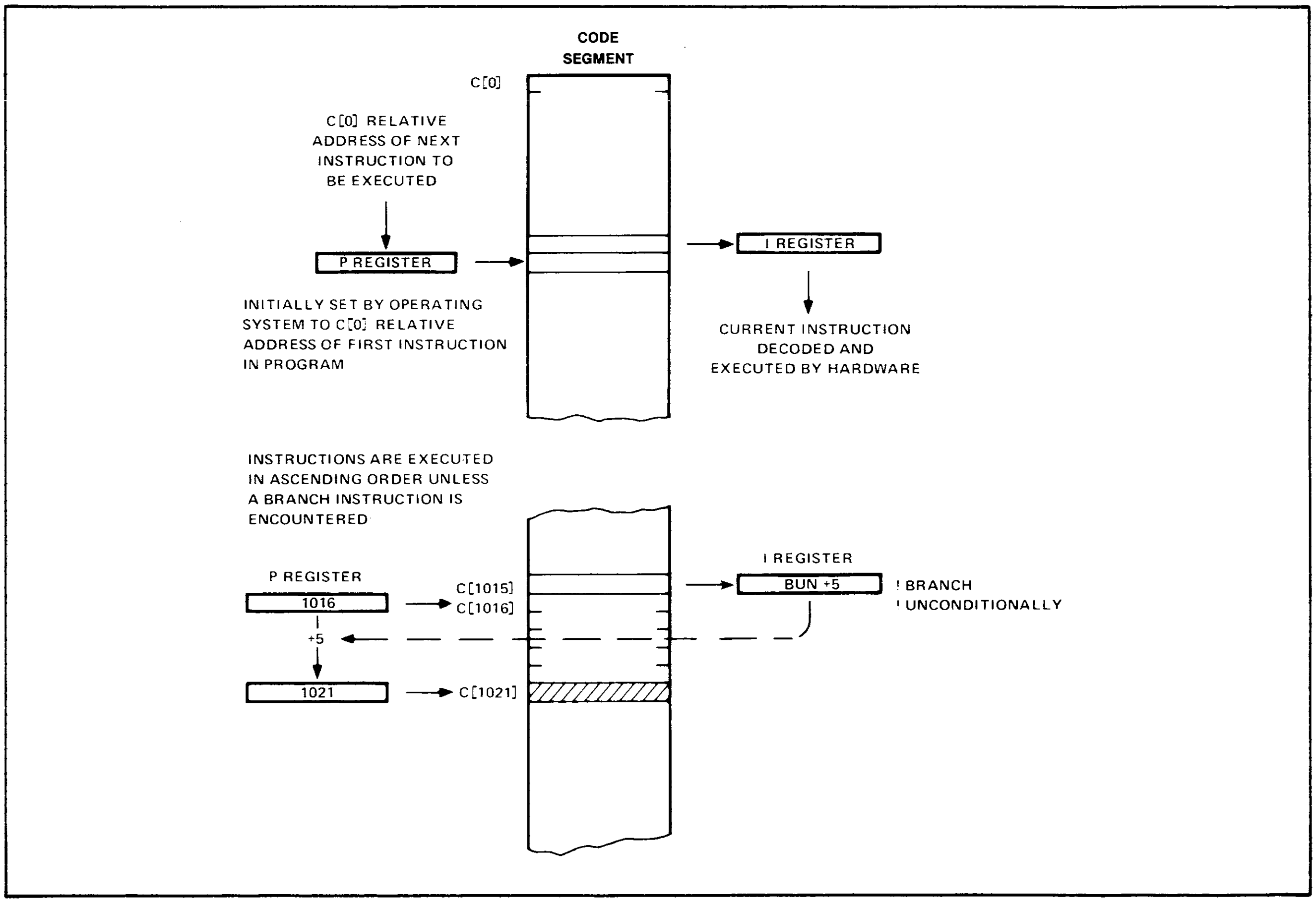

CODE SEGMENT
C[0]
C[0] RELATIVE ADDRESS OF NEXT INSTRUCTION TO BE EXECUTED
P REGISTER
I REGISTER
INITIALLY SET BY OPERATING SYSTEM TO C[0] RELATIVE ADDRESS OF FIRST INSTRUCTION IN PROGRAM
CURRENT INSTRUCTION DECODED AND EXECUTED BY HARDWARE
INSTRUCTIONS ARE EXECUTED IN ASCENDING ORDER UNLESS A BRANCH INSTRUCTION IS ENCOUNTERED
P REGISTER
1016
+5
1021
C[1015]
C[1016]
C[1021]
I REGISTER
BUN +5
! BRANCH
! UNCONDITIONALLY

Figure 2-21. P Register and I Register

I REGISTER. The I (for instruction) Register contains the machine instruction currently being executed. When the current instruction is completed, this 16-bit register is filled with the instruction in a code segment pointed to by the current setting of the P Register. The contents of the P Register are then incremented by one, as described above.

ADDRESSING. Addresses for branching (and for constants) in a code segment are calculated relative to the current setting of the P Register. This is referred to as self-relative addressing.

Instructions that reference a code segment have an eight-bit field for specifying a relative displacement from the current P Register setting. The range of the displacement is therefore -128:+127 words. An example, the BUN instruction, is shown in Figure 2-22.

The location that is addressed by the displacement is referred to as the directly addressable location. This may be the location referenced by the instruction (i.e., it may be the branch location or it may contain the constant) or may itself contain a self-relative address. If the latter, then the referenced location is a relative displacement from the directly addressable location. Whether the direct location is the one referenced by the instruction or contains a self-relative address, is specified by the indirect bit, <i>, in the instruction.

The address of the location in a code segment referenced by an instruction is called "branch^addrs" (branch address). This is the address placed in the P Register when a program branch is taken:

```
P := branch^addrs;
    .
    .
I := code [ P ];

    ("code" refers to a code segment.)
```

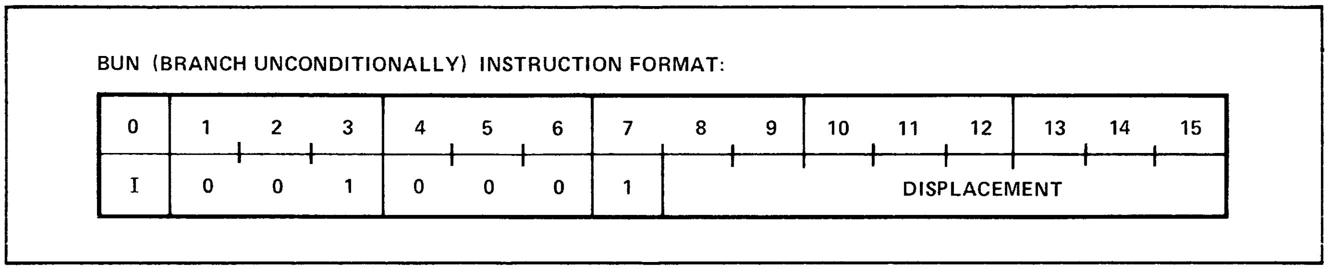


Figure 2-22. Displacement Field for Code Segment Instructions

and used when fetching a program constant from memory:

```
   A := code [ branch^addrs ];

       (A is the top element of the Register Stack.)
```

The address calculated by adding the displacement to the current P register setting is referred to as "dir^branch^addrs" (direct branch address):

```
   dir^branch^address = P + <displacement>;
```

If the referenced location is within the range of the displacement (i.e., P [-128:+127]) then direct addressing is indicated and the direct branch address is used as the branch address. If the referenced location is beyond the range of the displacement, then indirection is indicated and the referenced location (branch^addrs) is a relative displacement from the direct branch address.

Direct addressing is specified by the <i> (indirection) bit, I.<0>, of the instruction equal to "0"; bits I.<8:15> are a two's-complement number (bit I.<8> is the sign bit) giving a positive or negative displacement from the current P Register setting. Therefore

```
      branch^addrs = dir^branch^addrs;
```

Indirect addressing is specified by the <i> bit of the instruction equal to "1"; bits I.<8:15> are a positive or negative displacement from the current P Register setting. Therefore

```
      branch^address = dir^branch^address + code [dir^branch^address];
```

Verbally, the C[0]-relative direct branch address is first calculated (a displacement from the current P Register setting). Then the contents of the direct location (containing a displacement from itself) is added to the direct branch address. The result is the C[0]-relative branch address.

Examples of both direct and indirect addressing are given in Figure 2-23. The "I" in the LWP 9,I instruction signifies indirect addressing.

In addition to direct and indirect addressing, an offset value in a hardware register can be added to the address of the direct or indirect location before the final address is calculated. This permits a code segment location to be referenced as an offset from a base location (this is called indexing). Indexing in a code segment is discussed in Section 3, "Instruction Set", under the LWP instruction.

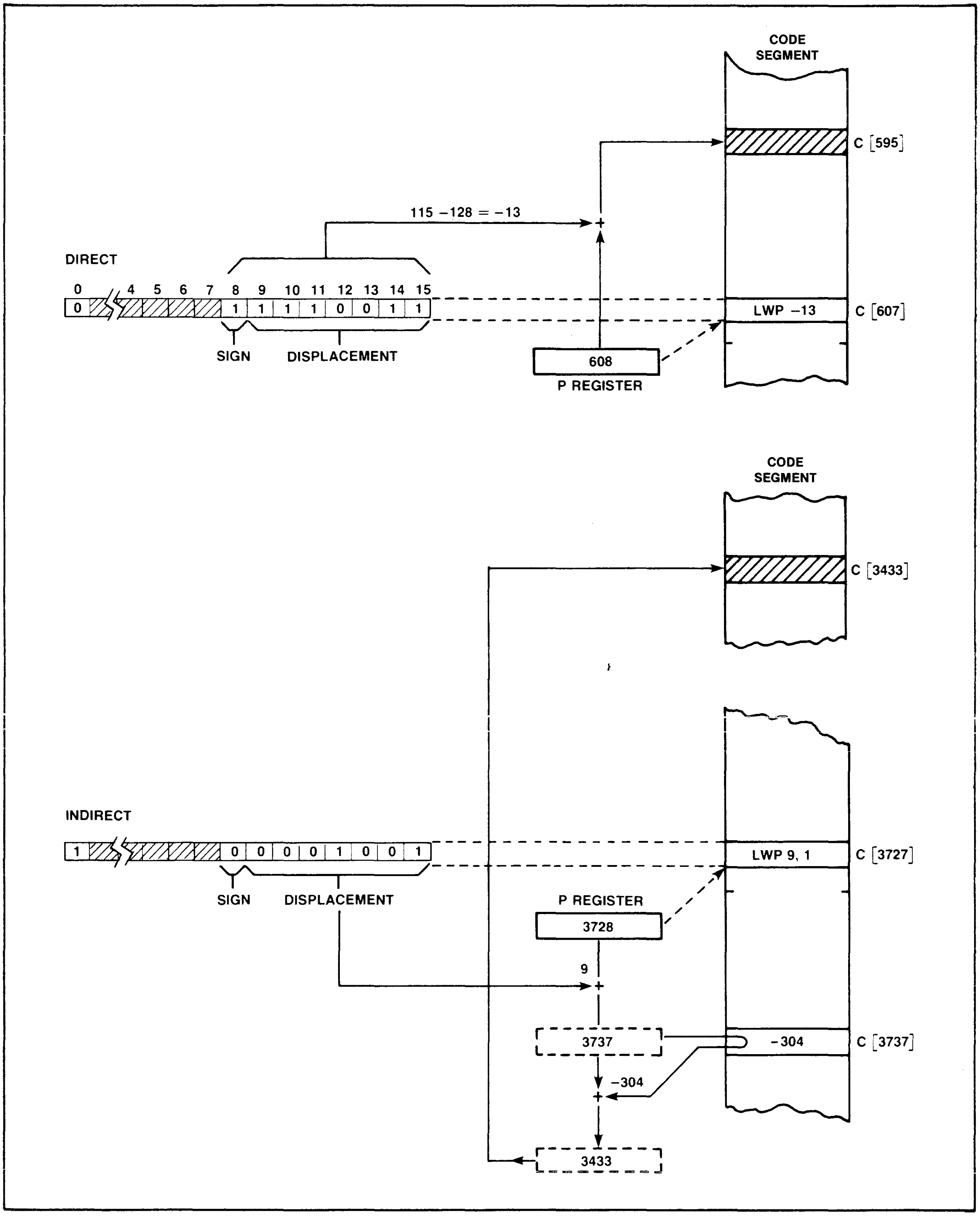


Figure 2-23. Addressing in the Code Segment

Addressing of byte elements (with indexing) is also permitted in the code segment, though restricted to only half of the segment (the same half in which the current P Register setting is located). Byte addressing is discussed in Section 3 under the LBP (load byte from program) instruction.

Data Segment

DATA STORAGE AND ACCESS. The data segment contains a program's temporary storage locations (i.e., variables). Information in this segment consists of single-element items, multiple-element items (arrays), and address pointers. Input/output transfers (which are performed on behalf of application programs by the GUARDIAN File System) are via arrays in a program's data segment.

Part of the data segment is used for dynamic allocation of storage when procedures are invoked (see "Procedures"); this area is referred to as the "memory stack."

The data segment consists of up to 65,536 16-bit words. Addresses in the data segment start at G[0] (global data, word 0) and progress consecutively through G[65,535]. See Figure 2-24. The "memory stack" portion of the data segment is limited to the lower 32,768 words (i.e., G[0:32,767]).

Data is accessed through use of the memory reference instructions. Locations in the data segment are addressed either through the address field in a memory reference instruction (this is called direct addressing) or through an address pointer in memory (this is called indirect addressing). Additionally, the memory reference instructions permit an offset value (in a hardware register) to be added to a direct or indirect address before a final address is calculated. This permits one data element to be referenced as an offset from another data element (this is called indexing). The memory reference instructions are:

LDX:	Load Index register from data segment
NSTO:	Non-destructive Store from Register Stack into data segment
LOAD:	Load word into Register Stack from data segment
STOR:	Store word from Register Stack into data segment
LDB:	Load Byte into Register Stack from data segment
STB:	Store Byte from Register Stack into data segment
LDD:	Load Doubleword into Register Stack from data segment
STD:	Store Doubleword from Register Stack into data segment
ADM:	Add to Memory

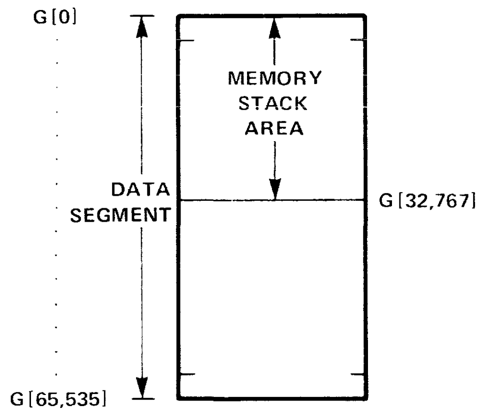


Figure 2-24. Data Segment Addressing Range

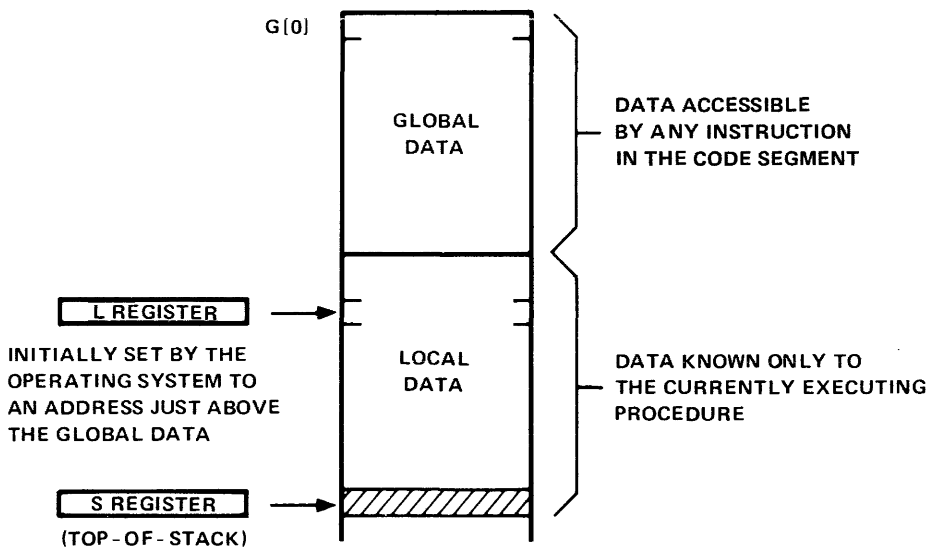


Figure 2-25. L Register and S Register

The data segment is logically separated into three areas: global, local, and sublocal. Each logical area has an addressing base so that relative addressing can be performed. The logical areas are described in the following paragraphs and illustrated in Figure 2-25.

- Global Area

  Data within the global area is addressable by any instruction in the program. The addressing base of the global area is defined as G[0].

  The beginning of the global area coincides with the beginning of the data segment. Thus, the G[0]-relative address of an item is its logical address within the data segment. G[0] is logical address 0.

- Local Area

  Data within the local area is known only to the currently executing procedure. The local area is defined by the 16-bit L Register. The L (for local) Register contains the G[0]-relative address of the word at the beginning of this area. The addressing base of the local area is defined as L[0].

  When a procedure is called, a new local area is defined. This occurs because the address contained in the L Register advances to point above the current local area (the caller's local area is then undefined). Conversely, when a procedure exits, the exiting procedure's local area is deleted (and the preceding local area redefined) because the address in the L Register recedes back to its previous setting.

- Top-of-Stack (or Sublocal) Area

  Data in the top-of-stack area is known only to the currently executing procedure. The top-of-stack location is defined by the 16-bit S Register. The S (for stack) Register contains the G[0]-relative address of the last word currently defined in the memory stack (this is not to be confused with the last word in the total area set aside for the memory stack). The addressing base of the top-of-stack area is defined as S[0].

  During execution of a procedure, the address in the S Register advances as elements are moved from the Register Stack to the top of the memory stack (PUSHed) and recedes as elements are moved from the top of the memory stack to the Register Stack (POPed). The address also advances when procedures and subprocedures are invoked and recedes when they are exited.

ADDRESSING. Data elements in the data segment are fetched and stored by the hardware in terms of word addresses, regardless of the type of operand involved. (The instruction set microcode also provides for the addressing of bytes within a word, as described in the sections on "Direct Addressing" and "Indirect Addressing" that follow.) For purposes of explanation, "data" refers to a data segment and "address" refers to the G[0]-relative address of a word referenced by an instruction. Together, "data" and "address" are used to indicate access to a location in a data segment referenced by an instruction:

```
   A := data [ address ];
```

is a LOAD instruction (A is the top of the Register Stack).

All addressing in the data segment is relative to one of the three addressing bases: G[0], L[0], or S[0]. Instructions that reference memory data locations contain a 9-bit address field for specifying one of the three addressing bases and a relative displacement from that base. Four addressing modes are provided for addressing relative to these bases. The address indicated by the address field in a memory reference instruction is referred to as the direct address. The addressing modes are: G-relative, L-plus-relative, L-minus-relative, and S-minus-relative. These are described in the following paragraphs. Figure 2-26 shows an example of a memory reference instruction and defines the bit patterns for the four addressing modes. Figure 2-27 illustrates each of the addressing modes.

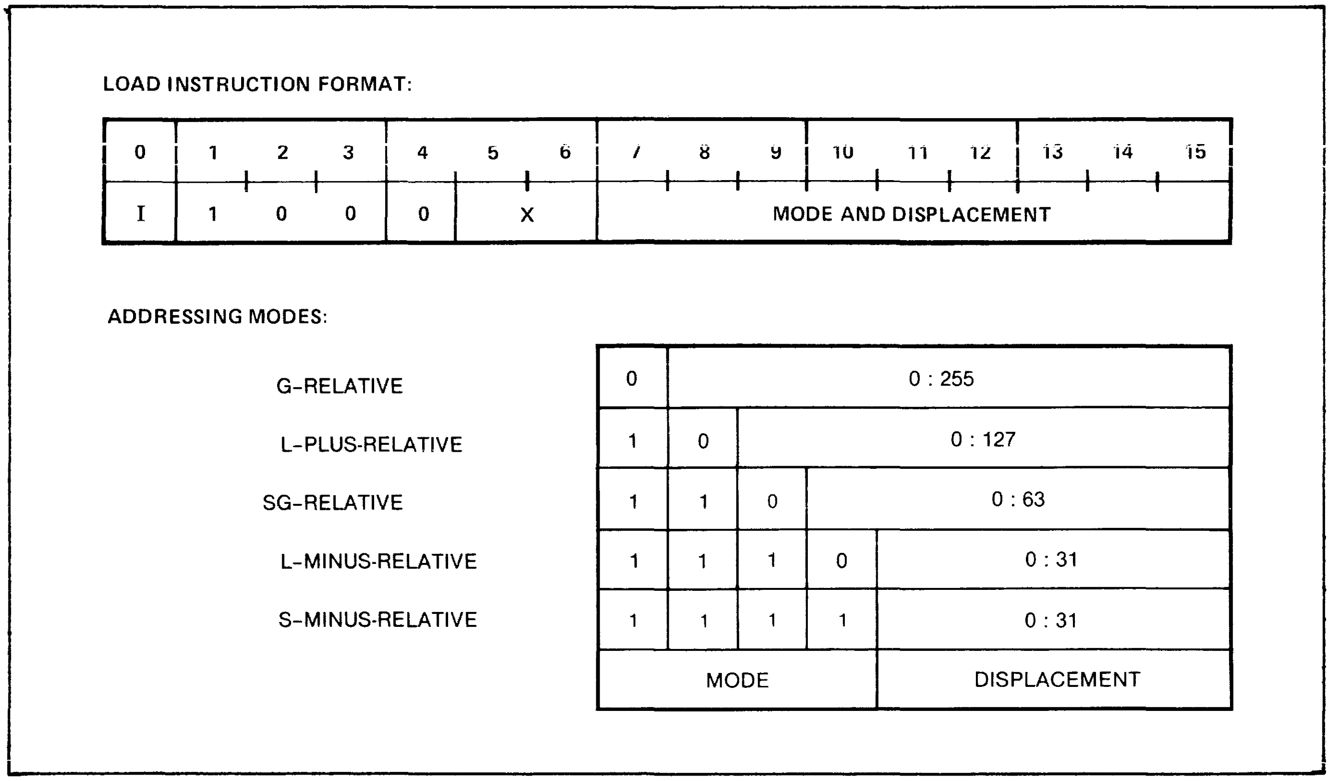


Figure 2-26. Mode and Displacement Field for Memory Reference Instructions

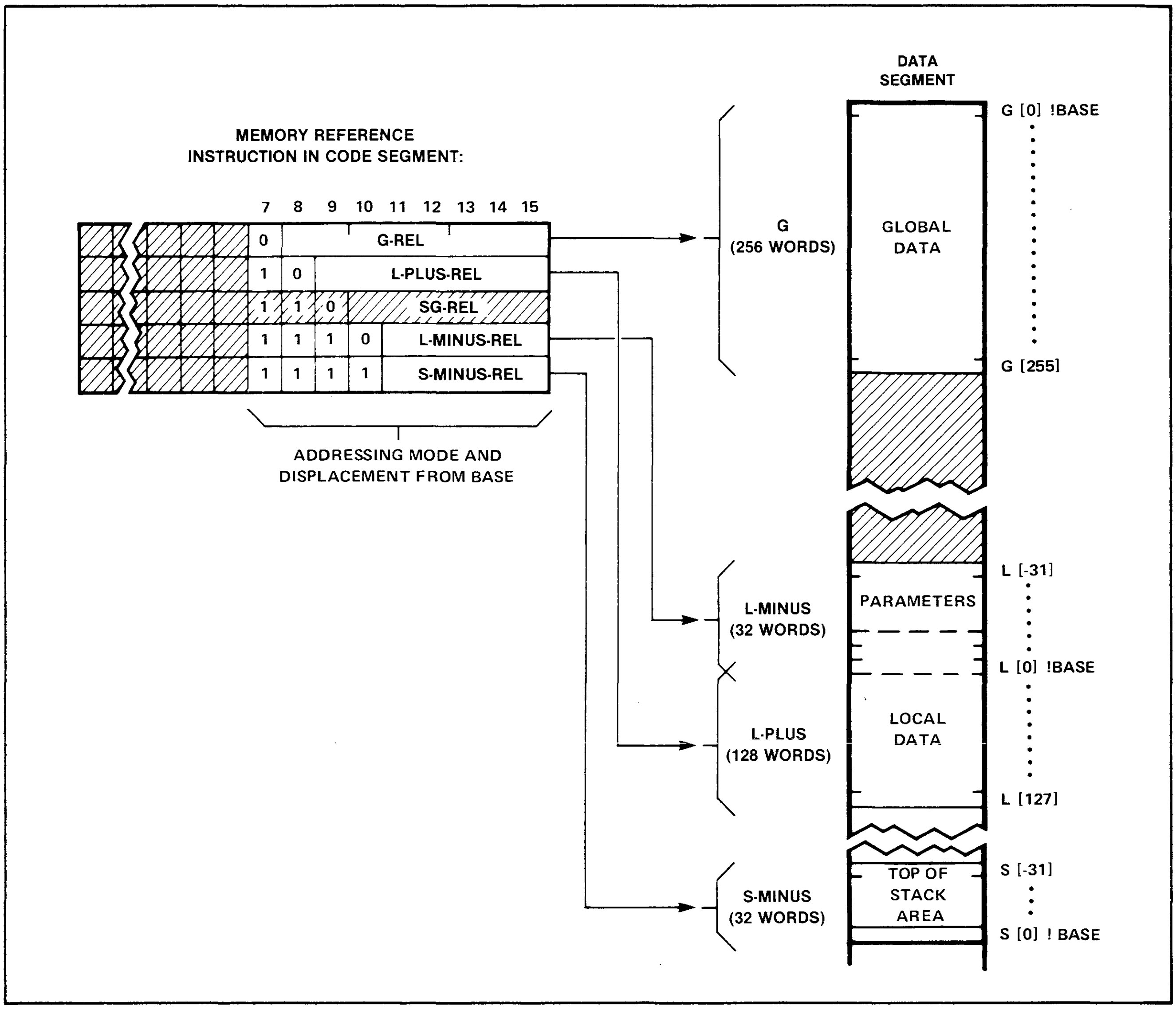


Figure 2-27. Memory Reference Instruction Addressing Modes

- G-Relative Mode

  This mode addresses the first 256 locations in the global area (G[0:255]). The G-relative mode is indicated by bit I.<7> of a memory reference instruction equal to 0; bits I.<8:15> specify a positive word displacement from G[0]. That is:

  ```
  direct^address := I.<8:15>
  ```

- L-Plus-Relative Mode

  This mode addresses the first 128 words of a procedure's local data area (L[0:127]). The L-plus-relative mode is indicated by bits I.<7:8> of a memory reference instruction equal to 10 (binary); bits I.<9:15> specify a positive word displacement from the current L[0]. The hardware calculates a G[0]-relative address by adding I.<9:15> to the contents of the L Register:

  ```
  direct^address := L + I.<9:15>
  ```

- L-Minus-Relative Mode

  This mode addresses the 32 words just below and including the word pointed to by the current L Register setting, L[-31:0] (this area is used for procedure parameter passing). The L-minus-relative addressing mode is indicated by bits I.<7:10> of the memory reference instruction equal to 1110 (binary); bits I.<11:15> are a negative word displacement from the current L[0]. The hardware calculates a G[0]-relative address by subtracting I.<11:15> from the contents of the L Register:

  ```
  direct^address := L - I.<11:15>
  ```

- S-Minus-Relative Mode

  This mode addresses the 32 words just below, and including, the current top-of-stack word (S[-31:0]). (This area is used for a subprocedure's sublocal data and for temporary storage of the Register Stack contents by the PUSH and POP instructions). The S-minus-relative mode is indicated by bits I.<7:10> equal to 1111 (binary); bits I.<11:15> are a negative word displacement from the current S[0]. The hardware calculates a G[0]-relative address by subtracting I.<11:15> from the contents of the S Register:

  ```
  direct^address := S - I.<11:15>
  ```

An additional addressing mode is provided that accesses the operating system's data segment from the user environment--the SG-Relative mode (see "Environment Register" for an explanation of user environment). This mode addresses the first 64 locations of the operating system's data segment (SG[0:63]) and is usable only by procedures executing in privileged mode (e.g., the operating system). The SG-relative addressing mode is indicated by bits I.<7:9> of a memory reference instruction equal to 110 (binary). Bits I.<10:15> are a positive word displacement from SG[0]. (See "Calling External Procedures" for an explanation of SG-relative addressing.)

<u>Direct Addressing</u>. If the <i> (indirection) bit, I.<0>, of a memory reference instruction is a "0", then direct addressing is specified. The ranges of directly addressable locations in the data segment are:

G[0:255]	256 words	G-Relative Mode
L[0:127]	128 words	L-Plus-Relative Mode
L[-31:0]	32 words	L-Minus-Relative Mode
S[-31:0]	32 words	S-Minus-Relative Mode

With direct addressing, the address of an operand referenced by an instruction, relative to one of the addressing bases, is specified in the address field of the memory reference instruction. Therefore,

```
address := direct^address
```

and only one memory reference is needed to access the referenced memory location. Figure 2-28 gives an example of direct addressing.

If a byte operand is referenced, it is in the left half of the referenced location:

```
byte := data [ address ].<0:7>
```

If doubleword operand is referenced, it consists of two words starting at the referenced location:

```
doubleword := data [ address:address + 1] ! two words.
```

Quadruplewords cannot be accessed as such by any of these modes. A quadrupleword must be accessed as some combination of smaller units, such as two doublewords or four words.

<u>Indirect Addressing</u>. If the <i> (indirection) bit, I.<0>, of a memory reference instruction is a "1", then indirect addressing is specified. The range of indirect addressing is G[0:65,535] (i.e., any location in the data segment).

With indirect addressing, the address of the referenced location, relative to G[0], is contained in a location that can be addressed directly (the contents of the direct location are referred to as an address pointer). Two memory references are needed to access the referenced location; the first to fetch the address,

```
address := data [ direct^address ];
```

the second to access the operand. Figure 2-29 gives an example.

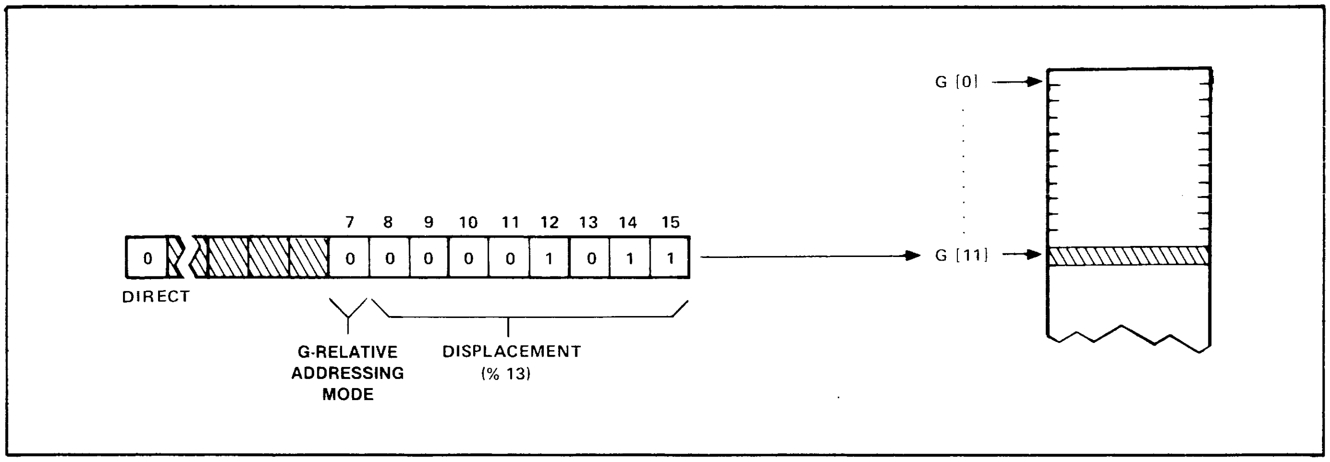


Figure 2-28. Direct Addressing in the Data Segment

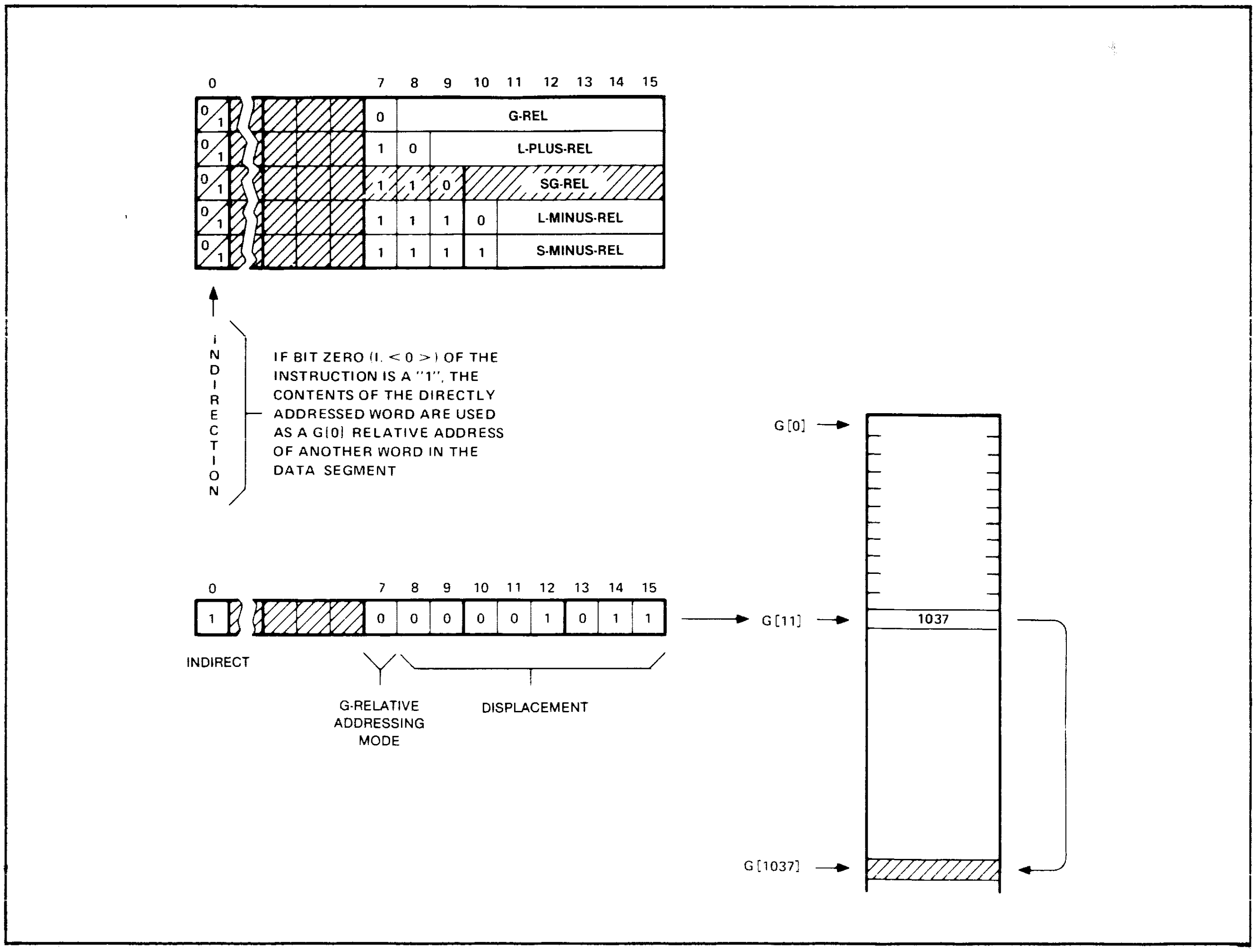

0 7 8 9 10 11 12 13 14 15
0 G-REL
1 0 L-PLUS-REL
1 1 0 SG-REL
1 1 1 0 L-MINUS-REL
1 1 1 1 S-MINUS-REL
INDIRECTION
IF BIT ZERO (I. < 0 >) OF THE INSTRUCTION IS A "1", THE CONTENTS OF THE DIRECTLY ADDRESSED WORD ARE USED AS A G[0] RELATIVE ADDRESS OF ANOTHER WORD IN THE DATA SEGMENT
1
0 0 0 0 0 1 0 1 1
INDIRECT
G-RELATIVE ADDRESSING MODE
DISPLACEMENT
G[0]
G[11]
1037
G[1037]

Figure 2-29. Indirect Addressing in the Data Segment

If a byte operand is accessed, the address pointer contains a G[0]-relative byte address. Bits <0:14> of the address pointer are the word address of the byte operand, bit <15> of the address pointer indicates whether the referenced byte is in the left-hand part of the word, <0:7> or the right-hand part, <8:15>:

```
   byteaddress := data [ direct^address ];

   address := byteaddress.<0:14>;
```

and the referenced byte is

```
   byte := if byteaddress.<15> then
              data [ address ].<8:15>   ! right byte.
           else
              data [ address ].<0:7>;   ! left byte.
```

An example is shown in Figure 2-30.

Note that, because a byte address is effectively divided by two (to provide a word address), and the maximum byte address is 65,535, addressing of bytes is limited to the lower 32,768 words of a data segment (the memory stack area).

If a doubleword operand is accessed, the address pointer contains a G[0]-relative word address:

```
   address := data [ direct^address ];
```

and the referenced doubleword is

```
   doubleword := data [ address:address + 1]
```

Indexing. Indexing is used to reference memory locations relative to a data element in memory. A typical use is when an element in an array is accessed.

Generally, indexing is done as follows. An initial address is first calculated as described previously (any addressing mode as well as direct and indirect addressing is permitted). This initial address is then used as a base address for indexing. The indexing value, contained in an index register (referred to as "X"), is added to the initial address to provide the address of the referenced operand. This is shown in the upper part of Figure 2-31.

Any one of three registers in the Register Stack (R[5:7]) can be used as index registers. The register to be used for indexing is specified in the <x> (index) field, I.<5:6>, that is part of all memory reference instructions. (Note the instruction format in the lower part of Figure 2-31.) The index field corresponds to Register Stack elements as follows:

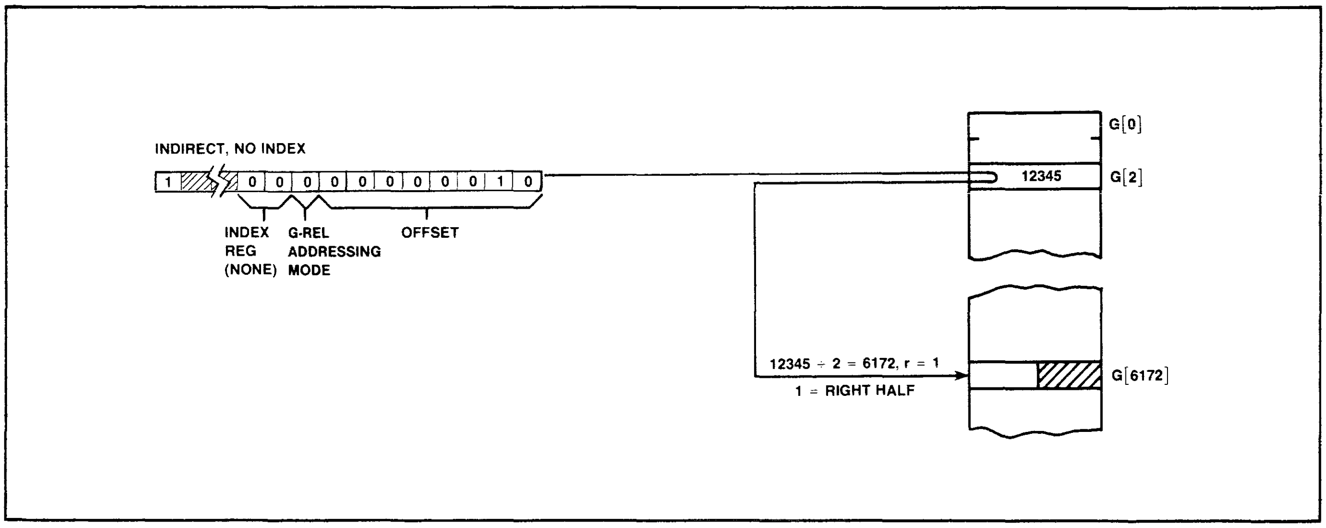


Figure 2-30. Indirect Byte Addressing in the Data Segment

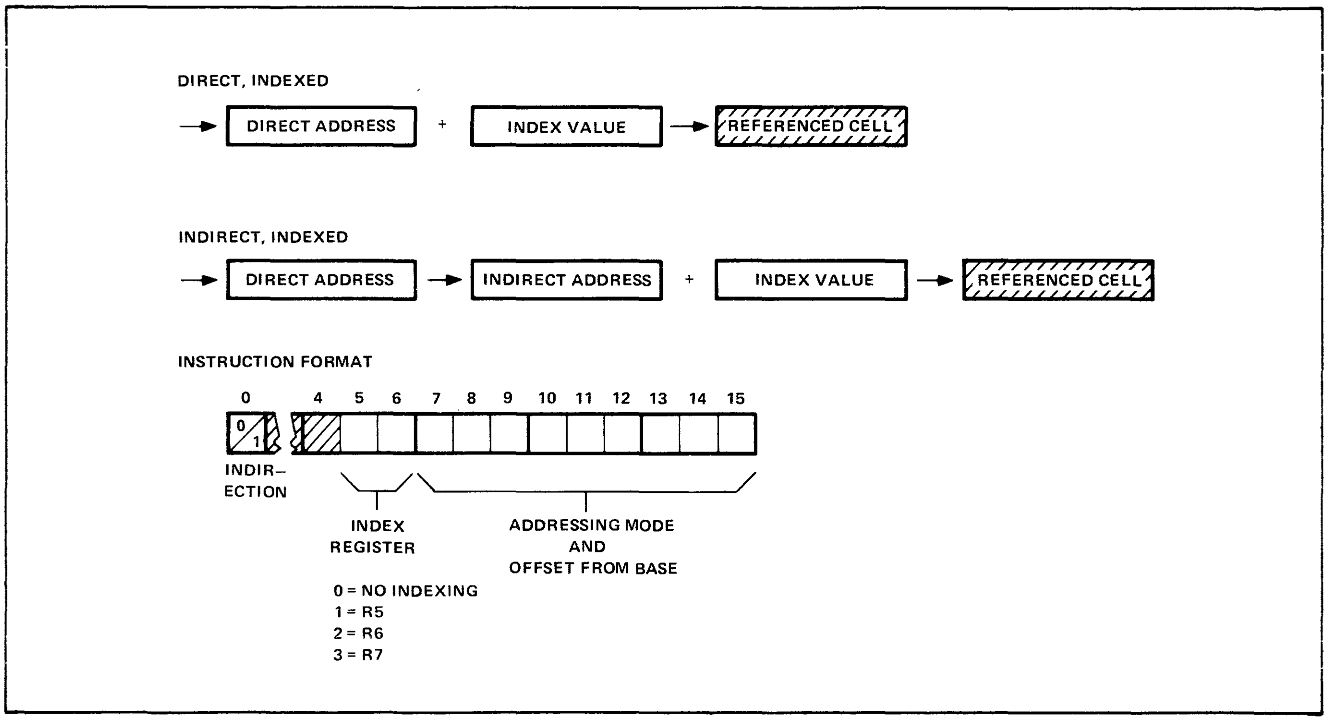


Figure 2-31. Indexing

I.<5:6> VALUE	INDEX REGISTER
0	X = no indexing
1	X = R[5]
2	X = R[6]
3	X = R[7]

An index register can contain values from -32,768 through +32,767 to provide direct word and doubleword addressing of any location in the data area (all addressing is modulo 65,535). The value in an index register is always treated as an element indexing value. That is, if a byte instruction is being executed, the contents of an index register are treated as a byte offset; if a doubleword instruction is being executed, the contents are treated as a doubleword offset.

Specifically,

- For direct, indexed addressing of word operands,

  ```
  address := direct^address + X
  ```

  the contents of the index register, X, are added to the direct address; and the referenced element (referred to as "wordx") is

  ```
  wordx := data [ address ]
  ```

- For indirect, indexed addressing of word operands,

  ```
  address := data [ direct^address ] + X

  wordx := data [ address ]
  ```

- For direct, indexed addressing of byte operands,

  ```
  byteaddress := 2 * direct^address + X
  ```

  The direct^address (a word address) is multiplied by two to obtain a byte address. The indexing value (a byte offset) is added to that. The G[0]-relative address of the referenced byte is converted to a word address as follows:

  ```
  address := byteaddress.<0:14>;
  ```

  And the referenced byte (referred to as "bytex") is

  ```
  bytex := if byteaddress.<15> then
              data [ address ].<8:15> ! right byte.
           else
              data [ address ].<0:7>  ! left byte.
  ```

- For indirect, indexed addressing of byte operands,

  ```
  byteaddress := data [ direct^address ] + X
  ```

  The address pointer indicated by "data [direct^address]" contains a byte address. X, which contains a byte offset, is added to the byte address. The "address" and "bytex" are then determined as described above.

- For direct, indexed doubleword operands,

  ```
  address := direct^address + 2 * X
  ```

  That is, the indexing value (a doubleword element index) is multiplied by two to provide a word index. This value is added to the initial address (also a word address) to generate a G[0]-relative word address, and the element referenced (referred to as "dwordx") is

  ```
  dwordx := data [ address : address + 1 ] ! two words.
  ```

- For indirect, indexed doubleword operands,

  ```
  address := data [ direct^address ] + 2 * X
  ```

  The address pointer indicated by "data [direct^address]" contains a word address. X, which contains a doubleword offset, is multiplied by two (to generate a word offset) and added to the initial address. The "dwordx" is the same as described above.

Figure 2-32 shows examples of word and byte indexing.

Three instructions deal with loading and modifying index register contents. They are:

LDX:	Load an Index register from data segment
LDXI:	Load an Index register with Immediate operand
ADXI:	Add to an Index register the Immediate operand

An additional instruction is used for branching on the contents of an index register. It is:

BOX:	Branch on Index register less than A (top of register stack) or increment index register

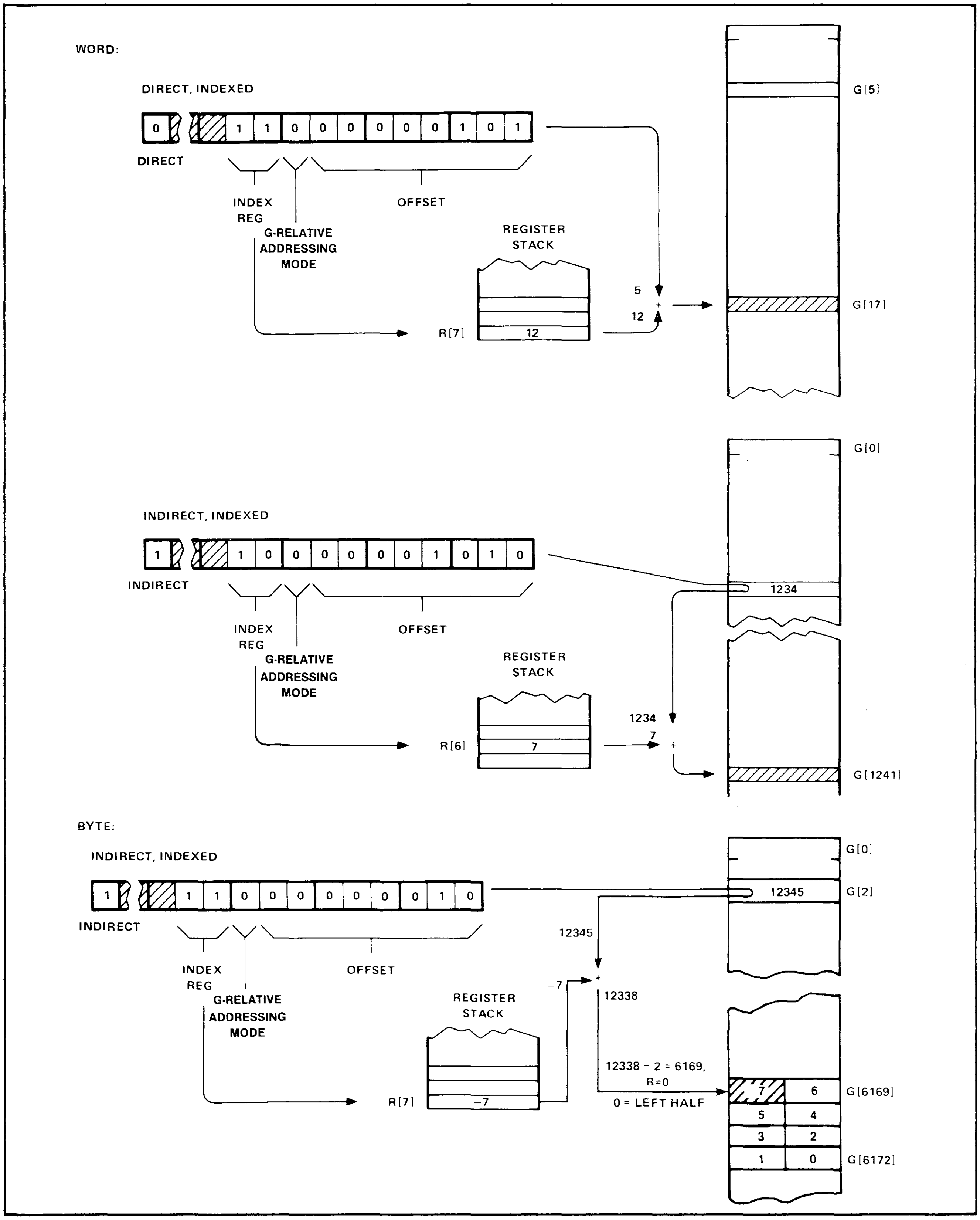


Figure 2-32. Examples of Indexing

Register Stack

The Register Stack is where arithmetic computations are performed and, except for the Compare Words and Compare Bytes instructions, where comparisons are made. The Register Stack consists of eight 16-bit registers, designated R[0] (Register Stack, element 0) through R[7]; see Figure 2-33. Three elements of the Register Stack, R[5:7], also double as index registers (see "Indexing").

A typical operation to add two numbers in the Register Stack is as follows: the operands are first loaded into the Register Stack using LOAD instructions, an IADD (integer add) instruction is then executed performing the desired arithmetic, the result then stored back into memory using a STOR instruction. Grouped together to form a program, the preceding operation would look like this:

```
LOAD G + 002 ! load data element G[2] onto Register Stack
LOAD G + 003 ! load data element G[3] onto Register Stack
IADD         ! integer add
STOR G + 004 ! store the result from the Register Stack into G[4]
```

The condition of the register stack for each of these instructions is shown in Figure 2-34.

Usually, elements in the Register Stack are addressed implicitly. That is, an instruction operates on the top element (or elements) without specifying the actual register(s) involved. The current top element of the Register Stack is defined by the Register Stack Pointer, RP. RP, which is a three-bit field in the Environment Register (next described), contains the register number, 0:7, of the top element. The RP setting is incremented when operands are loaded into the Register Stack:

```
RP := RP + <size of element> ;
```

and decremented when arithmetic is performed or results are stored:

```
RP := RP - <size of element> ;
```

The empty state of the Register Stack is defined as RP = 7. The full state is also RP = 7. There is no protection against rolling RP over from 7 to 0.

The operation of the Register Pointer for the above program example is shown in Figure 2-35.

The elements in the Register Stack are named as to their location relative to the current top element. The top element is designated

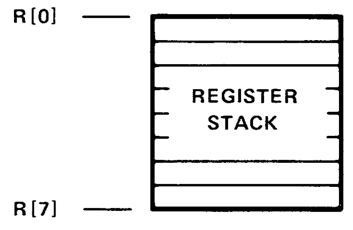


Figure 2-33. Register Stack

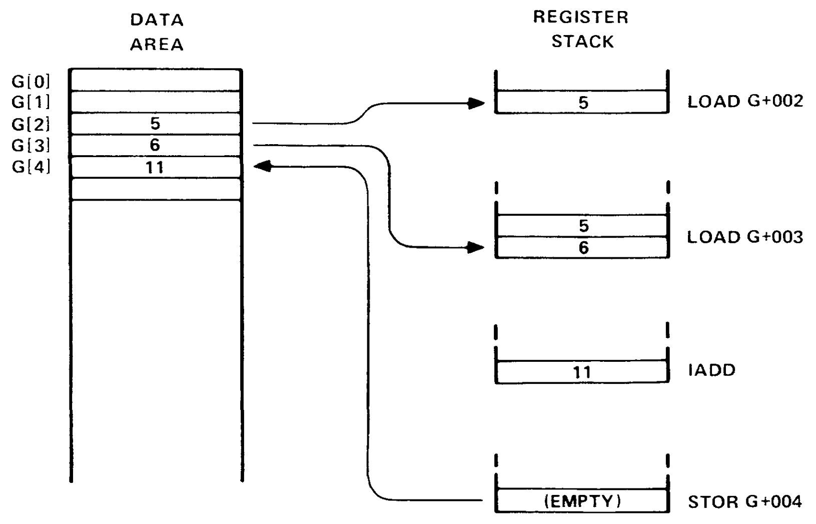


Figure 2-34. Example of Register Stack Operation

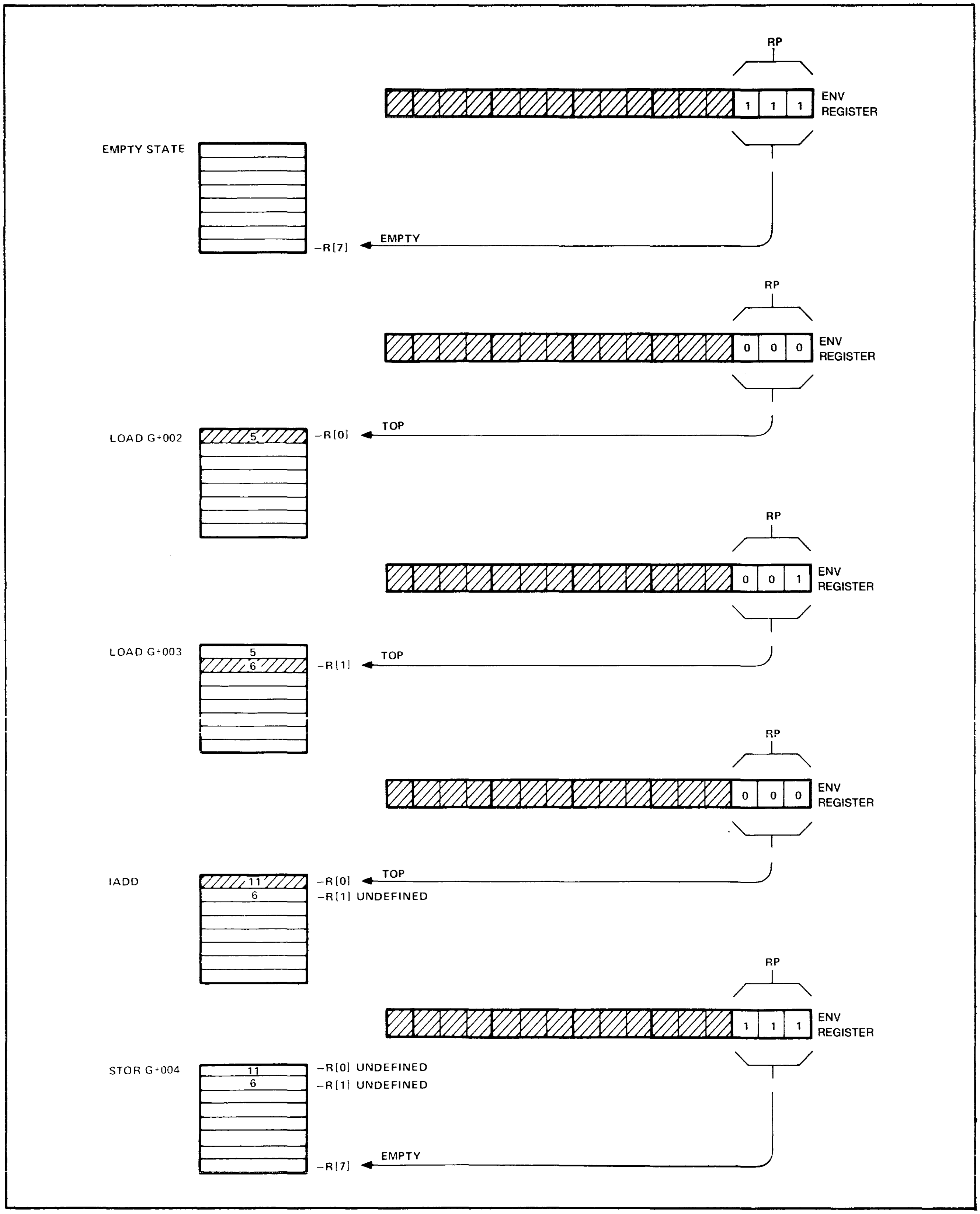


Figure 2-35. Action of the Register Pointer

"A", the second from the top "B", and so on through "H":

```
A  = RP       ! top of Register Stack
B  = RP [-1]
C  = RP [-2]
D  = RP [-3]
E  = RP [-4]
F  = RP [-5]
G  = RP [-6]
H  = RP [-7]
```

Examples of register naming are shown in Figure 2-36.

Environment Register

The 16-bit ENV (for Environment) Register maintains the IPU state of the currently executing process. The individual bits and bit fields of the ENV Register are continually referenced and updated by the IPU hardware and firmware. The ENV Register contents are saved (along with the contents of the P and L Registers) by the firmware as part of the executing state of a process when a procedure is invoked or when an interrupt occurs. The firmware restores the ENV Register to its previous state when the procedure or interrupt finishes.

The format of the ENV Register is shown in Figure 2-37. The following paragraphs describe the meanings of the bits in this register. (The four high-order bits are reserved for use as flags by the microcode.)

LIBRARY SPACE BIT. The LS bit (ENV.<4>) works with the CS bit (7) to define the current code segment. When this bit is a "1", one of the alternate (or "library") code segments is made current, rather than one of the standard segments--system code or user code, as selected by the CS bit. In the case of "system" selection by CS, the System Code Extension is selected as the library segment; in the case of "user" selection by CS, the user's Library Code segment is selected.

PRIVILEGED MODE BIT. The PRIV bit (ENV.<5>), when a "1", means that the program is currently executing in privileged mode and is permitted to perform privileged operations. Privileged operations are characterized by having the potential to adversely affect the operating system if misused. Some examples of privileged operations are: sending data over an interprocessor bus (SEND), initiating input/output operations (EIO), calling privileged procedures, and accessing system tables. Normally, only the operating system executes in privileged mode; privileged operations are performed on behalf of application programs by the operating system.

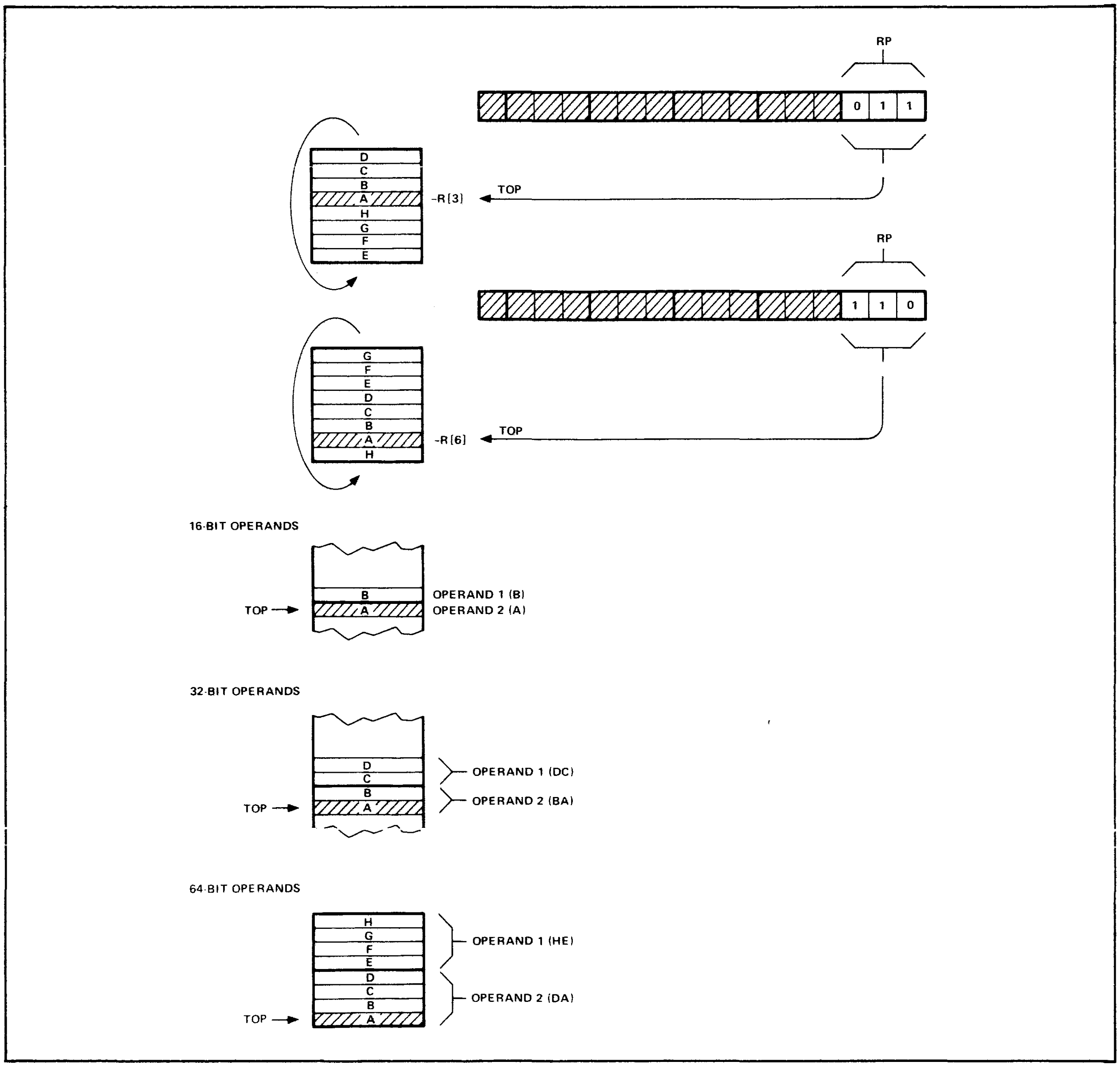


Figure 2-36. Naming Registers in the Register Stack

Nonprivileged programs can perform privileged operations only indirectly, by calling procedures designated "callable". (Callable procedures execute in privileged mode, but can be called by nonprivileged procedures.) When a nonprivileged procedure calls a callable procedure, its nonprivileged state is restored on return.

Instructions designated privileged can be executed only if the PRIV bit in the ENV Register is a "1". If a nonprivileged program (i.e., PRIV = 0) attempts to execute a privileged instruction or call a privileged procedure, the firmware transfers control to the operating system Instruction Failure Trap Handler.

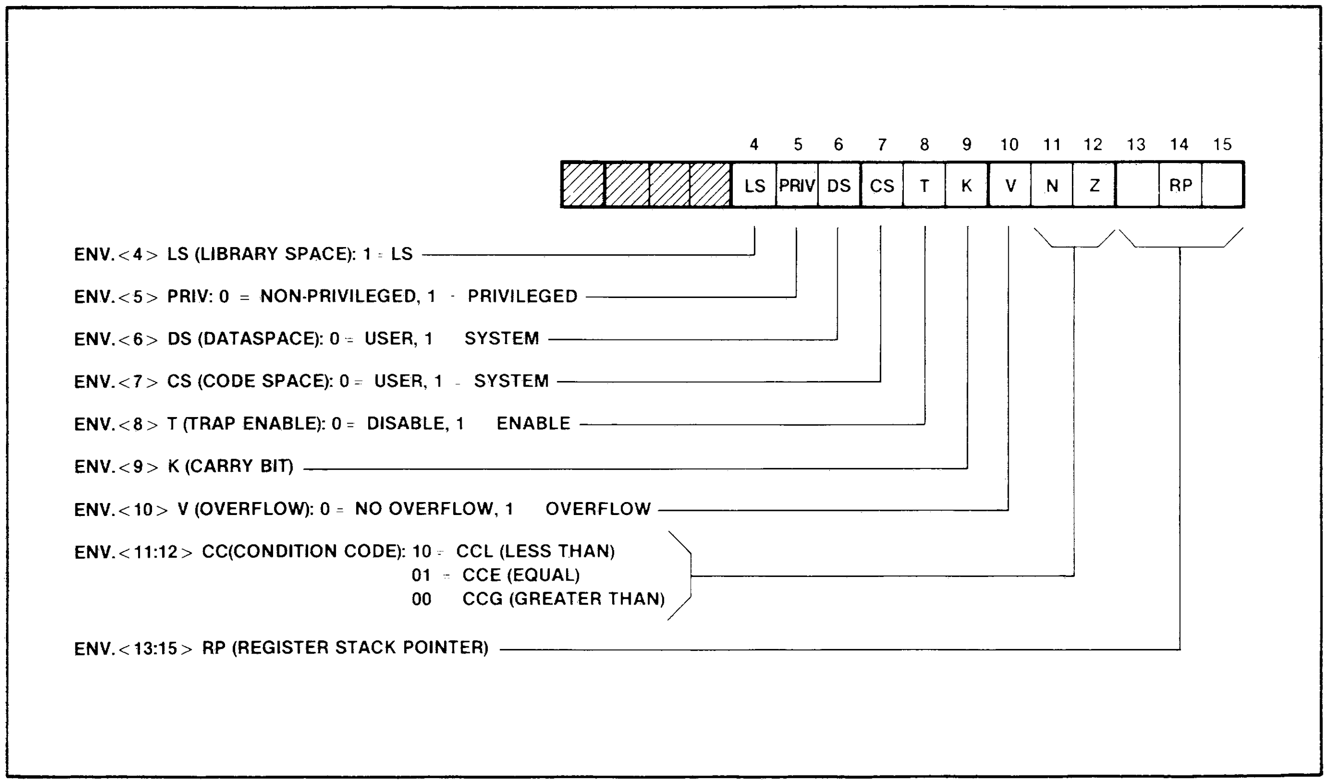

4 5 6 7 8 9 10 11 12 13 14 15
LS
PRIV
DS
CS
T
K
V
N
Z
RP
ENV.<4> LS (LIBRARY SPACE): 1 = LS
ENV.<5> PRIV: 0 = NON-PRIVILEGED, 1 = PRIVILEGED
ENV.<6> DS (DATASPACE): 0 = USER, 1 = SYSTEM
ENV.<7> CS (CODE SPACE): 0 = USER, 1 = SYSTEM
ENV.<8> T (TRAP ENABLE): 0 = DISABLE, 1 = ENABLE
ENV.<9> K (CARRY BIT)
ENV.<10> V (OVERFLOW): 0 = NO OVERFLOW, 1 = OVERFLOW
ENV.<11:12> CC(CONDITION CODE): 10 = CCL (LESS THAN)
01 = CCE (EQUAL)
00 = CCG (GREATER THAN)
ENV.<13:15> RP (REGISTER STACK POINTER)

Figure 2-37. Environment Register

DATA SPACE BIT. The DS bit (ENV.<6>) defines the "current" data segment. This specifies which data area is to be accessed when a data reference is made. DS, when "0", specifies the user data segment; "1" specifies the system data segment. (Programs executing in privileged mode can make explicit system data references regardless of the state of the DS bit through use of the SG-relative addressing mode.)

CODE SPACE BIT. The CS bit (ENV.<7>), together with the LS bit (ENV.<4>), defines the "current" code segment. This specifies which code segment is to be accessed when an instruction or code area constant is fetched. CS, when "0", specifies the User Code segment (or user's Library Code Segment if LS is "1"); "1" specifies the System Code segment (or System Code Extension if LS is"1").

TRAP ENABLE BIT. The T bit (ENV.<8>) specifies whether or not control is to be transferred to the operating system if an arithmetic overflow occurs or a divide with a divisor of zero is attempted. If T is a "1" and an arithmetic overflow occurs (V, ENV.<10>, = 1), control is transferred to the operating system Arithmetic Overflow Interrupt Handler (see the GUARDIAN Operating System Programming Manual for possible recovery procedures). If T is a "0", control remains with the program having the overflow condition.

Generally, the T bit is under control of the operating system. However, application programs can set T to "0" by means of the SETE instruction if it is desired to handle arithmetic overflow conditions locally.

CARRY BIT. The K bit (ENV.<9>), when "1", indicates that a carry out of the high-order bit position occurred when executing an arithmetic instruction on a 16-, 32-, or 64-bit operand. The state of the K bit reflects the last arithmetic type instruction executed. The state of the K bit is also altered as the result of executing a scan instruction (SBW or SBU).

Two instructions test the state of the carry bit. They are:

BIC:	Branch if carry
BNOC:	Branch if no carry

OVERFLOW BIT. The V bit (ENV.<10>), if a "1", indicates that an overflow condition occurred or a divide (IDIV) with a divisor of zero was attempted. Overflow is generally associated with arithmetic operations on 16-, 32-, and 64-bit operands. Overflow also occurs in a LDIV instruction if the quotient cannot be represented in 16 bits, or in floating-point arithmetic if the exponent is too large or too small (see "Number Representation" earlier in this section).

The state of the V bit is tested by the BNOV (Branch if no overflow) instruction.

CONDITION CODE BITS. This two-bit field (ENV.<11:12>) forms the Condition Code. The Condition Code generally reflects the outcome of a computation, comparison, bus transfer, or input/output operation. (The Condition Code is also set by the GUARDIAN File System to reflect the outcome of File System calls.)

The two bits that form the Condition Code are designated:

N = negative or numeric, ENV.<11>, and

Z = zero or alphabetic, ENV.<12>.

The Condition Code has three states. They are:

```
CCL = less than,    ENV.<11:12> = 10  (N = 1, Z = 0)
CCE = equal to,     ENV.<11:12> = 01  (N = 0, Z = 1)
CCG = greater than, ENV.<11:12> = 00  (N = 0, Z = 0)
```

The state of the Condition Code is tested by the following branch instructions:

```
BLSS: Branch if CCL              BLEQ: Branch if CCL or CCE
BEQL: Branch if CCE              BLEG: Branch if CCL or CCG
BGTR: Branch if CCG              BGEQ: Branch if CCE of CCG
```

The Condition Code is set explicitly by the following instructions:

```
CCL: Set CCL
CCE: Set CCE
CCG: Set CCG
```

The following paragraphs define the manner of setting the Condition Code in various cases.

Following a Computation. In this case, a hardware "cc (x)" operation sets the Condition Code bits as follows:

```
cc (x):
  N := if x < 0 then 1 else 0; ! negative
  Z := if x = 0 then 1 else 0; ! zero
```

x is the operand.

Therefore, for a computation,

```
  CCL:  operand < 0
  CCE:  operand = 0
  CCG:  operand > 0
```

Following a computation, the Condition Code reflects the resultant value in a data area location, the top of the Register Stack, or in an index register. The location reflected by the Condition Code depends on the last instruction executed (see Section 3 for particulars). For example, a simple program to add two numbers and then store the result affects the Condition Code as follows:

```
   Data in Global Area
      G [2]  =   5
      G [3]  =  -5

   LOAD G + 002
      sets the Condition Code to CCG (5 on the top of the register
      stack)

   LOAD G + 003
      sets the Condition Code to CCL (-5 on the top of the register
      stack)

   IADD
      sets the Condition Code to CCE (0 on the top of the register
      stack)

   STOR G + 004
      does not change the Condition Code
```

For a Comparison. In this case, a hardware "cc (x:y)" operation (for signed operands) or a "cc (x':'y)" operation (for unsigned operands) sets the Condition Code bits as follows:

```
  for a signed comparison,          for an unsigned comparison,

  CCL:  x  <  y                     CCL:  x  '<'  y
  CCE:  x  =  y                     CCE:  x   =   y
  CCG:  x  >  y                     CCG:  x  '>'  y
```

In the table above, "operandl" refers to the first element loaded onto the Register Stack (i.e., the second element from the top of the stack), and "operand2" refers to the top element in the Register Stack. When two arrays are compared by a COMW or COMB instruction, "operandl" refers to the element in the destination array, and "operand2" refers to the element in the source array. The single quote marks surrounding an operator symbol signify a logical rather than arithmetic operation; thus ':' and '<' are logical comparison operations.

For a Byte Test. In this case, a hardware "ccb (x)" operation sets the Condition Code bits as follows:

```
ccb (x):
  N := if "0" <= x <= "9" then 1   ! numeric.
                          else 0 ; ! not numeric.
  Z := if "A" <= x <= "Z"
          or
          "a" <= x <= "z" then 1   ! alpha.
                          else 0 ; ! not alpha.
```

Therefore, for a byte test,

CCL:	ASCII numeric
CCE:	ASCII alpha
CCG:	ASCII special

For byte test, the Condition Code is set according to bits <8:15> of the operand on the top of the Register Stack when a BTST (Byte Test) or any "load byte" instruction (LDB, LBP, LBA, LBAS, LBX, LBXX) is executed. A Condition Code of CCL indicates that an ASCII numerical character (i.e., "0, 1, ..., 9") is on the top of the register stack. CCE indicates a lowercase or uppercase ASCII alphabetical character (i.e., "a, b, ..., z" or "A, B, ..., Z"), CCG indicates an ASCII special character (i.e., not numerical and not alphabetical).

For IPB Communication. For the Condition Code setting result from interprocessor bus communication, see the interprocessor bus description elsewhere in this section and see the description of the SEND instruction in Section 3.

For input/output, see the input/output channel description in this section and the EIO, IIO, and HIIO instructions in Section 3.

REGISTER STACK POINTER BITS. This three-bit field (ENV.<13:15>) defines the current top element of the Register Stack. The value of RP is implicitly changed by instructions that operate on values on the top of the Register Stack. RP is incremented as instructions are executed to load operands into the Register Stack, decremented when computations are performed or results stored.

The STRP instruction is used to explicitly set the RP value.

ENV REGISTER INITIAL SETTINGS. The ENV Register is given the following setting as a result of a cold load:

%3447

This setting specifies: privileged mode, system data, system code, traps disabled, no carry, overflow, CCG, and RP = 7.

The ENV Register is given the following setting as a result of an interrupt:

%3447

This setting specifies: privileged mode, system data, system code, traps disabled, no carry, overflow, CCG, and RP = 7.

NOTE

The overflow bit is set in the initial ENV on a NonStop II processor to distinguish it from a NonStop processor, whose initial ENV setting is %3407.

SETE INSTRUCTION. The SETE instruction is used to alter the ENV Register contents. ENV.<8:15> can be set to any value desired; the bits of ENV.<0:7> are either cleared or left unchanged. This prevents nonprivileged processes from becoming privileged and/or accessing system data. A similar mechanism is used in the EXIT instruction to restore the ENV Register contents when a procedure finishes. The programmer should take care when clearing ENV.<0:7> on NonStop II systems, since it is possible to inadvertently clear the Library Space (LS) bit, ENV.<4>.

Procedures and the Memory Stack

A procedure is a functional block of instructions that, when called into execution, performs a specific operation. A procedure can perform an operation as simple as adding two numbers or as complex as locating an entry in a data base. A program typically consists of many procedures.

Several characteristics of procedures are:

- A procedure can be called into execution (invoked) from any point in a program.
- Procedures are assigned a "callability" attribute. The attribute specifies whether or not the caller must be executing in privileged mode and whether or not the called procedure executes in privileged mode.

- The caller need not be concerned with its environment or the environment of the procedure it called, because:
  - The caller's environment is automatically saved by the hardware when a procedure is called and is restored by the hardware when the called procedure finishes.
  - When a procedure is called into execution, it is allocated a temporary storage area called a local data area. The local data area is known only to the executing procedure and is logically separate from other procedures' local data areas.
- Parameters (or arguments) can be passed to a procedure for evaluation. The parameters can be actual operands or can be addresses of operands.
- A procedure can return a value (such as the result of a computation) to its caller.
- A procedure itself can contain one or more subprocedures. A subprocedure is similar to a procedure in that it is also a functional block of instructions, called into execution to perform a specific operation. There are several similarities between procedures and subprocedures: a subprocedure, like a procedure, is allocated a temporary (sublocal) storage area while it executes, parameters can be passed to a subprocedure, and a subprocedure can return a value to its caller. Some significant differences between procedures and subprocedures are: different instructions are used to call a subprocedure than a procedure, a subprocedure has no "callability" attribute (it executes in the mode of its caller), and the amount of sublocal storage available to a subprocedure is significantly less than the amount of local storage available to a procedure. In addition, a subprocedure can be called only by the procedure that contains it.

A procedure consists of a contiguous block of instruction codes and program constants in a code segment. All procedures that comprise a program are in the same code segment, except for any system or user library procedures called (these are in the System Code segment, System Code Extension, or User Library code segment). The address of the first instruction in a procedure is called the "entry point". The entry points for all procedures in a program are located in a table, known to the hardware, called the Procedure Entry Point (PEP) table. The PEP itself is located at the beginning of the code segment. See Figure 2-38. The External Entry Point Table, also shown in Figure 2-38, is discussed later under the heading "Calling External Procedures". This table begins on a page boundary, with entries consecutively assigned backward toward the end of code, using the first available space that fits (either on the same page as the end of code, or on a separate page).

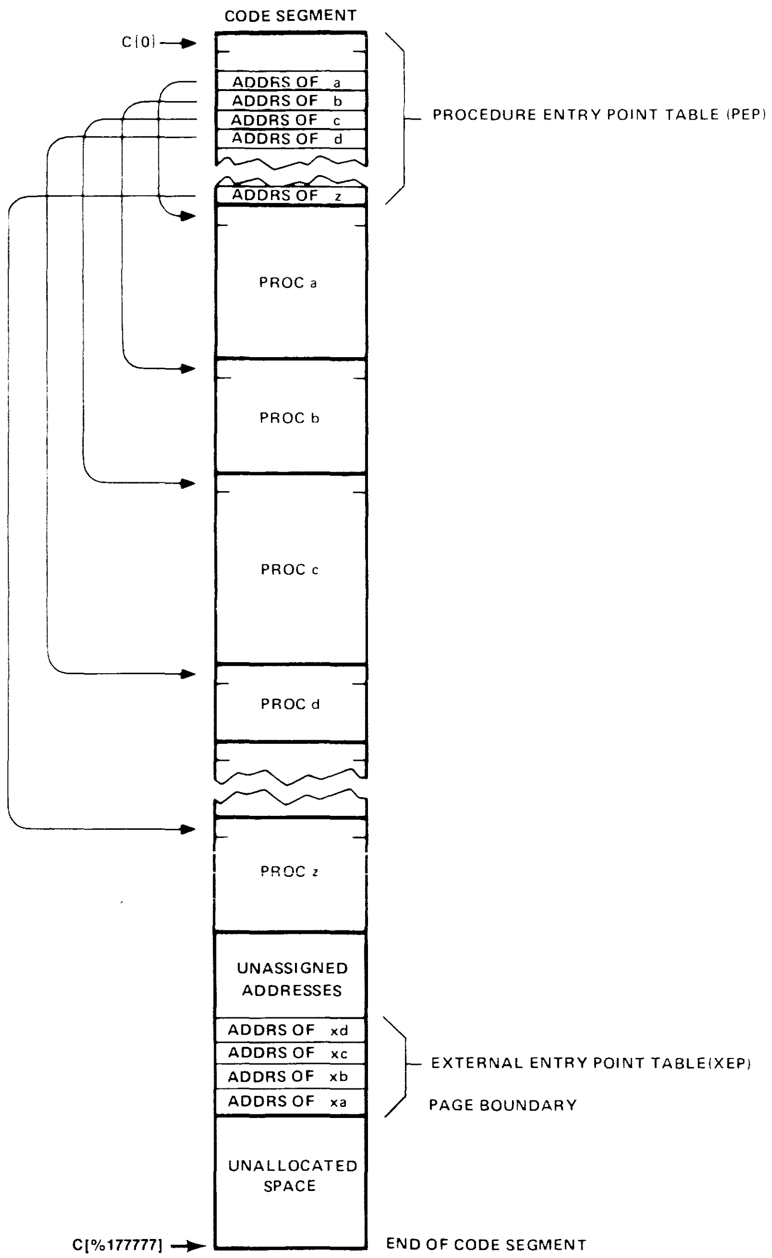


Figure 2-38. Procedure Entry Point and External Entry Point Tables

Procedures are invoked using the PCAL (Procedure Call) instruction. During PCAL execution, the caller's environment (specifically, the address of the instruction following the PCAL and the current ENV and L Register settings) is saved in a three-word stack marker. The stack marker is written at the current top of the memory stack. The PCAL instruction then references the entry in the Procedure Entry Point table corresponding to the procedure being called. The address in the PEP entry is placed in the P Register so that the next instruction executed is the instruction at the entry point of the procedure.

The last instruction that a procedure executes is an EXIT instruction. The EXIT instruction is used to return control to the caller. Specifically, the caller's ENV and L Register settings are restored and the return address (i.e., that of the instruction following the PCAL) is set into the P Register.

An example of a procedure call and exit is shown in Figure 2-39.

ATTRIBUTES. So that a nonprivileged process cannot execute in privileged mode and so that execution of privileged operations can be controlled, every procedure has one of the following attributes:

- Nonprivileged
  Procedures having this attribute are callable by any procedure in the program. They execute in the same mode (i.e., privileged or nonprivileged) as the caller. This is the attribute typically given to procedures in an application program.

- Callable
  Procedures having this attribute are also callable by any procedure in the program but execute in privileged mode (i.e., PRIV = "1"). The caller's mode is restored when a callable procedure exits. This attribute is typically assigned only to operating system procedures. It is used so that a controlled interface exists between a nonprivileged application program and the privileged operating system.

- Privileged
  Privileged procedures execute in privileged mode and are callable only by procedures currently executing in privileged mode. An attempt by a nonprivileged procedure to call a privileged procedure results in control being transferred to the operating system Instruction Failure Trap Handler. This attribute should be used only by the operating system. It is typically used when an operation, if done improperly, would have an adverse effect on processor module operation. A nonprivileged application program's only interface to an operating system privileged procedure is through a callable procedure.

In the PEP, procedure entry points are grouped according to attribute. There are three groups: the first is nonprivileged procedures, the second is callable procedures, and the last is privileged procedures.

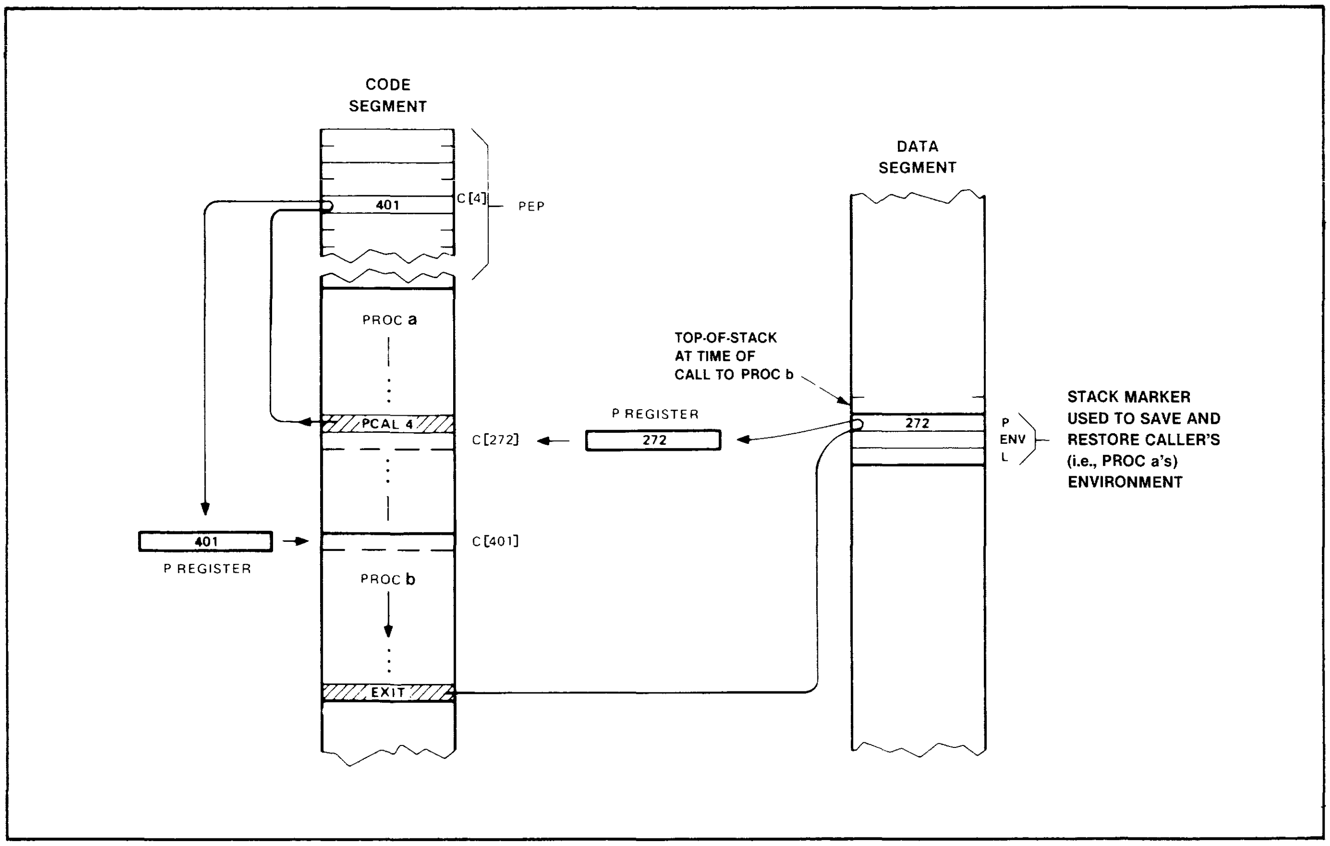

CODE SEGMENT
401
C[4]
PEP
DATA SEGMENT
PROC a
TOP-OF-STACK AT TIME OF CALL TO PROC b
P REGISTER
PCAL 4
C[272]
272
272
P
ENV
L
STACK MARKER USED TO SAVE AND RESTORE CALLER'S (i.e., PROC a's) ENVIRONMENT
401
P REGISTER
C[401]
PROC b
EXIT

Figure 2-39. Procedure Call and Exit

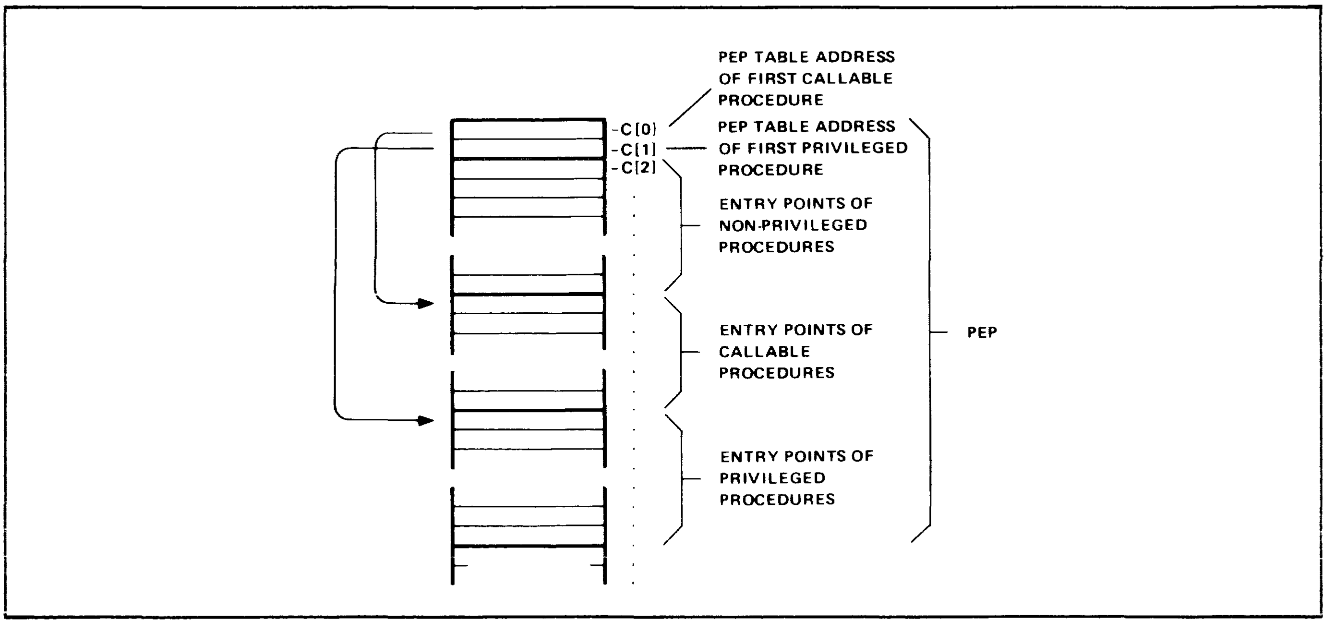

PEP TABLE ADDRESS OF FIRST CALLABLE PROCEDURE
-C[0]
-C[1]
-C[2]
PEP TABLE ADDRESS OF FIRST PRIVILEGED PROCEDURE
ENTRY POINTS OF NON-PRIVILEGED PROCEDURES
ENTRY POINTS OF CALLABLE PROCEDURES
PEP
ENTRY POINTS OF PRIVILEGED PROCEDURES

Figure 2-40. First Entries in Procedure Entry Point Table

The first two words in the PEP Table, C[0:1], describe where the callable and privileged entry points begin in the PEP. Specifically, C[0] is the address of the first PEP entry for a callable procedure, and C[1] is the address of the first PEP entry for a privileged procedure. See Figure 2-40. These words are used to check whether a nonprivileged caller is attempting to invoke a privileged procedure.

PCAL INSTRUCTION. The steps involved when a Procedure Call instruction is executed are described below, with step numbers referring to the accompanying illustration, Figure 2-41. Note that before the PCAL executes, the procedure parameters (and the mask word or words, for procedures with a variable number of parameters) must be pushed onto the stack.

1. The caller's environment is saved in a three-word stack marker.

   ```
   data [S+1] := P;    !
   data [S+2] := ENV;  ! stack marker.
   data [S+3] := L;    !
   ```

   The stack marker is stored in the top-of-stack location plus one as indicated by the address in the S Register. The stack marker contains the following information:

   - the current P Register setting (the address of the instruction following the PCAL)
   - the current ENV Register setting
   - the current L Register setting (the beginning of the caller's local data area).

2. If the calling procedure is not executing in privileged mode, the "callability" attribute of the procedure being called is checked.

   First, the PEP Number field of the PCAL instruction is compared with the entry in C[0] (the address of the first PEP entry for callable procedures). If the PEP Number is greater than or equal to the C[0] entry, then this is a call to a callable or privileged procedure, so a second check is made: the PEP Number field of the PCAL instruction is compared with the entry in C[1] (the address of the first PEP entry for privileged procedures). If the PEP Number is greater than or equal to the entry in C[1], then this is a call to a privileged procedure, so an Instruction Failure trap occurs and the PCAL instruction aborts. Otherwise, this is a call to a callable procedure, so the PRIV bit is set.

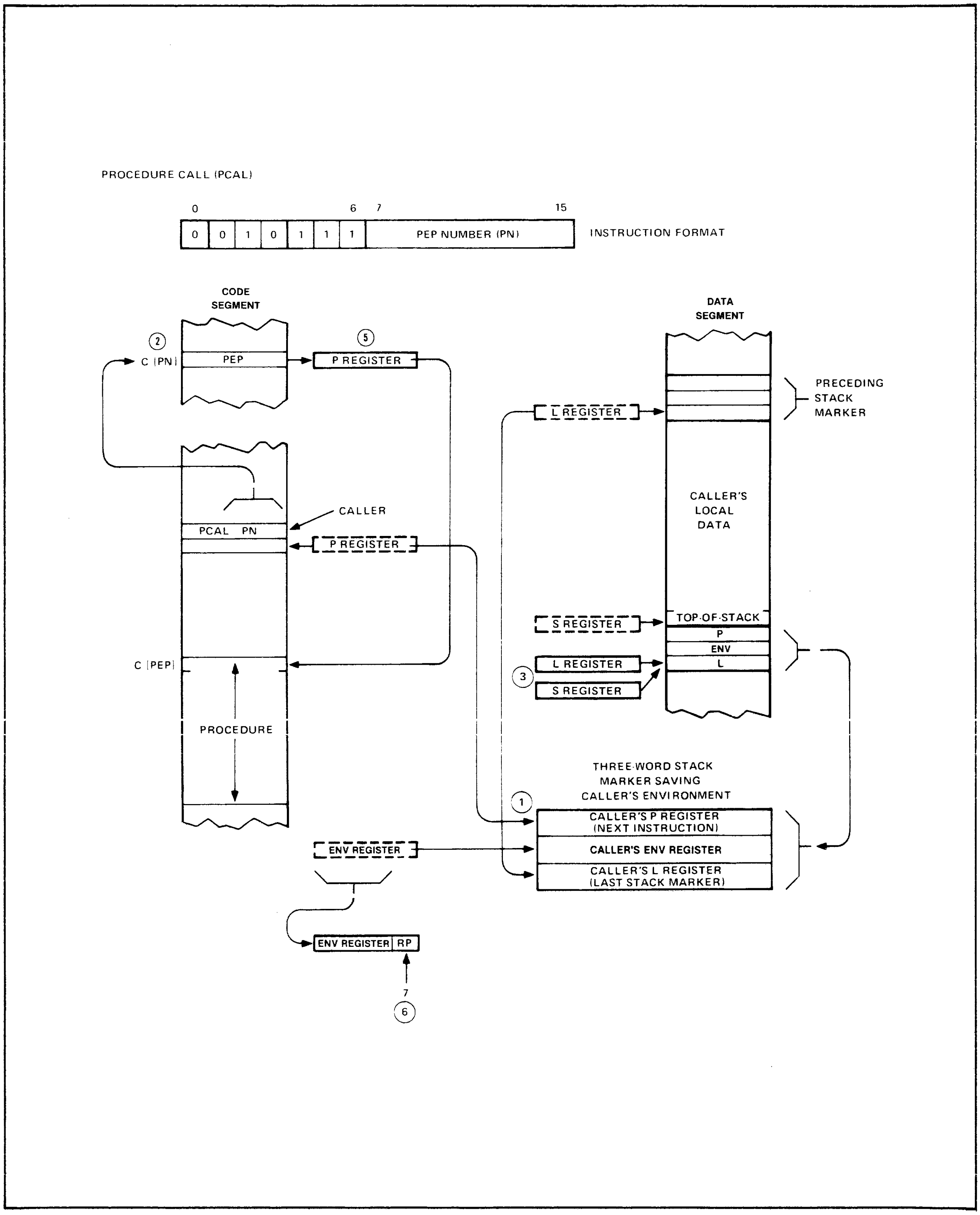


Figure 2-41. Execution of PCAL Instruction

3. The S and L Registers are set with the G[0]-relative address of the new top-of-stack location (the third word of the stack marker).

   ```
   L := S := S+3;
   ```

   The new L Register setting defines the base of the local area for the procedure being called.

4. The new S Register setting is tested for an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the operating system Stack Overflow trap (and the PCAL instruction is aborted).

   ```
   if S '>' 32767 then stack^overflow^trap;
   ```

5. The C[0]-relative address of the procedure being called is obtained from the PEP table entry pointed to by the <PEP number> field in the PCAL instruction. This address is put in the P Register so that the next instruction executed will be the first instruction of the called procedure.

6. Finally, the Register Stack Pointer, RP, is given an initial value of seven (stack empty).

   ```
   RP :=  7;
   ```

Following the PCAL, the instructions comprising the procedure are executed. The last instruction that a procedure executes is an EXIT instruction.

EXIT INSTRUCTION. The EXIT instruction uses the three-word stack marker to restore the caller's environment. The sequence is as follows, with reference to Figure 2-42.

1. The S Register setting is moved below the local area, the stack marker, and any parameters to the exiting procedure.

   ```
   S := L - <S decrement>;
   ```

   The <S decrement> value is subtracted from the current L register setting and placed in the S Register. The value of <S decrement> is three (for the stack marker) plus the number of words of parameter and mask information passed to the exiting procedure.

2. The P Register is set with the P Register value saved in the stack marker at L[-2].

   ```
   P := data [L-2];
   ```

   The next instruction to be executed will be the one following the PCAL instruction.

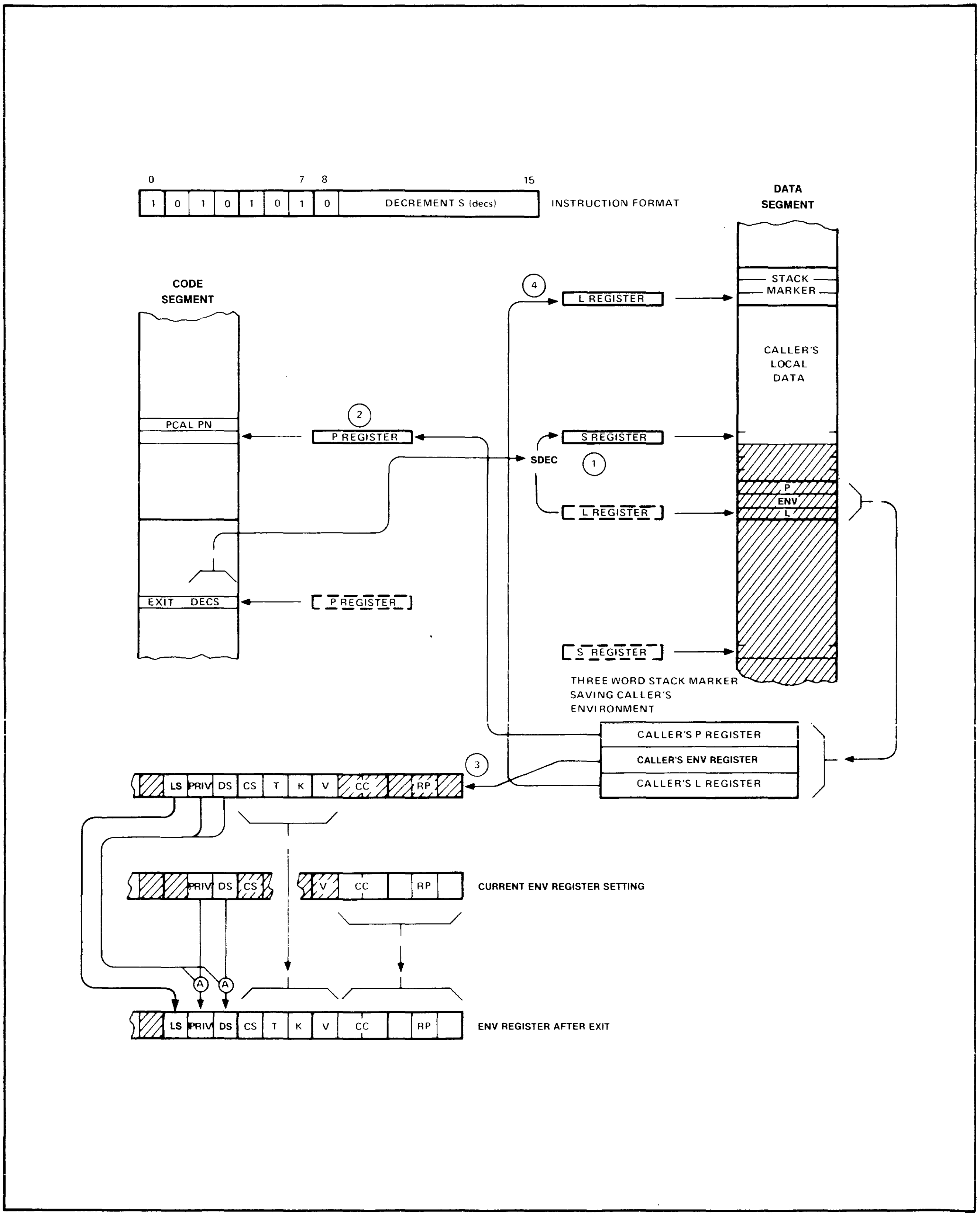


Figure 2-42. Execution of EXIT Instruction

3. The ENV Register is restored from a combination of the current ENV Register setting and the ENV Register value saved in the Register Stack at L[-1].

   The mode (privileged or nonprivileged) and data area are reestablished to be the lesser of the caller's and the current settings. This is so that a nonprivileged user cannot exit with privileged capability. The caller's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) are reestablished from L[-1]. Z and N (Condition Code) are left at their current settings to reflect the results of the call. RP is left at its current setting so that a value in the Register Stack can be returned to the caller.

4. The L Register is restored from the L Register value saved in the stack marker at L[0].

   ```
   L := data [L];
   ```

   This moves L back to point to the preceding stack marker, thereby reestablishing the preceding local data area.

The instruction following the PCAL instruction then executes.

Memory Stack Operation

Figures 2-43a and b depict an example of a memory stack operation from an initial state (i.e., start of process execution) through a call to, and subsequent return from, a procedure. The purpose of the diagram is to show the action of the L and S Registers as a procedure generates its local variables and prepares to call a procedure by passing parameters, how L and S are set when a procedure is called, and how L and S are set when the return is made to the caller.

1. Initial State

   After the operating system has loaded a program into memory but before the first instruction of the process executes, the following initial conditions are present: the process's global variables are initialized and present, and the L and S Registers are set to the address of the word just above the global area. There are no local variables defined at this time.

2. Proc "A" generates its local variables

   The first few instructions of a procedure generate the procedure's local variables. As the local variables are generated, the S Register setting increases, defining a new upper limit to the procedure's local area. Note that the L Register setting does not change.

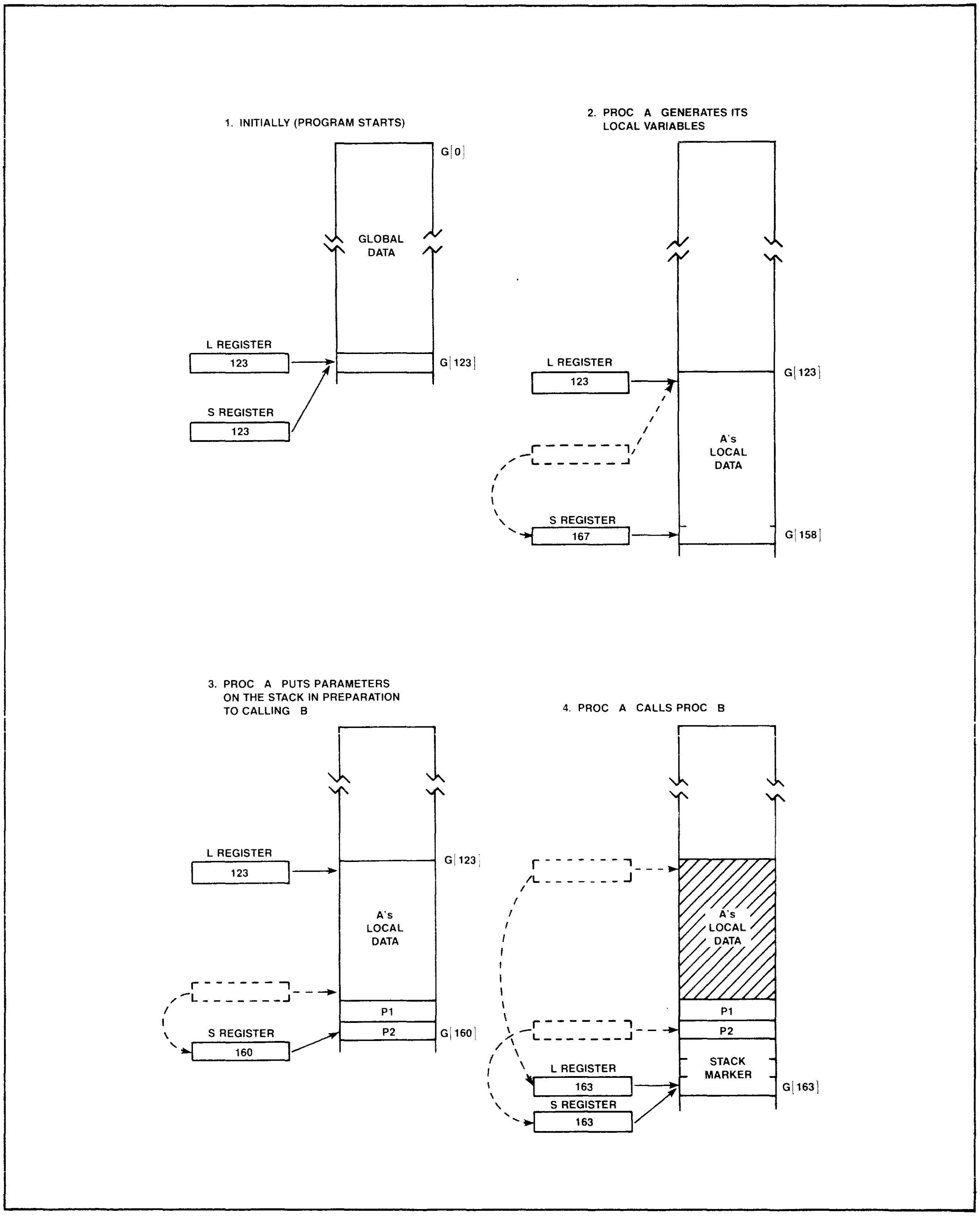


Figure 2-43a. L and S Registers in Procedure Calls

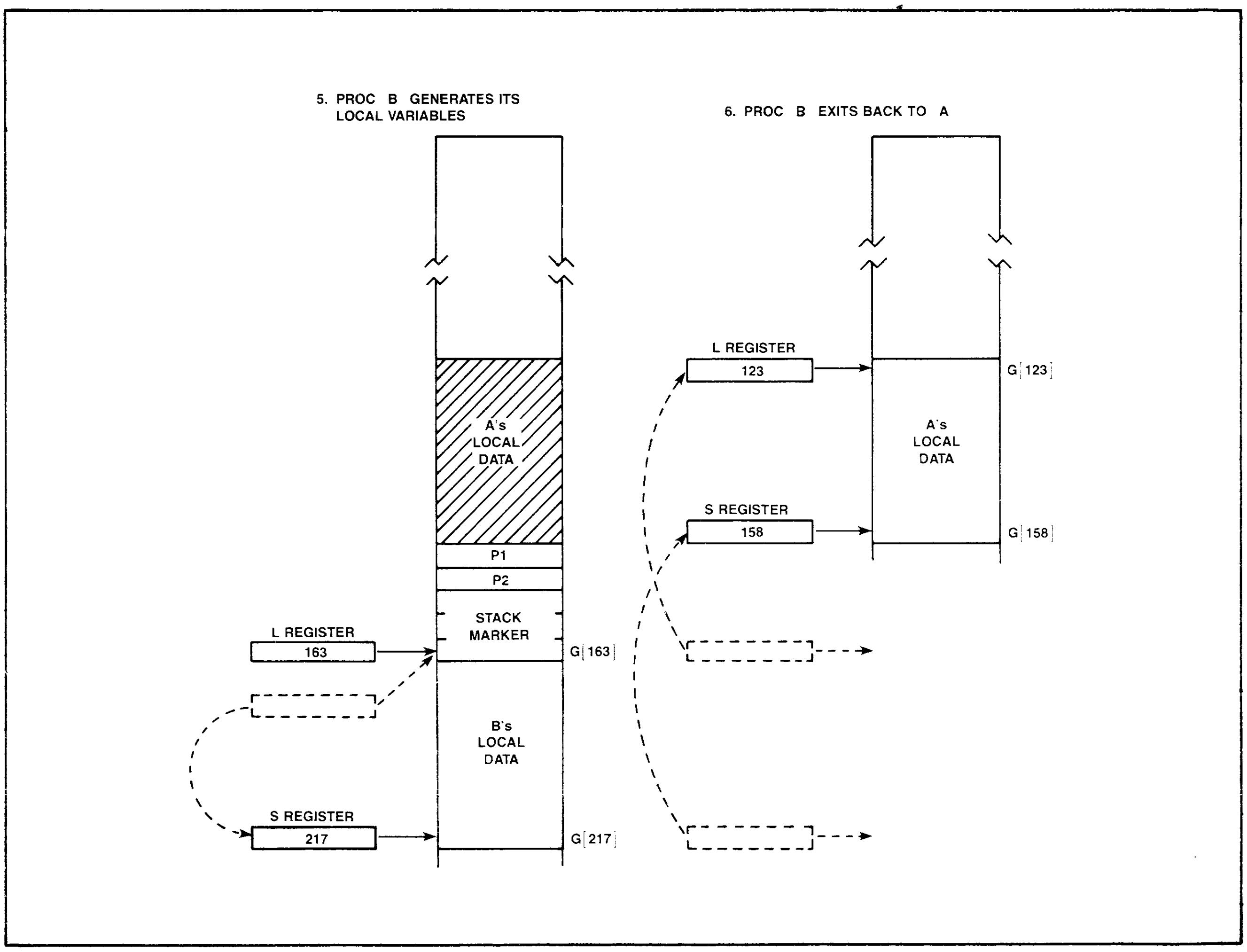


Figure 2-43b. L and S Registers in Procedure Calls

3. Proc "A" passes parameters to "B"

   In preparation for calling the procedure "B", the parameter words (two in this example) are placed on the top-of-stack location as indicated by the S Register setting. The S Register setting is increased by two to account for the parameters.

4. "A" calls "B"

   After the parameters are loaded onto the memory stack, a PCAL instruction is executed. Execution of the PCAL instruction places a three-word stack marker at the current S Register setting plus one (just above the parameters). L and S Registers are given a new setting; they both point to the third word of the stack marker. The new L Register setting defines the start of "B's" local area. At this point, no local variables have been generated for the procedure "B". (Note that "A's" local area, which is normally addressed relative to the L Register, is no longer addressable by the L-plus addressing mode.)

5. Proc "B" generates its local variables

   In the same manner as procedure "A" did, procedure "B" generates its local variables. This increases the S Register setting accordingly so that the S Register defines the new upper limit to "B's" local area.

6. Proc "B" exits back to proc "A"

   When procedure "B" completes, an EXIT instruction is executed to return to "A". Execution of the EXIT instruction moves the L Register setting back to the beginning of "A's" local area and moves the S Register setting back to the top-of-stack location that was in effect before the parameters were loaded on the stack (this is accomplished by the <S decrement> value in the EXIT Instruction). Specifically, for the return to the procedure "A", the EXIT instruction is

   ```
   EXIT 5
   ```

   This deletes the three-word stack marker from the top-of-stack plus the two parameter words.

GENERATION OF AND ACCESS TO LOCAL DATA. Unlike the global data area, which exists at all times, the local data area for a procedure exists only while the procedure is actually executing. The local variables are generated and initialized by instructions at the start of a procedure's code. Thus a procedure can be called any number of times (and in fact can call itself) and each call generates a fresh copy of the procedure's local data area.

An example of the instructions used to generate the following local variables will next be considered (referring to Figure 2-44):

```
INT i,            !  L[1]
    j := 5,       !  L[2]
    .k [0:31];    !  L[3]  (pointer to k, which starts at L[4])
```

These are three local variables declared in a TAL source program: "i" is a one-word uninitialized variable, "j" is a one-word variable initialized with the value 5, "k" is an indirectly addressed array variable consisting of 32 words. The instructions to generate these variables are:

```
ADDS    +001     ! Add to S
LDI     +005     ! Load Immediate
LADR   L+004     ! Load Address
PUSH     711     ! PUSH to Memory
ADDS    +040     ! Add to S
```

The ADDS instruction increments the S Register setting by one. This allocates one word for the variable "i".

The LDI instruction puts the initialization value for "j" (5) on the top of the Register Stack.

The LADR instruction calculates the G[0]-relative address of the first word of the indirect array "k" and puts the address on the top of the Register Stack.

The PUSH instruction performs two functions: 1) it puts both the initialization value in "j" and the address of the array "k" into L[2] and L[3] of the process's stack, respectively, and 2) increments the S Register setting by two to allocate the two words needed for "j" and the address pointer to "k".

The ADDS instruction increments the S Register setting by 32 (octal 40). This allocates 32 words for the indirect array "k".

Following the generation of the local variables, the local area for this example consists of:

```
L[1]    = i
L[2]    = j (initialized with a value of 5)
L[3]    = an address pointer to the array "k"
L[4:35] = the array "k"
```

Once allocated, data in the local area is addressed relative to the current L Register setting using the L-plus addressing mode. As illustrated, this mode can access local data directly, or can use the direct address as an address pointer (indexing is also permitted).

The top-of-stack area is addressable implicitly through use of the PUSH and POP instructions. These are illustrated in Figure 2-45. The PUSH instruction is used to store the Register Stack contents,

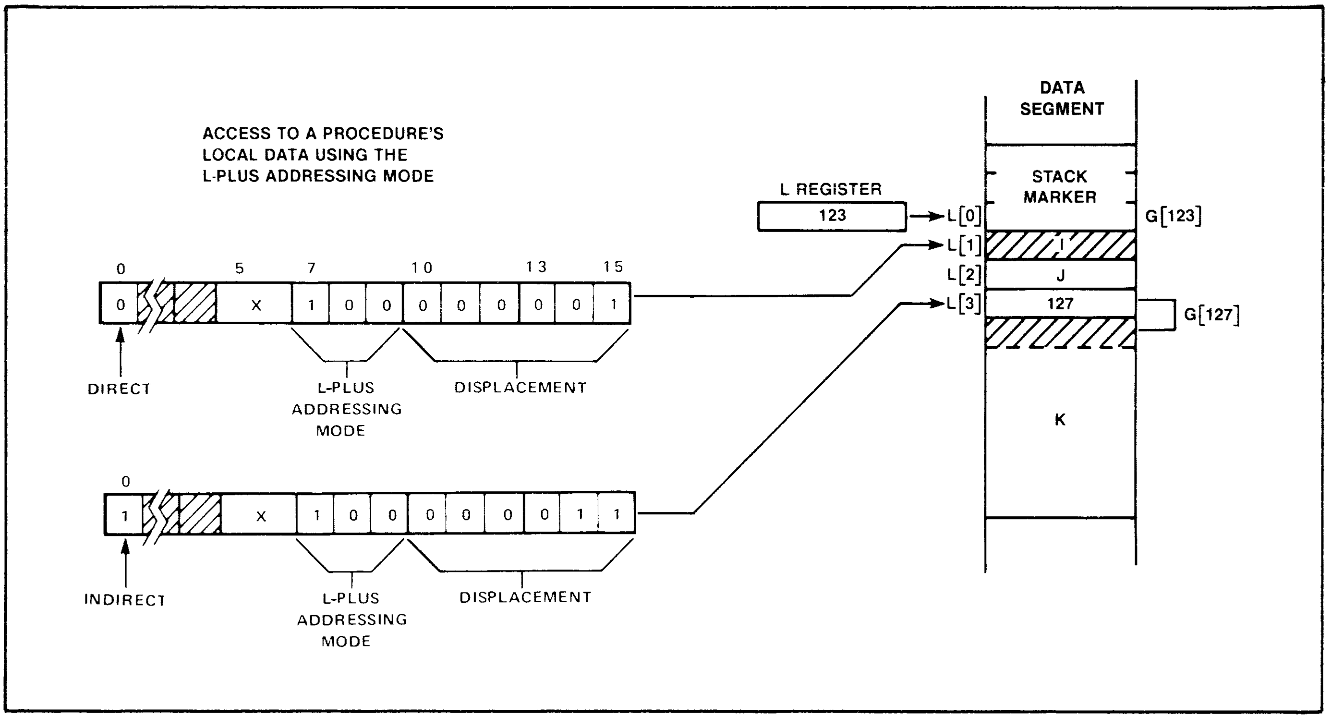


Figure 2-44. L-Plus Addressing Mode

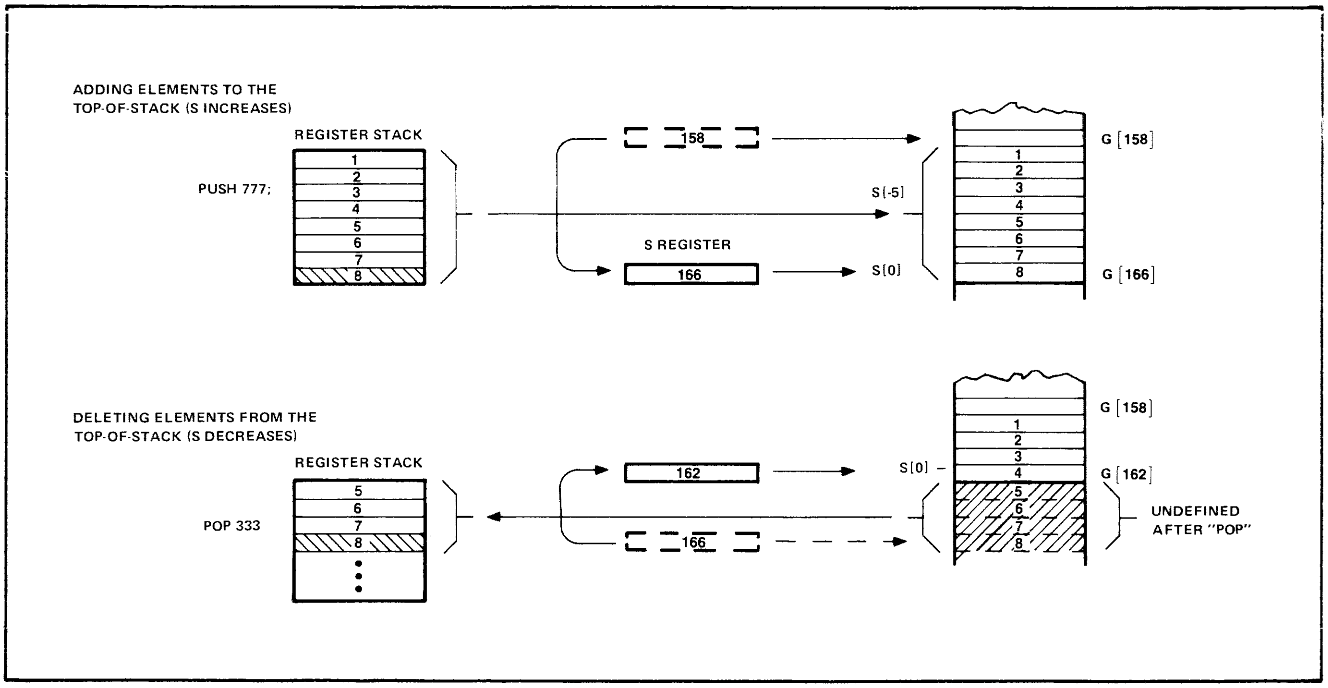


Figure 2-45. PUSH and POP Instructions

usually prior to calling a procedure, on the top of the memory stack. When a PUSH instruction is executed, the S Register setting is incremented by the number of words pushed. The POP instruction is used to restore the Register Stack contents from the top of the memory stack, then decrement the S Register setting accordingly.

PARAMETER PASSING. Parameters are passed to a procedure in the top-of-stack area. Naturally, there must be coordination between the caller and the called when passing parameters. The caller must know the order in which a procedure expects parameters, and whether a parameter is to be an actual operand (called a "value" parameter) or an address pointer (called a "reference" parameter).

Before the caller invokes a procedure, the parameters are prepared in the Register Stack. The actual operands (for value parameters) and the addresses of operands (for reference parameters) are loaded into the Register Stack in the order required by the procedure being called. The address of a reference parameter is obtained by the execution of an LADR (load address) instruction. The parameters that have been prepared in the Register Stack are loaded on the top of the memory stack by executing a PUSH instruction (which increments the S Register accordingly).

An example will now be considered to show the instructions used to prepare the top of the memory stack area for parameter passing. This example uses the variables declared in the preceding example, and is illustrated in Figure 2-46. The procedure being called is of the form:

```
PROC b (pl,p2);
   INT pl,.p2;
```

Parameter "pl" is a value parameter, therefore the procedure expects an actual value to be passed. Parameter "p2" is a reference parameter and, therefore, the procedure expects the G[0]-relative address of a variable to be passed.

The call being made from procedure "A" is:

```
CALL b (j,i);
```

The instructions to pass these two parameters are:

```
LOAD L +002
LADR L +001
PUSH 711
```

The LOAD instruction puts the contents of the variable "j" (the value 5) on the top of the Register Stack. (This is the parameter passed as "pl", a value parameter, to "B".)

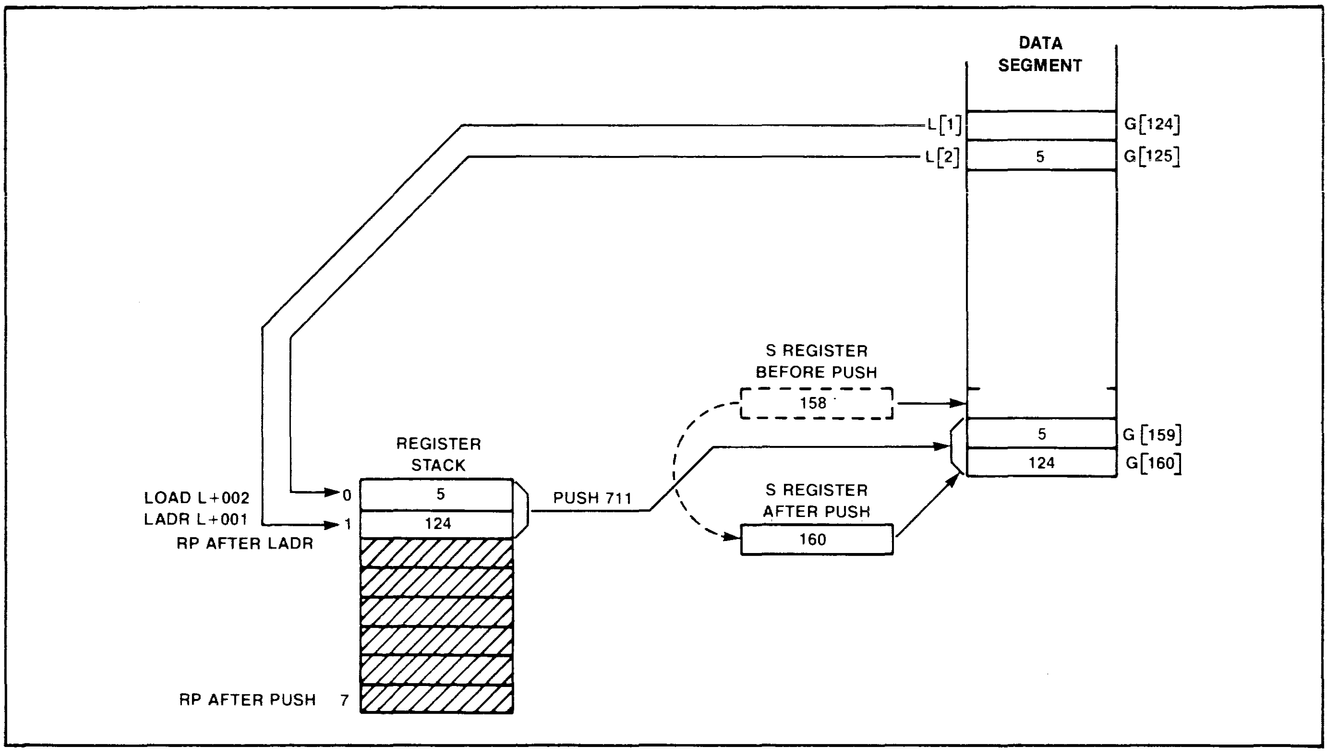


Figure 2-46. Parameter Passing

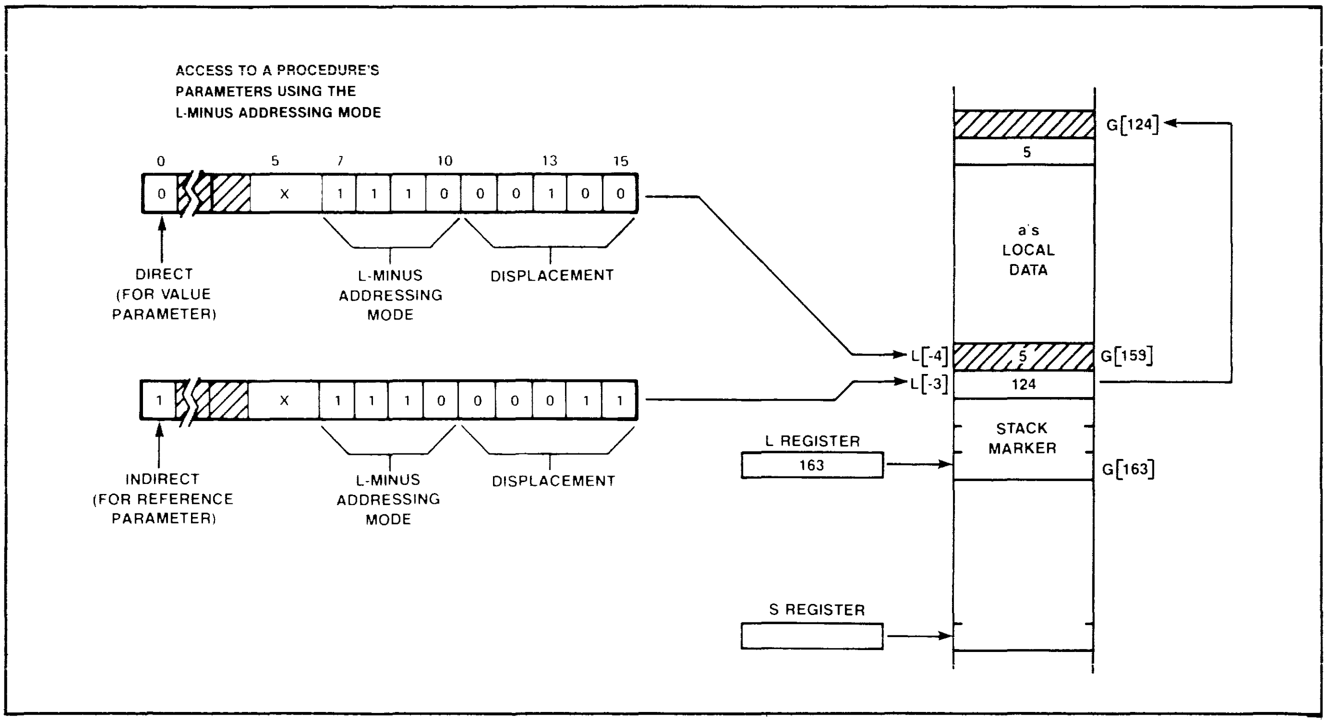


Figure 2-47. Parameter Access

The LADR instruction calculates the G[0]-relative address of the variable "i" and puts the address on the top of the Register Stack. (This is the parameter passed as "p2", a reference parameter, to "B".)

The PUSH instruction places the two parameters from the Register Stack on the top of the memory stack and increments the S Register setting by two.

PARAMETER ACCESS. Parameters are accessed by using the L-minus addressing mode. This mode provides access to the 32 locations just below and including the current L Register setting (L[-31:0]). Subtracting the three words used for the stack marker, this leaves 29 words addressable as parameters. If value parameters are passed, the parameter location is addressed directly (<i>, indirect, bit of a memory reference instruction = 0); if reference parameters are passed, the parameter location is used as an indirect address (<i> bit = 1). Indexing in either mode is permitted.

Figure 2-47 shows an example of both value and reference parameter access.

RETURNING A VALUE TO THE CALLER. A procedure can return a value to its caller via the top of the Register Stack. This, like parameter passing, requires coordination between the caller and the called. That is, the calling procedure must know the element size of the return value (i.e., number of words comprising the value).

The following paragraphs describe an example of a procedure, named "f", that returns a value, and the instructions used to do so. The example is illustrated in Figure 2-48.

The procedure is of the form:

```
   INT PROC f (x);
      INT x;

     BEGIN
       RETURN x * x;
     END;
```

This procedure returns the square of a number, "x". The instructions to return the square of "x" are:

```
   LOAD L -003      !  parameter x is obtained from L-003
   LOAD L -003      !  load another copy of x
   IMPY             !  squared result now exists in R[0]
   EXIT 4           !  delete stack marker and parameter x
```

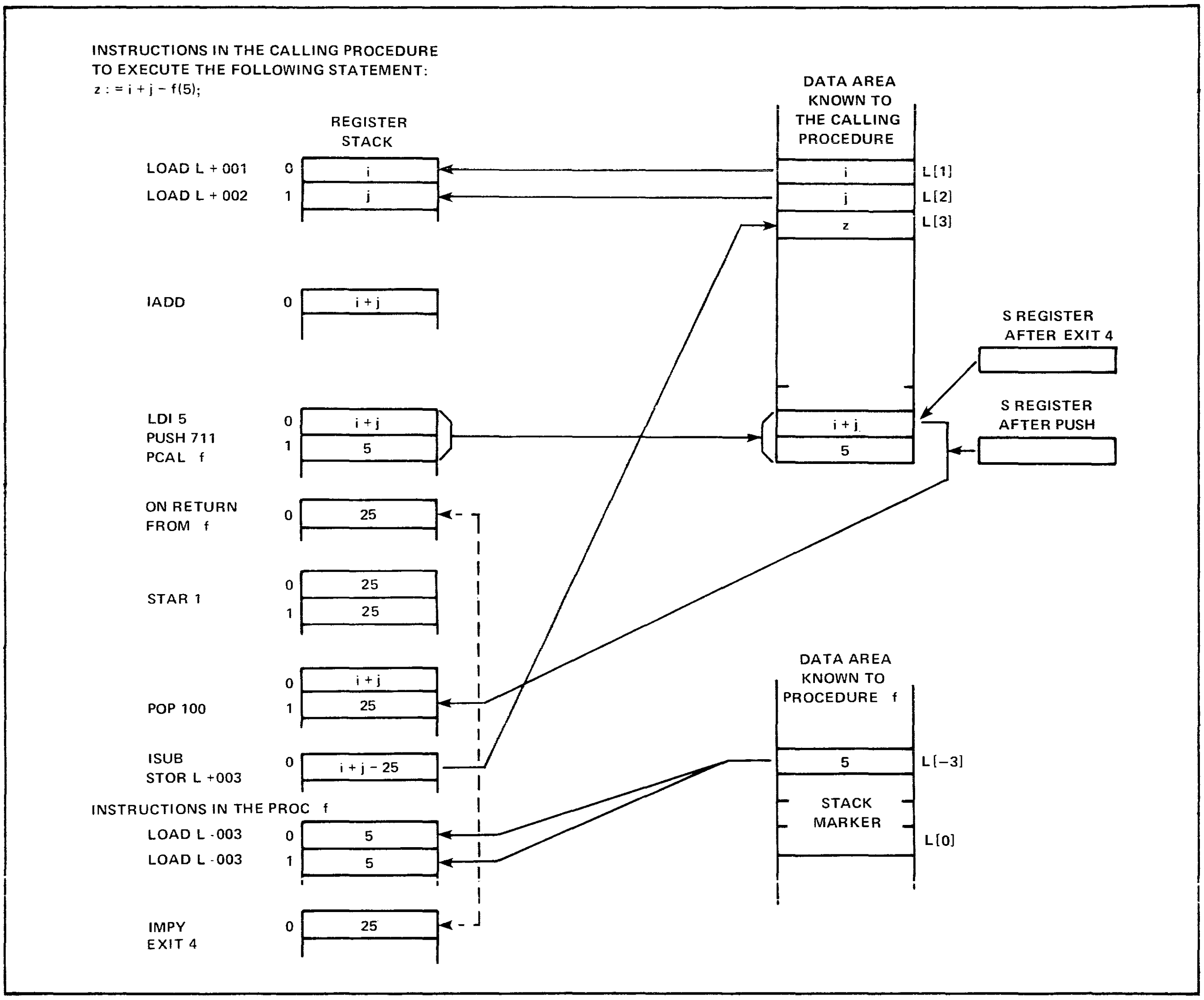


Figure 2-48. Value Returned via Register Stack

The first LOAD instruction loads the parameter "x" onto the top of the Register Stack. Following the LOAD, the RP setting is 0. (The RP setting is 7 when a procedure begins executing.) The second LOAD again loads the parameter "x". Following this load, the RP setting is 1.

The IMPY instruction multiplies the values in the Register Stack, leaving the result of the multiplication in R[0]. Following this operation, the RP setting is 0.

The EXIT instruction causes a return to the caller, deleting the parameter and stack marker (1 + 3 = 4), but leaving the squared value on the top of the stack.

A call is now made to procedure "f", as follows:

```
z := i + j - f(5);
```

That is, subtract the square of 5 from the sum of the contents of the variables "i" and "j" then store the result in the variable "z". Variables "i", "j", and "z" are local variables at L[1], L[2], and L[3] respectively.

The instructions to perform this operation are:

```
LOAD  L +001   ! load "i"
LOAD  L +002   ! load "j"
IADD           ! "i" + "j"
LDI     +005   ! load parameter to "f"
PUSH     711   ! push sum and parameter onto memory stack
PCAL           ! procedure call to "f"
STAR       1   ! move returned value from R[0] to R[1]
POP      100   ! bring saved sum back to R[0]
ISUB           ! subtract returned value from "i+j" sum
STOR  L +003   ! store result into "z"
```

The first three instructions calculate the sum of "i" + "j" and leave the result in R[0]. The LDI +005 instruction loads the parameter to "f" onto the top of the Register Stack at R[1].

The PUSH instruction pushes R[0:1] onto the memory stack. Following the PUSH, the two top-of-memory-stack locations contain:

```
S[-1] = sum of "i" + "j"
S[0]  = 5, the parameter to "f"
```

This clears the register stack for use by the procedure which now is invoked by the PCAL instruction. On the return from "f", R[0] of the Register Stack contains the square of 5.

The STAR instruction moves the return value in the R[0] register stack location to R[1] in preparation for the subtraction from the sum of "i" + "j".

The POP 100 instruction brings the sum of "i" + "j" (calculated previously) into R[0] and sets RP to 1 (to point to the returned value).

The ISUB Instruction subtracts the return value of "f" from the sum of "i" + "j". The STOR instruction stores the result in the variable "z", and RP becomes 7.

STACK MARKER CHAIN. In examples shown previously, only one procedure call occurred and, therefore, only one stack marker was generated. However, in practice, there may be several stack markers (and local areas) present in a memory stack at once. This occurs when a called procedure calls another procedure and that procedure calls still another procedure, etc. The nature of this "chain" of stack markers and the action of the L and S Registers is such that the returns are always made in the reverse order of the calls, and the local data areas are redefined as the returns are made.

Figure 2-49 shows the condition of a memory stack after the following calls have taken place:

```
In procedure "a", CALL b;

    In procedure "b", CALL c;

        In procedure "c", CALL d;
```

The procedure "d" is currently executing.

Specifically, the L Register, which is given a new (higher) setting when a procedure is called, and the local data areas, which are allocated and generated relative to the current L Register setting, result in a stack of procedure environments that are physically placed in the chronological order in which the calls were made. (Remember, when a procedure is called, the stack marker is placed at the current S Register setting plus one. In this manner, a procedure's local data is always retained when it calls another procedure.) The stack markers, which contain the environment of the preceding procedure (and point to the preceding stack marker) restore the preceding environments in the reverse order of the calls.

SUBPROCEDURES. Subprocedures are invoked using the BSUB (branch to subprocedure) instruction. Because the BSUB is a branching-type instruction, the subprocedure entry point is calculated as a self-relative address. Execution of the BSUB instruction differs from other branching instructions in that it places a return address on the top of the memory stack. See Figure 2-50. Note that before the BSUB executes, the subprocedure parameters must be pushed onto the stack.

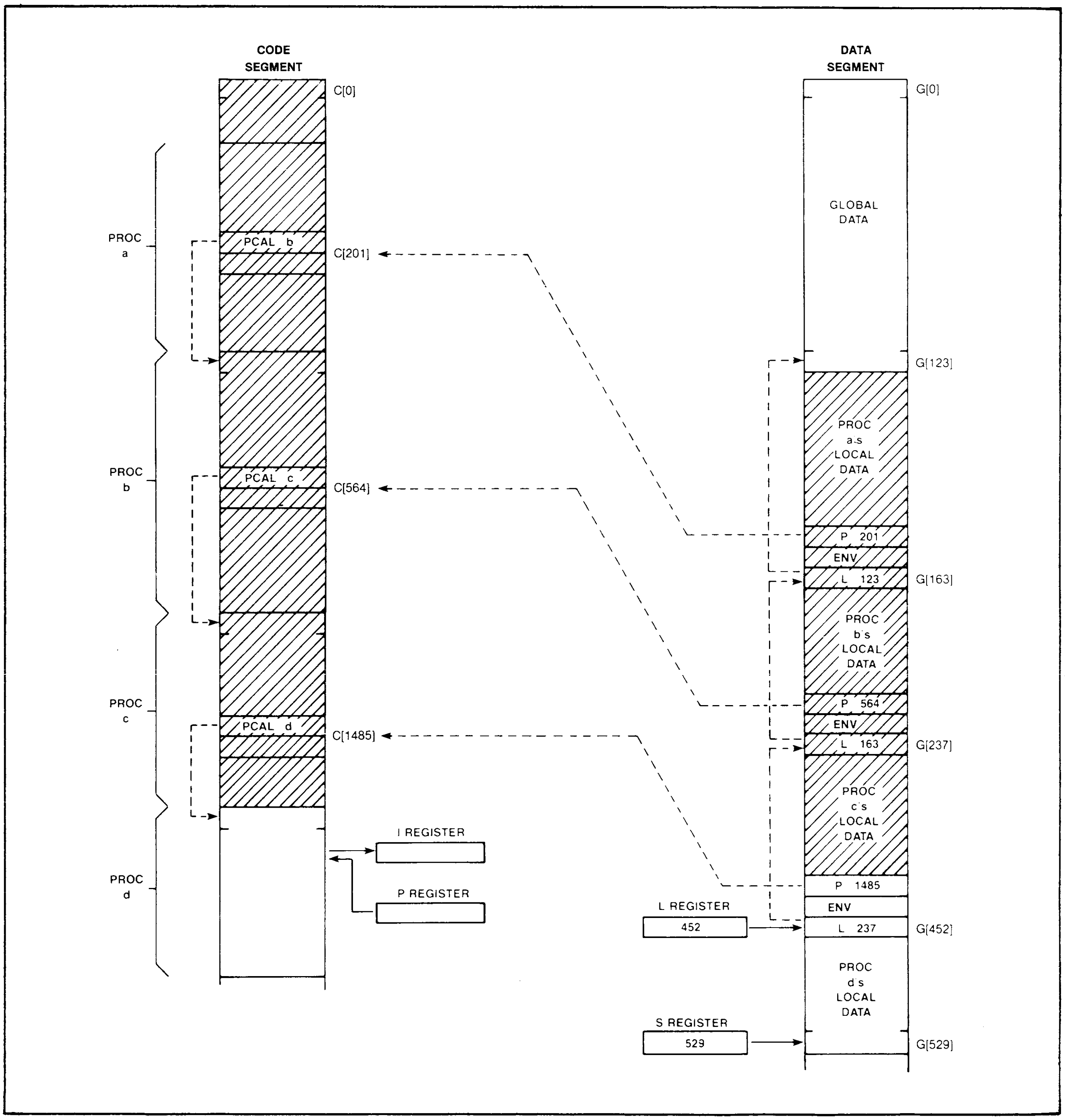


Figure 2-49. Stack Marker Chain

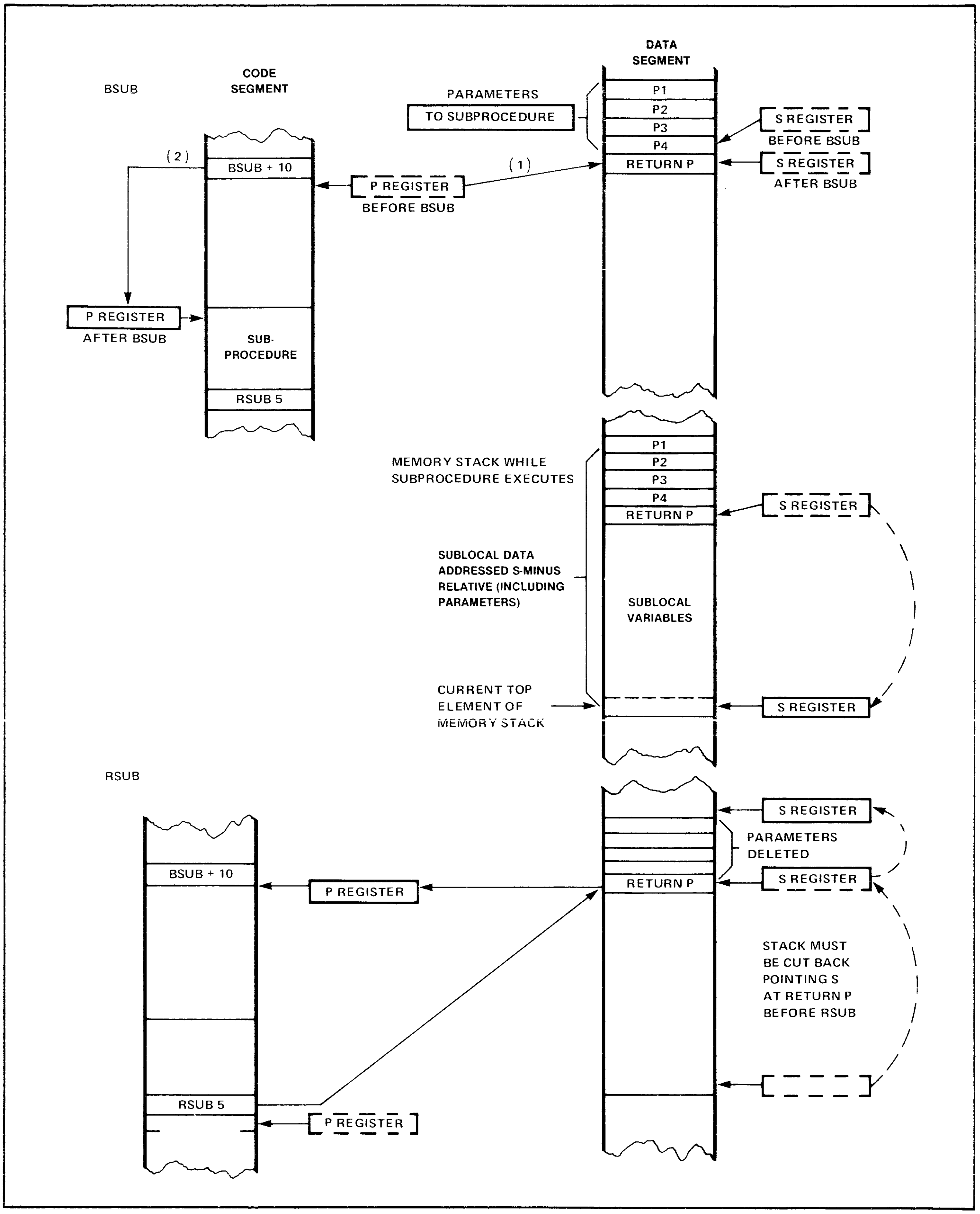


Figure 2-50. Subprocedure Calls

Specifically, the steps involved when a BSUB instruction is executed are as follows:

1. The return address (i.e., that of the instruction following the BSUB) is placed on the top of the memory stack.

   ```
   S := S + 1;
   data[S] := P;
   ```

2. The self-relative branch address of the subprocedure is put into the P Register.

   ```
   P := branch^address;
   ```

The last instruction that a subprocedure executes is an RSUB (return from subprocedure) instruction. The RSUB instruction returns control to the instruction following the BSUB instruction by putting the return address, at the current top of memory stack location, into the P Register:

```
P := data [S];
S := S - <S decrement>;
```

The <S decrement> value is used to move the S Register setting below the sublocal data area. <S decrement> is at least one, to account for the one-word return address.

The sublocal data area consists of a subprocedure's variables and parameters. It is addressable using the S-minus addressing mode, shown in Figure 2-51. This provides direct access to the 32 locations including and below the current S Register setting (i.e., S[-31:0]).

LOGICAL MEMORY

Logical memory (for nonprivileged users using nonextended addressing) is separated into six segments, each of which is defined by its own map. These six segments, as shown in Figure 2-52, are:

Map	Segment
0	User Data
1	System Data
2	User Code
3	System Code
4	User Library Code
5	System Code Extension

The memory segments defined by the odd-numbered maps (1, 3, 5) contain the GUARDIAN operating system. Since there is only one operating system in a processor, this is a permanent assignment of maps. The memory segments defined by the even-numbered maps (0, 2, 4) contain

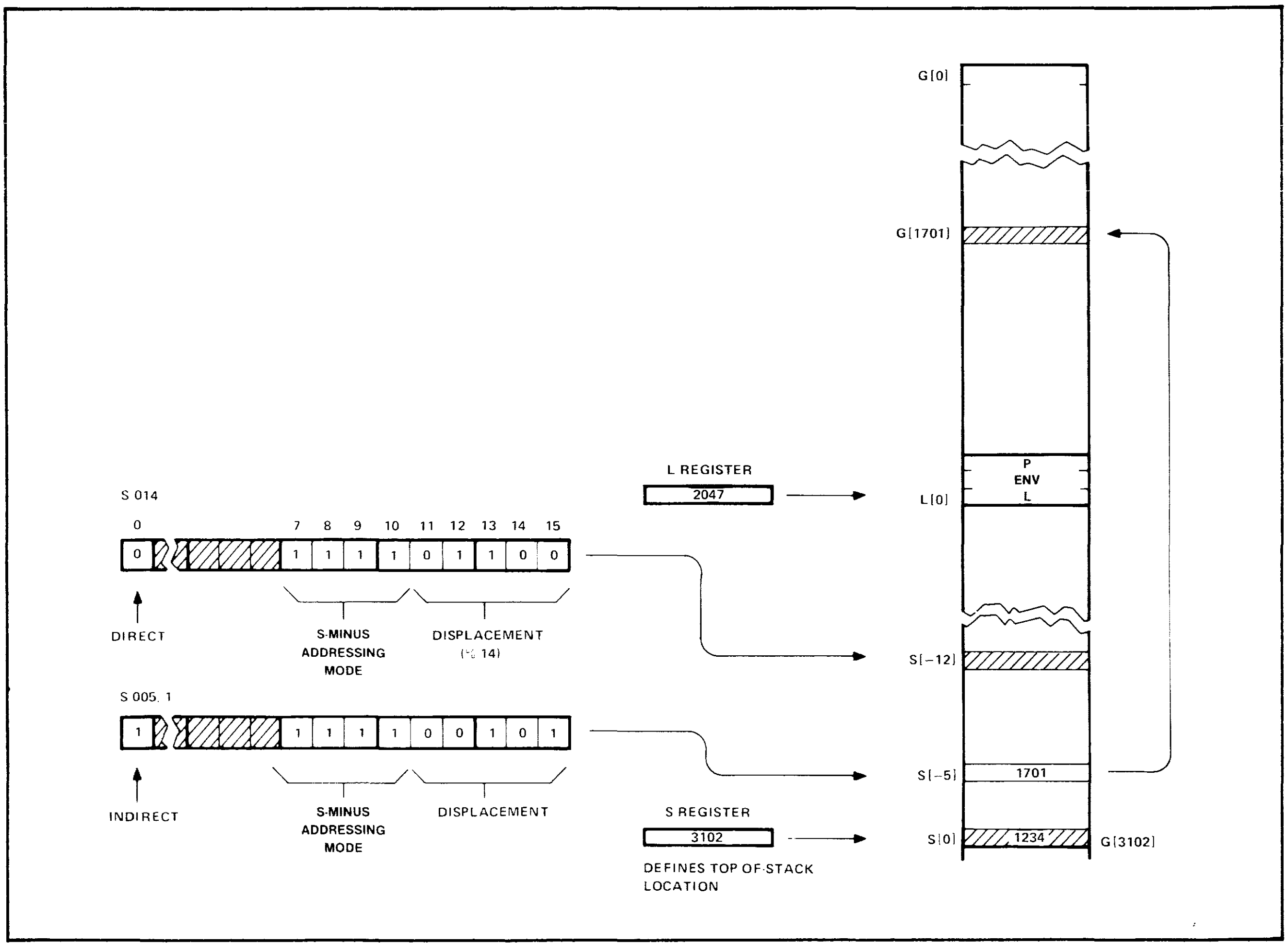


Figure 2-51. Example of S-Minus Addressing

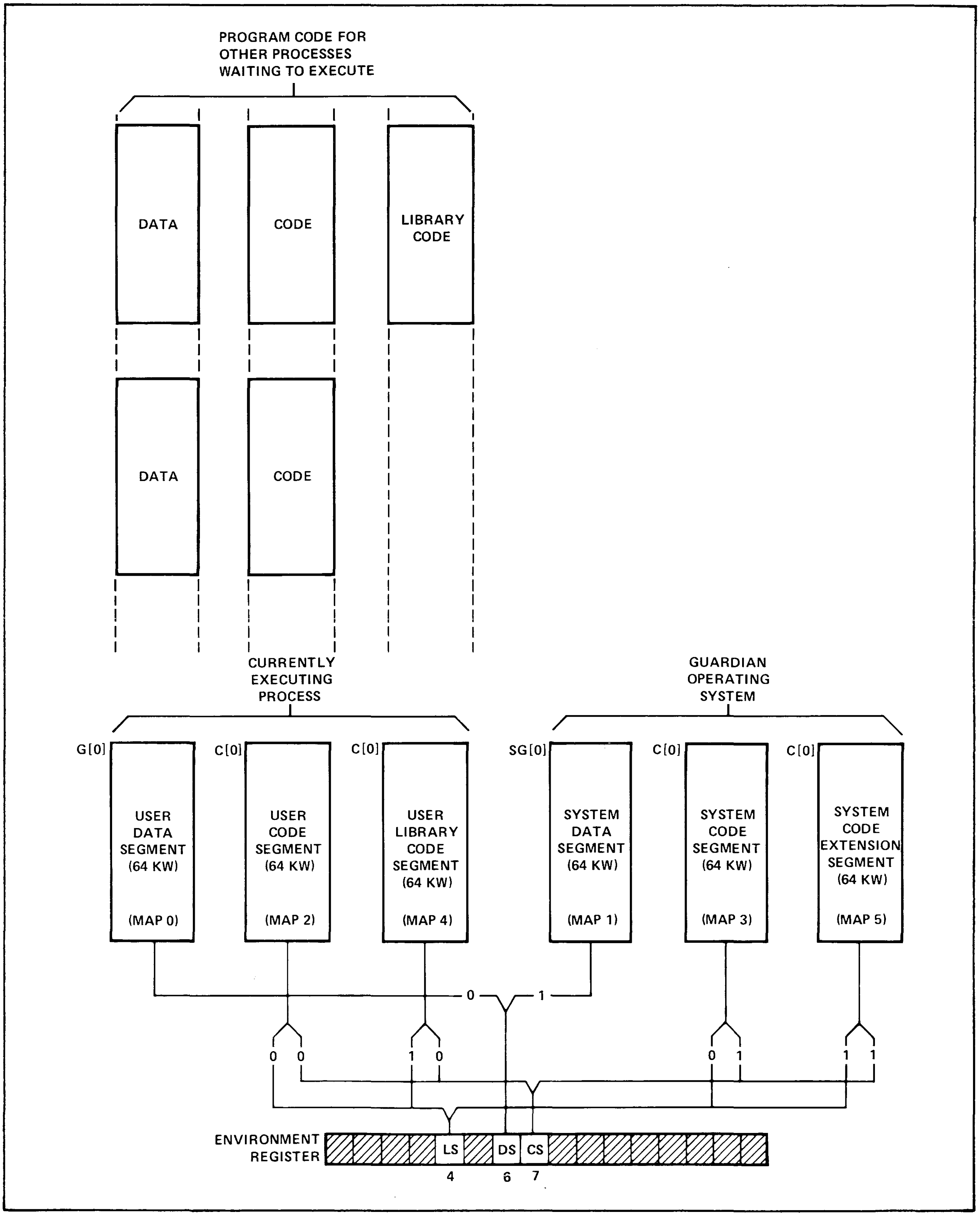

PROGRAM CODE FOR OTHER PROCESSES WAITING TO EXECUTE
DATA
CODE
LIBRARY CODE
DATA
CODE
CURRENTLY EXECUTING PROCESS
GUARDIAN OPERATING SYSTEM
G[0]
USER DATA SEGMENT (64 KW) (MAP 0)
C[0]
USER CODE SEGMENT (64 KW) (MAP 2)
C[0]
USER LIBRARY CODE SEGMENT (64 KW) (MAP 4)
SG[0]
SYSTEM DATA SEGMENT (64 KW) (MAP 1)
C[0]
SYSTEM CODE SEGMENT (64 KW) (MAP 3)
C[0]
SYSTEM CODE EXTENSION SEGMENT (64 KW) (MAP 5)
0
1
0 0
1 0
0 1
1 1
ENVIRONMENT REGISTER
LS
DS
CS
4
6
7

Figure 2-52. Logical Memory

the code and data of the currently executing process. Since many processes typically exist in a processor (including user application processes, i/o processes, compiler processes, GUARDIAN processes, etc.), the actual code and data indicated by these maps switches each time a different process comes into execution. Every such process performs its addressing relative to its own G[0] and C[0] bases.

For any single memory-referencing instruction, only one code segment and one data segment can be used. This selection, from among the six segments of logical memory, is made by the existing state of three bits in the Environment Register. As shown in Figure 2-52, the selection of a data segment is made by the state of the DS bit (bit 6). If DS is a "1", the System Data segment is accessed by the instruction; if DS is a "0", the User Data segment is accessed. The selection of a code segment is made by the combined states of the LS and CS bits, as follows:

LS	CS		
0	0	User Code	(Map 2)
0	1	System Code	(Map 3)
1	0	User Library Code	(Map 4)
1	1	System Code Extension	(Map 5)

The User Code and System Code segments defined by Maps 2 and 3 are referred to as the "standard" code segments, whereas the alternate code segments defined by Maps 4 and 5 (User Library Code and System Code Extension) are referred to as the "library" code segments. There is some difference in the way the library segments are used by a user and by the system, in that the user's library segment contains procedures that all belong to one program; on the other hand, the system's code extension segment is simply an extension of the standard system code segment, altogether containing the many procedures that make up the GUARDIAN operating system. This code resides in the two memory segments defined by Maps 3 and 5, which provide a total capacity of 128k words.

The System Data segment (64k words defined by Map 1) contains various system values and tables. This space is accessible by all programs, but only if the DS or PRIV bit in the Environment Register is set. SG addressing and the location of system tables is discussed under subsequent headings in the next few pages.

CALLING EXTERNAL PROCEDURES

Procedures in an external code segment can be called and executed as efficiently as a program's own procedures. The XCAL (external procedure call) instruction and the SG-relative addressing mode are two important features that make this possible.

Figure 2-53 illustrates an example of a call from a User Code segment to a procedure in the System Code segment. (The general method applies also to any external calls between any of the four code segments--User Code, User Library Code, System Code, and System Code Extension.) When the application program calls the external procedure, an XCAL instruction is executed. This instruction places a three-word stack marker on the top of the user stack and moves L and S in the same manner as a PCAL instruction (i.e., defines a new local area). However, instead of transferring control directly to a procedure within the segment, control is vectored out of the segment (via its XEP, External Entry Point Table) into another code segment (through that segment's PEP, Procedure Entry Point Table). In this example, the System Code Segment's Procedure Entry Point table (PEP) is used to determine the procedure's starting address, and the CS bit in the ENV Register is set to "1" so that instructions will be executed from the System Code segment. The DS bit, however, remains a "0" so that the user environment (as opposed to the system environment) is still in effect. The local area for the system procedure is therefore in the User Data segment. Specifically, the steps involved when the XCAL instruction is executed are:

1. The caller's environment is stored in a stack marker.

   ```
   data [S+1] := P;
   data [S+2] := ENV;
   data [S+3] := L;
   ```

2. The C[0]-relative address of the procedure being called is obtained by a two-step process. First, the XCAL instruction specifies a location in the caller's External Entry Point Table (XEP; refer back to Figure 2-38). Then, the XEP entry is used to locate the desired code segment (bits 0 through 3 of the entry specify a map number) and Procedure Entry Point address (bits 7 through 15 of the entry specify a PEP number), which in this case is in the System Code segment's Procedure Entry Point Table. This address is put in the P Register so that the next instruction executed will be the first instruction of the system procedure.

3. If the calling procedure is not executing in privileged mode, the callability attribute of the system procedure being called is checked.

   ```
   map := 3;    ! system code map, in this case
   temp := <PEP number>;
   if not PRIV then
     if temp >= mem(3,0) then    ! call to callable
       begin
         if temp >= mem(3,1) then    ! call to privileged
           instruction^failure^trap;
         PRIV := 1; ! set privileged mode
       end;

   P := mem(pepmap,temp)    ! get entry point address into P
   ```

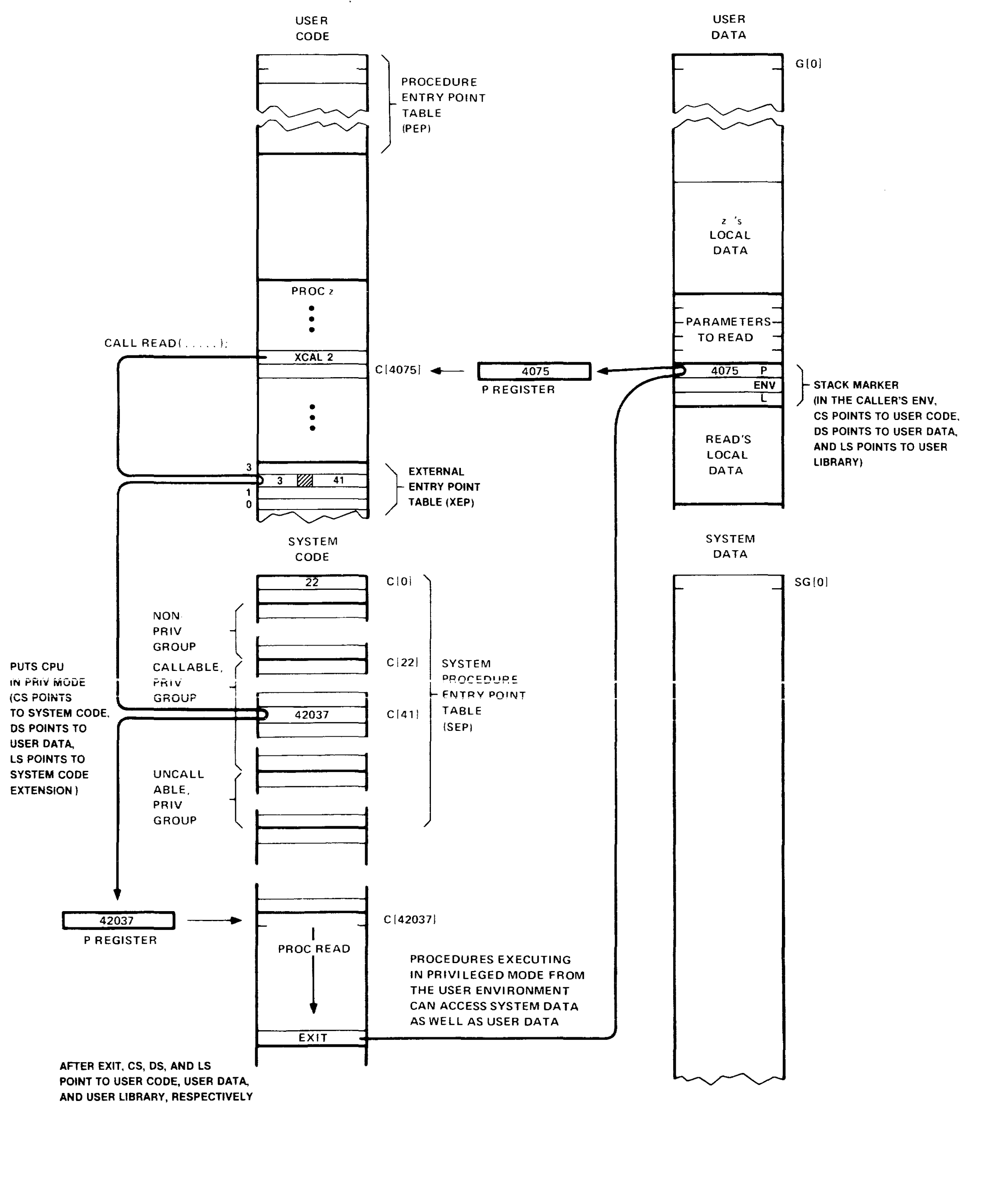

USER CODE
PROCEDURE ENTRY POINT TABLE (PEP)
PROC z
CALL READ(. . . . .);
XCAL 2
C[4075]
4075
P REGISTER
3
3
41
1
0
EXTERNAL ENTRY POINT TABLE (XEP)
USER DATA
G[0]
z 's LOCAL DATA
PARAMETERS TO READ
4075 P
ENV
L
STACK MARKER (IN THE CALLER'S ENV, CS POINTS TO USER CODE, DS POINTS TO USER DATA, AND LS POINTS TO USER LIBRARY)
READ'S LOCAL DATA
SYSTEM CODE
22
C[0]
NON PRIV GROUP
CALLABLE, PRIV GROUP
C[22]
SYSTEM PROCEDURE ENTRY POINT TABLE (SEP)
42037
C[41]
UNCALLABLE, PRIV GROUP
PUTS CPU IN PRIV MODE (CS POINTS TO SYSTEM CODE, DS POINTS TO USER DATA, LS POINTS TO SYSTEM CODE EXTENSION)
SYSTEM DATA
SG[0]
42037
P REGISTER
C[42037]
PROC READ
EXIT
PROCEDURES EXECUTING IN PRIVILEGED MODE FROM THE USER ENVIRONMENT CAN ACCESS SYSTEM DATA AS WELL AS USER DATA
AFTER EXIT, CS, DS, AND LS POINT TO USER CODE, USER DATA, AND USER LIBRARY, RESPECTIVELY

Figure 2-53. System Procedure Call and Exit

4. The S and L Registers are set with the G[0]-relative address of the new top-of-stack location.

   ```
   L := S := S + 3;
   ```

   The new L Register setting defines the base of the local area for the system procedure being called.

5. The new S Register setting is tested for an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the operating system Stack Overflow trap (and the XCAL instruction is aborted).

   ```
   if S > 32767 then stack^overflow^trap;
   ```

6. The CS bit of the ENV Register is set to 1 and the LS bit is set to 0, so that further code area references will be in the System Code segment (in this example). LS and CS are set based on the map number in the XEP Table.

7. Finally, the Register Stack Pointer, RP, is given an initial value of seven (stack empty).

When the system procedure finishes, the usual EXIT instruction is executed. The CS bit is restored from the stack marker so that the next instruction is executed from the User Code segment.

If the system procedure must access the System Data segment from the user environment it is given the attribute "callable" (so that it can be called by the nonprivileged application program) and executes in privileged mode. Executing in privileged mode permits the procedure to make use of the "SG" addressing mode. This addressing mode, illustrated in Figure 2-54, provides access to the System Data segment (and, therefore any system tables) even when DS indicates User Data.

The SG-Relative mode for a memory reference instruction allows direct addressing of the first 64 locations of the operating system's data segment (SG[0:63]). This mode is indicated by bits I.<7:9> of the memory reference instruction equal to 110. Bits I.<10:15> are a positive word displacement from SG[0]:

```
direct^address = I.<10:15>
```

The data map used for the SG-relative addressing mode is determined by the function:

```
datamap:
  if I.<7:9> = 6 and PRIV then 1      ! system data map.
                          else DS;    ! current data map.
```

Indirect addressing and indexing are both permitted with the SG-relative addressing mode.

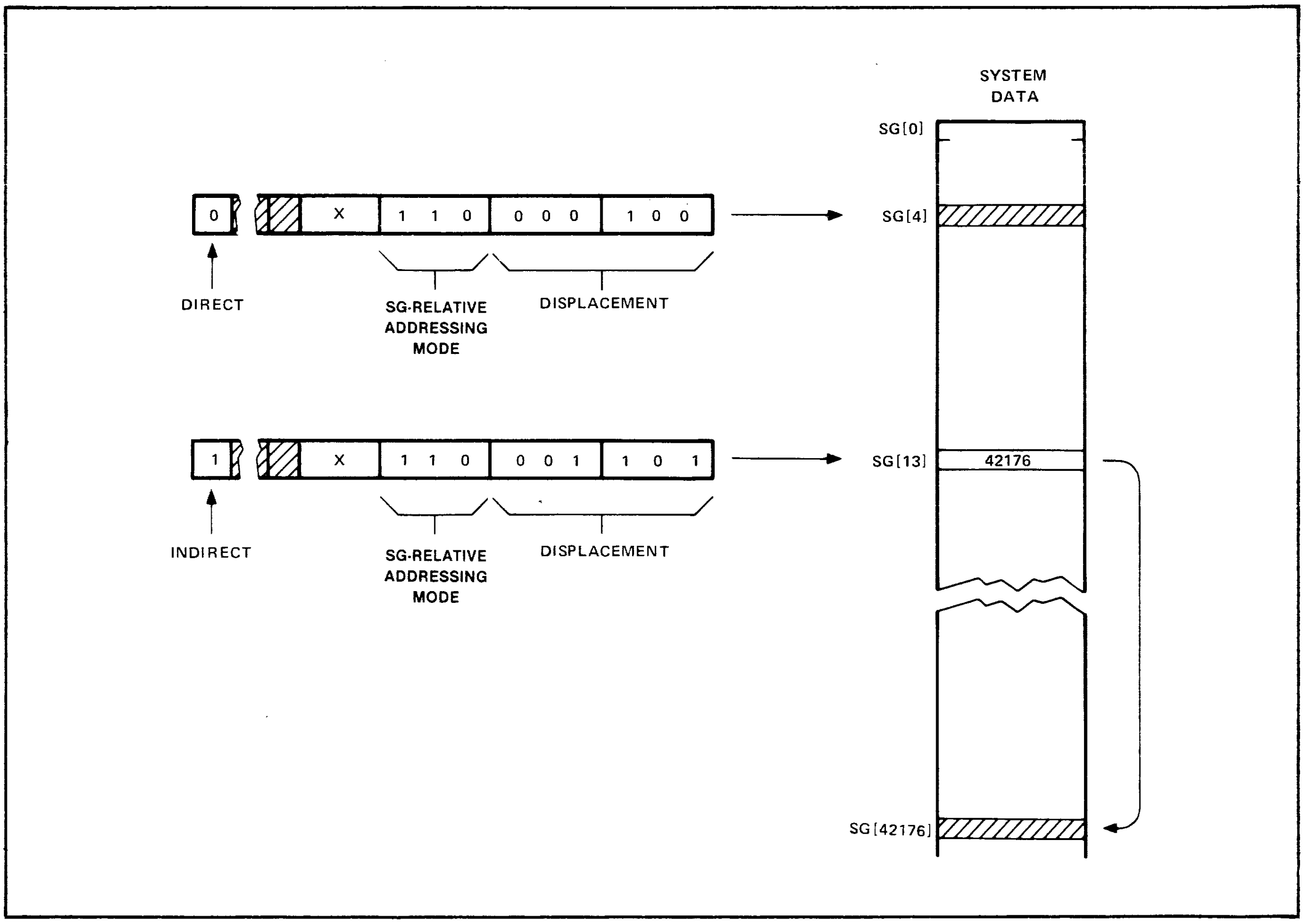


Figure 2-54. SG-Relative Addressing Mode

Executing in privileged mode while in the user environment also means that data can be moved, compared, and scanned (with the MOVW, MOVB, COMW, COMB, SBW, and SBU instructions) between the User Data segment and the System Data segment. (The File System uses a MOVW instruction to transfer data between the User Data segment and the System Data segment.)

SYSTEM TABLES

Some processor-known data assignments within the first two pages of the System Data segment are listed in Table 2-2. Note that all of page 1 is assigned to use for the I/O Control Table. Both pages 0 and 1 of this segment are always located in pages 0 and 1 of physical memory.

The locations of the major tables discussed at length later in this section are illustrated in Figure 2-55, and briefly described in the following paragraphs.

SYSTEM INTERRUPT VECTOR. SG[%1200:%1337] is the System Interrupt Vector (SIV). This table contains 24 four-word entries; each entry defines the executing environment for one of the operating system interrupt handlers (see "Interrupt System").

BUS RECEIVE TABLE. SG[%1400:%1477] is the Bus Receive Table (BRT). This table contains 16 four-word entries, each of which is assigned to manage the interprocessor transfers for one processor module. Each entry describes the number of words expected and the system buffer location where the data is to be stored (see "Interprocessor Buses").

I/O CONTROL TABLE. SG[%2000:%3777] is the I/O Control Table (IOC). This table contains 256 entries corresponding to the 256 subchannels that can be connected to an i/o channel. Each entry describes the number of bytes to be transferred and the system buffer location where the data transfer takes place (see "Input/Output Channel").

Table 2-2. System Data Segment Table Values

Location	Contents
%2	Dummy Priority Value
%3	Current Process Control Block Pointer
%4:%77	Software Values
%100:%101	Ready List Header
%102	Dummy Priority Value
%103:%106	Microsecond Counter
%107:%110	Time List Header
%111:%114	OSP I/O Control Block
%115:%116	Memory Breakpoint Trap Address
%117	Trace Buffer Base
%120	Trace Buffer Limit
%121	Trace Buffer Pointer
%122	LIGHTS Save Area
%123	Breakpoint Table Base
%124	Breakpoint Table Entry Size
%125	Breakpoint Table Limit
%1153:%1177	Processor Dump Save Area
%1200:%1337	System Interrupt Vector
%1340:%1357	Currently Mapped Segment Table
%1360:%1377	Interprocessor Bus Error Packet
%1400:%1477	Bus Receive Table
%2000:%3777	Input/Output Control Table

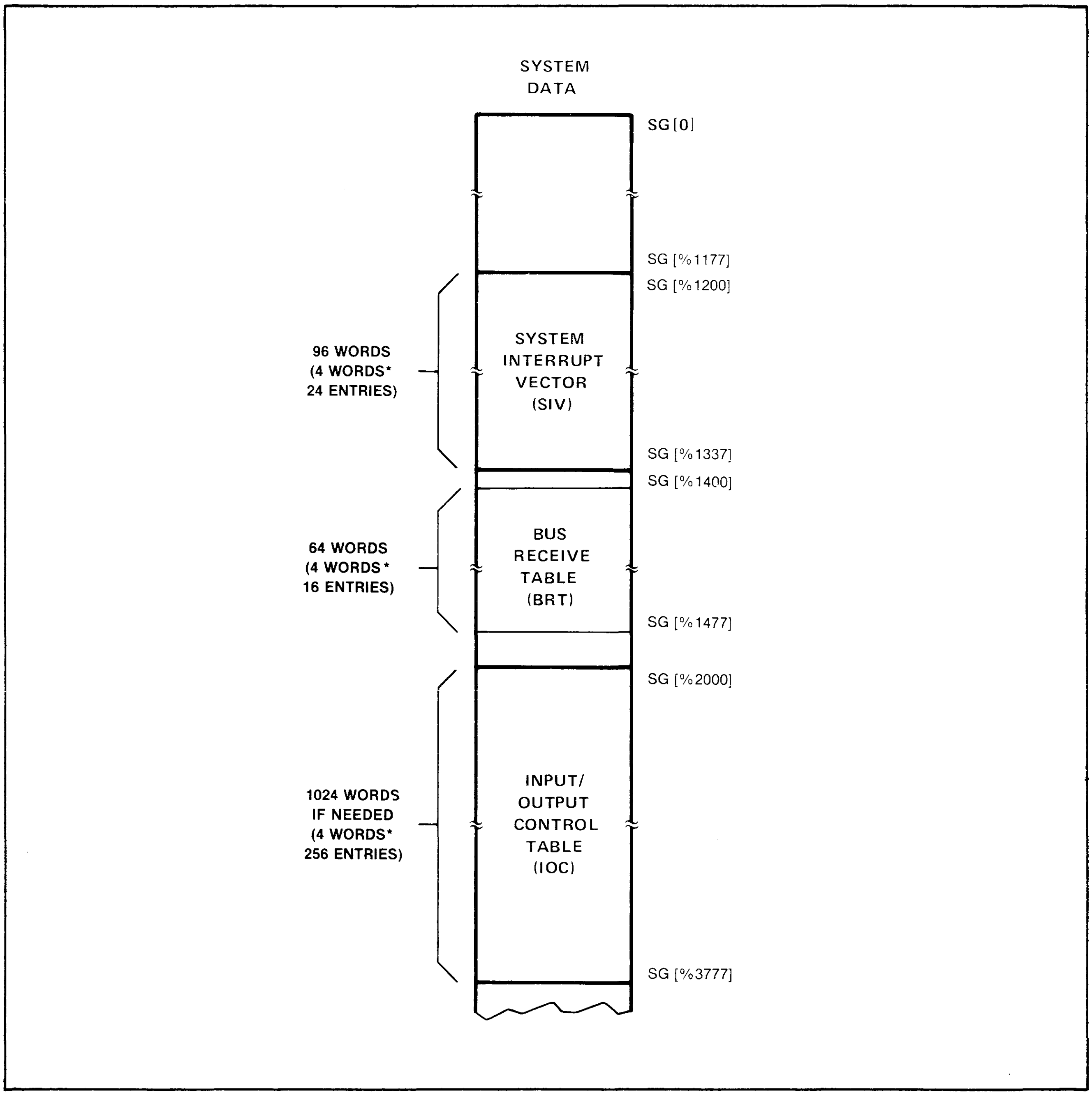

SYSTEM DATA
SG [0]
SG [%1177]
SG [%1200]
96 WORDS (4 WORDS* 24 ENTRIES)
SYSTEM INTERRUPT VECTOR (SIV)
SG [%1337]
SG [%1400]
64 WORDS (4 WORDS* 16 ENTRIES)
BUS RECEIVE TABLE (BRT)
SG [%1477]
SG [%2000]
1024 WORDS IF NEEDED (4 WORDS* 256 ENTRIES)
INPUT/ OUTPUT CONTROL TABLE (IOC)
SG [%3777]

Figure 2-55. Dedicated Memory Locations in System Data

INTERRUPT SYSTEM

The interrupt system transfers control to a specific location in the operating system (called an interrupt handler) upon the occurrence of any of the conditions listed in Table 2-3.

Table 2-3. Interrupt Conditions

Interrupt No.	Event
0	Special channel error
1	Uncorrectable memory error
2	Memory access breakpoint
3	Instruction failure
4	Page fault
5	Undefined
6	Undefined
7	OSP (Operations & Service Processor) i/o
8	Power fail
9	Correctable memory error
10	High-priority i/o
11	Interprocessor bus receive completion
12	Undefined
13	Time list
14	Standard i/o
15	Dispatcher
16	Power on
17	Stack overflow
18	Arithmetic overflow or divide by zero
19	Instruction breakpoint
20-23	Undefined

Generally, when an interrupt occurs the interrupted environment is saved in an interrupt stack marker. An operating system interrupt handler executes to process the particular interrupt. Then an IXIT (interrupt exit) instruction is executed to restore the interrupted environment. See Figure 2-56.

INT and Mask Registers

Three registers are associated with interrupts: two 16-bit interrupt registers and a 16-bit Mask Register. The bit assignments of these registers are illustrated in Figure 2-57. Only four bits of INTB are relevant to interrupts; however, these four are the highest-priority interrupt bits, being examined first at the conclusion of each instruction. The interrupts represented by the bits of INTA are

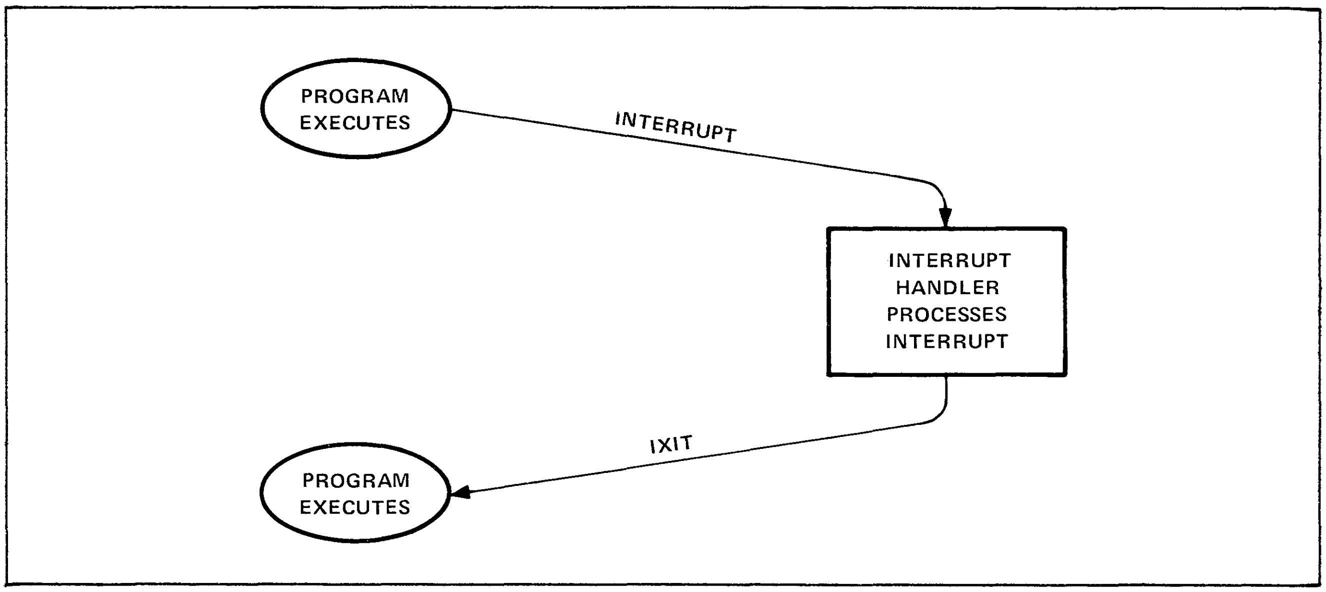


Figure 2-56. General Interrupt Sequence

"maskable." That is, the corresponding bits of the Mask Register are used by the operating system to allow or disallow particular interrupt types at various critical or noncritical times. Bit 6 of INTA (arithmetic overflow or divide by zero) is separately masked by the Trap Enable bit of the Environment Register (ENV.<8>), but is used in a similar way to enable or disable that interrupt. For all maskable interrupts, the interrupt condition is ignored if the corresponding Mask bit is a "0", and will continue to be deferred until the Mask bit is set to "1". The checking operation is performed by a logical AND of the two registers.

Most interrupt types can occur only at the end of an instruction, when the hardware routinely checks for the presence of "1" bits in the interrupt registers. However, three interrupt types (power on, uncorrectable memory error, and page fault) are "preemptive"; that is, they will interrupt during an executing instruction. Also, certain long-running instructions (e.g., the Move instructions) may be interrupted during execution.

If two or more interrupt conditions exist simultaneously in INTA, and each has its corresponding Mask Register bit set, the interrupt type with the highest priority (lowest bit number) takes precedence; the others are deferred until the interrupt handler finishes executing and executes an IXIT instruction.

Interrupts for stack overflow, instruction failure, and instruction breakpoint have entries neither in the interrupt registers nor in the Mask Register; these cause an interrupt whenever they occur, ignoring

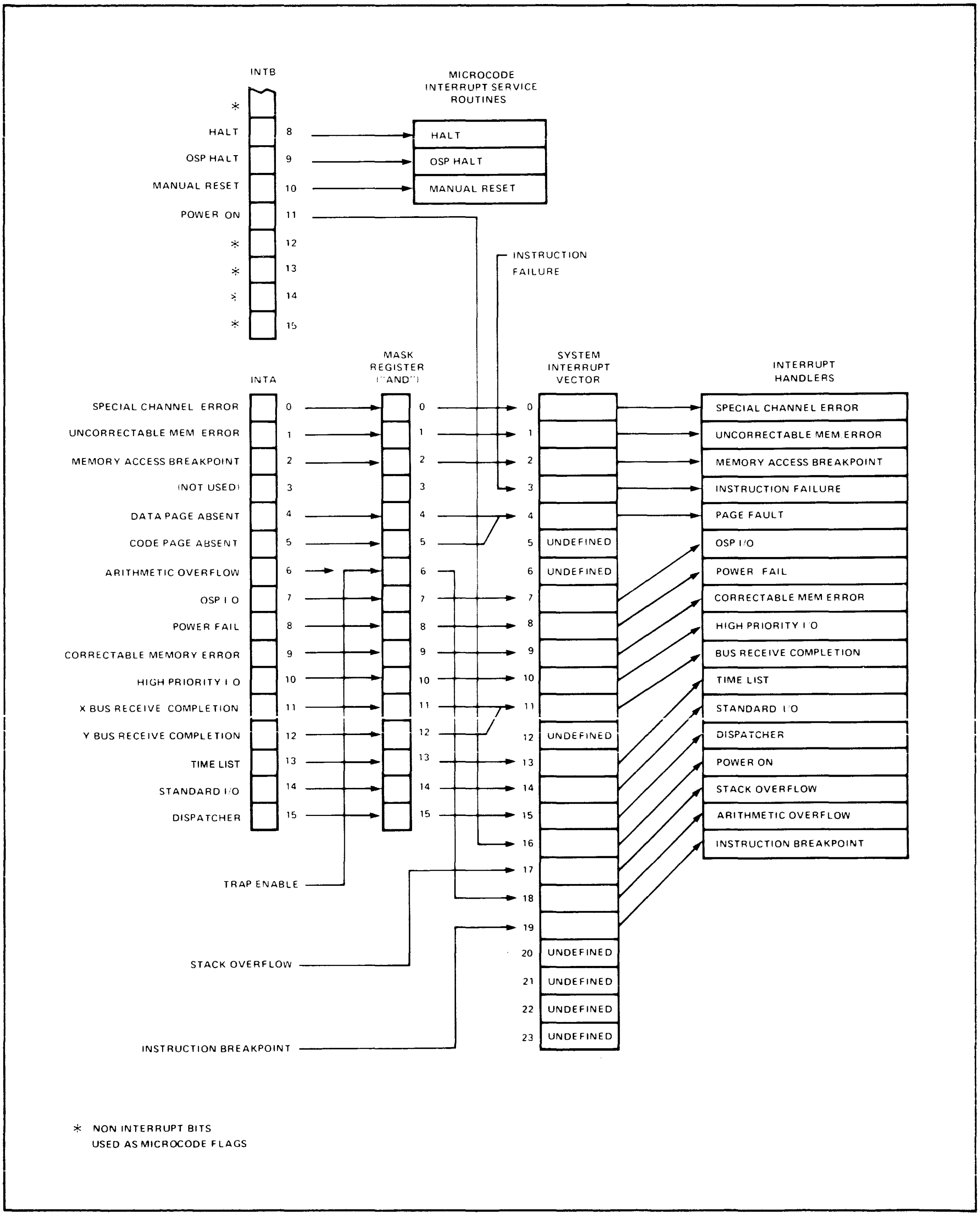


Figure 2-57. INT and MASK Registers

priority. The hardware-only interrupts (halt, OSP halt, and manual reset) are serviced entirely within microcode.

As shown in the diagram (Figure 2-57), detected interrupt conditions are passed to software interrupt handlers through the System Interrupt Vector, which is discussed next.

System Interrupt Vector

Each interrupt event that is to be serviced by software has a corresponding entry in the System Interrupt Vector (SIV). The SIV, which is initialized by the operating system, defines the executing environment for each of the 17 operating system interrupt handlers. The SIV, shown in Figure 2-58, begins at system data location %1200 and contains 24 four-word entries (seven are undefined).

Each four-word entry in the System Interrupt Vector contains the following information:

Li	=	L Register setting for interrupt handler
Mi	=	MASK Register setting for interrupt handler
Pi	=	P Register setting of first instruction in interrupt handler
Vi	=	Interrupt-related parameter put here by firmware

The following paragraphs further describe the functions of each of these entries, as illustrated in Figure 2-59.

- Li: This is the address in the system data area for an interrupt handler's local storage (stack).
- Mi: This is a mask value for masking off unwanted interrupts while an interrupt handler executes. The MASKi value in the SIV entry is ANDed with the current MASK register setting to derive a new setting. This permits nesting of interrupts of different types.
- Pi: This is the system code address of the interrupt handler's entry point.
- Vi: This is a location where an interrupt-related parameter may be returned by firmware.

Interrupt Stack Marker

When an interrupt occurs, the interrupted environment is saved in an interrupt stack marker. The interrupt stack marker is placed at Li[-4:0] in the interrupt handler's stack; see Figure 2-59. The interrupt stack marker contains the following information:

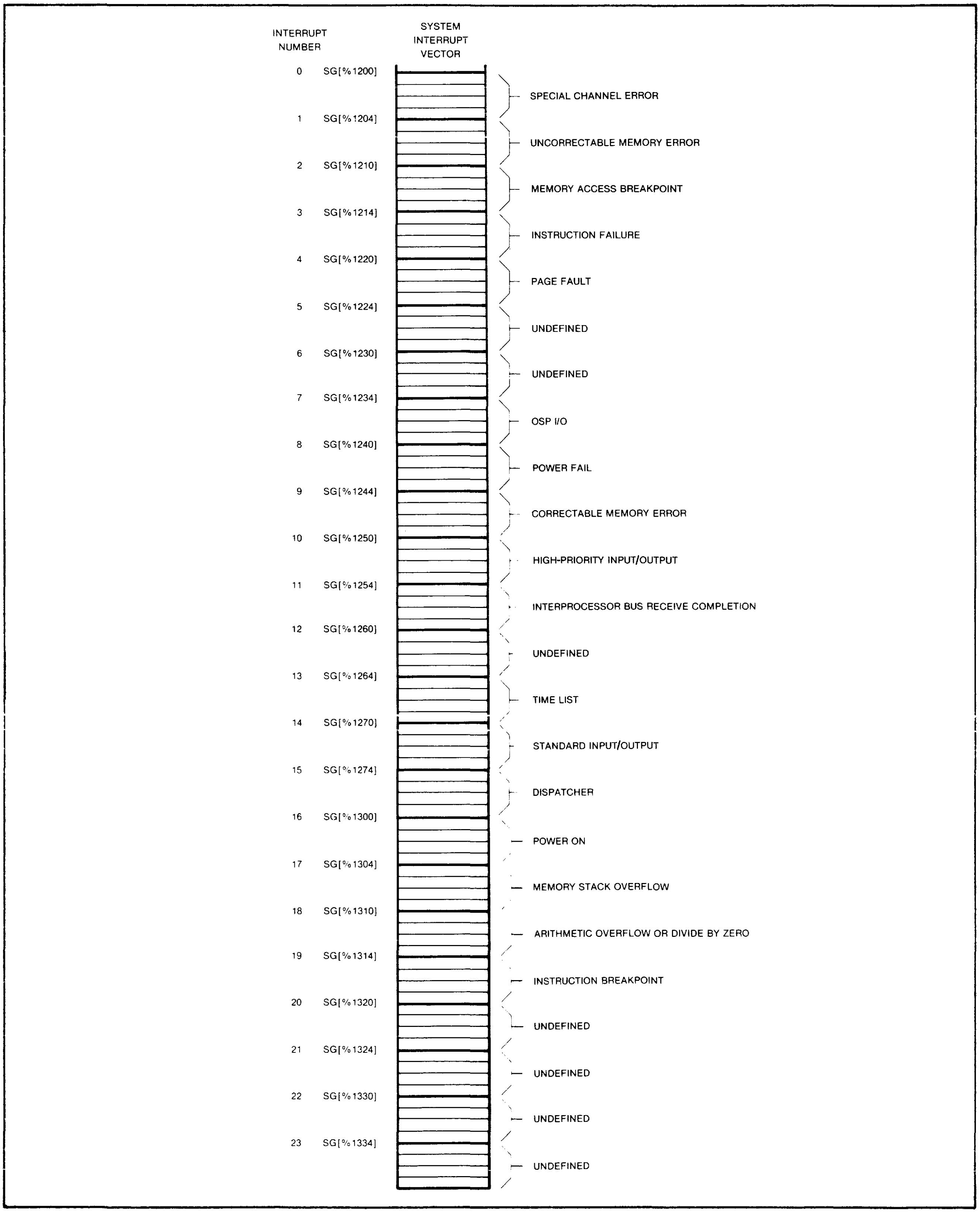


Figure 2-58. System Interrupt Vector

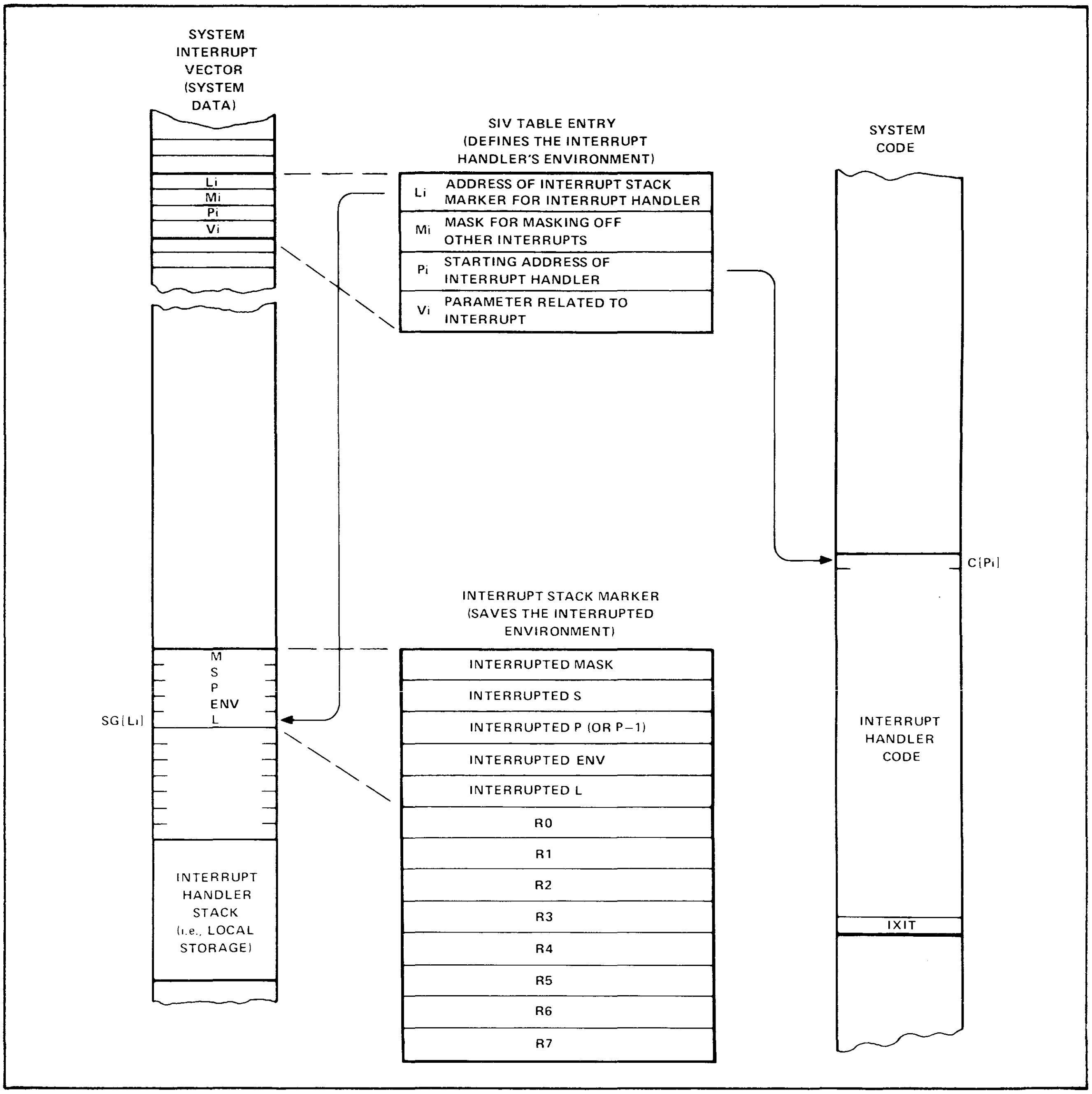

SYSTEM INTERRUPT VECTOR (SYSTEM DATA)
Li
Mi
Pi
Vi
SIV TABLE ENTRY (DEFINES THE INTERRUPT HANDLER'S ENVIRONMENT)
Li ADDRESS OF INTERRUPT STACK MARKER FOR INTERRUPT HANDLER
Mi MASK FOR MASKING OFF OTHER INTERRUPTS
Pi STARTING ADDRESS OF INTERRUPT HANDLER
Vi PARAMETER RELATED TO INTERRUPT
SYSTEM CODE
C(Pi)
INTERRUPT STACK MARKER (SAVES THE INTERRUPTED ENVIRONMENT)
M
S
P
ENV
L
SG(Li)
INTERRUPTED MASK
INTERRUPTED S
INTERRUPTED P (OR P-1)
INTERRUPTED ENV
INTERRUPTED L
R0
R1
R2
R3
R4
R5
R6
R7
INTERRUPT HANDLER STACK (i.e., LOCAL STORAGE)
INTERRUPT HANDLER CODE
IXIT

Figure 2-59. SIV Entry and Interrupt Stack Marker

```
Li[-4] = M, the MASK Register setting at the time of the interrupt
Li[-3] = S, the S Register setting at the time of the interrupt
Li[-2] = P, the P Register setting at the time of the interrupt
Li[-1] = ENV, the ENV Register setting at the time of the interrupt
Li[0]  = L, the L Register setting at the time of the interrupt
```

In addition, each time an interrupt occurs the current contents of the Register Stack (R0 through R7) are saved in the first eight locations of local storage (i.e., SG[Li+1] through SG[Li+8]).

Interrupt Sequence

An interrupt (i is the interrupt number) is defined as:

```
if INTA.<i> land MASK.<i> then    ! an interrupt occurred
  begin
    Vi := interrupt parameter;    ! if any
    sysdata[Li-4] := MASK;        !
    sysdata[Li-3] := S;           !
    sysdata[Li-2] := P;           ! interrupt stack marker
    sysdata[Li-1] := ENV;         !
    sysdata[Li]   := L;           !
    sysdata[Li+1] := R0           !
       thru                       ! saved Register Stack
    sysdata[Li+8] := R7           !
    ENV  := %3447;                ! PRIV, DS, CS, V, RP = 7
    L    := Li;
    S    := L + 8;
    P    := Pi;
    MASK := MASK LAND Mi;
  end;
```

An example is discussed in the following paragraphs, with reference to Figures 2-60 and 2-61. (The first 10 steps are shown in Figure 2-60.)

1. An interrupt condition occurs (in this example, a device is requesting standard i/o servicing).

   ```
   INTA.<14> := 1;
   ```

2. The current instruction completes executing and, since MASK.<14> is a "1", an interrupt occurs.

   ```
   if INTA land MASK then     ! interrupt.
     begin
   ```

3. There is no interrupt parameter for a standard i/o interrupt.

4. The interrupted environment (including the current MASK and S Register settings) is saved in the area pointed to by Li in the SIV entry for the standard i/o interrupt.

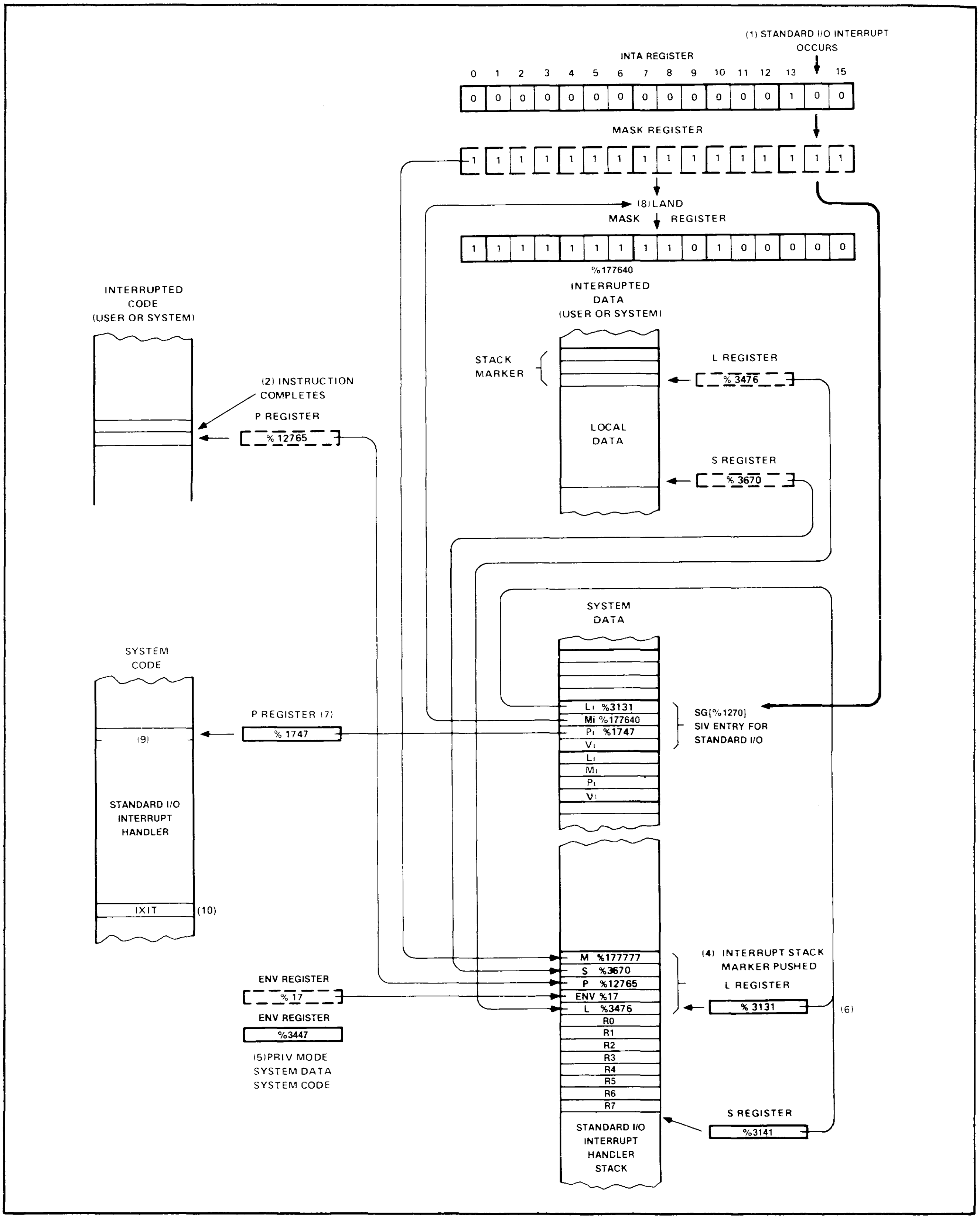


Figure 2-60. Interrupt Sequence

```
          sysdata[Li-4] := MASK; !
          sysdata[Li-3] := S;    !
          sysdata[Li-2] := P;    ! interrupt stack marker
          sysdata[Li-1] := ENV;  !
          sysdata[Li]   := L;    !
          sysdata[Li+1] := R0    !
            thru                 ! saved Register Stack
          sysdata[Li+8] := R7    !
```

5. The PRIV (privileged mode), DS (data space), and CS (code space) bits in the ENV Register are set. This defines the interrupt handler executing environment.

   ```
   ENV := %3447;
   ```

6. The L and S Registers are set with the address of the interrupt handler's local data area. This is the value Li in the SIV entry for the standard i/o interrupt.

   ```
   L := Li;
   S := L + 8;
   ```

7. The P Register is set with the address of the first instruction in the Standard I/O Interrupt Handler. This is the value Pi in the SIV entry for standard i/o.

   ```
   P := Pi;
   ```

8. The Mi value in the SIV entry is ANDed with the current MASK Register setting to derive a new MASK Register setting.

   ```
   MASK := MASK land Mi;
   ```

9. The first instruction of the Standard I/O Interrupt Handler executes.

   ```
   I := code[P];
   ```

10. The interrupt handler runs to completion, unless the interrupt handler's mask allows interrupts or purposely unmasks any or all interrupts and corresponding interrupts do occur. Finally, an IXIT instruction is executed to return to the interrupted process.

11. The IXIT instruction (see Figure 2-61) restores the interrupted environment saved in the interrupt stack marker (at L[-4:0]); that is, the MASK, S, P, ENV, and L Registers are returned to their pre-interrupt values.

    ```
    MASK := sysdata [L-4];     !  (a)
    S    := sysdata [L-3];     !  (b)
    P    := sysdata [L-2];     !  (c)
    ENV  := sysdata [L-1];     !  (d)
    L    := sysdata [L];       !  (e)
    ```

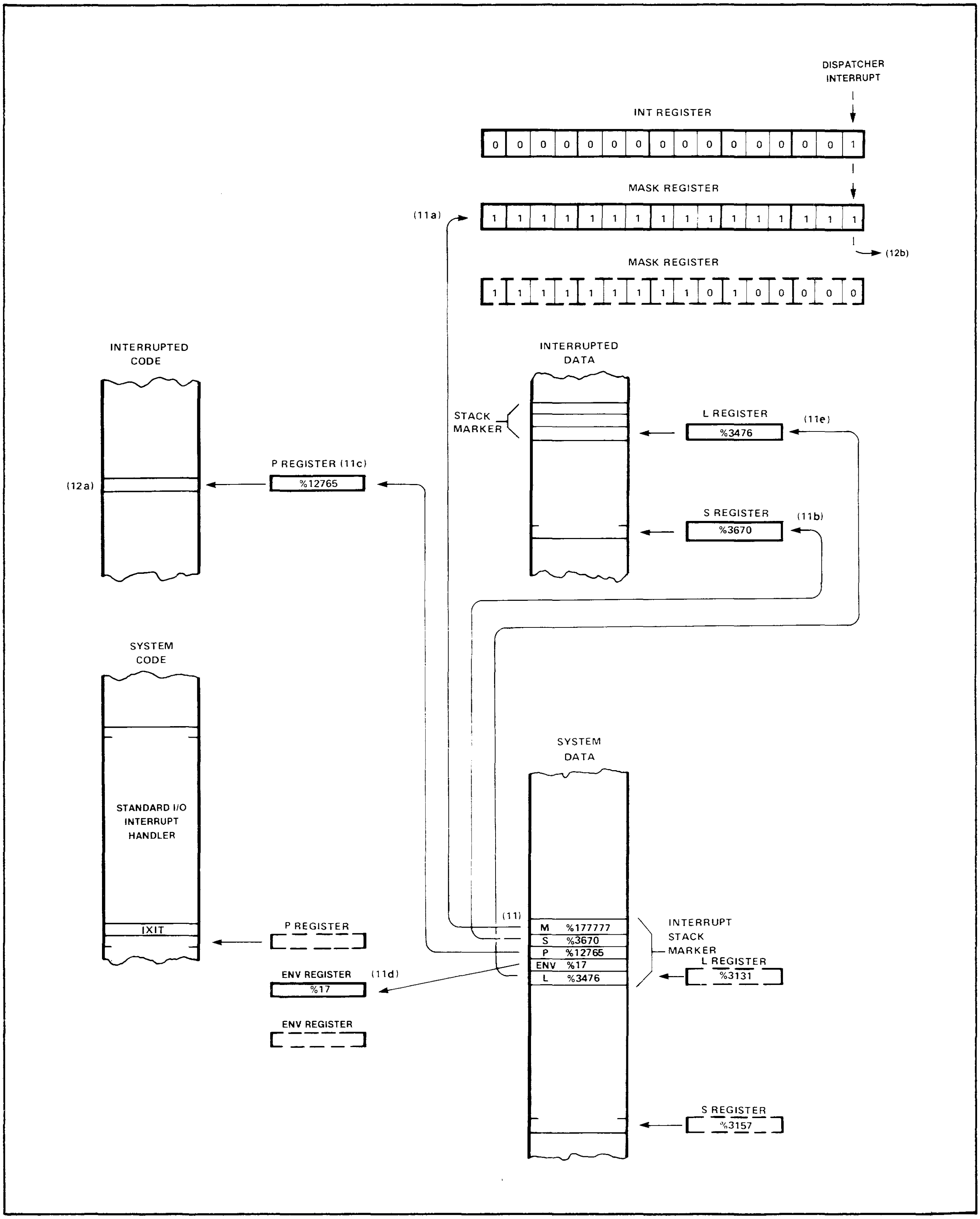


Figure 2-61. IXIT Sequence

Also the Register Stack (values saved in L+1 through L+8) is returned to its pre-interrupt condition.

12a. If no interrupt is pending when the IXIT instruction completes, process execution resumes at the point of interruption.

12b. If another interrupt is pending, the interrupt sequence is repeated from step 1, using the appropriate SIV entry to set up the interrupt handler's environment.

Interrupt Types

The following paragraphs describe each of the interrupt types.

SPECIAL CHANNEL ERROR (0). This interrupt occurs when the i/o channel detects types of errors that require software servicing. The error number is placed in the parameter word. Certain errors have a second error word giving the subchannel address and command, which is found in R7 on entry to the interrupt handler.

UNCORRECTABLE MEMORY ERROR (1). This interrupt occurs when a memory word is accessed by the IPU and contains an error which cannot be corrected. The parameter contains the logical address of the page at fault and the six syndrome bits generated by the error correction circuitry. These syndrome bits provide information for Tandem service personnel. The format of the parameter word is:

Vl.<0:5>	=	logical page
Vl.<6:11>	=	syndrome
Vl.<12:15>	=	map number

The contents of the data word that was in error is found in R7 on entry to the interrupt handler.

MEMORY ACCESS BREAKPOINT (2). This interrupt occurs when the memory breakpoint has been armed by the SMBP instruction and the breakpoint memory address has been accessed in the desired manner. There is no parameter. No interrupt occurs if the breakpoint was armed by the Operations and Service Processor (OSP); instead, the processor performs a system freeze and enters the idle loop.

INSTRUCTION FAILURE (3). This interrupt occurs when an unimplemented instruction is executed, or when execution of a privileged instruction is attempted by a program which is not in privileged mode, or when an abnormal condition is detected during the execution of certain instructions. The parameter for this trap is the current instruction.

PAGE FAULT (4). This interrupt occurs when an attempt is made to access an absent memory page (i.e., its map entry "absent" bit is set to 1). The parameter word is:

V4.<0:5>	=	logical page
V4.<12:15>	=	map number

OSP I/O COMPLETION (7). The i/o completion interrupt for the Operations and Service Processor occurs when either a read or a write operation to the OSP completes. The parameter word indicates the status, as follows:

0	normal read completion
1	normal write completion
%177777	character overrun detected on a read
%177776	write interrupt with negative byte count
%177775	read interrupt with zero or negative byte count

POWER FAIL (8). This interrupt occurs when a processor module power failure is detected. A minimum of five milliseconds is available for processing after this interrupt occurs before power is lost. There is no parameter.

CORRECTABLE MEMORY ERROR (9). This interrupt occurs when a memory error occurred and can be corrected. The parameter word is of the same form as that for an uncorrectable memory error.

HIGH-PRIORITY I/O COMPLETION (10). This interrupt occurs when a device that is connected to the high-priority interrupt poll line requires servicing. There is no parameter.

INTERPROCESSOR BUS RECEIVE COMPLETION (11). This interrupt occurs when a transmission is received on either the X-bus or the Y-bus. The parameter word is of the following form:

```
V11.<0>      = bus
                 0  received on X-bus
                 1  received on Y-bus
```

V11.<1:7> = status

0	normal completion
1	unexpected packet
2	checksum error
3	misrouted packet
4	"unsequenced" packet
5	sequence error
6	illegal extended buffer address

V11.<8:15> = processor number of sender

In addition, R7 contains the checksum+1 computed by the microcode when a checksum error is detected.

TIME LIST (13). Every 10 milliseconds the microcode detects an interval clock micro-interrupt and decrements the wait time of the element at the head of the Time List. If it has gone to zero, control passes to the Time List Interrupt Handler; otherwise, no action is taken. There is no parameter.

STANDARD I/O COMPLETION (14). This interrupt occurs when a device that is connected to the standard interrupt poll line requires servicing. There is no parameter.

DISPATCHER (15). This interrupt occurs when a DISP or SNDQ instruction is executed, or when a PSEM or VSEM instruction is executed that requires operating system aid. Bit 15 of the parameter word is set on a DISP, bit 14 is set on a SNDQ, bits 13 and 15 are set on a PSEM when the semaphore cannot be obtained, and bit 12 is set when a VSEM instruction must release a blocked process. No part of the parameter word is ever cleared by the processor. If a Dispatcher interrupt is pending but the contents of the parameter word are zero, the interrupt is cleared.

POWER ON (16). This interrupt occurs when power is applied following a power failure when memory is in a valid state and the maps have been successfully loaded with no uncorrectable memory errors. The contents of Loadable Control Store are invalid. There is no parameter for this interrupt.

STACK OVERFLOW (17). This interrupt occurs when S exceeds 32,767 (i.e., the limit of the memory stack) following the execution of any instruction which can change the S Register setting-- SETS, PCAL, XCAL, ADDS, BSUB, or PUSH. There is no parameter.

ARITHMETIC OVERFLOW (18). This interrupt occurs when the T (trap enable) and V (arithmetic overflow) bits in the ENV Register are simultaneously set to 1. There is no parameter.

INSTRUCTION BREAKPOINT (19). This interrupt occurs when a BPT instruction is executed, or when an EXIT or DXIT instruction is executed with ENV.<1> set to 1 in the stack marker. The parameter is the instruction which caused the interrupt.

INTERPROCESSOR BUSES

A NonStop II computer system has two interprocessor buses, designated the X-bus and the Y-bus. Each processor module in the system is connected to both buses and is capable of communicating with any processor module (including itself) over either bus. See Figure 2-62.

With any given interprocessor bus transfer, one processor module is the source (and initiator), the other is the destination (and receiver). Before a processor module can receive data over an

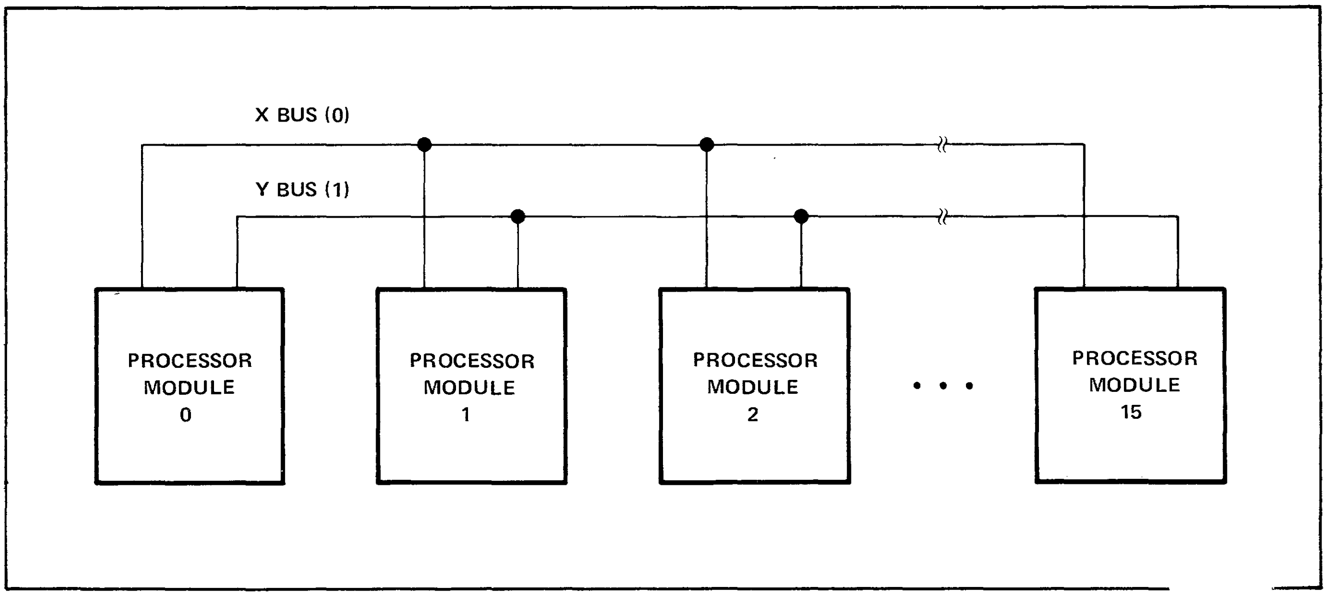


Figure 2-62. Processor Module Addressing

interprocessor bus, the operating system first configures an entry in a table known as the Bus Receive Table (BRT). Each BRT entry contains, among other things, the address where the incoming data is to be stored and the number of bytes expected.

To transfer data over a bus (see Figure 2-63), a SEND instruction is executed in the source processor module. The SEND instruction specifies the bus to be used for the transfer, the destination processor module, the number of bytes to be sent, the source location in memory of the data to be sent, the sender's processor number, a timeout value, and a sequence number. While the source processor module is executing the SEND instruction and sending data over the bus, the firmware in the destination processor module is storing the data away according to the appropriate BRT entry (this occurs concurrently with program execution). When the destination processor module receives the expected number of bytes (the bus transfer is complete), a Bus Receive interrupt is posted.

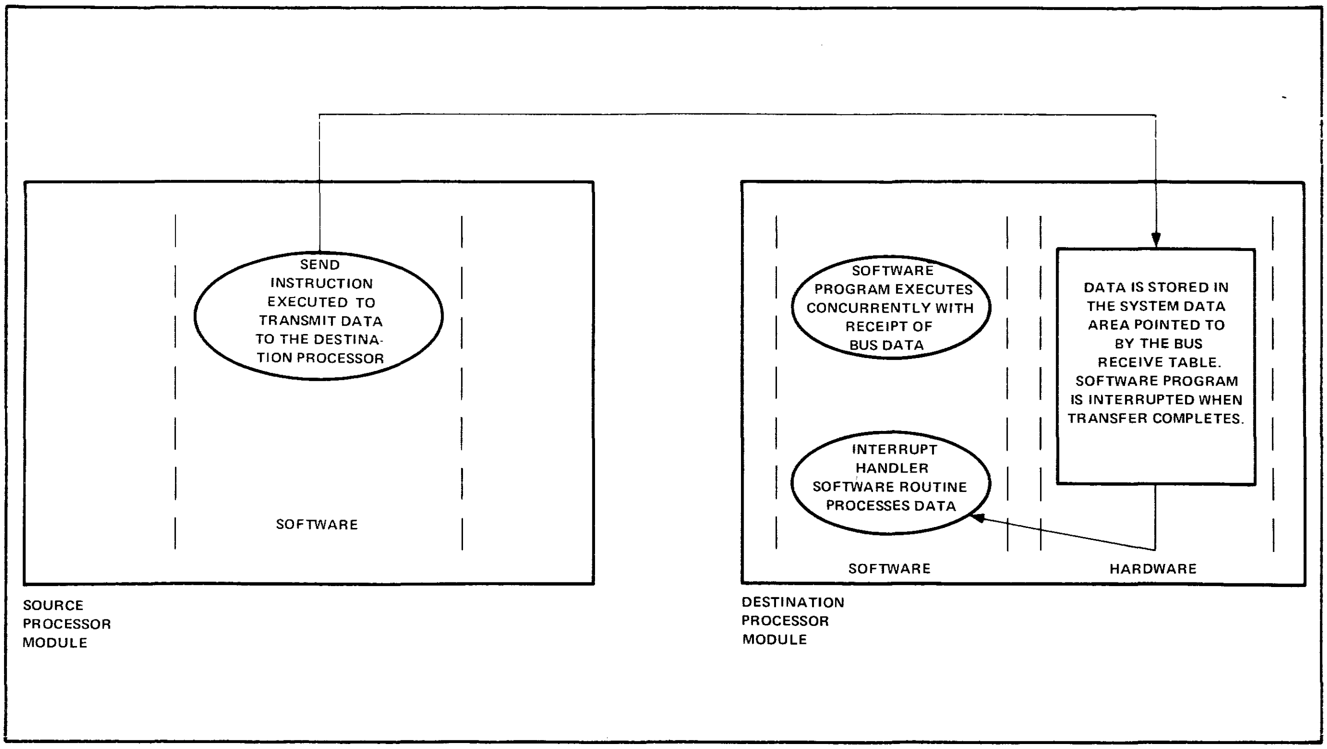

SEND INSTRUCTION EXECUTED TO TRANSMIT DATA TO THE DESTINATION PROCESSOR
SOFTWARE
SOURCE PROCESSOR MODULE
SOFTWARE PROGRAM EXECUTES CONCURRENTLY WITH RECEIPT OF BUS DATA
DATA IS STORED IN THE SYSTEM DATA AREA POINTED TO BY THE BUS RECEIVE TABLE. SOFTWARE PROGRAM IS INTERRUPTED WHEN TRANSFER COMPLETES.
INTERRUPT HANDLER SOFTWARE ROUTINE PROCESSES DATA
SOFTWARE
HARDWARE
DESTINATION PROCESSOR MODULE

Figure 2-63. Simplified Bus Transfer Sequence

Bus Receive Table

The Bus Receive Table (BRT) contains 16 four-word entries, which correspond to the 16 processor modules possible in a system. The table begins at location SG[%1400].

Each entry in the BRT (see format in Figure 2-64) contains the address in the virtual memory where the incoming data is to be stored, a count of the number of bytes expected, and the expected sequence number. (Refer to the "Memory Access" discussion for a description of virtual memory addressing using absolute extended addresses.)

If a processor is to receive data over a designated bus, the corresponding bit in the interrupt Mask Register must be a "1". These mask bits, when on, enable both the receipt of data and the interrupt itself. The bits are:

X-Bus Receive Enable = MASK.<11>
Y-Bus Receive Enable = MASK.<12>

SEND Instruction

The SEND instruction expects seven parameter words in the Register Stack. These are shown in Figure 2-64, and are described as follows.

- G.<15> specifies the bus (0 = X bus, 1 = Y bus) to be used.
- F.<0:15> is the sequence number to be sent.
- E.<0:7> specifies the sender processor module, and E.<8:15> specifies the receiving processor module.
- D.<0:15> is a value that is subtracted from 32,768 to derive the number of 0.8-microsecond units allotted to completing a single packet (16-word) transfer. The timeout period is restarted for each packet transferred. (This parameter is normally zero when the operating system issues a SEND.)
- C.<0:15> and B.<0:15> form the absolute extended (byte) address of the buffer containing the data to be transferred.
- A.<0:15> is an unsigned count of the number of data bytes to be transferred.

Following execution of the SEND instruction, the condition code is set to either of two values:

CCL = Packet Timeout
CCE = Successful

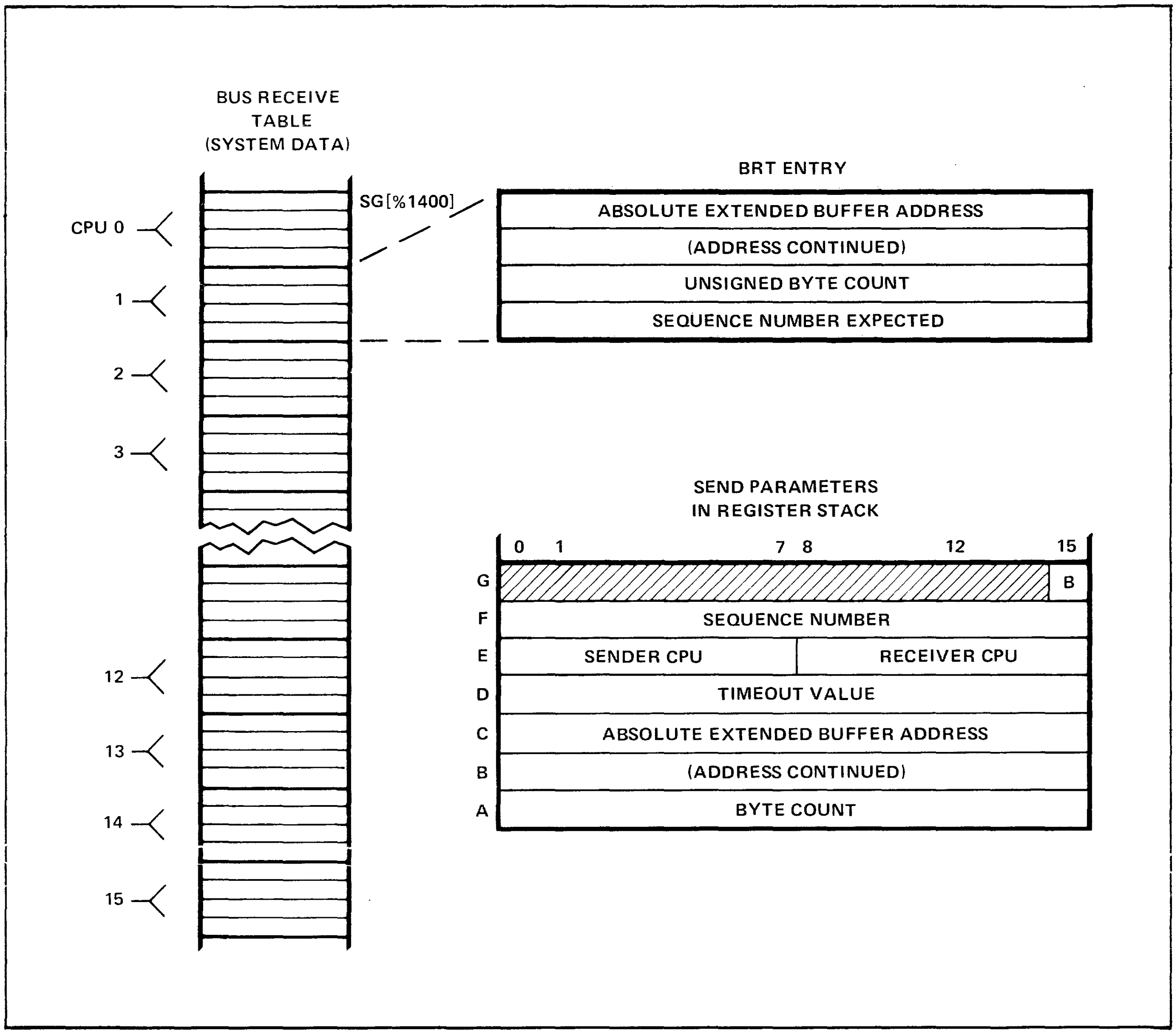


Figure 2-64. Formats Associated with Bus Transfers

Specifically, the SEND instruction executes as follows:

1. The hardware checks whether the OUTQ is empty, since it must be empty when the send begins. If the OUTQ is not empty, the hardware checks for interrupts and services any that are pending. Then it checks for a timer overflow. If the timer did not overflow, it updates the timer and begins step 1 again. If a timer overflow occurred, indicating that the OUTQ did not become empty within the timeout period, a packet timeout occurs and the SEND is aborted. Timeout is defined as:

   0.8(32768 - D) microseconds

2. If data remains to be sent (i.e., count <> 0), it is placed in the OUTQ (bytes 4 through 29, or OUTQ[2:14]). If there are fewer than 26 bytes to be transferred, OUTQ[2:14] is padded with zeros. The sequence number is placed in OUTQ[1] and the routing word in OUTQ[0]; an odd parity checksum is calculated and placed in OUTQ[15]. The packet is then sent, and the transfer address and count parameters are updated. The transfer address is an absolute extended address, and the count is an unsigned byte count.

3. If no data remains to be sent, the SEND is flagged internally as "done" and the condition code is set to CCE to indicate a successful completion.

4. If a packet timeout occurs, the operation is also flagged internally as "done". However, the condition code is set to CCL to indicate a packet timeout.

5. The sequence repeats back to step 2.

Bus Transfer Sequence

As previously stated, there must be coordination between the source processor module and the destination module in regard to the number of bytes to be transferred. The operating system accomplishes this by preceding each transfer with a separate transfer (i.e., SEND) of a predetermined number of bytes of control information. In general, this control information tells the operating system in the destination module to expect a specified number of bytes over a specified bus. In the following example, illustrated in Figures 2-65a and b, assume that the initial transfer has taken place. The operating system in the destination module has configured the appropriate BRT entry for receiving 400 bytes.

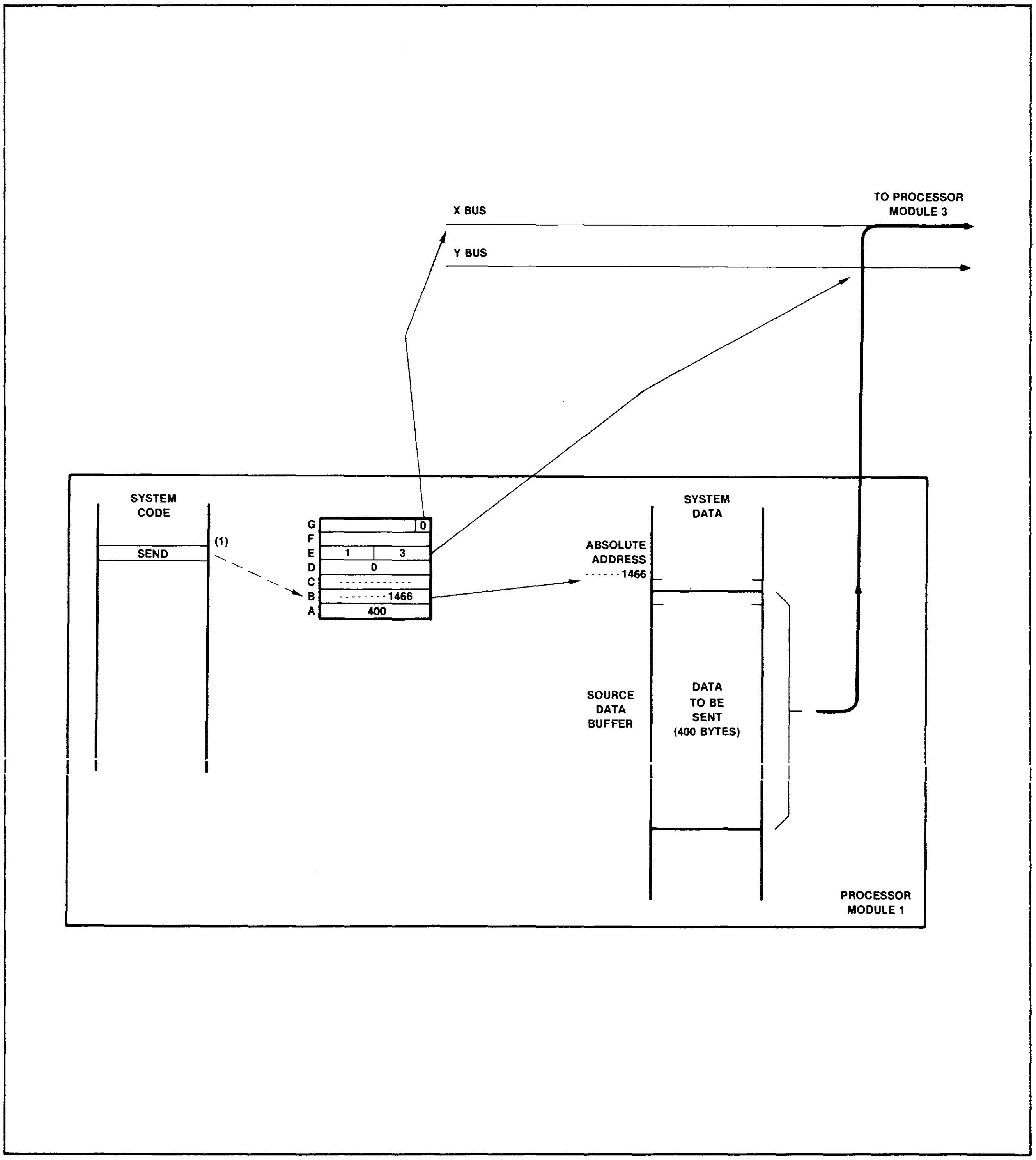


Figure 2-65a. Bus Transfer Sequence (Send)

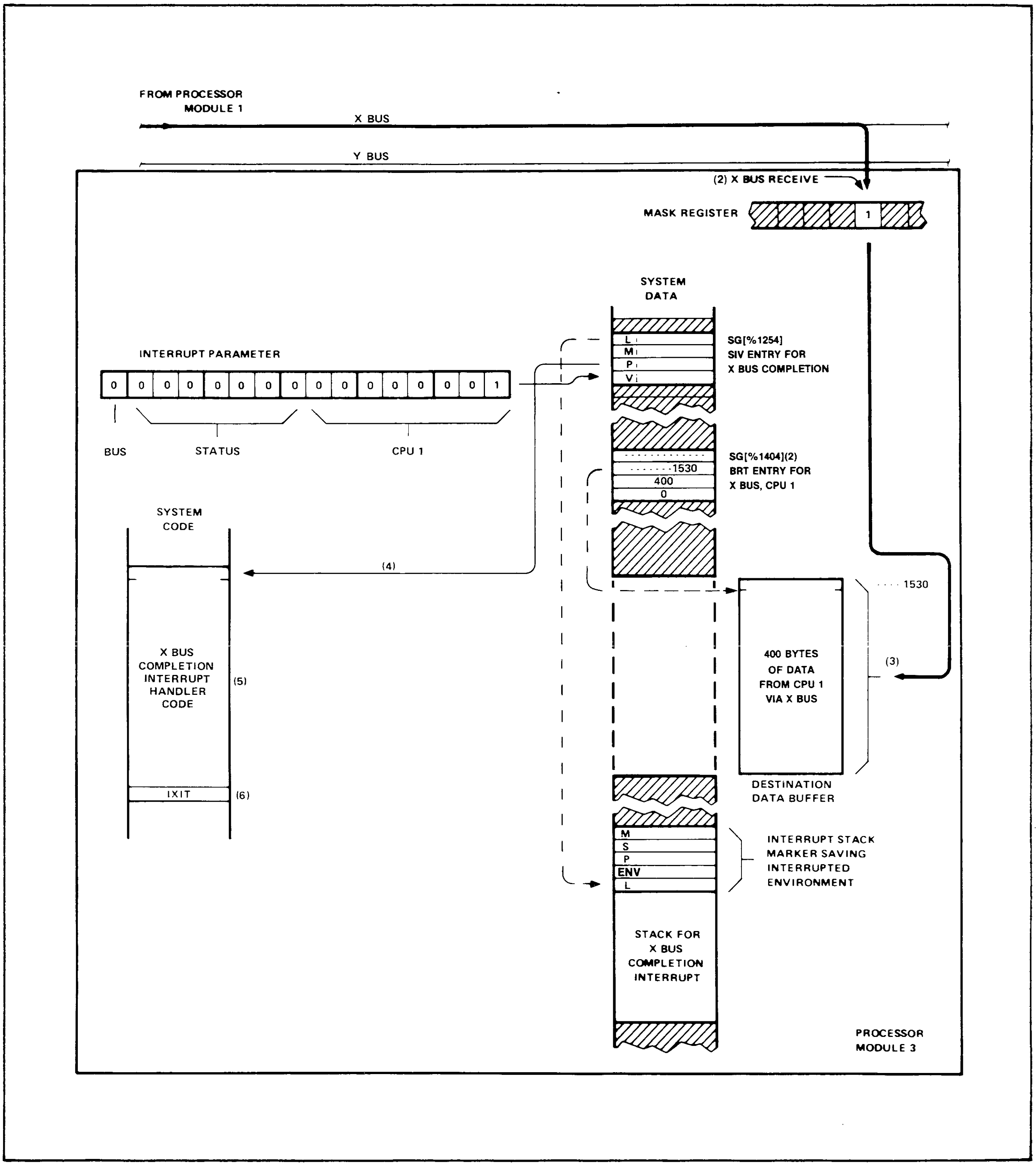


Figure 2-65b. Bus Transfer Sequence (Receive)

1. A SEND instruction is executed in the source processor module (processor module 1). The SEND parameters specify:

   - X-Bus to Processor Module 3 (stack register G).
   - A sequence number (ignored in this example) (F).
   - Sender cpu 1 and receiving cpu 3 (E).
   - A packet timeout value of 0 (meaning that a timeout occurs if a single packet transfer takes longer than 26 milliseconds) (D).
   - A source buffer location address of 1466, which represents only the word and byte field values (11 bits of B) of the full 32-bit virtual memory address. (This is an absolute extended address. For simplicity, the other 21 bits of the address, representing the segment and page fields, are ignored throughout this example. Refer to the "Memory Access" discussion for a description of virtual memory addressing using absolute extended addresses. Also note that since extended addresses are byte addresses, transfers on odd byte boundaries are permitted.)
   - A count of 400 bytes to be transmitted (A).

   The SEND instruction transmits the 400 bytes to processor module 3 via the X-bus, then completes. The parameters are deleted from the Register Stack and the condition code is set to CCE (indicating a successful operation).

2. Meanwhile, processor module 3, which has been previously readied for this transfer, has MASK.<11> set to a "1" to enable receipt of data over the X-bus and has its BRT entry for processor module 1 configured as follows:

   - The transfer address where the incoming data is to be stored, starting at byte address 1530.
   - The count of the number of bytes expected, 400.
   - The initial sequence number.

3. The data, as received, is stored away as indicated by the BRT entry. As the data is stored, the transfer address is incremented accordingly and the count is decremented accordingly.

4. When the count in the BRT entry reaches zero, 400 bytes have been received. At this point an interrupt occurs through the SIV (System Interrupt Vector) for interprocessor bus completion. The parameter associated with this type of interrupt contains the processor module number of the source processor module, the bus flag (0 in this example), and the error (also 0 in this example).

5. The interrupt handler code for bus completion now executes. Because INT.<11> in the interrupt register is now set, further data transmissions to this processor module over the X-bus are rejected. Additionally, the Mi word in the SIV entry for bus completion masks off further interrupts in the MASK.<11:12> positions.

6. When the IXIT instruction executes, the previous MASK register setting is restored. Since the interrupt handler has already reset INT.<11>, processor module 3 is again enabled for receiving data over the X-bus.

Figure 2-66 shows the relationships of the transfer address, count, and sequence number in the BRT entry, and also the incoming data storage in the transfer location.

OUTQ, INQ, and Packets

The interprocessor buses are significantly faster than memory. Therefore each processor has a buffered interface to both buses, consisting of two 16-word output buffers (called OUTQ X and OUTQ Y), and two 16-word input buffers (called INQ X and INQ Y). See Figures 2-67a and b.

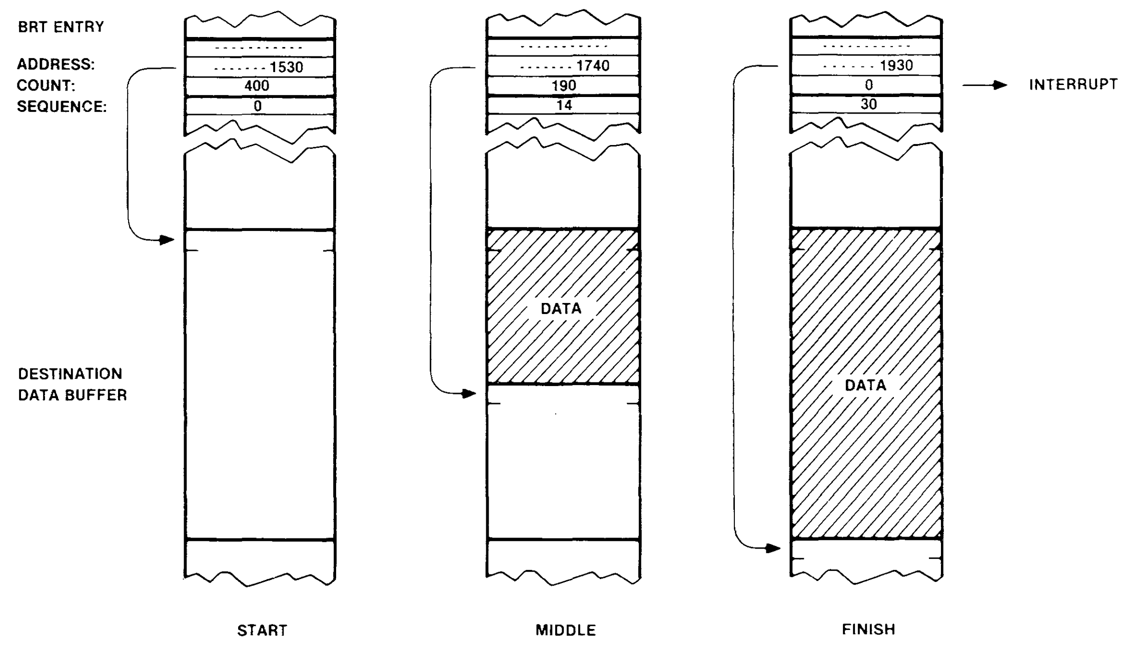


Figure 2-66. Incoming Data Storage

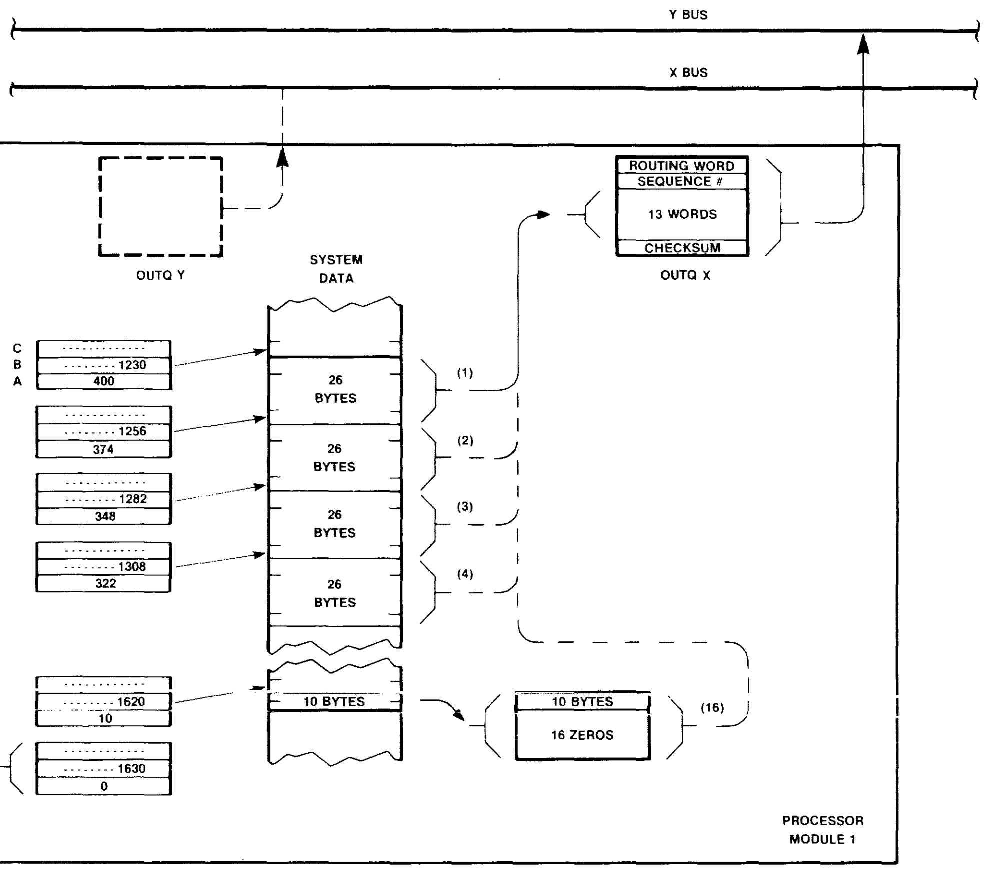


Figure 2-67a. Sending and Receiving Packets

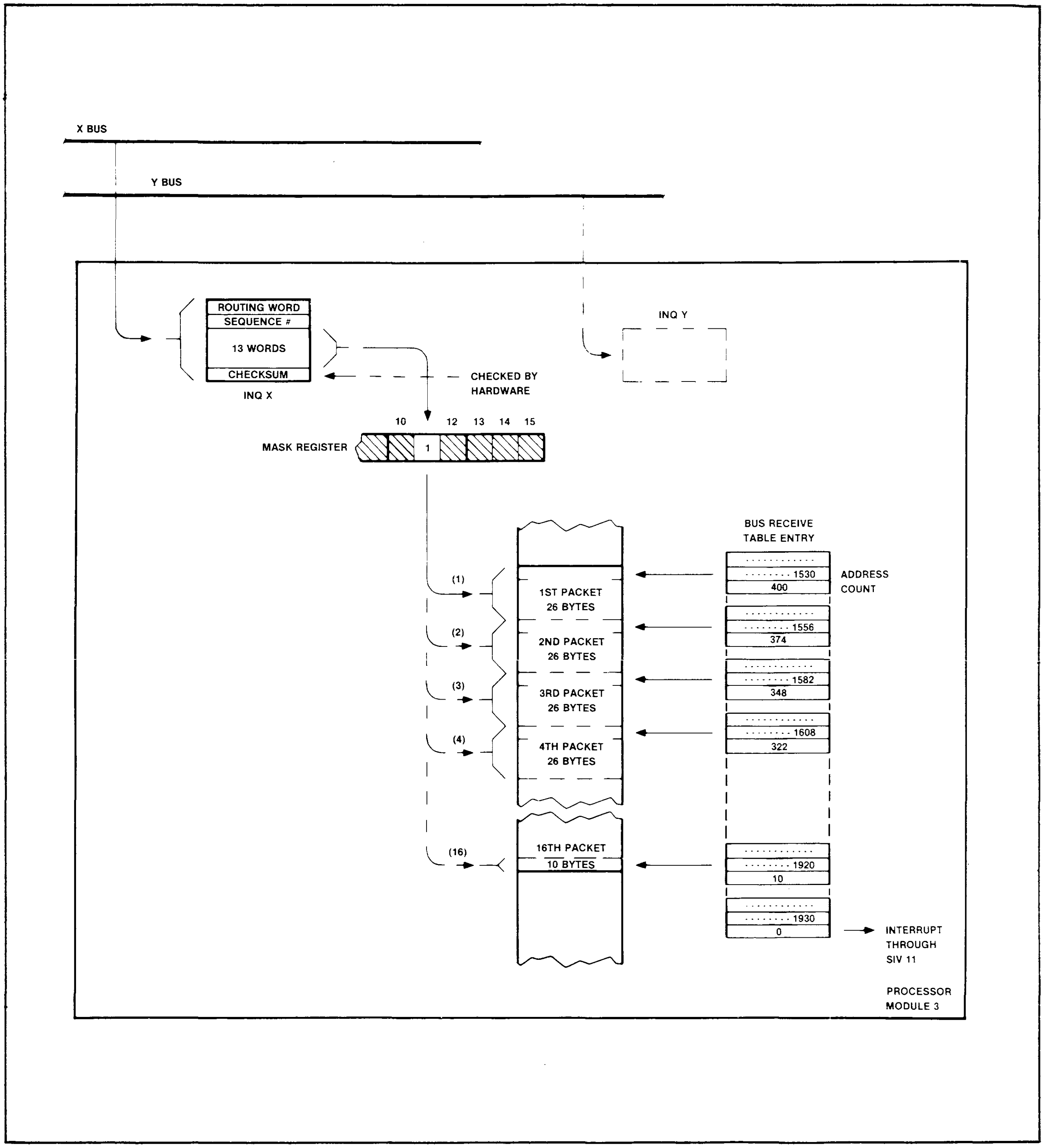


Figure 2-67b. Sending and Receiving Packets

Data is transmitted over a bus in the form of 16-word packets. The SEND instruction fills the output buffer with 26 data bytes (13 words), plus a one-word sequence number, one word for sender and receiver numbers, and a one-word odd-parity checksum. The instruction then signals the bus interface hardware that it has a packet ready for transmission. After the 16-word packet is transmitted, execution of the SEND instruction resumes at the point where it left off. If the last packet of the block contains less than 26 data bytes, the remaining data bytes are filled in with zeros. The SEND instruction terminates when the last packet is transmitted.

When either of the INQ X or INQ Y buffers in the destination processor module is filled and the corresponding MASK register bit is a "1", a microinterrupt occurs. The action taken by the processor module during the microinterrupt (which is transparent to the executing process and to the operating system) is:

- The count in the BRT entry is checked. If the count indicates that data is expected, 26 bytes (or less if the count is less) are read into memory at the location specified. The transfer address and count are then updated accordingly.

- The checksum of the packet is checked. If the checksum is valid and the count still exceeds zero, the INQ is marked empty (permitting further transmissions to take place) and the normal instruction execution sequence continues.

- If the count is now zero or if any transmission error is detected (checksum error, incorrect target, sequence error, etc.), the INT register bit associated with the bus used for the transmission sets, and an interrupt occurs. In the case of a transmission error, the count word is not updated. When a normal receive completes, the count word will contain zero.

INT and MASK Registers

These registers have a direct bearing on the ability of a processor module to accept data over an interprocessor bus. As shown in Figure 2-68, data packets from the buses are accepted into INQ X or INQ Y whenever the data is sent to this module (provided that the INQ is empty). Once the data is accepted, the corresponding bit in the Interrupt Register (bit 11 and/or 12 of INTA) is then set. If the corresponding bit of the Mask Register is also set (i.e., Mask and INTA bits ANDed together), a Bus Receive interrupt occurs that causes the IPU to transfer data to memory.

If a source processor module attempts a SEND to a processor module that is not enabled for receiving data (Mask bit inhibits destination IPU from emptying its INQ), the source module receives a Packet Timeout indication.

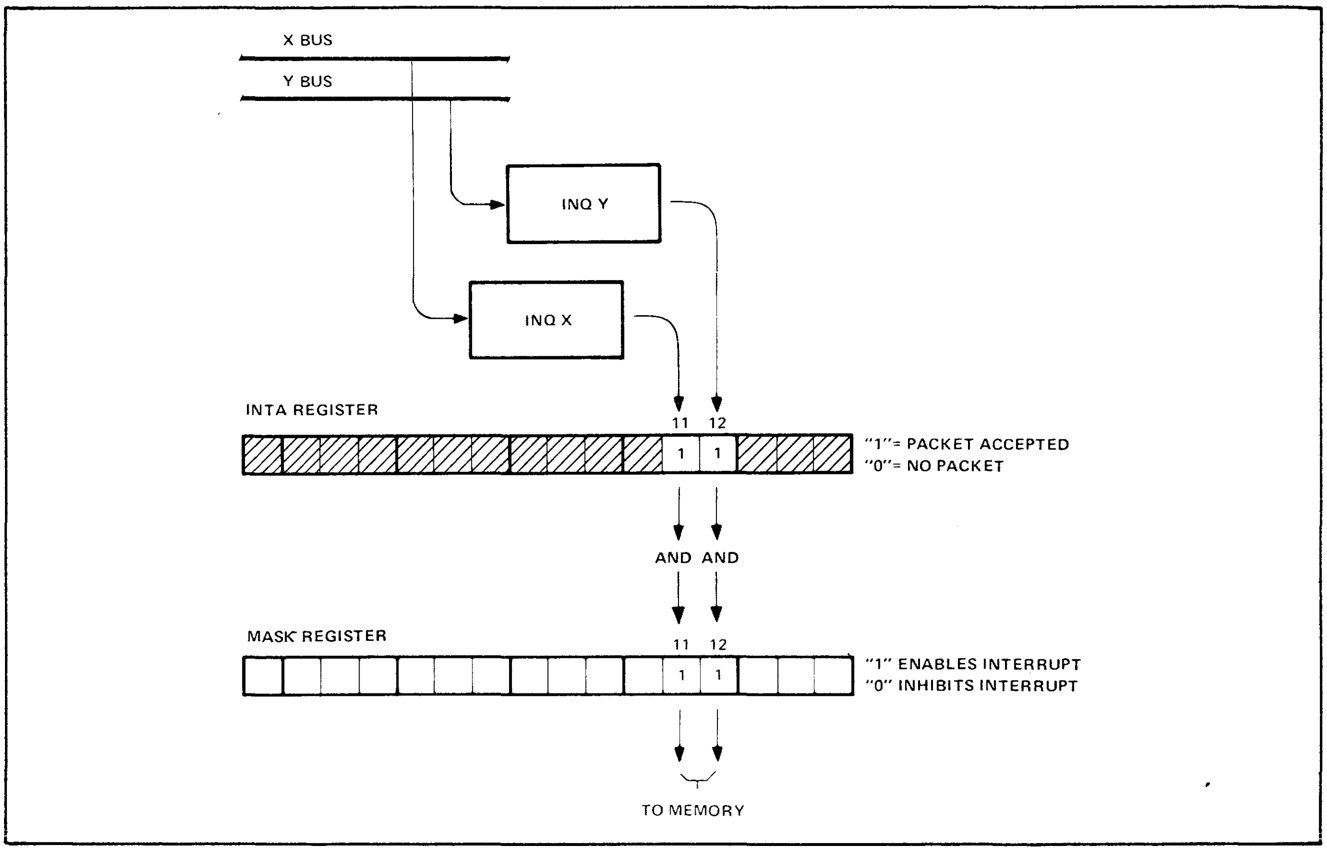


Figure 2-68. Bus Receive Enabling

INPUT/OUTPUT CHANNEL

Each processor module has a single block-multiplexed input/output channel through which all input/output takes place. Device-dependent i/o controllers are attached to the channel, and each controller may have one or more subchannels. A processor may address up to 256 subchannels. See Figure 2-69. Each controller is connected to two different processors, and the subchannel numbers that it responds to need not be the same on both processors. (Dual-port operation is considered later in this section.)

The first subchannel number for a given controller must be a multiple of 8, and the remaining subchannels follow in consecutive order.

The operating system performs input/output operations (see Figure 2-70) by first configuring an entry in a system table called the I/O Control table (IOC). The IOC contains 256 entries, one for each subchannel that can possibly communicate over the i/o channel. Each entry contains the address of the data buffer and a count of the number of bytes to be transferred. Once the entry corresponding to the device is configured, an EIO (Execute I/O) instruction is executed to initiate the i/o transfer; the actual data transfer is performed concurrently with program execution. When the transfer completes, an interrupt to an operating system interrupt handler takes place. In the interrupt handler, an IIO (Interrogate I/O) instruction or an HIIO (High-priority Interrogate I/O) instruction is executed to check the outcome of the operation.

I/O Control Table

The data to be transferred between memory and a specific unit is determined by an entry in the I/O Control Table (IOC). As illustrated earlier (Figure 2-55 and Table 2-2) this table occupies all of the second page of the System Data segment. It contains a four-word entry for every possible subchannel which may be connected to a processor module. See Figure 2-71.

The first word of the the IOC entry specifies the starting address of the i/o buffer in virtual memory. Bits 6 through 9 specify one of the maps, and bits 10 through 15 specify the starting logical page number within the map. It is permissible for i/o buffers to cross map boundaries.

The second word of the IOC entry specifies the number of bytes remaining to be transferred. This value is decremented after each word transfer.

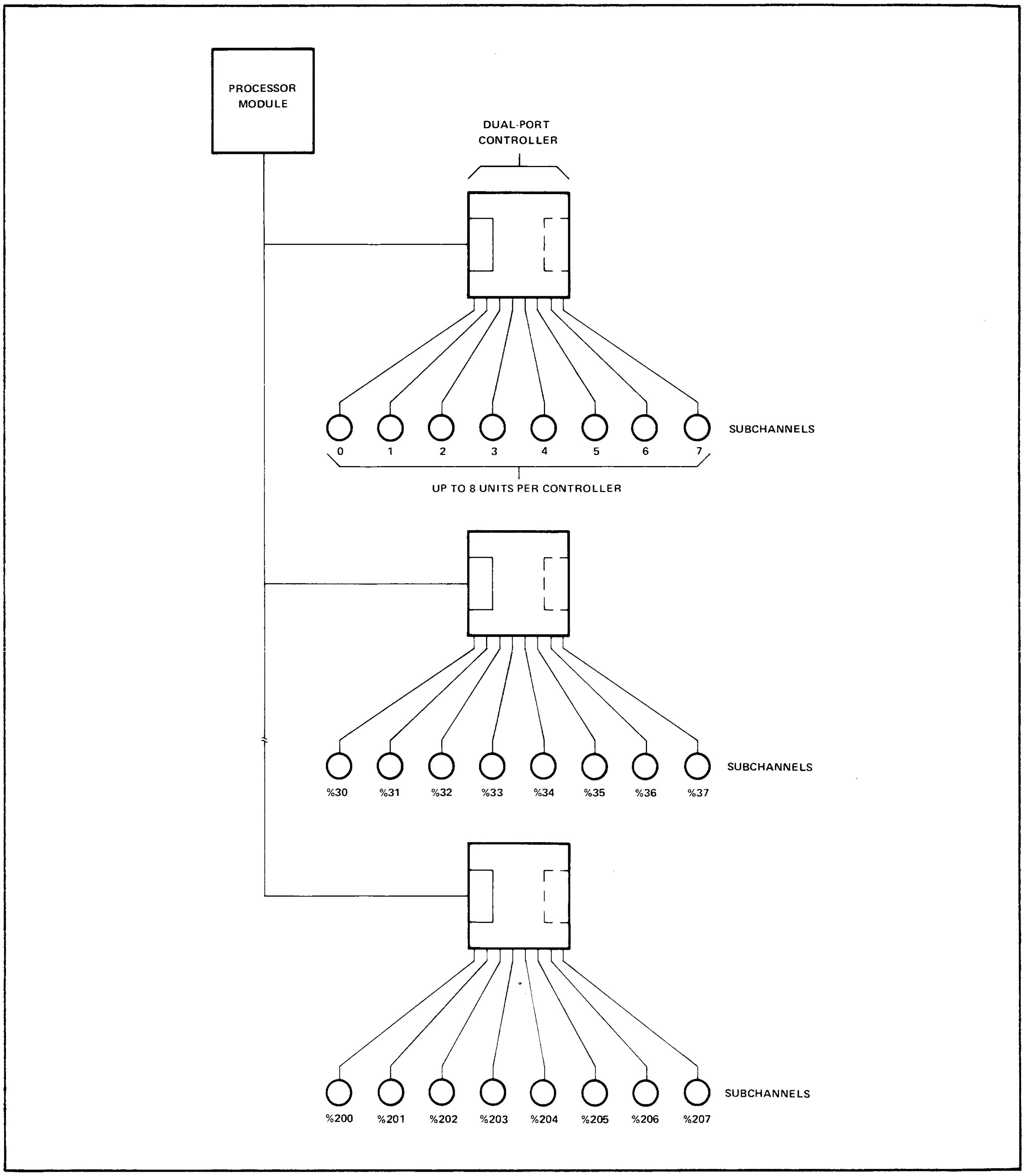


Figure 2-69. I/O Channel Addressing

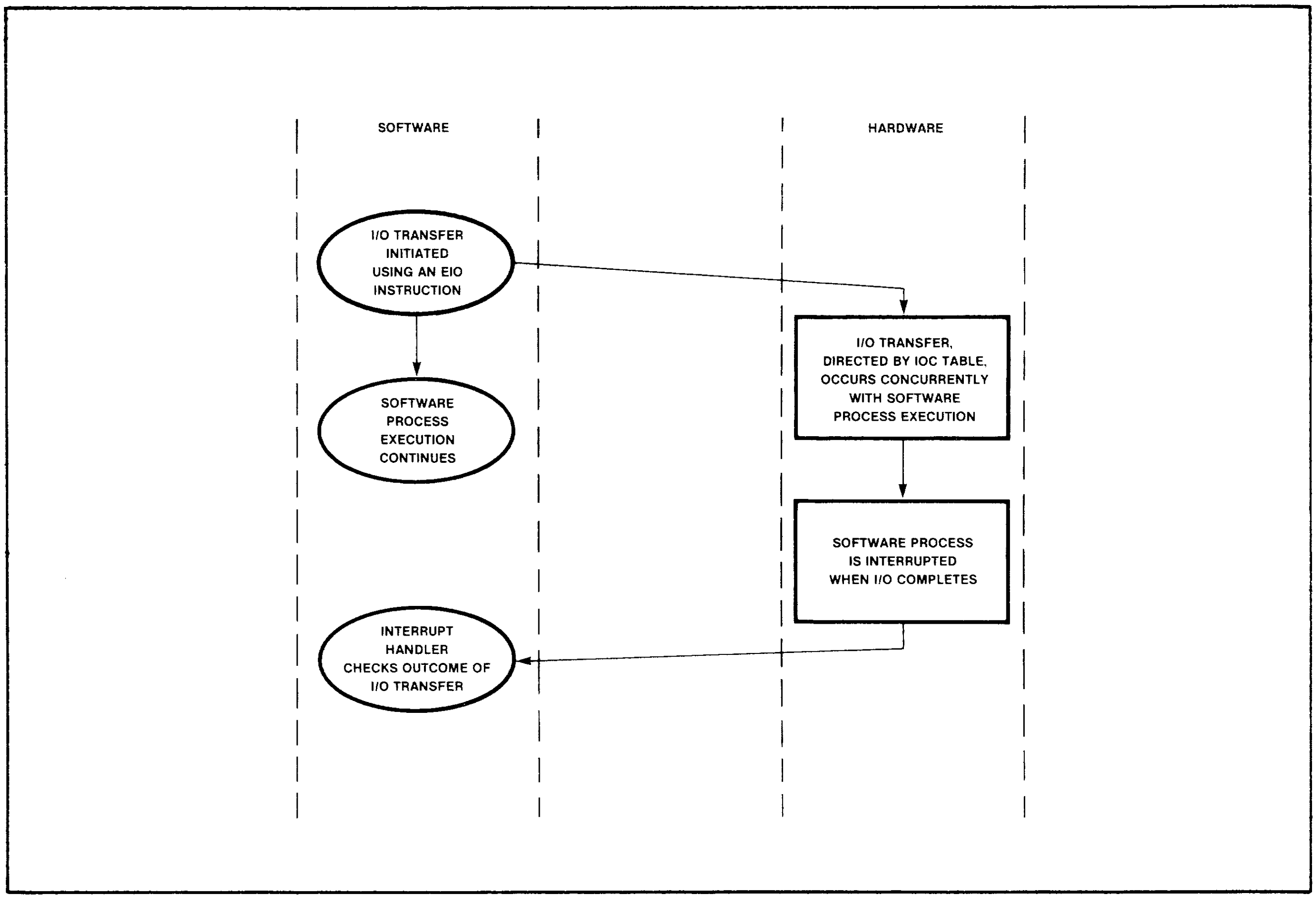
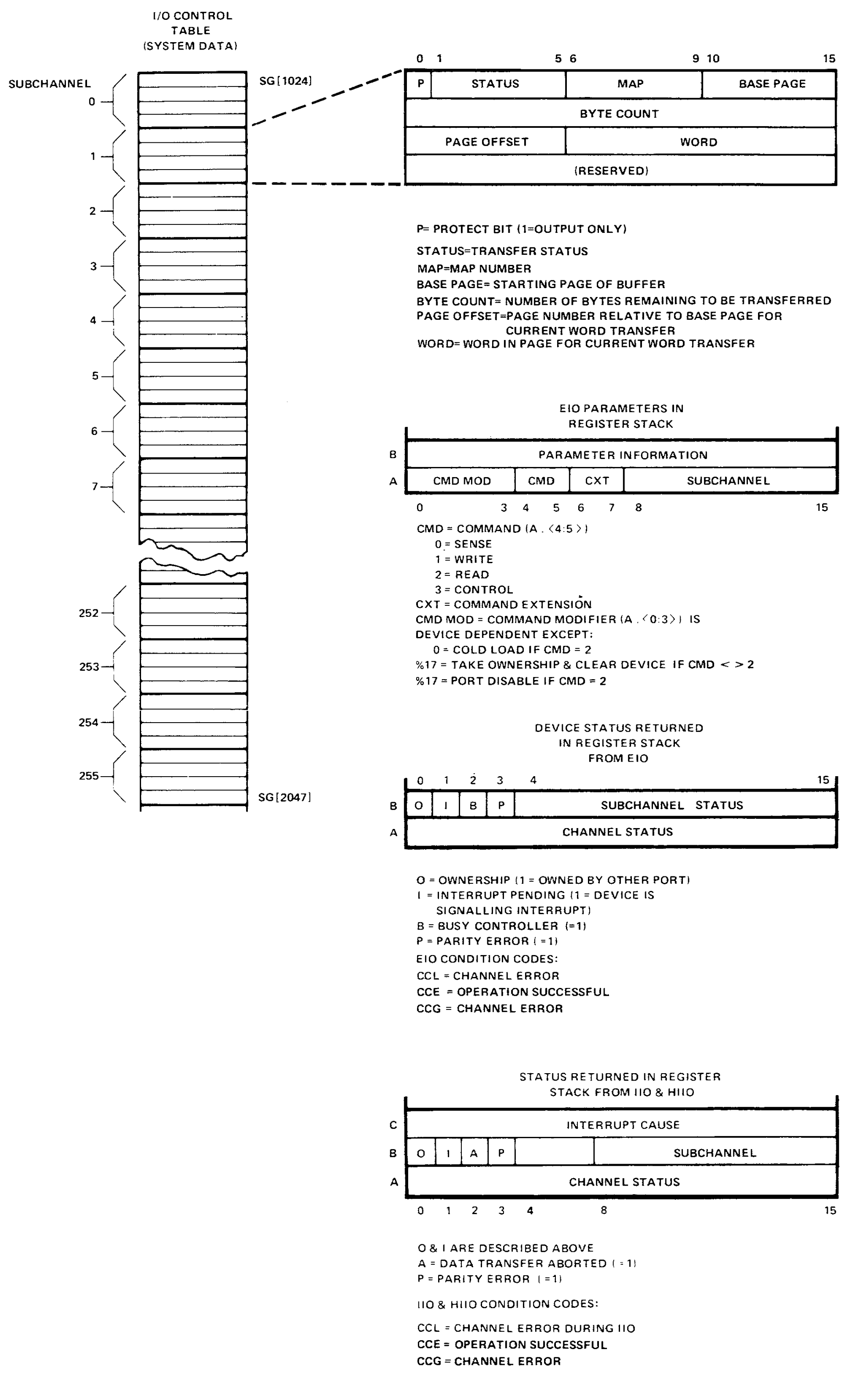

SOFTWARE
HARDWARE
I/O TRANSFER INITIATED USING AN EIO INSTRUCTION
SOFTWARE PROCESS EXECUTION CONTINUES
INTERRUPT HANDLER CHECKS OUTCOME OF I/O TRANSFER
I/O TRANSFER, DIRECTED BY IOC TABLE, OCCURS CONCURRENTLY WITH SOFTWARE PROCESS EXECUTION
SOFTWARE PROCESS IS INTERRUPTED WHEN I/O COMPLETES

Figure 2-70. Simplified I/O Sequence



P= PROTECT BIT (1=OUTPUT ONLY)
STATUS=TRANSFER STATUS
MAP=MAP NUMBER
BASE PAGE= STARTING PAGE OF BUFFER
BYTE COUNT= NUMBER OF BYTES REMAINING TO BE TRANSFERRED
PAGE OFFSET=PAGE NUMBER RELATIVE TO BASE PAGE FOR CURRENT WORD TRANSFER
WORD= WORD IN PAGE FOR CURRENT WORD TRANSFER

CMD = COMMAND (A . <4:5>)
  0 = SENSE
  1 = WRITE
  2 = READ
  3 = CONTROL
CXT = COMMAND EXTENSION
CMD MOD = COMMAND MODIFIER (A . <0:3>) IS DEVICE DEPENDENT EXCEPT:
  0 = COLD LOAD IF CMD = 2
%17 = TAKE OWNERSHIP & CLEAR DEVICE IF CMD < > 2
%17 = PORT DISABLE IF CMD = 2

O = OWNERSHIP (1 = OWNED BY OTHER PORT)
I = INTERRUPT PENDING (1 = DEVICE IS SIGNALLING INTERRUPT)
B = BUSY CONTROLLER (=1)
P = PARITY ERROR (=1)
EIO CONDITION CODES:
CCL = CHANNEL ERROR
CCE = OPERATION SUCCESSFUL
CCG = CHANNEL ERROR

O & I ARE DESCRIBED ABOVE
A = DATA TRANSFER ABORTED (=1)
P = PARITY ERROR (=1)
IIO & HIIO CONDITION CODES:
CCL = CHANNEL ERROR DURING IIO
CCE = OPERATION SUCCESSFUL
CCG = CHANNEL ERROR

Figure 2-71. Formats Associated with Input/Output

The third word of the IOC entry specifies the current word in the buffer that needs to be transferred. Since the page offset value given in bits 0 through 5 is relative to the base page value given in the first word of the entry, these two values are added together to derive the actual logical page in memory currently being accessed for word transfers. This value is incremented after each word transfer.

To prevent erroneous data transfers, the operating system either sets the second word in IOC entry to zero when transfers are not expected, or, if the last transfer was outbound, sets the protect bit. If a device attempts to transfer data and the byte count is zero, the i/o channel aborts the operation, causing an interrupt to occur. In such a case, the status returned by the device as a result of an IIO or HIIO reflects the error.

EIO Instruction

To perform an I/O operation, the IOC entry for the unit must first be correctly initialized. An EIO instruction is then executed, specifying the controller, unit, command, and other parameter information. These parameters are placed in B and A of the Register Stack. (See format in Figure 2-71.)

The parameters to the EIO instruction are described as follows:

- The parameter information word in B is a device-dependent parameter that is sent to the specified device.
- Command bits A.<0:5> specify the operation that the device is to perform. The CMD bits, A.<4:5>, specify the general type of command:

  0 = sense
  1 = write
  2 = read
  3 = control

  The CMD MOD bits, A.<0:3>, modify the command, allowing up to 64 device-dependent commands.

  Three configurations of these fields are reserved:

CMD	CMD MOD	Description
2	0	Perform cold load
3	%16	Disable port (kill)
3	%17	Take ownership and clear device

- The CXT bits, A.<6:7>, are available as command extension bits, specific to each device that requires them.

- The subchannel field, A.<8:15>, specifies one of 256 subchannels.

The EIO instruction replaces the two parameter words by two words containing the device status, and sets the condition code according to the outcome of the instruction. The condition code settings are as follows:

CCL:	Channel error (while executing EIO)
CCE:	Operation successful
CCG:	Channel, controller, or device error

The device status is of the form:

B.<0>	=	ownership
B.<1>	=	interrupt pending
B.<2>	=	busy
B.<3>	=	parity error
B.<4:15>	=	subchannel status
A.<0:15>	=	channel status

The status bits returned in B have the following meanings:

- O <ownership>, B.<0> is a "1" if the device is owned by other port. No data is transferred.

- I <interrupt pending>, B.<1> is a "1" if the device is interrupting. No data is transferred.

- B <busy>, B.<2> indicates that the device is already executing an i/o transfer (this includes seeking on a disc or rewinding on a magnetic tape). No data is transferred because of this EIO.

- P <parity>, B.<3> indicates (if a "1") that a parity error occurred.

IIO and HIIO Instructions

Following the successful initiation of an i/o operation by an EIO instruction, an interrupt occurs when the operation completes. At this point, an IIO (or HIIO) instruction must be executed to determine the cause of the interrupt. (IIO is "Interrogate I/O"; HIIO is "High-Priority Interrogate I/O".) When the IIO or HIIO is executed, the highest priority device with an interrupt pending returns its subchannel number, and status pertaining to the interrupt.

The three status words returned by the execution of an IIO or HIIO instruction to the Register Stack are of the form:

C.<0:15>	=	interrupt cause
B.<0>	=	ownership
B.<1>	=	interrupt pending
B.<2>	=	aborted
B.<3>	=	parity error
B.<8:15>	=	subchannel number
A.<0:15>	=	channel status

The status bits have the following meanings:

- The interrupt-cause field, C.<0:15>, is related to the particular subchannel that is interrupting.
- O (ownership), B.<0>, is a "1" if the controller is "owned" by the alternate port (see the description of "Dual Port Controllers and Ownership" that follows).
- I (interrupt pending), B.<1>, is a "1" if the device has an interrupt pending. Normally this should not be set at this time; otherwise some problem is indicated.
- A (aborted) B.<2>, is a "1" if the data transfer was aborted.
- P (parity error), B.<3>, is a "1" if a parity error was detected during the data transfer sequence.
- The subchannel field, B.<8:15>, is the controller and unit number associated with the interrupt.
- The channel status field, A.<0:15>, defines a possible channel error and may have the following values:

%000000	No error detected by the channel
%000100	Device Status <0:3> non-zero
%000200	Channel detected a parity error on RIC (Read Interrupt Command)
%000400	Channel detected a parity error on RIST (Read Interrupt Status) or RDST (Read Status)
%177777	Instruction timed out waiting for the i/o channel to become available
%1-----	Channel Status = IOBUS Control Field

Following execution of an IIO or an HIIO instruction, the condition code is set as follows:

CCL:	Channel error (while executing the instruction)
CCE:	Operation successful
CCG:	Channel, controller, or device error

Input/Output Sequence

A typical data transfer sequence over the input/output channel is depicted in Figure 2-72. The sequence is as follows:

1. Instructions in the i/o driver procedure are executed to configure the IOC entry for the subchannel where the transfer is to take place. In this case, the IOC entry is at SG[%2030] for subchannel 6.

2. The EIO parameters are loaded onto the Register Stack.

3. An EIO instruction is executed. The parameter information is sent to subchannel 6.

4. To indicate its outcome, the EIO instruction returns two status words to the top of the Register Stack and sets the Condition Code. These are checked by subsequent instructions.

5. Meanwhile, the data transfer takes place. Data is transferred from subchannel 6 to the location in memory indicated by the IOC entry for that subchannel. As the data is transferred into memory, the transfer address and count word in the IOC are updated accordingly.

6. When the count word in the IOC reaches zero, indicating that the transfer is completed, the channel signals the controller. The controller stops transferring and signals the IPU with an interrupt. The INTA.<14> bit in the interrupt register is set to "1" to signal interrupt pending. If the corresponding bit in the MASK register is set, an interrupt through the SIV entry for Standard I/O (at SG[696]) occurs. The Mi entry in the SIV causes any further standard i/o interrupts to be deferred while the i/o completion interrupt handler is active.

7. The interrupt handler executes an IIO instruction. Executing IIO signals the highest priority interrupting controller to stop interrupting and returns three words of status information to the top of the Register Stack. (Controller priorities are set into the hardware at installation time, and may be adjusted by Tandem field service representatives as necessary for load balancing.) The status words contain the subchannel number of the interrupting device as well as interrupt cause and channel status information.

8. When the interrupt handler for standard i/o completes, an IXIT instruction is executed. IXIT restores the previous Mask Register value (which allows any pending standard i/o interrupt to occur) and attempts to return control to the interrupted code. Typically the operating system intervenes at this point and the i/o process and, later, the user process are notified of the completion of the original input/output request.

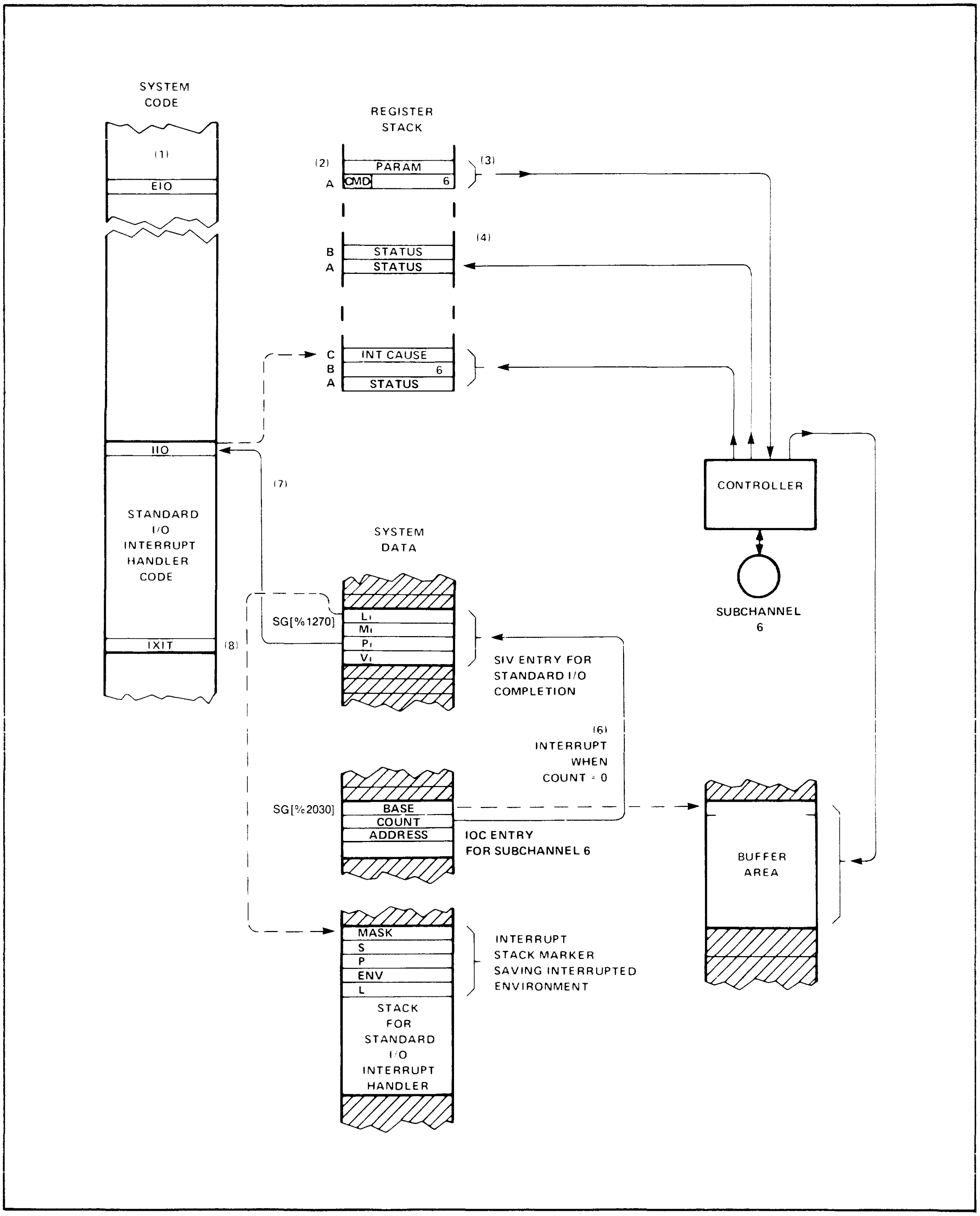


Figure 2-72. Input/Output Sequence

Dual-Port Controllers and Ownership

Each controller in the NonStop II computer system is connected to the input/output channels of two processor modules. This provides redundant communication paths to i/o devices. As shown in Figure 2-73, this means that a single subchannel has entries in the IOC's of two processor modules. Note that the ports need not have the same subchannel address on both channels.

Although each controller has two ports and is fully capable of communicating through either i/o channel, only one channel is used during normal operation; the other channel, as far as a particular controller is concerned, is not used. The i/o channel through which communication to a particular controller occurs is said to "own" the controller. All input/output transfers (i.e., control and data) occur through the channel owning the controller. This is illustrated in Figure 2-74.

Each of the two ports in a controller contains a flag bit known as the "ownership" error bit. The state of these bits determine the channel from which the controller will accept commands. An operating system configuration parameter specifies which channel is to be the primary channel of communication for a particular controller.

The operating system transfers data only through the owned side. (An attempt to communicate through the unowned side results in the EIO instruction being rejected with an ownership error). If, during the course of a data transfer, the primary path to the controller (i.e., the primary processor module, channel, or port) becomes inoperable, the operating system generally executes a "take ownership" operation (of an EIO instruction) over the alternate (backup) channel. (One exception: in case of a port failure on a multiple-controller device, the operation is retried using another controller, with no change of ownership.) The "ownership" bits in the controller switch over to point to the alternate i/o channel. All subsequent data transfers now occur through this channel.

Each port also has two "disable" bits that are separate from its ownership bits. A disable bit, if a "1", prevents a controller from transmitting information through that port onto an i/o channel. The disable bit is set by an EIO instruction "set disable" command. Normally, this is used by the operating system when a controller performs some unexpected action that could affect the entire channel. The disable bit is associated with a port, so if the malfunction is in one port, normal communication with the controller still occurs via the other port.

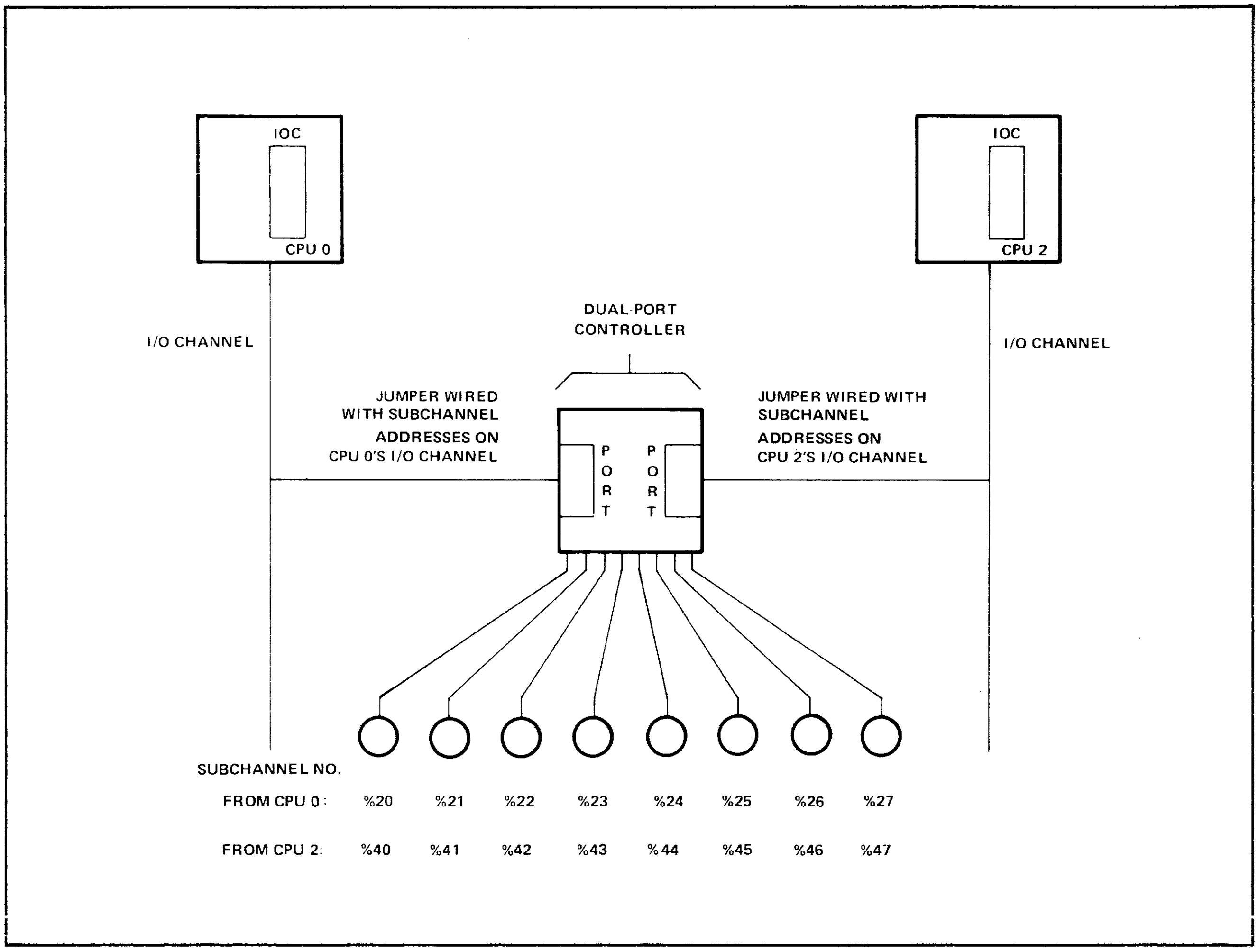

IOC
CPU 0
IOC
CPU 2
I/O CHANNEL
I/O CHANNEL
DUAL-PORT CONTROLLER
JUMPER WIRED WITH SUBCHANNEL ADDRESSES ON CPU 0'S I/O CHANNEL
JUMPER WIRED WITH SUBCHANNEL ADDRESSES ON CPU 2'S I/O CHANNEL
PORT
PORT
SUBCHANNEL NO.
FROM CPU 0: %20 %21 %22 %23 %24 %25 %26 %27
FROM CPU 2: %40 %41 %42 %43 %44 %45 %46 %47

Figure 2-73. Dual-Port Addressing

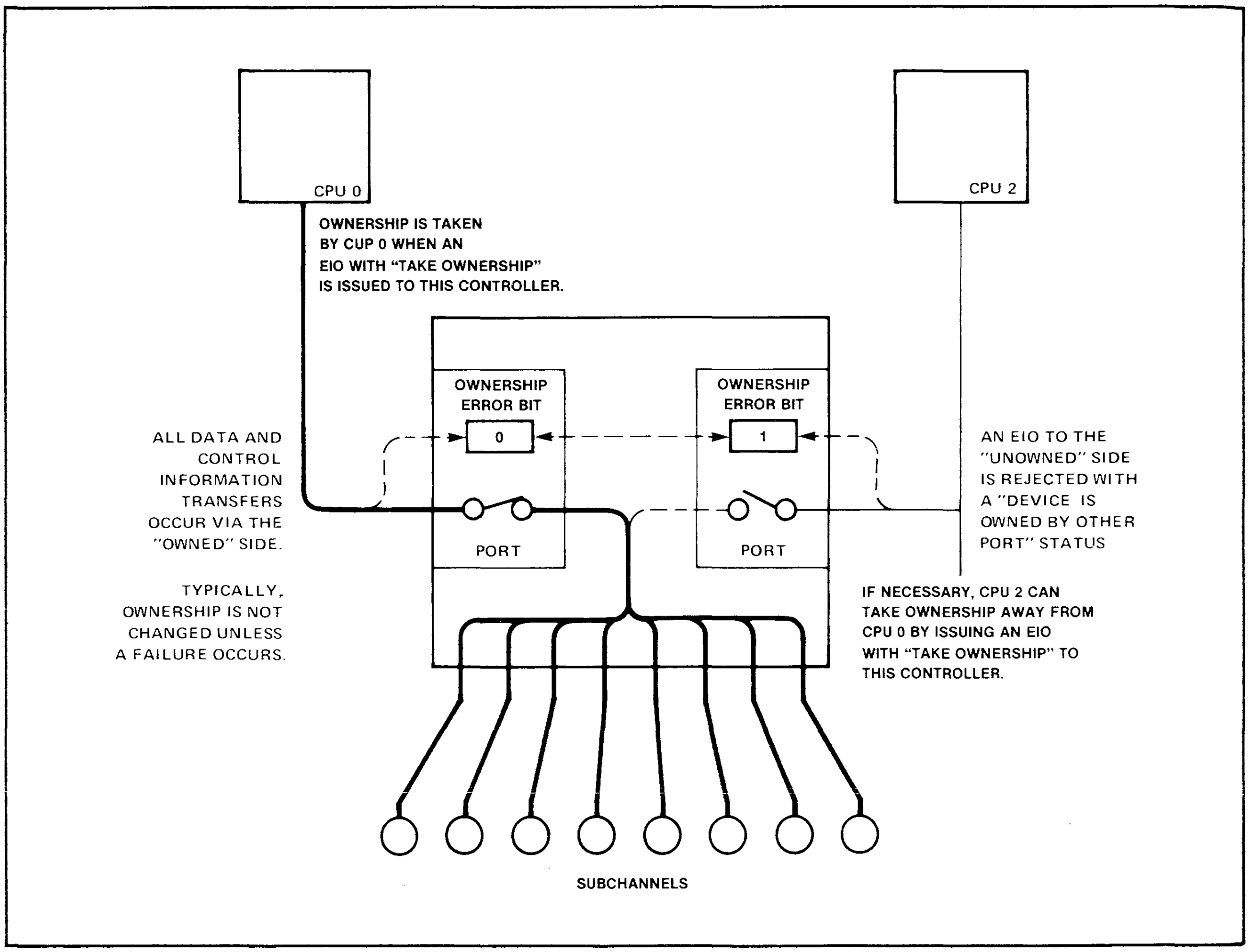

CPU 0
CPU 2
OWNERSHIP IS TAKEN BY CUP 0 WHEN AN EIO WITH "TAKE OWNERSHIP" IS ISSUED TO THIS CONTROLLER.
OWNERSHIP ERROR BIT
0
OWNERSHIP ERROR BIT
1
ALL DATA AND CONTROL INFORMATION TRANSFERS OCCUR VIA THE "OWNED" SIDE.
TYPICALLY, OWNERSHIP IS NOT CHANGED UNLESS A FAILURE OCCURS.
AN EIO TO THE "UNOWNED" SIDE IS REJECTED WITH A "DEVICE IS OWNED BY OTHER PORT" STATUS
PORT
PORT
IF NECESSARY, CPU 2 CAN TAKE OWNERSHIP AWAY FROM CPU 0 BY ISSUING AN EIO WITH "TAKE OWNERSHIP" TO THIS CONTROLLER.
SUBCHANNELS

Figure 2-74. I/O Controller Ownership

I/O Channel Interrupts

A controller signals an interrupt to the IPU when its associated transfer has completed. A controller also interrupts if it is necessary to terminate a transfer prematurely.

When simultaneous interrupts occur on an i/o channel, a priority scheme determines which interrupt is handled first. A subchannel continues to interrupt until cleared. Normally, this clearing is done via an IIO or HIIO instruction.

High-Priority I/O

Two levels of interrupt are available on an i/o channel: standard i/o and high-priority i/o. Standard i/o is characterized by controllers that interrupt through the SIV entry for standard i/o. Likewise, high-priority i/o is characterized by controllers that interrupt through the SIV entry for high-priority i/o. Whether a controller interrupts with standard or high priority is determined by a jumper connection on a controller.

High-priority i/o is used by applications requiring an ultra-fast response time (as in some communications environments). The operating system never masks off the high-priority interrupt position, thereby ensuring that no matter what is executing in a processor module, a high-priority interrupt will be recognized instantly.

MEMORY ACCESS

Logical vs. Physical Memory

Physical memory consists of some number of pages of main memory, each page holding 2048 bytes in specific fixed locations.

Logical memory, on the other hand, is not defined in terms of physical locations; instead, it is defined in terms of segments. A segment is a contiguous logical address space rather than a partition of memory. Thus, for example, if a program occupies 30 pages of a code segment (which allows for 64 pages), the other 34 pages are not wasted physical memory--only unused addresses.

Here is a list of thumbnail definitions for terms that are used in the following discussions.

<u>Standard Addressing Terms</u>	
page:	2048 bytes
logical page number:	0 to 63
logical address:	logical page, word, and byte
physical page number:	0 to 8191
physical address:	physical page, word, and byte
segment (nonextended):	a 1- to 64-page logical address space
logical segment:	any segment mapped by Maps 0 thru 5
<u>Extended Addressing Terms</u>	
relative segment number:	0 to 8191
absolute segment number:	0 to 8191
extended address:	segment, logical page, word, and byte
extended data segment:	1 byte to 128 megabytes
<u>Memory Entities</u>	
physical memory:	Up to 8192 pages of main memory
virtual memory:	Up to 524288 pages of disc + main memory
logical memory:	Up to 6 logical segments (Maps 0 thru 5)

In general usage, the term "segment" is usually understood to mean a nonextended segment--that is, 1 to 64 pages. When referring to an "extended" data segment, it is usually fully described as such.

Logical memory, the segments mapped by Maps 0 through 5, changes as different processes come into execution, since new sets of code and data are mapped by the "user maps." Thus, logical memory forms a time-variable subset of virtual memory.

Note also that there are four kinds of addresses. For standard (16-bit) addresses, there are logical and physical addresses. For extended (32-bit) addresses, there are relative and absolute addresses.

Memory Table Formats

Figure 2-75 illustrates the formats for the various address word and table entries. The following paragraphs describe each of these formats.

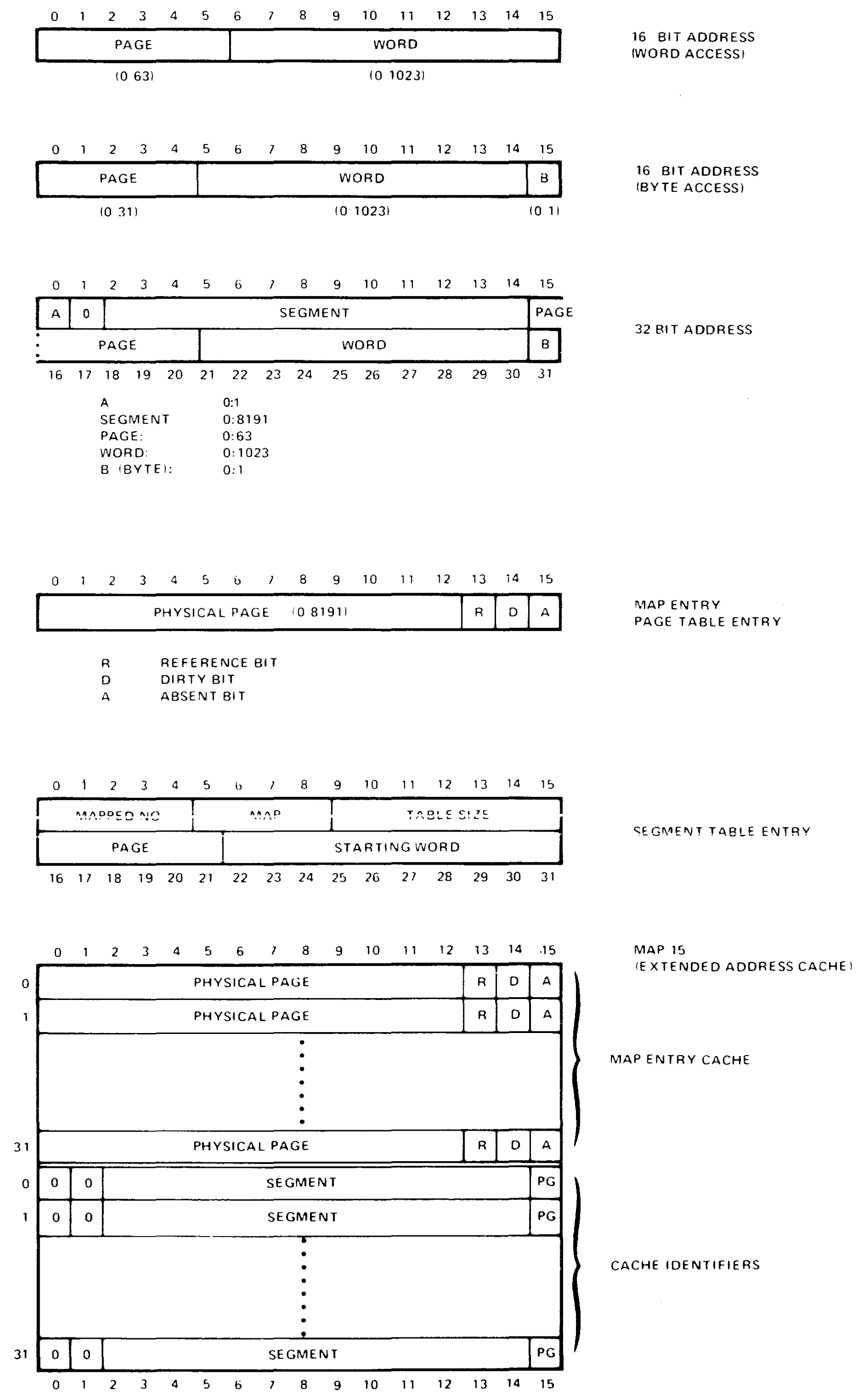


Figure 2-75. Formats Used in Memory Access Operations

16-BIT ADDRESS. 16-bit addresses are normally used to access both code and data. Depending on whether the instruction being executed is a word-addressing instruction or a byte-addressing instruction, a 16-bit address can take one of two forms, as shown in the first two formats. For word access, the first six bits (0 through 5) specify the logical page number. Bits 6 through 15 then specify which of the 1024 words on that page is the desired word. For byte access, bit 15 is used to specify a particular byte within a word: 0 for the left byte and 1 for the right byte. The page field of the address word in this case is therefore one bit smaller (bits 0 through 4), allowing only the first 32 pages of a segment to be accessed for byte access --that is, the first 32768 words of the segment. (For code addressing, however, both halves of the segment can be accessed, since the address is taken to be in the same 32 pages as the current setting of the P Register.)

32-BIT ADDRESS. This is the address format required for accessing extended data segments. The operating system can also use extended addressing to access any segment in virtual memory, either in absolute mode or in relative mode. Bit 0 of the address doubleword is used to specify the mode: 0 for relative mode (as in all user applications) or 1 for absolute mode (restricted to privileged users). Bit 1 is always 0. Bits 2 through 14 specify one of 8192 segments of virtual memory; bits 15 through 20 specify the page within the segment; bits 21 through 30 specify the word within the page; and bit 31 specifies the byte within the word if byte access is required. This format provides a 30-bit virtual address space (1073 megabytes). Unlike the 16-bit address form, the 32-bit address does not borrow a bit from the page field to allow a byte specifier; thus all 32-bit addresses are byte addresses.

MAP ENTRY. The processor uses entries kept in map registers to convert logical addresses to physical addresses. All words in the maps are formatted as shown in the map entry/Page Table entry layout (except in Maps 14 and 15, described below). Bits 0 through 12 specify a physical page number in the range of 0 through 8191. However, if the Absent bit (bit 15) is a 1, the page is logically absent, and attempting to access it will cause a page fault interrupt. Bit 13, the Reference bit, is set to 1 on any access to the page, and bit 14, the Dirty bit, is set to 1 on any write access to the page. These two bits are used by the Memory Management software to select the best pages for overlay when absent pages need to be brought into physical memory from disc, and to keep track of whether a page that is being replaced must first be copied to disc (i.e., is a dirty page). Since maps are loaded from Page Tables, this format also applies to Page Table entries and entries in the Map Entry Cache (see "Extended Address Cache Entries" below).

SEGMENT TABLE ENTRY. Segment Table entries are used to define the location of a Page Table for a particular segment. (For an explanation of the Page Tables, see the discussion of "Absolute Segment Addressing" later in this section.) Page Tables that are currently not in use (i.e., not "mapped") are located in a memory pool called MAPPOOL; however, if the table being sought is currently in a map, the only valid copy of the Page Table is the one in the map. In the latter case, bits 0 through 4 are used to specify that map number, and all other bits in the entry can be disregarded. But an entry of five 1's in this field indicates that the Page Table is not in a map, and in this case bits 5 through 31 are used to locate the table within MAPPOOL. Bits 5 through 8 specify which map defines the location of the desired Page Table; bits 9 through 15 specify the table size in words; bits 16 through 21 specify which entry in the map defines the physical page number; and bits 22 through 31 specify the word location on that page at which the Page Table actually starts.

EXTENDED ADDRESS CACHE ENTRIES. The Extended Address Cache (Map 15) is divided into two halves. The first 32 entries comprise the Map Entry Cache, and the second 32 entries are used for cache identifiers. Each entry of the Map Entry Cache is formatted identically to the map entry described above. The cache identifiers, however, each contain a 13-bit segment number and a single bit that represents the most significant bit of a page number. These bits are used to determine that the corresponding entry in the first half (the Map Entry Cache) is correct for the logical page being addressed.

Memory Maps

The complete set of maps for one processor is a 16 by 64 array of 1024 registers; that is, there are 16 maps, each consisting of 64 individual registers. These map registers define the logical memory and are used to provide the logical-to-physical address translation on an access to memory.

Each 64-register map defines a 64k word address space (maximum). These maps are used as follows:

0 User Data Segment. This map is loaded with the Page Table that defines the data space of a particular program when that program is activated. If DS is set to 0, all data references will be into the space defined by this map unless they are via instructions which use either extended addresses or the SG-relative addressing mode.

1 System Data Segment. This map defines space for system tables and stacks and for the interrupt handlers. The space defined by this map is common to all programs, but it may be accessed only if DS or PRIV is set. The following fixed tables known to the processor reside in the first two pages of this space:

Dummy Priority Value, (=%0)	%2
Current Process Control Block (CPCB)	%3
Ready List (RLIST)	%100:%101
Dummy Priority Value, (=%377)	%102
Microsecond Counter (CLOCK)	%103:%106
Time List Header (TLIST)	%107:%110
OSP I/O Control Block	%111:%114
Memory Breakpoint Trap Address (BPADDR)	%115:%116
Trace Buffer Base (TRBASE)	%117
Trace Buffer Limit (TRLIM)	%120
Trace Buffer Pointer (TRACE)	%121
LIGHTS Save Area	%122
Breakpoint Table Base (BPBASE)	%123
Breakpoint Table Entry Size (BPSIZE)	%124
Breakpoint Table Limit (BPLIM)	%125
Processor Dump Save Area	%1153:%1177
System Interrupt Vector (SIV)	%1200:%1337
Currently Mapped Segments (CMSEG)	%1340:%1357
Interprocessor Bus Error Packet	%1360:%1377
Bus Receive Table (BRT)	%1400:%1477
Input/Output Control Table (IOC)	%2000:%3777

System data pages 0 and 1 are always assigned to physical memory pages 0 and 1; these pages are always mapped. Physical page 2 is used as the power fail map save area. This page need not be mapped via any map during normal operation.

2 User Code Segment. All code space references specify the segment defined by this map if the CS and LS bits in the ENV Register are 0. In addition, the LWUC instruction always references this segment regardless of the ENV Register bit settings. This map is loaded with the Page Table that defines the code space of a particular program when that program is activated.

3 System Code Segment. All code space references (except via the LWUC instruction) specify the segment defined by this map if the LS bit in the ENV Register is 0 and the CS bit in the ENV Register is 1. This space is common to all programs.

4 User Library Code Segment. All code space references (except via the LWUC instruction) specify the segment defined by this map if the LS bit in the ENV Register is 1 and the CS bit in the ENV Register is 0. This map is loaded with the Page Table that defines the library code space of a particular program (if such space exists for the program) when that program is activated.

5 System Code Extension Segment. All code space references (except via the LWUC instruction) specify the segment defined by this map if the LS bit in the ENV Register is 1 and the CS bit in the ENV Register is 1. This space may be viewed as an extension to the System Code segment and is common to all programs.

6-13 Buffer Space. Buffers for i/o transfers and the Page Tables are normally mapped into this space.

14 This map is reserved by the system for special purposes, and is divided into several areas:

Microcode Scratch Registers	Entries 0:27
Segment Table (SEG)	28:43
Physical Page Segment Table (PHYSEG)	44:51
Physical Page/Logical Page Table (PHYPAGE)	52:59
Extended Address Base (Segment Base)	60:61
Extended Address Limit (Segment Limit)	62:63

15 Extended Address Cache. See Figure 2-75.

Absolute Segment Addressing

Each processor is viewed as having up to 8192 segments of virtual memory, with each segment having from 1 to 64 pages. This allows a processor to access up to 536,870,912 words of memory--that is, 64 times its maximum possible physical memory.

Segment numbers may be in the range of 0 through 8191, page numbers in the range of 0 through 63, and byte-in-page numbers in the range of 0 through 2047. This then gives each processor a virtual address space of the size: 8192*64*2048 bytes, or 1073 megabytes.

However, such an address requires 30 bits to represent it. To accommodate this, a 32-bit addressing word is used. An extended address is a 32-bit value having the following format (see Figure 2-75):

0	Absolute
1	Not Used (=0)
2:14	Segment
15:20	Logical Page
21:30	Word
31	Byte

The Absolute addressing bit (A) indicates whether the address is to be a relative address (=0) or is absolute (=1).

The Segment field (2:14) indicates the number of the segment (0:8191) in which the item is found.

The Page field (15:20) defines the logical page (0:63) within the segment.

The Word field (21:30) defines the word (0:1023) within the page.

The Byte field (31) defines the byte (0:1) within the word.

Each segment has an entry in the Segment Table, which contains the address of the Page Table for the segment. Each segment's Page Table contains entries which define the physical memory location (if present) where each page of the segment resides.

Access to memory then occurs as follows. First, the segment number is used as an index into the Segment Table to find the address of the Page Table; second, the page number is added to the address of the Page Table and this is used to read the physical page number from memory; finally, the physical page number is used with the word address to access the desired word in memory.

The Segment Table provides, for each segment, a two word entry formatted as follows (see Figure (2-75):

0:4	Map Number if Mapped	
5:8	Map Number of Page Table	
9:15	Table Size	
16:21	Page Number	address of Page Table
22:31	Starting Word	within the map

The first Map Number field (0:4) indicates the number (0:15) of the map which contains the segment's Page Table if the segment's Page Table has been loaded into one of the maps, or contains a %37 if the segment is not currently mapped. (A segment, such as a process' code space, might be in a map, such as the User Code map, when an extended address reference was made to it. In such a case, the Page Table entry in the map is accessed rather than the copy of the Page Table in memory.)

The second Map Number field (5:8) defines the map (0:15) which defines the address space containing the Page Table for the segment.

The Table Size field (9:15) defines the number of pages (0:64) that are contained in the segment.

The Page Number and Starting Word fields (16:31) define the address (within the space mapped by the map defined in bits 5:8) where the Page Table for the segment is stored.

Each segment's Page Table contains a one-word entry for each page in the segment. Each of these entries is of the same format as entries in a map (see Figure 2-75).

Using the above defined data structures, a byte with an absolute extended address in logical memory is found by the following steps:

1. First, the Page Table is found by indexing into the Segment Table using the Segment Number field of the address.

2. The Page Number field of the address is used to access the Page Table to see if the page is in main memory. If the page is not in main memory, indicated by the Absent bit being set, then a Page Fault interrupt occurs.

3. On the other hand, if the page is in main memory, then the Physical Page field of the Page Table entry is used to select a physical page of main memory.

4. Finally, the Word and Byte fields of the address specify one of the 2048 bytes on that page in memory for access.

If a page fault occurs, then the operating system must bring the page into main memory. The instruction which got the page fault is then retried.

On any access to a given page, the R bit of the map element for that page is set to 1 if it is not already set, and if the access is a write, the D bit is set to 1 as well.

Byte addressing is not handled by the map or the memory, but must be done by the IPU. On a byte read, the word containing the byte is read, and then the IPU selects the appropriate byte. On a byte write, the word containing the byte is read, the byte is changed by the IPU, and then the word is written back to memory.

Relative Segment Addressing

Although internally the operating system must use absolute segment numbers, this is never the case for user processes. A relative segment mechanism is defined which is the default mode of access. A relative segment address is similar to the absolute segment address, except that the Segment Number field defines a relative rather than an absolute segment. The two types of addresses are differentiated by the A (Absolute) bit in the address, and only privileged programs may use absolute extended addresses.

The first four relative segment numbers are defined for standard (register-relative) addressing of code and data--though extended addresses may also reference these segment numbers. These four defined segment numbers are:

0 Current Data Segment. The DS bit of ENV selects whether Map 0 or Map 1 holds the Page Table for the appropriate segment. This provides access to the same segment that a LOAD G+0 would access.

1 System Data Segment. The PRIV bit of ENV selects whether Map 0 or Map 1 holds the Page Table for the appropriate segment. This provides access to the same segment that a LOAD SG+0 would access.

2 User Code Segment. Map 2 holds the Page Table for the appropriate segment. This provides access to the same segment that an LWUC instruction would access.

3 Current Code Segment. The combination of the LS and CS bits in ENV defines the map number of the map which holds the Page Table for the appropriate segment. This provides access to the same segment that instructions are fetched from or that an LWP instruction would access.

Extended Data Segments

For the four relative segments previously mentioned, the limitation exists that the size of a segment is 64 pages (128k bytes), which in turn puts definite limits on program and data structure sizes. However, this limit is greatly expanded for access to data in the fifth relative segment type:

4-n Extended Data Segment. As many absolute segments as necessary are allocated to accommodate the extended segment size requested in an ALLOCATESEGMENT procedure call to the operating system. The segment size is specified as a number of bytes.

This segment is not defined by a map, but is accessed via the Segment Table and one or more Page Tables. Each process has a segment base register and a segment limit register maintained by the operating system. A relative segment number of 4 or higher results in the address being checked against the limit register, and then the base register is added to the logical address to form an absolute extended address.

To minimize the number of memory accesses to the various tables, two special applications of Maps 14 and 15 are used. First, the relocation values for the current process are saved in four map entries:

Map 14, entries 60:61	Segment Base (base extended address)
Map 14, entries 62:63	Segment Limit (one's complement of the maximum allowed address)

Second, Map 15 is used as a cache for map entries. After the extended address has been optionally relocated and bounds-tested, the cache is examined to see if the appropriate page of the segment has its Page Table map entry in it. This is done by reading MAP[15,32+(page mod 32)] and comparing that value with the high-order word of the extended address. If they are equal, then MAP[15, page mod 32] contains the Page Table map entry needed, and memory may be accessed via that map entry. On the other hand, if there is no match, then the entry in the cache must be written back to the appropriate Page Table (to save the current R, D, and A values), and the correct entry can then be cached.

The first half of the cache holds Page Table map entries (see Figure 2-75), and the second half of the cache holds entries which identify the Page Table map entry that has been cached. This latter entry consists of the segment number in bits 2 through 14, and the most significant bit of the page number in bit 15. An entry with the value %177777 indicates that the corresponding cache entry is empty.

Extended Address Instructions

The NonStop II processor provides a new class of instructions to access data using extended addresses. These instructions are capable of accessing memory which is not referenced in any of the maps. An example of this is the MVBX instruction, which allows bytes to be moved from one extended address to another. In addition, all interprocessor bus transfers use these addresses, thus opening up the processor's entire address space for transfers.

The following is a list of extended addressing instructions. These 23 instructions are nonprivileged, and most are supported by TAL language constructs. (Exceptions are MNDX, XSMX, and CDX.)

ANX	AND to Extended Memory
ORX	OR to Extended Memory
MNDX	Move Words While Not Duplicate
XSMX	Compute Checksum Extended
CDX	Count Duplicate Words Extended
LBX	Load Byte Extended
SBX	Store Byte Extended
LWX	Load Word Extended
SWX	Store Word Extended
LDDX	Load Doubleword Extended
SDDX	Store Doubleword Extended
LQX	Load Quadrupleword Extended
SQX	Store Quadrupleword Extended
DFX	Deposit Field Extended
MVBX	Move Bytes Extended
MBXR	Move Bytes Extended, Reverse
MBXX	Move Bytes Extended, Checksum
CMBX	Compare Bytes Extended
SCS	Set Code Segment
LWXX	Load Word Extended
SWXX	Store Word Extended
LBXX	Load Byte Extended
SBXX	Store Byte Extended

Memory Errors

Correctable and uncorrectable memory errors are reported to the processor either as interrupts or as i/o termination conditions. An uncorrectable error generally indicates that the page should no longer be used. A correctable error, on the other hand, may occur because of either a transient failure or a hard error. A hard error can be detected by rewriting a page that gets a correctable error and then seeing if the error occurs again. A privileged instruction, CMRW, is used by the operating system for this purpose; this instruction holds off memory accesses by the i/o channel while a word of memory is being rewritten.

SECTION 3

INSTRUCTION SET

GENERAL INFORMATION

The instruction set of the NonStop II system, including the decimal arithmetic and floating-point options, consists of approximately 280 machine instructions. This section provides text descriptions of all these instructions, with the exception of those reserved for operating system use. Diagrams are also included showing the action of some of the more commonly used instructions. To locate the text description for any instruction, refer to the alphabetical listing under "Instructions" in the general index at the back of this manual.

These descriptions assume familiarity with the information presented in Section 2. For explanations of terms and concepts mentioned here, refer to the Index to find the appropriate portions of Section 2.

In addition, Appendixes A and B provide a number of useful reference tables pertaining to the instruction set.

Instructions in this section are categorized by general function and discussed under the following headings:

- 16-Bit Arithmetic
- 32-Bit Signed Arithmetic
- 16-Bit Signed Arithmetic (Register Stack Element)
- Decimal Arithmetic Store and Load (Standard Instructions)
- Decimal Integer Arithmetic (Standard and Optional Instructions)
- Decimal Arithmetic Scaling and Rounding (Standard and Optional Instructions)
- Decimal Arithmetic Conversions (Optional Instructions)
- Floating-Point Arithmetic (Optional Instructions)
- Extended Floating-Point Arithmetic (Optional Instructions)
- Floating-Point Conversions (Optional Instructions)
- Floating-Point Functionals (Optional Instructions)
- Register Stack Manipulation
- Boolean Operations
- Bit Deposit and Shift
- Byte Test

Memory Stack to/from Register Stack
Load and Store Via Address on Register Stack
Branching
Moves, Compares, Scans, and Checksum Computations
Program Register Control
Routine Calls and Returns
Interrupt System
Bus Communication
Input/Output
Miscellaneous
Operating System Functions

NOTE

The instruction descriptions in this section state the conditions under which Overflow is set in the ENV Register. For details on the setting of the Condition Code and Carry bits, refer to "Program Environment" in Section 2. Unless otherwise stated, "stack" refers to the Register Stack.

16-BIT ARITHMETIC (Top of Register Stack)

IADD (000210). Integer (signed) Add A to B. A is added to B in integer form. A and B are then deleted from the stack and the sum is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LADD (000200). Logical (unsigned) Add A to B. A and B are added as 16-bit positive integers. A and B are then deleted from the stack and the result pushed on. Carry is set if the addition overflows bit 0. Condition Code is set.

ISUB (000211). Integer (signed) Subtract A from B. A is subtracted from B in integer form. A and B are deleted and the difference is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LSUB (000201). Logical (unsigned) Subtract A from B. A is subtracted from B logically. A and B are then deleted from the stack and the result pushed on. Carry is set if A is less than or equal to B. Condition Code is set.

IMPY (000212). Integer (signed) Multiply A times B. B is multiplied by A in integer form. A and B are deleted from the stack and the result pushed on. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LMPY (000202). Logical (unsigned) Multiply A times B. A and B are multiplied as 16-bit positive integers. A and B are then replaced by the doubleword result, with the least significant half in A. Overflow is implicitly cleared. Condition Code is set.

IDIV (000213). Integer (signed) Divide B by A. B is divided by A in integer form. A and B are deleted from the stack and the result pushed on. Overflow is set if the divisor is zero, or if the result is greater than 32767 or less than -32768. Condition Code is set.

LDIV (000203). Logical (unsigned) Divide CB by A, leaving the remainder in B. The 32-bit positive integer in C and B is divided by the 16-bit positive integer in A. The divisor and dividend are deleted from the stack, the remainder is pushed onto the stack (B), and the quotient is pushed onto the stack (A). Overflow is set if the original C is greater than or equal to the original A. Condition Code is set.

INEG (000214). Integer (signed) Negate A. A is converted to its two's complement form. Overflow is set if the original operand was -32768. Condition Code is set.

LNEG (000204). Logical (unsigned) Negate A. A is converted to its two's complement. Carry is set if the original value of A is zero. Condition Code is set.

ICMP (000215). Integer (signed) Compare B with A. B is compared to A in integer form and the Condition Code set accordingly. A and B are then deleted from the stack.

LCMP (000205). Logical (unsigned) Compare B with A. B is logically compared to A and the Condition Code set accordingly. A and B are then deleted from the stack.

16-Bit Arithmetic

CMPI (001---). Compare A with Immediate Operand. The Condition Code is set as a result of the 16-bit integer comparison of A and the immediate operand. A is then deleted from the stack. Examples of the use of immediate operands are shown in Figure 3-1.

ADDI (104---). Add Immediate Operand to A. The immediate operand is added to A in integer form. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

LADI (003---). Logical (unsigned) Add Immediate Operand to A. The immediate operand is added to A in 16-bit unsigned integer form. Condition Code is set.

32-BIT SIGNED ARITHMETIC

DADD (000220). Double Add DC to BA. The two doubleword integers contained in DC and BA are added in doubleword integer form. Both operands are then deleted, and the doubleword result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DSUB (000221). Double Subtract BA from DC. The doubleword integer contained in BA is subtracted in doubleword integer form from the doubleword integer in DC. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DMPY (000222). Double Multiply DC by BA. The doubleword integer contained in DC is multiplied in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31). Carry can be set, and Condition Code is set on the result.

DDIV (000223). Double Divide DC by BA. The doubleword integer contained in DC is divided in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than (2**31)-1 or less than -(2**31), or if the divisor (BA) is zero. Carry can be set, and Condition Code is set on the result.

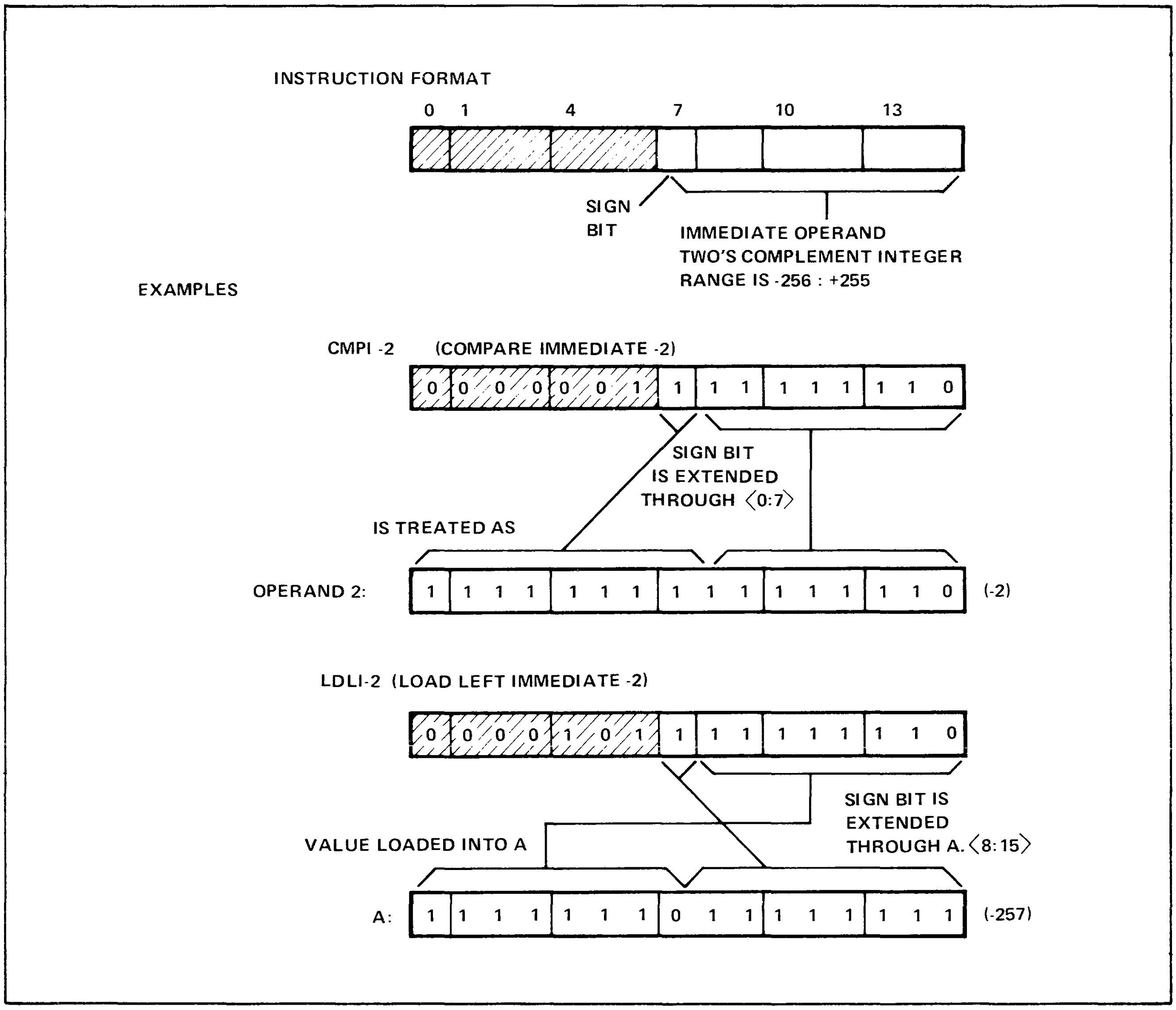

INSTRUCTION FORMAT
0 1 4 7 10 13
SIGN BIT
IMMEDIATE OPERAND
TWO'S COMPLEMENT INTEGER
RANGE IS -256 : +255
EXAMPLES
CMPI -2 (COMPARE IMMEDIATE -2)
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0
SIGN BIT IS EXTENDED THROUGH ⟨0:7⟩
IS TREATED AS
OPERAND 2: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 (-2)
LDLI-2 (LOAD LEFT IMMEDIATE -2)
0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 0
SIGN BIT IS EXTENDED THROUGH A.⟨8:15⟩
VALUE LOADED INTO A
A: 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 (-257)

Figure 3-1. Immediate Operand

DNEG (000224). Double Negate BA. The doubleword integer contained in BA is replaced with its two's complement. Overflow is set if the original operand was -(2**31). Carry can be set, and Condition Code is set on the result.

DCMP (000225). Double Compare DC with BA. The Condition Code in the ENV Register is set as a result of the doubleword integer comparison of DC and BA. Both operands are then deleted from the stack.

DTST (000031). Double Test BA. The Condition Code is set according to the contents of the doubleword contained in BA.

CDI (000307). Convert Double to Integer. The doubleword integer in BA is converted to a singleword integer by copying the contents of A into B and deleting A. Overflow is set if the doubleword quantity is greater than 32767 or less than -32768.

CID (000327). Convert Integer to Double. The singleword integer in A is extended to a doubleword quantity on the top of the Register Stack. A is copied into H, and then A is filled with zeros if A was positive, or ones if A was negative; the Register Pointer is incremented to give the result in BA.

MOND (000001). Minus One Double. A doubleword minus one is pushed onto the top of the Register Stack (BA). Condition Code is set.

ZERD (000002). Zero Double. A doubleword zero is pushed onto the top of the Register Stack (BA). Condition Code is set.

ONED (000003). One Double. A doubleword of one is pushed onto the top of the Register Stack (BA). Condition Code is set.

16-BIT SIGNED ARITHMETIC (Register Stack Element)

NOTE

For binary coding details of the first four instructions that follow (ADRA, SBRA, ADAR, SBAR), refer to Table A-7 in Appendix A. For ADXI, refer to Table A-4.

ADRA (00014-). Add Register to A. The contents of the register pointed to by the Register field of the instruction is added in integer form to register A. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

SBRA (00015-). Subtract Register from A. The contents of the register pointed to by the Register field of the instruction are subtracted in integer form from register A. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

ADAR (00016-). Add A to a Register. A is added in signed integer form to the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

SBAR (00017-). Subtract A from a Register. A is subtracted in signed integer form from the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

~~ADXI (104---). Add Immediate Operand to an Index Register. The~~ immediate operand is added in signed integer form to the contents of the index register specified by the "x" field of the instruction. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, and Condition Code is set on the result.

DECIMAL ARITHMETIC STORE AND LOAD (Standard Instructions)

NOTE

For binary coding details of the following two instructions, refer to Table A-8 in Appendix A.

QST (00023-). Quadruple Store. The quadrupleword operand contained in EDCB is stored in the effective memory location indicated by A plus 4 times the index value. No indexing occurs for coding 000230. For code 000231, 000232, or 000233, indexing for the effective address uses register R[5], R[6], or R[7], respectively. The quadrupleword operand and A are then deleted from the stack.

QLD (00023-). Quadruple Load. The quadrupleword operand contained in the effective memory location indicated by A plus 4 times the index value is fetched. A is deleted, and the fetched quadrupleword is pushed onto the stack. No indexing occurs for coding 000234. For code 000235, 000236, or 000237, indexing for the effective address uses register R[5], R[6], or R[7], respectively. Condition Code is set on the loaded quadrupleword.

DECIMAL INTEGER ARITHMETIC (Standard and Optional Instructions)

QADD (000240). Quadruple Add. The two quadrupleword integers contained in HGFE and DCBA are added in quadrupleword integer form. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is a standard instruction.)

QSUB (000241). Quadruple Subtract. The quadrupleword integer contained in DCBA is subtracted in quadruple-length integer form from the quadrupleword integer in HGFE. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is a standard instruction.)

QMPY (000242). Quadruple Multiply. The quadrupleword integer contained in HGFE is multiplied in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Carry can be set, and Condition Code is set on the result. (This is an optional instruction.)

QDIV (000243). Quadruple Divide. The quadrupleword integer contained in HGFE is divided in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the divisor (DCBA) is zero. Condition Code is set. (This is an optional instruction.)

QNEG (000244). Quadruple Negate. The quadrupleword integer contained is DCBA is replaced with its two's complement. Overflow is set if the original operand was -(2**63). Condition Code is set on the result. (This is an optional instruction.)

QCMP (000245). Quadruple Compare. The Condition Code in the Environment Register is set according to the quadruple integer comparison of HGFE (operand 1) and DCBA (operand 2). (See Table A-3 for Condition Code settings; the "a" states apply for compares.) Both operands are then deleted from the stack. (This is an optional instruction.)

DECIMAL ARITHMETIC SCALING AND ROUNDING (Standard and Optional Instructions)

NOTE

For binary coding details of the following three instructions, refer to Table A-8 in Appendix A.

QUP (00025-). Quadruple Scale Up. The operand value in DCBA is multiplied by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result. (This is a standard instruction.)

QDWN (00025-). Quadruple Scale Down. The operand value in DCBA is divided by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. (This is a standard instruction.)

QRND (000263). Quadruple Round. Five is added to the operand in DCBA if the operand is positive (-5 is added if negative), and the result is divided by 10. The new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. (This is an optional instruction.)

DECIMAL ARITHMETIC CONVERSIONS (Optional Instructions)

CQI (000264). Convert Quad to Integer. The four-word value in DCBA is converted to an integer by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 32767 or less than -32768.

CQL (000246). Convert Quad to Logical. The four-word value in DCBA is converted to a logical value by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 65535.

CQD (000247). Convert Quad to Double. The four-word value in DCBA is converted to a doubleword by extracting the least significant two words. DCBA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the operand was greater than (2**31)-1 or less than -(2**31).

CQA (000260). Convert Quad to ASCII. The binary-coded quadrupleword integer in FEDC is converted to a string of ASCII-coded digits (decimal base), and stores them in the memory space defined by a starting byte address in B and a byte count in A. If the conversion results in a truncation of leading digits, overflow is set. Condition Code is set on the original value.

CIQ (000266). Convert Integer to Quad. The singleword integer in A is extended to a quadrupleword quantity, filling the most significant three words with zeros if A was positive, or ones if A was negative. A is deleted, and the quadrupleword result is pushed onto the stack.

CLQ (000267). Convert Logical to Quad. The singleword logical quantity in A is extended to a quadrupleword quantity, filling the most significant three words with zeros. A is deleted, and the quadrupleword result is pushed onto the stack.

CDQ (000265). Convert Double to Quad. The doubleword integer in BA is extended to a quadrupleword quantity, filling the most significant two words with zeros if B is positive, or ones if B is negative. BA is deleted, and the quadrupleword result is pushed onto the stack.

CAQ (000262). Convert ASCII to Quad. A string of ASCII-coded digits in memory, defined by a starting byte address in B and a byte count in A, is converted to a binary-coded quadrupleword integer. A and B are deleted, and the quadrupleword result is pushed onto the stack. If a nondigit ASCII code is encountered, only the preceding digits are converted, and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). If overflow is set, the value in DCBA is undefined.

CAQV (000261). Convert ASCII to Quad with Initial Value. A string of ASCII-coded digits in memory, defined by a starting byte address in F and a byte count in E, is converted to a binary-coded quadrupleword integer in DCBA. DCBA contains an initial value (greater than or equal to zero) which is scaled by 10, providing a high-order value to which the converted value is added to produce the result in DCBA. If a nondigit ASCII code is encountered, only the preceding digits are converted, and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than (2**63)-1 or less than -(2**63). If overflow is set, the value in DCBA is undefined.

FLOATING-POINT ARITHMETIC (Optional Instructions)

NOTE

For the range of floating-point numbers, refer to "Number Representation" in section 2.

FADD (000270). Floating-Point Add. The floating-point quantities in DC and BA are added in floating-point form. Both operands are deleted, and the two-word result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FSUB (000271). Floating-Point Subtract. The floating-point quantity in BA is negated, and then DC and BA are added in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FMPY (000272). Floating-Point Multiply. The floating-point quantities in DC and BA are multiplied in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FDIV (000273). Floating-Point Divide. The floating-point quantity in DC is divided in floating-point form by the floating-point quantity in BA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

FNEG (000274). Floating-Point Negate. The floating-point quantity in BA (if not zero) is negated. The sign of BA is reversed from positive to negative or negative to positive, and the Condition Code reflects the final state of the sign (see Table A-3).

FCMP (000275). Floating-Point Compare. The Condition Code is set according to the comparison of DC (operand 1) with BA (operand 2). (See Table A-3 for Condition Code settings; the "a" states apply for comparisons.) Both operands are then deleted from the stack.

EXTENDED FLOATING-POINT ARITHMETIC (Optional Instructions)

NOTE

For the range of extended floating-point numbers, refer to "Number Representation" in section 2.

EADD (000300). Extended Add. The extended floating-point quantities in HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

ESUB (000301). Extended Subtract. The extended floating-point quantity in HGFE is negated, and then HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EMPY (000302). Extended Multiply. The extended floating-point quantities in HGFE and DCBA are multiplied in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EDIV (000303). Extended Divide. The extended floating-point quantity in HGFE is divided in extended floating-point form by the extended floating-point quantity in DCBA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

ENEG (000304). Extended Negate. The extended floating-point quantity in DCBA (if not zero) is negated. The sign of DCBA is reversed from positive to negative or negative to positive. Overflow is cleared, and the Condition Code reflects the final state of the sign.

ECMP (000305). Extended Compare. The Condition Code is set according to the comparison of HGFE (operand 1) with DCBA (operand 2). Both operands are then deleted from the stack.

FLOATING-POINT CONVERSIONS (Optional Instructions)

CEF (000276). Convert Extended to Floating. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. DCBA is deleted, and the two-word result is pushed onto the stack.

CEFR (000277). Convert Extended to Floating, Rounded. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. The new quantity is rounded according to the contents of truncated bit 7 of C. DCBA is deleted, and the two-word result is pushed onto the stack.

CFI (000311). Convert Floating to Integer. The floating-point quantity in BA is converted to a singleword signed integer. A is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFIR (000310). Convert Floating to Integer, Rounded. The floating-point quantity in BA is converted to a singleword signed integer, with rounding according to the contents of the most significant fractional bit. A is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFD (000312). Convert Floating to Double. The floating-point quantity in BA is converted to a doubleword signed integer in BA. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CFDR (000313). Convert Floating to Double, Rounded. The floating-point quantity in BA is converted to a doubleword signed integer in BA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CED (000314). Convert Extended to Double. The extended floating-point quantity in DCBA is converted to a doubleword signed integer. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CEDR (000315). Convert Extended to Double, Rounded. The extended floating-point quantity in DCBA is converted to a doubleword signed integer, with rounding according to the contents of the most significant fractional bit. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than (2**31)-1 or less than -(2**31). Condition Code is set on the result.

CEI (000337). Convert Extended to Integer. The extended floating-point quantity in DCBA is converted to a singleword signed integer. CBA is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CEIR (000316). Convert Extended to Integer, Rounded. The extended floating-point quantity in DCBA is converted to a singleword signed quantity, with rounding according to the contents of the most significant fractional bit. CBA is deleted, and the singleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFQ (000320). Convert Floating to Quadruple. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CFQR (000321). Convert Floating to Quadruple, Rounded. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CEQ (000322). Convert Extended to Quadruple. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CEQR (000323). Convert Extended to Quadruple, Rounded. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than (2**63)-1 or less than -(2**63). Condition Code is set on the result.

CFE (000325). Convert Floating to Extended. The floating-point quantity in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.

CIF (000331). Convert Integer to Floating. The signed integer in A is converted to a floating-point quantity. A is deleted, and the two-word result is pushed onto the stack.

CDF (000306). Convert Double to Floating. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with truncation if the result exceeds 23 significant bits.

CDFR (000326). Convert Double to Floating, Rounded. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with rounding if the result exceeds 23 significant bits.

CQF (000324). Convert Quadruple to Floating. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with truncation if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CQFR (000330). Convert Quadruple to Floating, Rounded. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with rounding if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CIE (000332). Convert Integer to Extended. The signed integer in A is converted to an extended floating-point quantity. A is deleted, and the four-word result is pushed onto the stack.

CDE (000334). Convert Double to Extended. The doubleword signed integer in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.

CQE (000336). Convert Quadruple to Extended. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with truncation if the result exceeds 55 significant bits.

CQER (000335). Convert Quadruple to Extended, Rounded. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with rounding if the result exceeds 55 significant bits.

FLOATING-POINT FUNCTIONALS (Optional Instructions)

IDX1 (000344). Calculate Index, 1 Dimension. For a one-dimensional array, IDX1 compares the subscript value in B against lower and upper bounds in a two-word table in the current code segment starting at the address specified in A. If the value is in bounds, the element offset value is computed and is stored in register R[7]. If the subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. BA is then deleted.

IDX2 (000345). Calculate Index, 2 Dimensions. For a two-dimensional array, IDX2 compares the subscript values in B and C against lower and upper bounds in a 4-word table in the current code segment starting at the address in A. If the values are in bounds, the element offset value is computed and stored in register R[7]. If a subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. CBA is then deleted.

IDX3 (000346). Calculate Index, 3 Dimensions. For a three-dimensional array, IDX3 compares the subscript values in B, C, and D against lower and upper bounds in a 6-word table in the current code segment starting at the address in A. If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. DCBA is then deleted.

IDXP (000347). Calculate Index, Code Space. For an n-dimensional array, IDXP compares the subscript values in n stack registers (B, C, D, etc.) against lower and upper bounds in a table in the current code segment (2n words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions.) If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. All stack data used is deleted.

IDXD (000317). Calculate Index, Data Space. For an n-dimensional array, IDXD compares the subscript values in n stack registers (B, C, D, etc.) against lower and upper bounds in a table in the current data segment (2n words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions.) If the values are in bounds, the element offset value is computed and stored in register R[7]. If any subscript is out of bounds, overflow is set, R[7] receives the erroneous subscript, and CCL indicates too low or CCG indicates too high. All stack data used is deleted.

REGISTER STACK MANIPULATION

EXCH (000004). Exchange A and B. A and B of the Register Stack are interchanged. Condition Code is set on the result in A.

DXCH (000005). Double Exchange BA with DC. The doubleword contained in DC is interchanged with the doubleword contained in BA. Condition Code is set on the result in BA.

DDUP (000006). Double Duplicate BA in DC. The doubleword in the top two registers of the stack is duplicated by pushing a copy of it onto the Register Stack. Condition Code is set.

NOTE

For binary coding details of the following three instructions (STAR, NSAR, LDRA), refer to Table A-7 in Appendix A.

STAR (00011-). Store A in a Register. The A Register contents are stored in the register pointed to by the Register field of the instruction. A is then deleted from the stack.

NSAR (00012-). Non-destructive Store A into a Register. The A Register is stored in the register pointed to by the Register field of the instruction.

LDRA (00013-). Load A from a Register. The contents of the register pointed to by the Register field of the instruction are pushed onto the stack. Condition Code is set.

NOTE

For binary coding details of the following three instructions (LDI, LDXI, LDLI), refer to Table A-4 in Appendix A.

LDI (100---). Load Immediate Operand into A. The immediate operand is pushed onto the stack, with the sign bit propagating into the high-order bits. Condition Code is set.

LDXI (10----). Load Index Register with Immediate Operand. The index register specified by the "x" field of the instruction is loaded with the immediate operand, and the sign bit propagates into the high-order bits. Condition Code is set.

LDLI (005---). Load Left Immediate Operand into bits 0:7 of A. The immediate operand, shifted left eight places, is loaded into A, with the sign bits propagating into the low-order bits of A. Condition Code is set.

BOOLEAN OPERATIONS

Figure 3-2 illustrates the fundamental principles of boolean operations as performed by four of the instructions. Figure 3-3 shows the equivalent operations as performed on immediate operands.

LAND (000010). Logical AND A with B. A and B are logically ANDed. The two words are deleted from the stack and the result pushed on. Condition Code is set.

LOR (000011). Logical OR A with B. A and B are merged by a logical inclusive OR. A and B are deleted and the result pushed onto the stack. Condition Code is set.

XOR (000012). Logical Exclusive OR A with B. The two words in A and B of the Register Stack are combined by a logical exclusive OR. The two words are then deleted and the result is pushed onto the stack. Condition Code is set.

NOT (000013). One's Complement A. The word contained in Register A of the stack is converted to its one's complement. Condition Code is set.

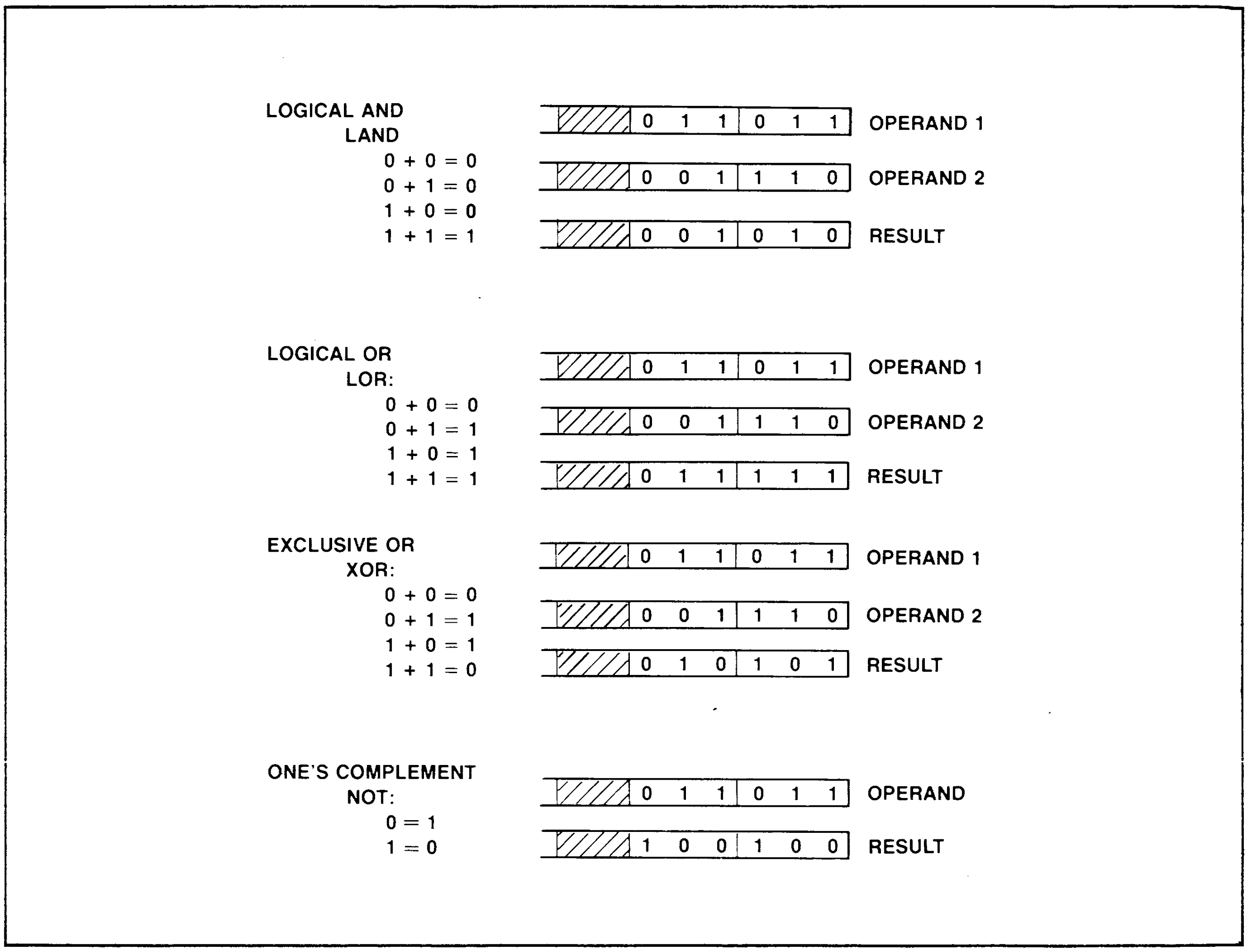


Figure 3-2. Boolean Operations

NOTE

For binary coding details of the following four instructions (ORRI, ORLI, ANRI, ANLI), refer to Table A-4 in Appendix A.

ORRI (004---). OR Right Immediate Operand with A. The 8-bit immediate operand is merged with the A Register by a logical inclusive OR. The sign bit is not propagated, but is actually part of the instruction; see Figure 3-3. Condition Code is set.

ORLI (004---). OR Left Immediate Operand with A. The 8-bit immediate operand is shifted left eight places and merged with A by a logical inclusive OR. The sign bit is not propagated, but is actually part of the instruction; see Figure 3-3. Condition Code is set.

ANRI (006---). AND Right Immediate Operand to A. The 8-bit immediate operand is extended to 16 bits by propagating the sign into the high-order bits, and the resulting integer is logically ANDed to A; see Figure 3-3. Condition Code is set.

ANLI (007---). AND Left Immediate Operand with A. The 8-bit immediate operand is shifted left eight places, the sign bit is propagated into the low-order bits, and the resulting integer is logically ANDed to A; see Figure 3-3. Condition Code is set.

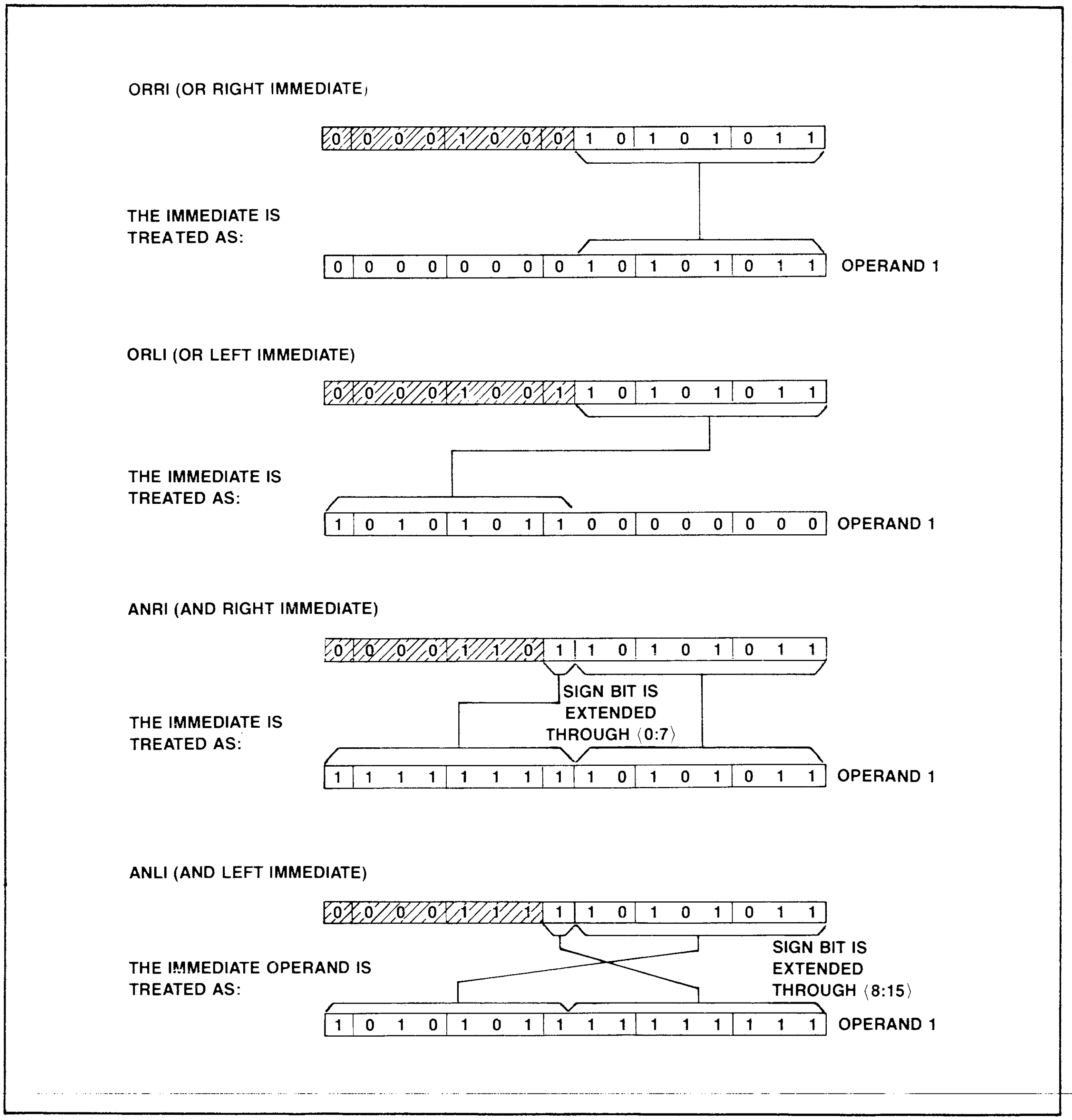


Figure 3-3. Boolean Instructions with Immediate Operands

BIT DEPOSIT AND SHIFT

DPF (000014). Deposit Field in A. This instruction combines the words contained in registers A and C of the stack as a function of a mask word contained in register B of the stack. A logical OR operation is performed on the logical AND of B and C and the logical AND of not B and A, so that all bits in C corresponding to ones in B are deposited into corresponding bits in A. The original three words are deleted from the stack and the result pushed onto the stack. Condition Code is set. An example of this operation is shown in Figure 3-4.

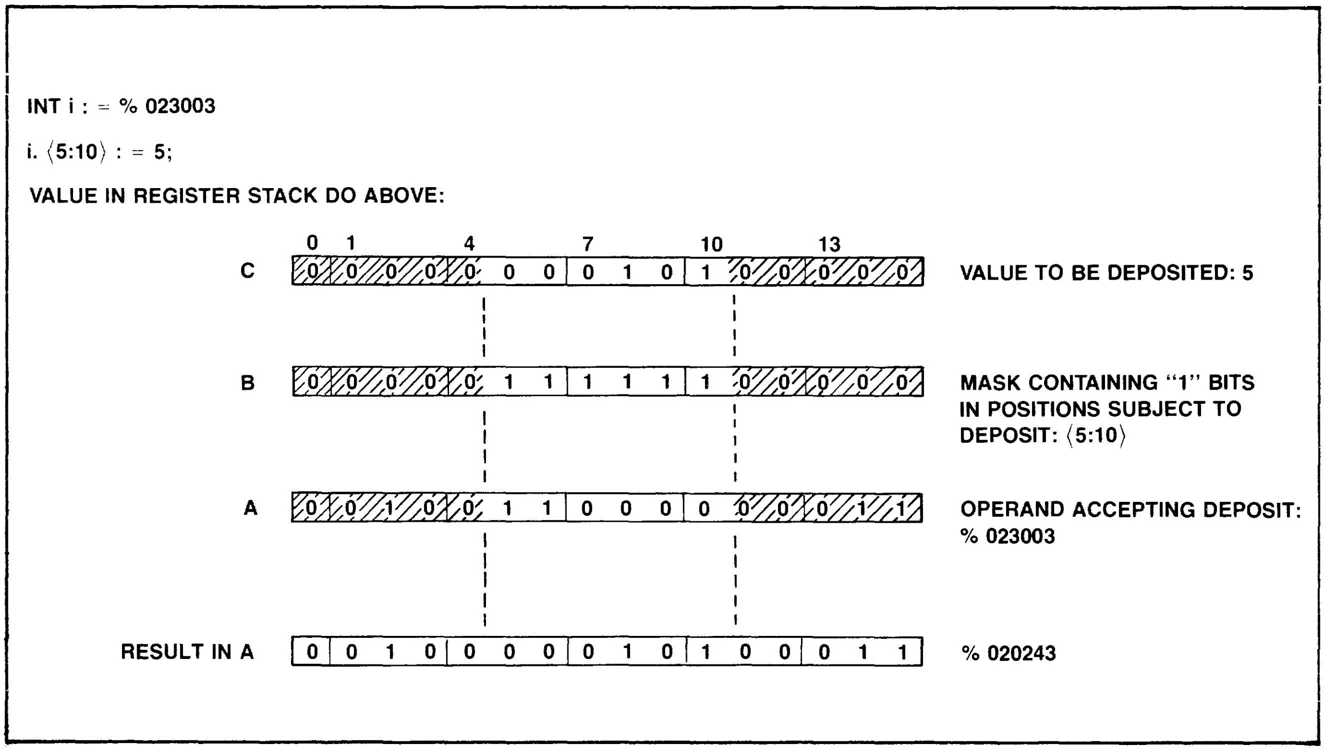


Figure 3-4. Deposit Field Example

LLS (0300--). Logical (unsigned) Left Shift. If the Shift Count field is zero, the word contained in B is shifted left by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted left by that number. Condition Code is set. Figure 3-5 presents a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

DLLS (1300--). Double Logical (unsigned) Left Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted left by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted left by that number. Condition Code is set.

LRS (0301--). Logical (unsigned) Right Shift. If the Shift Count field is zero, the word contained in B is shifted right by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted right by that number. Condition Code is set.

DLRS (1301--). Double Logical (unsigned) Right Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted right by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted right by that number. Condition Code is set.

ALS (0302--). Arithmetic (signed) Left Shift. If the Shift Count field is zero, the word contained in B is shifted left preserving the sign bit by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted left, preserving the sign bit, by that number. Condition Code is set.

DALS (1302--). Double Arithmetic (signed) Left Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted left, preserving the sign bit, by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted left, preserving the sign bit, by that number. Condition Code is set.

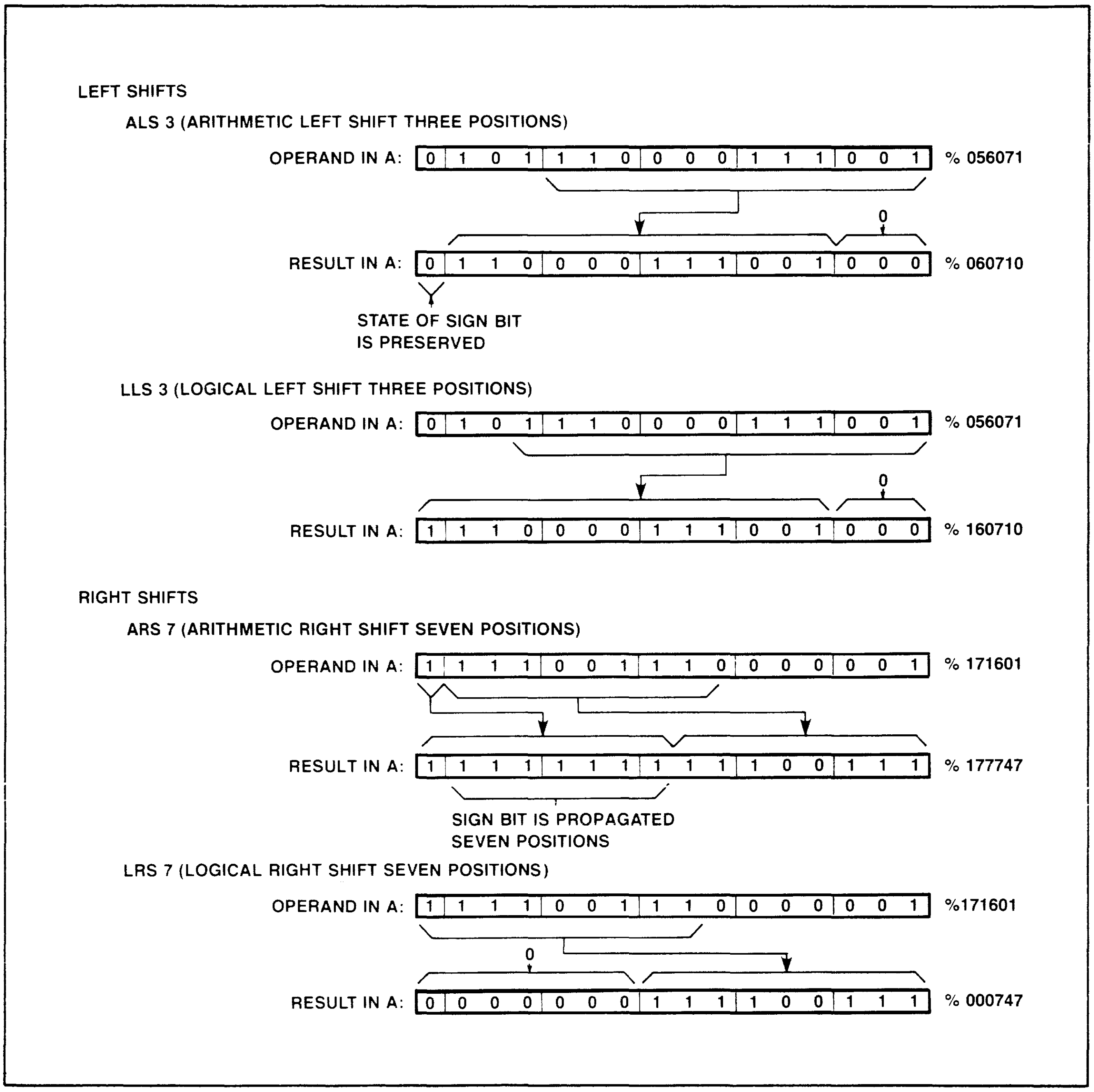

LEFT SHIFTS
ALS 3 (ARITHMETIC LEFT SHIFT THREE POSITIONS)
OPERAND IN A: 0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1
% 056071
0
RESULT IN A: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 0 0
% 060710
STATE OF SIGN BIT IS PRESERVED
LLS 3 (LOGICAL LEFT SHIFT THREE POSITIONS)
OPERAND IN A: 0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1
% 056071
0
RESULT IN A: 1 1 1 0 0 0 0 1 1 1 0 0 1 0 0 0
% 160710
RIGHT SHIFTS
ARS 7 (ARITHMETIC RIGHT SHIFT SEVEN POSITIONS)
OPERAND IN A: 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 1
% 171601
RESULT IN A: 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1
% 177747
SIGN BIT IS PROPAGATED SEVEN POSITIONS
LRS 7 (LOGICAL RIGHT SHIFT SEVEN POSITIONS)
OPERAND IN A: 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 1
%171601
0
RESULT IN A: 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1
% 000747

Figure 3-5. Arithmetic vs. Logical Shifts

ARS (0303--). Arithmetic (signed) Right Shift. If the Shift Count field is zero, the word contained in B is shifted right, propagating the sign bit, by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, A is shifted right, propagating the sign bit, by that number. Condition Code is set.

DARS (1303--). Double Arithmetic (signed) Right Shift. If the Shift Count field is zero, the doubleword contained in CB is shifted right, propagating the sign bit, by the count (modulo %377) contained in A. A is then deleted from the stack. However, if Shift Count is not zero, BA is shifted right, propagating the sign bit, by that number. Condition Code is set.

BYTE TEST

BTST (000007). Byte Test A. The Condition Code is set on the value of the test byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. A is deleted after the test.

MEMORY TO/FROM REGISTER STACK

NOTE

For binary coding details of the first twelve instructions below (LWP through ADM), refer to Table A-3 in Appendix A.

LWP (-2----). Load Word from Program (Code) Area into A. The contents of the address which is computed as a function of displacement (a signed 8-bit value), and optionally indexing and/or indirection, is pushed onto the Register Stack. Condition Code is set on the loaded word. Figure 3-6 illustrates the addressing operations for the LWP instruction.

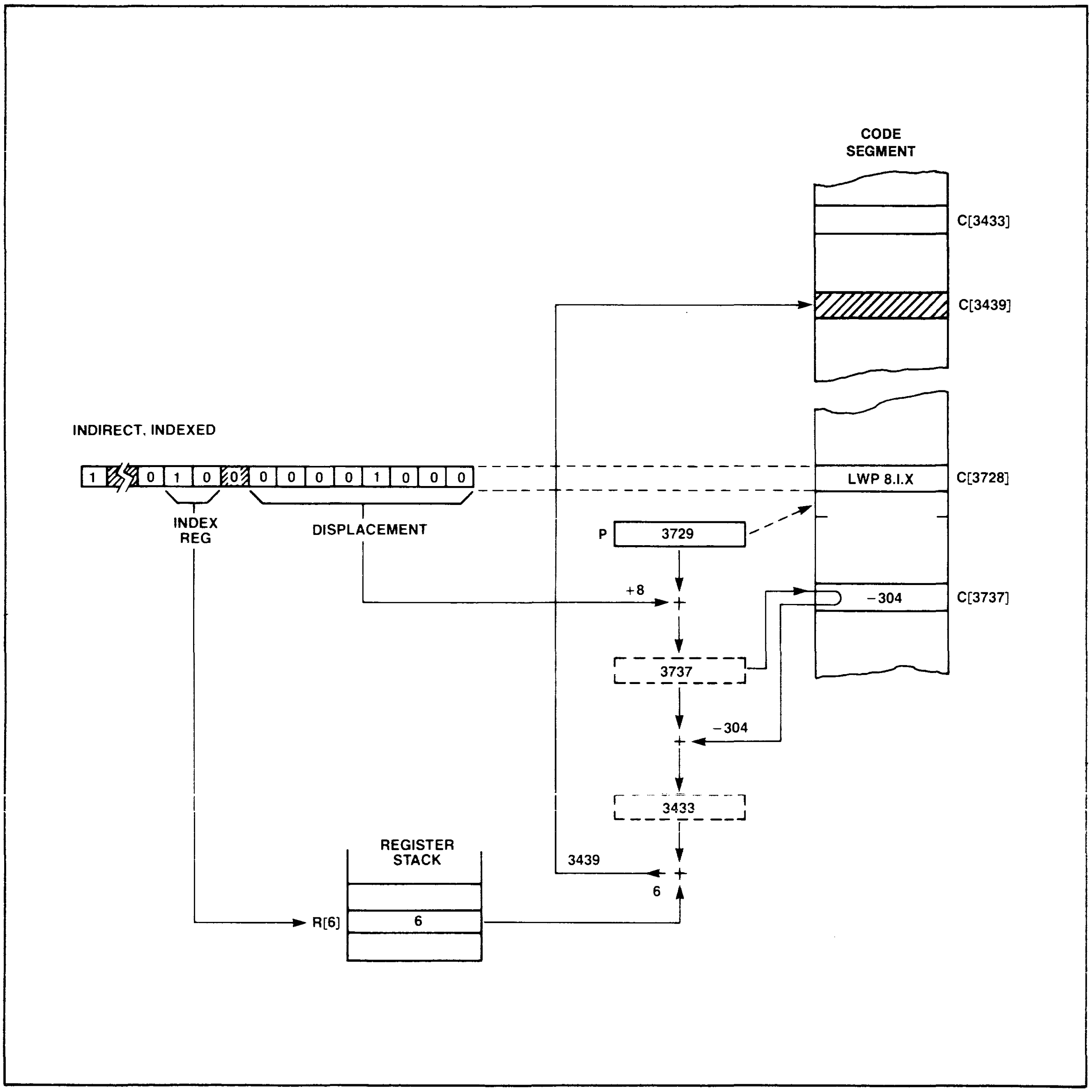


Figure 3-6. LWP Instruction Addressing

LBP (-2-4--). Load Byte from Program (Code) Area into A. The contents of the P-relative byte address which is computed as a function of displacement (a signed 8-bit value), and optionally indexing and/or indirection, is pushed onto the Register Stack. The high-order byte is set to zero. If the P Register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the address will always be relative to whichever half of the segment P currently indicates. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character. Figure 3-7 illustrates the addressing operations for the LBP instruction, assuming addresses in the first half of the code segment.

LDX (-3----). Load Index Register from Data Space. The index register specified by the "x" field of the instruction is loaded with the contents of the effective memory address. Condition Code is set. Figure 3-8 shows the instruction word format for memory data reference instructions, such as LDX.

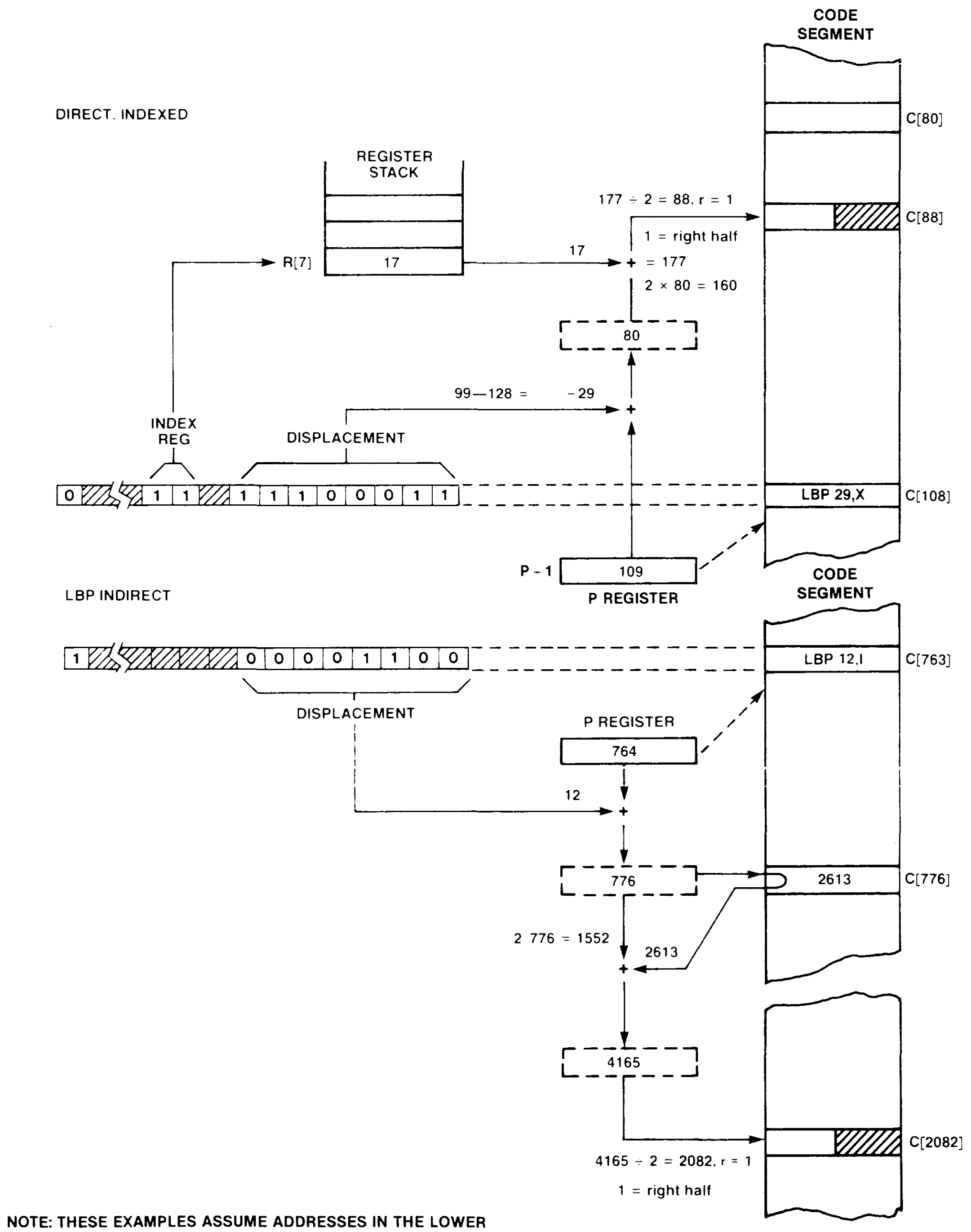
NSTO (-34---). Nondestructive Store from A. The contents of the A Register are stored into effective address memory location. The Register Stack is not modified.

LOAD (-40---). Load A from Data Space. The contents of the effective address memory location are pushed onto the stack. Condition Code is set.

STOR (-44---). Store A into Data Space. The contents of the A Register are stored into the effective memory location. A is then deleted from the stack.

LDB (-5----). Load A with Byte from Data Space. The contents of the effective memory location are loaded into bits 8:15 of A. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character.

STB (-54---). Store Byte from A to Data Space. The contents of the byte in bits 8:15 of A are stored in the effective memory location.



NOTE: THESE EXAMPLES ASSUME ADDRESSES IN THE LOWER HALF OF THE CODE SEGMENT, i.e., P.<0> = 0. IF P.<0> = 1, %100000 IS ADDED TO THE COMPUTED ADDRESS BEFORE THE BYTE IS FETCHED FROM MEMORY.

Figure 3-7. LBP Instruction Addressing

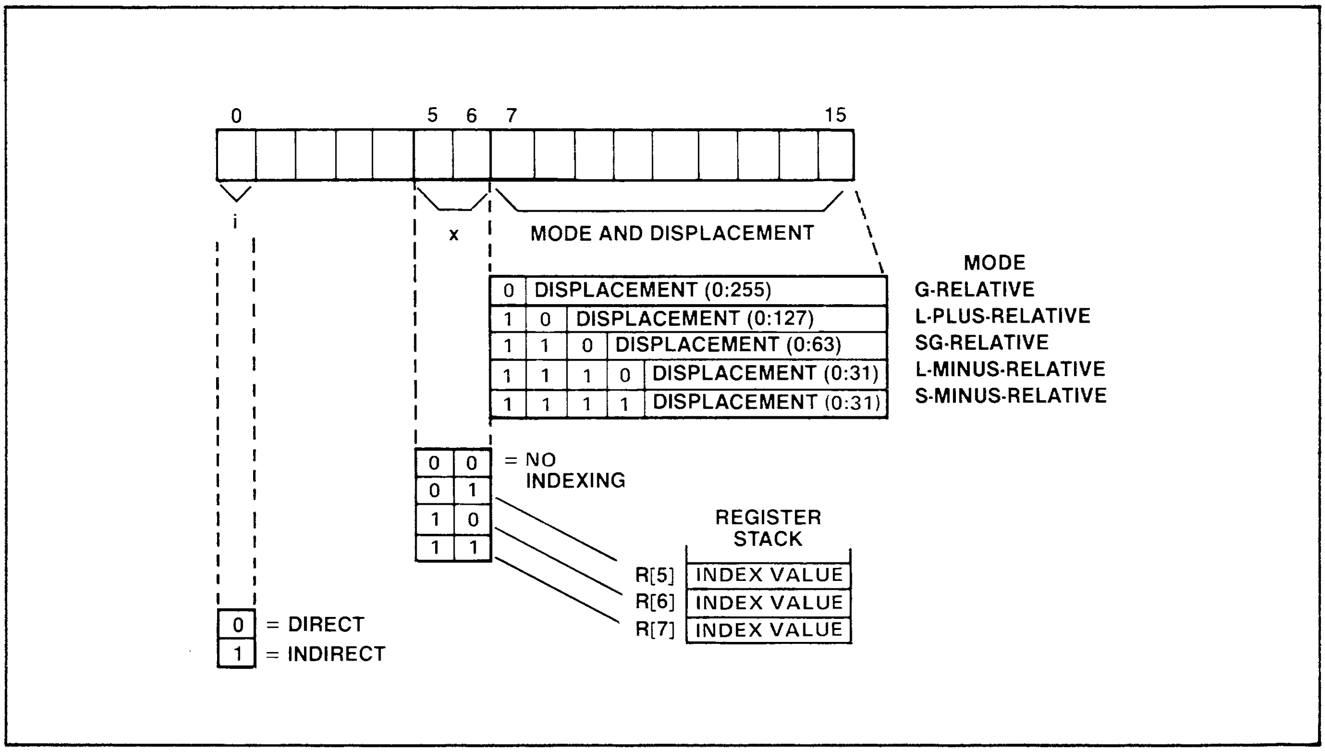


Figure 3-8. Memory Reference Instruction Format

Memory to/from Register Stack

LDD (-6----). Load Double from Data Space into BA. The doubleword integer contained in the effective memory location is pushed into the stack. Condition Code is set. Figure 3-9 illustrates the addressing methods for doubleword instructions.

STD (-64---). Store Double from BA into Data Space. The contents of BA are stored in the effective memory location. BA is deleted.

LADR (-7----). Load G-Relative Address of Variable into A. The G-relative address of the variable is pushed onto the stack.

ADM (-74---). Add A to Variable in Data Space. The A Register is added in integer form to the contents of the effective memory location and the Condition Code is set on the sum. Overflow is set if the result is greater than 32767 or less than -32768. Carry can also be set. A is then deleted from the stack.

NOTE

For binary coding details of the following six instructions (PUSH through SBXX), refer to Table A-5 in Appendix A.

PUSH (024nrc). Push Registers to Data Space. This instruction transfers the contents of a specified number of elements in the Register Stack to the top of the data stack in memory. The "n" field of the instruction is the value to which RP will be set following the instruction; the "r" field specifies the last register stack element to be pushed; the "c" field is the number of registers minus one that will be pushed to memory. Following the PUSH instruction, the S Register points to the last element pushed onto the memory stack. If the resultant value of S is greater than %77777, a stack overflow trap occurs. Figure 3-10 illustrates the bit fields and the action of the PUSH instruction.

POP (124nrc). Pop Data Space to Registers. This instruction loads the Register Stack with the top elements of the data stack (as indicated by the current S Register setting). The "n" field of the instruction indicates the value RP will have following the instruction; the "r" field specifies the last Register Stack element to be loaded from memory; the "c" field specifies the number of registers minus one that will be loaded. If the resultant value of S is greater than %77777, a stack overflow trap occurs. Figure 3-10 illustrates the bit fields and the action of the POP instruction.

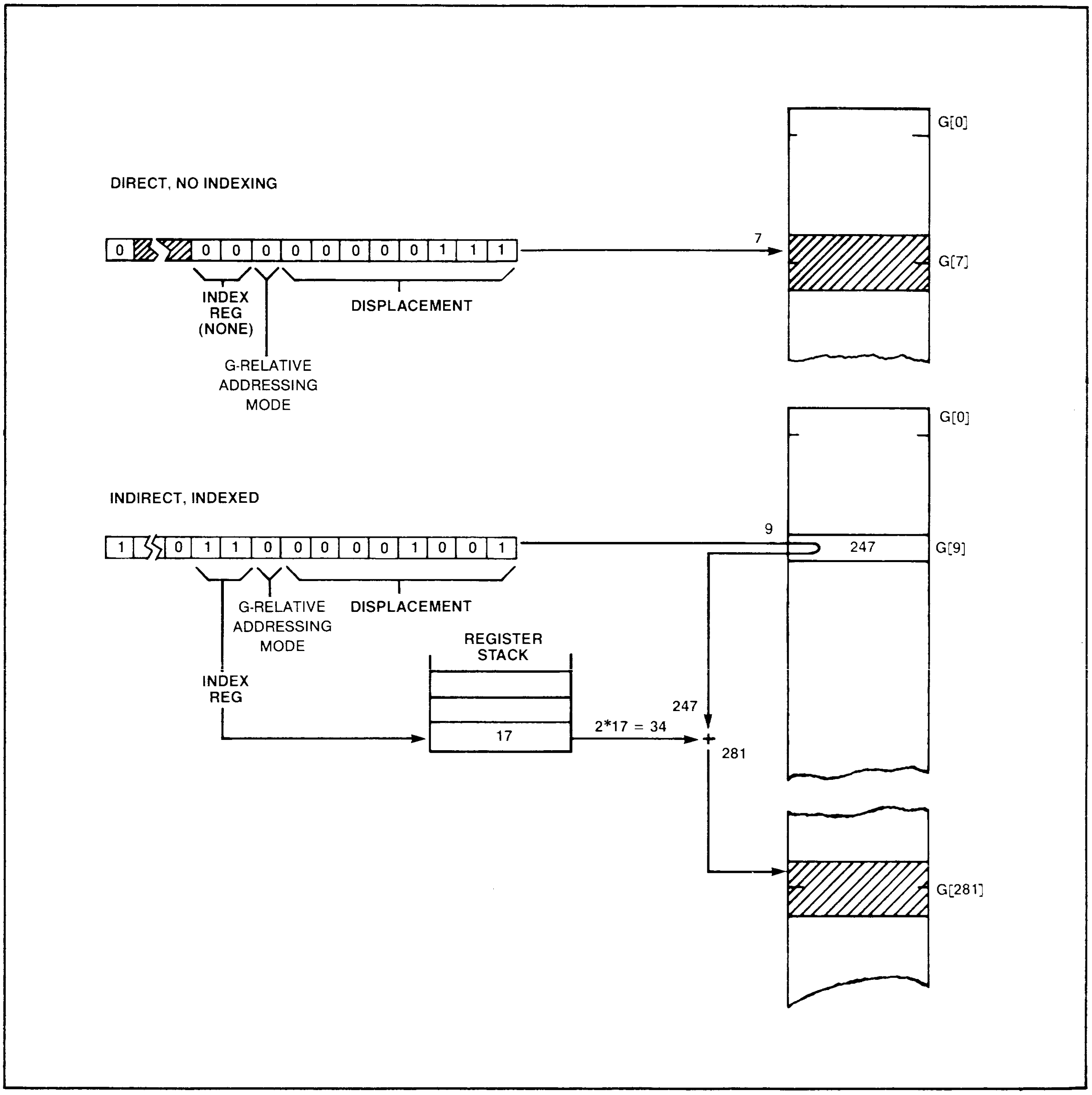


Figure 3-9. Doubleword Addressing

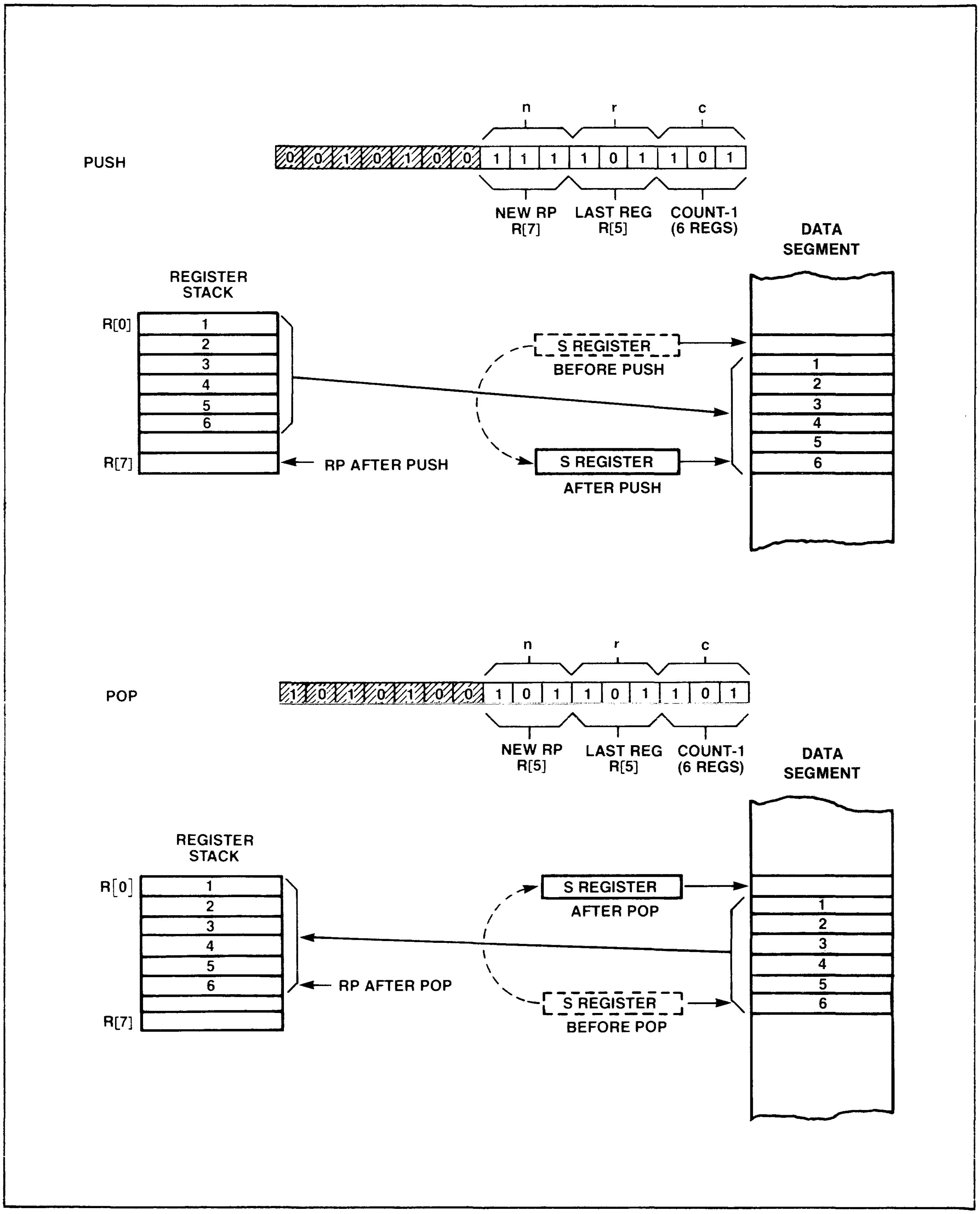


Figure 3-10. PUSH and POP Instructions

LWXX (0254--, 0264--). Load Word Extended, Indexed. The word contained in a computed extended memory location is loaded onto the stack, replacing the prior contents of A. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either the current L Register value (coded 0254--) or G[0] (coded 0264--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is shifted left one bit position (multiplication by 2, since this instruction requires word addressing rather than byte addressing) and is then added to the extended memory pointer to address the word that is to be loaded. Condition Code is set.

SWXX (0255--, 0265--). Store Word Extended, Indexed. The word contained in B is stored into a computed extended memory location. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either the current L Register value (coded 0255--) or G[0] (coded 0265--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is shifted left one bit position (multiplication by 2, since this instruction requires word addressing rather than byte addressing) and is then added to the extended memory pointer to address the location that is to receive the word being stored.

LBXX (0256--, 0266--). Load Byte Extended, Indexed. The byte contained in a computed extended memory location is loaded onto the stack, replacing the prior contents of A. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either the current L Register value (coded 0256--) or G[0] (coded 0266--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the byte that is to be loaded. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBXX (0257--, 0267--). Store Byte Extended, Indexed. The byte contained B.<8:15> is stored into a computed extended memory location. The extended memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value which is either the current L Register value (coded 0257--) or G[0] (coded 0267--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the location that is to receive the byte being stored.

LOAD AND STORE VIA ADDRESS ON REGISTER STACK

ANS (000034). AND to SG Memory. The word in B is logically ANDed to a word in the System Data segment that is specified by a 16-bit address in A. The result remains in the System Data location, and A and B are deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV Register. Condition Code is set.

ORS (000035). OR to SG Memory. The word in B is logically ORed to a word in the System Data segment that is specified by a 16-bit address in A. The result remains in the System Data location, and A and B are deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV Register. Condition Code is set.

ANG (000044). AND to Memory. The word in B is logically ANDed to a word in the current data segment that is specified by a 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ORG (000045). OR to Memory. The word in B is logically ORed to a word in the current data segment that is specified by a 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ANX (000046). AND to Extended Memory. The word in C is logically ANDed to a word in extended memory that is specified by a 32-bit address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

ORX (000047). OR to Extended Memory. The word in C is logically ORed to a word in extended memory that is specified by a 32-bit address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

LWUC (000342). Load Word from User Code Space. A word in the user code segment, specified by a 16-bit address in A, is loaded onto the stack, replacing the prior contents of A. Condition Code is set.

LWAS (000350). Load Word via A from System. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (system or user) is determined by the DS bit (bit 6) of the ENV Register. Condition Code is set.

LWA (000360). Load Word via A. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. LWA accesses the current data segment only. Condition Code is set.

SWAS (000351). Store Word via A into System. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV Register.

SWA (000361). Store Word via A. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. SWA accesses the current data segment only.

LDAS (000352). Load Double via A from System. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV Register. Condition Code is set.

LDA (000362). Load Double via A. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). LDA accesses the current data segment only. Condition Code is set.

Load and Store via Address on Register Stack

SDAS (000353). Store Double via A into System. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. CBA is then deleted. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV Register.

SDA (000363). Store Double via A. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. CBA is then deleted. SDA accesses the current data segment only.

LBAS (000354). Load Byte via A from System. The byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, replacing the prior contents of A. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV Register. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

LBA (000364). Load Byte via A. The byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, replacing the prior contents of A. LBA accesses the current data segment only. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBAS (000355). Store Byte via A into System. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. If privileged mode is in effect when this instruction is executed, A refers to an address in the System Data segment. Otherwise data segment selection (user or system) is determined by the DS bit (bit 6) of the ENV Register.

SBA (000365). Store Byte via A. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. SBA accesses the current data segment only.

DFS (000357). Deposit Field into System Data. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See Figure 3-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) If privileged mode is in effect, the destination is in the System Data segment; otherwise, the destination is in the current data segment. A, B, and C are then deleted. Condition Code is set.

DFG (000367). Deposit Field in Memory. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See Figure 3-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) DFG accesses the current data segment. Condition Code is set.

LBX (000406). Load Byte Extended. The byte in the extended memory location specified by the 32-bit address in registers B and A is loaded onto the Register Stack (bits 8 through 15 of A), after the address in BA is deleted. The left byte is zero. The Condition Code is set on the value of the loaded byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCL indicates special ASCII character.

SBX (000407). Store Byte Extended. The byte in bits 8 through 15 of C is stored into the extended memory location specified by the 32-bit address in registers B and A. C, B, and A are then deleted.

LWX (000410). Load Word Extended. The word in the extended memory location specified by the 32-bit address in registers B and A is loaded into register A (after the address in BA is deleted). Condition Code is set.

SWX (000411). Store Word Extended. The word in register C is stored into the extended memory location specified by the 32-bit address in registers B and A. C, B, and A are then deleted.

Load and Store via Address on Register Stack

LDDX (000412). Load Doubleword Extended. The doubleword starting at the extended memory location specified by the 32-bit address in registers B and A is loaded onto the register stack, replacing the prior contents of B and A. Condition Code is set.

SDDX (000413). Store Doubleword Extended. The doubleword in registers D and C is stored into extended memory starting at the location specified by the 32-bit address in registers B and A. All four words are then deleted from the Register Stack.

LQX (000414). Load Quadrupleword Extended. The quadrupleword starting at the extended memory location specified by the 32-bit address in registers B and A is loaded into registers DCBA of the Register Stack (after the address in BA is deleted). Condition Code is set.

SQX (000415). Store Quadrupleword Extended. The quadrupleword in registers FEDC is stored into extended memory (8 bytes) starting at the location specified by the 32-bit address in registers B and A. All six words are then deleted from the Register Stack.

DFX (000416). Deposit Field Extended. Using the mask bits in register C, this instruction deposits the bits in register D into the extended memory location specified by the 32-bit address in registers B and A. All four words are then deleted from the Register Stack. (See Figure 3-4 and DPF description under "Bit Deposit and Shift" for further details on this operation.) Condition Code is set.

SCS (000444). Set Code Segment. Registers B and A are assumed to contain a 17-bit byte address. This instruction sets a logical segment number into the segment number field (bits 0 through 14 of B) to formulate a complete 32-bit address. Only two values may be set for this field: 3 (indicating current code segment) if either the CS or LS bit of the Environment Register contains a one; 2 (indicating User Code segment) if both of these bits are zero.

LQAS (000445). Load Quadrupleword via A from SG. The quadrupleword contained in the four memory locations starting at the location pointed to by the address in A is loaded into DCBA (after the address in A is deleted). The address in A refers to an address in the System Data segment. Condition Code is set. This is a privileged instruction.

SQAS (000446). Store Quadrupleword via A to SG. The quadrupleword in registers EDCB is stored into the four memory locations starting at the location pointed to by the address in A. The address in A refers to an address in the System Data segment. All five words are then deleted from the Register Stack. This is a privileged instruction.

BRANCHING

NOTE

For binary coding details of the following branch instructions, refer to Table A-6 in Appendix A.

BIC (-100--). Branch if CARRY. If the carry bit (K) in the Environment Register is set (K = 1), then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed. Figure 3-11 compares direct and indirect branching.

BUN (-104--). Branch Unconditionally. A direct or indirect unconditional branch is taken (depending on the "i" field of the instruction).

BOX (-1-4--). Branch on X Less Than A and Increment X. If the index register as specified by the "x" field of the instruction is less than A, that index register is incremented and a direct or indirect branch is taken (depending on the "i" field of the instruction). If X is greater than or equal to A, then A is deleted from the stack and the next instruction is executed.

BGTR (-11---). Branch if CC is Greater. If the Condition Code in the ENV Register is CCG (N = 0, Z = 0) then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

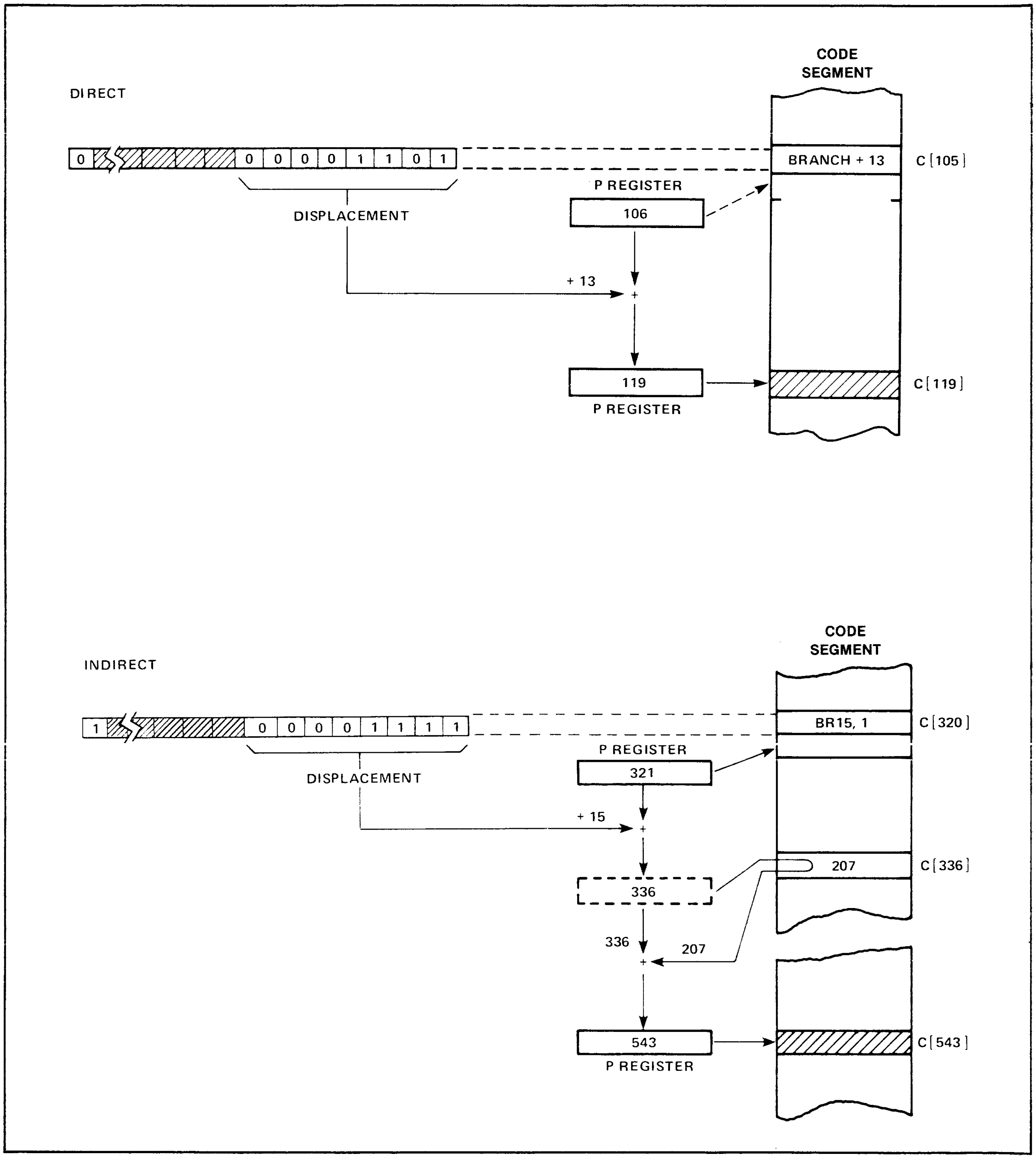


Figure 3-11. Direct vs. Indirect Branching

BEQL (-12---). Branch if CC is Equal. If the Condition Code in the ENV Register is CCE (N = 0, Z = 1), then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BGEQ (-13---). Branch if CC is Greater or Equal. If the Condition Code in the ENV Register is CCG or CCE (N = 0) then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BLSS (-14---). Branch if CC is Less. If the Condition Code in the ENV Register is CCL (N = 1) then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BAZ (-144--). Branch on A Zero. If the A Register equals zero then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the A Register does not equal zero, then the next instruction is executed. In either case, A is deleted from the stack.

BNEQ (-15---). Branch if CC is not equal. If the Condition Code in the ENV Register is not CCE (Z = 0) then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BANZ (-154--). Branch on A Not Zero. If the A Register is non-zero then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the A Register equals zero, then the next instruction is executed. In either case, A is deleted from the stack.

BLEQ (-16---). Branch if CC is Less or Equal. If the Condition Code in the ENV Register is CCL or CCE (N = 1 or Z = 1) then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BNOV (-164--). Branch if no OVERFLOW. If the overflow bit (V) in the ENV Register is not set (V = 0), then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BNOC (-17---). Branch if no CARRY. If the carry bit (K) in the ENV Register is not set (K = 0), then a direct or indirect branch is taken (depending on the "i" field of the instruction). If the condition is not met, then the next instruction is executed.

BFI (000030). Branch Forward Indirect. The instruction expects an offset from the current P register setting to be contained in A. An indirect branch is then made through the location specified by P + A. Figure 3-12 illustrates the action of the BFI instruction.

MOVES, COMPARES, SCANS, AND CHECKSUM COMPUTATIONS

Figure 3-13 provides a comparison of ascending and descending moves, compares, and scans, as described in the following paragraphs. Bit 9 of the instruction word specifies ascending (0) or descending (1). Interrupts can occur between words (or bytes) moved or compared on each of these instructions.

MNGG (000226). Move Words While Not Duplicate. Register D is assumed to contain a destination address in the current data segment, and register C is assumed to contain a source address in the current data segment. The MNGG instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, B contains the source address, and C contains the destination address.

CDG (000366). Count Duplicate Words. Beginning at the address (in the current data segment) specified in register C, and for a maximum count of words specified in register B, this instruction counts the number of duplicate words in the buffer. Register A is incremented on each duplicate found, and may contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and C contains the address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by C and C-1. This instruction is intended to be used in conjunction with MNGG.

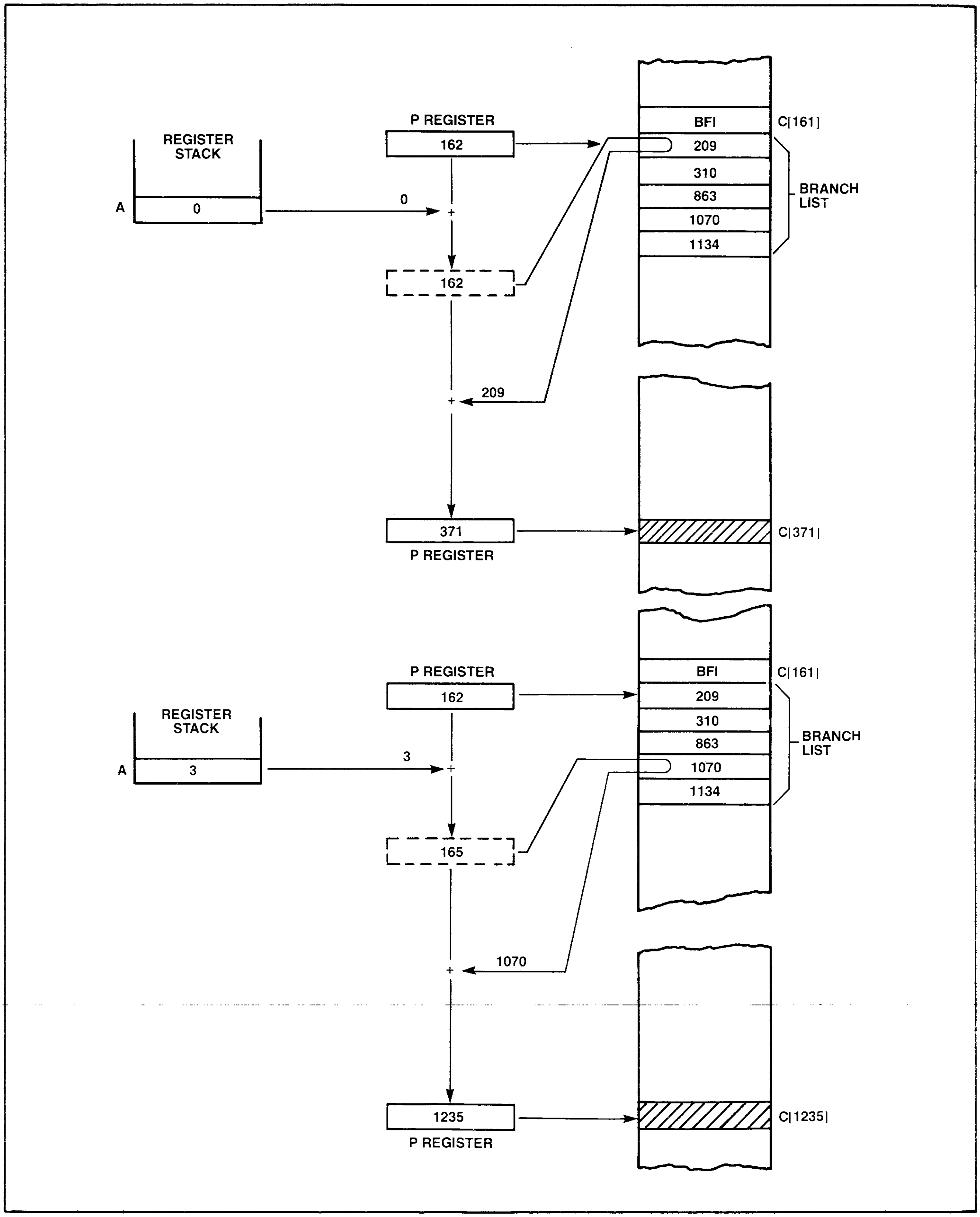


Figure 3-12. Branch Forward Indirect

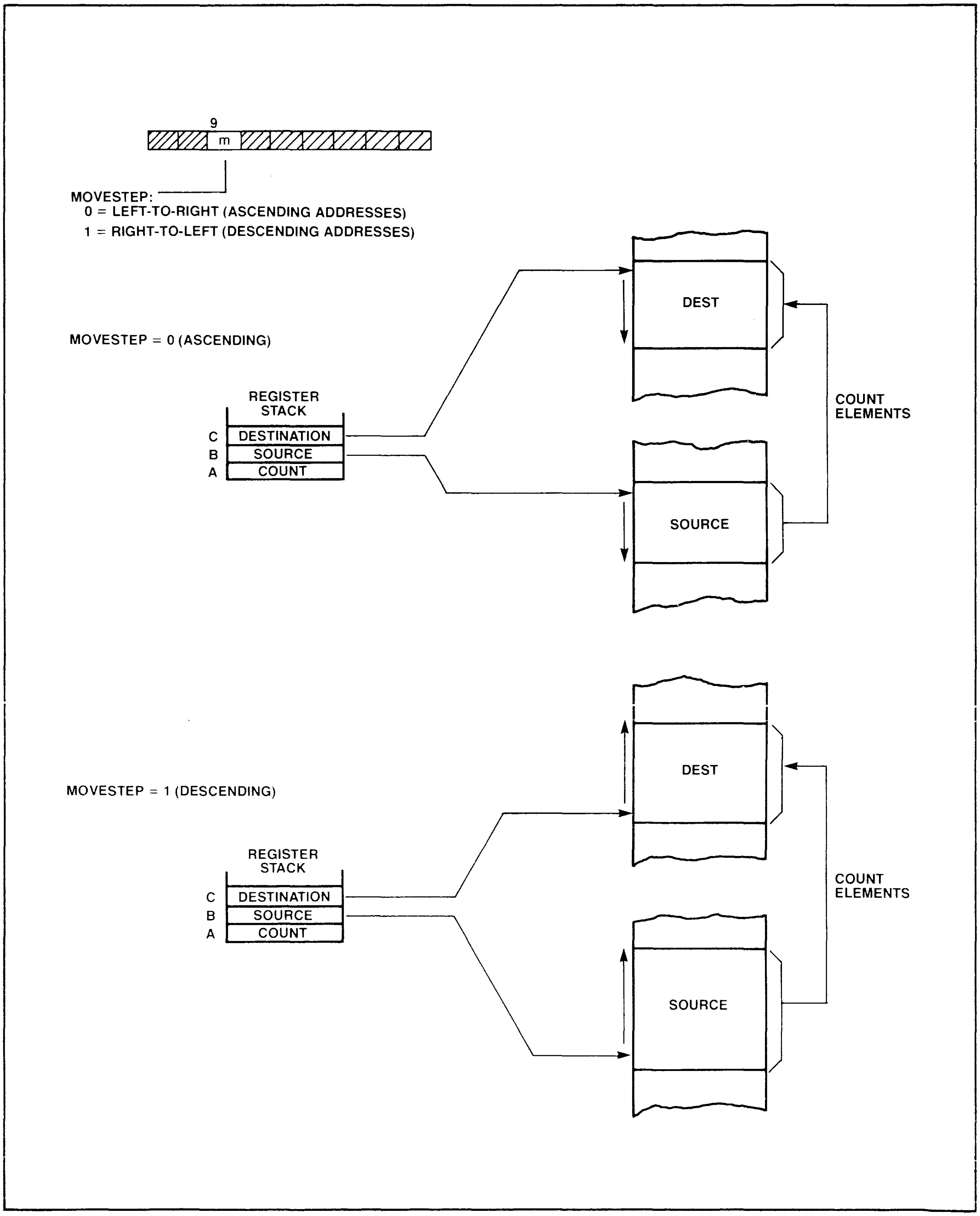


Figure 3-13. Directions for Moves, Compares, and Scans

NOTE

For binary coding details of the following six move instructions (MOVW, MOVB, COMW, COMB, SBW, SBU), refer to Table A-5 in Appendix A.

MOVW (026---). Move Words. This instruction transfers a specified number of words from one area of memory to another. The instruction expects A to contain a word count, B to contain the source word address, and C to contain the destination word address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and Privileged Bits of the ENV Register. The "m" field of the instruction (see format diagram at the top of Figure 3-13) determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each move. The "n" field of the instruction is the value to which RP is set upon instruction end. The move is made one word at a time from the source to the destination. After each word transfer the addresses are decremented or incremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next word is moved. Interrupts can occur after each compare.

MOVB (126---). Move Bytes. This instruction transfers a specified number of bytes from one area of memory to another. The instruction expects A to contain a byte count, B to contain the source byte address, and C to contain the destination byte address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and Privileged Bits of the ENV Register. The "m" field of the instruction determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each move. The "n" field of the instruction is the value to which RP is set upon instruction end. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are decremented or incremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next byte is moved. If the source is a code segment and the P Register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the source and destination addresses will always be ~~relative to whichever half of the segment P currently indicates.~~ Interrupts can occur after each compare.

COMW (0262--). Compare Words. This instruction compares one area of memory with another, a word at a time, until a miscompare occurs or until a specified number of comparisons have been made. The words being compared are treated as unsigned quantities. COMW expects A to contain a word count, B to contain a source word address and C to contain a destination word address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and Privileged bits of the ENV Register. The "m" field determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The "n" field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the address by one according to the "m" field, and decrements the word count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, CC indicates the results of the compare or CCE due to A going to zero. Interrupts can occur after each compare.

COMB (1262--). Compare Bytes. This instruction compares one area of memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. It expects A to contain a byte count, B to contain a source byte address and C to contain a destination byte address. The source and destination maps to be used are specified by the "s" and "d" fields of the instruction and by the DS, CS, LS, and Privileged bits of the ENV Register. If the source address is in a code segment, the byte address is taken to be in the same 64K half of the code space as the current P Register value. The "m" field determines whether the source and destination addresses will be incremented ("m" = 0) or decremented ("m" = 1) after each comparison. The "n" field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the address by one according to the "m" field, and decrements the byte count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, CCG indicates that the byte at C is greater than the byte at B, or CCL indicates that the byte at C is less than the byte at B; A indicates the number of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly, and C and B will point to the next locations not compared. Interrupts can occur after each compare.

SBW (1264--). Scan Bytes While. The SBW instruction expects A to contain a comparison byte in bits 8:15 and B to contain the byte address of the string to be scanned. The map to be used is determined by the "s" field of the instruction and by the DS, CS, LS, and Privileged bits of the ENV Register. The "m" field of the instruction determines if the source address will be incremented ("m" = 0) or decremented (<m> = 1) after each comparison. The scan is terminated when either a null byte is found in the string or a byte in the string does not match the test byte in A. When null byte termination occurs, the Carry (K) bit in the ENV Register is set. In either termination case, B points to the byte address that caused termination. RP is set to the "n" field of the instruction at instruction termination. Interrupts can occur after each compare.

SBU (1266--). Scan Bytes Until. The SBU instruction expects A.<8:15> to contain a test byte and B to contain the byte address of the string to be scanned. The map to be used is determined by the "s" field of the instruction and by the DS, CS, LS, and Privileged Bits of the ENV Register. The "m" field of the instruction determines if the scan address will be incremented ("m" = 0) or decremented (<m> = 1) after each comparison. The scan is terminated when either a null byte is found in the string or the test byte matches a byte in the string. The Carry (K) bit is set in the ENV Register when null byte termination occurs. In either case, B points to the byte address that caused the scan to cease. RP is set to the "n" field of the instruction at termination. Interrupts can occur after each compare.

MNDX (000227). Move Words While Not Duplicate, Extended. FE is assumed to contain a 32-bit destination address in extended memory, and DC is assumed to contain a 32-bit source address. The MNDX instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, CB contains the source address, and ED contains the destination address. Interrupts can occur after each compare.

CDX (000356). Count Duplicate Words, Extended. Beginning at the 32-bit address (in extended memory) specified in DC, and for a maximum count of words specified in B, this instruction counts the number of duplicate words in the buffer. A is incremented on each duplicate found, and may contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and DC contains the extended address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by DC and DC-2. Interrupts can occur after each compare. This instruction is intended to be used in conjunction with MNDX.

MVBX (000417). Move Bytes Extended. This instruction transfers a specified number of bytes from one area of extended memory to another. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are incremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends. Interrupts can occur after each compare.

MBXR (000420). Move Bytes Extended, Reverse. This instruction transfers a specified number of bytes from one area of extended memory to another, using reverse (decrementing) addresses. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are decremented and A is decremented. If A is equal to zero the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends. Interrupts can occur after each compare.

MBXX (000421). Move Bytes Extended, and Checksum. This instruction transfers a specified number of bytes from one area of extended memory to another, and computes a checksum value after each byte is moved. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, ED to contain a 32-bit destination byte address, and F to contain the initial checksum value. The move is made one byte at a time from the source to the destination. After each byte transfer the addresses are incremented, A is decremented, and new checksum is entered in F. If A is equal to zero, the instruction ends; otherwise the next byte is moved. Five words are deleted from the Register Stack when the instruction ends, leaving the final checksum value in A. Interrupts can occur after each compare.

CMBX (000422). Compare Bytes Extended. This instruction compares one area of extended memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. It expects A to contain a byte count, CB to contain a 32-bit source byte address and ED to contain a 32-bit destination byte address. The instruction fetches the contents of the source and destination addresses, compares them, increments the addresses by one, and decrements the byte count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, CCG indicates that the byte at ED is greater than the byte at CB, or CCL indicates that the byte at ED is less than the byte at CB; A indicates the count of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly; ED and CB point to the bytes after the last ones compared, and A is 0. Interrupts can occur after each compare.

XSMG (000343). Compute Checksum in Current Data. Starting at the address defined in register B, for a count of words defined in register A, the XSMG instruction exclusive-ORs each word into register C. When the count goes to zero, the two top words on the stack are deleted, leaving the final checksum in register A. The address in B refers to the current data segment only.

XSMX (000333). Compute Checksum Extended. Starting at the extended memory location defined by the 32-bit address in CB, for a count of words defined in register A, the XSMX instruction exclusive-ORs each word into register D. When the count goes to zero, the three top words on the stack are deleted, leaving the final checksum in register A.

PROGRAM REGISTER CONTROL

SETL (000020). Set L with A. The contents of the L Register, which points to the current stack marker, are replaced with the contents of register A. A is then deleted from the Register Stack.

SETS (000021). Set S with A. The contents of the S Register, which points to the top word of the stack in memory, are replaced with the contents of register A. A is then deleted from the stack. A Stack Overflow trap occurs if the result is greater than 32767.

SETE (000022). Set ENV with A. The least significant eight bits of the Environment Register (ENV) are replaced with the lower eight bits of the A Register. The most significant eight bits of the Environment Register are logically ANDed with the upper eight bits of the A Register. Thus this instruction may only clear the PRIV, DS, CS, and LS bits of the Environment Register, and may not set them. The programmer should take care with this instruction on NonStop II systems, since it is possible to inadvertently clear the Library Space (LS) bit, ENV.<4>.

SETP (000023). Set P with A. The contents of the Program Counter (P) are replaced with the contents of the A Register. A is deleted from the stack, and control is transferred to the new location indicated by P.

RDE (000024). Read ENV into A. The contents of the Environment Register (ENV) are pushed onto the Register Stack.

RDP (000025). Read P into A. The contents of the Program Counter (P) are pushed onto the Register Stack.

STRP (00010-). Set RP. The register pointer is set to the value in the Register field of the instruction. For binary coding details, see Table A-7 in Appendix A.

ADDS (002---). Add Immediate Operand to S. The signed immediate operand is added to the S register in integer form. If the resultant S is greater than 32767, then a Stack Overflow trap occurs.

CCL (000015). Set Condition Code to Less. A Condition Code of CCL (N = 1 and Z = 0) is set into the ENV Register.

CCE (000016). Set Condition Code to Equal. A Condition Code of CCE (N = 0 and Z = 1) is set into the ENV Register.

CCG (000017). Set Condition Code to Greater. A Condition Code of CCG (N = 0 and Z = 0) is set into the ENV Register.

ROUTINE CALLS AND RETURNS

PCAL (027---). Procedure Call. Control is transferred to an instruction specified by an entry in the Procedure Entry Point (PEP) Table; the specific PEP entry is indicated by the PEP Number field of the instruction. First, a three word stack marker, consisting of the current P, ENV, and L, is stored on the top of the current stack. If the caller is not privileged, the PEP Number is checked against PEP[0] and PEP[1] to see if the call is legal. If the call is not legal, an instruction failure trap occurs. (If the caller is privileged no checks are made.) L and S are set to S + 3 to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Finally, P is set from the PEP entry and control is transferred to the procedure.

XCAL (127---). External Procedure Call. The XCAL instruction is used to invoke procedures that are outside the current code segment. Control is transferred to an instruction in the external segment by a three-step sequence: 1) a number in the XEP field of the instruction refers to an entry in the XEP table of the current code segment; 2) the XEP entry specifies a PEP entry in one of the other three code segments that are currently mapped; 3) the PEP entry of the other code segment specifies a procedure entry point within that segment. See detailed description in Section 2 under the heading, "Calling External Procedures".

SCMP (000454). Set Code Map. This instruction is used to establish a code map number in register A for use by the DPCL instruction (next described). The instruction determines which code map defines the currently executing code (by examining the CS and LS bits of ENV) and loads the code map number into A.<0:3>. The code map number is equal to (LS*2 + CS + 2). In typical usage, succeeding instructions would pass this value to a procedure which would then issue the DPCL instruction.

DPCL (000032). Dynamic Procedure Call. Control is transferred to an instruction specified by an entry in the Procedure Entry Point (PEP) table; the specific PEP entry is indicated by bits 7:15 of A in the Register Stack. Bits 0:3 of register A specify the code map to use (2 = User Code, 3 = System Code, 4 = User Library, 5 = System Code Extension; any other value defaults to 2). First, a three word stack marker, consisting of the current P, ENV, and L, is stored on the top of the current stack. If the caller is not privileged, the PEP Number is checked to see if the call is legal. If the call is not legal, an Instruction Failure trap occurs. If the caller is privileged, no checks are made. L and S are set to S + 3 to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Next, if the call is to a callable system procedure, the PRIV bit in the ENV Register is set. CS is set to 1 if A.<0:3> is 3 or 5; otherwise it is set to 0. LS is set to 1 if A.<0:3> is 4 or 5; otherwise it is set to 0. Finally, P is set from the PEP entry, transferring control to the procedure.

EXIT (125---). Exit from Procedure. This instruction is used to return from a procedure called by a PCAL, XCAL, or DPCL instruction. EXIT assumes L[-2] to L[0] to contain a standard three-word stack marker consisting of P, ENV, and L. S is moved below the current stack marker and any parameters by setting it with the "S decrement" value subtracted from the current L Register setting. P is set to the return P value contained in L[-2] of the current stack marker. The caller's ENV Register value is set as follows: the mode (privileged or nonprivileged) and data area are reinstated to the lesser of the caller's and the current settings (e.g., a privileged caller can be made nonprivileged on the return, but not vice versa); the caller's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) are reinstated from L[-1]; Z and N (Condition Code) and RP are set to those of the current procedure. L is moved back to the preceding stack marker, thereby reinstating the preceding local data area, by setting L with the contents of the L[0] of the current stack marker.

DXIT (000072). DEBUG Exit. This instruction is used to reestablish the environment present at the time the DEBUG procedure was called. P, ENV, and L are restored from the stack marker generated by the DEBUG call, and S is reset to its value at the time of the call to DEBUG. This is a privileged instruction.

BSUB (-174--). Branch to Subprocedure. S is incremented by one and the return address (P) is saved in that location. Then a direct or indirect unconditional branch is taken (depending on the "i" field of the instruction). For binary coding details, see Table A-6 in Appendix A.

RSUB (025---). Return from Subroutine. This instruction is used to return from a subroutine called by a BSUB instruction. The instruction assumes that the return address is on the top of the memory stack (indicated by S) and returns control to that address. S is set to S - "S decrement". "S decrement" may be any number from 0 to 255; however, in order to delete the return address from the stack, it must be at least 1. For binary coding details, see Table A-5 in Appendix A.

INTERRUPT SYSTEM

RIR (000063). Reset Interrupt Register. This instruction is used by the operating system interrupt handlers to reset the appropriate INTA Register bit after an interrupt has occurred. Some interrupt bits must be reset (along with the clearing of a MASK bit) in order to allow further interrupts through that SIV (System Interrupt Vector Table) entry. The instruction expects A to contain the number of the bit in the INTA Register that is to be reset. This is a privileged instruction.

XMSK (000064). Exchange MASK with A. The contents of the MASK Register are interchanged with the contents of the A Register. This is a privileged instruction.

IXIT (000071). Interrupt Exit. This instruction is used by the operating system interrupt procedures to return control to the interrupted process. At the time the interrupt occurred, a stack marker was generated at the L pointed to by the System Interrupt Vector Table (SIV) for the specific interrupt. This was a special five-word marker (see Figure 2-59) that consisted of the MASK, S, P, ENV, and L at the time of the interrupt. This instruction reestablishes this environment (by loading the five registers with the values in the stack marker, and loading the Register Stack with the values in L+1 through L+8) and resumes execution of the interrupted process. At the time this instruction is executed, the needed values in L-4 through L+8 must be present and DS must be equal to one. This is a privileged instruction.

DISP (000073). Dispatch. This instruction sets bit 15 of INTA, and also sets Vi.<15> in the System Interrupt Vector (SIV) table entry for the Dispatcher interrupt. If bit 15 of MASK is set, a Dispatcher interrupt occurs immediately following this instruction (provided there are no interrupts of higher priority pending). Control is then transferred to the operating system Dispatcher whose location is pointed to by the SIV table entry. This is a privileged instruction.

BUS COMMUNICATION

TOTQ (000056). Test Out Queues. This instruction sets CCE if neither of the two Out Queues is full, or CCG if at least one Out Queue is full.

SEND (000065). Send Data over Interprocessor Bus. The SEND instruction expects register A to contain a byte count and registers CB to contain the absolute extended address of the source buffer. Register D is the OUTQ Full Timer; the timeout value is computed as: (32768 - <timeout>) times 0.8 specifies the time in microseconds for the specified bus to become ready (e.g., <timeout> of 0 = 32768 * 0.8 microseconds). Register E bits 0:7 specify the sender cpu and 8:15 specify the destination cpu. Register F specifies a sequence number, and register G bit 15 specifies which bus is to be used (0 = X, 1 = Y).

Data in the buffer is transmitted in 16-word packets consisting of 26 data bytes (13 words) plus three words for sequence number, sender and receiver cpu numbers, and checksum. Packets are transmitted until the byte count is zero. If the byte count is not a multiple of 26, then the last packet is padded with zeros to round the number of data bytes up to 26. Condition Code CCE indicates successful completion, and the Register Stack is marked empty.

If a timeout condition occurs, a Condition Code of CCL is returned, and the instruction terminates. The Out Queue is cleared.

SEND is a privileged instruction.

INPUT/OUTPUT

RSW (000026). Read the Switch Register into A. The contents of the Switch Register are pushed onto the Register Stack. Condition Code is set.

SSW (000027). Store A into Switch Register. The contents of the A Register are set in the Register Display and into sysstack[%122]. A is then deleted.

EIO (000060). Execute Input/Output. The EIO instruction expects bits 8:15 of A to contain the subchannel number, bits 0:7 of A to contain a command to its controller, and 0:15 of B to contain a parameter which is to be passed to that controller via the channel. The instruction first checks to see if the channel is available. If not it loops, waiting for channel availability but testing for other interrupts. When the channel becomes available, the command and address are sent to the controller by the channel via the LAC (Load Address and Command) T-bus command and the parameter is sent to the controller which is now selected via the LPRM (Load Parameter) T-bus command. Device status is then read from the controller via the RDST (Read Device Status) T-bus command. RP is decremented by one, and if there were no channel errors, device status is placed in A, the controller is then deselected via the DSEL (Deselect) T-bus command, the Condition Code is set to CCE and the instruction terminates. If there was a channel error, the ABTI (Abort Instruction) T-bus command is issued to the controller, deselecting it and terminating its activity. The contents of IOD, although probably invalid due to the channel error, are placed in A for evaluation. The Condition Code is set to CCL and the instruction terminates. This is a privileged instruction.

IIO (000061). Interrogate I/O. This instruction is used by the operating system interrupt handler to get the interrupt cause and interrupt status from a controller and to reset that interrupt. It first checks to see if the channel is available. If not it loops, waiting for channel availability but testing for other interrupts. When the channel is available, first rank 0 and then rank 1 of the i/o system are polled via the LPOL (Low Poll) T-bus command. The interrupting controller on the highest rank with the highest priority is then selected via the SEL (Select) T-bus command. The channel then loads the controller's interrupt cause into the C register, the interrupt status into the B register, and the channel status into the A register. Then the interrupt in the controller is cleared. If there were no channel errors indicated in A, and if interrupt status bits 0:3 are equal to zero, then CCE is set, and the instruction terminates. If there was a channel error then CCL is set, and the instruction terminates. CCG is set in the event of a device error or parity error. This is a privileged instruction.

HIIO (000062). High-Priority Interrogate I/O. This instruction is used by the operating system's high-priority interrupt handler to get the interrupt cause and status from a high-priority controller and to reset the corresponding interrupt. Execution is identical to the IIO instruction, except that HPOL (high priority polls) TBUS commands are issued and only controllers with the high-priority interrupt jumper installed can respond. This is a privileged instruction.

RCHN (000447). Reset I/O Channel. This instruction is used by the operating system to control the i/o channel in the event of a catastrophic error. If register A contains a value greater than or equal to zero, RCHN resets the i/o channel; if A contains a negative value, RCHN performs a lockup on the channel. Condition Code CCE indicates that the reset or lockup was performed, or CCL indicates that the channel was not available. This is a privileged instruction.

MISCELLANEOUS

NOP (000000). No Operation.

RCLK (000050). Read Clock. This instruction reads the quadrupleword microsecond counter (located in the System Data segment), adds the instantaneous value of the 14-bit hardware microsecond counter to it, and pushes the result onto the Register Stack. Note that since the software counter is updated only every 10 microseconds (each time the hardware counter rolls over), adding the hardware count to it provides an accurate clock indication at the instant that RCLK is executed.

RCPU (000051). Read CPU Number. This instruction reads this processor's cpu number from bits 0:7 of INTB and pushes this value onto the register stack.

BPT (000451). Instruction Breakpoint Trap. This instruction, although necessarily nonprivileged, can be used only by system software (DEBUG); proper operation requires access to the Environment Register, which requires privileged capability. The instruction assumes that DEBUG has inserted the BPT instruction at some user-specified point in the code, and has saved the instruction that formerly occupied that location in the Breakpoint Table in the System Data segment. When the code containing the BPT instruction is executed, BPT is normally executed twice--once when encountered following the preceding instruction, and once again to resume program execution at the following instruction. A bit (1) in the Environment Register is used as a flag to differentiate the two functions.

When BPT is first executed, bit 1 of the Environment Register is zero, which causes an interrupt to be generated (through SIV 19) to DEBUG. DEBUG sets ENV bit 1 to one and, after user debugging has been completed, returns to the interrupted code at the BPT instruction. This time, BPT first sets ENV bit 1 back to zero, then searches the Breakpoint Table, locates the saved instruction, loads that instruction into the Instruction (I) Register, and sets the microcode entry point for that instruction into the ROMA Register. Thus the breakpointed instruction gets executed, and execution proceeds normally to the succeeding instruction.

OPERATING SYSTEM FUNCTIONS

The following groups of instructions, most of them privileged, are used solely to implement certain operating system and diagnostic functions in firmware. These instructions are not intended for use in any user applications, and are listed here only for completeness.

Resource Management

XADD	(000033)	XRAY Add
MXON	(000040)	Mutual Exclusion On
MXFF	(000041)	Mutual Exclusion Off
SNDQ	(000052)	Signal a Send Is Queued
SFRZ	(000053)	System Freeze
DOFS	(000057)	Disc Record Offset
DLEN	(000070)	Disc Record Length
HALT	(000074)	Processor Halt
PSEM	(000076)	"P" a Semaphore
VSEM	(000077)	"V" a Semaphore
RPV	(000216)	Read PROM Version Numbers
WWCS	(000400)	Write LCS
VWCS	(000401)	Verify LCS
RWCS	(000402)	Read LCS
FRST	(000405)	Firmware Reset
RSMT	(000436)	Read from Operations and Service Processor (OSP)
WSMT	(000437)	Write to Operations and Service Processor (OSP)
RIBA	(000440)	Read INTB and INTA Registers
XSTR	(000442)	XRAY Start Timer
XSTP	(000443)	XRAY Stop Timer
BCLD	(000452)	Bus Cold Load
TPEF	(000453)	Test Parity Error Freeze Circuits

Memory Management

MAPS	(000042)	Map in a Segment
UMPS	(000043)	Unmap a Segment
RMAP	(000066)	Read Map
SMAP	(000067)	Set Map
CRAX	(000423)	Convert Relative to Absolute Extended Address
RSPT	(000424)	Read Segment Page Table Entry
WSPT	(000425)	Write Segment Page Table Entry
RXBL	(000426)	Read Extended Base and Limit
SXBL	(000427)	Set Extended Base and Limit
LCKX	(000430)	Lock Down Extended Memory
ULKX	(000431)	Unlock Extended Memory
CMRW	(000432)	Correctable Memory Error Read/Write
RMEM	(000434)	Read Memory
WMEM	(000435)	Write Memory
SVMP	(000441)	Save Map Entries
BNDW	(000450)	Bounds Test Words

List Management

DLTE	(000054)	Delete Element from List
INSR	(000055)	Insert Element into List
MRL	(000075)	Merge onto Ready List
FTL	(000206)	Find Position in Time List
DTL	(000207)	Determine Time Left for Element

Trace and Memory Breakpoint

TRCE	(000217)	Add Entry to Trace Table
SMBP	(000404)	Set Memory Breakpoint

APPENDIX A

HARDWARE INSTRUCTION LISTS

This appendix provides a number of reference tables pertaining to the instruction set of the NonStop II system.

The first two tables list all instructions in the instruction set with their mnemonics and opcodes, first in alphabetical order and then grouped by type of instruction. The remaining tables provide binary coding details for most of the instructions, grouped according to the coding patterns of the fields of the instruction words. (For example, all memory reference instructions are listed together.) These tables break down each instruction, bit by bit, into its component parts, indicate the operands, results, and ENV register bit settings, and show relationships between similar instructions.

The following tables are included in this appendix:

A key at the end of each table explains the symbols used.

NOTE

For some instructions, the six-digit opcode notation used in Tables A-1 and A-2 cannot give complete information about the opcode. For instance, the distinctions between QUP and QDWN, ORRI and ORLI, and LWP and LBP cannot be clearly shown. For complete information, refer to the entries for these instructions in Tables A-3 through A-9.

Table A-1. Alphabetical List of Instructions

Mnemonic	Description	Octal Code	
ADAR	Add A to Register	00016-	
ADDI	Add Immediate	104---	
ADDS	Add to S	002---	
ADM	Add to Memory	-74---	
ADRA	Add Register to A	00014-	
ADXI	Add to Index Immediate	104---	
ALS	Arithmetic Left Shift	0302--	
ANG	AND to Memory	000044	
ANLI	AND Left Immediate	007---	
ANRI	AND Right Immediate	006---	
ANS	AND to SG Memory	000034	
ANX	AND to Extended Memory	000046	
ARS	Arithmetic Right Shift	0303--	
BANZ	Branch on A	-154--	
BAZ	Branch on A Zero	-144--	
BCLD	Bus Cold Load	000452	*
BEQL	Branch if Equal	-12---	
BFI	Branch Forward Indirect	000030	
BGEQ	Branch if Greater or Equal	-13---	
BGTR	Branch if Greater	-11---	
BIC	Branch if Carry	-10---	
BLEQ	Branch if Less or Equal	-16---	
BLSS	Branch if Less	-14---	
BNDW	Bounds Test Words	000450	*
BNEQ	Branch if Not Equal	-15---	
BNOC	Branch if No Carry	-17---	
BNOV	Branch if No Overflow	-164--	
BOX	Branch on X	-1-4--	
BPT	Instruction Breakpoint Trap	000451	
BSUB	Branch to Subprocedure	-174--	
BTST	Byte Test	000007	
BUN	Branch	-104--	
CAQ	Convert ASCII to Quad	000262	$
CAQV	Convert ASCII to Quad with Initial Value	000261	$
CCE	Condition Code Equal to	000016	
CCG	Condition Code Greater than	000017	
CCL	Condition Code Less than	000015	
CDE	Convert Doubleword to Extended Float	000334	#
CDF	Convert Doubleword to Float	000306	#
CDFR	Convert Doubleword to Float (Round)	000326	#
CDG	Count Duplicate Words	000366	
CDI	Convert Doubleword to Integer	000307	
CDQ	Convert Doubleword to Quad	000265	$
CDX	Count Duplicate Words Extended	000356	
CED	Extended Float to Doubleword	000314	#
CEDR	Extended Float to Doubleword (Round)	000315	#

Table A-1. Alphabetical List of Instructions (Continued)

CEF	Extended Float to Float........................	000276	#
CEFR	Extended Float to Float (Round)................	000277	#
CEI	Extended Float to Integer......................	000337	#
CEIR	Extended Float to Integer (Round)..............	000316	#
CEQ	Extended Float to Quadrupleword................	000322	#
CEQR	Extended Float to Quadrupleword (Round)........	000323	#
CFD	Floating to Doubleword.........................	000312	#
CFDR	Floating to Doubleword (Round).................	000313	#
CFE	Floating to Extended Float.....................	000325	#
CFI	Floating to Integer............................	000311	#
CFIR	Floating to Integer (Round)....................	000310	#
CFQ	Floating to Quadrupleword......................	000320	#
CFQR	Floating to Quadrupleword (Round)..............	000321	#
CID	Convert Integer to Doubleword..................	000327	
CIE	Convert Integer to Extended Float..............	000332	#
CIF	Convert Integer to Floating....................	000331	#
CIQ	Convert Integer to Quad........................	000266	$
CLQ	Convert Logical to Quad........................	000267	$
CMBX	Compare Bytes Extended.........................	000422	
CMPI	Compare Immediate..............................	001---	
CMRW	Correctable Memory Error Read/Write............	000432	*
COMB	Compare Bytes..................................	1262--	
COMW	Compare Words..................................	0262--	
CQA	Convert Quad to ASCII..........................	000260	$
CQD	Convert Quad to Doubleword.....................	000247	$
CQE	Convert Quad to Extended.......................	000336	#
CQER	Convert Quad to Extended (Round)...............	000335	#
CQF	Convert Quad to Floating.......................	000324	#
CQFR	Convert Quad to Floating (Round)...............	000330	#
CQI	Convert Quad to Integer........................	000264	$
CQL	Convert Quad to Logical........................	000246	$
CRAX	Convert Relative to Absolute Extended..........	000423	*
DADD	Double Add.....................................	000220	
DALS	Double Arithmetic Left Shift...................	1302--	
DARS	Double Arithmetic Right Shift..................	1303--	
DCMP	Double Compare.................................	000225	
DDIV	Double Divide..................................	000223	
DDUP	Double Duplicate...............................	000006	
DFG	Deposit Field in Memory........................	000367	
DFS	Deposit Field in System........................	000357	
DFX	Deposit Field in Extended Memory...............	000416	
DISP	Dispatch.......................................	000073	*
DLEN	Disc Record Length.............................	000070	@
DLLS	Double Logical Left Shift......................	1300--	
DLRS	Double Logical Right Shift.....................	1301--	
DLTE	Delete from Linked List........................	000054	*
DMPY	Double Multiply................................	000222	
DNEG	Double Negate..................................	000224	
DOFS	Disc Record Offset.............................	000057	@

Table A-1. Alphabetical List of Instructions (Continued)

DPCL	Dynamic Procedure Call..........................	000032	
DPF	Deposit Field..................................	000014	
DSUB	Double Subtract................................	000221	
DTL	Determine Time Left for Element................	000207	*
DTST	Double Test....................................	000031	
DXCH	Double Exchange................................	000005	
DXIT	DEBUG Exit.....................................	000072	*
EADD	Extended Floating-Point Add....................	000300	#
ECMP	Extended Floating-Point Compare................	000305	#
EDIV	Extended Floating-Point Divide.................	000303	#
EIO	Execute I/O....................................	000060	*
EMPY	Extended Floating-Point Multiply...............	000302	#
ENEG	Extended Floating-Point Negate.................	000304	#
ESUB	Extended Floating-Point Subtract...............	000301	#
EXCH	Exchange.......................................	000004	
EXIT	Exit Procedure.................................	125---	
FADD	Floating-Point Add.............................	000270	#
FCMP	Floating-Point Compare.........................	000275	#
FDIV	Floating-Point Divide..........................	000273	#
FMPY	Floating-Point Multiply........................	000272	#
FNEG	Floating-Point Negate..........................	000274	#
FRST	Firmware Reset.................................	000405	*
FSUB	Floating-Point Subtract........................	000271	#
FTL	Find Position in Time List.....................	000206	*
HALT	Processor Halt.................................	000074	*
HIIO	High-Priority Interrogate I/O..................	000062	*
IADD	Integer Add....................................	000210	
ICMP	Integer Compare................................	000215	
IDIV	Integer Divide.................................	000213	
IDX1	Calculate Index, 1 Dimension...................	000344	#
IDX2	Calculate Index, 2 Dimension...................	000345	#
IDX3	Calculate Index, 3 Dimension...................	000346	#
IDXD	Calculate Index, Bounds in Data Space..........	000317	#
IDXP	Calculate Index, Bounds in Code Space..........	000347	#
IIO	Interrogate I/O................................	000061	*
IMPY	Integer Multiply...............................	000212	
INEG	Integer Negate.................................	000214	
INSR	Insert Element into Linked List................	000055	*
ISUB	Integer Subtract...............................	000211	
IXIT	Interrupt Exit.................................	000071	*
LADD	Logical Add....................................	000200	
LADI	Logical Add Immediate..........................	003---	
LADR	Load Address...................................	-7----	
LAND	Logical AND....................................	000010	
LBA	Load Byte via A................................	000364	
LBAS	Load Byte via A from System....................	000354	
LBP	Load Byte from Program.........................	-2-4--	
LBX	Load Byte Extended.............................	000406	
LBXX	Load Byte Extended, Indexed....................	0256--, 0266--	

Table A-1. Alphabetical List of Instructions (Continued)

LCKX	Lock Down Extended Memory........................	000430	*
LCMP	Logical Compare..................................	000205	
LDA	Load Double via A................................	000362	
LDAS	Load Double via A from System....................	000352	
LDB	Load Byte..	-5----	
LDD	Load Double......................................	-6----	
LDDX	Load Double Extended.............................	000412	
LDI	Load Immediate...................................	100---	
LDIV	Logical Divide...................................	000203	
LDLI	Load Left Immediate..............................	005---	
LDRA	Load Register to A...............................	00013-	
LDX	Load X...	-3----	
LDXI	Load X Immediate.................................	10----	
LLS	Logical Left Shift...............................	0300--	
LMPY	Logical Multiply.................................	000202	
LNEG	Logical Negate...................................	000204	
LOAD	Load...	-4----	
LOR	Logical OR.......................................	000011	
LQAS	Load Quadrupleword via A from SG.................	000445	*
LQX	Load Quadrupleword Extended......................	000414	
LRS	Logical Right Shift..............................	0301--	
LSUB	Logical Subtract.................................	000201	
LWA	Load Word via A..................................	000360	
LWAS	Load Word via A from System......................	000350	
LWP	Load Word from Program...........................	-2----	
LWUC	Load Word from User Code Space...................	000342	
LWX	Load Word Extended...............................	000410	
LWXX	Load Word Extended, Indexed......................	0254--, 0264--	
MAPS	Map In a Segment.................................	000042	*
MBXR	Move Bytes Extended, Reverse.....................	000420	
MBXX	Move Bytes Extended, Checksum....................	000421	
MNDX	Move Words while Not Duplicate, Extended.........	000227	
MNGG	Move Words while Not Duplicate...................	000226	
MOND	Minus One Double.................................	000001	
MOVB	Move Bytes.......................................	126---	
MOVW	Move Words.......................................	026---	
MRL	Merge onto Ready List............................	000075	*
MVBX	Move Bytes Extended..............................	000417	
MXFF	Mutual Exclusion Off.............................	000041	*
MXON	Mutual Exclusion On..............................	000040	*
NOP	No Operation.....................................	000000	
NOT	Not..	000013	
NSAR	Non-Destructive Store A in a Register............	00012-	
NSTO	Non-Destructive Store............................	-34---	
ONED	One Double.......................................	000003	
ORG	OR to Memory.....................................	000045	
ORLI	OR Left Immediate................................	0044--	
ORRI	OR Right Immediate...............................	004---	
ORS	OR to SG Memory..................................	000035	

Table A-1. Alphabetical List of Instructions (Continued)

ORX	OR to Extended Memory	000047	
PCAL	Procedure Call	027---	
POP	Pop from Stack	124---	
PSEM	"P" a Semaphore	000076	*
PUSH	Push to Stack	024---	
QADD	Quad Add	000240	
QCMP	Quad Compare	000245	$
QDIV	Quad Divide	000243	$
QDWN	Quad Scale Down	00025-	
QLD	Quad Load	00023-	
QMPY	Quad Multiply	000242	$
QNEG	Quad Negate	000244	$
QRND	Quad Round	000263	$
QST	Quad Store	00023-	
QSUB	Quad Subtract	000241	
QUP	Quad Scale Up	00025-	
RCHN	Reset I/O Channel	000447	*
RCLK	Read Clock	000050	
RCPU	Read Processor Number	000051	
RDE	Read E Register	000024	
RDP	Read P Register	000025	
RIBA	Read INTA and INTB Registers	000440	*
RIR	Reset Interrupt	000063	*
RMAP	Read Map	000066	*
RMEM	Read Memory	000434	*
RPV	Read PROM Version Numbers	000216	*
RSMT	Read from Operations and Service Processor	000436	*
RSPT	Read Segment Page Table Entry	000424	*
RSUB	Return from Subprocedure	025---	
RSW	Read Switches	000026	
RWCS	Read LCS	000402	*
RXBL	Read Extended Base and Limit	000426	*
SBA	Store Byte via A	000365	
SBAR	Subtract A from a Register	00017-	
SBAS	Store Byte via A into System	000355	
SBRA	Subtract Register from A	00015-	
SBU	Scan Bytes Until	1266--	
SBW	Scan Bytes While	1264--	
SBX	Store Byte Extended	000407	
SBXX	Store Byte Extended, Indexed	0257--, 0267--	
SCMP	Set Code Map	000454	
SCS	Set Code Segment	000444	
SDA	Store Double via A	000363	
SDAS	Store Double via A into System	000353	
SDDX	Store Double Extended	000413	
SEND	Send	000065	*
SETE	Set ENV Register	000022	
SETL	Set L Register	000020	
SETP	Set P Register	000023	

Table A-1. Alphabetical List of Instructions (Continued)

SETS	Set S Register	000021	
SFRZ	System Freeze	000053	*
SMAP	Set Map	000067	*
SMBP	Set Memory Breakpoint	000404	*
SNDQ	Signal a Send Is Queued	000052	*
SQAS	Store Quadrupleword via A to SG	000446	*
SQX	Store Quadrupleword Extended	000415	
SSW	Set Switches	000027	
STAR	Store A in Register	00011-	
STB	Store Byte	-54---	
STD	Store Double	-64---	
STOR	Store	-44---	
STRP	Set RP	00010-	
SVMP	Save Map Entries	000441	*
SWA	Store Word via A	000361	
SWAS	Store Word via A into System	000351	
SWX	Store Word Extended	000411	
SWXX	Store Word Extended, Indexed	0255--, 0265--	
SXBL	Set Extended Base and Limit	000427	*
TOTQ	Test OUTQ	000056	@
TPEF	Test Parity Error Freeze Circuits	000453	*
TRCE	Add an Entry to the Trace Table	000217	*
ULKX	Unlock Extended Memory	000431	*
UMPS	Unmap a Segment	000043	*
VSEM	"V" a Semaphore	000077	*
VWCS	Verify LCS	000401	*
WMEM	Write to Memory	000435	*
WSMT	Write to Operations and Service Processor	000437	*
WSPT	Write Segment Page Table Entry	000425	*
WWCS	Write to LCS	000400	*
XADD	XRAY Add	000033	*
XCAL	External Call	127---	
XMSK	Exchange Mask	000064	*
XOR	Exclusive OR	000012	
XSMG	Checksum Block	000343	
XSMX	Checksum Block Extended	000333	
XSTP	XRAY Stop Timer	000443	*
XSTR	XRAY Start Timer	000442	*
ZERD	Zero Double	000002	

The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

*	indicates a privileged instruction.
@	indicates an instruction designated for operating system use only.
$	indicates a decimal arithmetic optional instruction.
#	indicates a floating-point arithmetic optional instruction.

Table A-2. Categorized List of Instructions

16-Bit Arithmetic (Top of Register Stack)			
IADD	Integer Add	000210	
LADD	Logical Add	000200	
ISUB	Integer Subtract	000211	
LSUB	Logical Subtract	000201	
IMPY	Integer Multiply	000212	
LMPY	Logical Multiply	000202	
IDIV	Integer Divide	000213	
LDIV	Logical Divide	000203	
INEG	Integer Negate	000214	
LNEG	Logical Negate	000204	
ICMP	Integer Compare	000215	
LCMP	Logical Compare	000205	
CMPI	Integer Compare Immediate	001---	
ADDI	Integer Add Immediate	104---	
LADI	Logical Add Immediate	003---	
32-Bit Signed Arithmetic			
CDI	Convert Double to Integer	000307	
CID	Convert Integer to Double	000327	
DADD	Double Add	000220	
DSUB	Double Subtract	000221	
DMPY	Double Multiply	000222	
DDIV	Double Divide	000223	
DNEG	Double Negate	000224	
DCMP	Double Compare	000225	
DTST	Double Test	000031	
MOND	(Load) Minus One Double	000001	
ZERD	(Load) Zero Double	000002	
ONED	(Load) One Double	000003	
16-Bit Signed Arithmetic (Register Stack Element)			
ADRA	Add Register to A	00014-	
SBRA	Subtract Register from A	00015-	
ADAR	Add A to Register	00016-	
SBAR	Subtract A from Register	00017-	
ADXI	Add to Index Immediate	104---	
Decimal Arithmetic Load and Store			
QLD	Quadruple Load	00023-	
QST	Quadruple Store	00023-	
Decimal Integer Arithmetic			
QADD	Quadruple Add	000240	
QSUB	Quadruple Subtract	000241	
QMPY	Quadruple Multiply	000242	$
QDIV	Quadruple Divide	000243	$
QNEG	Quadruple Negate	000244	$
QCMP	Quadruple Compare	000245	$

Table A-2. Categorized List of Instructions (Continued)

Decimal Arithmetic Scaling and Rounding			
QUP	Quadruple Scale Up	00025-	
QDWN	Quadruple Scale Down	00025-	
QRND	Quadruple Round	000263	$
Decimal Arithmetic Conversions			
CQI	Convert Quad to Integer	000264	$
CQL	Convert Quad to Logical	000246	$
CQD	Convert Quad to Double	000247	$
CQA	Convert Quad to ASCII	000260	$
CIQ	Convert Integer to Quad	000266	$
CLQ	Convert Logical to Quad	000267	$
CDQ	Convert Double to Quad	000265	$
CAQ	Convert ASCII to Quad	000262	$
CAQV	Convert ASCII to Quad with Initial Value	000261	$
Floating-Point Arithmetic			
FADD	Floating-Point Add	000270	#
FSUB	Floating-Point Subtract	000271	#
FMPY	Floating-Point Multiply	000272	#
FDIV	Floating-Point Divide	000273	#
FNEG	Floating-Point Negate	000274	#
FCMP	Floating-Point Compare	000275	#
Extended Floating-Point Arithmetic			
EADD	Extended Floating-Point Add	000300	#
ESUB	Extended Floating-Point Subtract	000301	#
EMPY	Extended Floating-Point Multiply	000302	#
EDIV	Extended Floating-Point Divide	000303	#
ENEG	Extended Floating-Point Negate	000304	#
ECMP	Extended Floating-Point Compare	000305	#
Floating-Point Conversions			
CEF	Convert Extended to Floating	000276	#
CEFR	Convert Extended to Floating, Rounded	000277	#
CFI	Convert Floating to Integer	000311	#
CFIR	Convert Floating to Integer, Rounded	000310	#
CFD	Convert Floating to Double	000312	#
CFDR	Convert Floating to Double, Rounded	000313	#
CED	Convert Extended to Double	000314	#
CEDR	Convert Extended to Double, Rounded	000315	#
CEI	Convert Extended to Integer	000337	#
CEIR	Convert Extended to Integer, Rounded	000316	#
CFQ	Convert Floating to Quad	000320	#
CFQR	Convert Floating to Quad, Rounded	000321	#
CEQ	Convert Extended to Quad	000322	#
CEQR	Convert Extended to Quad, Rounded	000323	#

Table A-2. Categorized List of Instructions (Continued)

CFE	Convert Floating to Extended.................	000325	#
CIF	Convert Integer to Floating..................	000331	#
CDF	Convert Double to Floating...................	000306	#
CDFR	Convert Double to Floating, Rounded..........	000326	#
CQF	Convert Quad to Floating.....................	000324	#
CQFR	Convert Quad to Floating, Rounded............	000330	#
CIE	Convert Integer to Extended..................	000332	#
CDE	Convert Double to Extended...................	000334	#
CQE	Convert Quad to Extended.....................	000336	#
CQER	Convert Quad to Extended, Rounded............	000335	#
Floating-Point Functionals			
IDX1	Calculate Index, 1 Dimension.................	000344	#
IDX2	Calculate Index, 2 Dimensions................	000345	#
IDX3	Calculate Index, 3 Dimensions................	000346	#
IDXP	Calculate Index, Bounds in Code Space........	000347	#
IDXD	Calculate Index, Bounds in Data Space........	000317	#
Register Stack Manipulation			
EXCH	Exchange A with B............................	000004	
DXCH	Double Exchange..............................	000005	
DDUP	Double Duplicate.............................	000006	
STAR	Store A in a Register........................	00011-	
NSAR	Non-Destructive Store A in a Register........	00012-	
LDRA	Load A from a Register.......................	00013-	
LDI	Load Immediate...............................	100---	
LDXI	Load Index Immediate.........................	10----	
LDLI	Load Left Immediate..........................	005---	
Boolean Operations			
LAND	Logical AND..................................	000010	
LOR	Logical OR...................................	000011	
XOR	Exclusive OR.................................	000012	
NOT	NOT..	000013	
ORRI	OR Right Immediate...........................	004---	
ORLI	OR Left Immediate............................	0044--	
ANRI	AND Right Immediate..........................	006---	
ANLI	AND Left Immediate...........................	007---	
Bit Shift and Deposit			
DPF	Deposit Field................................	000014	
LLS	Logical Left Shift...........................	0300--	
DLLS	Double Logical Left Shift....................	1300--	
LRS	Logical Right Shift..........................	0301--	
DLRS	Double Logical Right Shift...................	1301--	
ALS	Arithmetic Left Shift........................	0302--	
DALS	Double Arithmetic Left Shift.................	1302--	
ARS	Arithmetic Right Shift.......................	0303--	
DARS	Double Arithmetic Right Shift................	1303--	

Table A-2. Categorized List of Instructions (Continued)

Byte Test		
BTST	Byte Test	000007
Memory Stack to/from Register Stack		
LWP	Load Word from Program	-2----
LBP	Load Byte from Program	-2-4--
PUSH	Push Registers to Memory	024---
POP	Pop Memory to Registers	124---
LWXX	Load Word Extended, Indexed	0254--, 0264--
SWXX	Store Word Extended, Indexed	0255--, 0265--
LBXX	Load Byte Extended, Indexed	0256--, 0266--
SBXX	Store Byte Extended, Indexed	0257--, 0267--
LDX	Load Index	-3----
NSTO	Non-Destructive Store	-34---
LOAD	Load Word	-4----
STOR	Store Word	-44---
LDB	Load Byte	-5----
STB	Store Byte	-54---
LDD	Load Double	-6----
STD	Store Double	-64---
LADR	Load Address of Variable	-7----
ADM	Add to Memory	-74---
Load and Store via Address on Register Stack		
ANS	AND to SG Memory	000034
ORS	OR to SG Memory	000035
ANG	AND to Current Data	000044
ORG	OR to Current Data	000045
ANX	AND to Extended Memory	000046
ORX	OR to Extended Memory	000047
LWUC	Load Word from User Code Segment	000342
LWAS	Load Word via A from System	000350
LWA	Load Word via A	000360
SWAS	Store Word via A into System	000351
SWA	Store Word via A	000361
LDAS	Load Double via A from System	000352
LDA	Load Double via A	000362
SDAS	Store Double via A into System	000353
SDA	Store Double via A	000363
LBAS	Load Byte via A from System	000354
LBA	Load Byte via A	000364
SBAS	Store Byte via A into System	000355
SBA	Store Byte via A	000365
DFS	Deposit Field into System Data	000357
DFG	Deposit Field in Current Data	000367

Table A-2. Categorized List of Instructions (Continued)

LBX	Load Byte Extended	000406	
SBX	Store Byte Extended	000407	
LWX	Load Word Extended	000410	
SWX	Store Word Extended	000411	
LDDX	Load Doubleword Extended	000412	
SDDX	Store Doubleword Extended	000413	
LQX	Load Quadrupleword Extended	000414	
SQX	Store Quadrupleword Extended	000415	
DFX	Deposit Field Extended	000416	
SCS	Set Code Segment	000444	
LQAS	Load Quadrupleword via A from SG	000445	*
SQAS	Store Quadrupleword via A to SG	000446	*
Branching			
BIC	Branch if Carry	-10---	
BUN	Branch Unconditionally	-104--	
BOX	Branch on Index	-1-4--	
BGTR	Branch if CC Greater	-11---	
BEQL	Branch if CC Equal	-12---	
BGEQ	Branch if CC Greater or Equal	-13---	
BLSS	Branch if CC Less	-14---	
BAZ	Branch if A Zero	-144--	
BNEQ	Branch if CC Not Equal	-15---	
BANZ	Branch if A Not Zero	-154--	
BLEQ	Branch if CC Less or Equal	-16---	
BNOV	Branch if no Overflow	-164--	
BNOC	Branch if no Carry	-17---	
BFI	Branch Forward Indirect	000030	
Moves, Compares, and Scans			
MNGG	Move Words While Not Duplicate	000226	
CDG	Count Duplicate Words	000366	
MOVW	Move Words	026---	
MOVB	Move Bytes	126---	
COMW	Compare Words	0262--	
COMB	Compare Bytes	1262--	
SBW	Scan Bytes While	1264--	
SBU	Scan Bytes Until	1266--	
MNDX	Move Words While Not Duplicate, Extended	000227	
CDX	Count Duplicate Words Extended	000356	
MVBX	Move Bytes Extended	000417	
MBXR	Move Bytes Extended Reverse	000420	
MBXX	Move Bytes Extended, and Checksum	000421	
CMBX	Compare Bytes Extended	000422	
Program Register Control			
SETL	Set L Register	000020	
SETS	Set S Register	000021	
SETE	Set ENV Register	000022	
SETP	Set P Register	000023	

Table A-2. Categorized List of Instructions (Continued)

RDE	Read E Register	000024	
RDP	Read P Register	000025	
STRP	Set Register Pointer	00010-	
ADDS	Add to S Register	002---	
CCL	Set CC Less	000015	
CCE	Set CC Equal	000016	
CCG	Set CC Greater	000017	
Routine Calls/Returns			
PCAL	Procedure Call	027---	
XCAL	External Procedure Call	127---	
SCMP	Set Code Map	000454	
DPCL	Dynamic Procedure Call	000032	
EXIT	Exit from Procedure	125---	
DXIT	DEBUG Exit	000072	*
BSUB	Branch to Subprocedure	-174--	
RSUB	Return from Subprocedure	025---	
Checksum Computation			
XSMG	Compute Checksum in Current Data	000343	
XSMX	Compute Checksum Extended	000333	
Interrupt System			
RIR	Reset INT Register	000063	*
XMSK	Exchange MASK Register	000064	*
IXIT	Exit from Interrupt Handler	000071	*
DISP	Dispatch	000073	*
RIBA	Read INTA and INTB Registers	000440	*
Bus Communication			
TOTQ	Test Out Queues	000056	@
SEND	Send Packet	000065	*
Input/Output			
RSW	Read Switch Register	000026	
SSW	Set Switch Register	000027	
EIO	Execute I/O	000060	*
IIO	Interrogate I/O	000061	*
HIIO	High-Priority Interrogate I/O	000062	*
RCHN	Reset I/O Channel	000447	*
Miscellaneous Nonprivileged			
NOP	No Operation	000000	
RCLK	Read Clock	000050	
RCPU	Read Processor Number	000051	
BPT	Instruction Breakpoint Trap	000451	

Table A-2. Categorized List of Instructions (Continued)

Resource Management			
XADD	XRAY Add	000033	*
MXON	Mutual Exclusion On	000040	*
MXFF	Mutual Exclusion Off	000041	*
SNDQ	Signal a Send Is Queued	000052	*
SFRZ	System Freeze	000053	*
DOFS	Disc Record Offset	000057	@
DLEN	Disc Record Length	000070	@
HALT	Processor Halt	000074	*
PSEM	"P" a Semaphore	000076	*
VSEM	"V" a Semaphore	000077	*
RPV	Read PROM Version Numbers	000216	*
WWCS	Write LCS	000400	*
VWCS	Verify LCS	000401	*
RWCS	Read LCS	000402	*
FRST	Firmware Reset	000405	*
RSMT	Read from Operations and Service Processor	000436	*
WSMT	Write to Operations and Service Processor	000437	*
XSTR	XRAY Start Timer	000442	*
XSTP	XRAY Stop Timer	000443	*
BCLD	Bus Cold Load	000452	*
TPEF	Test Parity Error Freeze Circuits	000453	*
Memory Management			
MAPS	Map In a Segment	000042	*
UMPS	Unmap a Segment	000043	*
RMAP	Read Map	000066	*
SMAP	Set Map	000067	*
CRAX	Convert Relative to Absolute Extended	000423	*
RSPT	Read Segment Page Table Entry	000424	*
WSPT	Write Segment Page Table Entry	000425	*
RXBL	Read Extended Base and Limit	000426	*
SXBL	Set Extended Base and Limit	000427	*
LCKX	Lock Down Extended Memory	000430	*
ULKX	Unlock Extended Memory	000431	*
CMRW	Correctable Memory Error Read/Write	000432	*
RMEM	Read Memory	000434	*
WMEM	Write Memory	000435	*
SVMP	Save Map Entries	000441	*
BNDW	Bounds Test Words	000450	*
List Management			
DLTE	Delete Element from List	000054	*
INSR	Insert Element into List	000055	*
MRL	Merge onto Ready List	000075	*
FTL	Find Position in Time List	000206	*
DTL	Determine Time Left for Element	000207	*

Table A-2. Categorized List of Instructions (Continued)

Trace and Breakpoints			
TRCE	Add Entry to Trace Table......................	000217	*
SMBP	Set Memory Breakpoint........................	000404	*

The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

*	indicates a privileged instruction.
@	indicates an instruction designated for operating system use only.
$	indicates a decimal arithmetic optional instruction.
#	indicates a floating-point arithmetic optional instruction.

Table A-3. Binary Coding, Memory Reference Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
I	2	0 X X	0 +/- ←	P	→	LWP a
I	2	0 X X	1 +/- ←	P	→	LBP b
I	3	0 X X	←	G,L,SG,S	→	LDX a
I	3	1 X X	←	G,L,SG,S	→	NSTO
I	4	0 X X	←	G,L,SG,S	→	LOAD a
I	4	1 X X	←	G,L,SG,S	→	STOR
I	5	0 X X	←	G,L,SG,S	→	LDB b
I	5	1 X X	←	G,L,SG,S	→	STB
I	6	0 X X	←	G,L,SG,S	→	LDD a
I	6	1 X X	←	G,L,SG,S	→	STD
I	7	0 X X	←	G,L,SG,S	→	LADR
I	7	1 X X	←	G,L,SG,S	→	ADM vk a
		P+	0 .	. . .	. . .	0:177
		P-	1 .	. . .	. . .	0:177
		G+	0 . .	. . .	. . .	0:377
		L+	1 0 .	. . .	. . .	0:177
		SG	1 1 0	. . .	. . .	0:77
		L-	1 1 1	0 . .	. . .	0:37
		S-	1 1 1	1 . .	. . .	0:37

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

v = Overflow

k = Carry

cc = Condition Codes:

a	L (result < 0) or (opr1 < opr2) E (result = 0) or (opr1 = opr2) G (result > 0) or (opr1 > opr2)
b	L (ASCII numeric) E (ASCII alpha) G (ASCII special)
c	L (channel error or timeout) E (no error) G (unusual condition)

Note: opr1 is first item pushed on stack; opr2 is second.

Table A-4. Binary Coding, Immediate Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
1	0	0	+/- ←	OPERAND	→	LDI a
1	0	0 X X	+/- ←	OPERAND	→	LDXI a
0	0	1	+/- ←	OPERAND	→	CMPI a
0	0	2	+/- ←	OPERAND	→	ADDS a
0	0	3	+/- ←	OPERAND	→	LADI k a
0	0	4	0 ←	OPERAND	→	ORRI a
0	0	4	1 ←	OPERAND	→	ORLI a
1	0	4	+/- ←	OPERAND	→	ADDI vk a
1	0	1 X X	+/- ←	OPERAND	→	ADXI vk a
0	0	5	+/- ←	OPERAND	→	LDLI a
0	0	6	+/- ←	OPERAND	→	ANRI a
0	0	7	+/- ←	OPERAND	→	ANLI a

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

vkcc: see Table A-3 footnote.

Table A-5. Binary Coding, Move/Shift/Call/Extended Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15		vkcc
0	2	4	N	LAST	COUNT-1	PUSH	
1	2	4	N	LAST	COUNT-1	POP	
0	2	5	0 ←	SDEC	→	RSUB	
1	2	5	0 ←	SDEC	→	EXIT	
0	2	5/6	4	DISPLACEMENT		LWXX	a
0	2	5/6	5	DISPLACEMENT		SWXX	
0	2	5/6	6	DISPLACEMENT		LBXX	b
0	2	5/6	7	DISPLACEMENT		SBXX	
0	2	6	0 0 RL	S S D	RP	MOVW	
0	2	6	0 1 RL	S S D	RP	COMW	a
1	2	6	0 0 RL	S S D	RP	MOVB	
1	2	6	0 1 RL	S S D	RP	COMB	a
1	2	6	1 0 RL	S S D	RP	SBW	k
1	2	6	1 1 RL	S S D	RP	SBU	k
0	2	7	←	PEP	→	PCAL	
1	2	7	←	PEP	→	XCAL	
0	3	0	0	← SHIFT	COUNT →	LLS	a
1	3	0	0	← SHIFT	COUNT →	DLLS	a
0	3	0	1	← SHIFT	COUNT →	LRS	a
1	3	0	1	← SHIFT	COUNT →	DLRS	a
0	3	0	2	← SHIFT	COUNT →	ALS	a
1	3	0	2	← SHIFT	COUNT →	DALS	a
0	3	0	3	← SHIFT	COUNT →	ARS	a
1	3	0	3	← SHIFT	COUNT →	DARS	a

RL (0/1) left-to-right (increasing addresses)
right-to-left (decreasing addresses)

SS (source map):
- 00 Current Data
- 01 System Data (Current Data if nonprivileged user)
- 10 Current Code
- 11 User Code

D = (destination map), data only
- 0 Current Data
- 1 System Data (Current Data if Nonprivileged User)

PEP = Procedure Entry Point Table

SDEC = stack S decrement

vkcc: see Table A-3 footnote.

Table A-6. Binary Coding, Branch Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	vkcc
I	1	0	0 +/-	P	→	BIC
I	1	0	4 +/-	P	→	BUN
I	1	0 X X	4 +/-	P	→	BOX
I	1	1	0 +/-	P	→	BGTR
I	1	2	0 +/-	P	→	BEQL
I	1	3	0 +/-	P	→	BGEQ
I	1	4	0 +/-	P	→	BLSS
I	1	4	4 +/-	P	→	BAZ
I	1	5	0 +/-	P	→	BNEQ
I	1	5	4 +/-	P	→	BANZ
I	1	6	0 +/-	P	→	BLEQ
I	1	6	4 +/-	P	→	BNOV
I	1	7	0 +/-	P	→	BNOC
I	1	7	4 +/-	P	→	BSUB

+/- (0/1) implies two's-complement notation; the sign is extended through bit 0 at execution.

I (0/1) indicates direct or indirect address.

Note: since the Program Counter register holds the address of the next instruction, a branch-self instruction (Branch *) would be coded: BUN P-1.

vkcc: see Table A-3 footnote.

Table A-7. Binary Coding, Stack Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE →		

7:15>				vkcc	<7:15>				vkcc
0	0	0	NOP		0	5	1	*RCPU	
0	0	1	MOND	a	0	5	2	*SNDQ	
0	0	2	ZERD	a	0	5	3	*SFRZ	
0	0	3	ONED	a	0	5	4	*DLTE	
0	0	4	EXCH	a	0	5	5	*INSR	
0	0	5	DXCH	a	0	5	6	@TOTQ	!
0	0	6	DDUP	a	0	5	7	@DOFS	c
0	0	7	BTST	b	0	6	0	*EIO	c
0	1	0	LAND	a	0	6	1	*IIO	c
0	1	1	LOR	a	0	6	2	*HIIO	
0	1	2	XOR	a	0	6	3	*RIR	
0	1	3	NOT	a	0	6	4	*XMSK	
0	1	4	DPF	a	0	6	5	*SEND	!
0	1	5	CCL	a	0	6	6	*RMAP	
0	1	6	CCE	a	0	6	7	*SMAP	
0	1	7	CCG	a	0	7	0	@DLEN	
0	2	0	SETL		0	7	1	*IXIT	
0	2	1	SETS		0	7	2	*DXIT	
0	2	2	SETE	!! !	0	7	3	*DISP	
0	2	3	SETP		0	7	4	*HALT	
0	2	4	RDE		0	7	5	*MRL	
0	2	5	RDP		0	7	6	*PSEM	
0	2	6	RSW	a	0	7	7	*VSEM	
0	2	7	SSW		1	0	reg	STRP	
0	3	0	BFI		1	1	reg	STAR	
0	3	1	DTST	a	1	2	reg	NSAR	
0	3	2	DPCL		1	3	reg	LDRA	a
0	3	3	*XADD		1	4	reg	ADRA	vk a
0	3	4	ANS	a	1	5	reg	SBRA	vk
0	3	5	ORS	a	1	6	reg	ADAR	vk
0	4	0	*MXON		1	7	reg	SBAR	vk a
0	4	1	*MXFF		2	0	0	LADD	k a
0	4	2	*MAPS		2	0	1	LSUB	k a
0	4	3	*UMPS		2	0	2	LMPY	v=0a
0	4	4	ANG	a	2	0	3	LDIV	v a
0	4	5	ORG	a	2	0	4	LNEG	k a
0	4	6	ANX	a	2	0	5	LCMP	a
0	4	7	ORX	a	2	0	6	*FTL	
0	5	0	RCLK		2	0	7	*DTL	

Table A-7. Binary Coding, Stack Instructions (Continued)

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE →		

<7:15>				vkcc		<7:15>				vkcc	
2	1	0	IADD	vk	a	4	0	5	*FRST		
2	1	1	ISUB	vk	a	4	0	6	LBX		b
2	1	2	IMPY	v	a	4	0	7	SBX		
2	1	3	IDIV	v	a	4	1	0	LWX		a
2	1	4	INEG	vk	a	4	1	1	SWX		
2	1	5	ICMP		a	4	1	2	LDDX		a
2	1	6	*RPV			4	1	3	SDDX		
2	1	7	*TRCE			4	1	4	LQX		a
2	2	0	DADD	vk	a	4	1	5	SQX		
2	2	1	DSUB	vk	a	4	1	6	DFX		a
2	2	2	DMPY	vk	a	4	1	7	MVBX		
2	2	3	DDIV	vk	a	4	2	0	MBXR		
2	2	4	DNEG	vk	a	4	2	1	MBXX		
2	2	5	DCMP		a	4	2	2	CMBX		!
2	2	6	MNGG		!	4	2	3	*CRAX		
2	2	7	MNDX		!	4	2	4	*RSPT	!	
3	3	3	XSMX			4	2	5	*WSPT		
3	4	2	LWUC		a	4	2	6	*RXBL		
3	4	3	XSMG			4	2	7	*SXBL		
3	5	0	LWAS		a	4	3	0	*LCKX		!
3	5	1	SWAS			4	3	1	*ULKX		!
3	5	2	LDAS		a	4	3	2	*CMRW		!
3	5	3	SDAS			4	3	4	*RMEM		a
3	5	4	LBAS		b	4	3	5	*WMEM		
3	5	5	SBAS			4	3	6	*RSMT		
3	5	6	CDX			4	3	7	*WSMT		
3	5	7	DFS		a	4	4	0	*RIBA		
3	6	0	LWA		a	4	4	1	*SVMP		
3	6	1	SWA			4	4	2	*XSTR		
3	6	2	LDA		a	4	4	3	*XSTP		
3	6	3	SDA			4	4	4	SCS		
3	6	4	LBA		b	4	4	5	*LQAS		a
3	6	5	SBA			4	4	6	*SQAS		
3	6	6	CDG			4	4	7	*RCHN		!
3	6	7	DFG		a	4	5	0	*BNDW		!
4	0	0	*WWCS		!	4	5	1	BPT		
4	0	1	*VWCS		!	4	5	2	*BCLD		
4	0	2	*RWCS			4	5	3	*TPEF		
4	0	4	*SMBP			4	5	4	SCMP		

Table A-7. Binary Coding, Stack Instructions (Continued)

* indicates a privileged instruction.
@ indicates an instruction designated for operating system use only.

vkcc: see Table A-3 footnote.

! = special vkcc meanings; see instruction definitions in Table B-1.

Table A-8. Binary Coding, Decimal Arithmetic Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE →		

<7:15>				vkcc	<7:15>				vkcc
2	3	0	+QST		2	5	0	+QUP	v a
2	3	1	+QST x5		2	5	1	+QDWN	v=0
2	3	2	+QST x6		2	5	2	+QUP(2)	v a
2	3	3	+QST x7		2	5	3	+QDWN(2)	v=0a
2	3	4	+QLD	a	2	5	4	+QUP(3)	v a
2	3	5	+QLD x5	a	2	5	5	+QDWN(3)	v=0a
2	3	6	+QLD x6	a	2	5	6	+QUP(4)	v a
2	3	7	+QLD x7	a	2	5	7	+QDWN(4)	v=0a
2	4	0	+QADD	vk a	2	6	0	CQA	v a
2	4	1	+QSUB	vk a	2	6	1	CAQV	v !
2	4	2	QMPY	v a	2	6	2	CAQ	v !
2	4	3	QDIV	v a	2	6	3	QRND	v=0a
2	4	4	QNEG	vk a	2	6	4	CQI	v
2	4	5	QCMP	a	2	6	5	CDQ	
2	4	6	CQL	v	2	6	6	CIQ	
2	4	7	CQD	v	2	6	7	CLQ	

+ indicates an instruction that is standard in all processors (not part of decimal option).

! CCE if entire string is ASCII digits, CCG if not.

vkcc: see Table A-3 footnote.

Table A-9. Binary Coding, Floating-Point Instructions

0	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15
0	0	0	← STACK OPERAND CODE →		

<7:15>				vkcc	
2	7	0	FADD	v	a
2	7	1	FSUB	v	a
2	7	2	FMPY	v	a
2	7	3	FDIV	v	a
2	7	4	FNEG		a
2	7	5	FCMP		a
2	7	6	CEF		a
2	7	7	CEFR		a
3	0	0	EADD	v	a
3	0	1	ESUB	v	a
3	0	2	EMPY	v	a
3	0	3	EDIV	v	a
3	0	4	ENEG		a
3	0	5	ECMP		a
3	0	6	CDF		a
3	0	7	+CDI		a
3	1	0	CFIR		a
3	1	1	CFI		a
3	1	2	CFD		a
3	1	3	CFDR		a
3	1	4	CED		a
3	1	5	CEDR		a

<7:15>				vkcc
3	1	6	CEIR	a
3	1	7	IDXD	a
3	2	0	CFQ	a
3	2	1	CFQR	a
3	2	2	CEQ	a
3	2	3	CEQR	a
3	2	4	CQF	a
3	2	5	CFE	a
3	2	6	CDFR	a
3	2	7	+CID	a
3	3	0	CQFR	a
3	3	1	CIF	a
3	3	2	CIE	a
3	3	4	CDE	a
3	3	5	CQER	a
3	3	6	CQE	a
3	3	7	CEI	a
3	4	4	IDX1	a
3	4	5	IDX2	a
3	4	6	IDX3	a
3	4	7	IDXP	a

+ indicates an instruction that is standard in all processors (not part of floating-point option).

vkcc: see Table A-3 footnote.

APPENDIX B

INSTRUCTION SET DEFINITION

This appendix consists of a table (B-1) giving brief definitions of all the instructions in the NonStop II instruction set, in numeric opcode order. A TAL-like notation is used for the definitions. This table is a specification of the instruction microcode, and is provided for those interested in microcode details such as the use of the register stack.

Table B-2 is a key to the symbols used in the instruction definitions.

Table B-1. Instruction Set Definition

Note: The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

*	indicates a privileged instruction.
@	indicates an instruction designated for operating system use only.
$	indicates a decimal arithmetic optional instruction.
#	indicates a floating-point arithmetic optional instruction.
op(x)	indicates that an operation similar to that performed by the instruction ´op´ should be done using the value(s) ´x´.

0 0 0 0 0 0	NOP	no operation	
0 0 0 0 0 1	MOND	minus one double	RP:=RP+2; cc(B:=A:=-1)
0 0 0 0 0 2	ZERD	zero double	RP:=RP+2; cc(B:=A:=0)
0 0 0 0 0 3	ONED	one double	RP:=RP+2; B:=0; cc(A:=1)
0 0 0 0 0 4	EXCH	exchange	A:=:B; cc(A)
0 0 0 0 0 5	DXCH	double exchange	BA:=:CD; cc(BA)
0 0 0 0 0 6	DDUP	double duplicate	RP:=RP+2; cc(BA:=DC)
0 0 0 0 0 7	BTST	byte test	ccb(A.<8:15>); RP:=RP-1
0 0 0 0 1 0	LAND	logical AND	cc(B:=B&A); RP:=RP-1
0 0 0 0 1 1	LOR	logical OR	cc(B:=B|A); RP:=RP-1

Table B-1. Instruction Set Definition (Continued)

0 0 0 0 1 2	XOR	exclusive OR	cc(B:=B xor A); RP:=RP-1
0 0 0 0 1 3	NOT	logical NOT	cc(A:= ˜ A)
0 0 0 0 1 4	DPF	deposit field	cc(C:=(C&B | A&˜B));
			RP:=RP-2
0 0 0 0 1 5	CCL	cond. code less	Z:=0; N:=1
0 0 0 0 1 6	CCE	cond. code equal	Z:=1; N:=0
0 0 0 0 1 7	CCG	cond. code greater	Z:=N:=0
0 0 0 0 2 0	SETL	set L register	L:=A; RP:=RP-1
0 0 0 0 2 1	SETS	set S register	S:=A; RP:=RP-1
0 0 0 0 2 2	SETE	set ENV register	ENV.<0:7>:=ENV.<0:7>&A.<0:7>;
			ENV.<8:15>:=A.<8:15>
0 0 0 0 2 3	SETP	set P register	P:=A; RP:=RP-1
0 0 0 0 2 4	RDE	read ENV register	RP:=RP+1; A:=ENV
0 0 0 0 2 5	RDP	read P register	RP:=RP+1; A:=P
0 0 0 0 2 6	RSW	read switches	RP:=RP+1; cc(A:=SWITCHES)
0 0 0 0 2 7	SSW	set switches	sysstack[%122]:=LIGHTS:=A;
			RP:=RP-1
0 0 0 0 3 0	BFI	branch forward	P:=P+A+code[P+A];
		indirect	RP:=RP-1
0 0 0 0 3 1	DTST	double test	cc(BA)
0 0 0 0 3 2	DPCL	dynamic procedure	stack[S+1:S+3]:=(P,ENV,L);
		call	m:=A.<0:3>; t:=A.<7:15>;
			if m<2 or m>5 then m:=2;
			if ˜ PRIV then
			{if t>=mem[m,0] then
			{if t>=mem[m,1]
			then priv trap;
			PRIV:=1
			}
			};
			L:=S:=S+3;
			LS:=(m-2).<14>;
			CS:=m.<15>;
			P:=code[t]; RP:=7
0 0 0 0 3 3*	XADD	XRAY add	if (t:=xmem[BA])<>0 then
		D=value to add to	{a:=%40000^(t+C)^0;
		counter	xmem[a:a+3]:=xmem[a:a+3]+D;
		C=offset to cntr	if D<0 and xmem[a:a+1]<0
		BA=extended addr	then xmem[a:a+3]:=0};
		of XRAY ptr	RP:=RP-4
0 0 0 0 3 4	ANS	AND to SG memory	cc(dest(A):=dest(A) & B);
			RP:=RP-2
0 0 0 0 3 5	ORS	OR to SG memory	cc(dest(A):=dest(A) | B);
			RP:=RP-2
0 0 0 0 3 6			*** undefined ***
0 0 0 0 3 7			*** undefined ***
0 0 0 0 4 0*	MXON	mutual exclusion	chkp(stack[(L-20) max 0]);
		on	chkp(stack[S+A.<8:15>]);
		A=<0:7> code size	if A.<0:7>
		<8:15>stack size	then chkp(code[P+A.<0:7>]);
			stack[L+1]:=MASK;
			MASK:=MASK & %177640;
			RP:=RP-1
0 0 0 0 4 1*	MXFF	mutual exclusion	MASK:=stack[L+1]
		off	

Table B-1. Instruction Set Definition (Continued)

0 0 0 0 4 2*	MAPS	map in a segment
		B=segment number
		A=map number
			if CMSEG[A]<>B then
			{if CMSEG[A]<>-1
			 then UMPS(A);
			 j:=B*2;
			 for i:=32 to
			    $min(64,32+SEG[j].<9:15>)
			 do
			  {if MAP[15,i].<0:14>=B then
			    {mem[SEG[j].<5:8>,
			         SEG[j+1]+i-32+
			         MAP[15,i].<15>*32]
			        := t := MAP[15,i-32];
			     MAP[15,i]:=-1}};
			 i:=0;
			 while i<SEG[j].<9:15> do
			 {MAP[A,i]:=
			        mem[SEG[j].<5:8>,
			            SEG[j+1]+i];
			  i := i+1};
			 while i <= 63 do
			 {MAP[A,i]:=1; i:=i+1};
			 SEG[j].<0:4> := A;
			 CMSEG[A] := B;
			};
			RP:=RP-2;
			!!! Note !!!
			the page table must be
			in memory

0 0 0 0 4 3*	UMPS	unmap a segment
		A=map number
			j:= SEG[CMSEG[A]*2].<9:15>;
			m:= SEG[CMSEG[A]*2].<5:8>
			p:= SEG[CMSEG[A]*2+1];
			for i := 0 to j-1 do
			 {mem[m,p+i]:=t:=MAP[A,i];
			SEG[CMSEG[A]*2].<0:4>:=%37;
			CMSEG[A] := -1;
			RP:=RP-1
			!!! Note !!!
			the page table must be
			in memory

0 0 0 0 4 4	ANG	AND to memory	cc(stack[A]:=stack[A] & B);
			RP:=RP-2

0 0 0 0 4 5	ORG	OR to memory	cc(stack[A]:=stack[A] | B);
			RP:=RP-2

0 0 0 0 4 6	ANX	AND to extended memory	cc(xmem[BA]:=xmem[BA] & C);
			RP:=RP-3

0 0 0 0 4 7	ORX	OR to extended memory	cc(xmem[BA]:=xmem[BA] | C);
			RP:=RP-3

0 0 0 0 5 0	RCLK	read clock	RP:=RP+4;
			DCBA:=sysstack[%103:%106]+
			     microsecond counter

0 0 0 0 5 1	RCPU	read processor #	RP:=RP+1; A:=processor #
0 0 0 0 5 2*	SNDQ	signal that a SEND is queued	set dispatcher interrupt; sysstack[%1277].<14>:=1
0 0 0 0 5 3*	SFRZ	system freeze	assert system freeze; halt

Table B-1. Instruction Set Definition (Continued)

0 0 0 0 5 4*	DLTE	delete an element from a doubly linked, circular list A=element address	if sysstack[A] <> 0 then {if sysstack[sysstack[A]+1] <> A or sysstack[sysstack[A+1]] <> A then Instruction Failure; f:=sysstack[A]; b:=sysstack[A+1]; sysstack[b]:=f; sysstack[f+1]:=b; sysstack[A]:=0; sysstack[A+1]:=0; }; RP:=RP-1 !!! Note !!! all memory locations accessed must be present
0 0 0 0 5 5*	INSR	insert an element into a doubly linked, circular list B=list header A=list element	if A=0 or sysstack[sysstack[B]+1] <> B or sysstack[sysstack[B+1]] <> B then Instruction Failure; f:=sysstack[B]; sysstack[B]:=A; sysstack[A]:=f; sysstack[A+1]:=B; sysstack[f+1]:=A; RP:=RP-2 !!! Note !!! all memory locations accessed must be present
0 0 0 0 5 6@	TOTQ	test OUTQ's	N:=0; Z:=1; if either OUTQ full then Z:=0
0 0 0 0 5 7@	DOFS	disc record offset A=record number on return, A holds offset into buffer of record	if A'>='512 or (A:=xmem[stack[L+2:3]-A*2]) '>='stack[L+4] then {P:=stack[L+5]; RP:=7};
0 0 0 0 6 0*	EIO	execute i/o	ioselect(A.subchannel); iocontrol(A.command,B); B:='device status'; cc(A:='channel status')
0 0 0 0 6 1*	IIO	interrogate i/o	RP:=RP+3; C:='interrupt cause'; B:='interrupt status'; cc(A:='channel status');
0 0 0 0 6 2*	HIIO	high-priority interrogate i/o	RP:=RP+3; C:='high-priority interrupt cause'; B:='high-priority interrupt status'; cc(A:='channel status');
0 0 0 0 6 3*	RIR	reset interrupt register	'clear interrupt' A.<12:15> RP:=RP-1
0 0 0 0 6 4*	XMSK	exchange mask	MASK:=:A

Table B-1. Instruction Set Definition (Continued)

0 0 0 0 6 5*	SEND	send	do
			{do until OUTQEMPTY or
		G=<15> bus	.8(32768-D) microsec;
		F=sequence #	if OUTQEMPTY then
		E=<0:7> sender	{if A<>0 then
		cpu #	{bus:=G.<15>
		<8:15> receiver	receiver:=E.<8:15>;
		cpu #	OUTQ[bus,0]:=E;
		D=OUTQ full timer	OUTQ[bus,1]:=F;
		CB=buffer address	for i:=4 to 29 do
		A=byte count	{if A <> 0 then
			{boq[bus,i]:=bxmem[CB];
			A:=A-1; CB:=CB+1}
			else boq[bus,i]:=0};
			OUTQ[bus,15]:=(-1) xor
			OUTQ[bus,0]
			... OUTQ[bus,14];
			D:=0;
			if (F:=F+1)=0 then
			{done:=true; N:=0; Z:=1};
			} else
			{done:=true; N:=0; Z:=1
			}
			} else
			{done:=true; N:=1; Z:=0;
			OUTQEMPTY:=true
			};
			} until done;
			RP:=RP-7
			!!! Note !!!
			xmem[CB:CB+A*2-1] must be
			in memory
0 0 0 0 6 6*	RMAP	read map	A:=MAP[A.<12:15>,A.<0:5>];
0 0 0 0 6 7*	SMAP	set map	MAP[A.<12:15>,A.<0:5>]:=B;
			RP:=RP-2
0 0 0 0 7 0@	DLEN	disc record length	if (A:=DOFS(A+1)-DOFS(A)) < 0
		A=record number	then {P:=stack[L+5]; RP:=7}
0 0 0 0 7 1*	IXIT	interrupt exit	(MASK,S,P,ENV,L)
			:=sysstack[L-4:L];
			R[0:7]:=sysstack[L+1:L+8]
			!!! Note !!!
			sysstack[L-4:L+8] must be
			present
			DS must be 1
0 0 0 0 7 2*	DXIT	DEBUG exit	S:=L-3;
			(P,ENV,L):=stack[L-2:L];
			if ENV.<0>
			then Instruction Breakpoint
0 0 0 0 7 3*	DISP	dispatch	set dispatcher interrupt;
			sysstack[%1277].<15>:=1
0 0 0 0 7 4*	HALT	processor halt	halt
0 0 0 0 7 5*	MRL	merge onto ready	t := sysstack[%101];
		list	while sysstack[t+2].<8:15> <
		A=PCB address	sysstack[A+2].<8:15>
			do t:=sysstack[t+1];
			if sysstack[CPCB+2].<8:15> <
			sysstack[A+2].<8:15>
			then DISP;
			insert A after t; RP:=RP-1

Table B-1. Instruction Set Definition (Continued)

0 0 0 0 7 6*	PSEM	"P" a semaphore	sysstack[A+2]:=sysstack[A+2] -1;
		CB=wait time	if < then
		A=semaphore addr	{set dispatcher interrupt;
			sysstack[%1277]:= sysstack[%1277] | 5}
			else {C:=1;
			sysstack[A+3]:=CPCB};
			RP:=RP-2
			!!! Note !!!
			sysstack must be resident
0 0 0 0 7 7*	VSEM	"V" a semaphore	sysstack[A+2]:=sysstack[A+2] +1;
		A=semaphore addr	if <= then
			{set dispatcher interrupt;
			sysstack[%1277].<12>:=1}
			else sysstack[A+3]:=0;
			RP:=RP-1
			!!! Note !!!
			sysstack must be resident
0 0 0 1 0 reg	STRP	set RP	RP:=reg
0 0 0 1 1 reg	STAR	store A in reg	R[reg]:=A; RP:=RP-1
0 0 0 1 2 reg	NSAR	non-destructive store A in reg	R[reg]:=A
0 0 0 1 3 reg	LDRA	load register to A	RP:=RP+1; cc(A:=R[reg])
0 0 0 1 4 reg	ADRA	add register to A	ccn(A:=A+R[reg])
0 0 0 1 5 reg	SBRA	subtract register from A	ccn(A:=A-R[reg])
0 0 0 1 6 reg	ADAR	add A to register	ccn(R[reg]:=R[reg]+A);
			RP:=RP-1
0 0 0 1 7 reg	SBAR	subtract A from register	ccn(R[reg]:=R[reg]-A);
			RP:=RP-1
0 0 0 2 0 0	LADD	logical add	ccl(B:=B+A); RP:=RP-1
0 0 0 2 0 1	LSUB	logical subtract	ccl(B:=B-A); RP:=RP-1
0 0 0 2 0 2	LMPY	logical multiply	cc(BA:=B´*´A); V:=0
0 0 0 2 0 3	LDIV	logical divide	V:=(C´>=´A);
			(C,B):=(CB ´mod´ A,CB´/´A);
			cc(B); RP:=RP-1
0 0 0 2 0 4	LNEG	logical negate	ccl(A:=-A)
0 0 0 2 0 5	LCMP	logical compare	cc(B´:´A); RP:=RP-2
0 0 0 2 0 6*	FTL	find position in time list	RP:=RP+1; BA:=CB;
			C:=sysstack[%107];
			while C<>%107 do
		BA=time value	{BA:=BA-sysstack[C+2:C+3];
			if < then done;
			C:=sysstack[C]}
			!!! Note !!!
			sysstack must be resident
0 0 0 2 0 7*	DTL	delete from time list	a:=A; t:=sysstack[%107];
			RP:=RP+1;
		A=element address	BA:=sysstack[t+2:t+3];
			while a <> t do
			{t:=sysstack[t];
			BA:=BA+sysstack[t+2:t+3]}
			!!! Note !!!
			sysstack must be resident
0 0 0 2 1 0	IADD	integer add	ccn(B:=B+A); RP:=RP-1
0 0 0 2 1 1	ISUB	integer subtract	ccn(B:=B-A); RP:=RP-1
0 0 0 2 1 2	IMPY	integer multiply	V:=˜(-32768<=B*A<=32767);
			cc(B:=B*A); RP:=RP-1
0 0 0 2 1 3	IDIV	integer divide	V:=˜(-32768<=B/A<=32767);
			cc(B:=B/A); RP:=RP-1
0 0 0 2 1 4	INEG	integer negate	ccn(A:=-A)
0 0 0 2 1 5	ICMP	integer compare	cc(B:A); RP:=RP-2

Table B-1. Instruction Set Definition (Continued)

0 0 0 2 1 6*	RPV	read PROM version numbers	RP:=RP+5; N:=0; Z:=1;
			CBA:=cs prom numbers
			D:=ept prom numbers
			E:=i/o channel prom number
			if i/o channel not available
			then {N:=1; Z:=0}

0 0 0 2 1 7*	TRCE	add an entry to the trace table
		EDCBA=entry
			if TRBASE´<´TRLIM then
			{sysstack[TRACE:TRACE+4]:=
			EDCBA;
			TRACE:=TRACE+5;
			if TRACE´>´TRLIM
			then TRACE:=TRBASE};
			RP:=RP-5

0 0 0 2 2 0	DADD	double add	ccn(DC:=DC+BA); RP:=RP-2
0 0 0 2 2 1	DSUB	double subtract	ccn(DC:=DC-BA); RP:=RP-2
0 0 0 2 2 2	DMPY	double multiply	ccn(DC:=DC*BA); RP:=RP-2

0 0 0 2 2 3	DDIV	double divide	ccn(DC:=DC/BA); V:= BA=0;
			RP:=RP-2

0 0 0 2 2 4	DNEG	double negate	ccn(BA:=-BA)
0 0 0 2 2 5	DCMP	double compare	cc(DC:BA); RP:=RP-4

0 0 0 2 2 6	MNGG	move words while not duplicate
		D=destination
		C=source
		B=count
		A=value<>to value of source
			while cc(B)<>"=" and stack[C]<>A do
			{A:=stack[D]:=stack[C];
			D:=D+1;
			C:=C+1;
			B:=B-1};
			RP:=RP-1

0 0 0 2 2 7	MNDX	move words while not duplicate
		FE=destination
		DC=source
		B=count
		A=value<>to value of source
			while cc(B)<>"=" and xmem[DC]<>A do
			{A:=xmem[FE]:=xmem[DC];
			FE:=FE+2;
			DC:=DC+2;
			B:=B-1};
			RP:=RP-1

0 0 0 2 3 0xx	QST	quad store	adr:=(if I=%230 then 0
			else R[I.<14:15>+4])*4+A;
			stack[adr:adr+3]:=EDCB;
			RP:=RP-5

0 0 0 2 3 4xx	QLD	quad load	adr:=(if I=%234 then 0
			else R[I.<14:15>+4])*4+A;
			RP:=RP+3;
			cc(DCBA:=stack[adr:adr+3])

0 0 0 2 4 0	QADD	quad add	ccn(HGFE:=HGFE + DCBA);
			RP:=RP-4

0 0 0 2 4 1	QSUB	quad subtract	ccn(HGFE:=HGFE - DCBA);
			RP:=RP-4

0 0 0 2 4 2$	QMPY	quad multiply	V:=if
			-2**63<=HGFE*DCBA<=2**63-1
			then 0 else 1;
			HGFE:=HGFE * DCBA;
			cc(HGFE);
			RP:=RP-4

0 0 0 2 4 3$	QDIV	quad divide	V:=if DCBA=0 then 1 else 0;
			HGFE:=HGFE / DCBA;
			cc(HGFE);
			RP:=RP-4

0 0 0 2 4 4$	QNEG	quad negate	DCBA:=-DCBA;
			ccn(DCBA)

0 0 0 2 4 5$	QCMP	quad compare	cc(HGFE:DCBA)

Table B-1. Instruction Set Definition (Continued)

0 0 0 2 4 6$	CQL	convert quad to logical	V:=if 0 <= DCBA <=2**16-1
			then 0 else 1;
			D:=A;
			RP:=RP-3

0 0 0 2 4 7$	CQD	convert quad to double	V:=if -2**31 <=DCBA<= 2**31-1
			then 0 else 1;
			DC:=BA;
			RP:=RP-2

0 0 0 2 5 nn0	QUP	quad scale up	DCBA:=DBCA*
			10**(I.<13:14>+1);
			V:=if -2**63<=DCBA<=2**63-1
			then 0 else 1;
			cc(DCBA)

0 0 0 2 5 nn1	QDWN	quad scale down	DCBA:=DBCA/
			10**(I.<13:14>+1);
			V:=0; cc(DCBA);

0 0 0 2 6 0$	CQA	convert quad to ASCII	cc(FEDC);
			B:=B+A;
			while A<>0 do
			{B:=B-1;
			bytedest(B):=
			%60+abs(FEDC) mod 10;
			FEDC:=FEDC/10;
			A:=A-1}
			V:=if FEDC=0 then 0 else 1;
			RP:=RP-6

0 0 0 2 6 1$	CAQV	convert ASCII to quad with initial value	V:=0;
			N:=1;
			while E<>0 and V=0 and N=1 do
			{ccb(t:=bytedest(F));
			if N=1 then
			{DCBA:=DCBA*10 + t&%17;
			V:=if DCBA<=2**63-1
			then 0 else 1;
			F:=F+1;
			E:=E-1}}
			cc(E) !cce if entire string
			!is ASCII digits.
			!ccg if not.
			!Note: initial value (DCBA)
			! should be positive.

0 0 0 2 6 2$	CAQ	convert ASCII to quad	RP:=RP+4;
			DCBA:=0;
			V:=0;
			N:=1;
			while E<>0 and V=0 and N=1 do
			{ccb(t:=bytedest(F));
			if N=1 then
			{DCBA:=DCBA*10 + t&%17;
			V:=if DCBA<=2**63-1
			then 0 else 1;
			F:=F+1;
			E:=E-1}}
			cc(E) !cce if entire string
			!is ASCII digits.
			!ccg if not.

0 0 0 2 6 3$	QRND	quad round	DCBA:=(if DCBA<0 then DCBA-5
			else DCBA+5) / 10;
			V:=0;
			cc(DCBA)

0 0 0 2 6 4$	CQI	convert quad to integer	V:=if -2**15 <=DCBA<= 2**15-1
			then 0 else 1;
			D:=A; RP:=RP-3;

Table B-1. Instruction Set Definition (Continued)

0 0 0 2 6 5$	CDQ	convert double to	(t,u):=BA;
		quad	s:=if B<0
			then %177777 else 0;
			RP:=RP+2;
			DCBA:=(s,s,t,u)
0 0 0 2 6 6$	CIQ	convert integer to	t:=A;
		quad	s:=if A<0
			then %177777 else 0;
			RP:=RP+3;
			DCBA:=(s,s,s,t)
0 0 0 2 6 7$	CLQ	convert logical to	t:=A;RP:=RP+3;
		quad	DCBA:=(0,0,0,t)
0 0 0 2 7 0#	FADD	floating add	t1:=exponent(C);
		DC:=DC+BA	t2:=exponent(A);
			if BA<>0 and DC<>0
			and abs(t1-t2)<24 then
			{sign1:=D.<0>;
			sign2:=B.<0>;
			D.<0>:=B.<0>:=1;
			exponent(C):=0;
			exponent(A):=0;
			s:=t1-t2;
			if s>=0 then
			BA:=BA´>>´s;
			else
			{DC:=DC´>>´-s;
			DC:=:BA;
			t1:=t2}
			if sign1=sign2 then
			{DC:=DC´+´BA;
			if carry then
			{DC:=DC´>>´1;
			t1:=t1+1;
			D.<0>:=1}}
			else
			{DC:=DC´-´BA;
			if not carry then
			{DC:=-DC;
			sign1:=~sign1}
			if DC=0 then
			t1:=sign1:=0
			else
			while D.<0>=0 do
			{DC:=DC´<<´1;
			t1:=t1-1}}
			DC:=DC´+´%400;
			if carry then
			t1:=t1+1;
			if t1.<6>=1 then
			call overflow;
			D.<0>:=sign1;
			exponent(C):=t1}
			else
			if DC=0 or t1-t2<=-24 then
			DC:=BA;
			cc(DC); RP:=RP-2
0 0 0 2 7 1#	FSUB	floating subtract	if BA<>0 then
		DC:=DC-BA	B.<0>:=~B.<0>;
			goto FADD

Table B-1. Instruction Set Definition (Continued)

```
0 0  0  2  7  2#|FMPY|floating multiply |if DC=0 or BA=0 then
                |    |DC:=DC*BA         |   DC:=0
                |    |                  |else
                |    |                  |  {t1:=exponent(C);
                |    |                  |   t2:=exponent(A);
                |    |                  |   exp:=t1+t2-255;
                |    |                  |   sign:=D.<0> xor B.<0>;
                |    |                  |   D.<0>:=B.<0>:=1;
                |    |                  |   exponent(C):=0;
                |    |                  |   exponent(A):=0;
                |    |                  |   DCBA:=DC´*´BA;
                |    |                  |   norm(DC);
                |    |                  |   DC:=DC´+´%400;
                |    |                  |   if carry then
                |    |                  |      exp:=exp+1;
                |    |                  |   if exp.<6>=1 then
                |    |                  |      call overflow;
                |    |                  |   D.<0>:=sign;
                |    |                  |   exponent(C):=exp}
                |    |                  |cc(DC); RP:=RP-2
0 0  0  2  7  3#|FDIV|floating divide   |if BA=0 then
                |    |DC:=DC/BA         |   call overflow;
                |    |                  |if DC<>0 then
                |    |                  |  {t1:=exponent(C);
                |    |                  |   t2:=exponent(A);
                |    |                  |   exp:=t1-t2+256;
                |    |                  |   sign:=D.<0> xor B.<0>;
                |    |                  |   D.<0>:=B.<0>:=1;
                |    |                  |   exponent(C):=0;
                |    |                  |   exponent(A):=0;
                |    |                  |   DC:=DC´/´BA;
                |    |                  |   norm(DC);
                |    |                  |   DC:=DC´+´%400;
                |    |                  |   if carry then
                |    |                  |      exp:=exp+1;
                |    |                  |   if exp.<6>=1 then
                |    |                  |      call overflow;
                |    |                  |   D.<0>:=sign;
                |    |                  |   exponent(C):=exp}
                |    |                  |cc(DC); RP:=RP-2
0 0  0  2  7  4#|FNEG|floating negate   |if BA<>0 then
                |    |BA:=-BA           |   B.<0>:=~B.<0>;
                |    |                  |cc(BA)
0 0  0  2  7  5#|FCMP|floating compare  |if D.<0> <> B.<0> then
                |    |DC:BA             |   cc(D:B)
                |    |                  |else
                |    |                  |  {sign:=D.<0>;
                |    |                  |   D.<0>:=B.<0>:=0;
                |    |                  |   t1:=exponent(C);
                |    |                  |   t2:=exponent(A);
                |    |                  |   if t1<>t2 then
                |    |                  |     if sign=0 then
                |    |                  |          cc(t1:t2)
                |    |                  |     else cc(t2:t1)
                |    |                  |   else
                |    |                  |     if sign=0 then
                |    |                  |          cc(DC:BA)
                |    |                  |     else cc(BA:DC)}
                |    |                  |RP:=RP-4
0 0  0  2  7  6#|CEF |convert extended  |exponent(C):=exponent(A);
                |    |to floating       |RP:=RP-2
```

Table B-1. Instruction Set Definition (Continued)

0 0 0 2 7 7#	CEFR	convert extended to floating with rounding	sign:=D.<0>; D.<0>:=1; exp:=exponent(A); DC:=DC´+´%400; if carry then {exp:=exp+1; if exp.<6> then V:=1} D.<0>:=sign; exponent(C):=exp; RP:=RP-2
0 0 0 3 0 0#	EADD	extended add HGFE:=HGFE+DCBA	t1:=exponent(E); t2:=exponent(A); if DCBA<>0 and HGFE<>0 and abs(t1-t2)<56 then {sign1:=H.<0>; sign2:=D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; s:=t1-t2; if s>=0 then DCBA:=DCBA´>>´s; else {HGFE:=HGFE´>>´-s; HGFE:=:DCBA; t1:=t2} if sign1=sign2 then {HGFE:=HGFE´+´DCBA; if carry then {HGFE:=HGFE´>>´1; t1:=t1+1; H.<0>:=1}} else {HGFE:=HGFE´-´DCBA; if not carry then {HGFE:=-HGFE; sign1:=~sign1} if HGFE=0 then t1:=sign1:=0 else while H.<0>=0 do {HGFE:=HGFE´<<´1; t1:=t1-1}} HGFE:=HGFE´+´%400; if carry then t1:=t1+1; if t1.<6>=1 then call overflow; H.<0>:=sign1; exponent(E):=t1} else if HGFE=0 or t1-t2<=-56 then HGFE:=DCBA; cc(HGFE); RP:=RP-4
0 0 0 3 0 1#	ESUB	extended subtract HGFE:=HGFE-DCBA	if DCBA<>0 then D.<0>:=~D.<0>; goto EADD

Table B-1. Instruction Set Definition (Continued)

0	0	0	3	0	2#	EMPY	extended multiply HGFE:=HGFE*DCBA	if HGFE=0 or DCBA=0 then HGFE:=0 else {t1:=exponent(E); t2:=exponent(A); exp:=t1+t2-255; sign:=H.<0> xor D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; HGFE:=HGFE´*´DCBA; norm(HGFE); HGFE:=HGFE´+´%400; if carry then exp:=exp+1; if exp.<6>=1 then call overflow; H.<0>:=sign; exponent(E):=exp} cc(HGFE); RP:=RP-4
0	0	0	3	0	3#	EDIV	extended divide HGFE:=HGFE/DCBA	if DCBA=0 then call overflow; if HGFE<>0 then {t1:=exponent(E); t2:=exponent(A); exp:=t1-t2+256; sign:=H.<0> xor D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; HGFE:=HGFE´/´DCBA; norm(HGFE); HGFE:=HGFE´+´%400; if carry then exp:=exp+1; if exp.<6>=1 then call overflow; H.<0>:=sign; exponent(E):=exp} cc(HGFE); RP:=RP-4
0	0	0	3	0	4#	ENEG	extended negate DCBA:=-DCBA	if DCBA<>0 then D.<0>:=~D.<0>; cc(DCBA)
0	0	0	3	0	5#	ECMP	extended compare HGFE:DCBA	if H.<0> <> D.<0> then cc(H:D) else {sign:=H.<0>; H.<0>:=D.<0>:=0; t1:=exponent(E); t2:=exponent(A); if t1<>t2 then if sign=0 then cc(t1:t2) else cc(t2:t1) else if sign=0 then cc(HGFE:DCBA) else cc(DCBA:HGFE)}
0	0	0	3	0	6#	CDF	convert double to floating	sign:=B.<0>; exp:=31+256; if sign=1 then BA:=-BA; if BA<>0 then {norm(BA); exponent(A):=exp; B.<0>:=sign}

Table B-1. Instruction Set Definition (Continued)

0	0	0	3	0	7	CDI	convert double to integer	if B+A.<0> <> 0 then V:=1; B:=A; RP:=RP-1

0	0	0	3	1	0#	CFIR	convert floating to integer with rounding	t:=15+256-exponent(A);
								sign:=B.<0>;
								if -2**15 <= BA <= 2**15-1
								then {B.<0>:=1;
								BA:=BA´>>´t;
								BA:=BA´+´%100000;
								if sign=1 then B:=-B
								else if B.<0>=1 then
								V:=1}
								else V:=1;
								cc(B); RP:=RP-1

0	0	0	3	1	1#	CFI	convert floating to integer	t:=15+256-exponent(A);
								sign:=B.<0>;
								if -2**15 <= BA <= 2**15-1
								then {B.<0>:=1;
								BA:=BA´>>´t;
								if sign=1 then B:=-B}
								else V:=1;
								cc(B); RP:=RP-1

0	0	0	3	1	2#	CFD	convert floating to double	t:=31+256-exponent(A);
								sign:=B.<0>;
								if -2**31 <= BA <= 2**31-1
								then {B.<0>:=1;
								exponent(A):=0;
								BA:=BA´>>´t;
								if sign=1 then
								BA:=-BA}
								else V:=1;
								cc(BA)

0	0	0	3	1	3#	CFDR	convert floating to double with rounding	t:=31+256-exponent(A);
								sign:=B.<0>;
								if -2**31 <= BA <= 2**31-1
								then {B.<0>:=1;
								exponent(A):=0;
								BAs:=BAs´>>´t;
								BAs:=BAs´+´%100000;
								if sign=1 then
								BA:=-BA
								else if B.<0>=1 then
								V:=1}
								else V:=1;
								cc(BA)

0	0	0	3	1	4#	CED	convert extended to double	t:=31+256-exponent(A);
								sign:=D.<0>;
								if -2**31 <= DCBA <= 2**31-1
								then {D.<0>:=1;
								DC:=DC´>>´t;
								if sign=1 then
								DC:=-DC}
								else V:=1;
								cc(DC); RP:=RP-2

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 1 5#	CEDR	convert extended to double with rounding	t:=31+256-exponent(A);
			sign:=D.<0>;
			if -2**31 <= DCBA <= 2**31-1
			then {D.<0>:=1;
			DCB:=(DCB´>>´t)
			´+´%100000;
			if sign=1 then
			DC:=-DC
			else if D.<0>=1 then
			V:=1}
			else V:=1;
			cc(DC); RP:=RP-2

0 0 0 3 1 6#	CEIR	convert extended to integer with rounding	t:=15+256-exponent(A);
			sign:=D.<0>;
			if -2**15 <= DCBA <= 2**15-1
			then {D.<0>:=1;
			DC:=(DC´>>´t)
			´+´%100000;
			if sign=1 then D:=-D
			else if D.<0>=1 then
			V:=1}
			else V:=1;
			cc(D); RP:=RP-3

0 0 0 3 1 7#	IDXD	calculate index offset and test indices for bounds violation
		
		(bounds table in data space)
			t:=stack[A];
			bc:=t.<0>; t.<0>:=0;
			indv:=0; psize:=1;
			s:=A;
			while t>0 do
			{lower:=stack[s:=s+1];
			upper:=stack[s:=s+1];
			if B<lower and bc=0 then
			{V:=1; t =0;
			cc(-1); R[7]:=B}
			if B>upper and bc=0 then
			{V:=1; t =0;
			cc(1); R[7]:=B}
			size:=upper-lower+1;
			B:=B-lower;
			indv:=indv+psize*B;
			psize:=psize*size;
			RP:=RP-1; t:=t-1}
			if V=0 then
			{R[7]:=indv;
			cc(R[7])}
			RP:=RP-1

0 0 0 3 2 0#	CFQ	convert floating to quad	t:=63+256-exponent(A);
			sign:=B.<0>; RP:=RP+2;
			if -2**63 <= DC <= 2**63-1
			then {D.<0>:=1;
			exponent(C):=0;
			B:=A:=0;
			DCBA:=DCBA´>>´t;
			if sign=1 then
			DCBA:=-DCBA}
			else V:=1;
			cc(DCBA)

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 2 1#	CFQR	convert floating to quad with rounding	t:=63+256-exponent(A);
			sign:=B.<0>; RP:=RP+2;
			if -2**63 <= DC <= 2**63-1
			then {D.<0>:=1;
			exponent(C):=0;
			B:=A:=s:=0;
			DCBAs:=(DCBAs´>>´t)
			´+´%100000;
			if sign=1 then
			DCBA:=-DCBA}
			else V:=1;
			cc(DCBA)

0 0 0 3 2 2#	CEQ	convert extended to quad	t:=63+256-exponent(A);
			sign:=D.<0>;
			if -2**63 <= DCBA <= 2**63-1
			then {D.<0>:=1;
			exponent(A):=0;
			DCBA:=DCBA´>>´t;
			if sign=1 then
			DCBA:=-DCBA}
			else V:=1;
			cc(DCBA)

0 0 0 3 2 3#	CEQR	convert extended to quad with rounding	t:=63+256-exponent(A);
			sign:=D.<0>;
			if -2**63 <= DCBA <= 2**63-1
			then {D.<0>:=1;
			exponent(A):=0;
			s:=0;
			DCBAs:=(DCBAs´>>´t)
			´+´%100000;
			if sign=1 then
			DCBA:=-DCBA}
			else V:=1;
			cc(DCBA)

0 0 0 3 2 4#	CQF	convert quad to floating	sign:=D.<0>; exp:=63+256;
			if sign=1 then
			DCBA:=-DCBA;
			if DCBA<>0 then
			{norm(DCBA);
			exponent(C):=exp;
			D.<0>:=sign}
			RP:=RP-2

0 0 0 3 2 5#	CFE	convert floating to extended	G:=exponent(A);
			exponent(A):=0;
			H:=0;
			RP:=RP+2

0 0 0 3 2 6#	CDFR	convert double to floating with rounding	sign:=B.<0>; exp:=31+256;
			if sign=1 then
			BA:=-BA;
			if BA<>0 then
			{norm(BA);
			BA:=BA´+´%400;
			if carry then
			exp:=exp+1;
			exponent(A):=exp;
			B.<0>:=sign}

0 0 0 3 2 7	CID	convert integer to double	H:=A; A := A>>15; RP:=RP+1

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 3 0#	CQFR	convert quad	sign:=D.<0>; exp:=63+256;
		to floating with	if sign=1 then
		rounding	DCBA:=-DCBA;
			if DCBA<>0 then
			{norm(DCBA);
			DC:=DC´+´%400;
			if carry then
			exp:=exp+1;
			exponent(C):=exp;
			D.<0>:=sign}
			RP:=RP-2
0 0 0 3 3 1#	CIF	convert integer	sign:=A.<0>; exp:=15+256;
		to floating	if sign=1 then A:=-A;
			if A<>0 then
			{norm(A);
			H:=exp;
			A.<0>:=sign}
			else H:=0;
			RP:=RP+1
0 0 0 3 3 2#	CIE	convert integer	sign:=A.<0>; exp:=15+256;
		to extended	if sign=1 then A:=-A;
			H:=G:=0;
			if A<>0 then
			{norm(A);
			F:=exp;
			A.<0>:=sign}
			else F:=0;
			RP:=RP+3
0 0 0 3 3 3	XSMX	checksum extended	while A<>0 do
		block	{D:=D xor xmem[CB];
		D=initial checksum	A:=A-1;
		CB=block address	CB:=CB+2};
		A=count	RP:=RP-3
0 0 0 3 3 4#	CDE	convert double	sign:=B.<0>; exp:=31+256;
		to extended	if sign=1 then BA:=-BA;
			H:=0;
			if BA<>0 then
			{norm(BA);
			G:=exp;
			B.<0>:=sign}
			else G:=0;
			RP:=RP+2
0 0 0 3 3 5#	CQER	convert quad	sign:=D.<0>; exp:=63+256;
		to extended with	if sign=1 then
		rounding	DCBA:=-DCBA;
			if DCBA<>0 then
			{norm(DCBA);
			DCBA:=DCBA´+´%400;
			if carry then
			exp:=exp+1;
			exponent(A):=exp;
			D.<0>:=sign}
0 0 0 3 3 6#	CQE	convert quad	sign:=D.<0>; exp:=63+256;
		to extended	if sign=1 then
			DCBA:=-DCBA;
			if DCBA<>0 then
			{norm(DCBA);
			exponent(A):=exp;
			D.<0>:=sign}

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 3 7#	CEI	convert extended to integer	t:=15+256-exponent(A);
			sign:=D.<0>;
			if -2**15 <= DCBA <= 2**15-1
			 then {D.<0>:=1;
			       D:=D´>>´t;
			       if sign=1 then D:=-D}
			 else V:=1;
			cc(D); RP:=RP-3

0 0 0 3 4 0			*** undefined ***
0 0 0 3 4 1			*** undefined ***
0 0 0 3 4 2	LWUC	load word from user code space	cc(A:=mem[2,A])

0 0 0 3 4 3	XSMG	checksum block
		
		C=initial checksum
		B=block address
		A=count
			while A<>0 do
			 {C:=C xor stack[B];
			  A:=A-1;
			  B:=B+1};
			RP:=RP-2

0 0 0 3 4 4#	IDX1	calculate index offset and test index bounds for 1 dimension
		
		(bounds table in code space)
			lower:=code[A];
			upper:=code[A+1];
			if B<lower then
			   {V:=1; cc(-1);
			    R[7]:=B}
			if B>upper then
			   {V:=1; cc(1);
			    R[7]:=B}
			if V=0 then
			  {R[7]:=B-lower;
			   cc(R[7])}
			RP:=RP-2

0 0 0 3 4 5#	IDX2	calculate index offset and test index bounds for 2 dimensions
		
		(bounds table in code space)
			lower:=code[A];
			upper:=code[A+1];
			if B<lower then
			   {V:=1; cc(-1);
			    R[7]:=B}
			if B>upper then
			   {V:=1; cc(1);
			    R[7]:=B}
			s:=upper-lower+1;
			B:=B-lower;
			lower:=code[A+2];
			upper:=code[A+3];
			if C<lower then
			   {V:=1; cc(-1);
			    R[7]:=C}
			if C>upper then
			   {V:=1; cc(1);
			    R[7]:=C}
			if V=0 then
			   {R[7]:=(C-lower)*s+B;
			    cc(R[7])}
			RP:=RP-3

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 4 6#	IDX3	calculate index
		offset and test
		index bounds
		for 3 dimensions
		
		(bounds table
		in code space)
			indv:=0; psize:=1;
			for i=1 to 3 by 1 do
			 {lower:=code[A];
			  upper:=code[A:=A+1];
			  if B<lower then
			    {V:=1;
			     cc(-1); R[7]:=B}
			  if B>upper then
			    {V:=1;
			     cc(1); R[7]:=B}
			  size:=upper-lower+1;
			  B:=B-lower;
			  indv:=indv+psize*B;
			  psize:=psize*size;
			  B:=A+1;
			  RP:=RP-1}
			if V=0 then
			  {R[7]:=indv;
			   cc(R[7])}
			RP:=RP-1

0 0 0 3 4 7#	IDXP	calculate index
		offset and test
		indices for
		bounds violation
		
		(bounds table
		in code space)
			t:=code[A];
			bc:=t.<0>; t.<0>:=0;
			indv:=0; psize:=1;
			s:=A;
			while t>0 do
			 {lower:=code[s:=s+1];
			  upper:=code[s:=s+1];
			  if B<lower and bc=0 then
			    {V:=1; t:=0;
			     cc(-1); R[7]:=B}
			  if B>upper and bc=0 then
			    {V:=1; t:=0;
			     cc(1); R[7]:=B}
			  size:=upper-lower+1;
			  B:=B-lower;
			  indv:=indv+psize*B;
			  psize:=psize*size;
			  RP:=RP-1; t:=t-1}
			if V=0 then
			  {R[7]:=indv;
			   cc(R[7])}
			RP:=RP-1

0 0 0 3 5 0	LWAS	load SG word via A	cc(A:=dest(A))
0 0 0 3 5 1	SWAS	stor SG word via A	dest(A):=B; RP:=RP-2

0 0 0 3 5 2	LDAS	load SG double
		via A
			RP:=RP+1;
			cc(BA:=dest(B:B+1))

0 0 0 3 5 3	SDAS	store SG double
		via A
			dest(A:A+1):=CB;
			RP:=RP-3;

0 0 0 3 5 4	LBAS	load SG byte via A	ccb(A:=bytedest(A))

0 0 0 3 5 5	SBAS	store SG byte
		via A
			bytedest(A):=B;
			RP:=RP-2

0 0 0 3 5 6	CDX	count duplicate
		words extended
		DC=buffer address
		B=buffer size
		A=duplicate count
			while B<>0 and
			 xmem[DC]=xmem[DC-2] do
			 {A:=A+1;
			  B:=B-1;
			  DC:=DC+2}

0 0 0 3 5 7	DFS	deposit field in
		SG memory
			cc(dest(A):=(dest(A) & ~B)
			                | (C & B));
			RP:=RP-3

0 0 0 3 6 0	LWA	load word via A	cc(A:=stack[A])
0 0 0 3 6 1	SWA	store word via A	stack[A]:=B; RP:=RP-2

0 0 0 3 6 2	LDA	load double via A	RP:=RP+1;
			cc(BA:=stack[B:B+1])

Table B-1. Instruction Set Definition (Continued)

0 0 0 3 6 3	SDA	store double via A	stack[A:A+1]:=CB;
			RP:=RP-3;

0 0 0 3 6 4	LBA	load byte via A	ccb(A:=bytedest(A))

0 0 0 3 6 5	SBA	store byte via A	bytedest(A):=B;
			RP:=RP-2

0 0 0 3 6 6	CDG	count duplicate words
		C=buffer address
		B=buffer size
		A=duplicate count
			while B<>0 and
			 stack[C]=stack[C-1] do
			 {A:=A+1;
			  B:=B-1;
			  C:=C+1}

0 0 0 3 6 7	DFG	deposit field in memory	cc(stack[A]:=(stack[A] & ~B)
			             | (C & B));
			RP:=RP-3

0 0 0 3 7 0 . . 0 0 0 3 7 7			*** undefined ***

0 0 0 4 0 0*	WWCS	write WCS
		D=WCS address
		C=buffer map
		B=buffer address
		
		A=ucode word count
			while A>0 do
			{WCS[D]:=mem[C,B]^mem[C,B+1]
			          ^mem[C,B+2].<0:3>;
			 if (A:=A-1)=0
			 then goto done;
			 D:=D+1;B:=B+2;
			 WCS[D]:=mem[C,B].<8:15>
			           ^mem[C,B+1]
			           ^mem[C,B+2].<0:11>;
			 D:=D+1; B:=B+3; A:=A-1;
			};
			done: N:=0; Z:=1; RP:=RP-4
			 !!! Note !!!
			all memory referenced must be present

0 0 0 4 0 1*	VWCS	verify WCS
		D=WCS address
		C=buffer map
		B=buffer address
		A=ucode word count
			N:=0;Z:=1;
			while Z and A>0 do
			{if WCS[D]<>mem[C,B]
			            ^mem[C,B+1]
			            ^mem[C,B+2].<0:3>
			 then {N:=1;Z:=0};
			 if N or (A:=A-1)=0
			 then goto done;
			 D:=D+1;B:=B+2;
			 if WCS[D]<>mem[C,B].<8:15>
			           ^mem[C,B+1]
			           ^mem[C,B+2].<0:11>
			 then {N:=1;Z:=0}
			 else {D:=D+1;B:=B+3;A:=A-1};
			};
			done: RP:=RP-4
			 !!! Note !!!
			all memory referenced must be present
			bus packets may not be received correctly while a VWCS is executing

Table B-1. Instruction Set Definition (Continued)

0 0 0 4 0 2*	RWCS	read WCS
		D=WCS address
		C=buffer map
		B=buffer address
		A=ucode word count
			while A>0 do
			{mem[C,B]^mem[C,B+1]
			  ^mem[C,B+2].<0:3>:=WCS[D];
			 if (A:=A-1)=0 then
			 then goto done;
			 D:=D+1;B:=B+2;
			 mem[C,B].<8:15>^mem[C,B+1]^
			  mem[C,B+2].<0:11>:=WCS[D];
			 D:=D+1;B:=B+3;A:=A-1};
			done: RP:=RP-4
			 !!! Note !!!
			all memory referenced must be present

0 0 0 4 0 3			*** undefined ***
0 0 0 4 0 4*	SMBP	set memory brkpt B.<0>=read flag .<1>=execute flag .<2>=write flag .<9:15>=high- order addr A=low-order addr	breakpointmode:=B.<0:2>; breakpointaddress:= B.<9:15>^A; BPADDR:=BA; RP:=RP-2; !!! Note !!! the address is a physical memory address any and all combinations of access flags may be set BA=0D will disable the trap
0 0 0 4 0 5*	FRST	firmware reset	reset and stop instruction execution
0 0 0 4 0 6	LBX	load byte extended	ccb(B:=bxmem[BA]);RP:=RP-1
0 0 0 4 0 7	SBX	store byte extnd.	bxmem[BA]:=C; RP:=RP-3
0 0 0 4 1 0	LWX	load word extended	cc(B:=xmem[BA]);RP:=RP-1
0 0 0 4 1 1	SWX	store word extnd.	xmem[BA]:=C; RP:=RP-3
0 0 0 4 1 2	LDDX	load double extnd.	cc(BA:=xmem[BA:BA+3])
0 0 0 4 1 3	SDDX	store dbl. extnd.	xmem[BA:BA+3]:=DC;RP:=RP-4

0 0 0 4 1 4	LQX	load quad extended	RP:=RP+2;
			cc(DCBA:=xmem[DC:DC+7])

0 0 0 4 1 5	SQX	store quad
		extended
			xmem[BA:BA+7]:=FEDC;
			RP:=RP-6

0 0 0 4 1 6	DFX	deposit field
		extended
			cc(xmem[BA]:=(xmem[BA] &
			        ~C | (D & C)));
			RP:=RP-4;

0 0 0 4 1 7	MVBX	move bytes
		extended
		ED=destination
		  address
		CB=source address
		A=byte count
			while A<>0 do
			 {bxmem[ED]:=bxmem[CB];
			  ED:=ED+1;
			  CB:=CB+1;
			  A:=A-1;};
			RP:=RP-5;

0 0 0 4 2 0	MBXR	move bytes
		extended reverse
		ED=destination
		  address
		CB=source address
		A=byte count
			while A<>0 do
			 {bxmem[ED]:=bxmem[CB];
			  ED:=ED-1;
			  CB:=CB-1;
			  A:=A-1;};
			RP:=RP-5;

0 0 0 4 2 1	MBXX	move bytes extnd.
		and checksum
		F=initial xsum
		ED=destination
		  address
		CB=source address
		A=byte count
			while A<>0 do
			 {bxmem[ED]:=t:=bxmem[CB];
			  F:=F xor t;
			  ED:=ED+1;
			  CB:=CB+1;
			  A:=A-1;};
			RP:=RP-5

Table B-1. Instruction Set Definition (Continued)

0 0 0 4 2 2	CMBX	compare bytes extended
		ED=destination address
		CB=source address
		A=byte count
			N:=0; Z:=1;
			while Z and A<>0 do
			{cc(bxmem[ED]:bxmem[CB]);
			if Z then
			{A:=A-1;ED:=ED+1;
			CB:=CB+1;}};
			RP:=RP-5

0 0 0 4 2 3*	CRAX	convert rel. to abs. ext. address	if B.<0:14>=0 then
			{B.<0:14>:=CMSEG[DS]}
			else if B.<0:14><=2 then
			{B.<0:14>:=CMSEG[B.<0:14>]}
			else if B.<0:14>=3 then
			{B.<0:14>:=CMSEG[cmap]}
			else if B.<0>=0 then
			{BA:=BA+segment base};
			B.<0>:=1;

0 0 0 4 2 4*	RSPT	read segment page table entry
		BA=ext. address
			xa:=CRAX(BA);
			p:=xa.<15:20>;
			s:=xa.<2:14>;
			K:=0;
			if MAP[15,p mod 32+32]
			= s^p.<10> then
			{B:=MAP[15,p mod 32]}
			else
			{if SEG[s*2].<0>=0 then
			B:=MAP[SEG[s*2].<0:4>,p]
			else
			{if p<SEG[s*2].<9:15> then
			B:=mem[SEG[s*2].<5:8>,
			SEG[s*2+1]+p]
			else {B:=1; K:=1}}};
			RP:=RP-1

0 0 0 4 2 5*	WSPT	write segment page table entry
		C=entry
		BA=ext. address
			xa:=CRAX(BA);
			p:=xa.<15:20>;
			s:=xa.<2:14>;
			if MAP[15,p mod 32+32]
			= s^p.<10> then
			{MAP[15,p mod 32]:=C}
			else
			{if SEG[s*2].<0>=0 then
			MAP[SEG[s*2].<0:4>,p]:=C
			else
			mem[SEG[s*2].<5:8>,
			SEG[s*2+1]+p]:=C};
			RP:=RP-3;

0 0 0 4 2 6*	RXBL	read extended base and limit	RP:=RP+4;
			DCBA:=MAP[14,60:63]

0 0 0 4 2 7*	SXBL	set extended base and limit	MAP[14,60:63]:=DCBA;
			RP:=RP-4

0 0 0 4 3 0*	LCKX	lock down extended memory
		D.<0>=lock only if already locked
		C=lock count
		BA=ext. address
			m:=RSPT(BA);
			p:=m.<0:12>;
			if m.<15>=0 and (D.<0>=0
			or PHYSEG[p]<0) then
			{if PHYSEG[p] < 0
			then
			{PHYSEG[p]:=PHYSEG[p]-C;
			K := 0}
			else
			{PHYSEG[p]:=-C;
			K := 1}
			Z:=1; N:=0}
			else {Z:=0; N:=1};
			RP:=RP-4

Table B-1. Instruction Set Definition (Continued)

0 0	0	4	3	1*	ULKX	unlock extended memory D=map entry mask C=unlock count BA=ext. address	m:=RSPT(xa:=CRAX(BA)); p:=m.<0:12>; if m.<15>=0 and (x:=PHYSEG[p]+C)<=0 then {if x<>0 then PHYSEG[p]:=x else {PHYSEG[p]:=xa.<2:14>; WSPT(BA, m&D)}; ccz(x)} else {Z:=0; N:=1}; RP:=RP-4
0 0	0	4	3	2*	CMRW	CME read/write B.<0:3>=map A=word address	N:=0;Z:=1; if I/O locked out then {mem[B.<0:3>,A] :=mem[B.<0:3>,A]; free I/O channel; if CME interrupt then Z:=0} else {N:=1; Z:=0}; RP:=RP-2
0 0	0	4	3	3			*** undefined ***
0 0	0	4	3	4*	RMEM	read mem	cc(B:=mem[B.<0:3>,A]); RP:=RP-1
0 0	0	4	3	5*	WMEM	write mem	mem[B.<0:3>,A]:=C; RP:=RP-3
0 0	0	4	3	6*	RSMT	read from OSP	enable read from OSP
0 0	0	4	3	7*	WSMT	write to OSP	write first character to OSP
0 0	0	4	4	0*	RIBA	read INTB and INTA registers	RP:=RP+2; B:=INTB; A:=INTA
0 0	0	4	4	1*	SVMP	save map entries	m:=word:=0; while word<%2000 do {memory[2,word]:= MAP[m.<12:15>,m.<0:5>] m:=m+%2000; if alu carry then m:=m+1; word:=word+1}
0 0	0	4	4	2*	XSTR	XRAY start timer D=disable flag C=offset to cntr BA=extended addr of XRAY ptr	if (t:=xmem[BA])<>0 then {a:=%40000^(t+C)^0; if xmem[a]<>D then {xmem[a]:=xmem[a]+1; a:=a+2; if (a+7).<0:5> <> a.<0:5> then Instruction Failure; xmem[a:a+7]:=xmem[a:a+7] -sysstack[%103:%106] -microsecond counter}}; RP:=RP-4
0 0	0	4	4	3*	XSTP	XRAY stop timer D=disable flag C=offset to cntr BA=extended addr of XRAY ptr	if (t:=xmem[BA])<>0 then {a:=%40000^(t+C)^0; if xmem[a]<>D then {xmem[a]:=xmem[a]-1; a:=a+2; if (a+7).<0:5> <> a.<0:5> then Instruction Failure; xmem[a:a+7]:=xmem[a:a+7] +sysstack[%103:%106] +microsecond counter}}; RP:=RP-4
0 0	0	4	4	4	SCS	set code segment BA=byte address in current code	if ENV.CS=1 or ENV.LS=1 then B.<0:14>:=3 else B.<0:14>:=2;
0 0	0	4	4	5*	LQAS	load SG quad via A	RP:=RP+3; cc(DCBA:=sysstack[A:A+3])

Table B-1. Instruction Set Definition (Continued)

0	0	0	4	4	6*	SQAS	store SG quad via	sysstack[A:A+3]:=EDCB;
							A	RP:=RP-5
0	0	0	4	4	7*	RCHN	reset I/O channel	if i/o channel available then
								{if A>=0
								then channel ioreset
								else channel lockup
								at %0777;
								N:=0; Z:=1}
								else {N:=1; Z:=0};
								RP:=RP-1
0	0	0	4	5	0*	BNDW	bounds test words	if A ´>´ L then
								cc(C:=1)
								else
								if B=0 or (C´<=´L-A and
							C=word address in	C+B-1´<=´L-A and C´<=´C+B-1)
							stack	or (C´>´L+350 and
							B=buffer size in	C´<=´C+B-1 and
							words	(C+B-1).<0:5> <
							A=number of words	SEG[CMSEG[0]*2].<9:15>)
							of parameters	then cc(C:=0)
							and stack marker	else cc(C:=1);
								RP:=RP-2
0	0	0	4	5	1	BPT	instruction	if ENV.<1> = 0
							breakpoint trap	then interrupt via SIV #19
								ENV.<1> := 0;
								i:=BPBASE;
								do
								{if sysstack[i]=CMSEG[cmap]
								and sysstack[i+1]=P-1
								then {I:=sysstack[i+2];
								roma:=EPT[I]};
								i:=i+BPSIZE}
								until i ´>´ BPLIM;
								Instruction failure
0	0	0	4	5	2*	BCLD	bus cold load	simulate a bus cold load
								from the panel
0	0	0	4	5	3*	TPEF	test parity error	RP:=RP+1;
							freeze circuits	A := if IPU error then 1
								else if MCB error then 2
								else if CCD error then 3
								else 0
0	0	0	4	5	4	SCMP	set code map	if A.<0:3>=0
								then A.<0:3>:=cmap
0	0	0	4	5	5			
			.					*** undefined ***
			.					
0	0	0	7	7	7			
0	0	1	-	-	-	CMPI	compare immediate	cc(A:imm); RP:=RP-1;
0	0	2	-	-	-	ADDS	add to S	S:=S+imm
0	0	3	-	-	-	LADI	logical add	ccl(A:=A´+´imm)
							immediate	
0	0	4	0--	-	-	ORRI	OR right immediate	cc(A:=A|I.<8:15>)
0	0	4	4--	-	-	ORLI	OR left immediate	cc(A:=A|(I.<8:15>´<<´8))
0	0	5	-	-	-	LDLI	load left	RP:=RP+1;
							immediate	cc(A:=imm rotate 8)
0	0	6	-	-	-	ANRI	AND right	cc(A:=A&imm)
							immediate	
0	0	7	-	-	-	ANLI	AND left immediate	cc(A:=A&(imm rotate 8))
1	0	0	-	-	-	LDI	load immediate	RP:=RP+1; cc(A:=imm)
1	0	0xx	-	-	-	LDXI	load x immediate	cc(X:=imm)
1	0	4	-	-	-	ADDI	add immediate	ccn(A:=A+imm)
1	0	4xx	-	-	-	ADXI	add x immediate	ccn(X:=X+imm)
1	1	0	0--	-	-	BIC	branch if carry	if K then branch

Table B-1. Instruction Set Definition (Continued)

I 1 1 0-- - -	BGTR	branch if greater	if ˜(N|Z) then branch
I 1 2 0-- - -	BEQL	branch if equal	if Z then branch
I 1 3 0-- - -	BGEQ	branch if greater or equal	if ˜ N then branch
I 1 4 0-- - -	BLSS	branch if less	if N then branch
I 1 5 0-- - -	BNEQ	branch if not equal	if ˜ Z then branch
I 1 6 0-- - -	BLEQ	branch if less or equal	if N|Z then branch
I 1 7 0-- - -	BNOC	branch no carry	if ˜ K then branch
I 1 0 4-- - -	BUN	branch unconditional	branch

I 1 0xx4-- - -	BOX	branch on X	if X<A then {X:=X+1; branch}
			else RP:=RP-1

I 1 4 4-- - -	BAZ	branch on A zero	if A=0 then branch; RP:=RP-1

I 1 5 4-- - -	BANZ	branch on A nonzero	if A<>0 then branch;
			RP:=RP-1

I 1 6 4-- - -	BNOV	branch if no overflow	if ˜ V then branch
I 1 7 4-- - -	BSUB	branch to subroutine	stack[S:=S+1]:=P; branch

I 2 0xx0-- - -	LWP	load word from program	RP:=RP+1;
			cc(A:=code[branchadr+X])

I 2 0xx4-- - -	LBP	load byte from program	RP:=RP+1;
			adr:=(if indirect then
			code[dba] else 0)
			+dba´<<´1+X;
			A:=code[adr.<0:14>
			+(dba&%100000)].
			<8*adr.<15>:8*adr.<15>+7>;
			ccb(A)

0 2 4 n r c	PUSH	push to stack	stack[S+1:S+c+1]
			:=R[(r-c)mod 8:r];
			RP:=n; S:=S+c+1

1 2 4 n r c	POP	pop from stack	R[(r-c)mod 8:r]
			:=stack[S-c:S];
			RP:=n; S:=S-c-1

0 2 5 0-- - -	RSUB	return from subroutine	P:=stack[S];
			S:=S-I.<8:15>

1 2 5 0-- - -	EXIT	exit procedure	(S,P,ENV,L):=(L-I.<8:15>, stack[L-2], (t:=stack[L-1])&ENV&%173000 | stack[L-1]&%4740 | ENV&%37, stack[L]); if t.<0> then Instruction Breakpoint
0 2 5 4 - - 0 2 6 4 - -	LWXX	load word extended indexed	cc(A:=xmem[A<<1+xbase])

0 2 5 5 - -
0 2 6 5 - -
	SWXX	store word extnded indexed	xmem[A<<1+xbase]:=B;
			RP:=RP-2

0 2 5 6 - - 0 2 6 6 - -	LBXX	load byte extended indexed	ccb(A:=bxmem[A+xbase])

0 2 5 7 - -
0 2 6 7 - -
	SBXX	store byte extnded indexed	bxmem[A+xbase]:=B;
			RP:=RP-2

1 2 5 4-- - -			*** undefined ***

0 2 6 00mssd n	MOVW	move words	while A>0 do
			{dest(C):=source(B);
			A:=A-1; B:=B+movestep;
			C:=C+movestep};
			RP:=n

Table B-1. Instruction Set Definition (Continued)

0 2 6 02mssd n	COMW	compare words	N:=0; Z:=1;
			while Z and A>0 do
			{cc(dest(C)´:´source(B));
			if Z then
			{A:=A-1; B:=B+movestep;
			C:=C+movestep}};
			RP:=n

1 2 6 00mssd n	MOVB	move bytes	while A>0 do
			{bytedest(C):=bytesource(B);
			A:=A-1; B:=B+movestep;
			C:=C+movestep};
			RP:=n

1 2 6 02mssd n	COMB	compare bytes	N:=0; Z:=1;
			while Z and A>0 do
			{cc(bytedest(C):
			bytesource(B));
			if Z then
			{A:=A-1; B:=B+movestep;
			C:=C+movestep}};
			RP:=n

1 2 6 40mssd n	SBW	scan bytes while	while bytesource(B)<>0 and
			bytesource(B)=A do
			B:=B+movestep
			K:=bytesource(B)=0; RP:=n

1 2 6 42mssd n	SBU	scan bytes until	while bytesource(B)<>0 and
			bytesource(B)<>A do
			B:=B+movestep
			K:=bytesource(B)=0; RP:=n

0 2 7 - - -	PCAL	procedure call	stack[S+1:S+3]:=(P,ENV,L);
			t:=I.<7:15>;
			if ˜ PRIV then
			{if t>=code[0] then
			{if t>=code[1]
			then priv trap;
			PRIV:=1}};
			L:=S:=S+3;
			P:=code[t]; RP:=7

1 2 7 - - -	XCAL	external call	if CMSEG[CMAP] = -1 then
			priv trap;
			stack[S+1:S+3]:=(P,ENV,L);
			i:=SEG[CMSEG[CMAP]*2]
			.<9:15>*%2000-1;
			m:=((code[i-I.<7:15>].<0:3>
			-2) mod 4)+2;
			t:=code[i-I.<7:15>].<7:15>;
			if ˜ PRIV then
			{if t>=mem[m,0] then
			{if t>=mem[m,1]
			then priv trap;
			PRIV:=1}};
			L:=S:=S+3;
			LS:=(m-2)/2;
			CS:=m.<15>;
			P:=code[t]; RP:=7

0 3 0 0 - -	LLS	logical left shift	computeshiftcount;
			cc(A:=A´<<´shiftcount)

0 3 0 1 - -	LRS	logical right shift	computeshiftcount;
			cc(A:=A´>>´shiftcount)

0 3 0 2 - -	ALS	arithmetic left shift	computeshiftcount;
			cc(A:=A<<shiftcount)

0 3 0 3 - -	ARS	arithmetic right shift	computeshiftcount;
			cc(A:=A>>shiftcount)

0 3 0 4-- - -			*** undefined ***

Table B-1. Instruction Set Definition (Continued)

1	3	0	0	-	-	DLLS	double logical left shift	computeshiftcount; cc(BA:=BA´<<´shiftcount)
1	3	0	1	-	-	DLRS	double logical right shift	computeshiftcount; cc(BA:=BA´>>´shiftcount)
1	3	0	2	-	-	DALS	double arithmetic left shift	computeshiftcount; cc(BA:=BA<<shiftcount)
1	3	0	3	-	-	DARS	double arithmetic right shift	computeshiftcount; cc(BA:=BA>>shiftcount)
1	3	0	4--	-	-			*** undefined ***
I	3	0xx	-	-	-	LDX	load X	cc(X:=word)
I	3	4xx	-	-	-	NSTO	nondestructive store	wordx:=A
I	4	0xx	-	-	-	LOAD	load	RP:=RP+1; cc(A:=wordx)
I	4	4xx	-	-	-	STOR	store	wordx:=A; RP:=RP-1
I	5	0xx	-	-	-	LDB	load byte	RP:=RP+1; ccb(A:=bytex)
I	5	4xx	-	-	-	STB	store byte	bytex:=A.<8:15>; RP:=RP-1
I	6	0xx	-	-	-	LDD	load double	RP:=RP+2; cc(BA:=dwordx)
I	6	4xx	-	-	-	STD	store double	dwordx:=BA; RP:=RP-2
I	7	0xx	-	-	-	LADR	load address	RP:=RP+1; A:=address+X
I	7	4xx	-	-	-	ADM	add to memory	ccn(wordx:=wordx+A); RP:=RP-1

Table B-2. Definitions of Symbols

x&y=	bitwise "and" of x and y
x|y=	bitwise "or" of x and y
x xor y=	bitwise "exclusive or" of x and y
x mod y=	x modulo y
~ x=	bitwise "complement" of x
x<<n=	x arithmetically shifted left n bits
x>>n=	x arithmetically shifted right n bits
x´<<´n=	x logically shifted left n bits
x´>>´n=	x logically shifted right n bits
x rotate n=	x´<<´n + x.<0:n-1>
x:y=	if x<y then -1 else if x=y then 0 else 1
x´<´y=	comparison of x and y as 16-bit unsigned numbers
x´:´y=	if x´<´y then -1 else if x=y then 0 else 1
x max y=	if x>y then x else y
x:=:y=	exchange x and y
x^y=	concatenate x and y
A=	R[RP]
address=	if indirect then mem[memmap, dir.adr.] else dir.adr.
B=	R[RP-1]
BA.<0:31>=	B.<0:15>^A.<0:15>
binq[bus,la]=	INQ[bus, la.<0:14>].byteflag
boq[bus,la]=	OUTQ[bus, la.<0:14>].byteflag
BPADDR=	sysstack[%115:%116]
BPBASE=	sysstack[%123]
BPLIM=	sysstack[%125]
BPSIZE=	sysstack[%124]
branch=	P:=branch address
branch address=	if indirect then code[dba] + dba else dba
BRT=	sysstack[%1400:%1777]
bxmem[xaddr]=	the byte at xaddr
byteaddress=	if indirect then mem[memmap,dir.adr.]+X else 2*dir.adr.+X
bytedest[la]=	mem[destmap,la.<0:14>].byteflag
byteflag=	<8*la.<15>:8*la.<15>+7>
bytesource[la]=	mem[srcmap, la.<0:14>+ (I.<10:11>=2)*P.<0>*%100000].byteflag
bytex=	mem[memmap, byteaddress.<0:14>].byteflag
C=	R[RP-2]
CB.<0:31>=	C.<0:15>^B.<0:15>
cc(x)=	Z:=(x=0); N:=(x<0)
ccb(x)=	Z:=("A"<=x<="Z") or ("a"<=x<="z"); N:=("0"<=x<="9")
CCE=	N:=0; Z:=1
CCG=	N:=0; Z:=0
CCL=	N:=1; Z:=0
ccl(x)=	cc(x); K:=adder carry
ccn(x)=	ccl(x); V:=adder overflow
chkp(x)=	if memory location "x" is absent then Page Fault
CLOCK=	sysstack[%103:%106]
cmap=	LS*2+CS+2
CMSEG=	sysstack[%1340:%1357]
code[la]=	mem[cmap, la]
computeshiftcount=	if I.<10:15>=0 then {shiftcount:=A.<8:15>; RP:=RP-1} else shiftcount:=I.<10:15>
CPCB=	sysstack[%3]
CS=	ENV.<7>
ccz(x)=	Z:=(x=0); N:=0;
D=	R[RP-3]
dba=	P+I.<9:15>-128*I.<8>
DC.<0:31>=	D.<0:15>^C.<0:15>
DCBA.<0:63>=	D.<0:15>^C.<0:15>^B.<0:15>^A.<0:15>

Table B-2. Definitions of Symbols (Continued)

dest[la]=	mem[destmap, la]	
destmap=	if I.<12>&PRIV then 1 else DS	
dir.adr.=	if I.<7>=0 then I.<8:15>	'global variable' (0:255)
	else	
	if I.<8>=0 then L+I.<9:15>	'local variable' (0:127)
	else	
	if I.<9>=0 then I.<10:15>	'system global' (0:63)
	else	
	if I.<10>=0 then L-I.<11:15>	'procedure parameter' (0:31)
	else	
	S-I.<11:15>;	'subroutine parameter' (0:31)
DS=	ENV.<6>	
dwordx=	mem[memmap, address+2*X:address+2*X+1]	
E=	R[RP-4]	
ED.<0:31>=	E.<0:15>^D.<0:15>	
ENV.<0:15>=	environment register	
EPT=	entry point table for instruction decoding	
extended address=	segment ^ page ^ word ^ byte	
F=	R[RP-5]	
FE.<0:31>=	F.<0:15>^E.<0:15>	
G=	R[RP-6]	
H=	R[RP-7]	
HGFE.<0:63>=	H.<0:15>^G.<0:15>^F.<0:15>^E.<0:15>	
I.<0:15>=	instruction register	
imm=	I.<8:15>-256*I.<7>	
indirect=	I.<0>	
INQ[0:1,0:15].<0:15>=	interprocessor bus in queues	
INTA.<0:15>=	interrupt register A	
INTB.<0:15>=	interrupt register B	
IOC=	sysstack[%2000:%3777]	
K=	ENV.<9>	
L.<0:15>=	local data pointer=location of current stack marker	
LIGHTS.<0:15>=	switch register output	
LS=	ENV.<4>	
MAP[0:15,0:63].<0:15>=	memory map	
MASK.<0:15>=	interrupt mask register	
mem[m,a]=	MEMORY[MAP[m,a.<0:5>].<0:12>, a.<6:15>]	
memmap=	if I.<7:9>=6 and PRIV then 1 else DS	
MEMORY[0:8191,0:1023].<0:15>=	physical memory	
movestep=	if I.<9> then -1 else 1	
N=	ENV.<11>	
OUTQ[0:1,0:15].<0:15>=	interprocessor bus out queues	
P.<0:15>=	program counter=1+location of current instruction	
PHYPAGE=	mem[%16, %150000:%167777]	
PHYSEG=	mem[%16, %130000:%147777]	
PRIV=	ENV.<5>	
PRIV TRAP=	cause an instruction failure interrupt	
RLIST=	sysstack[%100:%101]	
roma=	program counter for instruction microprocessor	
RP=	ENV.<13:15>	

Table B-2. Definitions of Symbols (Continued)

S.<0:15>=	stack pointer=location of last word of stack	
SD=	IPU scratch pad register. When the IPU is in the idle loop, it will indicate the reason:	
	%000000	HALT instruction
	%000014	bus cold load sequence error
	%000040	manual reset
	%000053	SFRZ instruction
	%000100	DDT halt interrupt
	%000115	OSP memory access breakpoint
	%000200	halt interrupt
	%000377	bus cold load checksum error
	%001000	i/o channel timeout on a cold load
	%001154	memory dump completed
	%002000	power-on interrupt with invalid memory
	%177772	illegal cold load switch setting
	%177773	i/o channel timeout on a tape dump
	%177774	error during memory dump to tape
	%177775	interrupt during memory dump to interprocessor bus
	%177776	uncorrectable memory error during map recovery following a power-on
	%177777	spurious interrupt
SEG=	mem[14, %70000:%127777]	
segment base=	MAP[14, 60:61]	
segment limit=	MAP[14, 62:63]	
SIV=	sysstack[%1200:%1337]	
source[la]=	mem[srcmap, la]	
srcmap=	if I.<10> then {if I.<11> then 2 else cmap} else if I.<11>&PRIV then 1 else DS	
stack[la] =	mem[DS, la]	
SWITCHES.<0:15>=	switch register input	
sysstack[la]=	mem[1, la]	
T=	ENV.<8>	
TLIST=	sysstack[%107:%110]	
TRACE=	sysstack[%121]	
TRBASE=	sysstack[%117]	
TRLIM=	sysstack[%120]	
UC=	ENV.<0>	
V=	ENV.<10>	
word=	mem[memmap, address]	
wordx=	mem[memmap, address+X]	
X=	if I.<5:6>=0 then 0 else R[I.<5:6>+4]	
xaddr.<0:31>=	a 32-bit extended address	
xbase=	stack[L*I.<5>+I.<10:15> : L*I.<5>+I.<10:15>+1]	
xmem[xaddr]=	the word located at xaddr	
Z=	ENV.<12>	

INDEX

◀ FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 482 CUPERTINO, CA. U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

TANDEM
COMPUTERS

19333 Vallco Parkway
Cupertino, CA U.S.A. 95014
Attn: Technical Communications—Software

◀ FOLD

STAPLE HERE

READER'S COMMENTS

Tandem welcomes your feedback on the quality and usefulness of its publications. Please indicat a specific *section* and *page* number when commenting on any manual. Does this manual have th desired completeness and flow of organization? Are the examples clear and useful? Is it easil understood? Does it have obvious errors? Are helpful additions needed?

Title of manual(s): ____________________

FOLD ➤

FOLD ➤

FROM:

Name ____________________

Company ____________________

Address ____________________

City/State ____________________ Zip __________

A written response is requested yes no

82077 B00

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, CA 95014