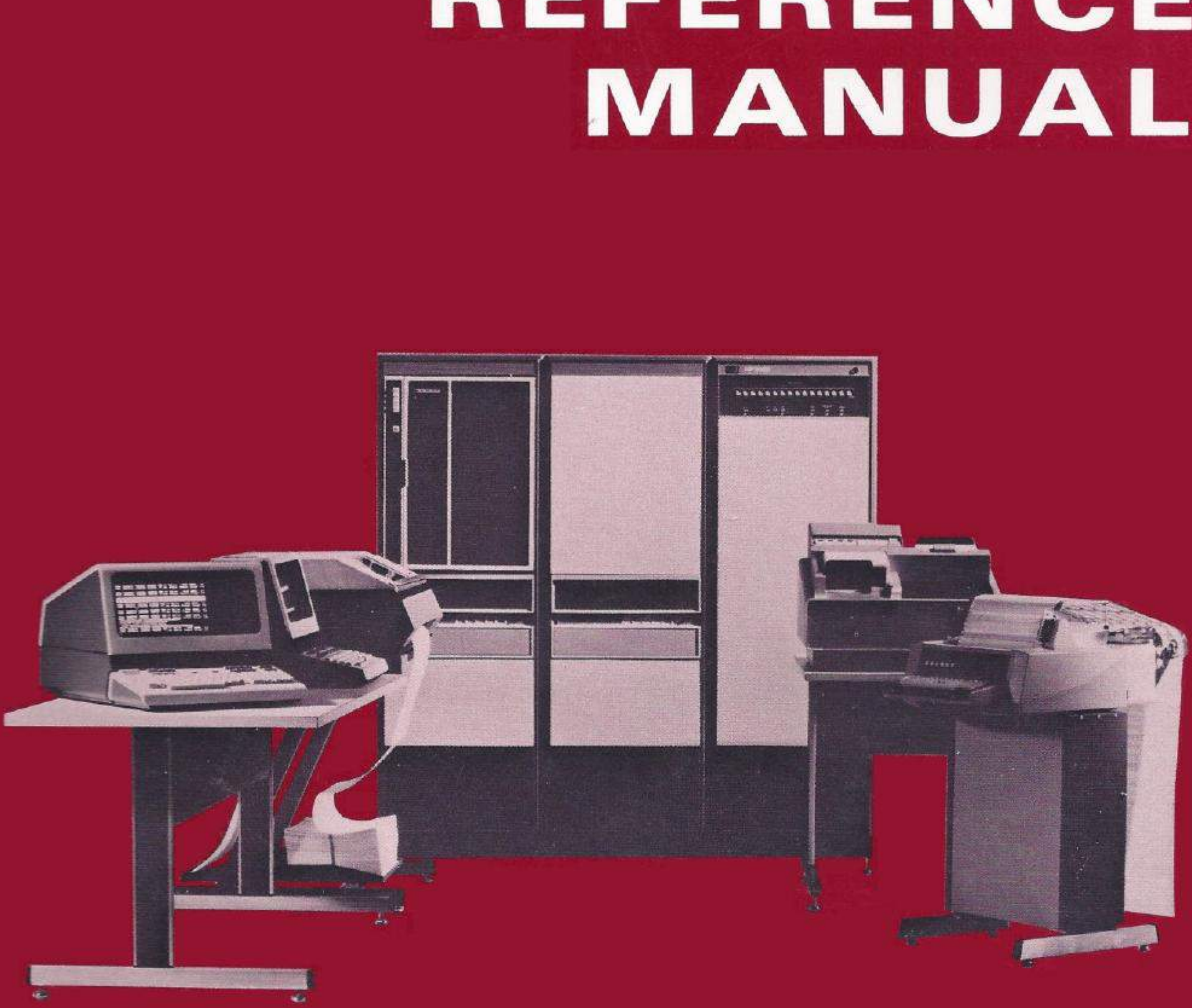
HEWLETT hp PACKARD

# HP 3000 COMPUTER SYSTEM

# REFERENCE MANUAL



HEWLETT PACKARD

# HP 3000 COMPUTER SYSTEM

## REFERENCE MANUAL

HP MANUAL PART NO. 03000-90019
MICROFICHE PART NO. 03000-90076

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, U.S.A.

Printed: SEPT 1973

# PREFACE

This is a hardware reference manual. Conceptually, however, the HP 3000 system is designed as an integrated system of hardware *and* software. This manual, therefore, must be regarded as a system description *from the hardware standpoint.*

The progress of computer technology inherently increases the complexity of hardware even as the use of that hardware becomes simpler and more convenient. Although the HP 3000 is not a large-scale computer, its hardware is nonetheless complex. In recognition of this fact, this manual has been made as conversational and illustrative as possible.

An INDEX OF TERMS is given in the Appendix. A page number refers to the first time a given term is used or defined in the text; the term is *italicized* on that page.

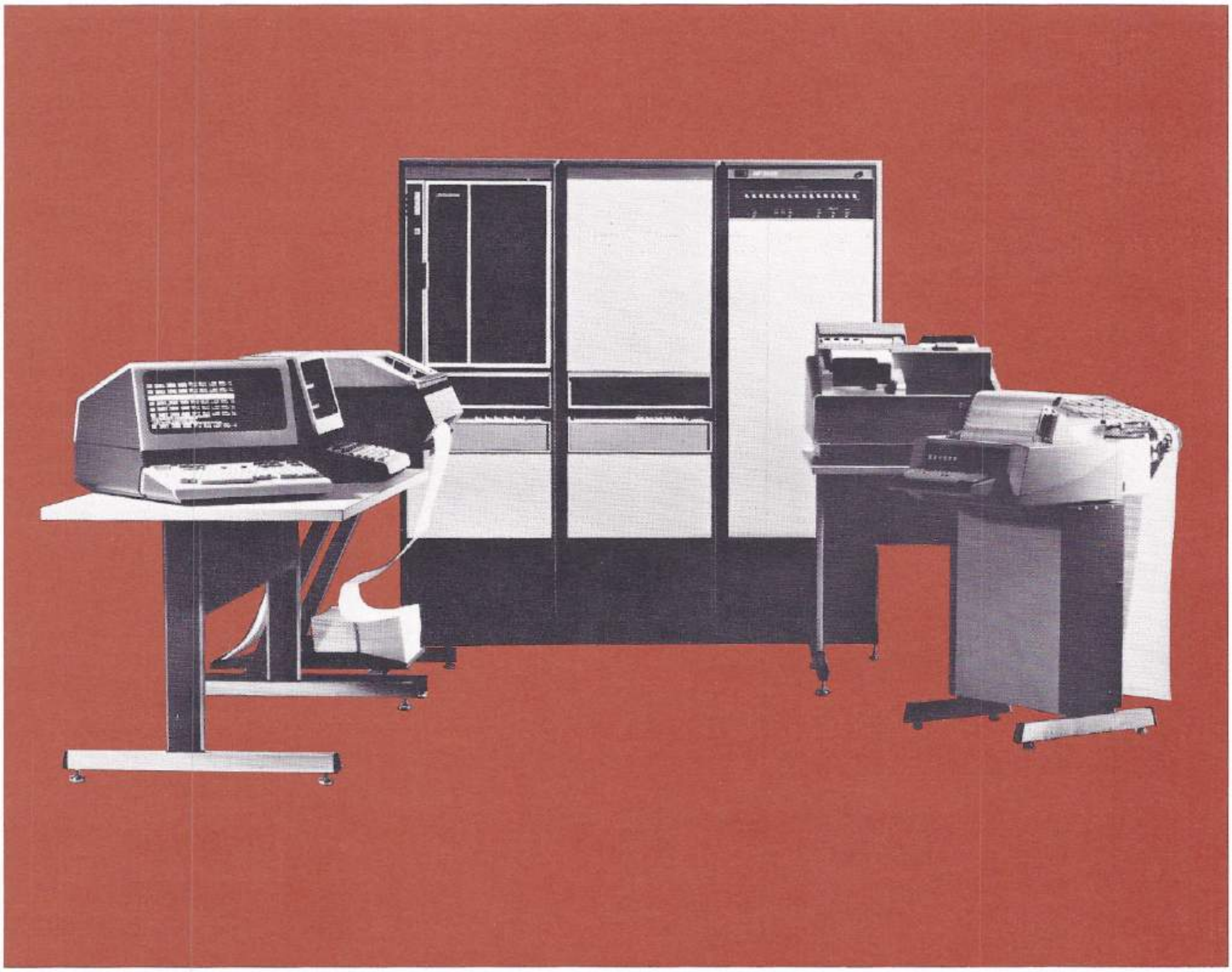
# CONTENTS

## ILLUSTRATIONS

## TABLES

# HP 3000

## COMPUTER SYSTEM



# INDEX OF HP 3000 MACHINE INSTRUCTIONS

## SINGLE WORD SHIFT INSTRUCTIONS

| | | |
|---|---|---|
| ASL | Arithmetic shift left | 5-9 |
| ASR | Arithmetic shift right | 5-9 |
| CSL | Circular shift left | 5-10 |
| CSR | Circular shift right | 5-10 |
| LSL | Logical shift left | 5-9 |
| LSR | Logical shift right | 5-9 |

## DOUBLE WORD SHIFT INSTRUCTIONS

| | | |
|---|---|---|
| DASL | Double arithmetic shift left | 5-10 |
| DASR | Double arithmetic shift right | 5-10 |
| DCSL | Double circular shift left | 5-11 |
| DCSR | Double circular shift right | 5-11 |
| DLSL | Double logical shift left | 5-10 |
| DLSR | Double logical shift right | 5-10 |

## TRIPLE WORD SHIFT INSTRUCTIONS

| | | |
|---|---|---|
| TASL | Triple arithmetic shift left | 5-11 |
| TASR | Triple arithmetic shift right | 5-11 |
| TNSL | Triple normalizing shift left | 5-11 |

## BIT TEST INSTRUCTIONS

| | | |
|---|---|---|
| SCAN | Scan bits | 5-14 |
| TBC | Test specified bit and set CC | 5-14 |
| TCBC | Test and complement bit, set CC | 5-15 |
| TRBC | Test and reset bit, set CC | 5-14 |
| TSBC | Test and set bit, set CC | 5-15 |
| TSBM | Test and set bits in memory, set CC | 5-15 |

## FIELD INSTRUCTIONS

| | | |
|---|---|---|
| DPF | Deposit field, A bits to B | 5-22 |
| EXF | Extract specified field, right-justify | 5-22 |

## I/O AND INTERRUPT INSTRUCTIONS

| | | |
|---|---|---|
| CIO* | Control I/O, direct | 5-27 |
| CMD* | Send command to module, direct | 5-27 |
| RIO* | Read I/O, direct | 5-26 |
| RMSK | Read device mask | 5-26 |
| SED* | Set enable/disable external interrupts | 5-25 |
| SIN* | Set interrupt | 5-28 |
| SIO* | Start I/O, block transfer | 5-26 |
| SIRF* | Set external interrupt reference flag | 5-28 |
| SMSK* | Set device mask | 5-26 |
| TIO* | Test I/O, direct | 5-27 |
| WIO* | Write I/O, direct | 5-27 |

## IMMEDIATE INSTRUCTIONS

| | | |
|---|---|---|
| ADDI | Add immediate to integer in A | 5-20 |
| ADXI | Add immediate to X | 5-21 |
| ANDI | Logical AND immediate with A | 5-22 |
| CMPI | Compare A with immediate, set CC | 5-20 |
| CMPN | Compare A with negative immediate | 5-21 |
| DIVI | Divide immediate into A | 5-21 |
| LDI | Load immediate to TOS | 5-20 |
| LDNI | Load negative immediate to TOS | 5-21 |
| LDXI | Load X immediate | 5-20 |
| LDXN | Load X negative immediate | 5-21 |
| MPYI | Multiply immediate with A | 5-20 |
| ORI | Logical OR immediate with A | 5-21 |
| SBXI | Subtract immediate from X | 5-21 |
| SUBI | Subtract immediate from A | 5-20 |
| XORI | Logical exclusive OR immediate | 5-22 |

## PROGRAM CONTROL INSTRUCTIONS

| | | |
|---|---|---|
| EXIT | Procedure and interrupt exit | 5-25 |
| HALT* | Halt, non-interruptable | 5-24 |
| PAUS* | Pause, interruptable | 5-24 |
| PCAL | Procedure call | 5-24 |
| SCAL | Subroutine call | 5-24 |
| SXIT | Subroutine exit | 5-25 |
| XEQ | Execute instruction in stack | 5-24 |

## REGISTER CONTROL INSTRUCTIONS

| | | |
|---|---|---|
| ADDS | Add operand to stack pointer | 5-23 |
| PSHR | Push specified registers onto stack | 5-22 |
| SETR* | Set specified registers from stack | 5-23 |
| SUBS | Subtract operand from stack pointer | 5-23 |
| XCHD* | Exchange DB and TOS | 5-23 |

## SPECIAL INSTRUCTIONS

| | | |
|---|---|---|
| LLBL | Load STT label onto stack | 5-19 |
| LLSH* | Linked list search | 5-19 |
| PLDA* | Privileged load from absolute address | 5-19 |
| PSTA* | Privileged store into absolute address | 5-19 |
| RSW | Read switch register | 5-19 |

## MOVE INSTRUCTIONS

| | | |
|---|---|---|
| CMPB | Compare bytes in two memory blocks, set CC | 5-18 |
| MOVE | Move words in memory, addresses +/- | 5-15 |
| MVB | Move bytes in memory, addresses +/- | 5-16 |
| MVBL* | Move words from DB+ to DL+ area | 5-16 |
| MVBW | Move bytes while of specified type | 5-18 |
| MVLB* | Move words from DL+ to DB+ area | 5-17 |
| SCU | Scan bytes until test or terminal byte | 5-17 |
| SCW | Scan bytes while equal to test byte | 5-17 |

| | | | | | |
|---|---|---|---|---|---|
| TOS | Top of stack | A | Top of stack | STT | Segment Transfer Table |
| * | Privileged instruction | B | Location below A | DB | Data Base |
| CC | Condition Code | C | Location below B | DL | Data Limit |
| X | Index Register | D | Location below C | | |

# SYSTEM FEATURES

The HP 3000 Computer System is a small-scale, disc-based system with true multiprogramming and multilingual capabilities. It is the first 16-bit computer system to incorporate such large-system features as a hardware stack architecture and variable-length code segmentation in a virtual memory scheme. As a result, the HP 3000 can simultaneously handle interactive and batch operations — each in more than one computer language.

These features have been achieved through an integrated hardware-software approach based on the specific demands of the multiprogramming environment. Hardware and software work together in an interrelated manner, with hardware performing many of the overhead operations that are conventionally done in software — such as environment changes on interrupt.

A powerful operating system optimizes multiprogrammed operations. System resources such as main memory storage, processor time, and peripherals are dynamically allocated to each user as needed. Each user on the system is independent and unaware of all other users; each "sees" only that part of the system required to solve his problem.

Hardware is organized on a modular basis. See figure 1-1 (shows four "modules"). Communication between modules occurs over a high-speed central data bus. Input/output data may be transferred directly to or from memory over the same bus, via a high-speed Selector Channel, or may be multiplexed via the I/O processor. In both cases the I/O channels execute I/O programs in parallel with CPU programs. Direct control of devices on the IOP bus is also possible by the CPU's direct I/O instructions. The configuration of either the module complement or the peripheral complement is easily changed to accommodate system expansions. Up to 7 modules and 253 I/O devices are possible in the hardware organization.

This section lists and describes the important hardware features of the HP 3000. Refer to Section II for a summary of software features.

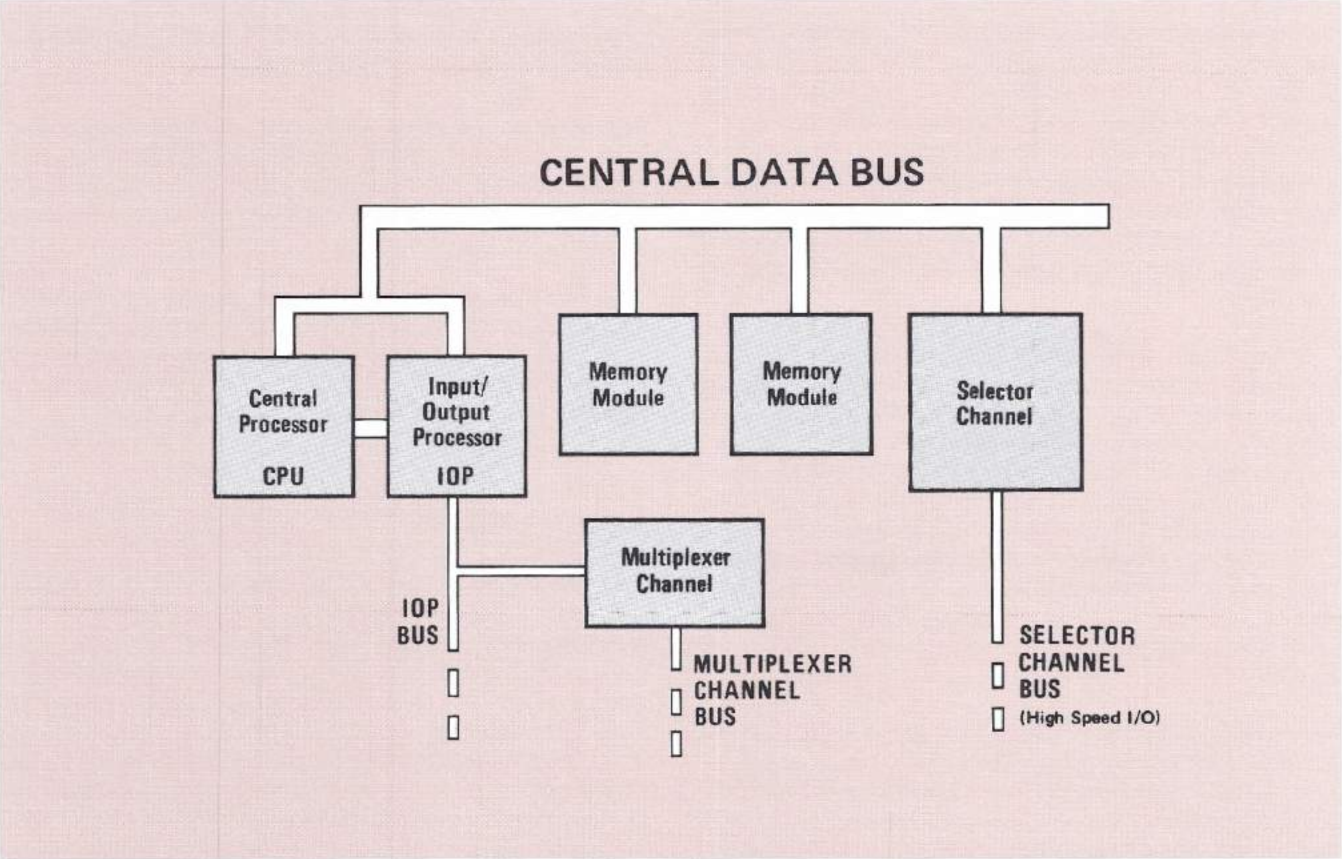


Figure 1-1. HP 3000 Modular Organization

# CENTRAL PROCESSOR

**ARCHITECTURE**

- Hardware-implemented stack
- Separation of code and data
- Non-modifiable, re-entrant code
- Variable-length code segmentation
- Virtual memory for code
- Dynamic relocatability of programs

**IMPLEMENTATION**

- Microprogrammed CPU
- 175 nanosecond microinstruction time
- Built-in memory protection, parity checking, power-fail/auto restart
- Protection between users
- Central data bus
- Concurrent I/O and CPU operations

**INSTRUCTIONS**

- 170 powerful instructions
- All instructions 16 bits in length
- 16- and 32-bit integer, 32-bit floating point hardware arithmetic
- Triple-word shifts to aid 48-bit floating point software

## ARCHITECTURE

The data for each user is organized as a data stack. In general, a stack is a storage area where the last item stored in is always the first item taken out. The stack structure provides an efficient mechanism for parameter passing, dynamic allocation of temporary storage, efficient evaluation of arithmetic expressions, and recursive subroutine or procedure calls. In addition, it enables rapid context switching — 21 microseconds to establish new environment on interrupt. In the HP 3000, all features of the stack (including checking for overflow and underflow) are implemented in hardware.

Code and data are maintained in strictly separate domains and cannot be intermixed (except that program constants may be present in code segments). This fact, plus the fact that code is non-modifiable while active in the system, permits code to be sharable and re-entrant. The two features, re-entrancy and stack-structured data, together make possible program recursion (a program calling itself) which is essential for efficient compilers and systems software. Also, since code is non-modifiable, exact copies of all active code can be retained on the swapping disc, thus allowing code to be overlayed without having to write it back out on the disc.

Variable-length segmentation of code and data is used to facilitate multiprogramming. This system, in comparison with paging schemes, minimizes "checkerboard" waste of memory resources due to internal fragmentation. The location and size of all active code segments is maintained in a Code Segment Table, known to both hardware and software. Software uses this table for dynamic memory management by the operating system. Hardware uses the table for procedure entry and exit. A similar table for data segments is known and managed by the software alone. Code segments may be up to 16,384 words in length. Data segments may be up to 32,768 words.

Segments are stored on a swapping disc and brought into main memory only when needed. This design results in a virtual memory which appears to be several times larger than the 65,536-word maximum size of the physical main memory.

All addressing of code and data is done relative to hardware address registers. Thus by simply changing the addresses, programs are dynamically relocatable in memory. The few instances where absolute addresses are required are privileged operations, handled by the operating system.

## IMPLEMENTATION

The entire instruction set of the HP 3000 is microprogrammed in a microprocessor within the CPU. The microprocessor executes each HP 3000 instruction by microprogrammed operations stored in an expandable read-only memory. By allowing microprogrammed hardware to execute certain repetitive functions such as moves and byte scans (normally software-implemented) the amount of code and total execution times are greatly reduced. In addition to the instruction set, other system functions have been microprogrammed, including the interrupt handler and a cold-start loader. The microprocessor executes its microinstructions at a 175-nanosecond rate.

The microprogrammed instructions routinely check for bounds violation during execution, and automatically interrupt to error handling routines if violations occur. Memory protection checks are usually overlapped with the operand fetch and therefore do not slow the execution. Three types of parity checking are provided: system parity error (checks validity of module numbers and commands during intermodule communication), memory address parity error, and data parity error. Power failure generates an interrupt to a Power Fail segment, and restoration of power generates an interrupt to a Power On segment for automatic restart. All of these features are standard in the HP 3000.

Several features contribute to the absolute privacy of each user's data and code. These include: microprogram checks for address bounds, file security guaranteed by the operating system, non-modifiable code, and uncallable code for system functions. Not only is each user protected from all others, but additionally the operating system is protected from all users.

The basic structure of independent modules organized around a central data bus permits high-speed internal data rates. A selector channel can transfer data in or out, via this bus, at rates up to 1.9 million bytes per second, using the currently available memory without interleaving. When not communicating over the bus, each module can run independently at its own speed. New equipment can also be added without having to go through a major system reconfiguration.

Another advantage of the modular structure is that it permits concurrent I/O and CPU operations, which are essential to multiprogramming and its usage of main memory.

## INSTRUCTIONS

There are 170 unique and meaningful instructions in the HP 3000 instruction set. Many of these have multiple actions which give a high complexity-to-instruction ratio. Code compression is achieved through the use of implicit no-address (stack) instructions, and the use of stack locations for operand addressing. All instructions except the 63 stack operations are in a 16-bit format; the stack ops may be packed two per word to further enhance the code density.

A complete set of arithmetic instructions provide integer (16-bit two's complement), double integer (32-bit two's complement), logical (16-bit positive integer), and floating point (32 bits including 23-bit precision mantissa) arithmetic. Special instructions like triple normalizing shift aid software implemented multiple precision floating point arithmetic.

Other special instructions are designated as privileged, meaning that they are usable only by the operating system or by users which the operating system permits to run in privileged mode.

# MEMORY

- Technology independent, speed independent
- One or two modules
- Interleaving provision
- Addressable to 64K words (131,072 bytes)
- 17 bits includes parity bit

Due to the modular construction of the HP 3000, memory modules are not restricted to specific characteristics, such as memory cycle time. Synchronous timing is required only when communicating with other modules over the central data bus. Thus memories may be of any type: low speed, or inexpensive, to high speed minimum access time — and may be mixed in the same system. Any currently offered memory technology (magnetic core, solid state, etc.) may be used and intermixed, and system updating is easily accomplished in the future as the state-of-the-art advances.

One or two memory modules may be used in the system. Modules are available for 32K, 48K, and 64K word configurations. In all cases the word length is 17 bits — 16 bits of data (one word or two bytes) and one parity bit. If two 32K word memory modules are present, two-way interleaving of memory addresses is accomplished by simply adjusting switches or jumpers within each memory module and in the CPU. Interleaving is then automatic, entirely dependent on the value of absolute memory addresses transmitted over the central data bus.

# I/O AND PERIPHERALS

### GENERAL

- Privileged control of I/O
- Concurrent I/O operations
- Three ways to implement I/O
- Direct memory access by all channels
- Device-independent I/O program execution
- Up to 253 devices
- Independent parameters for flexible I/O

### I/O SYSTEM

- Multiplexer channel
- Selector Channel
- Direct I/O

### INTERRUPT SYSTEM

- Up to 253 external interrupts
- Independent masking and priority structures
- Microprogrammed environment switching
- Common stack for interrupt processing
- Also 17 internal interrupts plus 7 traps

### PERIPHERALS

- Versatile mass storage units
- Card equipment
- Consoles/Terminals
- Line printers
- Punched tape equipment
- Data communications interfaces
- Add-ons supplied as complete I/O subsystems

## GENERAL

Input/output operations are defined as "privileged" operations by the HP 3000. Accordingly, I/O is normally performed for the user by the operating system, and the entire I/O system is not visible to the user. When the user asks to read a named file, he is only implicitly specifying the actual disc address of the file; the file system determines the explicit address for him from a disc file directory and performs the read. At another level, when a user asks the file system for a certain type of device by specifying a device class (e.g., magnetic tape, line printer, etc.), the file system takes care of allocating an actual device for the user. Users who must have actual contact with special devices (such as in real-time applications) are assigned their own device channels during system configuration and they bypass the file system.

All I/O devices can be operated concurrently (within system bandwidth). Peripherals that fail are taken off line by operator command.

There are three distinct means of implementing I/O. This results in efficient use of data paths in accordance with the capabilities of different peripheral devices. The three methods are described below under the heading "I/O System". Multiplexer channels are for medium to high speed synchronous and asynchronous I/O. Selector channels are for high speed synchronous I/O. Direct I/O is used for low speed asynchronous devices.

All selector and multiplexer channels have direct memory access. The CPU simply issues a "Start I/O" instruction to the device controller and the controller then assumes control of its own I/O program execution. The I/O program uses a unique set of commands (not related to the basic instruction set) to transfer information between memory and the external device. Note that once the device operation has been initiated, the CPU is free to continue processing. Both tasks run concurrently until the appropriate I/O command terminates the device transfer.

Since the initiating instruction, "Start I/O", is consistently the same for any type of device, programs can be written in a general, device-independent manner.

Device controllers are identified by a device number which is used to access the Device Reference Table (DRT). The DRT is known to both hardware and software. Since there can be a maximum of 253 entries in this table, the HP 3000 may have up to 253 devices in its I/O system. (Actual limitation is the 8-bit I/O address bus.)

In addition to a device number, there are two other characteristic numbers associated with each device. These are: data service priority and interrupt priority. Each of these values is completely independent of the others, and none is related to the physical location of devices or controllers. This mutual independence of characteristics provides the following advantages:

1. Device numbers can be assigned consecutively, starting at number 3 and proceeding up to the last assigned device in the system. When a new device is added, it is merely assigned the next higher available number (or any vacant number).
2. A new device added to the system may have its controller connected anywhere in the priority chain, independent of physical location within the cabinet.
3. Since data service priority and interrupt priority are independent of each other, a device which requires a high data transfer rate but interrupts infrequently (such as a disc) may be assigned a high data service priority but a low interrupt priority. Conversely, a device which has a low data rate but has an important interrupt significance (such as an alarm condition) may be configured to a high interrupt priority.

## I/O SYSTEM

Multiplexer Channel. Each multiplexer channel handles up to 16 devices. By multiplexing device inputs, cumulative data rates of 880,000 bytes per second are possible with the initial memory offered. Data from the multiplexer channel is applied directly to the I/O processor for transfer to memory via the central data bus.

Selector Channel. Selector channel data transfers bypass the I/O processor completely to provide for very high speed data transfer or additional I/O bandwidth. Transfer rates up to 1.9 million bytes per second for a single device are possible with non-interleaved memory. Up to eight device controllers can be handled by one selector channel; each device will complete its block transfer before another can be selected. The selector channel interface to the central data bus can accept two channels which can be handled simultaneously on a multiplexed basis.

Direct I/O. The HP 3000 instruction set includes four instructions for transferring information directly between I/O devices and the top of the stack in the CPU. These are: RIO (Read I/O), WIO (Write I/O), TIO (Test I/O), and CIO (Control I/O). Reading and writing is accomplished on a word-at-a-time basis, and would be used for low speed asynchronous devices. An Asynchronous Terminal Controller, which uses direct I/O, can handle 16 terminals at transfer rates up to 2400 baud.

## INTERRUPT SYSTEM

The interrupt system provides for up to 253 external interrupt levels. The priority level for each device is hardware determined when the system is configured. Interrupt priority is independent of data service priority. Interrupt priorities are easily changed by clip-on wires which determine the routing of the interrupt poll.

When interrupts occur, the microprogrammed interrupt handler automatically identifies each interrupt and grants control to the highest priority interrupt. Current operational status is saved by the microprogram, which then sets up the interrupt processing environment and transfers control to the interrupt routine. This microprogrammed context switching is performed in an average time of 21 microseconds (best case 18 microseconds, worst case 24.5 microseconds).

Interrupt routines operate on a common stack (Interrupt Control Stack) which is known to both hardware and software. This feature permits nesting of interrupt routines in the case of multiple interrupts, and further reduces environment switching time by about two microseconds if already operating on the Interrupt Control Stack.

The interrupt system also provides for 17 internal interrupts (for user errors, system violations, hardware faults, and power fail/restart) plus seven traps for arithmetic errors and illegal use of instructions.

## PERIPHERALS

Mass storage devices include both fixed- and moving-head disc files. The fixed-head disc files provide an average access time of only 8.7 milliseconds and a data transfer rate of 496,000 bytes per second. Such high-speed performance makes this device ideal for swapping disc files. Storage capacity is 2 or 4 million bytes per unit. For maximum flexibility and storage capacity, moving-head disc files are available. These units provide storage capacities from 5 to 50 million bytes and data transfer rates of up to 312,000 bytes per second. On-line storage can be expanded to more than one billion bytes. Low-cost magnetic tape units are available in 9-channel models. Recording densities are 800 or 1600 bpi at read/write speeds of 45 inches per second.

Two types of card readers operate at 600 and 1200 cards per minute. Card punches provide speeds of 250 cards per minute.

Reliable high-speed system communication is provided by either a 30 character per second (hard-copy output) terminal or a CRT display terminal. Standard ASR-33 equipment is also available for terminal use. A printer terminal is supplied as standard equipment for the system console.

Line printer output is generated at either 200 or 600 lines per minute. Both units provide 132-column print lines, using either 64 or 96 characters.

High-speed punched tape equipment reads at 500 characters per second. Punched tape output is available as a separate unit at 75 characters per second. Either paper, plastic, or mylar tape may be used with all units.

Also available are an asynchronous terminal controller and synchronous interfaces for data communications.

Hewlett-Packard furnishes available peripherals as complete I/O subsystems (including the device, interface, cables, etc.), to facilitate system expansion in the field.

# OPERATING ENVIRONMENTS

SECTION II

The HP 3000 hardware is typically accessible only through the operating system. Thus the user's operating environment is the HP 3000 system software.

Major contributions to the power of the HP 3000 system are provided by:

- A multiple-mode operating system
- A high-level systems programming language
- Standard programming languages
- Extensive user-aid software.

Because the HP 3000 is a multipurpose system, each user and function interfaces with the system at a level of sophistication appropriate to his own task. Each user runs in a protected environment free from interference by other users. Program protection is supplied by hardware, and file security is provided by software.

This section briefly describes the HP 3000 software under the four general categories. For more in-depth coverage, refer to the individual software reference manuals.

## OPERATING SYSTEM

The Multiprogramming Executive (MPE/3000) is the only operating system needed for the HP 3000, since it simultaneously manages both terminal and batch modes of operation.

Consistency and compatibility between operating modes are fundamental concepts of MPE/3000. Batch processing activities and interactive terminal users access the same software, and programs developed in the terminal mode may be utilized under batch mode to take advantage of available system peripherals.

Uniform access to disc files and standard input/output devices is accomplished through the File System. Files are accessed in two modes: sequential, which can have fixed or variable record lengths, or direct. Files are opened, operated on, and closed programmatically. Three levels of file security are selectable by the user.

The Dispatcher function allocates CPU time among programs in execution. All processes are entered into a master queue according to their priority. When execution has been interrupted (I/O, internal interrupt, time interrupt, etc.), CPU control is granted to the highest priority process ready to execute in main memory.

The memory management function dynamically allocates main memory space on a priority basis among contending users. Several programs can be active in memory concurrently. When a higher priority program must be serviced, the executing program is interrupted or overwritten (data is saved). Programs may be relocated anywhere in main memory and continue executing from the point of interruption.

BATCH MODE. Batch processing is the execution of user jobs that have been prepared on some input medium such as punched cards. Each job is self-contained and includes all necessary commands, programs, data, etc., for the operating system to use. No further instructions from the programmer are required during execution.

Several jobs can be submitted from one or more devices concurrently. Input jobs are organized in scheduling queues for execution, and when any executing program is suspended temporarily (e.g., waiting for input), MPE/3000 starts up the next highest priority job. Thus the system continues to operate at peak efficiency.

TERMINAL MODE. A user operating in terminal mode can be connected to the system either directly or through telephone lines. The user sitting at a keyboard terminal interacts with the system and receives immediate responses to his input. Since multiple terminals can be active at one time, they are processed through a time slicing technique (each active terminal is given an equal slice from each time period). MPE/3000 can continue to execute batch jobs at the same time as it is handling terminals.

Languages available to terminal users include HP extended FORTRAN, HP extended BASIC, COBOL, and the HP Systems Programming Language (SPL/3000).

## SYSTEMS PROGRAMMING LANGUAGE

Instead of the customary assembly language, a unique new language — developed especially for writing systems programs — has been designed for the HP 3000. This language is the Systems Programming Language, or SPL/3000. It is both a high-level language and a machine-dependent language, combining the best features of both.

The high-level features are provided by a powerful, procedure-oriented structure that is similar (but not equivalent) to the international language ALGOL. This structure permits efficient coding which can be written many times faster than if using standard assembly languages. Finished programs are self documenting, making programs easier to read as well as easier to write.

Machine-dependent features permit exact and efficient control of the hardware when such control is desired. The programmer can address hardware registers explicitly, extract and deposit variable bit fields, execute branches based on hardware status, and directly execute multi-function machine instructions such as SCAN, MOVE, PUSH and SET REGISTERS. An assemble statement effectively provides a built-in assembler.

Using both the high-level and machine-dependent features of SPL/3000, program execution times are very close to times achievable by comparable assembly language coding.

## STANDARD PROGRAMMING LANGUAGES

FORTRAN/3000. The FORTRAN compiler for the HP 3000 accepts a powerful extended version of ANSI standard FORTRAN (x3.9-1966). To support multi-terminal capabilities, FORTRAN/3000 has been extended to allow free-form program input from a terminal device. Other extensions include: full 128-character USASCII 8-bit character set; character string manipulation with multi-dimensional string arrays; all MPE/3000 file capabilities; recursive subroutines with dynamic allocation of temporary local storage; variable names may contain up to 15 characters; and mixed mode arithmetic.

BASIC/3000. BASIC is a simple language designed especially for interactive terminal use. The BASIC/3000 version includes all the features of the standard BASIC language, plus a large number of extensions which exploit the inherent system capabilities of the HP 3000. The result is the most powerful version of BASIC available with any computer system. Although the interpreter is designed primarily for use from terminals, the inclusion of a command/program/data file facility makes it usable in batch mode as well.

COBOL/3000. HP 3000 COBOL is based on the ANSI Standard COBOL (USAS x3.23—1968) at a level upward compatible with the highest level of the Federal Government Standard. (ECMA COBOL conforms with ANSI COBOL.) COBOL/3000 is an extremely powerful and versatile computer language. It is ideal for administrative, financial, accounting, agency, inventory, warehousing, distribution and other commercial EDP applications. COBOL/3000 consists of a basic nucleus and functional processing modules that provide capabilities for table handling, sequential or random file access, record sorting, program segmentation, and specifying text to be copied from a library. An additional functional processing module for interprogram communication provides the capability to call subprograms written in COBOL/3000 or other HP 3000 languages from COBOL/3000 programs.

## USER-AID SUBSYSTEMS

Support software includes several classes of common and useful subroutines. All are written in SPL/3000.

The HP 3000 Compiler Library is a collection of subroutines which provide common functions required by FORTRAN, SPL/3000, and BASIC programs, such as: extended precision floating point arithmetic, matrix operations, complex arithmetic, and trigonometric and mathematical functions.

The EDIT/3000 Text Editor permits the user to create and edit on-line/batch computer programs and ordinary manuscript text. It allows the user to manipulate files of upper and lower case ASCII characters. Lines, strings and characters can be inserted, deleted, replaced, searched for, etc. The files to be edited can be source language programs, such as FORTRAN, SPL, COBOL, etc., or textual material, such as reports.

TRACE/3000 is a programmable debugging tool for high-level languages (FORTRAN/3000 and SPL/3000). It allows the programmer to monitor the execution of a program. The programmer can use TRACE/3000 to check the state of the program whenever a variable is changed or a label is passed. In addition, the programmer can specify selective conditions for output of information; e.g. print data only when a variable exceeds a certain value, or when a variable is changed a specific number of times.

The SORT/3000 Subsystem provides the capability to sort and/or merge multiple files of sequential records into a sequential file. This permits users of the HP 3000 Computer System to arrange large quantities of records (a file) into a prescribed order. Sorting is based on keys (values of one or more data fields). Merging forms one sorted sequence of records by combining one or more previously sorted sequences of records.

A set of scientifically-oriented software includes an extensive Scientific Library (geometric functions, correlation, etc.), and a group of interactive Statistical Analysis Routines (STAR). The Scientific Library routines can be called by user programs written in FORTRAN/3000, SPL/3000, or BASIC/3000. Communication with STAR is done via commands (in batch mode) or questions and answers (in on-line terminal mode).

## HARDWARE DIAGNOSTICS

The diagnostic software for the HP 3000 hardware is divided into three levels to cover all possible problem situations. The System Diagnostic Monitor (SDM/3000) runs on-line diagnostics under control of the operating system. Useful work may continue while this diagnostic is being run. A set of stand-alone diagnostics may also be used, which runs directly on the central processor without the operating system. If the problem is such that the stand-alone diagnostics cannot be run, the microdiagnostics can be used. These microprograms replace the instruction set microprograms of the central processor and check the functions of the hardware from the inside out. The microdiagnostic hardware, the hardware maintenance panel, and the auxiliary control panel may be connected remotely to a computer over a modem-common carrier line to allow direct Hewlett-Packard assistance on difficult problems.

## FUNDAMENTAL OPERATING SOFTWARE

The software listed below is required for operation of the HP 3000, and is designated as Fundamental Operating Software. Other software described above is optional dependent on system application or configuration. The Fundamental Operating Software includes:

- MPE/3000 (Multiprogramming Executive)
- SDM/3000 (System Diagnostic Monitor)
- Compiler Library
- File Utilities
- TRACE/3000 (Symbol Trace Facility)
- SPL/3000

# CENTRAL PROCESSOR

The *Central Processor* module determines the basic characteristics of the hardware system. This module includes a *microprocessor*, which processes all the machine instructions, a complement of 20 hardware registers, various indicators, and the logic for processing interrupts and input/output functions.

This section describes and defines the component elements of the Central Processor module. However, since I/O and interrupts are extensive subjects, they are treated in separate sections later in this manual. The discussion here concentrates primarily on the CPU registers, their purposes, and formats. First, for an overall view of the module, the basic structure will be shown and discussed.

## MODULE STRUCTURE

Basically, the Central Processor module is divided into three major component sections. These are: Central Processor Unit (CPU), I/O Processor (IOP), and Module Control Unit (MCU). The MCU is shared by the CPU and the IOP. Refer to figure 3-1.

### CENTRAL PROCESSOR UNIT

The *CPU* accounts for most of the logic circuitry in the module. As shown in figure 3-1, the major elements are the microprocessor, the indicators, and the CPU registers (including the Next and Current Instruction Registers).

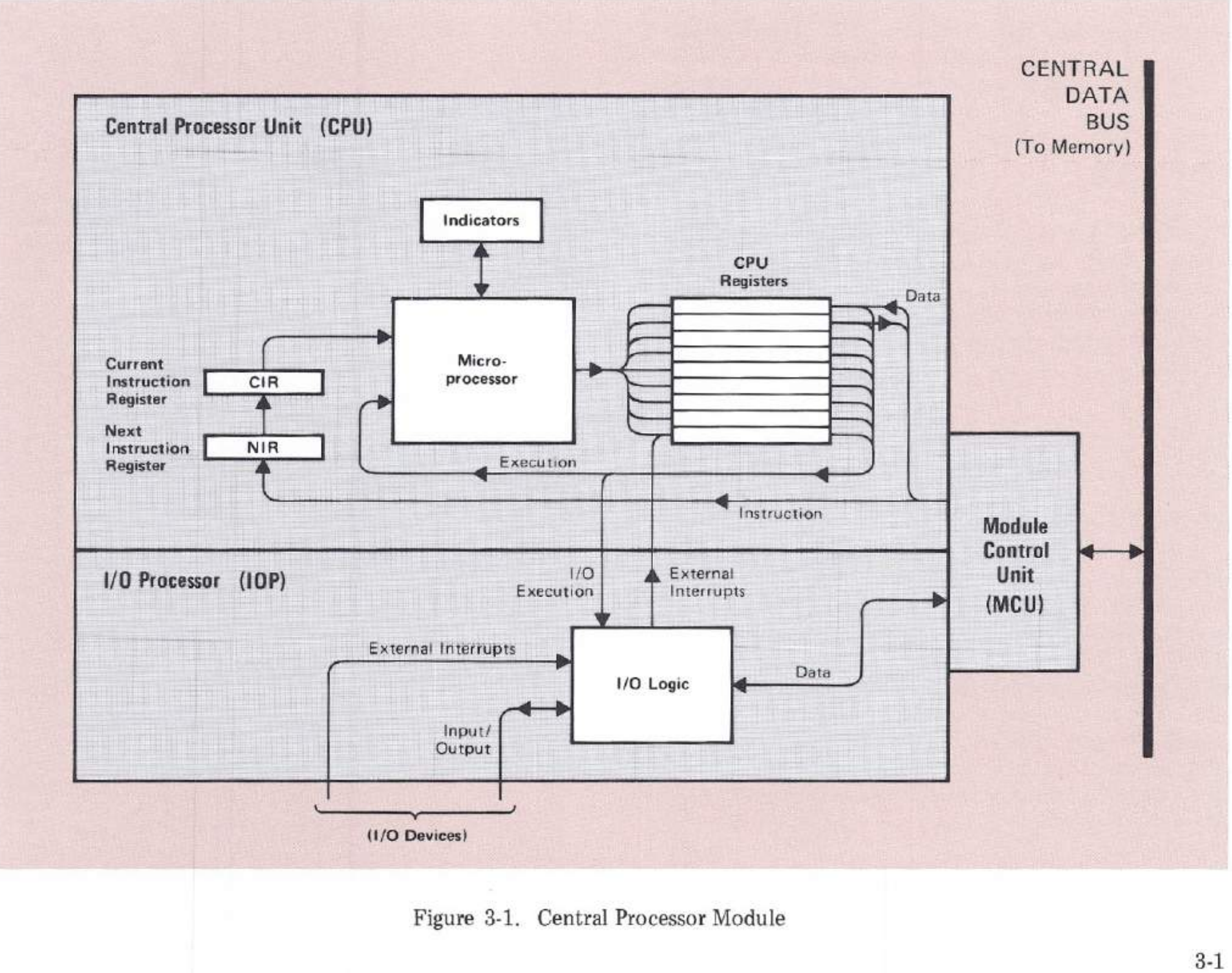


Figure 3-1. Central Processor Module

The basic sequence of events for the CPU is as follows: The microprocessor requests an instruction from memory via the MCU. When received, the instruction is loaded into the Next Instruction Register (NIR). When the current instruction is completely executed, the new instruction is transferred from NIR to the Current Instruction Register (CIR). This causes the microprocessor to begin executing a microprogram stored in its own internal solid-state memory. The microprogram manipulates and uses the contents of one or (usually) several CPU registers (see "Execution" arrow), according to the needs of the machine instruction being executed. The microprogram may change the state of one or more of the indicators, during execution, and may also initiate the transfer of operands or data to or from memory. At the conclusion of the microprogram, the desired action (such as a computation between two registers) will be complete, and the last step of the microprogram is to load the new NIR contents into CIR for execution of the next instruction.

## I/O PROCESSOR

There are seven I/O instructions which, when the CPU executes their respective microprograms, will cause the I/O logic to perform some function. (See "I/O Execution" arrow.) The IOP may cause an external device to transfer data to or from memory (or to or from a CPU register), or may cause the device to enable or request an interrupt. When a device interrupts, the IOP sets a bit in one of the CPU registers.

The hardware logic of the IOP is discussed extensively in Section VIII, and general input/output operations are described in Section VI. The interrupt system is discussed in Section VII. Refer to these sections for detailed information on I/O, interrupts, and the IOP.

## MODULE CONTROL UNIT

Most modules require a Module Control Unit (MCU) for inter-module communication via the central data bus. (The Selector Channel uses a Port Controller to perform MCU functions.) The MCU for the Central Processor module actually consists of two nearly-identical units, one for the CPU and one for the IOP. The IOP normally has higher priority in gaining access to the bus; however, when the CPU is attempting to complete a semi-completed operation (e.g., wants to transmit data to store in memory), the CPU takes higher priority.

In general, Central Processor communications via the MCU would normally be to or from a memory module.

## STACK

One of the fundamental features of the HP 3000 architecture is the data *stack* concept. Later, in Section IV, the operation of the stack will be described in detail. Here, the method of referring to elements in the stack will be defined. Refer to figure 3-2.

Figure 3-2 shows a data stack with 22 filled locations, all containing valid data, and 8 available unfilled locations. The stack area is delimited by the location defined as DB (Data Base) and the location defined as S (Stack pointer). The addresses DB and S are retained in dedicated CPU registers.

The data in the DB location is the oldest element on the stack. The data in the S location is the most current element. The location S is also referred to as the "Top of Stack" or TOS. Conventionally, the "top" is shown in diagrams "downward" from DB; this corresponds to the normal progression of writing software programs, which begins at the top of the page and proceeds downward.

To refer to previously stacked elements of data, "S-minus" relative addressing is used. Thus S-1 is the second element on the stack, S-2 is the third, and so on. S-minus relative addressing is one of the standard addressing conventions, as will be discussed later in this section.

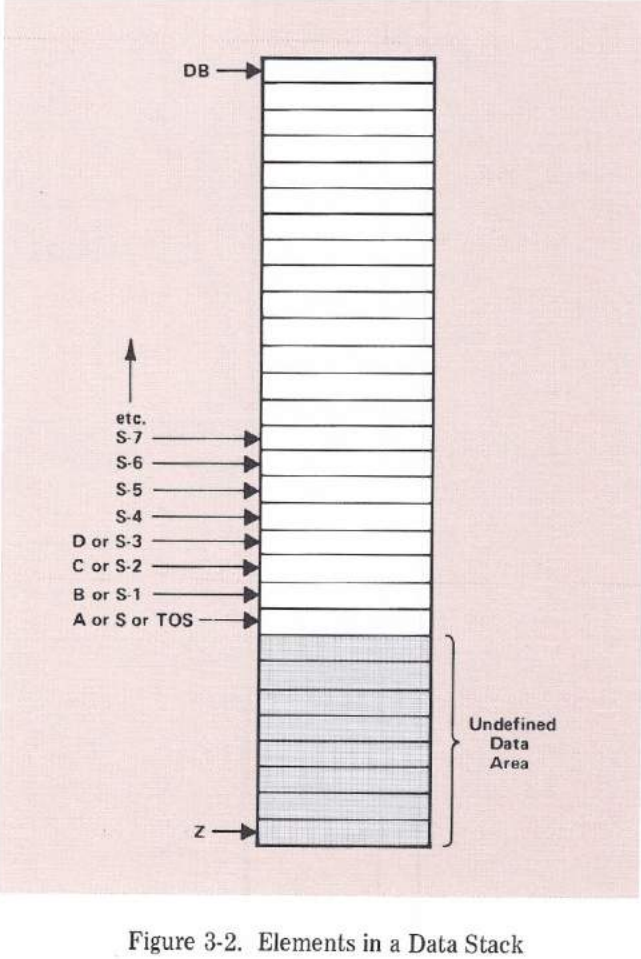


Figure 3-2. Elements in a Data Stack

Since the top four elements of the stack are the most frequently used, the letters A, B, C, D are also often used. With this convention, A is the top of the stack, or S, B is S-1, C is S-2, and D is S-3. There are, in fact, four CPU registers (TR0 through TR3) which are sometimes referred to by the logical names RA, RB, RC, and RD, and may at various times contain up to four of the topmost stack elements. However, these registers are not explicitly addressable; the S-minus addressing mode must be used to access their contents. The A, B, C, D designations are primarily a documentation convenience.

The area from S+1 to Z (the eight shaded locations) are available for adding more elements to the stack. When a data word is added to the stack, it is stored into the next available location and the S pointer is incremented by one to reflect the new TOS. This process is said to *push* a word onto the stack. To *delete* a word from the stack, the S pointer is simply decremented by one, thus putting the word into the undefined area.

# FORMATS

## DATA

There are six different *data formats* that are processable by System/3000 instructions. These are shown in figure 3-3. (The long floating point format is used primarily in software; the TNSL instruction is the only hardware operation which directly handles this format.)

BYTE FORMAT. Bytes are processed by five of the Move instructions (CMPB, MVB, MVBW, SCU, SCW), by two memory reference instructions (LDB and STB), and by the "byte test" instruction, BTST. Figure 3-3 shows the basic byte format, which usually contains an eight-bit data character, and the format for packing two bytes into a memory word. When bytes are processed by machine instructions, the bytes are individually addressed, fetched, and stored as though memory consisted of a number of eight-bit locations. (See "Addressing Conventions".) When consecutive bytes are addressed in memory with ascending addresses, the high order byte of a packed word is accessed first and the low order byte (bits 8 through 15) second.

LOGICAL FORMAT. In logical arithmetic, a 16-bit data word is taken as a positive integer, with an assumed binary point to the right of bit 15 and an assumed + sign to the left of bit 0. The range of possible integers is 0 through +65,535, decimal. The instruction set provides six instructions for logical arithmetic: LCMP, LADD, LSUB, LMPY, LDIV, and NOT. In addition and subtraction (LADD, LSUB), the only difference from integer adds and subtracts is that logical adds and subtracts do not set the Overflow indicator. In all other respects (16-bit result, Condition Code, and Carry), the results are the same. For addition, the Carry bit is set if a carry out of the most significant bit occurs; if the carry out does not occur, the Carry bit is cleared. For subtraction (which is accomplished by two's complementing the subtrahend and adding), Carry is set by a computation of A - B if B is less than A. Carry is cleared if B is greater than A. Thus if the Carry bit is set by LADD, the sum has exceeded +65,535, and if the Carry bit fails to be set by LSUB, the difference is less than zero. In either case the result is modulo $2^{16}$. For multiplication (LMPY), overflow cannot occur and the Carry bit has a special meaning (see definition). For division (LDIV), the Overflow (not Carry) bit is used, and indicates that the quotient is too large to be represented in 16 bits; The quotient in this case will be modulo $2^{16}$. When the Condition Code is set by a logical operator, it is set as if the result were a signed quantity. For example, CCL is set if bit 0 is a "1" ("negative" quantity).

SINGLE FIXED POINT FORMAT. The single-word fixed point format permits two's complement representation of both positive and negative integers. Bit 0 is a sign bit, and the remaining 15 bits define the quantity. The range of possible integers is -32,768 through +32,767. Bit 0 is a "0" for positive numbers and a "1" for negative numbers. The binary point is assumed to be to the right of bit 15. The instruction set provides 24 instructions for single-length integer arithmetic. These include various modes of addition, subtraction, incrementing and decrementing. In addition and subtraction (ADD, SUB), conventional two's complement arithmetic is used. Both Overflow and Carry indicators are provided. Overflow indicates that the computation result required more than 15 bits for the quantity and consequently overflowed into bit 0, the sign bit. For valid subtraction and addition, Carry should be set by SUB, but not by ADD. For multiplication and division, Carry is not used; Overflow indicates that the result cannot be contained in 15 bits plus sign.

DOUBLE FIXED POINT FORMAT. The double-word fixed point format is the same as the single-length format described in the preceding paragraph except that two words are linked together to form a 32-bit doubleword quantity. Bit 0 of the most significant word is the sign bit. The range of possible integers is approximately -2 billion to +2 billion. The instruction set provides six instructions for double-length integer arithmetic: DCMP, DADD, DSUB, DNEG, MPYL, and DIVL. For multiplication with MPYL, overflow cannot occur and the Overflow bit is always cleared; Carry is used for a special purpose (see MPYL definition). The operands for MPYL and the divisor and quotient for DIVL are single-word.

FLOATING POINT FORMAT. In this format, bit 0 of the most significant word is the sign bit, bits 1 through 9 are used to express the exponent, and the remaining bits represent the fraction. The binary point is assumed to be to the left of bit 10.

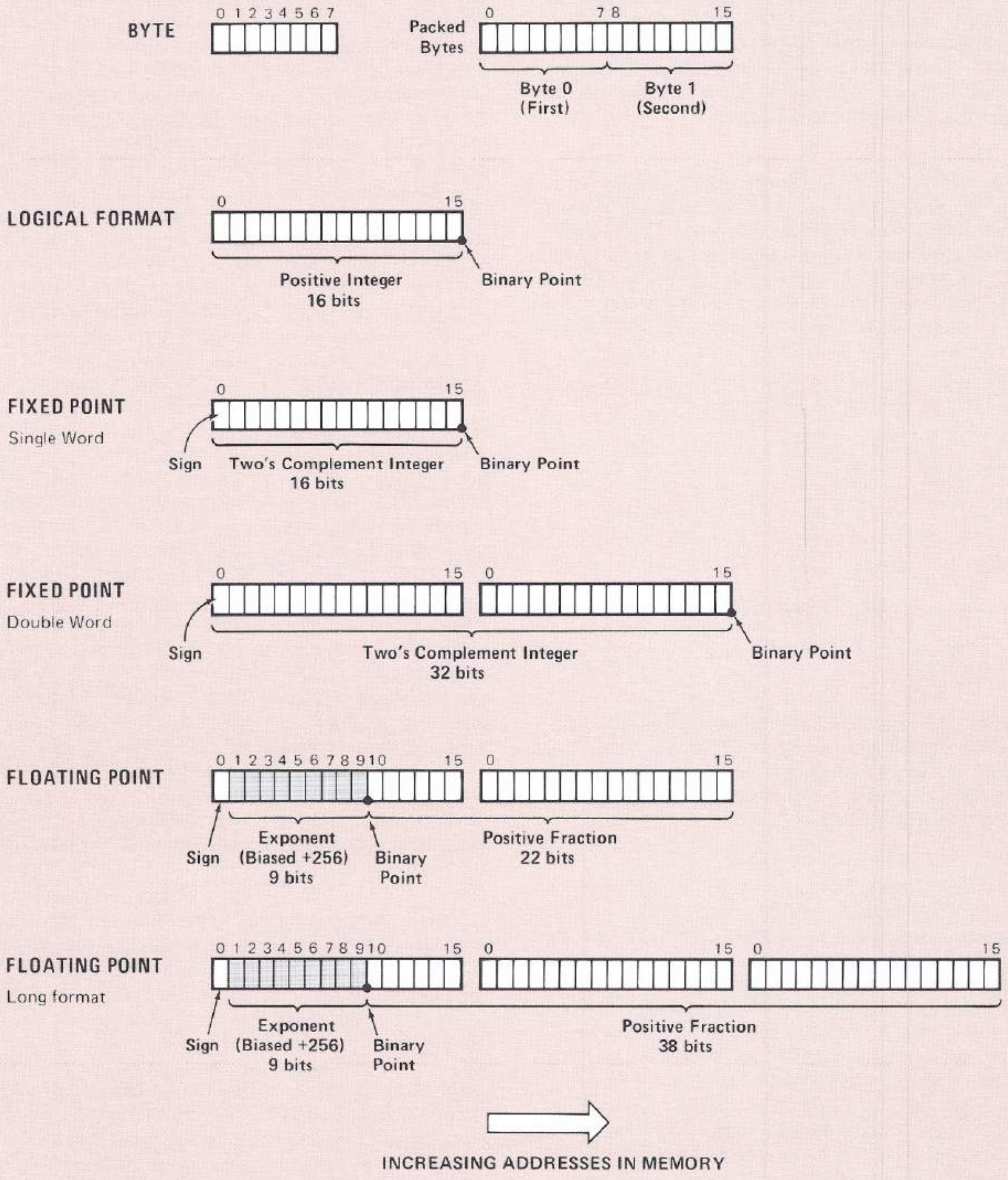


Figure 3-3. Data Formats

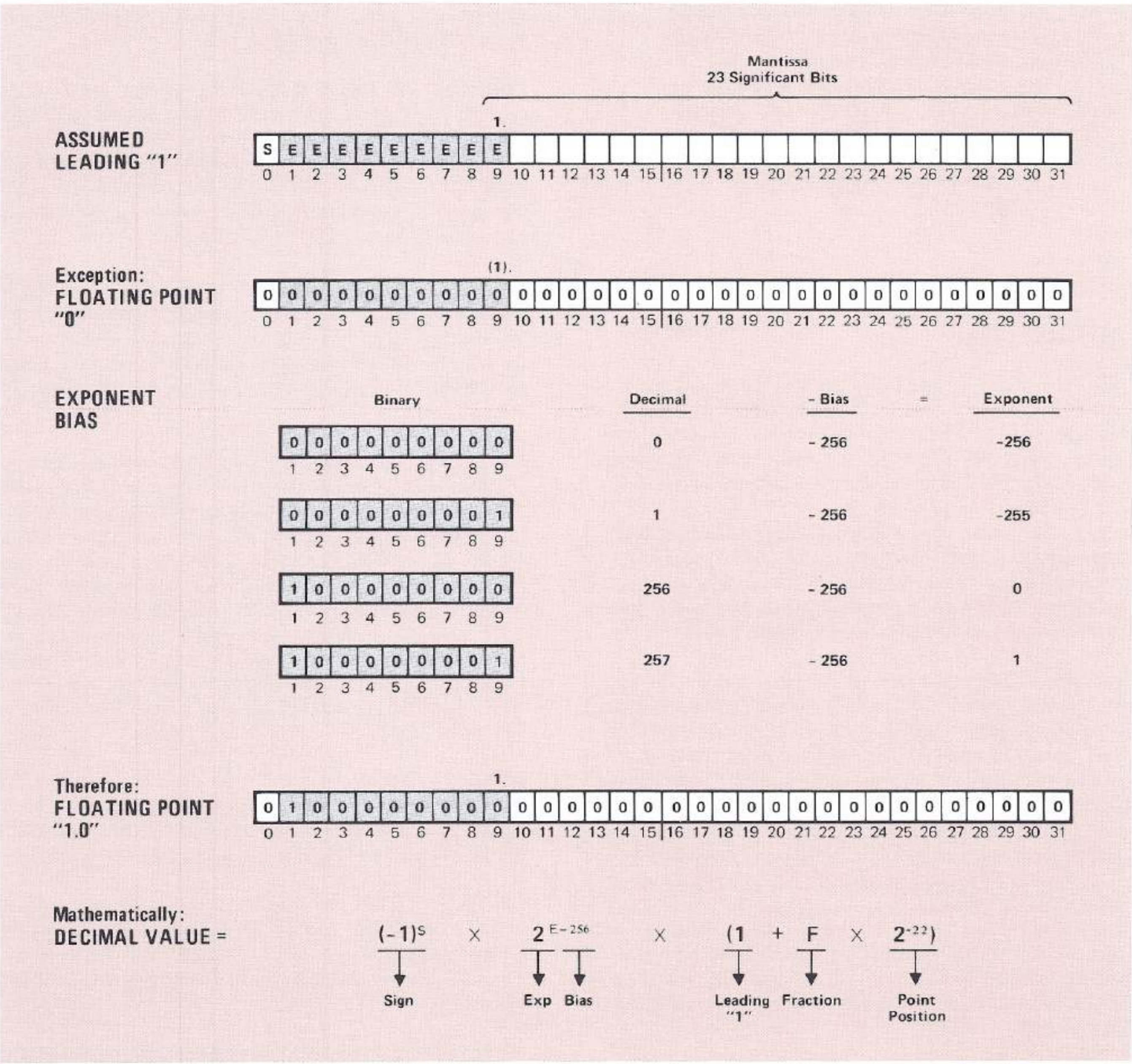
The floating point format used by the HP 3000 has some special features which are illustrated separately in figure 3-4. The important distinction is the use of "sign with +magnitude" representation. In this type of representation, the fraction is always positive, with the sign bit indicating the sign of the number. There is an assumed "1" to the left of the binary point. Thus all floating point numbers, by definition, exist in normalized form and the mantissa effectively has 23 bits. However, no bit is wasted on the leading "1", and all fraction bits are significant.

The exception to this convention is that zero is a word containing all "0"s. For this to be true, the assumed leading "1" is disregarded.

The exponent for floating point numbers is biased by +256. Since the nine exponent bits give a range of 0 through 511, subtracting the bias yields an exponent range of -256 through +255. Figure 3-4 shows four examples of exponent calculation. Note that if bit 1 is a "0", exponents are negative; if bit 1 is a "1", exponents are positive or zero.

Thus the floating point representation of 1.0 is a "1" in bit 1 and "0"s in all other bits. This indicates 1 X $2^0$.

Figure 3-4 also shows the mathematical equation for computing the value of a floating point number represented by the above conventions. (The exception: zero is defined as: S = E = F = 0.)



| Binary | Decimal | - Bias | = | Exponent |
|---|---|---|---|---|
| 0 0 0 0 0 0 0 0 0 | 0 | - 256 | | -256 |
| 0 0 0 0 0 0 0 0 1 | 1 | - 256 | | -255 |
| 1 0 0 0 0 0 0 0 0 | 256 | - 256 | | 0 |
| 1 0 0 0 0 0 0 0 1 | 257 | - 256 | | 1 |

$$(-1)^S \times 2^{E-256} \times (1 + F \times 2^{-22})$$

Figure 3-4. Floating Point Data Representation

The instruction set provides ten floating point instructions: FCMP, FADD, FSUB, FMPY, FDIV, FNEG, FLT, DFLT, FIXT, FIXR. Overflow indication is provided by the mathematical operations (FADD, FSUB, FMPY, FDIV), and by the "fix" instructions (FIXT, FIXR). The Carry indication is not used, except for a special purpose by the FIXT and FIXR instructions (see definitions).

LONG FLOATING POINT FORMAT. The long floating point format is the same as the standard format described above except that 16 fraction bits are added to the right of the second word. With only this change, the information given in figure 3-4 is also valid for this format. (Note that the "point position" modifier in the equation becomes $2^{-38}$ instead of $2^{-22}$.) Only one instruction, TNSL, directly uses the triple-length floating point format.

Note: In all cases where more than one word is used to represent a single unit of data, the words are stored in memory such that the least significant word is stored in the higher address location. For example, when pushing a doubleword or tripleword quantity onto the stack, the least significant word will be on the TOS.

## INSTRUCTIONS

The HP 3000 instruction set has been designed for maximum efficiency of bit usage in the instruction word. For this reason, the *instruction formats* do not necessarily always fall neatly into rigid field boundaries. There are, in fact, 23 distinct formats used by the instruction set.

Figure 3-5 shows the primary format in each of the 13 instruction groups. Exceptions are noted and can be obtained from Section V, where formats are given for each individual instruction. The following paragraphs briefly describe the basic formats shown in figure 3-5.

GENERAL FORMAT. The first format in figure 3-5 shows the general scheme for dividing the instruction word into code fields. Only the first field is rigidly adhered to. This field, bits 0 through 3, either defines a specific instruction code in the memory address group (or the "loop control" group), or else defines one of the *sub-opcode groups*. There are four sub-opcode groups: 1, 2, 3, and "stack ops". The field for *sub-opcodes* varies. For sub-opcodes 2 and 3, bits 4, 5, 6, and 7 are used, as shown. For sub-opcode group 1 codes, bits 5 through 9 are used, and for stack ops the remainder of the word is used. In some cases the sub-opcode will enable a third field, called a *mini-opcode* or a *special opcode*, in bits 8, 9, 10, and 11. The remainder of the word has a variety of special uses, and commonly is part of an "argument field".

STACK OP. The stack op format is defined by four "0"s in the first four bits. The remaining 12 bits are divided into two fields; stack op A and stack op B. Either or both of these fields may contain any of the 63 stack op instruction codes. Execution sequence is from left to right (A first, then B). Interrupts may occur between the execution of A and B. Also note that indicators (Carry, Overflow, and Condition Code) are set by the last executed stack op. If using only one of the two stack op fields, it is more efficient to use stack op A since the hardware always looks ahead to see if stack op B is a NOP; this permits the hardware to ignore the second field, resulting in a time saving.

SHIFT. The shift instruction group uses about half of the sub-opcode 1 group of codes. Sub-opcode group 1 is defined by 0001 in the first four bits. If bit 4, the Index bit, is a "1", the content of the Index register is added to the shift count in bits 10 through 15 to specify the number of places each data bit is shifted. Bits 5 through 9 encode the specific shift instruction.

BRANCH. The branch instructions account for 11 of the sub-opcode 1 group of codes. In the branch instruction format, bit 4 is used as an indirect bit (indirect if bit 4 = "1"). Bits 5 through 9 encode the specific branch instruction. Bits 11 through 15 give a P relative displacement (0 through 31), and bit 10 specifies whether the displacement is + or - relative to P ("0" = +, "1" = -).

BIT TEST. The bit test instructions, also in sub-opcode group 1, use bits 5 through 9 to specify the instruction. Bits 10 through 15 specify a bit position in the TOS word for testing. The bit position specified is modified by the addition of the Index register contents if the Index bit is set (bit 4 = "1").

MOVE. The move group of instructions accounts for eight of the codes specified by the sub-opcode 2 code 0000. Sub-opcode group 2 is defined by 0010 in the first four bits. Bits 8, 9, and 10 of the move instruction format encode the specific instruction. Bit 11 is used for some instructions to specify whether the source of the moved data is PB relative (bit 11 = "0") or DB relative (bit 11 = "1"). Bit 11 is also used in some cases as an additional code bit for specifying the instruction. Bits 12 and 13 are not used. Bits 14 and 15 are used to specify an S-decrement value to delete, if desired, the move parameters from the top of the stack.

SPECIAL. The special group uses four mini-opcodes. The mini-opcode group is also, like the moves, specified by the sub-opcode 2 code 0000. Bits 8 through 11, plus bit 15, encode the instruction. Bits 12, 13, and 14 are not used.

IMMEDIATE. The immediate instruction group uses codes in both sub-opcode group 2 (coded 0010) and sub-opcode group 3 (coded 0011). Bits 4 through 7 encode the instruction and bits 8 through 15 are used for the immediate operand.

FIELD. The format for field deposit and extract instructions is specified by two of the sub-opcode 2 group of codes. Bits 4 through 7 specify the instruction and the

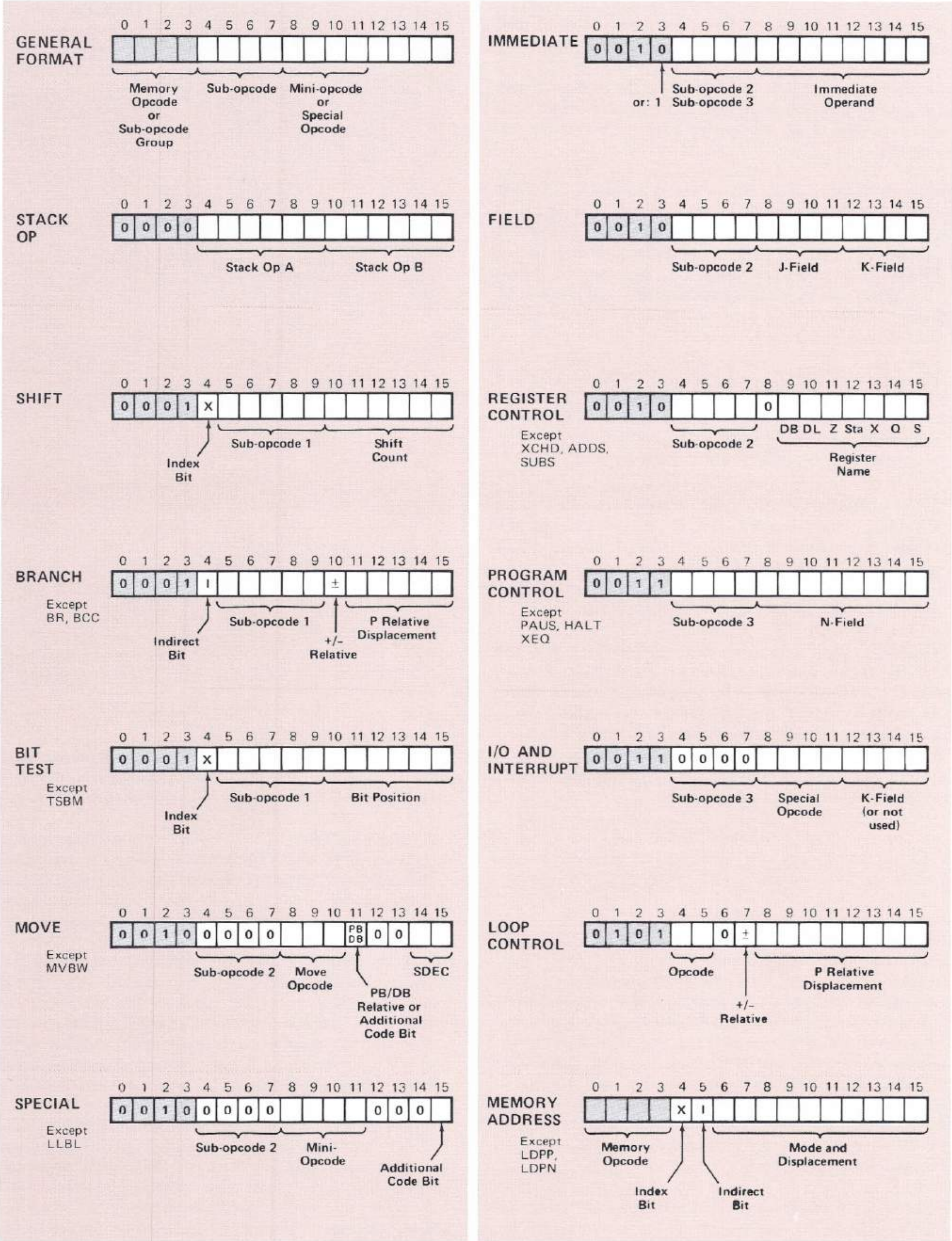


Figure 3-5. Instruction Formats

remaining eight bits are divided into a J-field and a K-field. The J-field specifies the starting bit number and the K field specifies the number of bits.

REGISTER CONTROL. The format for the register control instructions uses bits 9 through 15 to name a register and bits 4 through 7 in sub-opcode group 2 to specify the operation.

PROGRAM CONTROL. The program control instructions account for four of the sub-opcode 3 codes. Sub-opcode 3 is specified by 0011 in the first four bits. The instruction is encoded by bits 4 through 7, and the N-field in bits 8 through 15 is used either for a PL– displacement (PCAL AND SCAL) or to specify a number of parameters to be deleted on return from a procedure or subroutine (EXIT and SXIT).

I/O AND INTERRUPT. The I/O and interrupt instructions use 11 of the special opcodes (bits 8 through 11) defined by the sub-opcode 3 code of 0000. The K-field, bits 12 through 15, is used by some of the instructions for an S-displacement to locate a device number given in the stack.

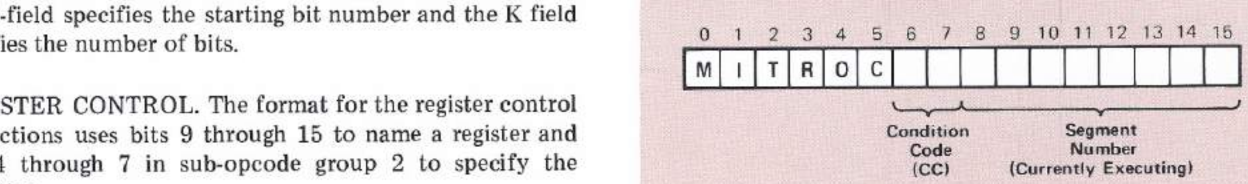
LOOP CONTROL. The loop control instructions are defined by a special coding of bits 4, 5, and 6 for memory opcode 05 (which is otherwise defined as the STOR instruction). Bits 8 through 15 give a P relative displacement for a branch address, and bit 7 specifies whether the displacement is + (= "0") or - (= "1") relative to P.

MEMORY ADDRESS. The memory address instruction format uses bits 0, 1, 2, and 3 to encode a specific instruction. Bits 6 through 15 give both an addressing mode and a displacement. (Refer to "Addressing Conventions", later in this section.) Bit 5 is used to specify indirect addressing (= "1"), if desired, and bit 4 is used to specify indexing (= "1"), if desired. If both indirect addressing and indexing are specified, post-indexing will occur.

## STATUS WORD

There is a *Status word* for each code segment in the system. At all times, the Status word associated with a given process indicates the machine status following the execution of the most recent instruction in that segment. The status for the currently executing segment is resident in the Status register, and is constantly being updated as each instruction is executed. For segments that are not current (suspended by either an interrupt or a procedure call), the Status word exists in a stack marker in a data stack. (See "Stack Marker Format" figure in Section IV.)

Figure 3-6 shows the format for the Status word. Note that bits 8 through 15 indicate the segment number of the currently executing code segment (when the particular Status word is resident in the Status register). Thus, when a Status word is pushed into a stack marker by an interrupt or procedure call, these bits identify the segment that is to be returned to when execution is resumed later.



**M Privileged Mode bit**
1 = Privileged Mode
0 = User mode

**I External Interrupts Enable/Disable**
1 = Enable
0 = Disable

**T User Traps Enable/Disable**
1 = Enable
0 = Disable

**R Right Stack Op Pending bit**
1 = Pending (Execution will proceed with Stack Op B)
0 = Not Pending

**O Overflow bit**

**C Carry bit**

**CC Condition Code**

| | |
|---|---|
| 00 = CCG | (Greater) |
| 01 = CCL | (Less) |
| 10 = CCE | (Equal) |

Figure 3-6. Status Word Format

The following descriptions of Status bits will assume that the Status word under discussion is resident in the Status register. All references to "current" conditions can also be inferred as "then current" conditions in the case of suspended segments or procedures.

Bit 0, the *Privileged Mode bit*, indicates that the current segment is running either in privileged mode (if a "1") or user mode (if a "0"). The state of this bit cannot be changed by machine instructions while resident in the Status register (except in privileged mode), and the PCAL and EXIT instructions include checks to prevent illegal mode changes by altering the non-current status Mode bits.

Bit 1 is used to enable or disable external interrupts. This bit also cannot be changed in user mode while current, and the EXIT instruction invokes a trap if a non-privileged user illegally altered the bit while non-current. The state of bit 1 may be changed only in privileged mode. (PCAL and EXIT disable external interrupts if they transfer control to the Trace, Absence, or STT Entry Uncallable segments, due to not being completely executable.)

Bit 2 is used to enable or disable user traps (parameters 1 through 5 for interrupt segment 17). The state of this bit may be changed in any mode while current (SETR instruction) or non-current (state not affected by EXIT).

Bit 3 is normally used only by the hardware. The computer hardware will set this bit to a "1" if the right stack opcode (bits 10 through 15) contain a valid instruction other than NOP. The hardware requires this information in case an interrupt occurs between the execution of the left and right stack ops. The state of bit 3 cannot be changed in user mode while current.

Bit 4 is the *Overflow* bit, and is one of the three *indicators* (along with Carry and Condition Code) which are set or cleared as an incidental operation by many of the machine instructions. (See "Indicators" following each instruction definition in Section V.) In general, Overflow is used as an indicator only by signed integer and floating point computations. If set (= "1"), the indication is that the result of the computation is too large to be represented in the available number of bits in the data format. For floating point, the setting of Overflow could also indicate that the result is too small to be represented. If the user traps are enabled (bit 2 set), an interrupt to segment 17 will occur in lieu of setting the Overflow indicator (except for integer overflow, which causes both results to happen). This will permit the system to generate a message to the user, indicating which type of overflow or underflow occurred. All user traps will set the Overflow indicator if traps are disabled.

Bit 5 is the *Carry* bit. The Carry indicator is used primarily by logical and integer arithmetic, and usually indicates a carry (= "1") or lack of carry (= "0") out of the most significant bit during a computation. The Carry bit is also used by some instructions as an indicator for special purposes which are stated in the instruction definitions.

Bits 6 and 7 are used for the *Condition Code*. Although several instructions make special use of the Condition Code (see definitions), the Condition Code typically indicates the state of an operand (or a comparison result with two operands). The operand may be a word, byte, doubleword, or tripleword, and may be located on the top of the stack, in the Index register, or in a specified memory location. Three codings are used: 00, 01, and 10. (The "11" combination is not used.) Except for the special interpretations, there are three basic patterns for interpreting these codes. The three patterns are shown in table 3-1.

The most common Condition Code pattern is pattern A, designated as CCA. In the CCA pattern, the Condition Code is set to 00 if the operand is greater than zero, to 01 if the operand is less than zero, or to 10 if the operand is exactly zero. Since this usage of the Condition Code is so common, the three codes 00, 01, and 10 are commonly named to reflect these meanings. Thus 00 is CCG ("Greater"), 01 is CCL ("Less"), and 10 is CCE ("Equal"). These names are primarily used for documentation convenience.

Pattern B for the Condition Code, designated as CCB, is used with byte oriented instructions. In the CCB pattern, the Condition Code is set to 00 if the operand byte is an ASCII numerical character, which would be represented by octal values 060 through 071. The code is set to 10 if the byte is an ASCII alphabetic character, which would be represented by octal values 101 through 132 for upper case letters, and 141 through 172 for lower case letters. The code is set to 01 if the byte is an ASCII special character, represented by the remaining octal values.

Pattern C for the Condition Code, designated as CCC, is used with comparison instructions. The Condition Code is set to 00 if operand 1 is greater than operand 2, or to 01 if operand 1 is less than operand 2, or to 10 if the operands are equal. In the instruction definitions, the first mentioned operand is "operand 1". For example, the definition for CMP reads: "The Condition Code is set to pattern C as a result of the integer comparison of the second word of the stack with the TOS." The second word of the stack is therefore operand 1, and the TOS is operand 2. (The Index of Instructions also defines this relationship, if the operands are listed.)

Table 3-1. Condition Codes

| | | |
|---|---|---|
| **CCA** | sets CC | = CCG (00) if operand > 0 |
| | | = CCL (01) if operand < 0 |
| | | = CCE (10) if operand =0 |
| **CCB** | sets CC | = CCG (00) if numerical (octal 060-071) |
| | | = CCL (01) if special char (all others) |
| | | = CCE (10) if alphabetic (upper 101 - 132<br>lower 141 - 172) |
| **CCC** | sets CC | = CCG (00) if operands 1 > 2 |
| | | = CCL (01) if operands 1 < 2 |
| | | = CCE (10) if operands 1 = 2 |

## CPU REGISTERS

Since the HP 3000 architecture is structured on code segments and data segments, most of the CPU registers are used for defining the segment limits and operating elements within the segments. As shown in figure 3-7, three of the CPU registers point to locations in a code segment; the segment so pointed to is defined as the current code segment. Six of the registers point to locations in a data segment; the segment so pointed to is defined as the current data segment. The following paragraphs define the functions of the individual registers.

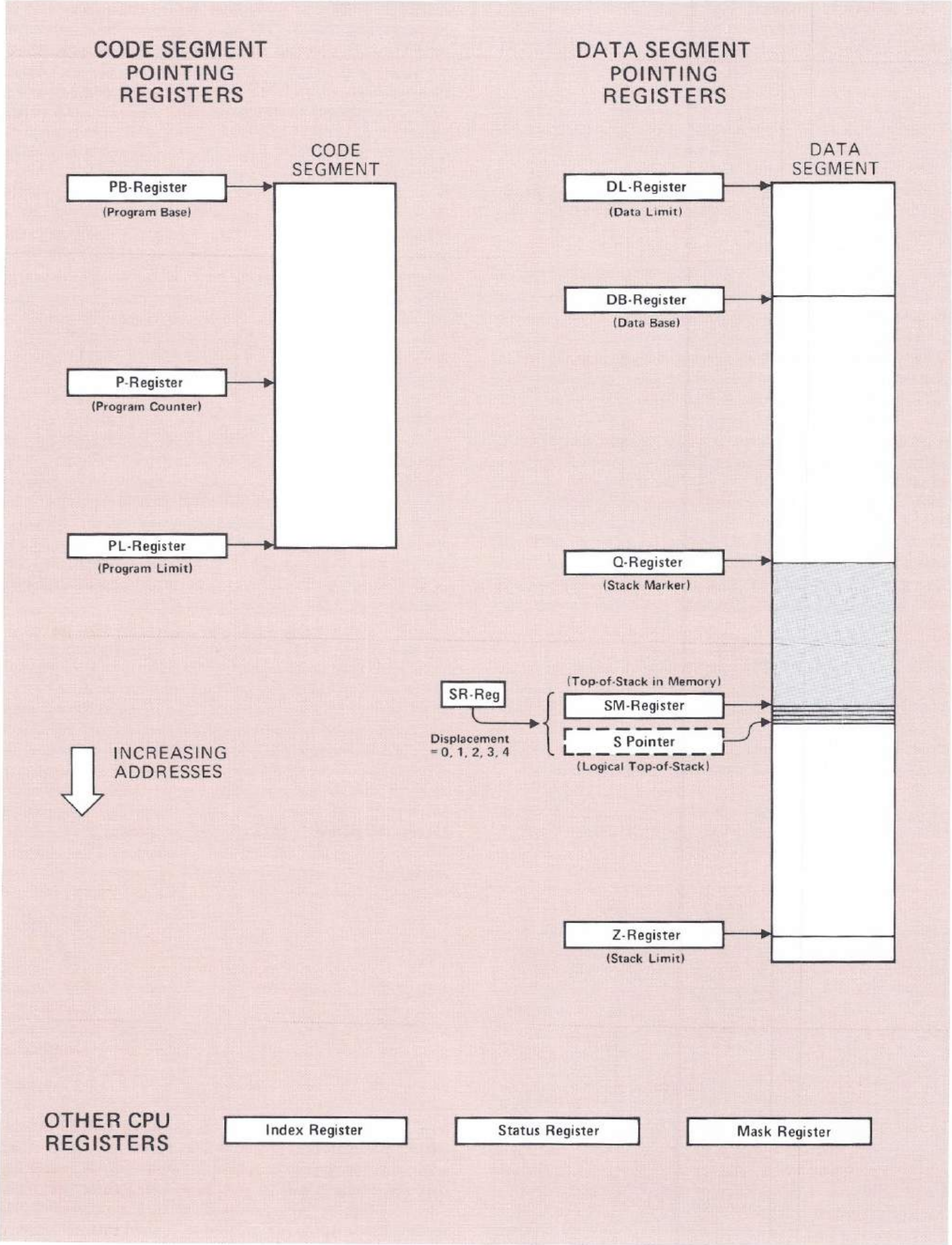


Figure 3-7. CPU Registers

## CODE SEGMENT REGISTERS

PB-REGISTER. The PB-register defines the *program base* of the code segment being executed. The register contains a 16-bit absolute address pointing to the first location of the code segment.

P-REGISTER. The P-register is the *program counter*. It contains a 16-bit absolute address pointing to the location of the instruction being executed. It can never point to a location beyond the limits defined by the PB- and PL-registers. An attempt to do so will invoke a Bounds Violation interrupt or a PCAL to the operating system.

PL-REGISTER. The PL-register defines the *program limit* of the code segment being executed. The register contains a 16-bit absolute address pointing to the last location of the code segment.

## DATA SEGMENT REGISTERS

DL-REGISTER. The DL-register defines the *data limit* of the current data segment. The register contains a 16-bit absolute address pointing to the first word of memory available to the user's data space.

DB-REGISTER. The DB-register defines the *data base* of the current user's stack. The register contains a 16-bit absolute address pointing to the first location of the directly addressable global area of the stack.

Q-REGISTER. The Q-register defines the current stack marker in the current data segment. The portion of the stack between Q and S represents data that is incurred by the current procedure or routine. The Q-register contains a 16-bit absolute address pointing to the fourth word of the current stack marker being used within the stack. The content of this register may be changed by a SETR instruction, but since bounds checking is always performed by the EXIT instruction, the location pointed to must be within the limits defined by the DB- and Z-registers (except that privileged mode may move Q below DB).

SM-REGISTER. The SM-register defines the last memory location of the current stack. The register contains a 16-bit absolute address pointing to the last accessed data location in memory. Since the SM-register may not necessarily point to the logical top of the stack, the S pointer, rather than the SM-register, is the address of interest for programming purposes. However, bounds checking is performed on the SM-register, which must be between the limits defined by the DB- and Z-registers (except that privileged mode may move S below DB).

SR-REGISTER. The SR-register defines the number of TOS elements that are in CPU stack registers. The register contains a 3-bit number which can only have one of the following values: 0, 1, 2, 3, or 4. This number is a positive displacement which, when added to the address in the SM-register, indicates the actual (or "logical") top of the stack.

S-POINTER. The S pointer defines the logical top of the stack. The S pointer is not a physical register but rather is logically comprised by adding together the SM- and SR-register contents.

Note: The principle of using two physical registers to create the S pointer is employed for hardware convenience in achieving fast execution times. For nearly all programming purposes, the existence of the SM- and SR-registers may be ignored, using instead only the value S.

Z-REGISTER. The Z-register defines the *stack limit* of the current user's stack. The register contains a 16-bit absolute address which points to the last location available to the stack. (Each data segment actually has about 13 locations beyond Z since bounds checks are made with SM instead of S, and also to allow space for stack markers due to an interrupt.)

## OTHER CPU REGISTERS

Three CPU registers not associated with code or data segments are the Index register, the Status register, and the Mask register. These are described in the following paragraphs.

INDEX REGISTER. The Index register is a 16-bit register which contains the index to be used by a machine instruction if indexing is specified. It may also be used to contain a parameter or address for other (non-memory addressing) instructions. The Index register is program accessible.

STATUS REGISTER. The Status register is a 16-bit register which indicates the current status of the computer hardware, including: the segment number of the currently executing code segment, the state of the three indicators (Overflow, Carry, and Condition Code), the current mode (privileged or user), enable/disable control bits for external interrupts and user traps, and stack opcode status. (Refer to "Status Word" format earlier in this section.)

MASK REGISTER. The Mask register is a 16-bit register which indicates the current mask being used to enable or disable specified groups of external interrupts. A "1" bit in any particular position enables the group of external interrupts which are specifically wired to be controlled by that bit; a "0" bit will disable the group of interrupts. The Mask register may be loaded from the TOS by the SMSK instruction (privileged) and may be read to the TOS by the RMSK instruction (not privileged).

# PRIVILEGED MODE

The HP 3000 has the capability of operating in either *privileged mode* or *user mode*, and is capable of switching dynamically from one mode to the other depending on the type of operation being executed at a given instant.

Privileged mode is characterized by the ability to execute the 19 privileged instructions and to call segments that have been declared "uncallable". In general, the *privileged user* is defined to be the operating system, which in most cases will be the Hewlett-Packard MPE/3000 executive software. Privileged operations, such as input/output, are performed by the operating system, operating in privileged mode. For an unprivileged user to perform such operations, it is necessary to call one of the callable intrinsics of the operating system, which will in turn call the uncallable intrinsics that will perform the operation on behalf of the user.

The mode currently in effect in the system is indicated at all times by bit 0 of the Status register. The state of this bit may be changed only in privileged mode.

The method of declaring a code segment uncallable involves the use of an "uncallable bit" in the format of local program labels. The format and application of program labels is discussed later, in Section IV.

# ADDRESSING CONVENTIONS

## MEMORY ADDRESSING

Earlier, in figure 3-5, the format for memory address instructions was shown to employ bits 6 through 15 for "mode and displacement". The following paragraphs explain and illustrate the six memory *addressing modes* and the respective displacement ranges. Refer to figure 3-8.

The HP 3000 uses *relative addressing* almost exclusively. (Only privileged instructions, including the I/O group and PLDA, PSTA, and LLSH, use absolute addresses.) Addressing may be relative to the location pointed to by the P-register, the DB register, the Q-register, or the S pointer. As shown in figure 3-8, addressing may be + or - with respect to P or Q, but only + with respect to DB and - with respect to S.

Note: When the letters P, Q, DB, etc., are used alone as in the preceding paragraph, the letter is interpreted to mean "the location pointed to by the P-register, Q-register, DB-register, etc." This convention simplifies such references in documentation and verbal communications.

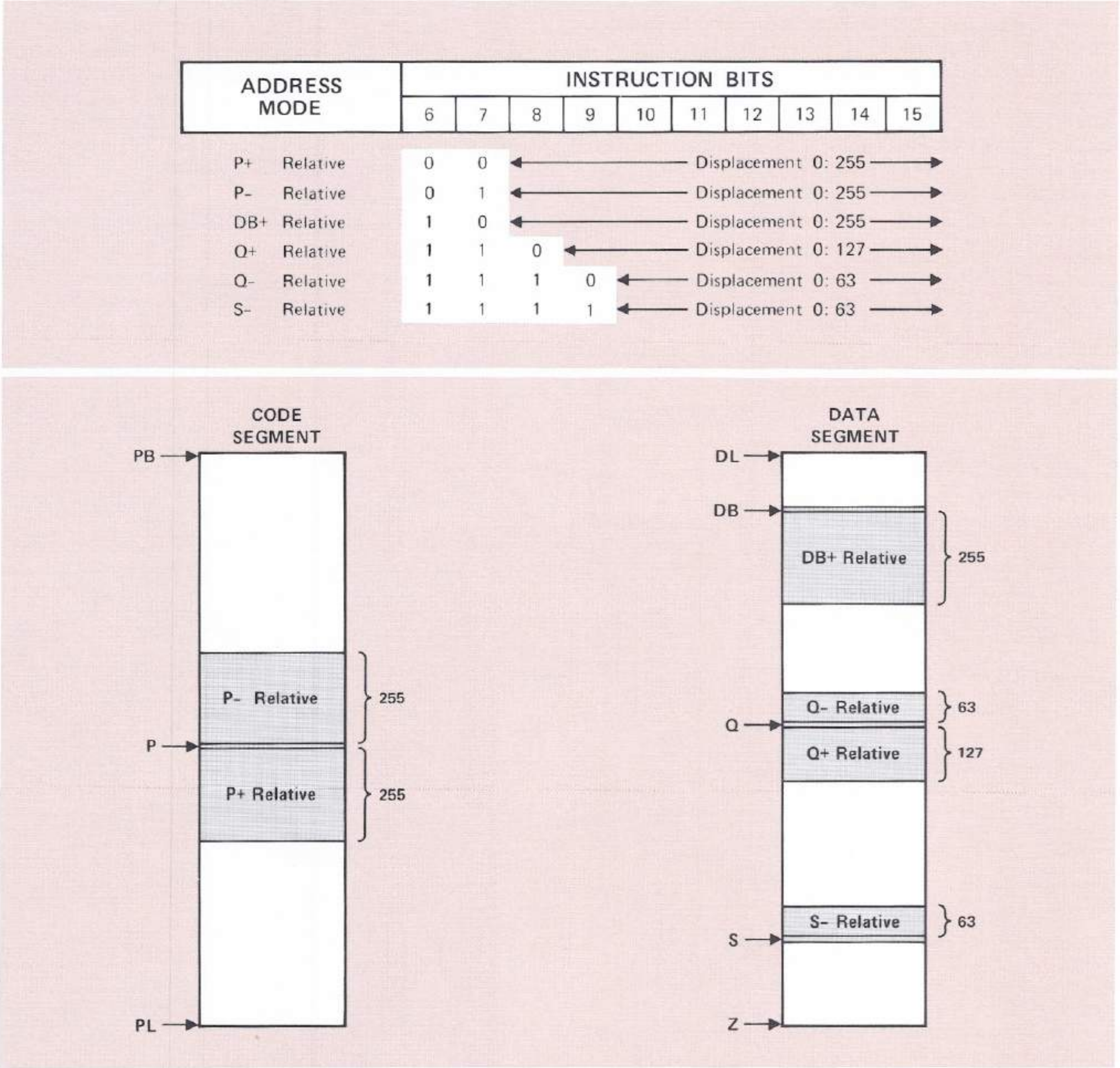
The ranges of displacement for the various modes of relative addressing are also shown in figure 3-8. (These ranges apply to direct, unindexed addressing; indirect addressing and indexing are discussed under separate headings.) The variety of displacement ranges is due to the particular coding required to specify a given mode. For example only two bits (6 and 7) are required to specify the P+, P-, and DB+ relative modes. This leaves bits 8 through 15 for a displacement, which therefore can be any value from 0 through 255. For Q+ mode, bits 9 through 15 give a displacement range of 0 through 127. For Q- and S- modes, bits 10 through 15 give a displacement range of 0 through 63. In order to provide the most efficient usage of bits, the mode codes are assigned according to respective needs for displacement range.

Note that the DB+, Q-, Q+, and S- addressing ranges may overlap. Also, DB+, Q+, and S- may actually address words currently held in TOS registers; this is automatically taken care of by the hardware.

P+ and P- addressing modes are typically used for branches and referencing of literals. The DB+ mode is used for referencing global variables and pointers (i.e., indirect addresses). The Q+ and Q- modes are useful for, respectively, local variable storage and passing of procedure parameters. The S- relative mode is typically used for accessing parameters in subroutines.

Not all memory address instructions are capable of using all six modes. The instruction definitions in Section V specify which modes are applicable to a given instruction. Some variation from the above outline of relative addressing can be expected in certain cases. For example, the PCAL, SCAL, and LLBL instructions (not in the memory address group) use PL- relative addressing. Also INCM, LDB, STB, and BCC deviate from this convention in their coding of bit 6.

Throughout this manual and in other HP 3000 documentation, the terms "displacement", "effective address", "relative address", and "base" are used in connection with memory addressing. These terms may be defined as follows: The *displacement* is a positive number which is given in the instruction word and points to a location "plus" or "minus" that number of locations from a given reference cell (also named in the instruction word). The location so



| ADDRESS MODE | INSTRUCTION BITS | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| P+ Relative | 0 | 0 | Displacement 0: 255 | | | | | | | |
| P– Relative | 0 | 1 | Displacement 0: 255 | | | | | | | |
| DB+ Relative | 1 | 0 | Displacement 0: 255 | | | | | | | |
| Q+ Relative | 1 | 1 | 0 | Displacement 0: 127 | | | | | | |
| Q– Relative | 1 | 1 | 1 | 0 | Displacement 0: 63 | | | | | |
| S– Relative | 1 | 1 | 1 | 1 | Displacement 0: 63 | | | | | |

Figure 3-8. Memory Addressing Modes

indicated may or may not be the *effective address*, which is the final computed address, after displacement calculation, indirect addressing (if any), and indexing (if any) have been resolved. The effective address is always an absolute address. The *relative address*, which can be extracted by an LRA instruction, is obtained by subtracting the *base* from the effective address; the base is either the PB address (program base) or the DB address (data base).

Addressing arithmetic is done "modulo 65K" words (i.e., 65,536 word addresses).

## INDIRECT ADDRESSING

One level of *indirect addressing* is permitted. Indirect addressing uses the location referenced by the initial displacement (the "indirect cell") to specify another location within the same code or data segment. In the case of program references, the indirect cell contains a self-relative address. In the case of data references, the indirect cell contains a DB+ relative address. Refer to figure 3-9.

For memory address instructions, indirect addressing is specified by bit 5 of the instruction word: "1" indicates

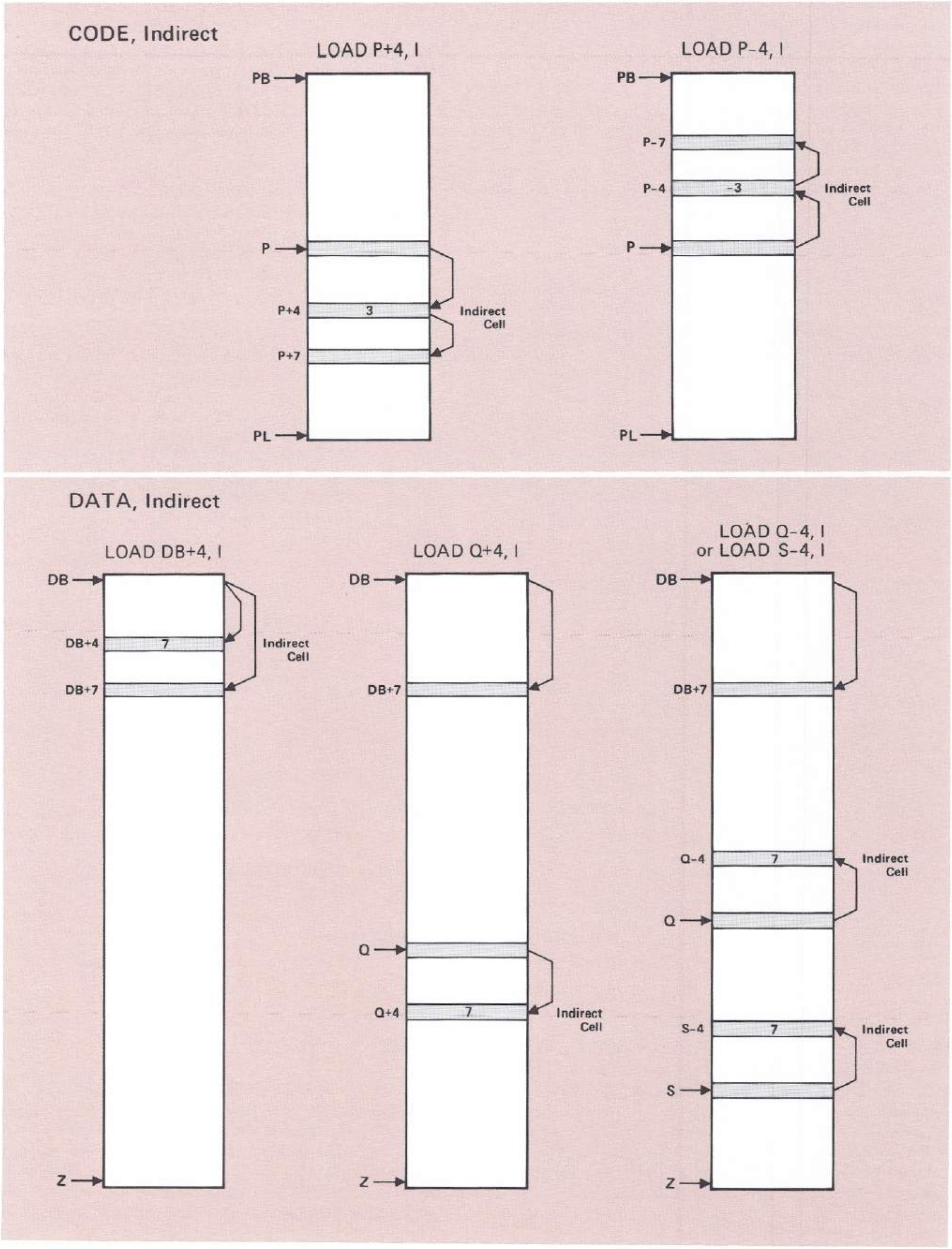


Figure 3-9. Examples of Indirect Addressing

indirect to be used. For the branch instructions (excluding BR), indirect addressing is specified by bit 4. See figure 3-5.

CODE INDIRECT. Figure 3-9 shows both P+ and P- examples of the indirect addressing in a code segment. The first example shows the actions occurring for an assumed instruction of "LOAD P+4, I". The displacement, +4, points to the indirect cell at P+4. The indirect cell contains a self-relative address of +3. This points to a location three addresses higher, or P+7. It is the content of this location which will be loaded onto the TOS by the instruction.

The second example illustrates "LOAD P-4, I". The displacement, -4, points to the indirect cell at P-4. This cell contains a self-relative address of -3, which is 177775 in octal. (The number can be positive or negative.) This points to the location at P-7, which is the effective address for the given instruction.

DATA INDIRECT. The first of the three examples of indirect addressing in a data segment illustrates "LOAD DB+4, I". The displacement, +4, points to the indirect cell at DB+4. This cell contains a DB+ relative address of 7. (This is not a self-relative address.) Thus the effective address is at DB+7. Note that it is possible for the effective address to be below as well as above the indirect cell.

The second data example illustrates "LOAD Q+4, I". The displacement, +4, points to the location four addresses above Q, which is the indirect cell. As in all data indirect cases, the indirect cell contains a DB+ relative address. Since, in this case, the content is 7, the effective address is again DB+7.

The third data example illustrates both the S- and the Q- modes. The displacement is again assumed to be -4, which points to an indirect cell at S-4 (for LOAD S-4, I) or at Q-4 (for LOAD Q-4, I). Since the content of the cell, in both cases, is assumed to be 7, the effective address is again DB+7.

## INDEXING

The content of the Index register is used for *indexing*, when specified by the "X" bit of instruction formats that include indexing capability. When the X bit (bit 4) is a "1", indexing is enabled. The memory address instructions use indexing to modify an operand address. Shift instructions use indexing to modify a shift count, and bit test instructions use indexing to modify a bit position number. The latter two instances are comparatively simple concepts and do not apply to memory addressing; the following paragraphs describe indexing only as it is used in memory addressing.

Figure 3-10 shows some examples of indexing. Unlike figure 3-9, this figure does not illustrate all combinations of cases. Figure 3-10 shows indexing when combined with positive and negative addressing modes (both direct), and an example of indirect, indexed addressing (positive mode only). Examples of these cases are given for both code and data segments. Note that in every case the index is assumed to be 5; this is established by the "LDXI 5" instruction which precedes each LOAD instruction used in the examples. This instruction loads the value 5 into the Index register.

CODE INDEXING. The first example in figure 3-10 shows the actions occurring for an assumed instruction "LOAD P+4, X". The displacement, +4, would by itself point to location P+4; however, by adding the index of 5 to the displacement, the location P+11 (octal) is addressed. It is the content of this location which will be loaded onto the TOS by the instruction.

The second example illustrates indexing with a negative addressing mode, P- in this case. The instruction at P indicates a displacement of 11, which would point at the P-11 location. The index of 5 indexes the address in a positive direction to finally address P-4.

The third code example shows indexing combined with indirect addressing. In all such cases, "post-indexing" is used; i.e., the indirect addressing is accomplished first (whether in a positive or a negative direction), and indexing proceeds in a positive or negative direction from the location so indicated. As shown in the example, the displacement of +4 points to the indirect cell at P+4. The content of P+4 is a self-relative address of 3, which points to location P+7; however, indexing adds 5 to this value, thus pointing at the final effective address at P+14 (octal).

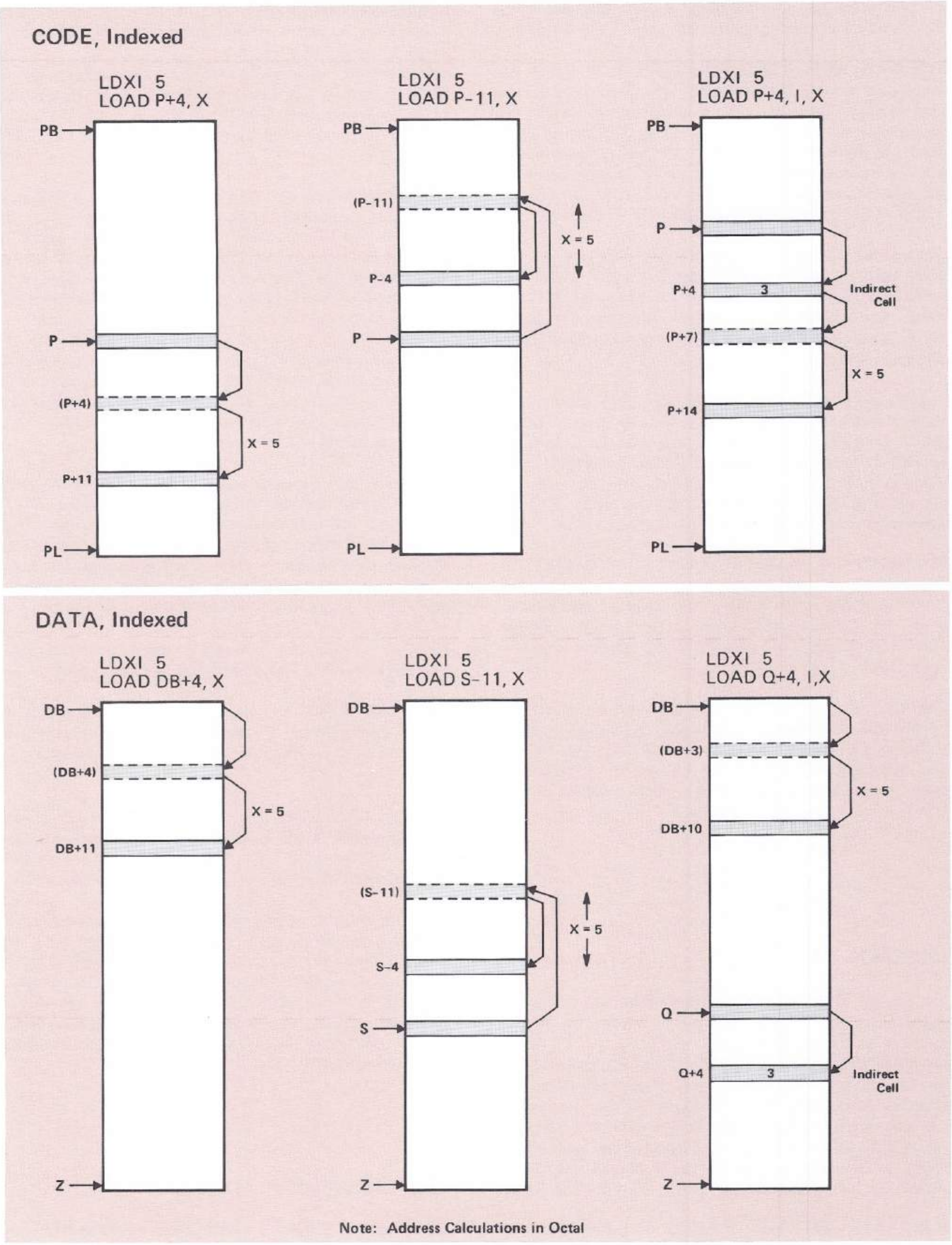
DATA INDEXING. The first data indexing example illustrates "LOAD DB+4, X". This displacement, +4, points at DB+4; this is modified by the index of 5 to point at DB+11.

The second data indexing example illustrates the S- mode, which is similar to the P- mode previously described. Since a positive index is specified, indexing proceeds in a positive direction from the location indicated by the displacement.

The final example illustrates data indexing combined with indirect addressing. Again, post-indexing is applied. The example instruction is "LOAD Q+4, I, X". The displacement, +4, points to the indirect cell at Q+4, which contains the value 3. Since indirect addresses for data are always DB+ relative, this points at location DB+3. This is modified by the addition of the index, 5, thus pointing at the final effective address DB+10 (octal).

## BYTE ADDRESSING

The Load Byte and Store Byte instructions (LDB, STB) and five of the move instructions (MVB, MVBW, CMPB, SCU, SCW) use the *byte addressing* convention. Since the



Note: Address Calculations in Octal

Figure 3-10. Examples of Indexing

HP 3000 central processor is not specifically organized as a byte processor, the byte addressing convention uses the content of the Index register, an indirect cell, or a stack word to specify the byte desired. For memory addressing (LDB, STB), the displacement value remains a word displacement. The byte data label in an indirect cell is an inflated value (two times the word displacement from DB). The contents of the Index register and/or an indirect cell indicate the desired byte in a byte array. For move instructions, one or two of the top-of-stack locations give a PB+ or DB+ relative byte index.

The byte addressing range is therefore restricted to 32K words (15 bits for word address, one for byte number). This implies restricting the stack size to 32K maximum range from DL to S.

Figure 3-11 shows the four different cases of byte addressing for memory address instructions (LDB and STB): direct; direct, indexed; indirect; and indirect, indexed. The convention for move instructions corresponds to the "direct, indexed" case shown in the figure; the difference is that the byte index would be obtained from a top-of-stack word rather than the Index register. The following paragraphs describe each of the four examples.

DIRECT. For direct, unindexed byte addressing, the displacement value given in the instruction word is strictly a word displacement and only the left byte of each word is addressable. As shown in figure 3-11, a "STB DB+7" instruction would store a byte from the TOS into the left byte of the DB+7 location.

DIRECT, INDEXED. The byte index in the Index register is assumed to be 5, established by a LDXI 5 instruction. The "STB DB+7, X" instruction directly addresses location DB+7, and the index of 5 accesses the sixth byte. (Note that the byte index starts at 0; all even indexes are left bytes and all odd indexes are right bytes.)

INDIRECT. In this example the byte index is given in the indirect cell. As in all indirect data addressing, the indirect reference is relative to DB. Thus "STB DB+7, I" initially addresses the indirect cell at DB+7 and the byte index of 46 accesses the 47th byte with respect to DB. This will be the left byte of DB+23. (Since there are two bytes per word, divide the byte index by two to identify the word location; a remainder of 0 indicates the left byte, 1 the right byte.)

INDIRECT, INDEXED. In the indirect, indexed mode, the displacement points to the indirect cell, the indirect cell points to the start of a byte array, and the index in the Index register points to the desired byte in the array. The example in figure 3-11 illustrates "STB DB+7, I, X". The index in the Index register is again assumed to be 5. The displacement points to the indirect cell at DB+7, which contains the value 40. Dividing this by two gives the starting word address of the array, location DB+20. Since

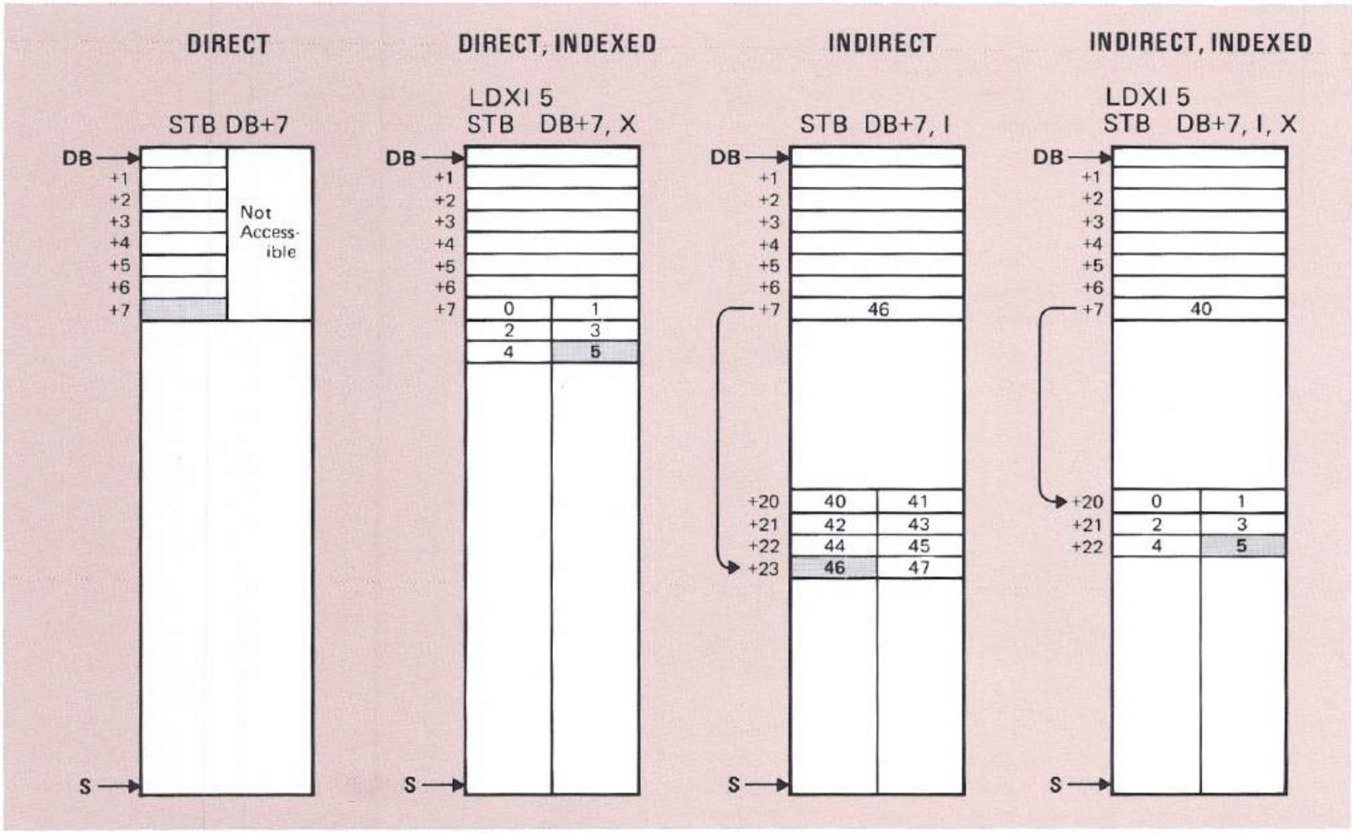


Figure 3-11. Examples of Byte Addressing

the index is 5, the location accessed is the sixth byte of the array. In this manner, the Index register acts like a byte index for ease of stepping through byte strings or byte arrays.

Refer also to the section on byte addressing under the heading "Access to DB- Area".

## DOUBLEWORD INDEXING

Two memory address type instructions, LDD and STD, permit doubleword indexing. When indexing is specified for these instructions, the hardware automatically multiplies the Index register content by two during computation of the effective address. Thus an index value of 4 would imply the fifth doubleword in a doubleword array.

## BOUNDS CHECKING

The central processor routinely checks all address references and top-of-stack movements to ensure that such operations remain within legal bounds. Many of the instruction definitions in Section V define the checks that are made; however the lack of such mention does not necessarily imply that no checks are made.

The following paragraphs summarize the basic bounds checks that occur for the applicable instruction types. Refer to table 3-2 and figure 3-12.

Table 3-2. Bounds Checks

| CHECK | DEFINITION | MODE |
|---|---|---|
| Program Transfer | PB ≤ E ≤ PL | Privileged, User |
| Program References | PB ≤ E ≤ PL | User only |
| Data References | DL ≤ E ≤ S | User only |
| Stack Overflow | SM > Z | Privileged, User |
| Stack Underflow | SM < DB | User only |
| E = Effective Address of Memory Reference | | |

PROGRAM TRANSFER. Program control cannot be passed (via PCAL, SCAL, or a branch) to any location beyond the limits defined by the contents of the PB-register and the PL-register. This rule applies to both privileged and user modes. For indirect branches, both the indirect reference and the direct reference must be within limits. This

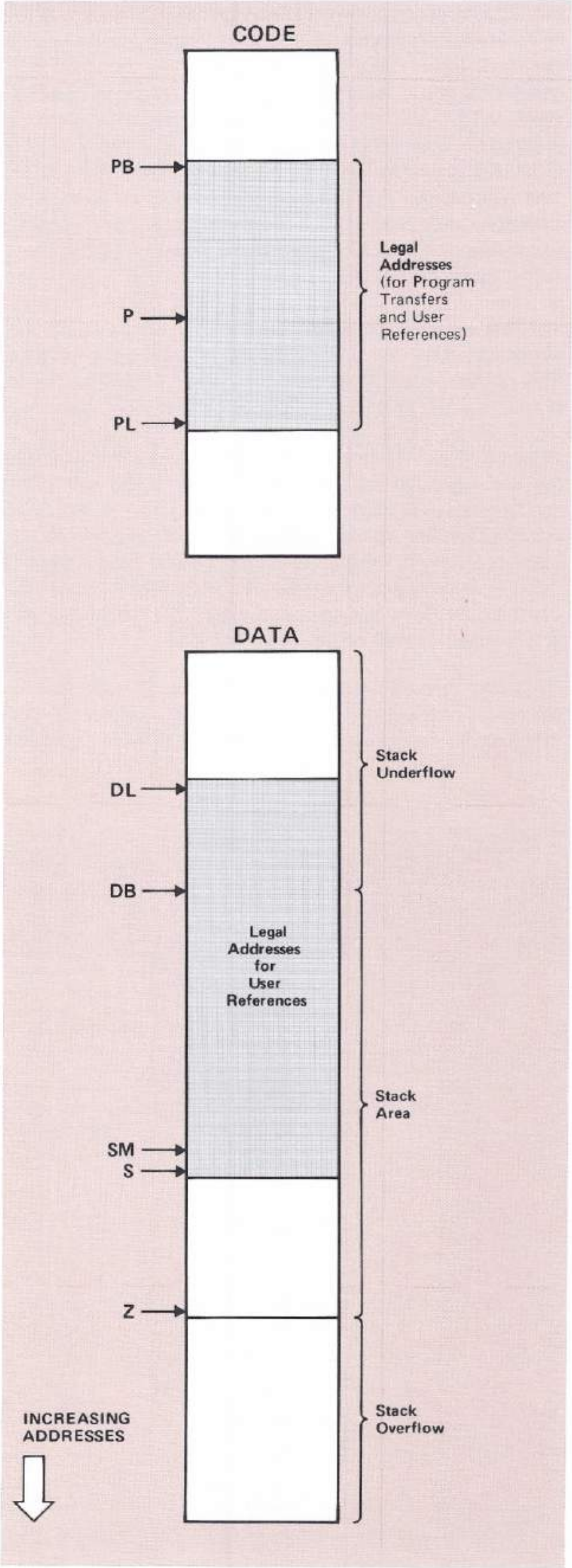

CODE
PB
P
PL
Legal Addresses (for Program Transfers and User References)
DATA
DL
DB
Legal Addresses for User References
SM
S
Z
Stack Underflow
Stack Area
Stack Overflow
INCREASING ADDRESSES

Figure 3-12. Addressing and Stack Bounds

also applies when branching indirect via the stack (see BR definition), except that the initial reference must be within the stack limits (DB,S) rather than within PB and PL. A bounds violation causes a Bounds Violation interrupt to segment 11.

PROGRAM REFERENCES. Some of the memory address instructions, all of the loop control instructions, some of the move instructions, and a few others, are capable of addressing locations in the code segment. In privileged mode, such references may be made without restriction. However, in user mode, the references (both direct and indirect) must be within the limits defined by PB and PL. A bounds violation causes a Bounds Violation interrupt to segment 11.

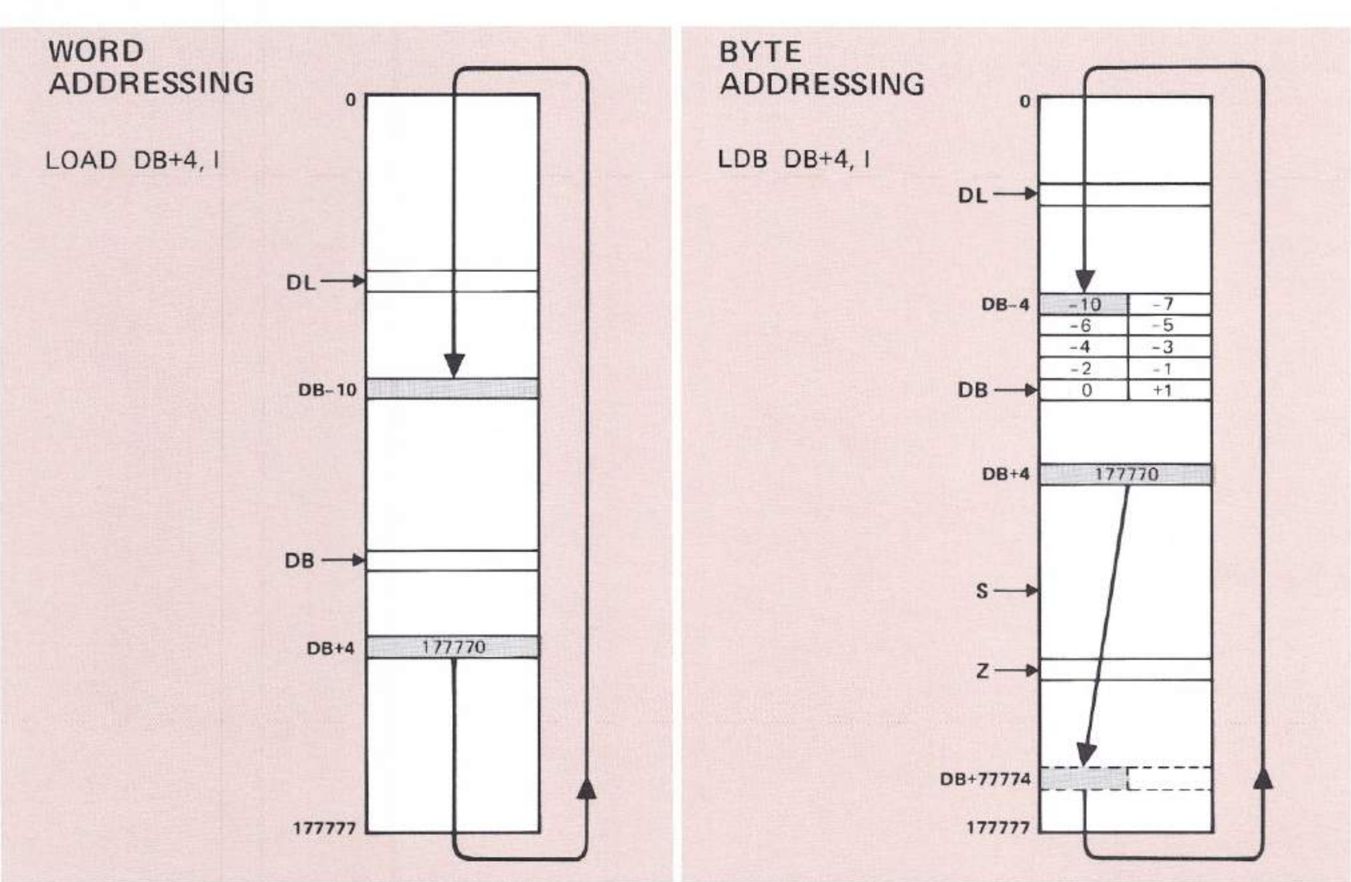
DATA REFERENCES. In privileged mode, data references are not subject to bounds checking. In user mode, data references (both direct and indirect) must be within the user's defined data area — that is, between DL and S. A bounds violation causes a Bounds Violation interrupt to segment 11.

STACK OVERFLOW. Neither privileged mode nor user mode may overflow the stack. A *stack overflow* is defined as the condition of moving the top-of-stack pointer beyond the stack limit. In a stricter sense, stack overflow occurs when SM exceeds Z. Since SM is not necessarily the actual top of the stack (may be coincident with S or up to four locations lower), and to allow marker space for the remote possibility of a procedure call and an interrupt while SM is at Z, there is a zone of about 13 locations beyond Z which could be filled with stack related data. A stack overflow causes an interrupt to segment 3, which, under the discretion of the operating system, may extend the stack limit.

STACK UNDERFLOW. A *stack underflow* is defined as the condition of moving the top-of-stack pointer below the data base or, more strictly, moving SM below DB. Since SM may or may not be coincident with S, underflow may occur even though S may be up to three locations above DB. Privileged mode is not subject to underflow checking. A violation in user mode, however, will cause a Stack Underflow interrupt to segment 13. Users can access the area between DL and DB by indirect addressing or indexing, as long as SM does not become less than DB.

## ACCESS TO DB- AREA

Both privileged and user modes have access to the data area between DB and DL through indirect addressing and index-



Address Calculations in Octal: **WORD** DB + 177770 = DB - 10
**BYTE** DB + (177770 ÷ 2) + 100000 = DB - 10

Figure 3-13. Access to DB- Area

ing. The privileged mode additionally has direct access by the privileged move instructions MVBL and MVLB. Figure 3-13 illustrates the technique of indirect addressing to access this area, using both word and byte examples.

WORD ADDRESSING. The left part of figure 3-13 shows how to access a word in the DB- area. Assume that we wish to load the contents of the location at DB-10 onto the stack, and that location DB+4 can be used for the indirect cell. Thus a "LOAD DB+4, I" instruction initially references the indirect cell at DB+4. The indirect cell contains, instead of a positive number, the two's complement of the desired DB displacement. In octal, the two's complement of 10 is 177770. Remember that the content of an indirect cell in a data segment is always a DB+ relative displacement. Thus, since addressing arithmetic is modulo 65K, adding 177770 to DB causes "wrap-around" and addresses the desired DB-10 location. (Indexing via the Index register may be applied from this point.)

BYTE ADDRESSING. The right part of figure 3-13 shows the technique of accessing a byte in the DB- area. Assume that we wish to load the DB-10 byte onto the stack, and that location DB+4 will again be used as the indirect cell. The "LDB DB+4, I" instruction initially references DB+4, which contains, instead of a positive byte number, the two's complement of the desired byte displacement from DB. In octal, the two's complement of 10 is 177770. Remember that byte indexes are converted to word indexes by dividing by two. This would indicate location DB+77774 (left byte), which may or may not exceed the upper limit of memory, depending on the current absolute value of DB.

To allow for byte addressing in additional data segments where DB may not be between DL and Z, a check for this condition is made. If DB is not between DL and Z (this should happen only in privileged mode and is then called *split stack*), the byte will then be accessed without further bounds checking. If, however, DB is between DL and Z, then in either mode the LDB instruction (or other byte addressing instruction) tests this address to see if it is within the required DL to Z range. If the address is not within this range (which should be the case, whether wrap-around has already occurred or not), the instruction will add 32K (100000 in octal) to the DB+77774 value. Assuming that wrap-around had not yet occurred, this addition would certainly cause wrap-around and thus address the byte at byte address DB-10 (left byte in location DB-4).

At this time, a second test is made to see if the effective address is in the DL to Z range. If the technique has been applied properly, the test will be affirmative and the byte will be transferred. However, if the second test fails, the action taken will depend on the current mode. In user mode, there will be a Bounds Violation interrupt to segment 11. In privileged mode, the result of the second test is ignored; execution continues even if our of bounds, using the second referenced byte.

# MEMORY SEGMENTATION

SECTION IV

As part of its basic architecture, the HP 3000 Computer System organizes all code and data into variable length segments which may be swapped in and out of main memory on demand.

The first half of this section ("Introduction" and "Code and Data Segments") describes the basic theory of segmentation. Since the mechanics described are automatically controlled by operating system software, the information is presented primarily as background material.

The second half of this section, however, ("Stack Operation" and "Examples of Stack Usage") focuses on the stack portion of a data segment. Since an understanding of the stack concept is essential to the overall system concept, the latter half of this section illustrates the principles of stack operation in detail at a fundamental level.

## INTRODUCTION

It is the purpose of this introduction to provide a bridge from the overall "system" viewpoint into the functionings of the hardware, as regards memory operations. Therefore no attempt will be made to explain the concepts of jobs and processes, any more than is necessary for the following discussions. The reader should refer to separate documentation for the software systems, if full definitions of these concepts are required.

First it is necessary to establish what is meant by *virtual memory*. As shown in figure 4-1, virtual memory consists of *primary memory* (the main memory) plus an area of mass storage called *secondary memory*, or the *swapping area*. The swapping area, typically on disc or drum memory, consists of a collection of pieces of code or data, defined as segments, which are not presently in core but which may be called in by the executing programs. A *segment* is the basic entity for transfers between core memory and the swapping area. Whether a segment is in main memory or *absent* (on disc), it is nevertheless part of the virtual memory. From the point of view of the user, he is working with a memory that appears to be many times larger than actual physical size. In fact, his own program may exceed the 65K-word maximum of main memory capacity, and still allow space for many other users on the same machine.

At this point the reader should be visualizing a dynamic situation in which various segments are being swapped rapidly between core memory and the swapping area of disc memory, according to the demands of the executing programs. Also bear in mind that several users may be on the machine at a given time, and that each user may have several segments.

Now the questions arise: where did the segments come from (i.e., how were they created), and how are they eventually eliminated? To answer these questions it is necessary to understand that there are two distinct types of segments, code segments and data segments. Thus there are two methods of origin. See figure 4-2.

A *code segment* consists entirely of information that is not subject to change during program execution. This includes the instructions of the program itself, constants, and an area for interprocedure links. No modifiable data may be interspersed with the instructions in a code segment, and in no way is it possible to write into or alter a code segment (or its formative parts) once it has been compiled. It is this feature which allows code to be *re-entrant*, meaning that a given sequence of instructions can be in simultaneous use by several users — or, can be entered several times by the same user, whether or not preceding entries are concluded. An example at the end of this section (Recursion) will illustrate a procedure which, after being entered by the main program, will call itself several times before any exit is given.

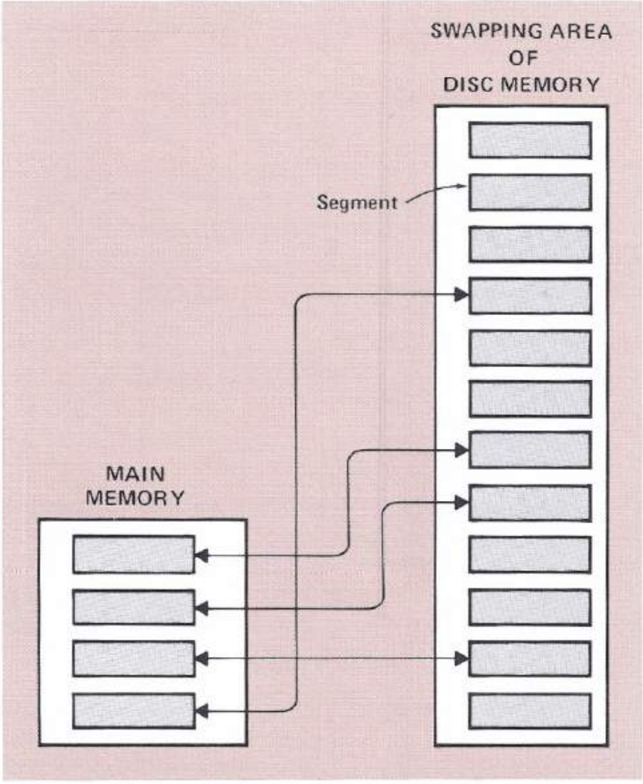


Figure 4-1. Virtual Memory

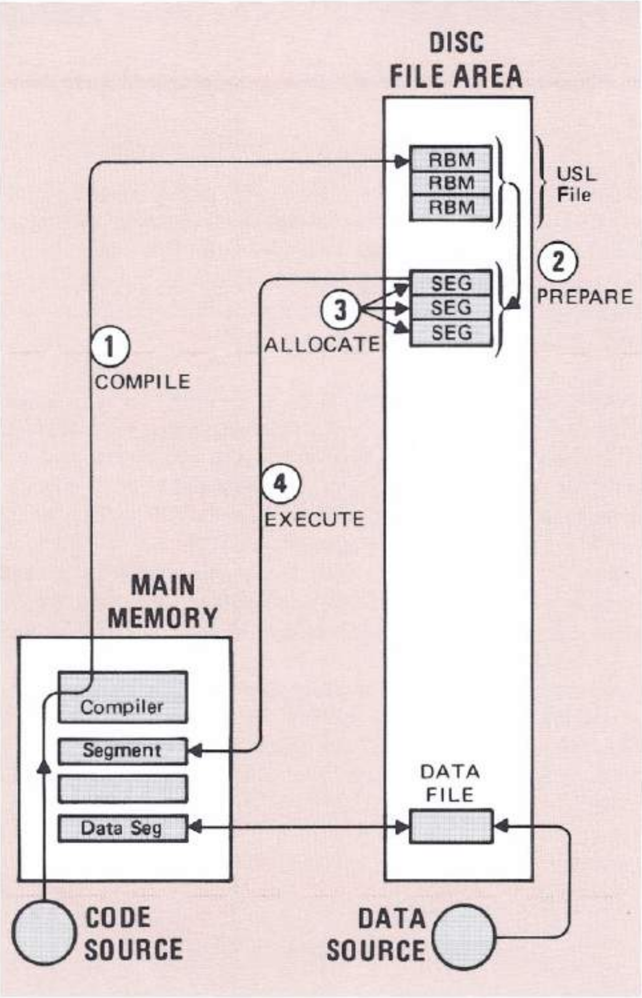


Figure 4-2. Sources of Segments

As shown in figure 4-2, user code entered into the computer exists in one of four states at various points in time. Initially, the user programs exist as source-language code. Then (step 1 in figure 4-2), the programs are translated into binary form by a process executing a compiler, and stored in the *file area* of disc memory. Each compiled program or subprogram exists in the file area as a *relocatable binary module* (RBM); the set of RBM's that result from compilation of a user's program onto disc make up a *user subprogram library* (USL) file.

The USL is not executable, however. Instead, it must be "prepared" for running (step 2 in figure 4-2). During preparation, the operating system binds the RBM's from the USL into linked code segments arranged in a *program file*. Each segment contains machine instructions produced from the user's program, plus linkages to other segments.

The next step (3, in figure 4-2) is to *allocate* the program when the user gives the command to run his program. In allocation, the operating system links the code segments to the *Code Segment Table*. Every allocated segment has an entry in the Code Segment Table, which is a set of reserved locations in main memory that tells both the hardware and the software exactly where each code segment is located. The table lists a memory address if the segment is main-memory resident, or a disc address if disc resident, plus the segment length. It is maintained by the operating system.

During allocation, the operating system also binds the segments from the program file to referenced external segments from a library. Once the segments are allocated, the USL becomes part of the virtual memory, and execution can begin. The operating system creates a process to run the program and individual segments are swapped into main memory for execution (step 4 in figure 4-2).

The *data segment*, also shown in figure 4-2, consists only of data. Like the code segment, a data segment is fully protected. No user (more strictly, no process) may have access to the data segment of another user (or process). Generally speaking, each process defined by a user causes a data segment to be created. Initially, when the code segments are allocated, the data segment contains no actual data, but consists only of an initial stack having some initializing information. (Stack is defined later.) But at least the data segment is allocated — that is, a place for data is established.

Like code segments, data segments have entries in a table, called the *Data Segment Table*, which keeps track of where each data segment is located. Unlike the Code Segment Table, however, the Data Segment Table's location is known only to the operating system software.

As execution progresses, data will enter and leave the data segment — perhaps as the result of various computations, or perhaps via an external data source.

Eventually the last instruction in a given process will be executed. At that time the operating system will *deallocate* all segments associated exclusively with that process. That is, they will lose their entries in the Code Segment Table and the Data Segment Table, and the respective code and data will be overlaid by other segments coming into the system. For a time, of course, the old code and data will physically continue to exist in the virtual memory, but there is no means by which this information can be retrieved. Thus if there is some information to be saved as the result of process execution, the process itself must save such information in the file area.

Referring back to figure 4-1, the reader should at this point be able to visualize not only the swapping of segments in and out of main memory, but also the creation and elimination of various segments as new user processes come into the system and other processes come to an end. Obviously the areas occupied by segments in both main memory and disc memory will dynamically shrink and expand according to demands placed on the system. (To maintain optimum efficiency, the operating system has a timer and method of keeping usage statistics, so that the less important or less frequently used segments are most eligible for temporary swapping out to the disc.)

Now that the basic concept of a segment has been introduced, it is possible to show how the segment fits into the overall scheme of things.

Figure 4-3 is an overview of the major system elements. This figure shows the software that might exist in the hardware at a given instant of time. It does not attempt to show the possible links between elements, nor the relationships that can exist among various processes. It is simply a snapshot view of elements, showing location and constitution. Note that the software exists either (or both) in main memory or in mass storage.

The following paragraphs describe each of the elements shown in figure 4-3.

## RESERVED MEMORY

Only 12 memory locations are "reserved" in the strictest sense — i.e., having a known, fixed address. These are the first 12 addresses. See table 4-1. In addition, however, there is also a permanent table which is reserved in the sense that, once established, each entry has a permanent allocation. The upper limit of the table, however, is flexible, depending on how many entries there are in the table. This table is the Device Reference Table (to be defined and discussed in a later section). It begins at octal location 14 and uses four locations for each device existent in the system.

The 12 fixed memory allocations can be divided into three groups of four locations each. In the first group, location 0 contains the *Code Segment Table Pointer*, which is the absolute address of the first entry in the Code Segment Table. Location 1 contains the *Data Segment Table Pointer*, location 2 contains the *Process Control Block Table Pointer*, and location 3 contains the *System Global Pointer*. (Note: these are dynamic assignments; for cold load operations, the hardware expects a cold-load value for the P-register in location 1.)

The second and third groups each apply to separate processors, if a dual-processor system is used. Locations 4 through 7 provide a Current Process Control Block pointer, two interrupt stack pointers, and an interrupt reference counter for processor 1. Octal locations 10 through 13 provide the same for processor 2. The Current Process Control Block pointers will be discussed in this section under the heading "Data Segments", and the interrupt stack pointers and counters will be discussed in the section on interrupt processing.

Table 4-1. Fixed Memory Allocations

| LOCATION | CONTENTS |
|---|---|
| 0 | Code Segment Table Pointer |
| 1 | Data Segment Table Pointer |
| 2 | Process Control Block Table Pointer |
| 3 | System Global Pointer |
| 4 | CPCB Pointer 1 |
| 5 | QI 1 |
| 6 | ZI 1 |
| 7 | Interrupt Counter 1 |
| 10 | CPCB Pointer 2 |
| 11 | QI 2 |
| 12 | ZI 2 |
| 13 | Interrupt Counter 2 |
| 14 | First Entry Device Reference Table (Device #3) |
| 15 | |
| 16 | |
| 17 | |

## SEGMENTED LIBRARY

A *segmented library* is a flexible means of sharing frequently used routines among many users. In addition to standard library routines, the user may enter and delete routines of his own in the libraries.

A library might be one procedure in a segment, a set of procedures in a segment, or a set of segments. As shown in figure 4-3, some segments which contain certain library routines are permanently allocated. That is, they have entries in the Code Segment Table. Other library segments remain in the file area until such time that a user makes a request for one of its routines. At that time the operating system will load the affected segments, create entries in the Code Segment Table, and provide appropriate links for the user to access the desired routine.

## OPERATING SYSTEM

The *operating system* is the master supervisory program, overseeing the allocation of memory, controlling the loader, swapping user segments in and out of main memory, designating time to individual users, and so on. The standard operating system for HP 3000 is the Multiprogramming Executive (MPE/3000). It consists of a number of separate programs and many procedures in the system segmented library file.

As indicated in figure 4-3, not all parts of the operating system need to be permanently resident in main memory. Certain modules may be retained in the file area, and be allocated on a requirement basis.

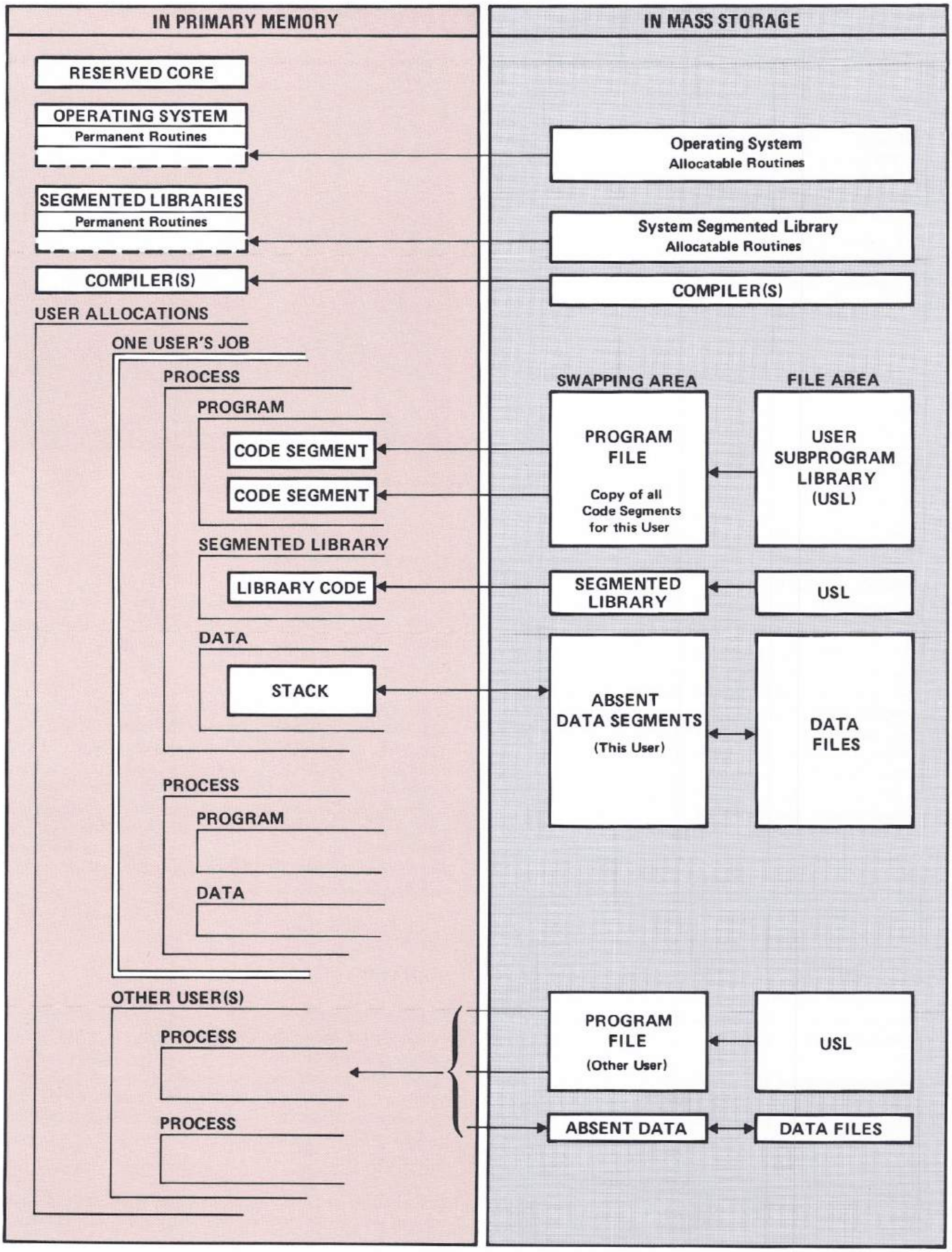


Figure 4-3. Software Elements in Hardware

### COMPILERS

Several language compilers are available. If only one compiler is used on a system, it might be designated to be permanently allocated. For a multicompiler system, however, it is more efficient to retain the permanent copy of each compiler in the file area, and to allocate the compiler in virtual memory only when required. Due to the re-entrant feature for all programs run on this computer, only one copy of a compiler needs to be present in main memory, regardless of how many users may be simultaneously compiling. The operating system keeps a count of how many users are using a given compiler, and when this count reaches zero, the compiler is deallocated.

### USER ALLOCATIONS

As shown in figure 4-3, the remaining space (after allocations for reserved memory, library, operating system, and compilers) is available for users. This space includes both main memory space and disc space. Bear in mind that the relative block sizes in figure 4-3 do not indicate comparative sizes of space; the file and swapping areas, for example, may be many times the size of any allocation in main memory.

USER JOB. When a user logs into the system, he establishes a *job*. During the course of his job he will execute one or more *programs* upon information contained in separate and distinct *data domains*. The user's data domain consists of all data that is used or generated during the course of a job.

PROCESS. A program is executed on the basis of individual processes. A *process* is not a program itself, but the unique execution of a program by a particular user at a particular time. Therefore, if the same program is run by several users, or more than once by the same user, it is used in several distinct processes.

The process is the basic executable entity in the system. It consists of a *Process Control Block* that defines and monitors the state of the process, a dynamically-changing set of code segments and a data area (stack) upon which these segments operate. The code segments used by a process can be shared with other processes, but its data stack is private (though the operating system does provide for communication of data between related processes).

Processes will again be mentioned under the headings of "Code Segments" and "Data Segments", but further details regarding their relationships, substates, priorities, means of data communication, queuing, dispatching, etc., are extraneous to the present discussion. Refer to the operating system documentation for this type of information.

CODE SEGMENT. Code segments were defined earlier as consisting primarily of instruction code, and being the basic entity for transfers of code between main memory and the swapping area. As shown in figure 4-3, a program may consist of several code segments.

One important point to note about code segments is that, since code cannot be changed after it is compiled, the copy of a segment in the swapping area is identical with any copy that has been transferred into core. Thus when the operating system decides to swap out a code segment, no actual transfer needs to take place. The operating system simply makes note that the segment is now absent, and may then overlay the core area occupied by that segment. This is unlike the data segment which, being constantly subject to change, must be physically transferred to disc if swapping is required. (Note unidirectional arrows for code segment swapping and bidirectional arrows for data segment swapping in figure 4-3.)

## CODE AND DATA SEGMENTS

The preceding introduction provided a bridge between the external aspects of the system and the inner workings of the hardware, which now follow. Attention is to be focused on main memory, regarding the swapping area only as a place where segments can be sent when main memory becomes too crowded.

At first, memory will be viewed as a whole, as a repository for some number of segments — whether they be code or data — with perhaps some spaces between. It will be shown how space is managed in an orderly and efficient manner. Following this, code segments and their interrelationship during execution will be discussed, followed finally by data segments and the stack concept.

### SEGMENTS IN MEMORY

Figure 4-4 shows four segments being present in memory. These are "assigned" segments. There are also three blank or "free" segments. In each case, whether assigned or free, the first eight locations of the segment comprise an 8-word link header, and the last location of the segment gives the size of that segment, divided by four. (The last word of the segment effectively points back to its header, and specifies the type of the preceding area — i.e., assigned or free.)

To assist the operating system in its task of filling memory with variable sized segments, the memory is threaded with two major systems of links. These are the *assigned memory links* and the *free space links.* The assigned memory links consist of pointers within the link headers of each assigned segment, which link all assigned segments. Similarly, the free space links consist of pointers within the link headers of each free segment, thus linking all free segments.

Linking pointers are given for both the forward direction and the backward direction. That is, one word in the header points ahead to the next assigned link (or next free link, as the case may be), and another word points back to the previous assigned link (or last free link). Note, as shown in the figure, that the links are not arranged in sequence of ascending memory addresses, but rather weave through memory in seemingly random manner. Actually, the assigned links are arranged according to usage statistics, so that the least used segments are at the head of the list and are thus most susceptible to overlay. The links are updated for this purpose each time an overlay is performed.

Other information given in the link headers includes: segment type, disc address, segment size, and process number. All of this information is used in memory management.

In typical operation, if the currently executing code requests an absent segment, the operating system will first obtain the size of the called segment from the Code Segment Table. It will then use the LLSH instruction to search through the free space list to see if there is a large enough free segment to accept the called segment. If this search fails, one or more assigned segments are selected for overlay.

For clarity, figure 4-4 shows only the next-assigned-link structure for the assigned list (and the pointer to the head of the free list). The free list links and the previous-assigned-link structure of the assigned list are not shown. The segment numbers shown are for reference purposes only; e.g., assigned segment number 0 points to assigned segment number 1, and so on. However, the pointers consist of absolute addresses, not segment numbers.

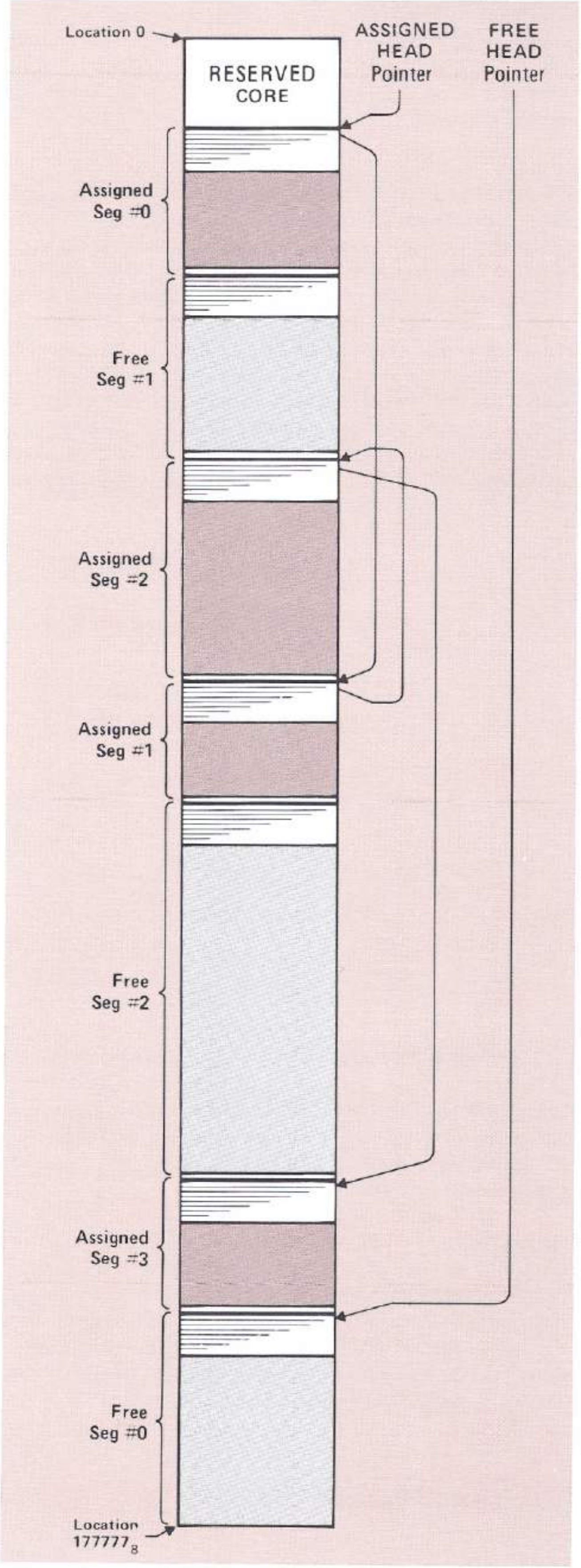


Figure 4-4. Main Memory Links

## CODE SEGMENTS

During the execution of one user's process, there will typically be several code segments in memory and a single data segment. Assume that the current process presently has two code segments in memory, as shown in figure 4-5. (The data segment, not shown, will be discussed later.)

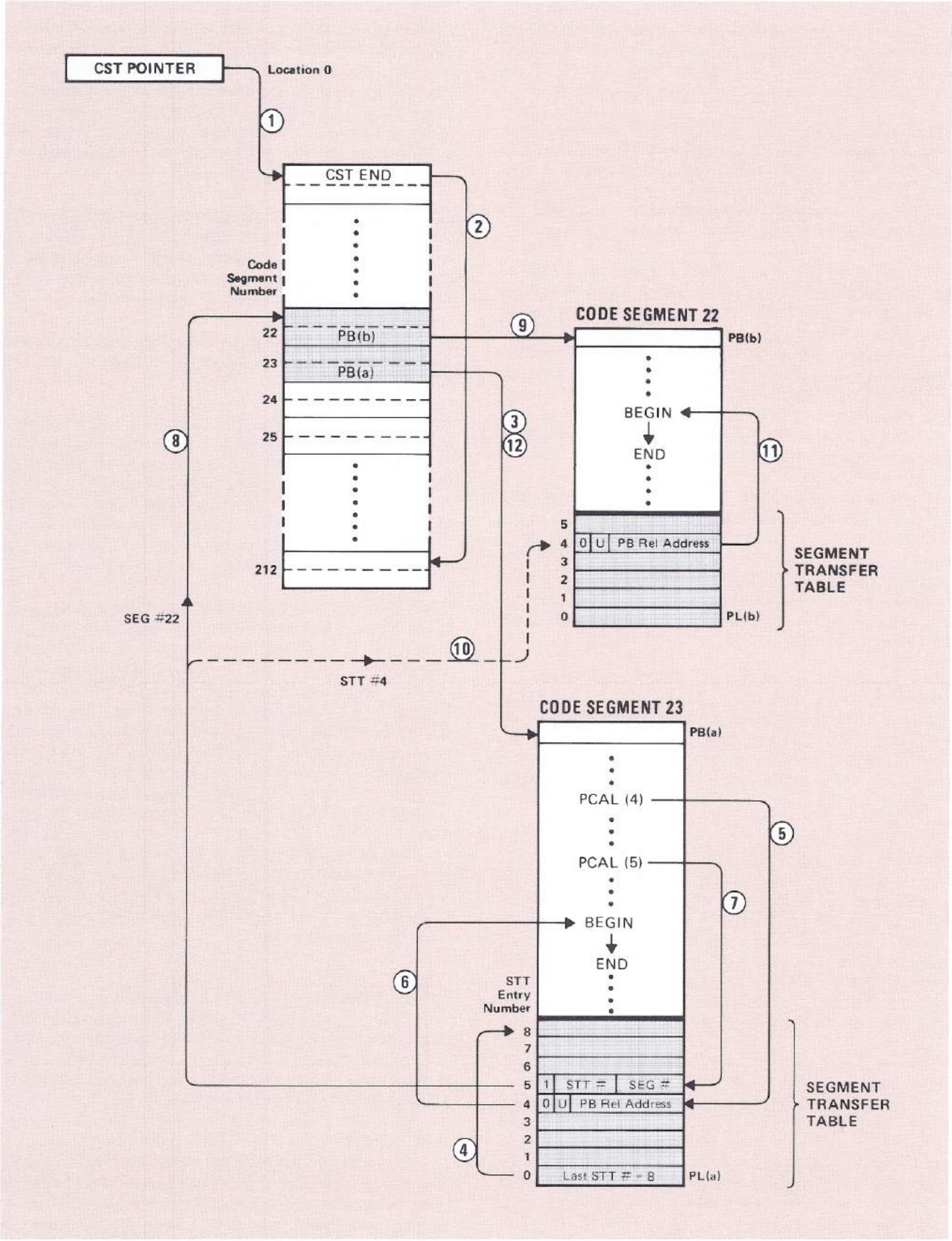


Figure 4-5. Procedure Calls Within and Between Code Segments

The purpose of figure 4-5 is to show how the system keeps track of where code segments are, and how references may be made from one segment to another. Although the figure illustrates hardware, it remains the responsibility of the operating system to control the actions shown here.

The Code Segment Table and the CST Pointer have both been mentioned before. In summary, it was explained that the CST Pointer is permanently resident in location 0, and that it contains an absolute address pointing (1) to the starting location of the Code Segment Table. This table tells where each code segment (present or absent) is located.

Each entry in the Code Segment Table has a unique number, called the *code segment number*, which identifies a particular segment. Each entry consists of a doubleword descriptor which includes the absolute address of the related segment and its length. (The format of CST entries is given in figure 4-6.) Entry number 0 in the table is unique in that it simply points (2) to the final entry in the table; this defines the length of the table for the benefit of the operating system in allocating core space for the table itself. Segment number 0 does not exist.

The example Code Segment Table in figure 4-5 presumably has 212 entries for all code segments of all users currently on the machine. Assume that one user is executing a process which requires code segments 22 through 25. Segments 22 and 23 are in core, since there has been a reference that has caused them to be brought in, whereas segments 24 and 25 are not presently needed and so are absent on disc.

The process is currently executing instructions in segment 23. This means that the address value contained in the second word of CST entry 23 has been loaded into the PB-register. Thus the PB-register is pointing (3) at PB(a). The PL-register, using a value derived from the segment length, is pointing at PL(a). The P-register is advancing from PB(a) toward PL(a).

The last nine locations of segment 23 are not part of the segment's code, but were added by the operating system when the segment was loaded into the virtual memory. This is the *Segment Transfer Table*, which contains linking references for every *procedure call* in the segment. A procedure call is an instruction which references a set of instructions elsewhere in the code segment; that set of instructions is structured as a *procedure*, to perform a standardized operation or computation and then return control to the instruction immediately succeeding the call instruction.

Note that entries in the Segment Transfer Table are numbered from the end back towards the code. Entry number 0 gives the Segment Transfer Table length (see STT Length word format in figure 4-6). This indicates (4) the number of the last STT entry, so that the hardware can make validity checks on procedure call references; for example a call to entry number 9 would be invalid. (If a call from within the segment is made to entry 0, the reference will be taken from the top of the stack instead of from the Segment Transfer Table. A call from outside a segment to entry 0 starts execution at the P = PB after checking the U bit.)

When the execution sequence reaches the first PCAL instruction, a reference is made (5) to the fourth entry of the Segment Transfer Table; i.e., since the PCAL instruction uses PL– addressing, the instruction references cell PL–4. This location contains a *local program label* (see format in figure 4-6), which implies that the called procedure is located within the same segment. The reference is a PB relative address pointing (6) to the beginning of a procedure or block.

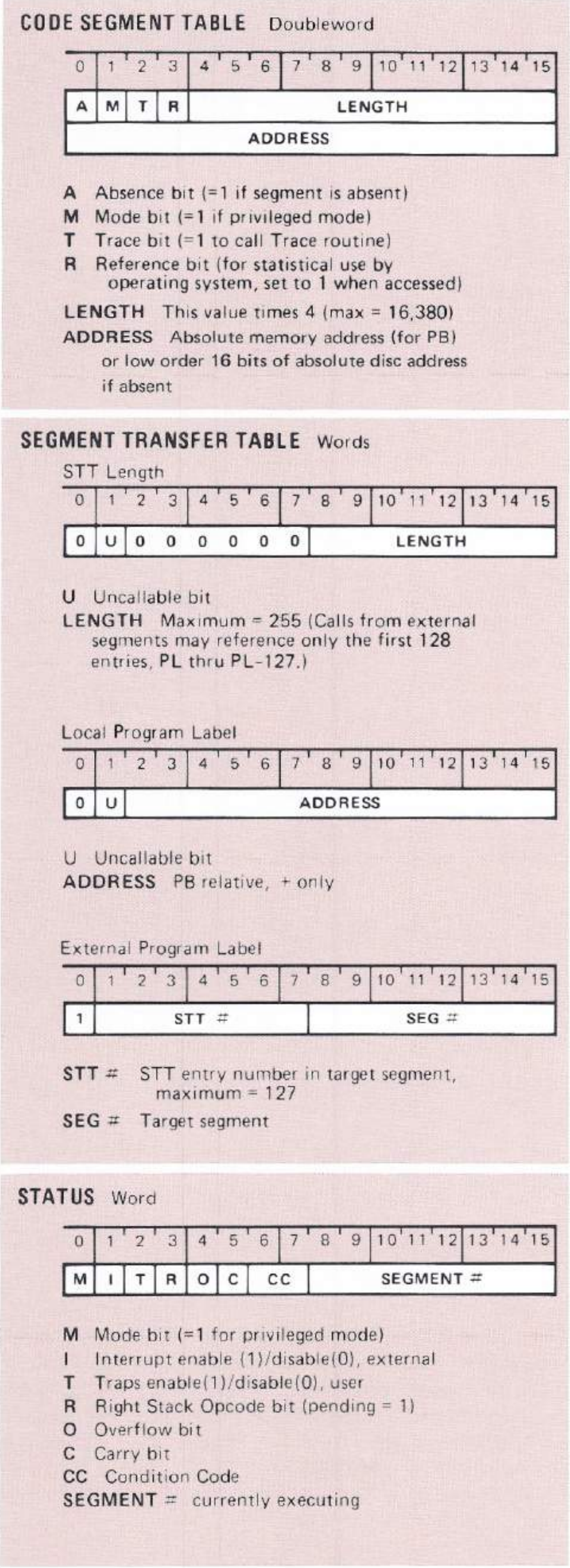
After some preparatory operations, which include saving the return address on the stack, the PCAL instruction transfers control to the procedure. Upon encountering an EXIT instruction in the procedure, control returns to the instruction immediately following the first PCAL.

In this example there were no references outside the current segment. In the following example an external reference is made.

When the execution sequence reaches the second PCAL, another call is made (7) to the Segment Transfer Table. The call requests the fifth entry in the table, which happens to be an *external program label*, indicated by a "1" in bit 0 (see format in figure 4-6). This implies that the called procedure is in some other segment. The contents of the label tells which segment, and also gives the STT number in that segment which must contain the local reference.

The PCAL instruction, after the usual preparatory operations (which include bringing the segment into main memory if it is absent), transfers control to the called procedure as follows. The segment number given in the external program label points (8) to a specific entry in the Code Segment Table; this is assumed to be entry number 22. A value for PB is picked up in the second word of this entry, and is loaded into the PB-register. This causes the PB-register to point (9) to the starting location of code segment 22 (PB(b)). The limit (PL(b)) is also established. Meanwhile, the STT value given in the external program label is pointing (10) to entry number 4 of the Segment Transfer Table. This causes a PB relative address to be picked up for the P-register. The P-register now points (11) to the starting address of the procedure or block, and execution begins. (If an STT number of 0 is given, execution would start at PB(b).)

Calling procedures outside of the segment in this manner is subject to a number of rules, checks, and safeguards. These ensure that the call is allowable, and that other users are fully protected from deliberate or accidental invasions of privacy. The way in which the operating system sets up the Segment Transfer Tables ensures that all transfers are legal for that process. Even if the user transfers via the top-of-stack reference into another user's code segment (assuming that it is callable) he can do no worse than execute part of that other segment. He will certainly render his own stack data meaningless, and furthermore can in no way read or relocate the other user's code or data. His end result is completely unpredictable, but would likely eventually invoke one of many possible error traps.



**CODE SEGMENT TABLE** Doubleword

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | M | T | R | LENGTH | | | | | | | | | | | |
| ADDRESS | | | | | | | | | | | | | | | |

**A** Absence bit (=1 if segment is absent)
**M** Mode bit (=1 if privileged mode)
**T** Trace bit (=1 to call Trace routine)
**R** Reference bit (for statistical use by operating system, set to 1 when accessed)

**LENGTH** This value times 4 (max = 16,380)
**ADDRESS** Absolute memory address (for PB) or low order 16 bits of absolute disc address if absent

**SEGMENT TRANSFER TABLE** Words

STT Length

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | U | 0 | 0 | 0 | 0 | 0 | 0 | LENGTH | | | | | | | |

**U** Uncallable bit
**LENGTH** Maximum = 255 (Calls from external segments may reference only the first 128 entries, PL thru PL-127.)

Local Program Label

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | U | ADDRESS | | | | | | | | | | | | | |

U Uncallable bit
**ADDRESS** PB relative, + only

External Program Label

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | STT # | | | | | | | SEG # | | | | | | | |

**STT #** STT entry number in target segment, maximum = 127
**SEG #** Target segment

**STATUS** Word

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M | I | T | R | O | C | CC | | SEGMENT # | | | | | | | |

**M** Mode bit (=1 for privileged mode)
**I** Interrupt enable (1)/disable(0), external
**T** Traps enable(1)/disable(0), user
**R** Right Stack Opcode bit (pending = 1)
**O** Overflow bit
**C** Carry bit
**CC** Condition Code
**SEGMENT #** currently executing

Figure 4-6. Formats Associated with Code Segments

In addition, if the operating system ascertains that a local reference in a segment is of a category that will not normally have external references to it, the operating system will set the *uncallable bit* in the STT entry. When this bit is set, no external references in user mode may be made to that procedure or block. One typical application of this bit is to prohibit direct user access to the uncallable intrinsics of the operating system — i.e., those operations that the operating system will perform on behalf of a user, but cannot be directly accessed by the user.

At the conclusion of the called procedure, control is returned to the original segment by the EXIT instruction. This instruction restores the Status register, which gives the segment number of the caller (see format in figure 4-6), and thus (12) returns the PB-register value back to PB(a). The saved P relative address on the stack re-establishes the return point, and execution continues at the location immediately following the second PCAL instruction.

## DATA SEGMENTS

In the introductory paragraph under "Code Segments" it was stated that one user's process typically has several code segments, but only one data segment. The following few pages deal with the data segment, particularly concentrating on the stack area of that segment.

As a beginning point of reference, figure 4-7 shows how the operating system establishes and keeps track of a particular data segment. As indicated by a note in the figure, this is accomplished by tables maintained by — and known only to — the operating system.

Assuming we are working with processor number 1 of a single- or dual-processor system, core location 4 contains the *Current Process Control Block pointer*. In the example shown, this pointer (1) has selected process number 31 by pointing to that particular block in the Process Control Block table. This means that process number 31 is currently being executed on the machine.

The Process Control Block contains considerable information pertaining to the control of that process, such as priority, queue pointers, wait flags, and so on. In addition, there is other information, such as saved stack register values (2), which is actually contained within the segment. This area of the segment is the *Process Control Block Extension*.

However, relevant to the present discussion, the most significant information is the data segment number. The data segment number points (3) to a doubleword descriptor in

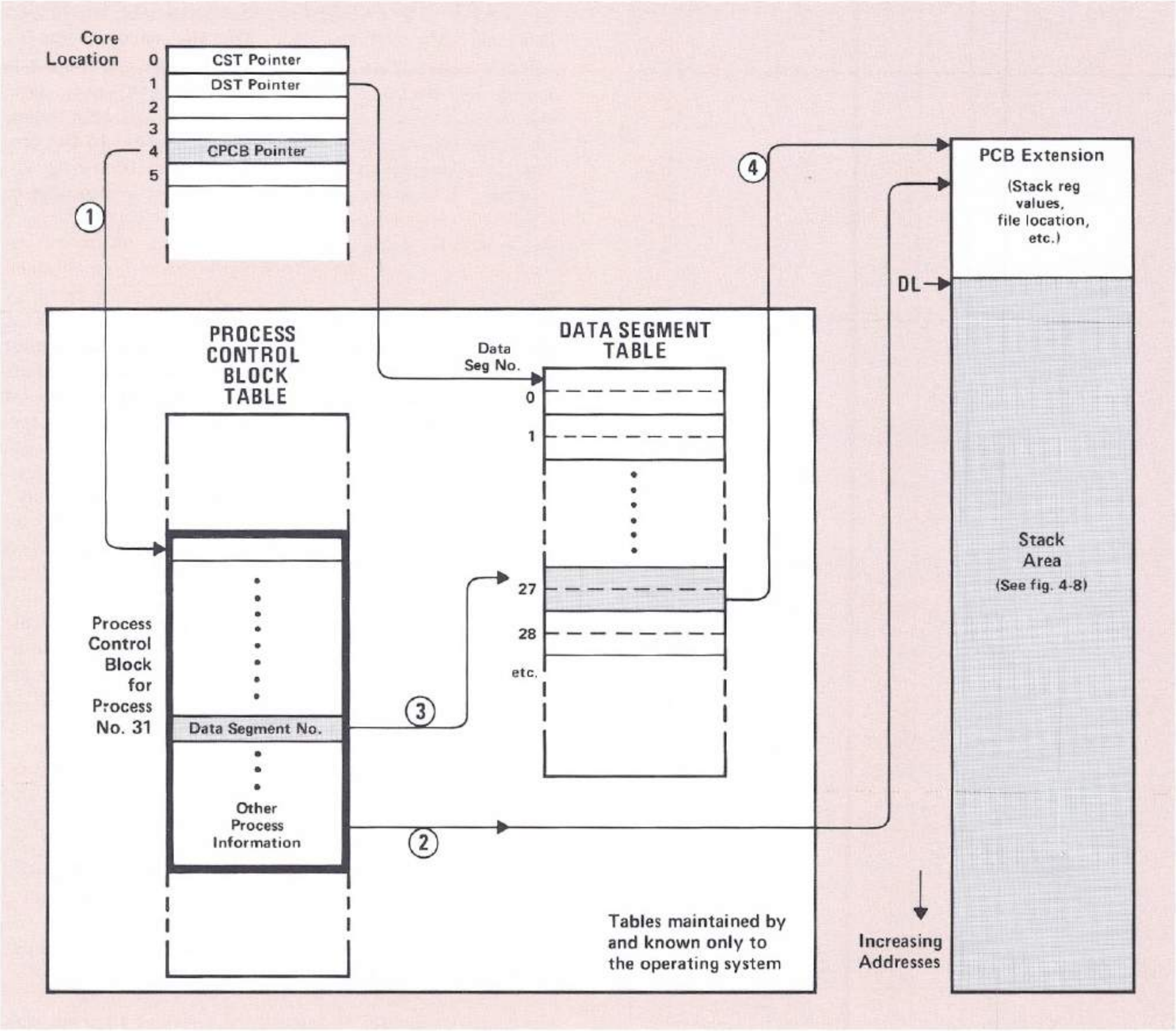

Core Location
0
1
2
3
4
5
CST Pointer
DST Pointer
CPCB Pointer
1
2
3
4
PROCESS CONTROL BLOCK TABLE
Process Control Block for Process No. 31
Data Segment No.
Other Process Information
DATA SEGMENT TABLE
Data Seg No.
0
1
27
28
etc.
Tables maintained by and known only to the operating system
PCB Extension
(Stack reg values, file location, etc.)
DL
Stack Area
(See fig. 4-8)
Increasing Addresses

Figure 4-7. Locating the Stack for One Process

the Data Segment Table. Assuming that the data segment for this process is number 27, entry number 27 in the Data Segment Table will be pointed to. The second word of this entry will give an absolute address pointing (4) to the beginning location of the segment.

The data segment itself includes two separate areas, one of which is the PCB Extension already mentioned. The second area is the stack area, beginning at the hardware-known location DL. The stack is where all dynamic computational operations take place, and it is the next major subject of discussion. The study of the stack, its operation and effects, will occupy the remaining portion of this section.

## STACK OPERATION

The *stack* can be defined as a linear list of data in which the last element added to the list is in the prime position for computational operations (comparable to an accumulator), and is the first element to be removed when the program needs data from the stack. This type of data structure is also more strictly identified as a "LIFO" (last in, first out) stack, since data is removed from the stack in the reverse order from which it was added.

Although many instructions can reference elements within the stack, it is the element currently on the top of the stack

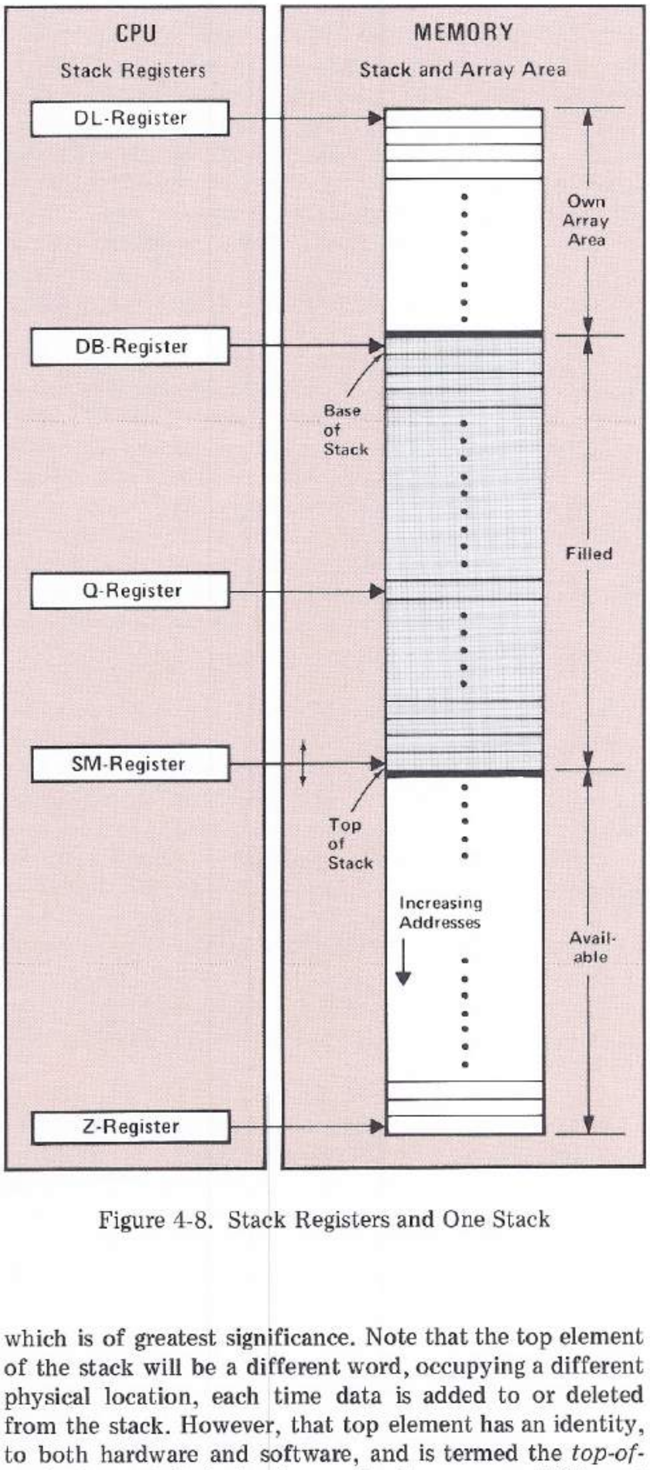


Figure 4-8. Stack Registers and One Stack

which is of greatest significance. Note that the top element of the stack will be a different word, occupying a different physical location, each time data is added to or deleted from the stack. However, that top element has an identity, to both hardware and software, and is termed the *top-of-stack* element. It is also known by its acronym, TOS, and loosely as the top of the stack.

Figure 4-8 shows the basic construction of the stack area and the way stack registers in the CPU delimit the various parts. Remember that there will normally be several stacks in memory, one for each process, but only one will be active at a given time. The stack registers point to the currently active stack.

The stack area is bounded at the low end by the DL-register and at the high end by the Z-register. A major division into two parts is delimited by the DB-register, which points to the base location of the stack. The area between the DB and DL locations is not part of the stack itself, but is closely associated with the stack by providing a dynamic area for such applications as dynamic arrays, symbol tables, etc. Since this area is not particularly relevant to the present discussion, it will be ignored in the following discussions. Its existence, however, should be acknowledged.

Just as the DB-register points to the base location of the stack, so the SM-register points to the current top-of-stack location (in memory). The convention of drawing stack diagrams corresponds to the manner in which code is written (or any written language), beginning at the top of the page and proceeding to the bottom. Thus the stack appears inverted, with the last entry (top-of-stack) toward the bottom of the diagram. Addresses increase in a downward direction.

Whereas the DB-register and Z-register contents are static, the SM-register content is constantly changing as the program progresses, moving up and down the stack area. At all times, the area between DB and SM is filled with valid data, while the area between SM and Z is available for additional data. Should the quantity of data exceed the available space, the attempt to move SM past Z will invoke an interrupt to the operating system, which may grant additional space (new Z value), one or more times—within certain limits.

Unlike the fluid cell-at-a-time movement of the SM pointer, the Q-register value moves sporadically in jumps. It is the purpose of the Q-register to retain the starting point of data relating to the current procedure. Thus when a new procedure begins, the Q pointer jumps ahead to establish a new starting point at the current top of the stack. Conversely, when a procedure ends, the Q pointer jumps back to the place it had marked earlier for the preceding procedure. This action will be illustrated shortly.

As far as the current procedure is concerned, its stack data consists of the locations from a "base" of Q to the current top of the stack.

In the foregoing discussion of basic stack structure, the SM-register was assumed to point at the absolute top of the stack. This is true only for the portion of the stack "in memory". In actual fact, provision is made to allow a few top words of the stack (maximum of four) to "spill over" into hardware registers in the CPU. This is shown in figure 4-9, where the three topmost words are actually in the CPU. The SM-register points to the last stack element in memory, but the actual top-of-stack is in the third CPU register. The actual top of the stack is designated as S.

The four registers in the CPU reserved for receiving top stack elements are scratch pad registers employed only by the CPU hardware. They may not be addressed externally. Externally, the programmer is interested only in the S location contents. The hardware defines the address for him to be at (in this example) the SM-register value plus 3. The value 3 is retained in the SR-register, a three-bit register, which will never indicate a value higher than 4.

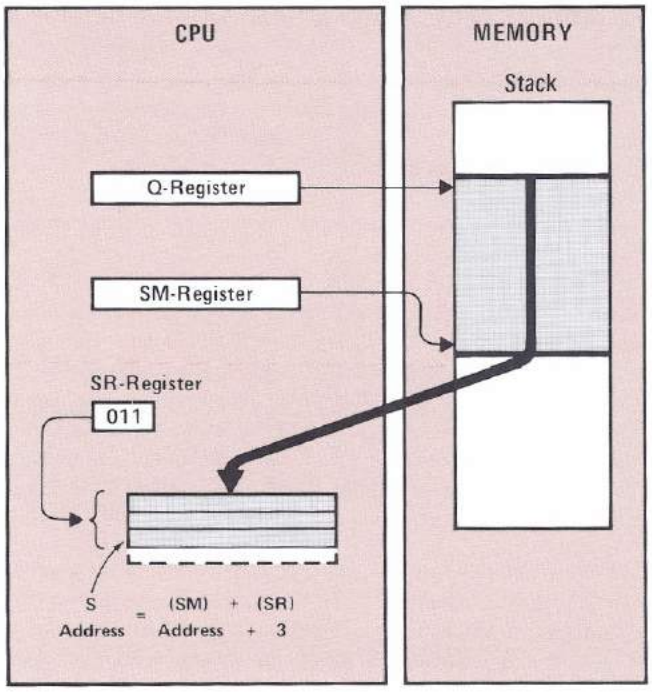


Figure 4-9. Top-of-Stack in the CPU

The address value S obtained by adding the SR-register contents to the SM-register contents is a completely valid address. In fact, when the CPU registers must be cleared for some other operation (e.g., a new procedure or an interrupt), the register contents are physically transferred to the numerically corresponding memory locations. In this example, the SM pointer would move up by three locations, and the SR-register content would become 0.

Again it must be stressed that the user is not usually aware of these registers. The reason for their existence is speed. For example, it is possible to perform computations on the four top elements of the stack without making a single memory fetch. A programmer may wish to optimize his code by watching the availability of operands in the registers as his algorithm progresses.

Since the actual top of the stack (S) is the value of interest, and since S is a valid address, the separate existence of SM and SR values is commonly disregarded, as in the following discussions.

The action of the Q-register in marking the starting location for each procedure's data is shown in figure 4-10.

This figure will be discussed in detail, but briefly, what has occurred in the example shown is the following. The currently executing code segment was working with data in the temporary storage area immediately following the "first Q" location. At that time, the Q-register was pointing at "first Q", S was indicating the top of the stack, and the Z-register was pointing to the end of the data segment. If the executing code segment never called a procedure, the stack picture would never get more complicated. However, at some point the code called a procedure (perhaps a lengthy mathematical routine) by means of a PCAL instruction. This caused additions to the stack as indicated (procedure A). New data was incurred as the procedure began, and S pointed to the top of that data as it was generated. Then procedure A called procedure B (perhaps a frequently used equation), which resulted in new additions to the stack, as shown. Then still later, procedure B called procedure C (perhaps a library routine for a trigonometrical function), resulting in a final picture of the stack as shown.

What will happen next is that procedure C will end, saving its answer in a convenient place for procedure B to access, and issuing an EXIT instruction. Then all the other stack additions due to procedure C will be eliminated (by moving the S and Q pointers back), and procedure B will continue its computations on its own stack data. Likewise, procedure B will come to an end, save its data, and exit, resulting in the elimination of the procedure B stack data. And finally procedure A will do the same, returning the net answer to the new top of the stack, on the main temporary storage area.

It is obvious from this brief outline of events that each time control is returned from the called procedure to the caller's procedure — within the code segment — the stack registers also return to the caller's data area. Thus the stack mark chain virtually eliminates system overhead in keeping track of lexicographical levels (nesting of procedures). For example, the simple return sequence described above, C-to-B-to-A-to main program, is not imperative. Procedure C could have been called again before the return to the main program was complete. Or other procedures (D,E,F, etc.) could enter the picture. But the return for both code and data will always remain perfectly in step — from the called to the caller.

Now the details. Beginning at the top of figure 4-10, note that the area between DB and the first Q is the *global data area*. The locations in this area are reserved by the process for variables (possibly arrays) which it has declared to be global for all procedures called by that process. That is, any procedure using this particular data segment may reference the variables in this area.

The individual locations in the global data area may contain an actual value, or may contain an indirect address pointing to some other location. (That other location either will contain the value or will be the start of an array.) Since DB relative addressing is limited to a maximum of DB+255, only the first 256 locations of this area may be addressed directly. These locations are denoted as the primary global data area. If the number of entries exceeds 256, indirect addressing must be used. Locations in this area (convenient for arrays) are denoted as the secondary global data area.

When the operating system finishes assigning space for the global variables, it points the Q-register at the next succeeding location (first Q). This is the actual start of the stack proper. Initially the S pointer is also pointed at this location, since there is as yet no data on the stack. As the

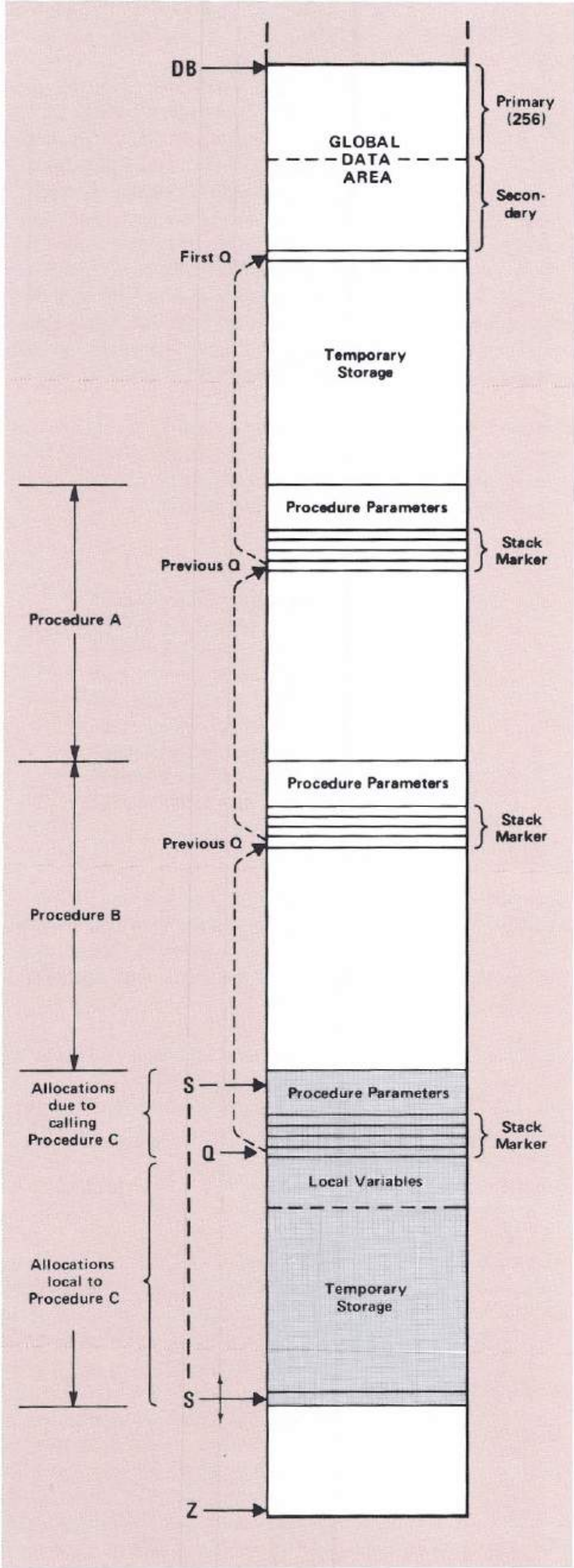


Figure 4-10. Stack Mark Chain

executing code segment proceeds to obtain, manipulate, and generate data for the stack, the S pointer moves away from Q, indicating at all times the top of such data. (Examples of typical operations will be given under the next major heading, "Examples of Stack Operation".)

Then at some time during execution of the code segment, it is assumed that Procedure A is called. Accompanying the call are a set of *procedure parameters* which are placed on the stack just prior to issuance of the PCAL instruction. These are actual parameters, to be substituted for formal parameters in the procedure, and are referenced by Q-addressing.

Calling the procedure causes a four-word *stack marker* to be placed on the stack. The format of this marker is shown in figure 4-11. The first word saves the current contents of the X-register. The second word saves the return address for the code segment — i.e., the P-register address (plus one) relative to the PB-register contents. The third word saves the Status register contents, which includes the code segment number of the caller, in case the called procedure is external to the current code segment. (This was described earlier under "Code Segments".) The fourth word is the one of most interest to the present discussion. This word contains the *delta Q* value, which tells how far back it is to the previous location to which Q was pointing. In this case, delta Q is pointing to "first Q". The Q-register now points at this delta Q location.

The sequence of events described in the preceding two paragraphs is repeated when procedures B and C are called. Each time, the Q-register will point to the delta Q location of the current stack marker, and the contents of that location will point back to the previous setting of Q. Thus it is seen that when procedure C is executing, there will be a chain of delta Q stack marks linking the present Q setting back to the first Q.

Just as the links are established as the procedures are called, so are they used and eliminated as the procedures are exited. When procedure C ends, the EXIT instruction returns S to equal Q, essentially placing the delta Q value temporarily on the top of the stack. This allows the EXIT instruction to compute a new value for the Q-register

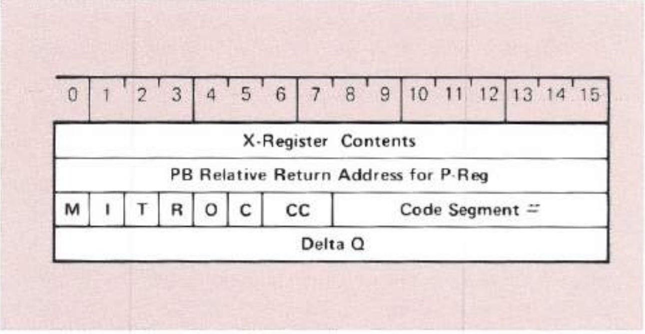


Figure 4-11. Stack Marker Format

("previous Q"), and it appropriately moves Q back. The EXIT instruction causes S to decrement step-by-step through the stack marker, restoring Status, P-, and X-register contents for procedure B.

Lastly, S is moved back to eliminate the unwanted parameters of procedure C. Presumably one or more parameters will be computed answers resulting from procedure C, and so S is only moved back so far as to preserve those desired answers (which are now on the top of the stack). This ability to move S back selectively is one of the functions of the EXIT instruction (refer to instruction definition).

Once again, the sequence of events described in the preceding two paragraphs are repeated, until all procedure data and stack marks are eliminated, and only the final answer is on the top of the stack.

As a final note, observe the breakdown of allocations for one procedure (procedure C illustrated). As shown, the procedure parameters and stack marker are allocations due to calling the procedure. The remaining locations are allocations local to the procedure, which are further broken down into an area for *local variables* and an area for *temporary storage*.

## EXAMPLES OF STACK USAGE

Up to now, the mechanics of the stack have been examined without the application of specific values or problems. To conclude this section, various examples of stack operation will be given. The examples are progressively instructive and, in each case, the advantages of this type of architecture over the register structured computer will be illustrated.

The examples do not necessarily show all the advantages of a stack machine. In fact one of the major advantages has already been shown — that of preserving code and data conditions by marking the stack. This facilitates rapid environment changes (e.g., swapping users), saves overhead for unlimited nesting of procedures, and helps to make code re-entrant. Another major advantage, that it allows fast interrupt handling, will be covered in a later section. The following examples are primarily designed to aid in understanding the stack concept.

### BASIC ARITHMETIC

Figure 4-12 shows a sequence of basic instructions being executed on some data which is presumed to exist in the stack. The upper row shows the most elementary method of adding and removing data to and from the stack, via load and delete instructions. The lower row shows the effects of four arithmetic instructions.

As shown for the initial stack condition (A), the data consists of six numbers in six consecutive locations. The Q-register points to the oldest element of the group, and S points to the element currently on the top of the stack. A Delete instruction (DEL), executed between A and B, causes the number 44 to be removed from the stack; this is accomplished by simply decrementing the S pointer by one. Then, between B and C, a LOAD instruction causes the number 37 to be loaded onto the stack; this is accomplished by storing the number 37 (from another memory location) into the location formerly occupied by the number 44, and then incrementing the S pointer by one.

Between C and D, an ADD instruction is executed. This instruction adds the two top elements of the stack together, deletes both from the stack, places the answer (100) on the top of the stack, and points S at the answer.

Note

> As mentioned previously, up to four of the top stack elements may exist in CPU registers. Obviously, to execute the ADD instruction, at least the two top elements must exist in the CPU. To ensure that this is the case, the hardware checks the content of the SR-register. If the number contained therein is not at least 2, one or more memory fetches are made so that the instruction can be carried out.

Between D and E, a Multiply instruction (MPY) is executed. This instruction multiplies the two top elements of the stack together, deletes both from the stack, places the answer (700) on the top of the stack, and points S at the answer.

To subtract (SUB), the top element is subtracted from the next-to-top element. Thus the answer at F is the result of 500-700, or -200. (As before, only the answer remains after computation is performed.) Finally, at G, negation is performed. This simply reverses the sign of the number on the top of the stack; in binary form a two's complement operation is performed.

Although the sequence A through G in figure 4-12 is a very simple series of operations, it does illustrate the advantages of the stack technique in computation. First, note that regardless of how many elements of data there are or what memory cells they occupy, the operand for each instruction is consistently the same — the top of the stack. This permits *implicit addressing*; i.e., since the operand is understood to be the top of the stack, it is not necessary to give an operand address in the instruction word. Thus (except for LOAD, which must specify a relative address to load from), the instruction can simply say "add", or "multiply", etc. The immediate benefit of this is that it allows code compression. Two instructions can be given in a single word. The sequence D through G, for example, can be given in

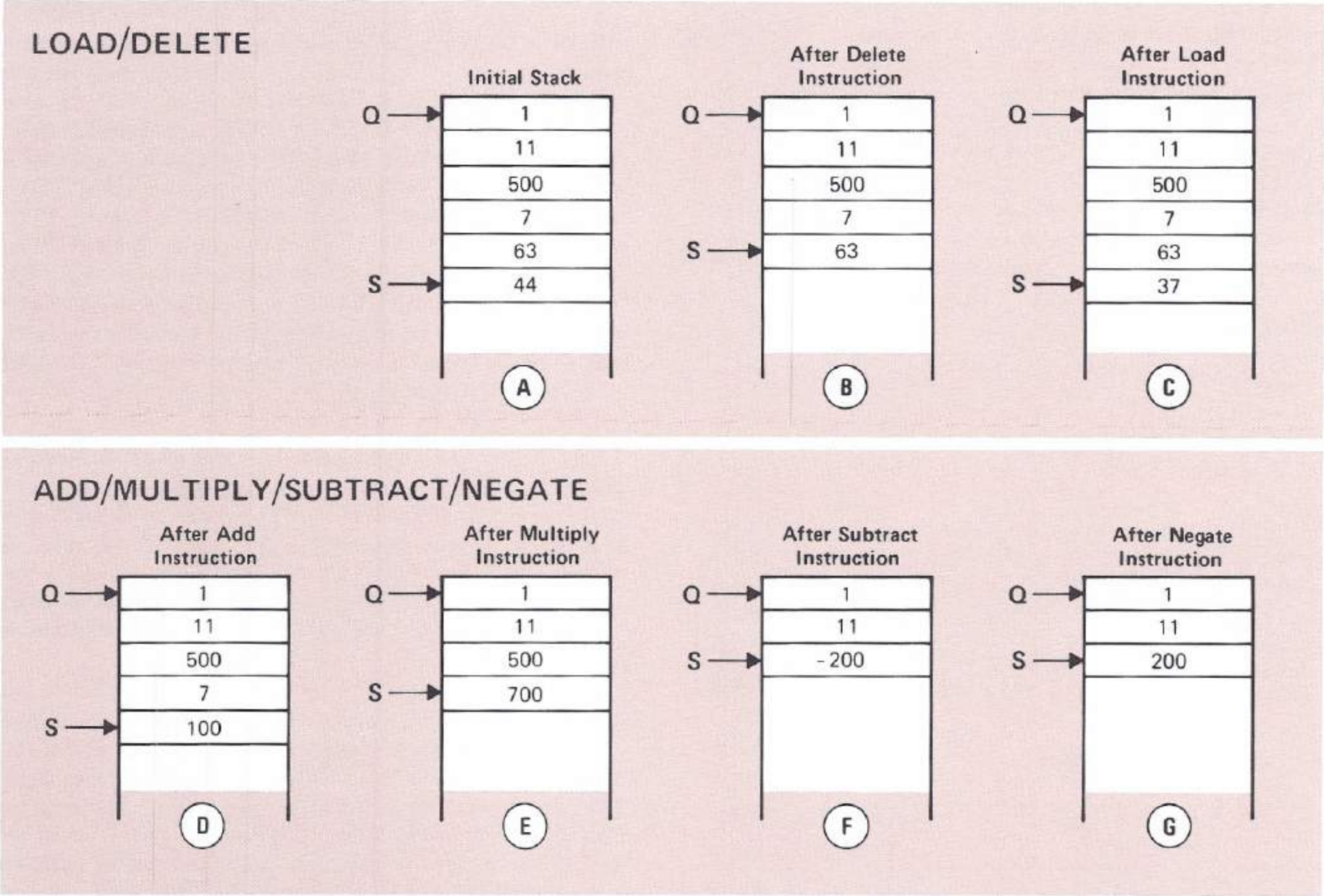


Figure 4-12. Basic Arithmetic Stack Operations

two instruction words. Since this reduces the number of memory fetches, the speed of computation is considerably increased.

A second point to note is that temporary storage of intermediate results is automatically provided. For example, once the parameters 63 and 37 (at C) have been added, they are no longer required and so are thrown away. But the answer, which is substituted on the top of the stack, is automatically in position (adjacent to 7) for the ensuing multiplication. Thus there is no need to provide a dedicated location to save the temporary quantity 100 (or any of the other intermediate results).

It is apparent that the order of placing elements on the stack is very important. However, it is one of the compiler's functions to provide the correct order, and (except in assembly mode) this is of little concern to the programmer.

## PROCEDURE CALLS

Figures 4-13 and 4-14 illustrate the operations involved in a procedure call. Figure 4-13 shows programmatically how a procedure is set up and called, and figure 4-14 shows what happens to the stack when the procedure is called and executed.

The purpose of this example is to demonstrate the ease and simplicity of *parameter passing* — i.e., the means by which a program can substitute *actual parameters* for the *formal parameters* declared in a procedure. In this example (see bottom block in figure 4-13), the formal parameters are J and K, and the actual parameters to be passed to the procedure are 25 and 10, respectively.

As shown in the bottom block of figure 4-13, the calling of a procedure has an equivalency in mathematical terms. That is, a procedure is like a predetermined equation, in this case "ANSWER = J/K". Calling the procedure is like a request to solve the equation for the specific values of 25 for J and 10 for K. Executing the procedure is to perform the computation, in this case getting an answer of 2. (To keep things simple, the example procedure will be made to work strictly with integer numbers; thus the fractional remainder 5/10 will automatically be discarded.)

The upper two boxes in figure 4-13 list two forms of the program that will accomplish the example procedure. The

## SOURCE LANGUAGE

```
              1      BEGIN INTEGER ANSWER;
            ┌ 2  INTEGER PROCEDURE QUOTIENT (J,K);
            │ 3  VALUE J,K;
Pro-        │ 4  INTEGER J,K;
cedure      │ 5           BEGIN
            │ 6               QUOTIENT ←— J/K;
            └ 7           END;
Call        { 8  ANSWER ←— QUOTIENT (25,10);
              9      END;
```

## MACHINE LANGUAGE

```
                   Assembly               Octal
          ┌ 10  LOAD Q-5                  041605
          │ 11  LOAD Q-4                  041604
Pro-      │ 12  DIV, DEL                  002340
cedure    │ 13  STOR Q-6                  051606
          └ 14  EXIT, 2                   031402
          ┌ 15  ZERO, NOP                 000600
          │ 16  LDI, 31                   021031
Call      │ 17  LDI, 12                   021012
          │ 18  PCAL, 20                  031020
          └ 19  STOR DB+0                 051000
            20  PCAL (to system)          031xxx
```

## MATHEMATICAL LANGUAGE

```
Procedure:      ANSWER = J/K

Call:           Solve ANSWER for
                  J = 25 and K = 10

Execution:      ANSWER = 25/10
                       = 2, remainder 5


                Note: Decimal 25 = Octal 31
                      Decimal 10 = Octal 12
```

Figure 4-13. Declaring and Calling a Procedure

top box shows how the program would be written in the source programming language. The middle box shows the machine language code that would be emitted by the compiler. The machine language code is shown both in assembly (or mnemonic) form, and in an octal form of the actual binary machine code.

Both the source and machine language versions of the program will now be considered on a line by line basis. First, the source language program.

Line 1 begins the program block, just as line 9 ends it. Although the entire program consists only of one procedure and a call to that procedure, it nevertheless remains necessary to enclose the program between a *BEGIN statement* and an *END statement*. These statements define a program. ANSWER is declared to be a *global variable* for this program by giving its name within the BEGIN statement. This will cause the variable ANSWER to reside in the global data area, and thus allow its access by another procedure — such as an output routine to print out the result. The type declaration INTEGER specifies that ANSWER will always be an integer, and tells the compiler to reserve one word for the result (rather than two or three). ANSWER is allocated the word at DB+0.

Lines 2 through 7 comprise the *procedure declaration*, which includes the *procedure head* (lines 2, 3, 4) and the *procedure body* (lines 5, 6, 7). The procedure declaration in a program cannot cause execution by itself; it must be called before any execution can take place. Thus the procedure declaration is always separate and distinct from the procedure call. They need not be immediately adjacent, as in this example.

Line 2 gives the *procedure name*, QUOTIENT, and declares that the procedure is of type INTEGER, which means that the result will be in integer form. It also gives the names of the formal parameters, J and K. Line 3 is the *value part* of the procedure declaration. Declaring J and K as values means that a value (rather than a pointer) will be passed as a procedure parameter, in both cases. This permits working with a copy and eliminates any need to change the actual parameter. Line 4 declares that actual parameters for J and K must be integers; if any other type is given (floating point, for example), a compilation error will result.

Line 5 begins the procedure body. Actually, since this procedure consists of only one statement, the BEGIN statement and END statement (line 7) are superfluous. They are included here, however, to illustrate the common form for a procedure (normally involving a compound statement). Line 6 is the *procedure statement*, the executable part of the procedure body. It is this statement which will cause the division of J by K, and will temporarily store the quotient as a procedure result, identified by the procedure name QUOTIENT.

The call to the procedure is given at line 8. This is an executable statement, as opposed to a procedure declaration. When this statement is encountered in a program, it will cause the procedure named QUOTIENT to be executed, passing actual parameters of 25 and 10 to the procedure, and will cause the global variable ANSWER to assume the value of the result. At this point (line 9) the program is complete.

Lines 10 through 19 show the machine language code which the compiler emits for the two executable statements in the program. That is, line 6 causes lines 10 through 14 to be generated, and line 8 causes lines 15 through 19 to be generated.

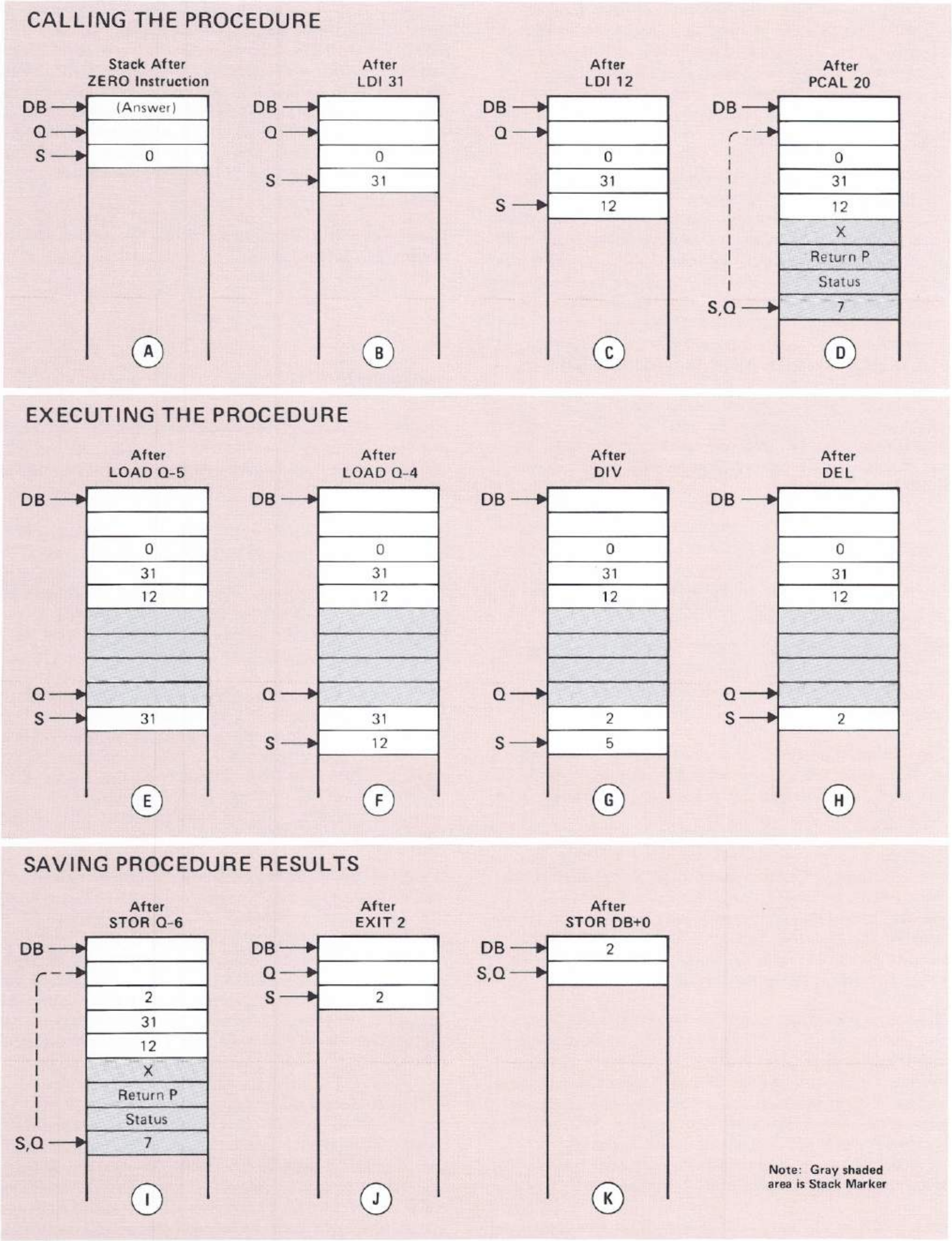


Figure 4-14. Executing a Simple Procedure

In order to explain the operation of the program in machine language, it is necessary to examine what is happening on the stack. Figure 4-14 will therefore be referred to in the following discussions. Furthermore, to aid in visualizing the operations, they will be described in chronological order; i.e., the machine language program will begin to execute at line 15.

First of all, it is assumed that the user has logged onto the system, has compiled the program, and is ready to run (or is running a program that will shortly encounter the statement in line 8). Loading the program has caused space to be allocated for the one global variable, ANSWER, which is at DB+0 (see A in figure 4-14). Since there are no other global variables, Q and S initially point at the immediately following location. (The content of that location will never be significant; in essence it is a dummy delta Q location.) It may be instructive to refer back to figures 4-10 and 4-8.

Additionally, during program loading, the operating system has evaluated the program in order to set the Z-register appropriately for an initial estimated stack size. Also, since no dynamic own arrays are declared, DL is set coincident with DB.

Now it is assumed that the user issues a system command to run the program or, in other words, to execute the procedure call given in line 8 of figure 4-13. This causes control to be passed to line 15 in the machine language program, where the sequence to call the procedure begins.

The first instruction is a ZERO, NOP. Executing this instruction puts a 0 on the stack and increments the S pointer (see A in figure 4-14). This reserves a location for the procedure result.

Next (B and C; lines 16 and 17), the parameter values 31 and 12 are passed directly from the instruction words to the stack (area reserved for procedure parameters). Octal notation is used for these values.

Then (D, and line 18) a procedure call instruction, PCAL, causes a four-word stack marker to be placed on the stack. The S and Q pointers point to the delta Q location of the marker, which now indicates 7 (the number of locations back to the initial Q location). It is assumed that entry number 20 in the Segment Transfer Table will direct the call to the correct procedure starting point.

Now execution of the procedure begins (line 10). The first two instructions (lines 10 and 11) load copies of the procedure parameters onto the top of the stack (E and F), using Q- relative addressing. The next instruction (line 12) divides the top-of-stack parameter into the next-to-top parameter, and substitutes the quotient (2) and the remainder (5) on the top of the stack, as shown at G. The second half of the same instruction (DEL) discards the remainder word by decrementing S, as shown at H.

To save the result, the STOR Q-6 (line 13) first copies the top-of-stack into the location reserved for the procedure result, formerly occupied by a 0, as shown at I. Then it is possible to exit from the procedure. The EXIT instruction (line 14) restores Q to its initial setting, and the "2" included with the instruction causes S to move back two locations past the stack marker. As shown at J, this leaves the result, 2, in the location reserved for QUOTIENT — now on the top of the stack. The EXIT instruction also returns program control to line 19, which causes the content for QUOTIENT to be stored in the location for ANSWER in the global data area. This produces the final result shown at K.

Finally (line 20), a procedure call to the system returns control back to the system.

## RECURSION

The last example in this series demonstrates the stack principles involved in a *recursive procedure*. A recursive procedure is one which calls itself one or more times during execution.

Recursion is a powerful programming technique which derives from the re-entrant capability of the code. The advantages and other considerations of this technique are beyond the scope of this manual, and the example to be given does not necessarily illustrate the niceties of the technique. Rather the example is intended to show only how recursion is accomplished on the stack.

The example chosen is purposely kept simple in order to provide continuity with the preceding example. (Note that the form of the source language program for this example, in table 4-2, is nearly identical to that of the preceding example in figure 4-13.) The procedure simply computes N! (N factorial), where N is the formal parameter. The procedure will be called with an actual parameter of 4, so that computation of 4! will be: 1 × 2 × 3 × 4 = 24.

In essence, this problem consists of repetitively multiplying the previous product by a parameter which is incremented by one on each repetition. To provide a starting point (initial "previous product"), the value 1 is automatically given. The procedure is designed to perform this multiplication sequence by repetitively calling itself, after it has been called once by the main program. Thus for any N, the procedure will be called N+1 times. In this example there will be one call by the main program and four recursive calls.

Table 4-2 lists the source and machine language forms of a program block to solve this problem. Since the source language program is so similar to the preceding example, it need not be discussed at this point. The machine language form has been slightly changed to more closely resemble an actual program listing. Some assumed PB relative addresses are given for each instruction, beginning at address 00114. The assumption here is that this program block is embedded in a larger "main" program. (Note that the

Table 4-2. Recursive Program

**SOURCE LANGUAGE**

```
                 :
        BEGIN INTEGER Y;
          INTEGER PROCEDURE FACTORIAL (N);
          VALUE N;
          INTEGER N;
              FACTORIAL := IF N = 0 THEN 1 ELSE N * FACTORIAL (N-1);
          Y := FACTORIAL (4)
        END;
                 :
```

**MACHINE LANGUAGE**

| PB Relative Addresses | Instructions | Octal Code | Comments |
|---|---|---|---|
| 00114 | LOAD Q- 004 | 041604 | Load parameter |
| 00115 | CMPI, 000 | 022000 | Test it for zero |
| 00116 | BNE P+ 003 | 141503 | If not zero, branch to 00121 |
| 00117 | LDI, 001 | 021001 | If zero, load 1 as initial multiplicand |
| 00120 | BR 006 | 140006 | Branch to 00126 (to Exit loops) |
| 00121 | ZERO, NOP | 000600 | Save space for intermediate product |
| 00122 | LOAD Q- 004 | 041604 | Load parameter |
| 00123 | SUBI, 001 | 023001 | Decrement for use as new parameter |
| 00124 | PCAL, 026 | 031026 | Recursive call |
| 00125 | MPYM Q- 004 | 111604 | Multiply parameter by TOS |
| 00126 | STOR Q- 005 | 051605 | Store this recursion's product |
| 00127 | EXIT, 001 | 031401 | Save the product and exit |
| 00130 | ZERO, NOP | 000600 | Save space for final product |
| 00131 | LDI, 004 | 021004 | Load initial actual parameter |
| 00132 | PCAL, 026 | 031026 | Main program's call to the procedure |
| 00133 | STOR DB 015 | 051015 | Save final product in global area |
| 00134 | PCAL, XXX | 031xxx | Return to system |

assigned STT entry for this procedure is assumed to be 026, and the global assignment for Y is DB+15.) The starting point for execution is at address 00130.

Figure 4-15 illustrates the program in flowchart form. Box 1 in the diagram calls the procedure (boxes 2 through 9), box 10 saves the result, and then control reverts to the main program at box 11. The procedure consists of two phases. The call phase begins when the procedure is called by the program, and is repeated four times. Briefly, what happens in this phase is that a succession of N values are placed on the stack, along with a space for intermediate answers. The N values are decremented to zero and then the exit phase begins. This phase successively multiplies an accumulating product by each of the N values loaded on the stack in the call phase — in the reverse order. On each loop unneeded stack information is deleted, saving only the answer for that loop, until only the final answer is left. At that time (box 9) the final EXIT instruction finds that its return address points back to the calling block, and so the final answer is stored in the global area and control reverts to the main program.

As will be shown in the following detailed discussion, the return address check at box 9 is not literally a test for a specific address. Rather it specifies a return to the address given in each stack marker. Obviously the last return (first one placed on the stack) will be a return to the outer block.

Figures 4-16 and 4-17 show the overall process of building up the stack by recursive calls, and then paring it down with recursive exits. These two figures will be used in the following discussions. Also the machine language program in table 4-2 will be referred to; individual lines will be identified by PB relative address, omitting the leading zeros.

MAIN PROGRAM CALL. As before, the main program has already reserved global space for the final answer (Y) before

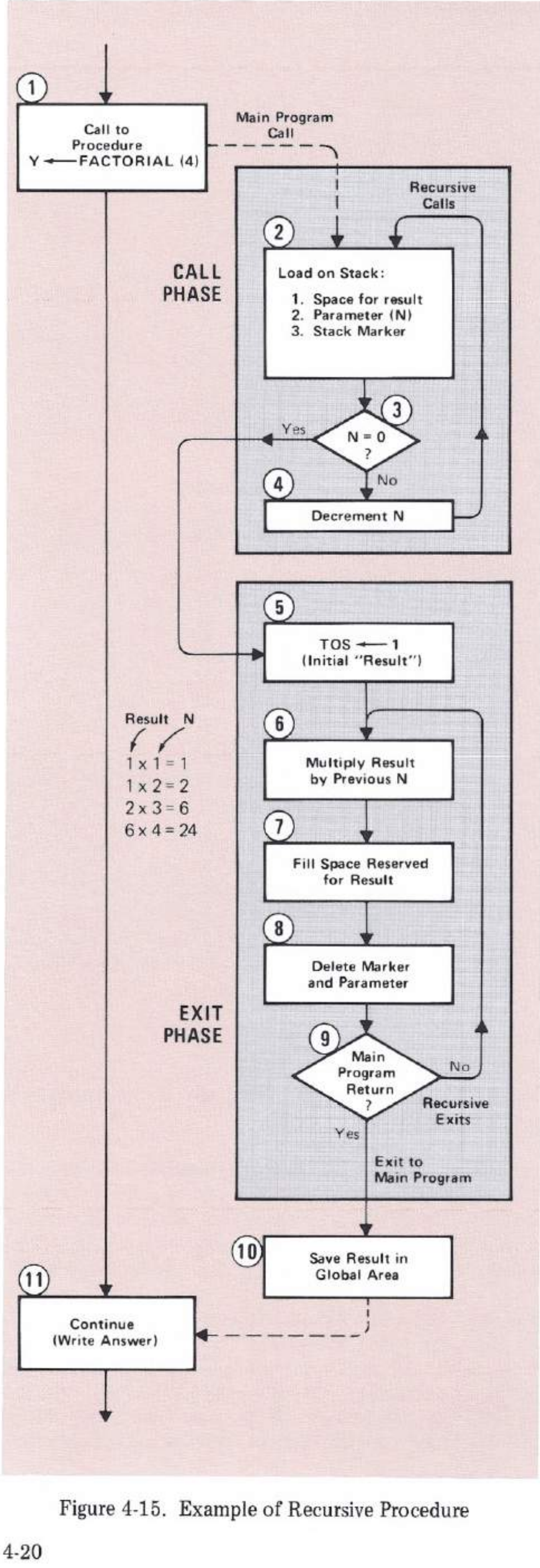


Figure 4-15. Example of Recursive Procedure

the procedure is called. When the call is given, the ZERO, NOP instruction at address 130 reserves space for the procedure result, FACTORIAL. (Compare stack pictures A and Z.) This is the first stack addition due to calling the procedure.

Next, the actual parameter 4 is loaded on (B), and then the PCAL instruction is issued. This causes the first stack marker to be loaded (C). This marker differs from the ones which will follow in that it contains return information to the outer block which called the present procedure. That is, the "return P" word is a P relative address for return to the caller in the code segment, and delta Q points back to the Q value that the caller was using earlier in the stack. Now, S and Q are both pointing at the last word of the first marker for this procedure.

TEST FOR ZERO. At addresses 114 and 115 (stack pictures D and E), the procedure parameter is first tested for zero. This is done by copying it onto the top of the stack (LOAD Q-4) and giving a CMPI 0 instruction. This instruction sets the condition code according to comparison results and deletes the tested word (E). Since the first test is non-zero (i.e., 4), the branch instruction at line 116 transfers control to address 121 (i.e., P+4). This test and branch will be repeated in each of the following recursion loops until the parameter has become zero.

FIRST RECURSIVE CALL. The branch to address 121 causes the procedure to call itself. As usual, the first action of the call is to load the procedure parameters onto the stack. The parameters in this case are the variable FACTORIAL and a decremented form of the original passed parameter. Thus the ZERO, NOP instruction reserves a location for FACTORIAL (see F), strictly for use by this recursion (i.e., distinct from the final FACTORIAL location reserved at A); then (G,H) the new parameter is obtained by copying the preceding value to the top of the stack (LOAD Q-4) and decrementing with a SUBI 1 instruction.

After loading parameters for the new call, another PCAL instruction is issued. This causes a new stack marker (see I) and, via the Segment Transfer Table, transfers control back to the starting point of the procedure, address 114. The new stack marker gives as its return P value the address immediately following the PCAL, which is 125. (This will be important to remember when the exit sequence is discussed.) Also, the delta Q value is 6, since the previous delta Q was six locations back.

SUCCESSIVE RECURSIONS. Now all of the steps described in the preceding three paragraphs are repeated, beginning with the parameter test for zero. Since the parameter is 3 on the second recursion, the branch to address 121 again occurs. The first actions, again, are to reserve a location for this recursion's answer (J) and to load a decremented parameter value of 2 (K and L). After this, the procedure call back to the beginning is again made, resulting in another stack marker (M) which is identical to the one generated on the first recursion.

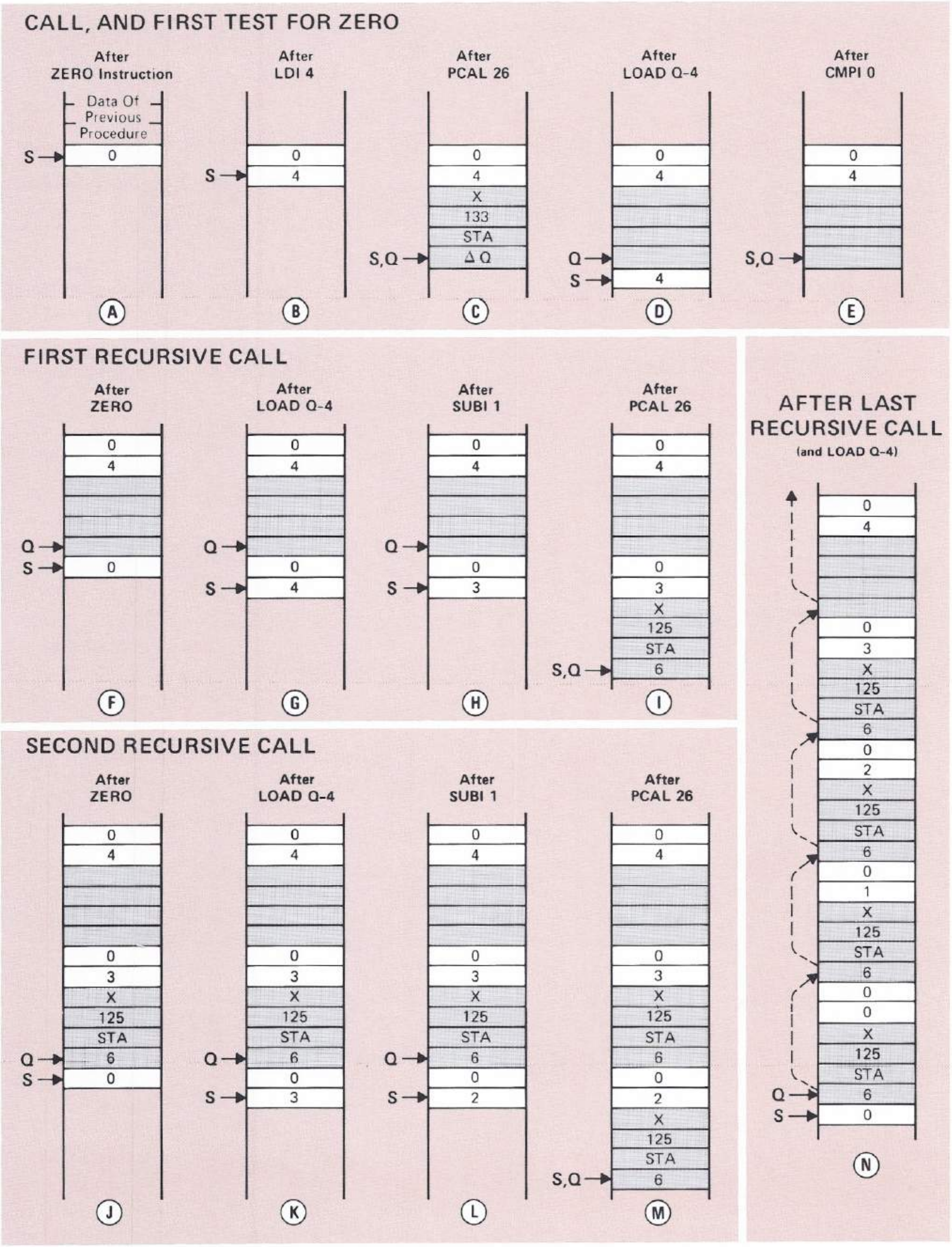


Figure 4-16. Recursive Calls

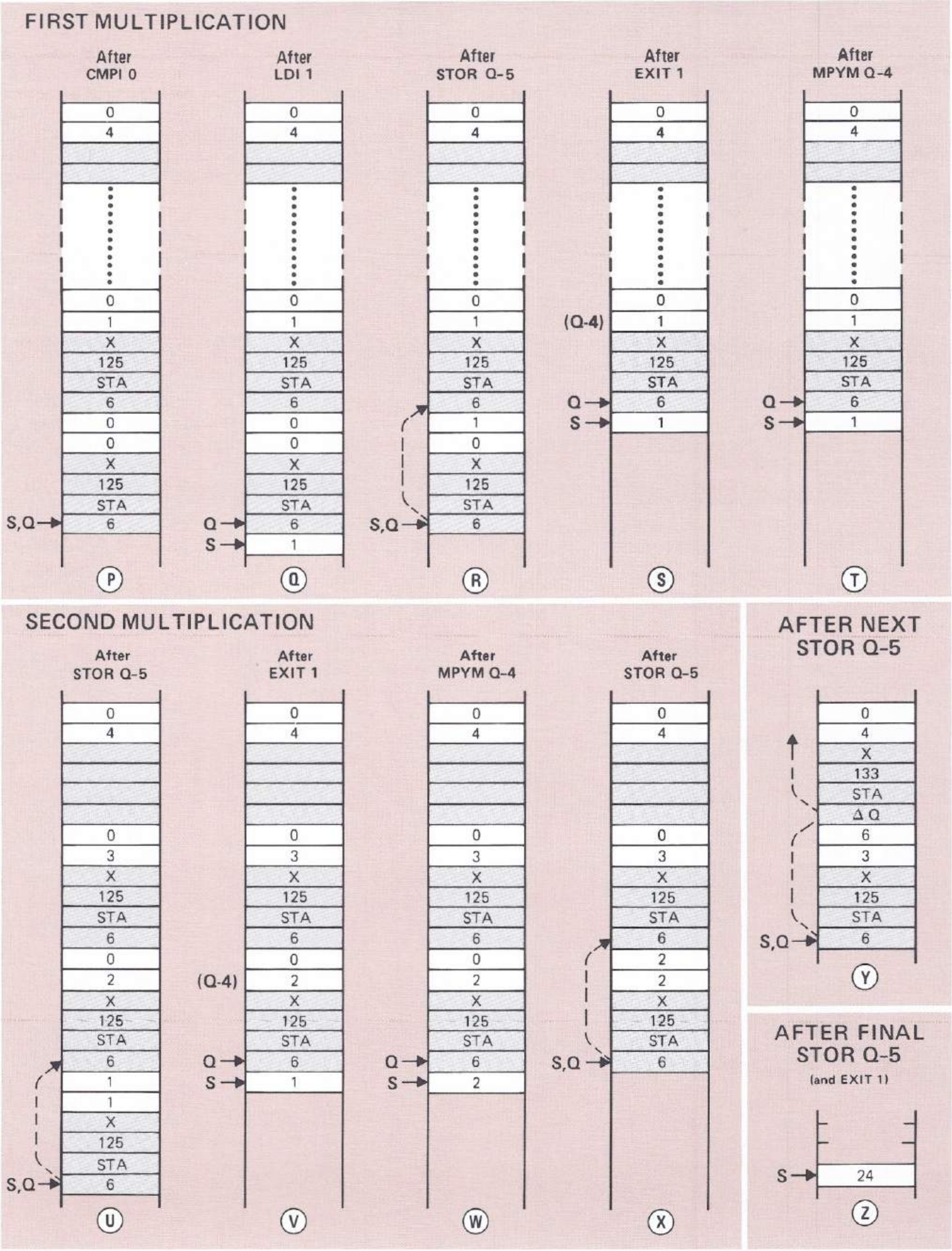


Figure 4-17. Recursive Exits

The third and fourth recursions repeat the entire process again, loading parameters of 1 and 0 followed each time by a stack marker. Thus when the final LOAD Q-4 occurs in preparation for the zero test, the stack appears as shown at N.

FIRST EXIT. The check at address 115 now finds that the parameter is zero. The checked copy of the parameter is deleted from the stack (P in figure 4-17) and the branch at address 116 transfers control to address 117 (rather than 121).

As mentioned earlier (fourth paragraph under the Recursion heading), an assumed value of 1 is necessary as an initial "previous product" in order to begin the multiplication loops. This is accomplished by a LDI 1 instruction (address 117), which puts a 1 on the top of the stack (see Q).

Then an unconditional branch at address 120 transfers control to address 126, where the "1" on the top of the stack is stored into the location reserved for this recursion's answer, as shown at R. The next instruction is then the EXIT 1 instruction at address 127. This causes Q to move back six locations (delta Q = 6) and S five locations (EXIT 1 deletes one of the two parameters), as shown at S. The return address for the P-register, as will be remembered from five paragraphs back, is the MPYM Q-4 instruction at address 125. This causes the parameter at Q-4 (1) to be multiplied by the 1 on the top of the stack, leaving the answer as the new top-of-stack element. Since 1 X 1 = 1 there is no apparent change from S to T, but in fact a multiplication has occurred.

FIRST RECURSIVE EXIT. The answer of the first multiplication is now stored in the location reserved for it (Q-5) as shown at U, by the STOR Q-5 instruction at address 126. The next instruction, at 127, is again the EXIT 1 instruction, which peels back the stack as shown at V and returns the P-register to the MPYM Q-4 instruction at address 125. The parameter for multiplication (at Q-4) is now 2, so the multiplication result at W is 2. Again, this is stored back in the location reserved for it (Q-5) as shown at X.

SUCCESSIVE EXITS. After saving the result, the next EXIT 1 is again encountered, causing the S and Q stack pointers to move back to the next marker, leaving the answer 2 on the top of the stack. The return for the P-register is again 125, so the MPYM Q-4 instruction multiplies 2 X 3, and the following STOR Q-5 puts the answer 6 into the reserved location as shown at Y.

Likewise, the last recursive exit causes the value 6 to be left on the top of the stack when the last return to address 125 is made. Then the final multiplication multiplies 6 X 4, and the last STOR Q-5 instruction puts the answer 24 into the location originally reserved for the end result FACTORIAL.

The last EXIT instruction finds the return for the Q-register (delta Q) pointing back to the origin of an earlier procedure, and so is no longer shown in the stack diagram at Z. However, since one parameter is saved, the final answer remains on the top of the stack, as shown. The P-register, meanwhile, returns to the next instruction in the outer block, which is the STOR DB 15 instruction at address 133. This saves the answer in the global area, and a final PCAL returns control to the system.

SECTION V

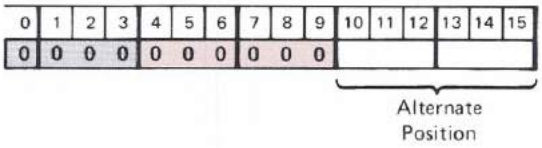
# INSTRUCTION SET

This section defines each of the 170 machine instructions in the HP 3000 instruction set. Where additional information would be helpful in understanding the operation of a particular instruction, an **Instruction Commentary** reference is given following the definition. In such cases, refer to the corresponding number under the heading, "Instruction Commentary", at the end of this section.

Unless specifically mentioned, the indicators (Condition Code, Overflow, and Carry) are unaffected by instruction execution.

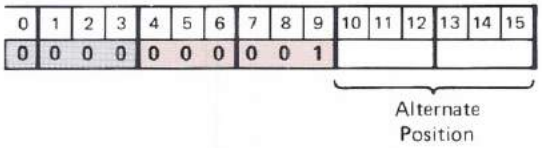
## STACK OP INSTRUCTIONS

### NO OP INSTRUCTION

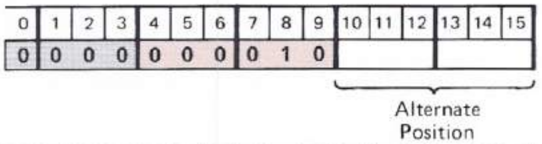


NOP No operation. The user's program space and data space remain unchanged.
Stack opcode: 00
Indicators: unaffected

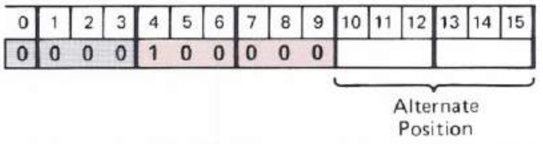
### DUPLICATE AND DELETE INSTRUCTIONS



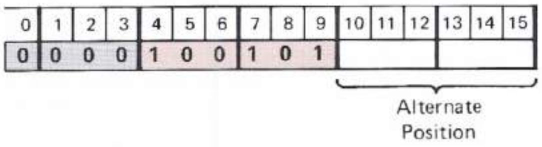
DELB Delete B. The second word of the stack is deleted and the stack is compressed. The content of the TOS is unchanged.
Stack opcode: 01
Indicators: unaffected



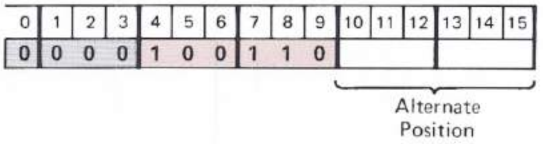
DDEL Double delete. The top two words of the stack are deleted.
Stack opcode: 02
Indicators: unaffected



DEL Delete A. The top word of the stack is deleted.
Stack opcode: 40
Indicators: unaffected



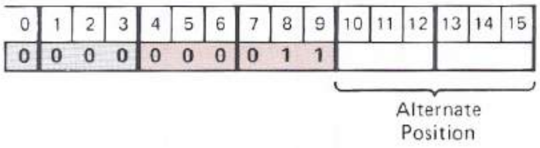
DUP Duplicate A. The top word of the stack is duplicated by pushing a copy of the TOS onto the stack.
Stack opcode: 45
Indicators: CCA



DDUP Double duplicate. The double word in the top two words of the stack is duplicated by pushing a copy of it onto the stack.
Stack opcode: 46
Indicators: CCA on new TOS double word

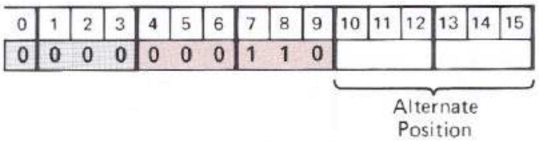
## ZERO INSTRUCTIONS

ZROX Zero X. The content of the Index register is replaced by zero.
Stack opcode: 03
Indicators: unaffected



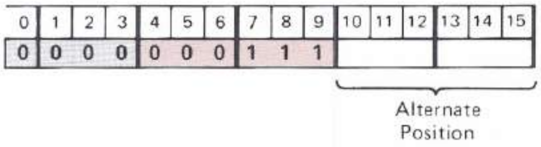
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | | | | | | |

ZERO Push zero. A zero word is pushed onto the stack.
Stack opcode: 06
Indicators: unaffected



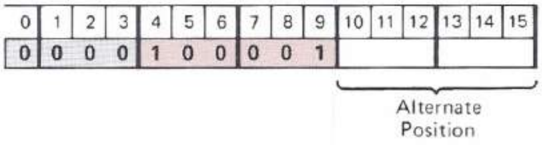
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | | | | | |

DZRO Push double zero. Two words containing all zeros are pushed onto the stack.
Stack opcode: 07
Indicators: unaffected



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | | | | | | |

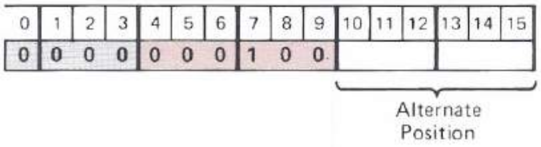
ZROB Zero B. The second word of the stack is replaced by zero. The TOS is unaffected.
Stack opcode: 41
Indicators: unaffected



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | | | | | | |

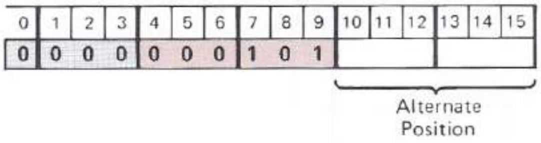
## INCREMENT/DECREMENT INSTRUCTIONS

INCX Increment X. The content of the Index register is incremented by one in integer form.
Stack opcode: 04
Indicators: CCA, Carry, Overflow



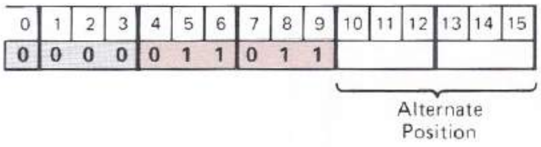
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | | | | | |

DECX Decrement X. The content of the Index register is decremented by one in integer form.
Stack opcode: 05
Indicators: CCA, Carry, Overflow



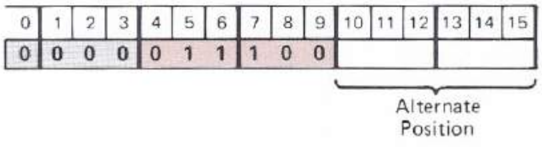
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | | | | | | |

INCA Increment A. The TOS is incremented by one in integer form.
Stack opcode: 33
Indicators: CCA, Carry, Overflow



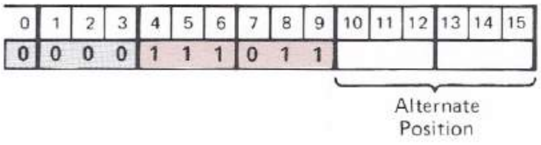
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | | | | | | |

DECA Decrement A. The TOS is decremented by one in integer form.
Stack opcode: 34
Indicators: CCA, Carry, Overflow



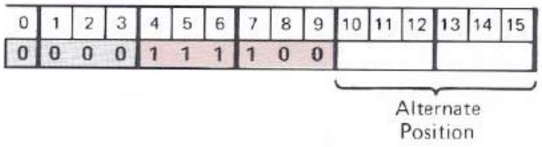
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | | | | | | |

INCB Increment B. The second word of the stack is incremented by one in integer form. The TOS is unaffected.
Stack opcode: 73
Indicators: CCA, Carry, Overflow



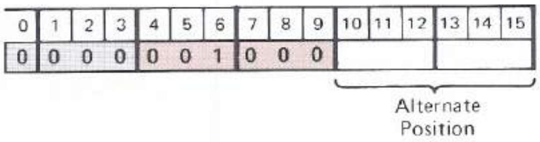
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | | | | | | |

DECB Decrement B. The second word of the stack is decremented by one in integer form. The TOS is unaffected.
Stack opcode: 74
Indicators: CCA, Carry, Overflow

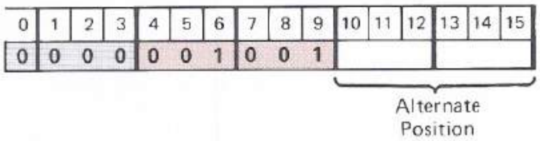


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | | | | | | |

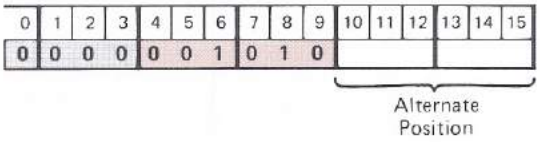
## DOUBLE INTEGER INSTRUCTIONS



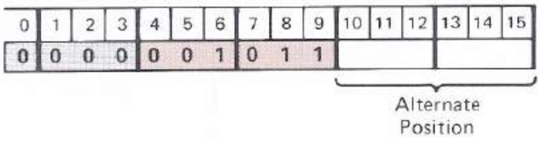
DCMP Double compare. The Condition Code is set to pattern C as a result of the doubleword integer comparison of D,C and B,A. The two double words are deleted from the stack.
Stack opcode: 10
Indicators: CCC



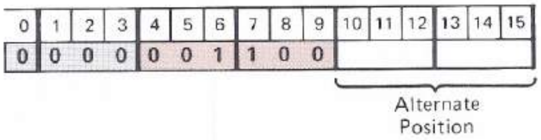
DADD Double add. The two doubleword integers contained in the top four elements of the stack are added in double length integer form (D,C + B,A) and they are deleted. The doubleword integer sum is pushed onto the stack (B,A).
Stack opcode: 11
Indicators: CCA, Carry, Overflow



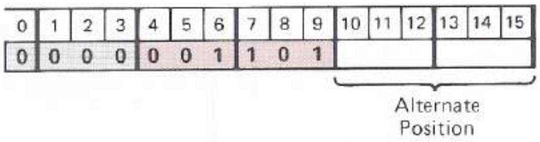
DSUB Double subtract. The doubleword integer contained in the top two words of the stack is subtracted from the doubleword integer contained in the third and fourth words of the stack (D,C - B,A). The top four words of the stack are deleted and the doubleword integer result is pushed onto the stack (B,A).
Stack opcode: 12
Indicators: CCA, Carry, Overflow



MPYL Multiply long. The top two words of the stack are multiplied in integer form. The words are replaced by the double length product, with the least significant half on the TOS. Overflow is cleared. Carry is cleared if the low order 16 bits represent the true result (i.e., if the high order 17 bits are either all zeros or all ones); otherwise, Carry is set.
**Instruction Commentary 1.**
Stack opcode: 13
Indicators: CCA, Carry, Overflow

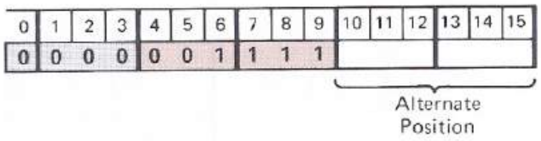


DIVL Divide long. The doubleword integer in the second and third elements of the stack is divided by the integer in the TOS (C,B ÷ A). The three words are deleted, and the quotient and remainder are pushed onto the stack (quotient in B, remainder in A).
Stack opcode: 14
Indicators: CCA, Overflow



DNEG Double negate. The doubleword integer contained in the top two words of the stack is negated (two's complemented) and replaces the original doubleword integer.
Stack opcode: 15
Indicators: CCA, Overflow

## INTEGER INSTRUCTIONS



CMP Compare. The Condition Code is set to pattern C as a result of the integer comparison of the second word of the stack with the TOS. Both words are deleted.
Stack opcode: 17
Indicators: CCC

ADD Add. The top two words of the stack are added in integer form and are then deleted. The resulting sum is pushed onto the stack.
Stack opcode: 20
Indicators: CCA, Carry, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | | | | | |

Alternate Position

SUB Subtract. The TOS is subtracted in integer form from the second word of the stack and both words are then deleted. The resulting difference is then pushed onto the stack.
Stack opcode: 21
Indicators: CCA, Carry, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | | | | | | |

Alternate Position

MPY Multiply. The top two words of the stack are multiplied in integer form. The two words are deleted and the least significant word of the double length product is pushed onto the stack. If the high order 17 bits of the double length product (including the sign bit of the second word) are not all zeros or all ones, Overflow is set.
**Instruction Commentary 1.**
Stack opcode: 22
Indicators: CCA, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | | | | | | |

Alternate Position

DIV Divide. The integer in the second word of the stack is divided by the integer on the TOS. The second word is replaced by the quotient, and the top word is replaced by the remainder.
Stack opcode: 23
Indicators: CCA on quotient, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | | | | | | |

Alternate Position

NEG Negate. The integer in the TOS is replaced by its two's complement.
Stack opcode: 24
Indicators: CCA, Overflow, Carry

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | | | | | |

Alternate Position

## TEST INSTRUCTIONS

TEST Test TOS. The condition code is set to pattern A according to the content of the TOS word.
Stack opcode: 25
Indicators: CCA

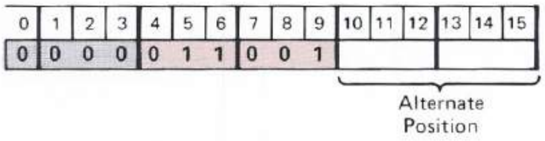
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | | | | | | |

Alternate Position

DTST Test double word on TOS. The condition code is set to pattern A according to the contents of the top two words of the stack. Also, Carry is cleared if the low order 16 bits of the doubleword result (TOS) represent the true integer value (i.e., if the high order 17 bits are either all zeros or all ones); otherwise, Carry is set.
**Instruction Commentary 1.**
Stack opcode: 27
Indicators: CCA, Carry

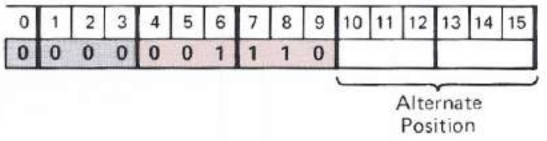
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | | | | | | |

Alternate Position

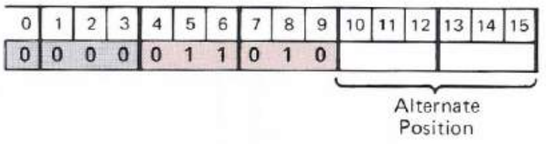


BTST Test byte on TOS. The Condition Code is set to pattern B according to the contents of the byte contained in the eight least significant bits of the TOS word (bits 8-15).
Stack opcode: 31
Indicators: CCB

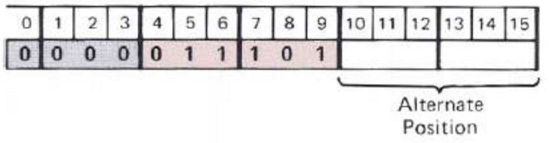
## EXCHANGE INSTRUCTIONS



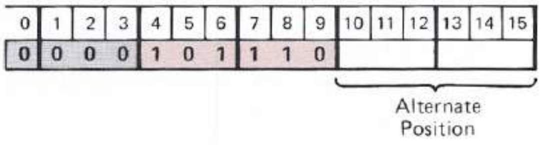
DXCH Double exchange. The top two doubleword pairs are interchanged on the stack.
Stack opcode: 16
Indicators: CCA on the new TOS double word



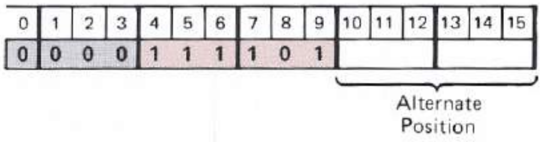
XCH Exchange A and B. The top two words of the stack are interchanged.
Stack opcode: 32
Indicators: CCA on the new TOS



XAX Exchange A and X. The content of the TOS and the Index register are interchanged.
Stack opcode: 35
Indicators: CCA on the new TOS

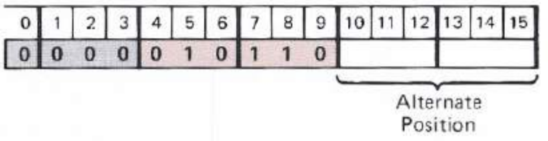


CAB Rotate A,B,C. The third word of the stack is removed from the stack, the two top words are compressed onto the rest of the stack, and the original third word is pushed onto the stack.
Stack opcode: 56
Indicators: CCA on the new TOS

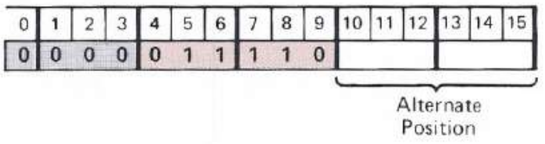


XBX Exchange B and X. The second word of the stack is interchanged with the content of the Index register.
Stack opcode: 75
Indicators: unaffected

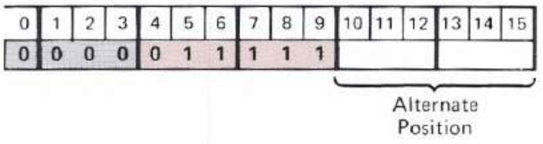
## INDEX INSTRUCTIONS



STBX Store B into X. The second word of the stack replaces the content of the Index register.
Stack opcode: 26
Indicators: CCA on the new X



ADAX Add A to X. The TOS is added in integer form to the content of the Index register. The sum replaces the content of the Index register, and the TOS is deleted.
Stack opcode: 36
Indicators: CCA on the new X, Carry, Overflow



ADXA Add X to A. The content of the Index register is added to the TOS, and the sum replaces the TOS.
Stack opcode: 37
Indicators: CCA on the new TOS, Carry, Overflow

LDXB Load X into B. The second word of the stack is replaced by the content of the Index register. The TOS is unaffected.
Stack opcode: 42
Indicators: CCA on the new B

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | | | | | | |

Alternate Position

STAX Store A into X. The TOS replaces the content of the Index register, and TOS is deleted from the stack.
Stack opcode: 43
Indicators: CCA on the new X

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | | | | | | |

Alternate Position

LDXA Load X onto stack. The content of the Index register is pushed onto the stack.
Stack opcode: 44
Indicators: CCA on the new TOS

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | | | | | | |

Alternate Position

ADBX Add B to X. The second word of the stack is added in integer form to the content of the Index register, and the result replaces the content of the Index register.
Stack opcode: 76
Indicators: CCA on the new X, Carry, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | | | | | |

Alternate Position

ADXB Add X to B. The content of the Index register is added in integer form to the second word of the stack, and the sum replaces the second word of the stack.
Stack opcode: 77
Indicators: CCA on the new B, Carry, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | |

Alternate Position

## FLOATING POINT INSTRUCTIONS

DFLT Double float. Converts the doubleword integer contained in the top two words of the stack to a floating point number with rounding.
**Instruction Commentary 2.**
Stack opcode: 30
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | | | | | | |

Alternate Position

FLT Float. Converts the integer on the TOS to a 32-bit floating point number with rounding. The TOS is deleted and the doubleword floating point result is pushed onto the stack.
**Instruction Commentary 2.**
Stack opcode: 47
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | | | | | | |

Alternate Position

FCMP Floating compare. The Condition Code is set to pattern C as a result of the floating point comparison of D,C with B,A. The two floating point double words are deleted.
Stack opcode: 50
Indicators: CCC

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | | | | | | |

Alternate Position

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | | | | | | |

Alternate Position

FADD Floating add. The two floating point numbers contained in the top four words of the stack are added in floating point form. The top four words of the stack are deleted and the two-word sum is pushed onto the stack.

**Instruction Commentary 2.**

Stack opcode: 51

Indicators: CCA, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | | | | | |

Alternate Position

FSUB Floating subtract. The floating point number contained in the top two words of the stack is subtracted in floating point form from the floating point number contained in the third and fourth words of the stack. The top four words of the stack are deleted and the two-word difference is pushed onto the stack.

**Instruction Commentary 2.**

Stack opcode: 52

Indicators: CCA, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | | | | | | |

Alternate Position

FMPY Floating multiply. The two floating point numbers contained in the top four words of the stack are multiplied in floating point form. The top four words of the stack are deleted and the two-word result is pushed onto the stack.

**Instruction Commentary 2.**

Stack opcode: 53

Indicators: CCA, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | | | | | | |

Alternate Position

FDIV Floating divide. The floating point number contained in the third and fourth words of the stack is divided by the floating point number contained in the top two words of the stack. The top four words of the stack are deleted and the two-word quotient is pushed onto the stack.

**Instruction Commentary 2.**

Stack opcode: 54

Indicators: CCA, Overflow

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | | | | | | |

Alternate Position

FNEG Floating negate. The floating point number contained in the top two words of the stack is negated in floating point form.

Stack opcode: 55

Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | | | | | | |

Alternate Position

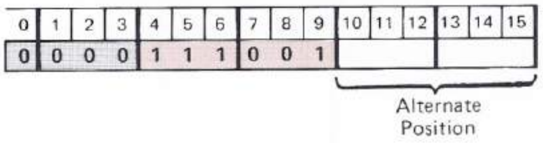
FIXR Fix and round. The floating point number contained in the top two words of the stack is converted to fixed point form and rounded. Carry is cleared if the low order 16 bits of the doubleword result (TOS) represent the true integer value (i.e., if the high order 17 bits are either all zeros or all ones); otherwise, Carry is set.

**Instruction Commentaries 1 and 2.**

Stack opcode: 70

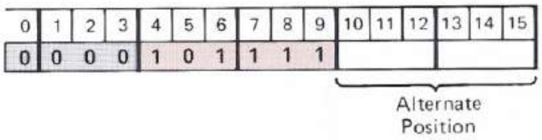
Indicators: CCA, Carry, Overflow

FIXT Fix and truncate. The floating point number contained in the top two words of the stack is converted to fixed point form and truncated. Carry is cleared if the low order 16 bits of the doubleword result (TOS) represent the true integer value (i.e., if the high order 17 bits are either all zeros or all ones); otherwise, Carry is set.
**Instruction Commentaries 1 and 2.**
Stack opcode: 71
Indicators: CCA, Carry, Overflow

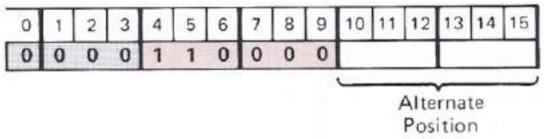


## LOGICAL INSTRUCTIONS

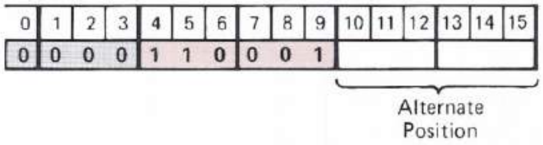
LCMP Logical compare. The Condition Code is set to pattern C as a result of the comparison of the second word of the stack with the TOS. The two words are then deleted from the stack.
Stack opcode: 57
Indicators: CCC



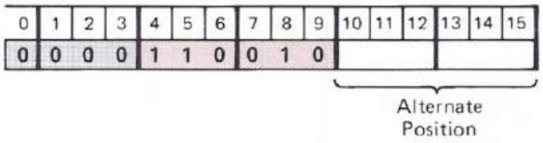
LADD Logical add. The top two words of the stack are added as 16-bit positive integers, and they are deleted from the stack. The resulting sum is pushed onto the stack.
Stack opcode: 60
Indicators: CCA (as a 2's complement result), Carry



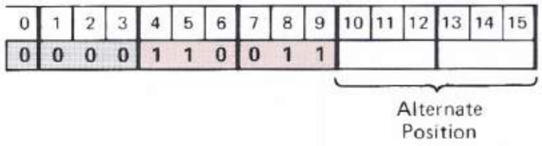
LSUB Logical subtract. The top word of the stack is subtracted in logical form from the second word and they are deleted. The resulting difference is pushed onto the stack.
Stack opcode: 61
Indicators: CCA (as a 2's complement result), Carry



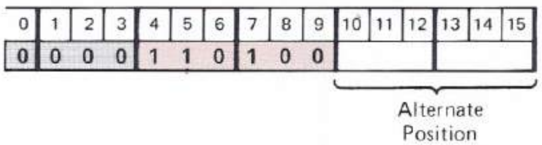
LMPY Logical multiply. The top two words of the stack are multiplied as 16-bit positive integers. The words are replaced by the double length product with the least significant half on the TOS. Carry is cleared if the TOS word of the result represents the true integer value (i.e., if the high order 16 bits are all zeros); otherwise, Carry is set.
**Instruction Commentary 1.**
Stack opcode: 62
Indicators: CCA (as a 2's complement result), Carry



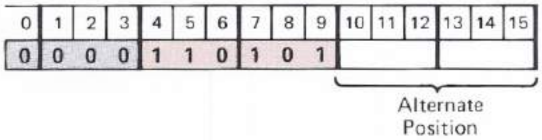
LDIV Logical divide. The 32-bit positive integer in the second and third words of the stack is divided by the 16-bit positive integer on the TOS (C,B ÷ A). The top three words are deleted. The quotient is pushed onto the stack (B) and then the remainder (A). If overflow occurs, the quotient will be modulo $2^{16}$
Stack opcode: 63
Indicators: CCA on quotient (as a 2's complement result), Carry



NOT One's complement. The top word of the stack is converted to its one's complement.
Stack opcode: 64
Indicators: CCA

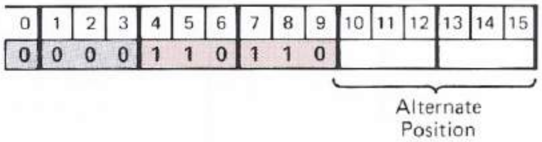


## BOOLEAN INSTRUCTIONS



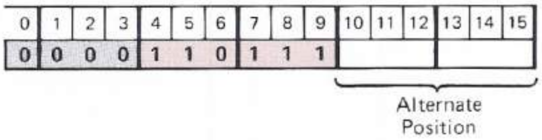
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | | | | | | |

OR Logical OR. The top two words of the stack are merged by a logical inclusive-OR. The two words are deleted and the result is pushed onto the stack.
Stack opcode: 65
Indicators: CCA on the new TOS



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | | | | | | |

XOR Logical exclusive-OR. The top two words of the stack are combined by a logical exclusive-OR. The two words are deleted and the result is pushed onto the stack.
Stack opcode: 66
Indicators: CCA on the new TOS

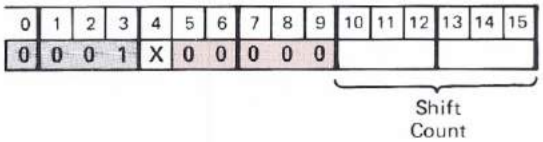


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | | | | | | |

AND Logical AND. The top two words of the stack are combined by a logical AND. The two words are deleted and the result is pushed onto the stack.
Stack opcode: 67
Indicators: CCA on the new TOS

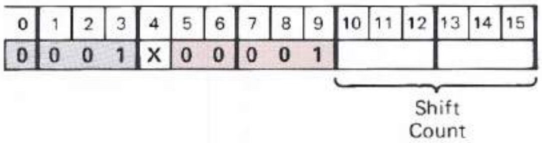
# SINGLE WORD SHIFT INSTRUCTIONS

All single word shift instructions: **Instruction Commentary 3.**



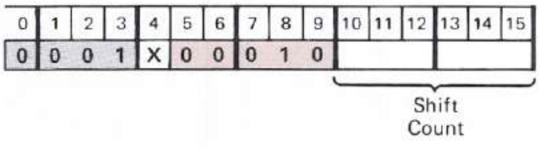
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 0 | 0 | 0 | | | | | | |

ASL Arithmetic shift left. The TOS is shifted left n bits, preserving the sign bit. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified (bit 4), the content of the Index register.
Sub-opcode 1: 00
Indicators: CCA



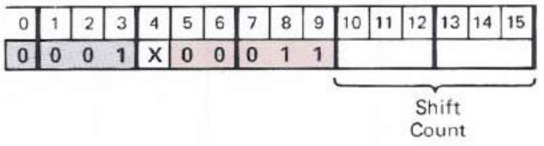
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 0 | 0 | 1 | | | | | | |

ASR Arithmetic shift right. The TOS is shifted right n places, propagating the sign bit. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 01
Indicators: CCA



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 0 | 1 | 0 | | | | | | |

LSL Logical shift left. The TOS is shifted left n bits logically. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 02
Indicators: CCA



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 0 | 1 | 1 | | | | | | |

LSR Logical shift right. The TOS is shifted right n bits logically. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 03
Indicators: CCA

CSL Circular shift left. The TOS is shifted left n bits circularly. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 04
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | | | | | | |

Shift Count

CSR Circular shift right. The TOS is shifted right n bits circularly. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 05
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 1 | | | | | | |

Shift Count

## DOUBLE WORD SHIFT INSTRUCTIONS

All double word shift instructions: **Instruction Commentaries 3 and 4.**

DASL Double arithmetic shift left. The double word contained in the top two words of the stack is shifted left n bits, preserving the sign bit (bit 0 of B). The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 20
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 0 | 0 | 0 | | | | | | |

Shift Count

DASR Double arithmetic shift right. The double word contained in the top two words of the stack is shifted right n bits, propagating the sign bit (bit 0 of B). The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 21
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 0 | 0 | 1 | | | | | | |

Shift Count

DLSL Double logical shift left. The double word contained in the top two words of the stack is shifted left n bits logically. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 22
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 0 | 1 | 0 | | | | | | |

Shift Count

DLSR Double logical shift right. The double word contained in the top two words of the stack is shifted right n bits logically. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 23
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 0 | 1 | 1 | | | | | | |

Shift Count

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 1 | 0 | 0 |  |  |  |  |  |  |

Shift Count

DCSL Double circular shift left. The double word contained in the top two words of the stack is shifted left n bits circularly. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 24
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 | 1 | 0 | 1 |  |  |  |  |  |  |

Shift Count

DCSR Double circular shift right. The double word contained in the top two words of the stack is shifted right n bits circularly. The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 25
Indicators: CCA

## TRIPLE WORD SHIFT INSTRUCTIONS

All triple word shift instructions: Instruction Commentaries 3 and 5.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 1 | 0 | 0 | 0 |  |  |  |  |  |  |

Shift Count

TASL Triple arithmetic shift left. The triple word integer contained in the top three words of the stack is shifted left n bits, preserving the sign bit (bit 0 of C). The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Subopcode 1: 10
Indicators: CCA on the new TOS triple word

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 1 | 0 | 0 | 1 |  |  |  |  |  |  |

Shift Count

TASR Triple arithmetic shift right. The triple word integer contained in the top three words of the stack is shifted right n bits, propagating the sign bit (bit 0 of C). The value of n (modulo 64) is the number specified in the argument field plus, if X is specified, the content of the Index register.
Sub-opcode 1: 11
Indicators: CCA on the new TOS triple word

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Reserved

TNSL Triple normalizing shift left. The top three words of the stack are shifted left arithmetically until bit 6 of C is a "1". Bits 0 through 5 of C are cleared ("0"). The shift count is stored in the Index register. The instruction initially clears the Index register unless X is specified ("1" in bit 4 of the instruction).
Sub-opcode 1: 16
Indicators: CCA on final value of top three words

# BRANCH INSTRUCTIONS

IABZ Increment A, branch if zero. The TOS is incremented. If the result is then zero, control is transferred to P ± displacement; otherwise to P+1.
Sub-opcode 1: 07
Indicators: CCA, Carry, Overflow
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | ± | | | | | |

Displacement

IXBZ Increment X, branch if zero. The Index register is incremented. If the result is then zero, control is transferred to P ± displacement; otherwise to P+1.
Sub-opcode 1: 12
Indicators: CCA, Carry, Overflow
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | ± | | | | | |

Displacement

DXBZ Decrement X, branch if zero. The Index register is decremented. If the result is then zero, control is transferred to P ± displacement; otherwise to P+1.
Sub-opcode 1: 13
Indicators: CCA, Carry, Overflow
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | ± | | | | | |

Displacement

BCY Branch on carry. If the Carry bit of the Status register is set ("1"), control is transferred to P ± displacement; otherwise to P+1.
Sub-opcode 1: 14
Indicators: Carry cleared
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | ± | | | | | |

Displacement

BNCY Branch on no carry. If the Carry bit of the Status register is clear ("0"), control is transferred to P ± displacement; otherwise to P+1.
Sub-opcode 1: 15
Indicators: Carry cleared
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | ± | | | | | |

Displacement

CPRB Compare range and branch. The integer in the Index register is tested to determine if it is within the interval defined by the upper bound integer on the TOS and the lower bound integer in the second word of the stack. The Condition Code is set by the comparison to a special pattern: CCE if within range, CCL if below range, CCG if above range. If the integer in the Index register is within the specified range, control is then transferred to P ± displacement; otherwise to P+1. The top two elements of the stack are deleted in either case.
Sub-opcode 1: 26
Indicators: CCE, CCL, CCG
Addressing modes: P relative (+/–)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | ± | | | | | |

Displacement

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | I | 1 | 0 | 1 | 1 | 1 | ± | | | | | |

Displacement

DABZ Decrement A, branch if zero. The TOS is decremented. If the result is then zero, control is transferred to P ± displacement; otherwise to P+1.

Sub-opcode 1: 27

Indicators: CCA, Carry, Overflow

Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | I | 1 | 1 | 0 | 0 | 0 | ± | | | | | |

Displacement

BOV Branch on overflow. If the Overflow bit of the Status register is set ("1"), control is transferred to P ± displacement; otherwise to P+1.

Sub-opcode 1: 30

Indicators: Overflow cleared

Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | I | 1 | 1 | 0 | 0 | 1 | ± | | | | | |

Displacement

BNOV Branch on no overflow. If the Overflow bit of the Status register is clear ("0"), control is transferred to P ± displacement; otherwise to P+1.

Sub-opcode 1: 31

Indicators: Overflow cleared

Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | I | 1 | 1 | 1 | 1 | 0 | ± | | | | | |

Displacement

BRO Branch on TOS odd. If the TOS is odd (bit 15 = 1), control is transferred to P ± displacement; otherwise to P+1. The TOS is deleted.

Sub-opcode 1: 36

Indicators: unaffected

Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | I | 1 | 1 | 1 | 1 | 1 | ± | | | | | |

Displacement

BRE Branch on TOS even. If the TOS is even (bit 15 = 0), control is transferred to P ± displacement; otherwise to P+1. The TOS is deleted.

Sub-opcode 1: 37

Indicators: unaffected

Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | X | I | 0 | ± | | | | | | | | |

Displacement
P Relative

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | X | 1 | 1 | | | | | | | | | |
| | | | | | | DB+ | 0 | Displacement | | | | | | | |
| | | | | | | Q+ | 1 | 0 | | | | | | | |
| | | | | | | Q- | 1 | 1 | 0 | | | | | | |
| | | | | | | S- | 1 | 1 | 1 | | | | | | |

BR Branch unconditionally. For P relative mode, control is transferred unconditionally to P ± displacement, plus (if specified) the value in X; may be indirect. For DB, Q, and S relative modes, control is transferred indirectly (only) via the location specified by DB, Q, or S ± the displacement; the content of the location so specified is added to PB (plus post-indexing if X is specified) to obtain the effective address for P.

### Instruction Commentary 6.

Memory opcode: 14, bits 5,6 = 00, 10, or 11

Indicators: unaffected

Addressing modes: P relative (+/-), direct or indirect
DB+ relative, indirect
Q+ relative, indirect
Q- relative, indirect
S- relative, indirect
Indexing available

BCC Branch on Condition Code. The Condition Code in the Status register is compared with conditions named in the CCF field of the instruction. If the named conditions are met, control is transferred to P ± displacement; otherwise to P+1. The displacement is limited to ±31. Control is transferred to the branch address under the following conditions:

If CCF = 0, never branch
= 1, branch if CC = CCL
= 2, branch if CC = CCE
= 3, branch if CC = CCL or CCE
= 4, branch if CC = CCG
= 5, branch if CC = CCG or CCL
= 6, branch if CC = CCG or CCE
= 7, always branch

Memory opcode: 14, bits 5,6 = 01
Indicators: unaffected
Addressing modes: P relative (+/-)
Direct or indirect

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | I | 0 | 1 | G | E | L | ± | | | | | |
| | | | | | | | CCF | | | Displacement | | | | | |

# BIT TEST INSTRUCTIONS

SCAN Scan bits. The TOS is shifted left until bit 0 contains a "1", then is shifted left one more bit. The shift count is left in the Index register, indicating the bit position which contained the "1". The instruction normally sets the Index register to -1 before beginning the shifts. However, if X is specified, the shift count adds on to the existing Index register content. If TOS is all zeros, the count will be 16 if unindexed, or X + 16 if indexed.
Sub-opcode 1: 06
Indicators: CCA on final TOS

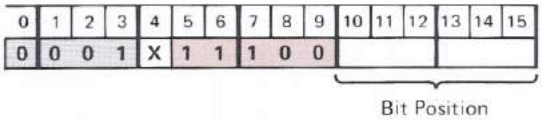
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | Reserved | | | | | |

TBC Test bit and set Condition Code. One bit of the TOS word is tested and the Condition Code is set to a special pattern depending on the state of the bit. The bit position to be tested is specified by the argument field of the instruction plus, if X is specified, the content of the Index register. If the number specified exceeds 15, the bit position indicated is modulo 16; e.g., bit 0 is tested for counts of 0, 16, 32, 48, etc.
Sub-opcode 1: 32
Indicators: CCE if the bit was "0"
CCL or CCG if the bit was "1"

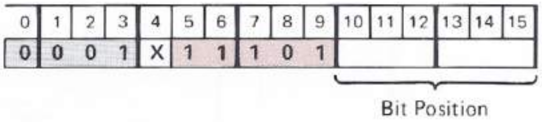
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 1 | 0 | 1 | 0 | | | | | | |
| | | | | | | | | | | Bit Position | | | | | |

TRBC Test and reset bit, set Condition Code. The operation of this instruction is identical to that of TBC except that the tested bit is reset to "0" after the test.
Sub-opcode 1: 33
Indicators: CCE if the bit was "0"
CCL or CCG if the bit was "1"

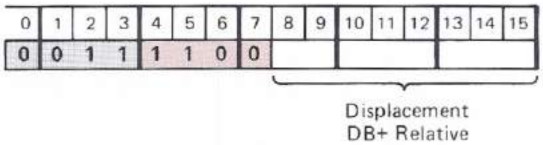
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 1 | 0 | 1 | 1 | | | | | | |
| | | | | | | | | | | Bit Position | | | | | |



TSBC Test and set bit, set Condition Code. The operation of this instruction is identical to that of TBC except that the tested bit is set to "1" after the test.
Sub-opcode 1: 34
Indicators: CCE if the bit was "0"
CCL or CCG if the bit was "1"



TCBC Test and complement bit, set Condition Code. The operation of this instruction is identical to that of TBC except that the tested bit is complemented after the test.
Sub-opcode 1: 35
Indicators: CCE if the bit was "0"
CCL or CCG if the bit was "1"



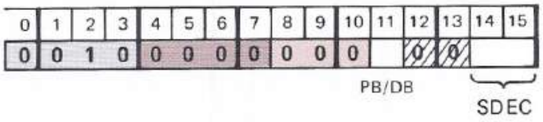
TSBM Test and set bits in memory, set Condition Code. A mask word on the TOS is compared by logical AND with the contents of the memory location specified by DB + displacement; the result replaces the TOS and sets the Condition Code to pattern A. At the same time, the mask word is merged (logical OR) with the content of the specified location and this result is stored back in the memory cell. Displacement range is 0 through +255. (The memory location is set to all "1s" during execution until the merged result is stored back in the cell; solves the hardware "semaphore" problem with dual CPU configurations.)

**Instruction Commentary 7.**

Sub-opcode 3: 14
Indicators: CCA on the new TOS
Addressing mode: DB+ relative

## MOVE INSTRUCTIONS

Note: All Move instructions are interruptable after each word (or byte) transfer and will continue from the point of interrupt when control is returned to the instruction.



MOVE Move words. This instruction transfers a specified number of words from one area of primary memory to another. The instruction expects a signed word count in A, a DB or PB relative displacement for a source address in B, and a DB relative displacement for a target address in C. As long as the word count in A has not been counted to zero, the transferring of data will continue as follows: The content of the memory location specified by DB + B or PB + B is transferred to the location specified by DB + C. If the word count in A is positive, the source and target displacement values in B and C are incremented by one on each transfer, and the word count is decremented by one. If the word count in A is negative, the source and target displacement values in B and C are decremented by one on each transfer, and the word count is incremented by one. Note that the word count is always changed by one toward zero. On completion of the block transfer, the instruction deletes from the stack the number of words specified by the SDEC

(S decrement) field of the instruction; the range of this field is 0 through 3.

### Instruction Commentary 8.

Move opcode: 0
Indicators: unaffected
Addressing modes: DB+ or PB+ for source
DB+ for target

MVB Move bytes. The MVB instruction transfers a specified number of bytes from one area of primary memory to another. The instruction expects a signed byte count in A, a DB or PB relative displacement for a source byte address in B, and a DB relative displacement for a target byte address in C. As long as the word count in A has not been counted to zero, the transferring of data will continue as follows: The content of the byte address location specified by DB + B or PB + B is transferred to the byte address location specified by DB + C. If the byte count in A is positive, the source and target displacement values in B and C are incremented by one on each transfer, and the byte count is decremented by one. If the byte count in A is negative, the source and target displacement values in B and C are decremented by one on each transfer, and the byte count is incremented by one. Note that the byte count is always changed by one toward zero. On completion of the block transfer, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified by the SDEC field of the instruction. This instruction can use split stack.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |  | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  | PB/DB |  |  | SDEC |  |

### Instruction Commentary 8.

Move opcode: 1
Indicators: unaffected
Addressing modes: Byte addressing
DB+ or PB+ for source
DB+ for target

MVBL Move words from DB+ to DL+. This instruction transfers a specified number of words from the DB+ area of the data segment to the DL+ area. The instruction expects a signed word count in A, a DB relative displacement for a source address in B, and a DL relative displacement for a target address in C. As long as the word count in A has not been counted to zero, the transferring of data will continue as follows: The contents of the memory location specified by DB + B is transferred to the location specified by DL + C. If the word count in A is positive, the source and target displacement values in B and C are incremented by one on each transfer, and the word count is decremented by one. If the word count in A is negative, the source and target displacement values in B and C are decremented by one on each transfer, and the word count is incremented by one. Note that the word count is always changed by one toward zero. On completion of the block transfer, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified by the SDEC field of the instruction. This instruction can use split stack.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | SDEC |  |

### Instruction Commentary 9.

Move opcode: 2, bit 11 = 0
Indicators: unaffected
Addressing modes: DB+ for source
DL+ for target
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | SDEC |  |

SCW

Scan while memory bytes equal test byte. The SCW instruction expects the TOS to contain a test character in the right byte and a terminal character in the left byte. The second word of the stack contains a DB relative displacement for a source byte address. The source byte is tested against the test character. If they are equal the source byte address is incremented and the next byte is tested. This continues until a source byte is found that is not the same as the test character. If the last character scanned is the same as the terminal character, the Carry bit is set; if not, the Carry bit is cleared. On completion of the scan, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified in the SDEC field of the instruction. This instruction can use split stack.

Move opcode: 2, bit 11 = 1

Indicators: Carry
CCB on the last character scanned

Addressing mode: Byte addressing, DB+

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | SDEC |  |

MVLB

Move words from DL+ to DB+. This instruction transfers a specified number of words from the DL+ area of the data segment to the DB+ area. The instruction expects a signed word count in A, a DL relative displacement for a source address in B, and a DB relative displacement for a target address in C. As long as the word count in A has not been counted to zero, the transferring of data will continue as follows: The contents of the memory location specified by DL + B is transferred to the location specified by DB + C. If the word count in A is positive, the source and target displacement values in B and C are incremented by one on each transfer, and the word count is decremented by one. If the word count in A is negative, the source and target displacement values in B and C are decremented by one on each transfer, and the word count is incremented by one. Note that the word count is always changed by one toward zero. On completion of the block transfer, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified by the SDEC field of the instruction.

**Instruction Commentary 9.**

Move opcode: 3, bit 11 = 0

Indicators: unaffected

Addressing modes: DL+ for source
DB+ for target

*This is a privileged instruction.*

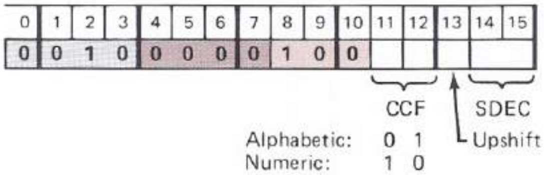
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | SDEC |  |

SCU

Scan until memory byte equals test byte or terminal byte. The SCU instruction expects the TOS to contain a test character in the right byte and a terminal character in the left byte. The second word of the stack contains a DB relative displacement for a source byte address. The source byte is tested against the test and terminal characters. If the source byte differs from both of these characters, the byte address is incremented and the next byte is tested. This continues until either the test character or the terminal character is encountered. The address of the character remains in the second word of the stack. If the last character scanned was the same as the test character, the Carry bit is cleared; if it was the same as the terminal character,

Carry is set. On completion of the scan, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified in the SDEC field of the instruction.
Move opcode: 3, bit 11 = 1
Indicators: Carry
Addressing mode: byte addressing, DB+

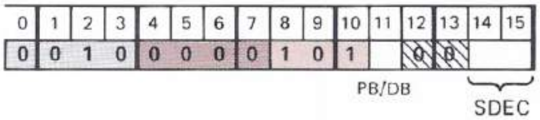
MVBW Move bytes while of specified type. This instruction transfers an unspecified number of bytes from one area of primary memory to another. The instruction expects a source byte address in the TOS and a DB relative displacement for a target byte address in the second word of the stack. As long as the source byte is of the type specified in the CCF field, it is moved to the target area. The target displacement value in B is incremented by one on each transfer. If the byte to be moved is a lower case letter and the upshift bit is on, the target byte will be an upshifted copy of the source byte. Byte transfers continue until the source byte is not of the proper type. On completion of the block transfer, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified by the SDEC field of the instruction. This instruction can use split stack.



**Instruction Commentary 8.**

Move opcode: 4
Indicators: CCB on the last character scanned
Addressing mode: Byte addressing, DB+

CMPB Compare bytes. This instruction scans two byte strings simultaneously until the compared bytes are unequal or until a specified number of comparisons have been made. CMPB expects a signed byte count in A, a DB or PB relative displacement for a source byte address in B, and a DB relative displacement for a target byte address in C. As long as the word count in A has not been counted to zero, the comparison proceeds as follows: The content of the byte address location specified by DB + B or PB + B is compared with the content of the byte address location specified by DB + C. If the byte count in A is positive, the source and target displacement values in B and C are incremented by one after each comparison, and the byte count is decremented by one. If the byte count in A is negative, the source and target displacement values in B and C are decremented by one after each comparison, and the byte count is incremented by one. Note that the byte count is always changed by one toward zero. The instruction terminates when either a comparison fails or the byte count in the TOS reaches zero. The Condition Code is set to a special pattern to indicate the terminating condition. On termination, the instruction deletes from the stack the number of words (0, 1, 2, or 3) specified by the SDEC field of the instruction. This instruction can use split stack.



**Instruction Commentary 8.**

Move opcode: 5
Indicators: CCE if byte count = 0
CCG if target byte > source byte (final)
CCL if target byte < source byte (final)
Addressing modes: Byte addressing
DB+ or PB+ for source
DB+ for target

# SPECIAL INSTRUCTIONS

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

RSW Read Switch register. The content of the Switch register is pushed onto the stack.
Mini-opcode: 14, bit 15 = 0
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

LLSH Linked List Search. This instruction searches through a linked list in memory, comparing a test word with a target word, until either: a target word is found that is equal to or greater than the test word, or a target word is all ones, or the specified count has been counted to zero. The instruction expects a positive count value in the Index register. Also, A must contain an absolute pointer into the linked list, B must contain the test word, and C is an offset which indicates the position, relative to each link, of the target location. At each step, the test word is compared to the content of the target location. If the target content is logically greater than or equal to the test word, or if the target content is all ones, the instruction terminates. Otherwise, the next link replaces the current link in A, the count in the Index register is decremented, and the instruction repeats until the count becomes zero. **Instruction Commentary 10.**
Mini-opcode: 14, bit 15 = 1
Indicators: CCL if terminated by X = 0
CCE if terminated by target ⩾ B
CCG if terminated by target = $2^{16} - 1$
Addressing mode: absolute ± offset
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

PLDA Privileged load from absolute address. The content of the Index register is a 16-bit absolute address; the content of this address is pushed onto the stack.
Mini-opcode: 15, bit 15 = 0
Indicators: CCA
Addressing mode: absolute
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |

PSTA Privileged store into absolute address. The content of the Index register is a 16-bit absolute address; the top word of the stack is stored into memory at that address, and then deleted from the stack.
Mini-opcode: 15, bit 15 = 1
Indicators: unaffected
Addressing mode: absolute
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | | | | | | | | |

Displacement
PL–

LLBL Load label. The label in the Segment Transfer Table (STT) at PL-N is loaded onto the TOS. The value N is a displacement given in the argument field of the instruction. If the label is local, it is converted to external type when loaded. To be valid, the value N must point to a location which is actually in the STT (i.e., N ⩽ STTL) in all cases; additionally, in the case of local labels, N must not exceed octal 177 (decimal 127), since this is the maximum range for external labels. An invalid value of N will invoke a STT Violation trap.

**Instruction Commentary 11.**

Sub-opcode 3: 07
Indicators: unaffected
Addressing mode: PL–

## IMMEDIATE INSTRUCTIONS

LDI Load immediate. The immediate operand N is pushed onto the stack. The value of N is given in the argument field of the instruction, and is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 02
Indicators: CCA on the new TOS

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

LDXI Load X immediate. The Index register is loaded with the immediate operand N. The value of N is given in the argument field of the instruction, and is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 03
Indicators: unaffected

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

CMPI Compare immediate. The Condition Code is set to pattern C as a result of the comparison of the TOS with the immediate operand N. The value of N is given in the argument field of the instruction, and is expressed as a positive integer in the range 0 through 255. The TOS is deleted.
Sub-opcode 2: 04
Indicators: CCC

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

ADDI Add immediate. The immediate operand N is added to the TOS in integer form, and the sum replaces the TOS. The value of N is given in the argument field of the instruction, and is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 05
Indicators: CCA on the new TOS, Carry, Overflow

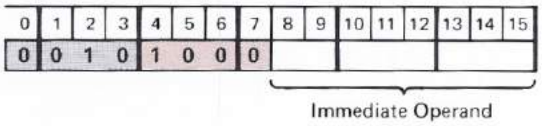
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

SUBI Subtract immediate. The immediate operand N is subtracted from the TOS in integer form, and the result replaces the TOS. The value of N is given in the argument field of the instruction, and is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 06
Indicators: CCA on the new TOS, Carry, Overflow

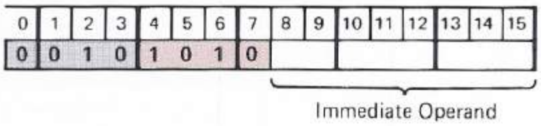
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

MPYI Multiply immediate. The immediate operand N is multiplied with the TOS in integer form; the 16-bit integer result replaces the TOS. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 07
Indicators: CCA on the new TOS, Overflow

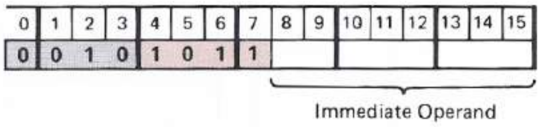
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |



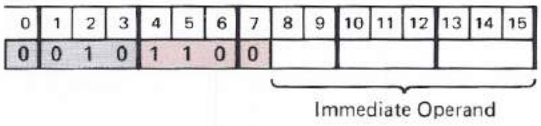
DIVI Divide immediate. The immediate operand N is divided into the TOS in integer form; the 16-bit integer quotient replaces the TOS. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 10
Indicators: CCA on the new TOS



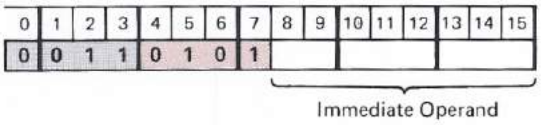
LDNI Load negative immediate. The immediate operand N is two's complemented and pushed onto the stack as a negative integer. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 12
Indicators: CCA on the new TOS, Overflow



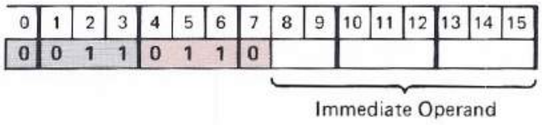
LDXN Load X negative immediate. The Index register is loaded with the 16-bit two's complement of the immediate operand N. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 2: 13
Indicators: unaffected



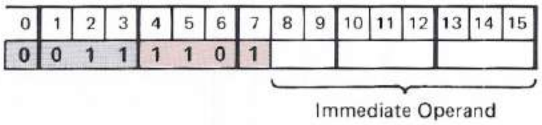
CMPN Compare negative immediate. The Condition Code is set to pattern C as a result of the comparison of the TOS with the two's complement of the immediate operand N. The value of N is expressed as a positive integer in the range 0 through 255. The TOS is deleted.
Sub-opcode 2: 14
Indicators: CCC



ADXI Add immediate to X. The immediate operand N is added to the content of the Index register in integer form. The sum replaces the Index register content. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 3: 05
Indicators: CCA on X



SBXI Subtract immediate from X. The immediate operand N is subtracted from the content of the Index register in integer form. The result replaces the Index register content. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 3: 06
Indicators: CCA on X



ORI Logical OR immediate. The immediate operand N is expanded to 16 bits with high order zeros and merged (inclusive OR) with the TOS; the result replaces the TOS. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 3: 15
Indicators: CCA

XORI Logical exclusive OR immediate. The immediate operand N is expanded to 16 bits with high order zeros and is combined by exclusive OR with the TOS; the result replaces the TOS. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 3: 16
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

ANDI Logical AND immediate. The immediate operand N is expanded to 16 bits with high order zeros and is combined by logical AND with the TOS; the result replaces the TOS. The value of N is expressed as a positive integer in the range 0 through 255.
Sub-opcode 3: 17
Indicators: CCA

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

# FIELD INSTRUCTIONS

EXF Extract field. A specified set of bits in the TOS are extracted and right justified, and the result, with high order zeros, replaces the TOS. The J field specifies the starting (leftmost) bit number in the source field, and the K field specifies the number of bits to be extracted.
**Instruction Commentary 12.**
Sub-opcode 2: 15
Indicators: CCA on the new TOS

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | | | | | | | | |
| | | | | | | | | J<br>Starting<br>Bit # | | | | K<br>Number<br>of bits | | | |

DPF Deposit field. A specified number of the least significant bits of the TOS are deposited in the second word of the stack, beginning at the bit number specified by the J field; the remaining bits of the second word of the stack are unchanged. The K field specifies the number of bits to be deposited. The source operand is deleted from the stack.
**Instruction Commentary 12.**
Sub-opcode 2: 16
Indicators: CCA on the new TOS

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | | | | | | | | |
| | | | | | | | | J<br>Starting<br>Bit # | | | | K<br>Number<br>of bits | | | |

# REGISTER CONTROL INSTRUCTIONS

PSHR Push registers. The content of a register (or the displacement it represents) specified by any bit 9 through 15 is pushed onto the stack. If more than one register (or displacement) is specified, the contents will be stacked in the order shown below, such that if all seven were specified, DB would be on the TOS after execution, DL - DB next, etc. Note that when S-DB is pushed, the value stacked will be as it existed before the execution of this instruction. Stack overflow occurs if S+7 exceeds Z, regardless of the number of registers pushed.

If bit 15 = 1, push S-DB
If bit 14 = 1, push Q-DB

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | ▨0 | | | | | | | |
| | | | | | | | | | DB | DL | Z | Sta | X | Q | S |

If bit 13 = 1, push Index register
If bit 12 = 1, push Status register
If bit 11 = 1, push Z–DB
If bit 10 = 1, push DL–DB
If bit  9 = 1, push DB register

Sub-opcode 2: 11
Indicators: unaffected

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | | | | | | | |
| | | | | | | | | | DB | DL | Z | Sta | X | Q | S |

SETR Set registers. The registers specified by bits 9 through 15 of the instruction are filled by an absolute value from the TOS for the Index, Status, and DB registers, and an absolute value computed by adding DB to the TOS (displacement value) for the others. If more than one register (or displacement) is specified, the registers will be loaded in the order shown below, such that if all seven were specified, the DB-register would receive the first TOS and the value for S would be computed from the seventh TOS. The TOS is deleted after each register is set. If the Z-register is set to ZI, the Interrupt Stack flag is set to "1"; otherwise it is cleared. The Dispatcher flag is always cleared on setting Z. SETR is a privileged instruction except for setting the Index register, Q, S, and bits 2 and 4 through 7 of the Status register (user traps enable/disable, Overflow, Carry, and Condition Code).

*If bit  9 = 1, load DB from TOS
*If bit 10 = 1, load DL from (DB+TOS)
*If bit 11 = 1, load Z from (DB+TOS)
*If bit 12 = 1, load Status reg from TOS
If bit 12 = 1, and not privileged mode: load Status bits 2, 4 thru 7 from same bits of TOS
If bit 13 = 1, load Index register from TOS
If bit 14 = 1, load Q from (DB + TOS)
If bit 15 = 1, load S from (DB + TOS)

Sub-opcode 2: 17
Indicators: unaffected (may be changed if bit 12 = "1")
*These are privileged operations.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

XCHD Exchange DB and TOS. This instruction expects a new DB value on the TOS. The current DB replaces that value on the TOS while the new value is placed in the DB register. Bits 12 through 15 are ignored.
Special opcode: 03
Indicators: unaffected
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

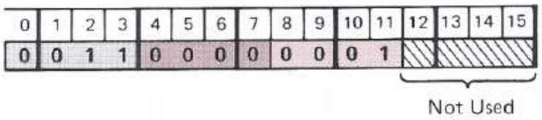
ADDS Add to S. The immediate operand N is added to S unless N is zero; if N is zero, the TOS content, minus one, is added to S instead.
**Instruction Commentary 13.**
Sub-opcode 3: 12
Indicators: unaffected

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | | | | | | | | |
| | | | | | | | | Immediate Operand | | | | | | | |

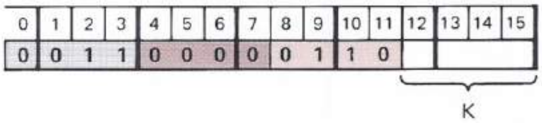
SUBS Subtract from S. The immediate operand N is subtracted from S unless N is zero; if N is zero, the TOS content, minus one, is subtracted from S instead.
**Instruction Commentary 13.**
Sub-opcode 3: 13
Indicators: unaffected

# PROGRAM CONTROL INSTRUCTIONS

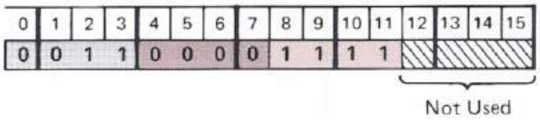
PAUS Pause. The computer hardware pauses; interrupts may occur. Bits 12 through 15 are ignored.
Special opcode: 01
Indicators: unaffected
*This is a privileged instruction.*



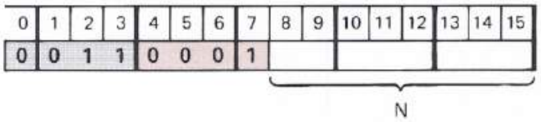
XEQ Execute stack word. The content of the word in the stack at S-K is placed in the Current Instruction Register to be executed. After execution, control is returned to the instruction after the XEQ unless a transfer of control was executed (branch, PCAL, etc.). If the word to be executed is a Stack Op, only the first position (bits 4 through 9) may be used; bits 10 through 15 must be a NOP. The value of K is 0 through 15 (decimal).
**Instruction Commentary 14.**
Special opcode: 06
Indicators: set by the executed instruction



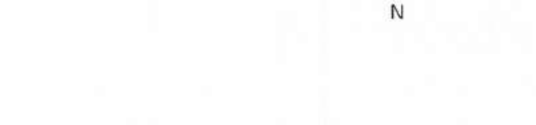
HALT The computer hardware halts; interrupts may not occur and manual intervention is required to restart the computer. Bits 12 through 15 are ignored.
Special opcode: 17
Indicators: unaffected
*This is a privileged instruction.*

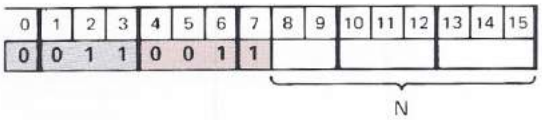


SCAL Subroutine call. Control is transferred to the location pointed to by the evaluation of the local label at PL-N, unless N is zero; if N is zero the local label is taken from the TOS and then deleted. The return address is then pushed onto the stack. Only local labels are allowed; non-local label gives STT Violation trap.
**Instruction Commentary 15.**
Sub-opcode 3: 01
Indicators: unaffected
Addressing modes:
Indirect via: PL - N (if N ≠ 0)
TOS (if N = 0)
Local Label: PB+



PCAL Procedure call. Control is transferred to the location pointed to by the evaluation of the program label at PL - N, unless N is zero; if N is zero, the program label is taken from the TOS and then deleted. Then a four word stack marker is placed on the stack, and Q and S are updated to point at this new marker. The program label may be local or external. If the Trace bit is on in the target CST entry, the PCAL will be made to the Trace segment. If a privileged user is calling a user segment, it will run in privileged mode.
**Instruction Commentary 16.**
Sub-opcode 3: 02
Indicators: unaffected
Addressing modes:
Indirect via: PL - N (if N ≠ 0)
TOS (if N = 0)
Local Label: PB+
External Label: via CST to local label in target segment



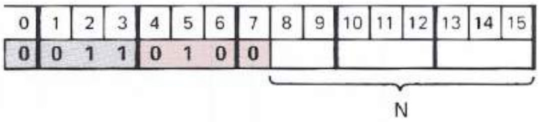


EXIT Exit from routine. This instruction is used to return from a routine called by the PCAL instruction or by an interrupt. A normal exit occurs by restoring the return address to P, restoring the previous contents of the Index and Status registers, and deleting all stack variables incurred by the called routine, plus its marker, plus N number of procedure parameters. The value of N may be any number from 0 through 255 for exits from PCAL routines; it must be 0 for exits from interrupt routines. An interrupt routine exits normally to the calling routine except when: a) exiting from the last routine to use the Interrupt Control Stack, or b) exiting from an external interrupt routine and there is another external interrupt pending. In case "a", the system automatically exits to the Dispatcher. In case "b", the new device number replaces the old device number on the stack (without changing the existing stack marker), and the new external interrupt is then processed. If the exit is from an external interrupt routine, EXIT clears the device's interrupt-active logic. If bit 0 of the return-P marker word is a "1", control is transferred to the Trace segment. If the return segment is absent, control is transferred to the Absence segment.

**Instruction Commentary 16.**

Sub-opcode 3: 03

Indicators: unaffected



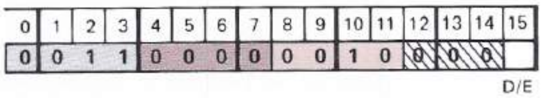
SXIT Exit from subroutine. This instruction is used to return from a subroutine called by the SCAL instruction. The SXIT instruction assumes that the return address is on the TOS, and returns program control to this address. The TOS is then deleted, plus N number of subroutine parameters. The value of N may be any number from 0 through 255.

**Instruction Commentary 15.**

Sub-opcode 3: 04

Indicators: unaffected

## I/O AND INTERRUPT INSTRUCTIONS



SED Set "enable/disable external interrupts" bit. The interrupt system is enabled or disabled according to the least significant bit (bit 15) of the instruction. If bit 15 is a "1", bit 1 of the Status register is set, thus enabling external interrupts. If bit 15 is a "0", bit 1 of the Status register is cleared, thus disabling external interrupts. Bits 12, 13, and 14 of the instruction are ignored.

Special opcode: 02

Indicators: unaffected

*This is a privileged instruction.*

SMSK Set mask. The SMSK instruction assumes that the TOS contains the mask word and transmits this word to all device controllers. Each "1" bit in the mask word sets each Mask flip-flop in the group of device controllers which are specifically wired to be controlled by that bit. Each "0" bit in the mask clears each Mask flip-flop in its group. If there is an I/O error (no acknowledgement), it means that the external interrupt system is in an unknown state. In this case, the SMSK instruction sets CCL Condition Code, and leaves the mask on the TOS. If there is no I/O error, the SMSK instruction deletes the mask from the stack and sets the Condition Code to CCE.

Special opcode: 04

Indicators: CCE if no error
CCL if error

*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ▨ | ▨ | ▨ | ▨ |
| | | | | | | | | | | | | Not Used | | | |

RMSK Read mask. This instruction transfers the 16-bit mask word from the Mask register to the TOS.

Special opcode: 05

Indicators: unaffected

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | ▨ | ▨ | ▨ | ▨ |
| | | | | | | | | | | | | Not Used | | | |

SIO Start I/O. The SIO instruction expects the absolute starting address of an I/O program to be on the TOS, and a device number to be in the stack at S-K. The instruction first checks if the device is ready by checking bit 0 of the device controller's Status register. If it is ready (bit = "1"), the TOS is stored into the first word location of the DRT entry for the device specified at S-K; an SIO command is then issued to the device controller to begin execution of its I/O program, the TOS is deleted, and the Condition Code is set to CCE. If the device is not ready (bit 0 of device status = "0"), the content of the device controller's Status register is pushed onto the stack and the Condition Code is set to CCG. If the device controller does not respond to the readiness test, the Condition Code is set to CCL and the instruction is terminated.

Special opcode: 07

Indicators: CCL = non-responding device controller
CCE = device ready
CCG = device not ready

*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | | | | |
| | | | | | | | | | | | | K | | | |

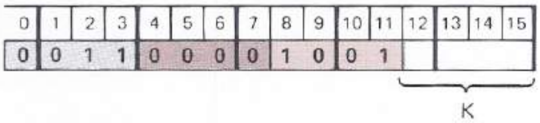
RIO Read I/O. This instruction expects a device number to be given in the stack at S-K. RIO first checks if the device is ready by checking bit 1 of the device controller's Status register. If it is ready (bit = "1"), the 16-bit direct data word from the device is pushed onto the stack and the Condition Code is set to CCE. If it is not ready (bit = "0"), the content of the device controller's Status register is pushed onto the stack and the Condition Code is set to CCG. If the device controller does not respond to the readiness test, the Condition Code is set to CCL and the instruction is terminated.

Special opcode: 10

Indicators: CCL = non-responding device controller
CCE = device ready
CCG = device not ready

*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | | | |
| | | | | | | | | | | | | K | | | |



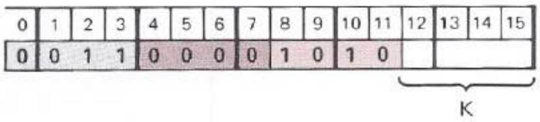
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | | | | |

WIO Write I/O. This instruction assumes that the TOS contains a direct data word and expects a device number to be given in the stack at S-K. WIO first checks if the device is ready by checking bit 1 of the device controller's Status register. If it is ready (bit = "1"), the word is transmitted to the specified device and then deleted from the stack; the Condition Code is set to CCE. If it is not ready (bit = "0"), the content of the device controller's Status register is pushed onto the stack and the Condition Code is set to CCG. If the device controller does not respond to the readiness test, the Condition Code is set to CCL and the instruction is terminated.

Special opcode: 11

Indicators: CCL = non-responding device controller
CCE = device ready
CCG = device not ready

*This is a privileged instruction.*



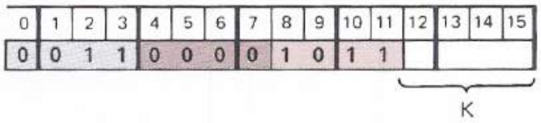
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | | | |

TIO Test I/O. This instruction expects a device number to be given in the stack at S-K. TIO obtains a copy of the device status word from the device controller, pushes it onto the stack, and sets the Condition Code to CCE. If the device controller does not respond, the Condition Code is set to CCL and the instruction is terminated.

Special opcode: 12

Indicators: CCE = responding device controller
CCL = non-responding device controller

*This is a privileged instruction.*



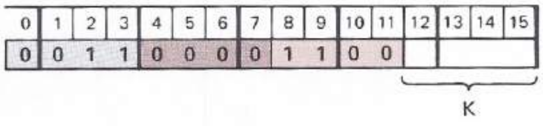
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | | | | |

CIO Control I/O. This instruction assumes that the TOS contains a control word and expects a device number to be given in the stack at S-K. CIO transmits the TOS to the specified device controller, along with a CIO signal. If the device controller acknowledges receiving the word, the TOS is deleted and the Condition Code is set to CCE. If the device controller does not respond, the Condition Code is set to CCL and the instruction is terminated.

Special opcode: 13

Indicators: CCE = responding device controller
CCL = non-responding device controller

*This is a privileged instruction.*



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | | | | |

CMD Command. This instruction assumes that the TOS contains a 16-bit data word to be sent to a system hardware module and expects a command word in the stack at S-K. Bits 13 through 15 of the command word specify the module number, and bits 10 and 11 are used to specify a module command. (The four possible commands depend upon application and do not form a part of this instruction's definition.) CMD sends the 16-bit data word and 2-bit command over the central data bus to the specified module, and then deletes the TOS. (Note: if the destination module is not ready, the CPU will not proceed until that module becomes ready; see "To Command a Module" in Section VIII.)

Special opcode: 14

Indicators: unaffected

*This is a privileged instruction.*

SIRF Set external Interrupt Reference Flag. This instruction expects a device number to be given in the stack at S−K. SIRF sets the IRF bit for the specified device to "0" and increments the Interrupt Counter by one. (The IRF bit is bit 0 of the fourth word of the Device Reference Table entry for a device. The Interrupt Counter is fixed memory location $7_8$ for processor 1 and fixed memory location $13_8$ for processor 2.) If the IRF bit is already a "0", the Interrupt Counter is not incremented.
Special opcode: 15
Indicators: unaffected
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | | | | |

K

SIN Set interrupt. This instruction expects a device number to be given in the stack at S−K. SIN sets the Interrupt Request flip-flop in the specified device controller and sets the Condition Code to CCE. If the device controller does not respond, the Condition Code is set to CCL and the instruction is terminated.
Special opcode: 16
Indicators: CCE = responding device controller
CCL = non-responding device controller
*This is a privileged instruction.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | | | | |

K

## LOOP CONTROL INSTRUCTIONS

TBA Test and branch, limit in A. This instruction expects the top three elements of the stack to be initialized as follows: A contains a limit, B contains a step size, and C contains a DB+ relative displacement for the address of a variable. TBA tests the variable against the limit. If the limit is not exceeded, control is transferred to the branch address at P ± displacement. If the limit is exceeded, the top three elements of the stack are deleted and execution continues at P + 1.
**Instruction Commentary 17.**
Memory opcode: 05, bits 4,5,6 = 000
Indicators: unaffected
Addressing mode: P relative (+/−)

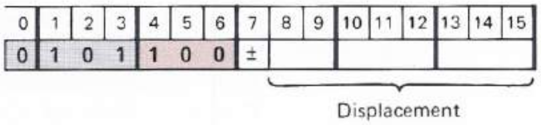
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | ± | | | | | | | | |

Displacement

MTBA Modify variable, test and branch, limit in A. This instruction expects the top three elements of the stack to be initialized as follows: A contains a limit, B contains a modifying step size, and C contains a DB+ relative displacement for the address of a variable. MTBA adds the step size to the variable in integer form, replaces the old variable with this new sum, and tests the new sum against the limit. If the limit is not exceeded, control is transferred to the branch address at P ± displacement. If the limit is exceeded, the top three elements of the stack are deleted and execution continues at P+1.
**Instruction Commentary 17.**
Memory opcode: 05, bits 4,5,6 = 010
Indicators: unaffected
Addressing mode: P relative (+/−)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | ± | | | | | | | | |

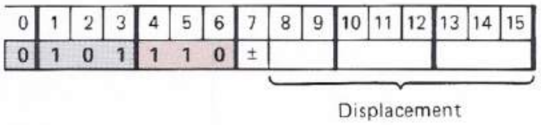
Displacement



TBX Test and branch, variable in X. This instruction requires that the Index register contains the variable and that the top two elements of the stack are initialized as follows: A contains a limit and B contains a step size. TBX tests the variable in X against the limit. If the limit is not exceeded, control is transferred to the branch address at P ± displacement. If the limit is exceeded, the top two elements of the stack are deleted and execution continues at P + 1.

**Instruction Commentary 17.**

Memory opcode: 05, bits 4,5,6 = 100
Indicators: unaffected
Addressing mode: P relative (+/– )

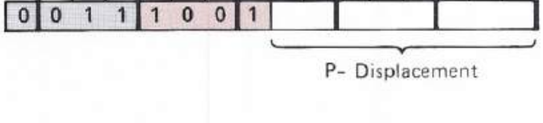


MTBX Modify variable in X, test and branch. This instruction requires that the Index register contains the variable and that the top two elements of the stack are initialized as follows: A contains a limit and B contains a modifying step size. MTBX adds the step size to the variable in integer form, replaces the old Index register contents with this new sum, and tests the new sum against the limit. If the limit is not exceeded, control is transferred to the branch address at P ± displacement. If the limit is exceeded, the top two elements of the stack are deleted and execution continues at P + 1.

**Instruction Commentary 17.**

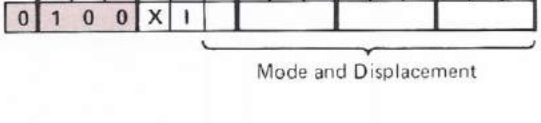
Memory opcode: 05, bits 4,5,6 = 110
Indicators: unaffected
Addressing mode: P relative (+/– )

## MEMORY ADDRESS INSTRUCTIONS



LDPP Load double from program, positive. The double word contained at P+N is pushed onto the stack. An attempt to load from beyond the limit defined by PL will cause a Bounds Violation interrupt to segment 11.

Sub-opcode 3: 10
Indicators: CCA
Addressing mode: P+ relative



LDPN Load double from program, negative. The double word contained at P–N is pushed onto the stack. An attempt to load from beyond the limit defined by PB will cause a Bounds Violation interrupt to segment 11.

Sub-opcode 3: 11
Indicators: CCA
Addressing mode: P– relative

LOAD Load word onto stack. The content of the effective address location is pushed onto the stack. An attempt to load from beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.

Memory opcode: 04
Indicators: CCA
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

STOR Store TOS into memory. The content of the TOS is stored into the effective address memory location, and is then deleted from the stack. In user mode, an attempt to store beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 05, bit 6 = 1
Indicators: unaffected
Addressing modes: DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | X | I | 1 | | | | | | | | | |

Mode and Displacement

CMPM Compare TOS with memory. The Condition Code is set to pattern C as a result of the comparison of the TOS with the content of the effective address location. The TOS is then deleted. An attempt to reference a location beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 06
Indicators: CCC
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | X | I | | | | | | | | | | |

Mode and Displacement

ADDM Add memory to TOS. The content of the effective address memory location is added in integer form to the TOS. The result replaces the operand on the TOS. An attempt to reference a location beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 07
Indicators: CCA, Carry, Overflow
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | X | I | | | | | | | | | | |

Mode and Displacement

SUBM Subtract memory from TOS. The content of the effective address memory location is subtracted in integer form from the TOS. The result replaces the operand on the TOS. An attempt to reference a location beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 10
Indicators: CCA, Carry, Overflow
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | X | I | | | | | | | | | | |

Mode and Displacement

MPYM Multiply TOS by memory. The TOS is multiplied in integer form by the content of the effective address memory location. The least significant word of the result replaces the operand on the TOS. An attempt to reference a location beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 11
Indicators: CCA, Overflow
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | X | I | | | | | | | | | | |

Mode and Displacement

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | X | I | 0 | | | | | | | | | |

Mode and Displacement

INCM Increment memory. The content of the effective address memory location is incremented by one in integer form. In user mode, an attempt to reference a location beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.

Memory opcode: 12, bit 6 = 0
Indicators: CCA, Carry, Overflow
Addressing modes: DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | X | I | 1 | | | | | | | | | |

Mode and Displacement

DECM Decrement memory. The content of the effective address memory location is decremented by one in integer form. In user mode, an attempt to reference a location beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.

Memory opcode: 12, bit 6 = 1
Indicators: CCA, Carry, Overflow
Addressing modes: DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | X | I | | | | | | | | | | |

Mode and Displacement

LDX Load Index. The content of the effective address memory location is loaded into the Index register. An attempt to load from beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.

Memory opcode: 13
Indicators: CCA
Addressing modes: P+, P– , DB+, Q+, Q– , S– relative
Direct or indirect
Indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | X | I | 0 | | | | | | | | | |

Mode and Displacement

LDB Load byte. The content of the effective byte address memory location is loaded into the right half of the TOS. If indirect addressing is used, the word referenced by the initial address (base + displacement) contains a DB+ relative byte address. If indexing is used, the effective byte address is obtained by adding the byte index in the Index register to the relative byte address. In user mode, an attempt to load from beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.

Memory opcode: 15, bit 6 = 0
Indicators: CCB
Addressing modes: Byte addressing
DB+, Q+, Q– , S– relative
Direct or indirect
(for final indirect: DB+ only)
Byte indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | X | I | 1 | | | | | | | | | |

Mode and Displacement

LDD Load double. The contents of the effective address memory location (E) and the succeeding location (E + 1) are pushed onto the stack. The content of E, the most significant word, is loaded into B; the content of E + 1, the least significant word, is loaded into A. If indirect addressing is used, the word referenced by the initial address (base + displacement) contains a DB+ relative word address. If indexing is used, the effective address is obtained by adding the doubleword index in the Index register to the relative

word address. In user mode, an attempt to load from beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 15, bit 6 = 1
Indicators: CCA
Addressing modes: DB+, Q+, Q– , S– relative
Direct or indirect
(for final indirect: DB+ only)
Doubleword indexing available

STB Store byte. The right byte (bits 8 through 15) of the TOS is stored into the effective byte address memory location and the TOS is deleted. If indirect addressing is used, the word referenced by the initial address (base + displacement) contains a DB+ relative byte address. If indexing is used, the effective byte address is obtained by adding the byte index in the Index register to the relative byte address. In user mode, an attempt to store beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 16, bit 6 = 0
Indicators: unaffected
Addressing modes: Byte addressing
DB+, Q+, Q– , S– relative
Direct or indirect
(for final indirect: DB+ only)
Byte indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | X | I | 0 |  |  |  |  |  |  |  |  |  |

Mode and Displacement

STD Store double. The top two words of the stack are stored into the effective address memory location (E) and the succeeding location (E + 1), and are then deleted from the stack. The content of B, the most significant word, is stored into E; the content of A, the least significant word, is stored into E + 1. If indirect addressing is used, the word referenced by the initial address (base + displacement) contains a DB+ relative word address. If indexing is used, the effective address is obtained by adding the doubleword index in the Index register to the relative word address. In user mode, an attempt to store beyond the limits defined by DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 16, bit 6 = 1
Indicators: unaffected
Addressing modes: DB+, Q+, Q– , S– relative
Direct or indirect
(for final indirect: DB+ only)
Doubleword indexing available

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | X | I | 1 |  |  |  |  |  |  |  |  |  |

Mode and Displacement

LRA Load relative address. The effective address is computed, then subtracted from the appropriate base register (PB for P± addressing or DB for DB+, Q±, and S– addressing). The resulting relative address is pushed onto the stack. An attempt to load from beyond the limits defined by PB and PL or (in user mode only) DL and S will cause a Bounds Violation interrupt to segment 11.
Memory opcode: 17
Indicators: unaffected
Addressing modes: P+, P– , DB+, Q+, Q–, S– relative
Direct or indirect
Indexing available

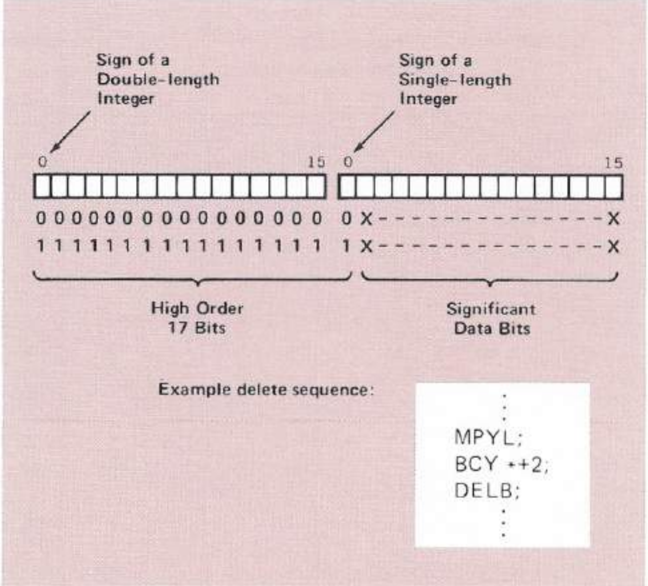
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | X | I |  |  |  |  |  |  |  |  |  |  |

Mode and Displacement

# INSTRUCTION COMMENTARY

**1** MPYL, MPY, DTST, FIXR, FIXT, LMPY. These six instructions provide for the deletion of the most significant word of a doubleword result. The assumption is that the result of the instruction (e.g., multiplication product) does not require more than 16 bits to represent it. The MPY instruction deletes automatically during execution; the remaining five instructions simply test the result and provide an indication (Carry bit) to note whether or not the low order word fully represents the true result. Thus, for these five, the programmer may choose to insert a delete sequence (see figure 5-1) to delete the high order word if it is insignificant.

For MPYL, DTST, FIXR, FIXT, and LMPY, the Carry bit is cleared if the high order 17 bits are all zeros or all ones. This test ensures that the sign bit of the single-length result will be the same as the sign of the double-length result. If


Sign of a Double-length Integer
Sign of a Single-length Integer
0
15
0
15
0000000000000000 0X-------------X
1111111111111111 1X-------------X
High Order 17 Bits
Significant Data Bits

Example delete sequence:

```
    :
MPYL;
BCY *+2;
DELB;
    :
```

Figure 5-1. Deleting a High Order Word

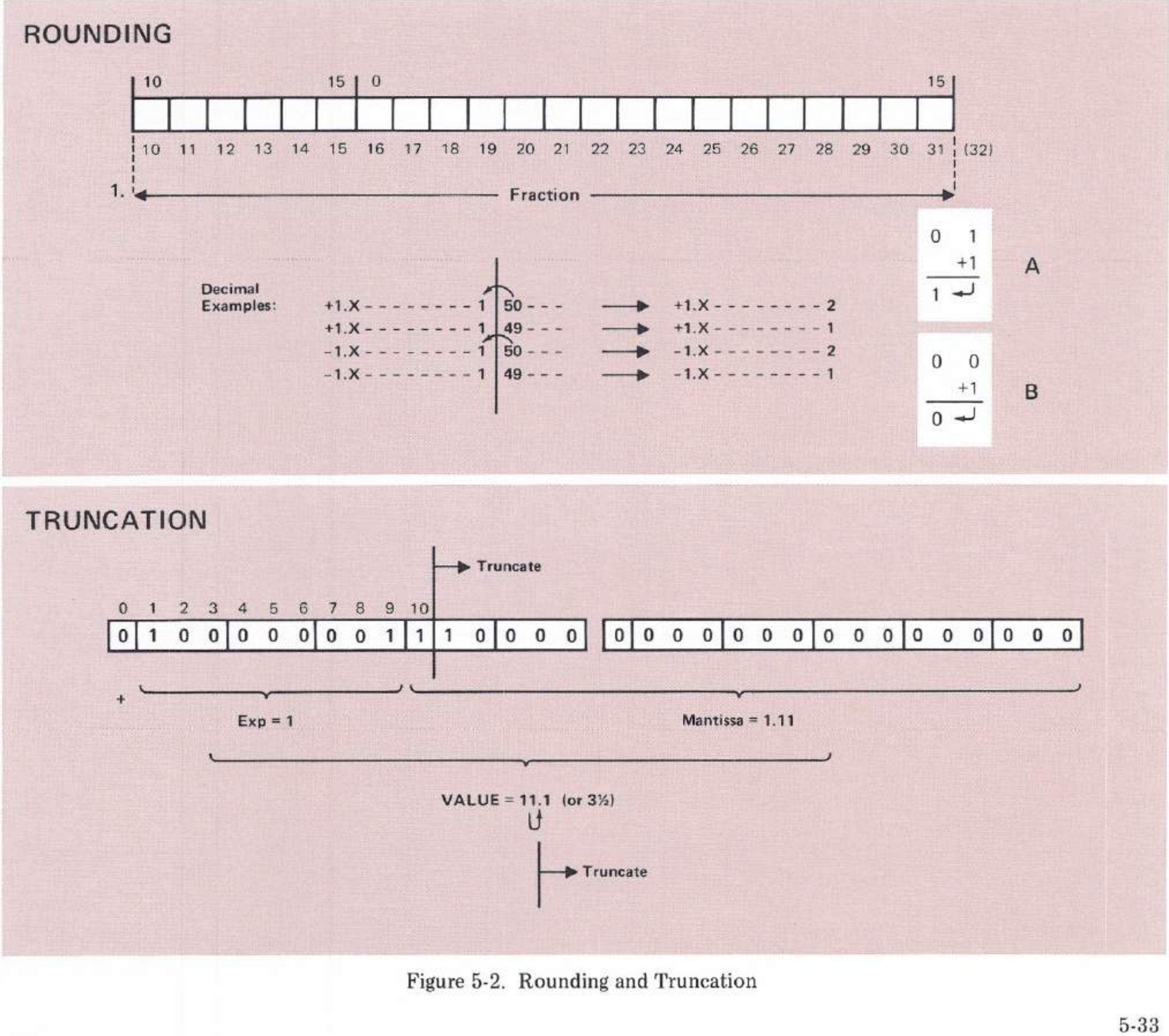
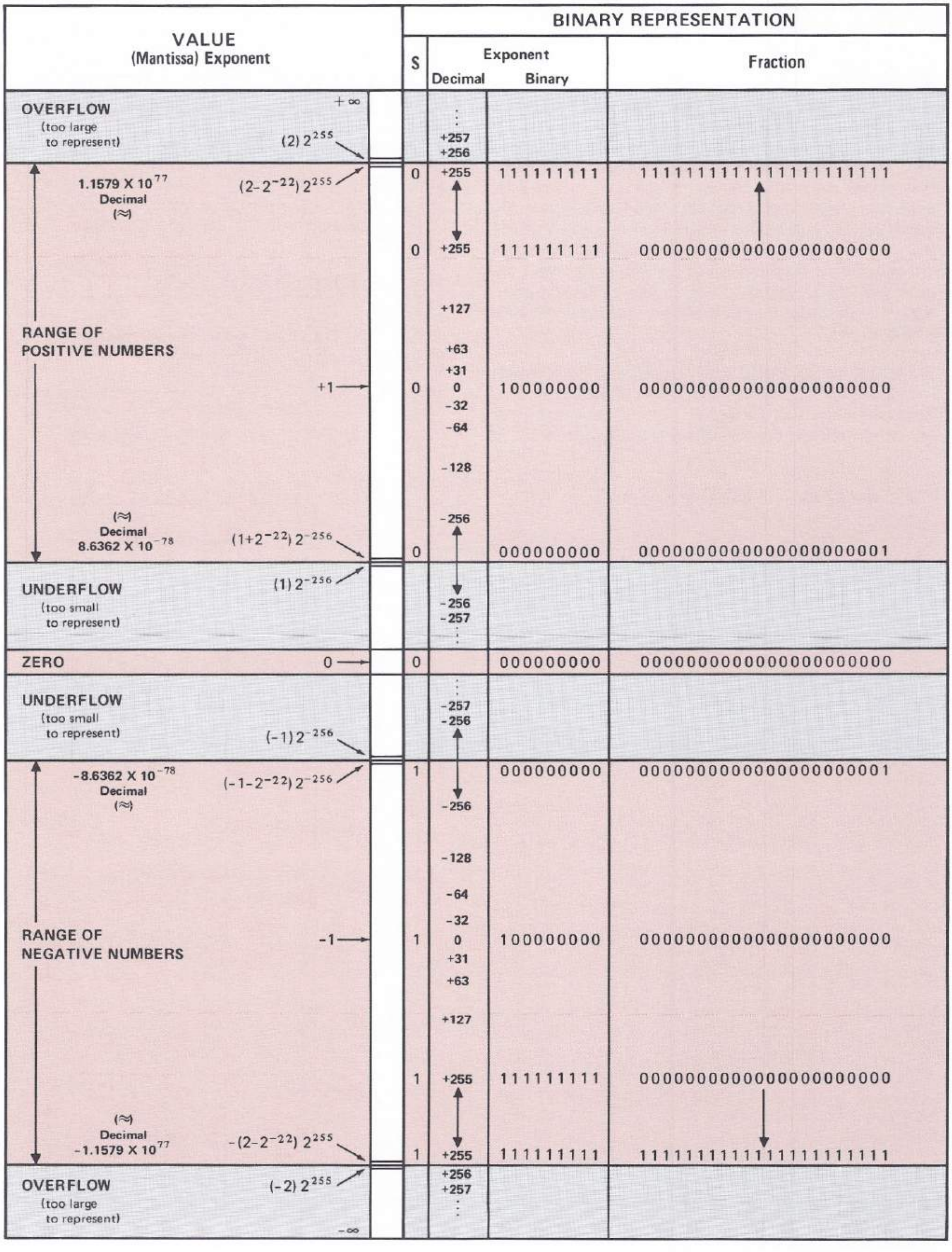


Figure 5-2. Rounding and Truncation



| VALUE<br>(Mantissa) Exponent | | BINARY REPRESENTATION<br>S | Exponent<br>Decimal | Exponent<br>Binary | Fraction |
|---|---|---|---|---|---|
| OVERFLOW<br>(too large<br>to represent) | + ∞<br>$(2)\ 2^{255}$ | | ⋮<br>+257<br>+256 | | |
| $1.1579 \times 10^{77}$<br>Decimal<br>(≈) | $(2-2^{-22})\ 2^{255}$ | 0 | +255 | 111111111 | 1111111111111111111111 |
| | | 0 | +255 | 111111111 | 0000000000000000000000 |
| | | | +127 | | |
| RANGE OF<br>POSITIVE NUMBERS | | | +63<br>+31 | | |
| | +1 | 0 | 0 | 100000000 | 0000000000000000000000 |
| | | | -32<br>-64<br>-128<br>-256 | | |
| (≈)<br>Decimal<br>$8.6362 \times 10^{-78}$ | $(1+2^{-22})\ 2^{-256}$ | 0 | | 000000000 | 0000000000000000000001 |
| UNDERFLOW<br>(too small<br>to represent) | $(1)\ 2^{-256}$ | | -256<br>-257<br>⋮ | | |
| ZERO | 0 | 0 | | 000000000 | 0000000000000000000000 |
| UNDERFLOW<br>(too small<br>to represent) | $(-1)\ 2^{-256}$ | | ⋮<br>-257<br>-256 | | |
| $-8.6362 \times 10^{-78}$<br>Decimal<br>(≈) | $(-1-2^{-22})\ 2^{-256}$ | 1 | | 000000000 | 0000000000000000000001 |
| | | | -256<br>-128<br>-64<br>-32 | | |
| RANGE OF<br>NEGATIVE NUMBERS | -1 | 1 | 0 | 100000000 | 0000000000000000000000 |
| | | | +31<br>+63<br>+127 | | |
| | | 1 | +255 | 111111111 | 0000000000000000000000 |
| (≈)<br>Decimal<br>$-1.1579 \times 10^{77}$ | $-(2-2^{-22})\ 2^{255}$ | 1 | +255 | 111111111 | 1111111111111111111111 |
| OVERFLOW<br>(too large<br>to represent) | $(-2)\ 2^{255}$<br>$-\infty$ | | +256<br>+257<br>⋮ | | |

Figure 5-3. Ranges of Floating Point Numbers

this is not the case, Carry is set, and the most significant word should not be deleted. For MPY, Overflow will be set if the test fails, meaning that MPYL should have been used instead of MPY.

**2** DFLT, FLT, FADD, FSUB, FMPY, FDIV, FIXR, FIXT. These eight floating point instructions use rounding or truncation in computing a final result and except for DFLT and FLT, are subject to both overflow and underflow. The following paragraphs explain these conditions as they apply to the HP 3000.

Rounding and Truncation. Figure 5-2 illustrates both rounding and truncation. Rounding is a simple matter of adding a "1" to whatever is in bit position 32. If bit 32 is a "1" (case A in the figure), adding "1" will cause a carry into bit 31, thus incrementing the representable value. If bit 32 is a "0" (case B), adding "1" will not cause a carry, and the representable value is unchanged.

Truncation is used only by the FIXT instruction and consists of discarding all fractional bits after computing the effective binary point position. This is shown in the lower part of figure 5-2, which illustrates the case of truncating the decimal number 3.5 to 3. The biased exponent (octal 401) represents an exponent of 1. The fraction, as stored, is .11 which, when combined with the assumed leading 1 gives a resultant mantissa of 1.11. The positive exponent of 1 implies that the effective binary point position is one place to the right. Thus the true binary value represented is 11.1, which is 3.5 in decimal. Therefore, in this case, truncation of the fraction consists of discarding all low order bits from 11 through 31.

Overflow and Underflow. Figure 5-3 illustrates overflow and underflow for floating point instructions. Overflow is caused by these instructions when the computed result (either positive or negative) is too large to be represented. Underflow is caused when the computed result is too small to be represented. The limits are defined in figure 5-3.

When user traps are enabled, an overflow or underflow trap will occur to indicate which type of error resulted. If the traps are not enabled, the Overflow bit will be set on either type of error.

It is possible to reconstruct correct answers from overflow or underflow results. If the exponent and fraction are both zero and there is an underflow, the result should be taken as +/- (depending on sign bit) $2^{-256}$. In all other cases, test bit 1 (most significant bit of exponent). If this bit is 0, add 512 (decimal) to the exponent; if it is 1, subtract 512 from the exponent.

**3** ASL, ASR, LSL, LSR, CSL, CSR. The actions of the six single word shift instructions are shown in figure 5-4. It is assumed that the shift count, specified in the argument

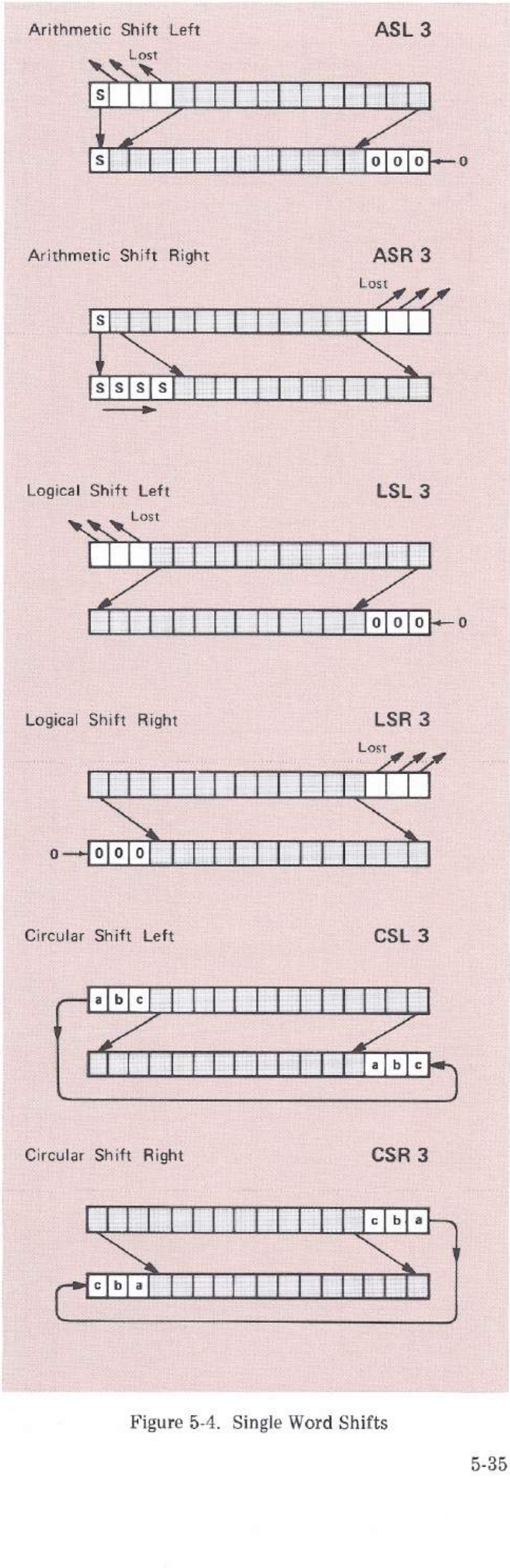


Figure 5-4. Single Word Shifts

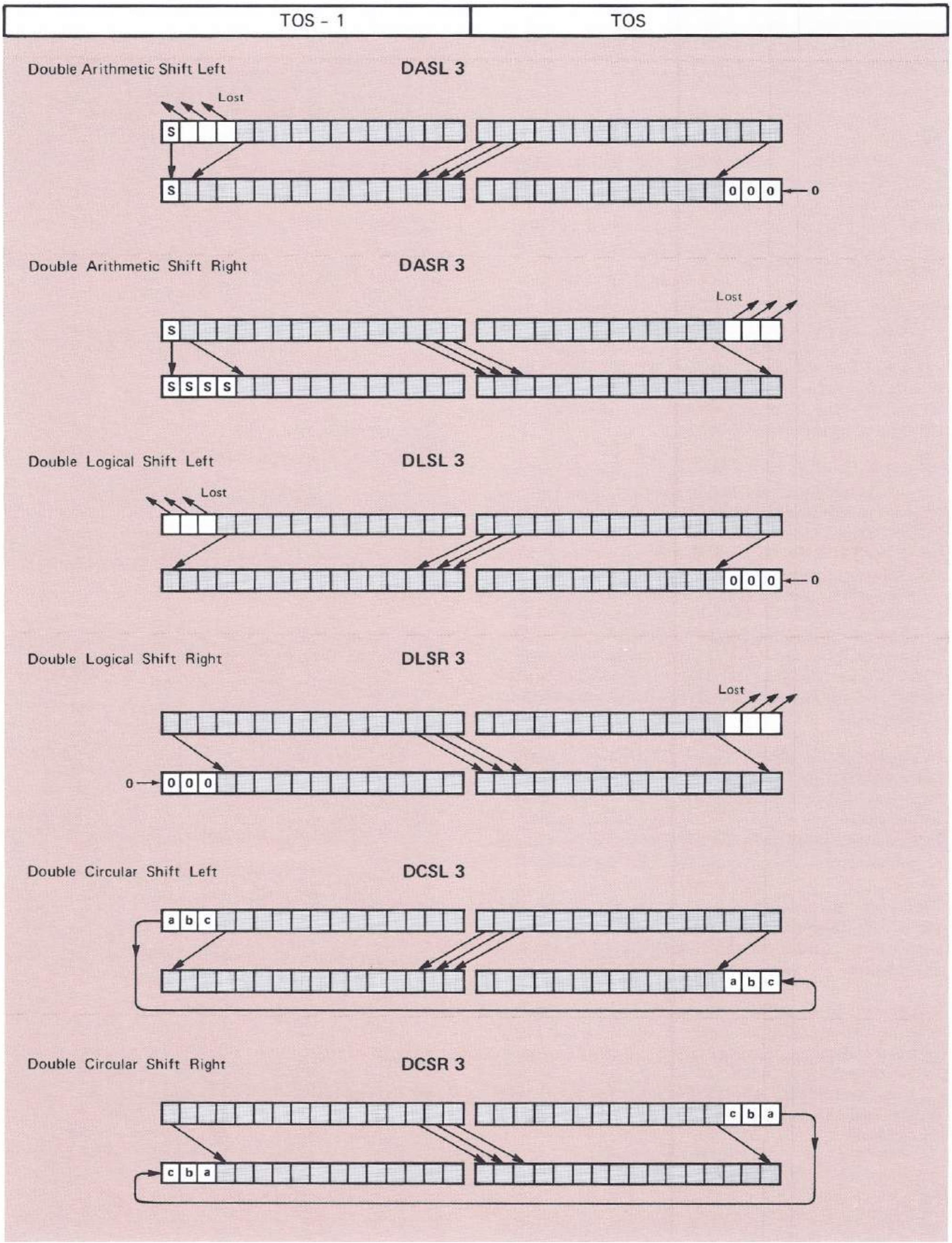

TOS - 1
TOS
Double Arithmetic Shift Left
DASL 3
Lost
S
S
0 0 0
0
Double Arithmetic Shift Right
DASR 3
Lost
S
S S S S
Double Logical Shift Left
DLSL 3
Lost
0 0 0
0
Double Logical Shift Right
DLSR 3
Lost
0
0 0 0
Double Circular Shift Left
DCSL 3
a b c
a b c
Double Circular Shift Right
DCSR 3
c b a
c b a

Figure 5-5. Double Word Shifts

field of the instruction, is 3 in each case. The before and after conditions of the TOS word are shown for each example.

In the case of arithmetic shifts, the sign bit is always preserved. When shifting left, the bits shifted out of bit 1 (most significant bit next to the sign bit) are lost; zeros are filled into the vacated low order bit positions. When shifting right, the sign bit is copied into the vacated high order bit positions, and bits shifted out of bit 15 (least significant bit) are lost.

In the case of logical shifts, all bits are shifted. Bits are lost out of the high end when shifting left and out of the low end when shifting right. Zeros are filled into the vacated bit positions.

In the case of circular shifts, no bits are lost. Bits shifted out of the high end when shifting left are filled into the vacated low order bit positions. When shifting right, bits shifted out of the low end are filled into the vacated high order bit positions.

Note that, for all shift instructions, the number of shifts is determined either by the value specified in the argument field of the instruction or, if X is specified ("1" in bit 4), by adding the argument field value to the Index register contents. This permits the number of shifts to be computed as well as explicitly specified.

All shift instructions except TNSL use the shift count in a modulo 64 manner. Thus if the final shift count is 100 octal (64 decimal), the data is not shifted at all. Furthermore, if the number of shifts equals or exceeds the number of magnitude bits (whether single, double, or triple word), the following will occur: for left arithmetic shifts and all logical shifts, the magnitude will be all-zero; for right arithmetic shifts, all magnitude bits will be the same as the sign bit; for circular shifts, the circular shifting will continue until the specified number of shifts (up to 63) have been achieved.

Except for TNSL (see Instruction Commentary 5) the execution of shift instructions does not alter the content of the Index register.

**4** DASL, DASR, DLSL, DLSR, DCSL, DCSR. The actions of the six double word shift instructions are shown in figure 5-5. The shift count, specified in the argument field of the instruction, is assumed to be 3 in each case. The before and after conditions of the two top words of the stack are given in each example. The TOS contains the least significant half of double word integers, and the second word (B, or TOS-1) contains the most significant half.

Double word arithmetic, logical, and circular shifts are the same as the corresponding single word shifts described above under Instruction Commentary 3 except for the word length. This means that, when shifting left, bits shifted out of the high end of the low order word are filled into the low end of the high order word. When shifting right, bits shifted out of the low end of the high order word are filled into the high end of the low order word. Similarly, on circular shifts, bits shifted out of one end of the double word are filled into the opposite end of the double word.

**5** TASL, TASR, TNSL. Figure 5-6 illustrates the actions of the three triple word shift instructions. Two of these, the arithmetic shifts, are the same as the single and double word shift instructions previously described in Instruction Commentaries 3 and 4, except that three words are shifted. The TOS contains the least significant word, B (or TOS-1) contains the middle word, and C (or TOS-2) contains the most significant word.

The TNSL (Triple Normalizing Shift Left) instruction is a special case. Instead of specifying a shift count, TNSL shifts left arithmetically until a "1" is shifted into bit 6 of the most significant word, and the number of shifts is counted in the Index register. The argument field is ignored. Bits 0 through 5 of the most significant word are cleared.

The TNSL instruction clears the Index register before beginning to shift unless X is specified in bit 4 of the instruction. If X is specified, the shift count adds on to the existing contents of the Index register. If bit 6 of C and all lower order bits are zero, a "1" cannot be shifted into bit 6 of C. TNSL initially tests for this condition and, if true, bypasses the shift operations and simply puts 42 into (or adds 42 to) the Index register. This is the value that would exist if the shifts were actually executed.

The purpose of the TNSL instruction is to normalize a triple word floating point number. Such a number has a 42-bit mantissa consisting of: a leading "1", 38 representable fraction bits, a rounding bit, and two guard bits at the least significant end. TNSL assumes that the number has previously been left-shifted three places in order to include the rounding and guard bits in the least significant word. Thus the leading "1", instead of being assumed to exist in the bit 9 position of C (see figure 5-6) is now moved to the bit 6 position.

**6** BR. The P relative mode of BR, the unconditional branch instruction, is a conventional P relative branch except for the indexing capability and the extended displacement range. Bits 8 through 15 are available to specify displacement, which therefore can be up to ±255.

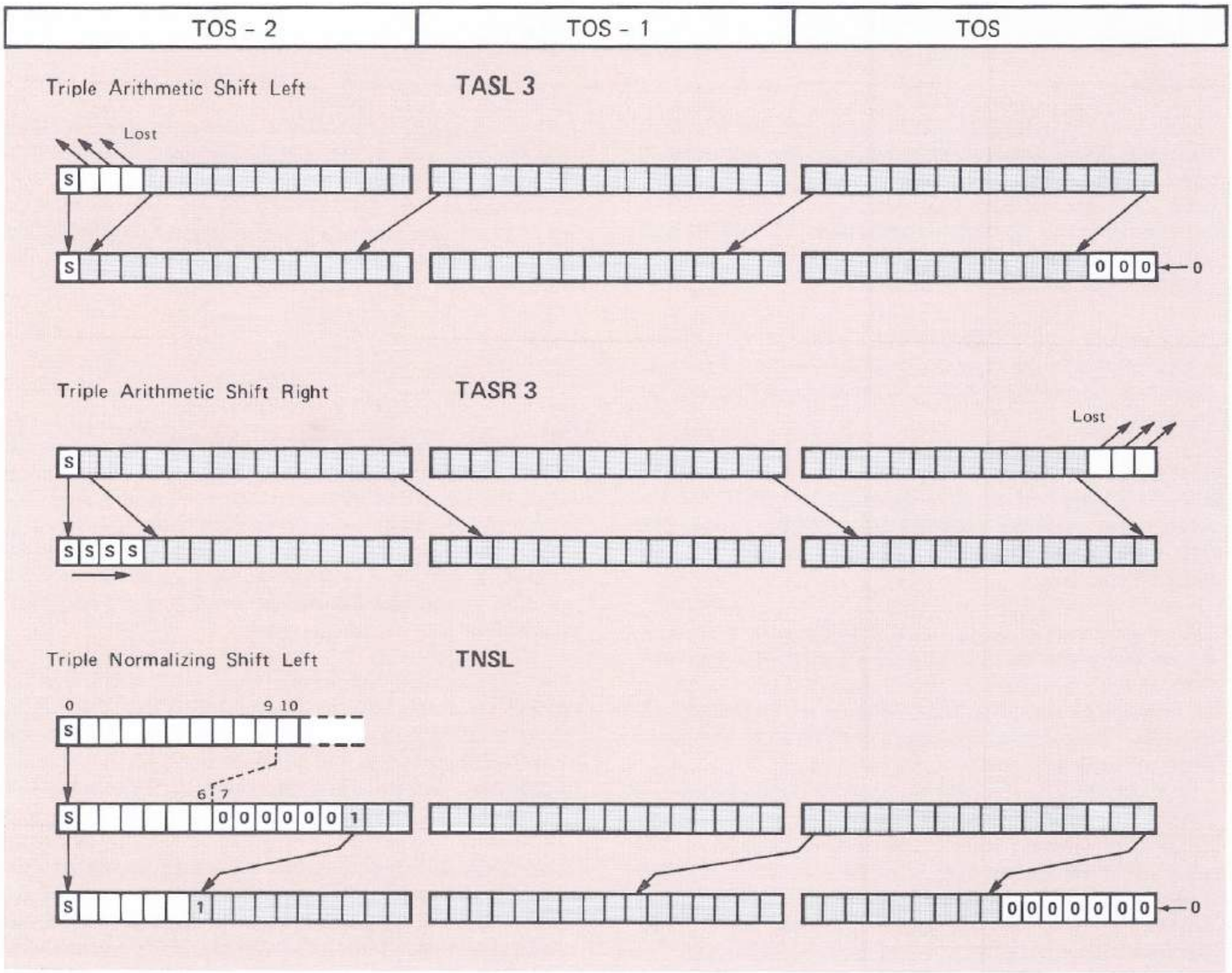


Figure 5-6. Triple Word Shifts

The DB, Q, and S relative modes, however, are unconventional in that they permit indirect branches through the data stack. (It is both illegal and impossible to have a direct branch to the stack; the coding of "01" for bits 5 and 6 encodes the BCC instruction.)

Figure 5-7 shows an example of the S- relative mode. Assume that the instruction in location P specifies the S-relative mode, with a displacement of 4, and indexing. This causes an indirect branch to S-4 in the data stack. The content of S-4 is then added to PB, thus pointing at location "a" in the code segment. Since indexing is specified, the value contained in the Index register is also added to the address being computed. Thus the ultimate effective address for the branch (next P) is location "a" displaced by the index value.

Note particularly that the indirect address given in the stack is relative to the program base, PB, not to P as is usually the case. Also note that the displacement is relative to a location in the stack (DB, Q, or S), and that indexing is applied after the indirect addressing has been accomplished.

The displacement range for the DB, Q, and S modes depends on which mode is selected. For DB+, bits 8 through 15 provide a range of 0 through +255. For Q+, bits 9 through 15 provide a range of 0 through +127. For Q- and S-, bits 10 through 15 provide a range of 0 through -63.

**7** TSBM. This instruction is primarily intended to reference a software lock word. Typical application would be in multiprogramming systems. When one process attempts to use a given critical portion of code, it will set a certain

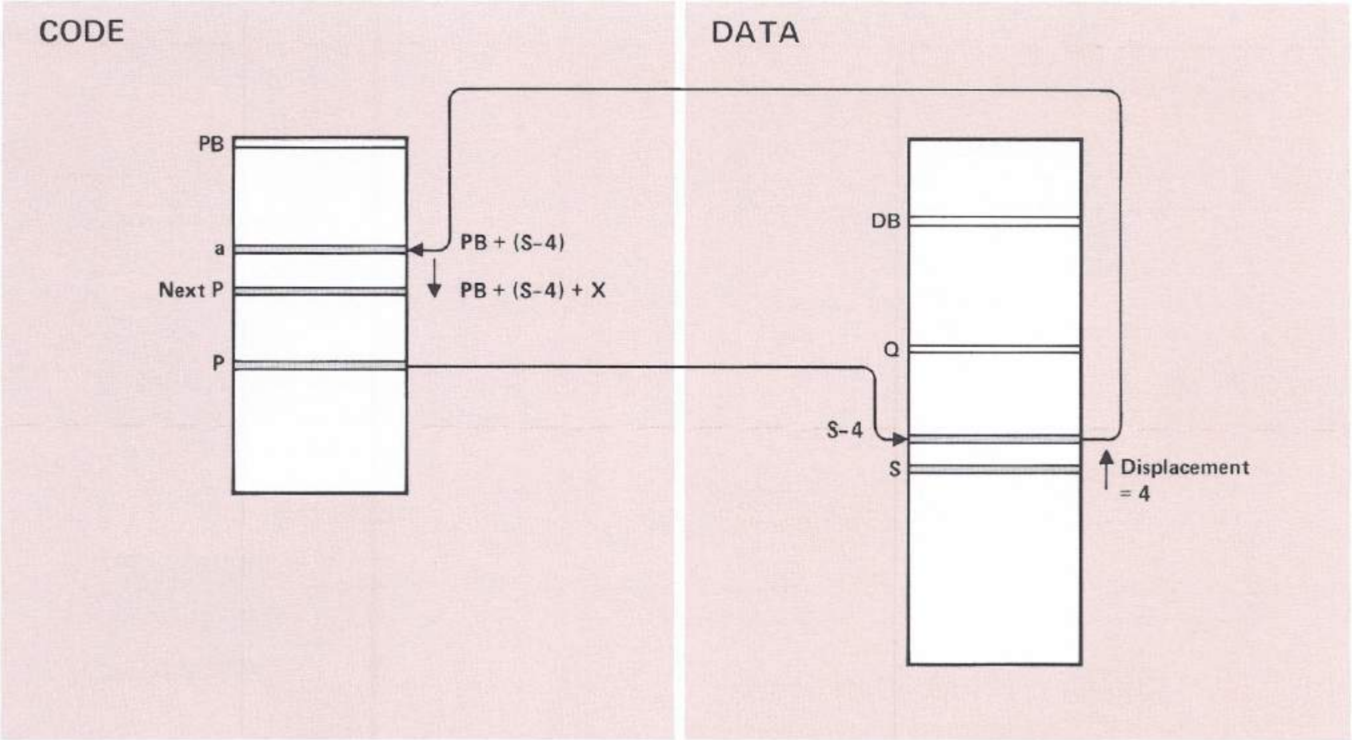


Figure 5-7. Indirect Branch via Stack

combination of bits in the lock word to "1", using the OR function of the TSBM instruction. Since there is the possibility that another process may already be using the code, the TSBM instruction also tests to see if any of the bits it has set to "1" already were in the "1" state. This is accomplished by the AND function. If any bits in the TOS word are "1" following the execution of TSBM, which can be checked by testing the Condition Code, the indication is that the code is currently in use. On completion of the routine or subroutine, the appropriate bits of the lock word would be cleared.

To preserve the lock mechanism in a dual-processor system (where the second processor could read the lock word in memory while the first processor is in the midst of changing its copy of the word for re-storage), the hardware automatically sets the memory cell to all "1"s until the first processor re-stores the modified lock word. Thus, no matter which bits the second processor tests, it can only assume that the code is currently in use.

**8** MOVE, MVB, MVBW, CMPB. These four instructions are members of the move group, and as such deal with strings of words or bytes. The first three physically move a word or byte string from one block of locations in primary memory to another. The CMPB instruction does not move data, but compares the data in two complete strings, byte by byte. The following paragraphs explain and compare the significant features of all four instructions. Refer to figure 5-8.

SOURCES. The MOVE, MVB, and CMPB instructions may take source data from either the code segment or the data segment. (For reference purposes, "source" and "target" terminology is retained for CMPB, even though there is no move operation.) If bit 11 of the instruction is a "0", source addresses are PB+ relative — i.e., from the code segment. If bit 11 is a "1", source addresses are DB+ relative — i.e., from the data segment. Figure 5-8 illustrates both cases. Note that the target for either case is in the DB+ area. (Disregard move-direction arrows for CMPB.) The MVBW instruction, however, may not use the PB relative source; sources for MVBW are DB relative only. The target need not be "higher" than the source; figure 5-8 shows examples only.

ASCENDING/DESCENDING ADDRESSES. The MOVE, MVB, and CMPB instructions have the capability of generating ascending or descending addresses for source and target locations. The direction is established by the sign of the count word, which is bit 0 of A, as shown in figure 5-8. If this bit is a "0", the sign is "+", and successive addresses are ascending (B and C incremented). If this bit is a "1" the sign is "–", and successive addresses are descending (B and C decremented). Note the +Count and -Count arrows in figure 5-8. The MVBW instruction uses only ascending addresses; this instruction does not use a count word, and the source and target words are given in A and B instead of B and C.

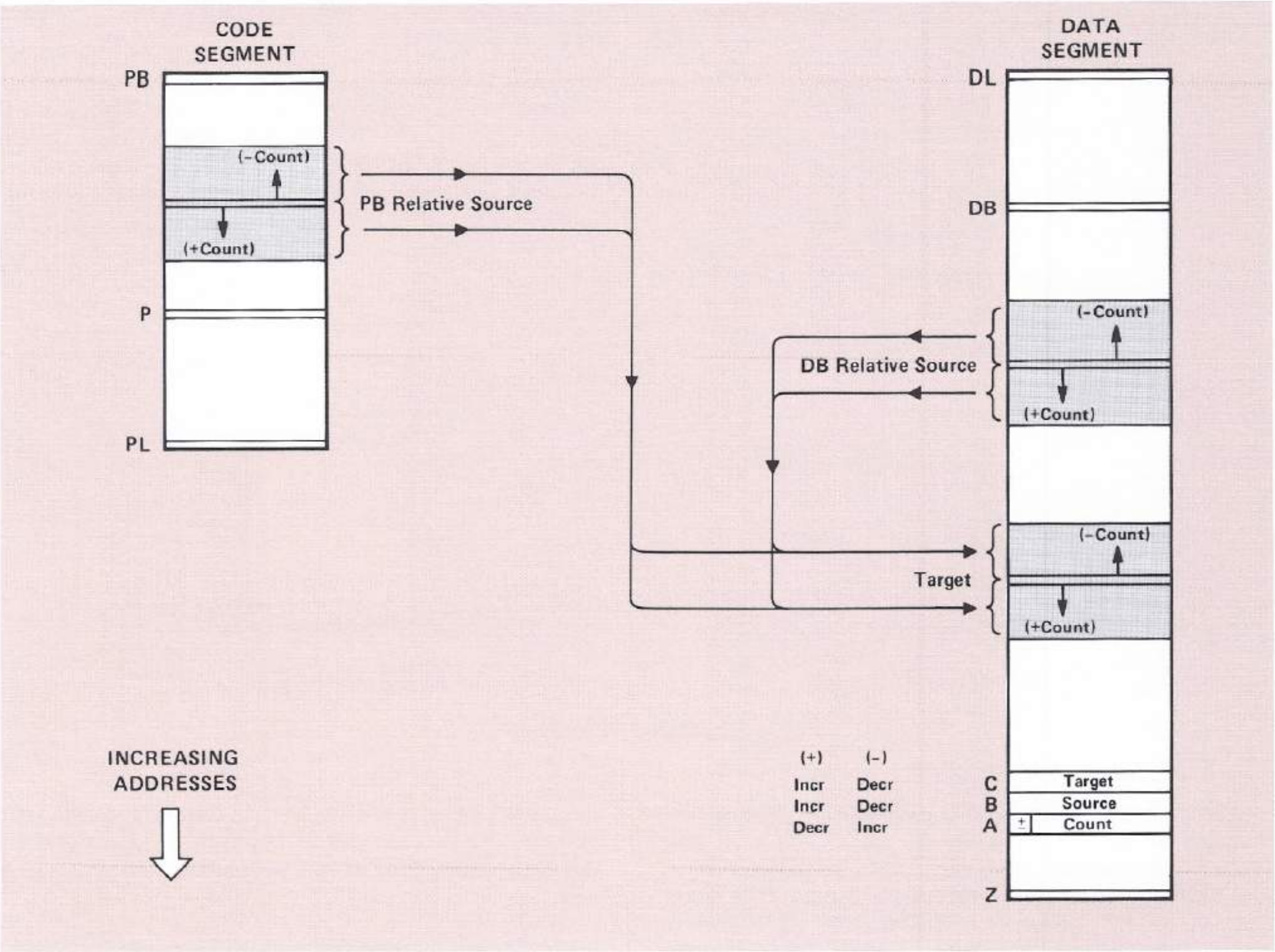


Figure 5-8. Examples of Moves

METHOD OF TERMINATION. The MOVE and MVB instructions are terminated only when the word or byte count becomes zero. The MVBW instruction is terminated only when a character of a specified type, either alphabetic or numeric, is encountered. The CMPB instruction has two methods of termination: when the byte count becomes zero, or when any two bytes being compared are unequal.

SPECIAL FEATURES. The MVBW instruction includes an "upshift" bit (bit 13). This bit, when set ("1"), will transpose any lower case source characters to upper case during the transfer. If not set ("0"), the source characters are unaltered by the instruction.

MOVES BEYOND TOS. In the event that the source or target of any move instruction advances into the top-of-stack area (A, B, C, D) or beyond, the count, source, and target words contained in A, B, and C will not be affected since these values are contained in top-of-stack registers. The memory locations directly corresponding to these registers will be used for the move (or comparison). The move instructions, incidentally, are the only ones which in any way distinguish between top-of-stack in memory and top-of-stack in the CPU. However, situations which encounter this distinction will be rare, since the area between TOS and Z is undefined, thus indicating a probable software error.

INTERRUPTS. All Move instructions are interruptable and will continue their operation after return from the interrupt. To do this, the count, source, and target addresses are kept updated in the TOS registers, which are pushed into memory upon interrupt.

**9** MVBL, MVLB. These two instructions have many characteristics of the other move instructions described above (Instruction Commentary 8). However, since they move data into or out of the data area between DL and DB, MVBL and MVLB are privileged instructions. The following paragraphs summarize the actions of these two instructions. Refer to figure 5-9.

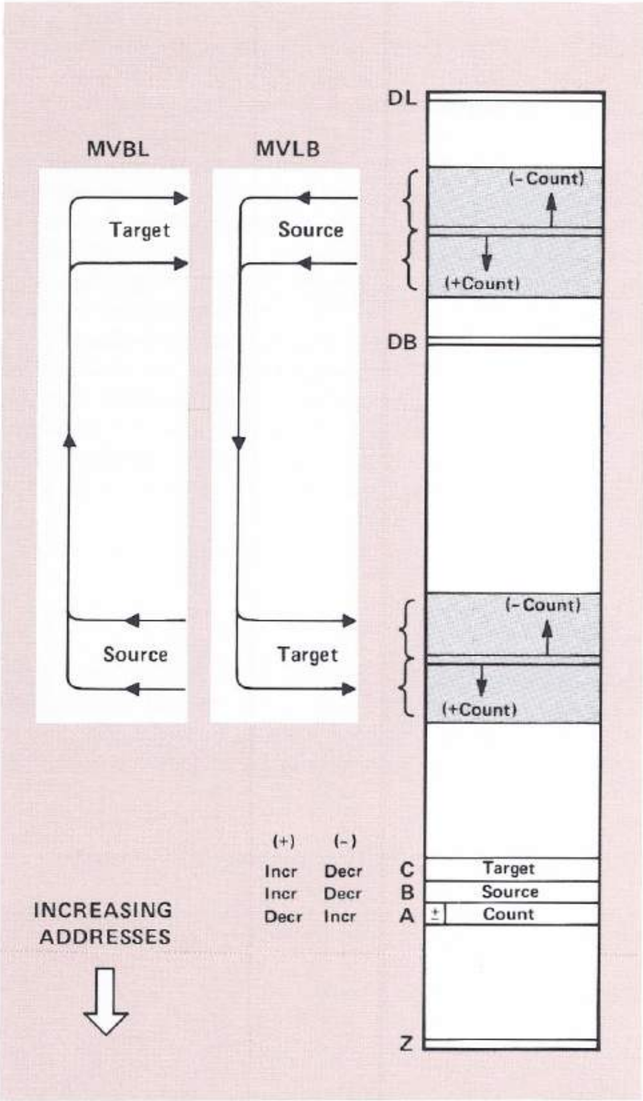


Figure 5-9. Examples of MVBL, MVLB

For MVBL, source data is taken from the DB+ area and the target is in the DL+ area. (A large enough displacement could put the target in the DB+ area.) For MVLB, source data is taken from the DL+ area and the target is in the DB+ area. Addresses for both instructions can be ascending or descending, depending on the state of the count sign. If this bit is a "0", the sign is "+", and successive addresses are ascending (B and C incremented). If this bit is a "1", the sign is "-", and successive addresses are descending (B and C decremented).

Both MVBL and MVLB are terminated when the word count becomes zero. The comment on "Moves Beyond TOS" under Instruction Commentary 8 also applies to these two instructions.

**10** LLSH. A typical application of the LLSH instruction was given in Section IV (see "Segments in Memory"). Basically, the intent in that case was to find a segment of primary memory at least as large as the segment size specified by the test word. Since the software knows how many links exist, it can load this value into the Index register for counting purposes. (Note that the list may have a terminator word consisting of all ones.)

Figure 5-10 illustrates the basic operation of the LLSH instruction. As shown, the top-of-stack (A) contains the link pointer. At all times, in successive fashion, this location contains the absolute address of the link word in the

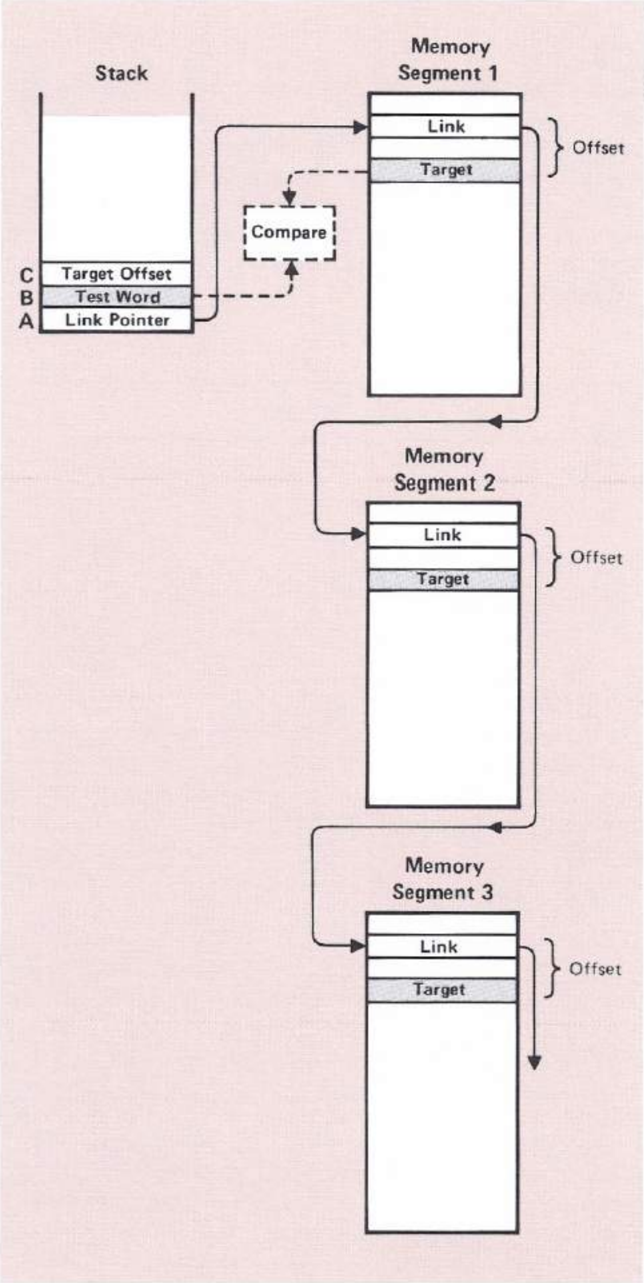


Figure 5-10. LLSH Operation

segment currently being tested. Location B in the stack is the test word, which would typically be a 16-bit number indicating the size of the segment which is to be loaded by the software. Location C is an offset indicating how far the target word is from the link word. Thus as shown, the comparison is between the test word and each target word.

On termination of the instruction, location A of the stack contains the absolute address of the searched-for segment, and a Condition Code of CCE indicates that the search was successful. If the search is not successful, Condition Code CCL or CCG will indicate the cause of termination.

**11** LLBL. The LLBL instruction will convert a local label to external type if it is not already of this type. The conversion is accomplished by forcing bit 0 of the TOS to the "1" state, loading bits 1 through 7 with the value of N (which is the STT entry number), and loading bits 8 through 15 with the corresponding bits of the Status register (i.e., the number of the currently executing code segment).

**12** EXF, DPF. Figure 5-11 compares the operations of EXF and DPF. In the case of EXF, only the TOS word is affected. Assuming values of 2 for J and 8 for K, bits 2 through 9 will be extracted and moved to bits 8 through 15 (i.e., right-justified). Bits 0 through 7, in this example, are filled with zeros. In the case of DPF, the two top words of the stack are affected. The second word of the stack (S-1) is assumed to contain a word that is arbitrarily represented here by the letters "a" through "p". Assuming values of 4 for J and 6 for K, the six least significant bits of the TOS word are deposited into the second word, beginning at bit 4 and ending at bit 9. The remaining bits of the second word are unchanged, and the combined result becomes the new TOS. Note that since the J and K fields each have four bits, they may specify values from 0 through 15 (decimal). The field may wrap around the end of the word; i.e., bit 15 is one bit to the left of bit 0.

**13** ADDS, SUBS. The reason for the "minus one" when using the TOS content to modify S is to delete the modifying parameter. A typical application of the ADDS instruction is to reserve a block of stack locations for procedure

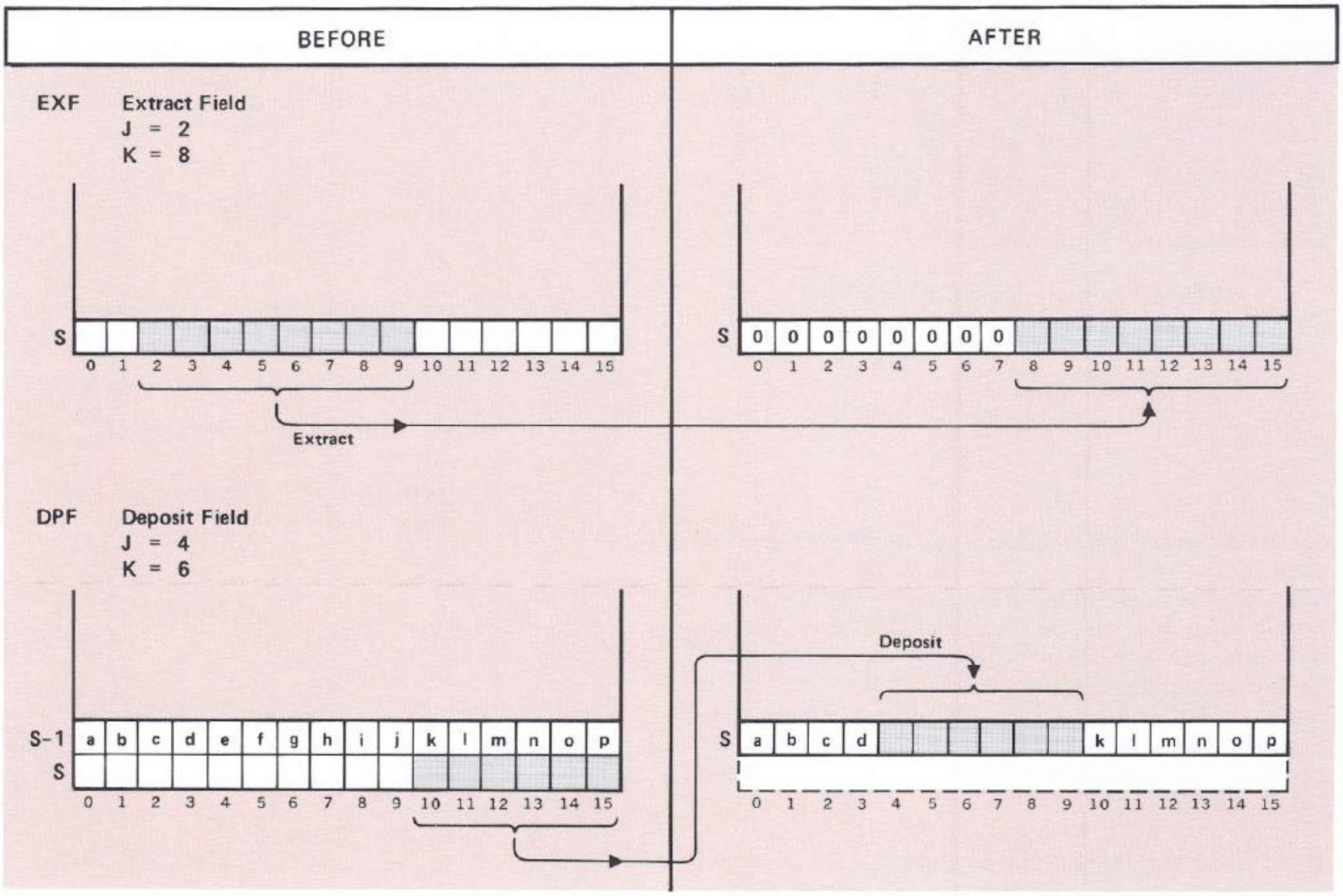


Figure 5-11. EXF and DPF Operation

variables. The number of locations so reserved may be either explicitly given in the instruction's operand field, or computed and accessed via the TOS. The effect of the instruction is simply to advance the top-of-stack pointer a given number of locations without specifying any contents. The SUBS instruction, conversely, deletes a specified number of stack locations.

**14** XEQ. The reason why the use of a second stack opcode (bits 10 through 15) is illegal is that there is no guarantee that it will be executed. If there should be an interrupt between the execution of the two stack operations, the program counter (P) will move on to the next instruction past XEQ. There is no provision to decrement P in order to go back to XEQ for the second stack operation. However, if no intervening interrupt does occur (for example, if the interrupt system is off), both stack opcodes can be executed. Also, possibly, a simple test can be programmed to check for an intervening interrupt. The indicators would be set according to the last executed stack operation.

**15** SCAL, SXIT. Figure 5-12 illustrates the operations for calling and exiting from a subroutine. Since only local labels may be used, operation is entirely within the current code segment. Assume that the system is executing instructions in the code segment shown in figure 5-12. At some point, P will encounter the "SCAL N" instruction, where N is some value 0 through 255. If the value of N is not 0, e.g., 8, this value will be subtracted from PL (i.e., PL-8), thus pointing at the ninth cell counting backward from PL. This must be within the Segment Transfer Table, whose first entry is PL-1. The eighth entry, in this case, contains a local program label (bit 0 = 0), which is a PB relative address pointing to the start of the subroutine. This address is converted to absolute (add to PB) and is loaded into the P-register, while the former value of P, plus one, is stored in the TOS as the return address. However, if N were 0, it would be assumed that the TOS contains the local label (subroutine starting address). This address, then, (made absolute) would be loaded into the P-register, while the former value of P, plus one, replaces the label on the TOS as the return address. In either case, once the P-register has its new address, the location so indicated will be fetched and subroutine execution begins.

The final instruction of the subroutine is SXIT. At this time the return address, pushed onto the stack by SCAL, is assumed to be on the top of the stack. It is the responsibility of the subroutine to provide this condition, which normally means deleting all variables incurred by the subroutine. The SXIT instruction simply takes the address contained in the TOS and puts it in the P-register, thus effecting a return to the calling routine. As a final step, SXIT deletes the TOS, since the return address is no longer needed, and may additionally move S back some number of locations specified by N. This would typically be used for deleting some of the parameters passed to the subroutine.

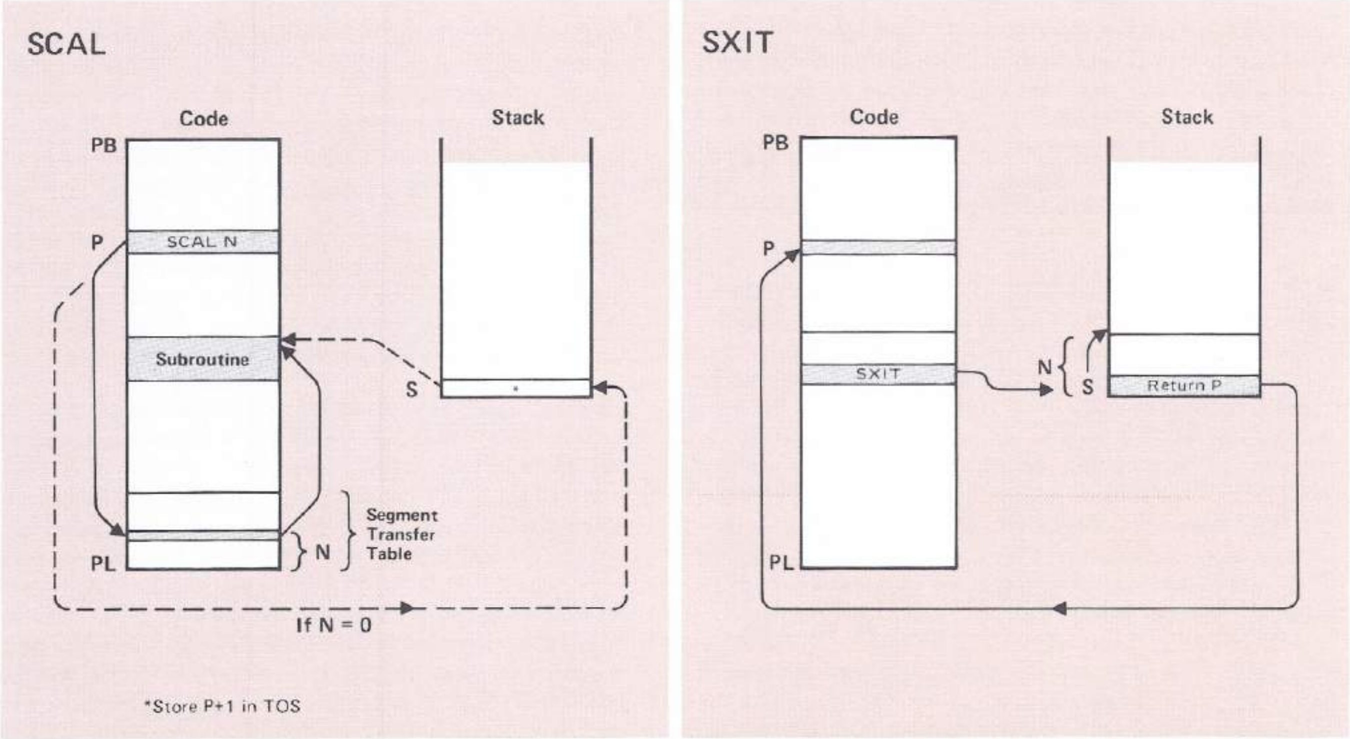


Figure 5-12. Subroutine Call and Exit

**16** PCAL, EXIT. These two instructions perform basically the same function as the SCAL and SXIT instructions described above (Instruction Commentary 15). That is, to call a routine and return from it to the point where it was called. However, since the routines in the case of PCAL/EXIT may be external to the current segment, possibly not even present in main memory, the operation is somewhat more complex. Furthermore, EXIT also has the capability of providing a return from various kinds of interrupt routines (not called by PCAL).

It would be redundant to explain here the mechanics of procedure calls and exits, since a detailed discussion was given earlier in Section IV (see "Code Segments" and "Data Segments"). If the mechanics are not thoroughly understood, read that section again, particularly with reference to figures 4-5 and 4-10. For interrupt concepts used by the EXIT instruction, refer also to Section VII, Interrupt System.

The following paragraphs describe the operations of PCAL and EXIT on a step-by-step basis, referring to flowcharts. It will frequently be assumed that the reader has a working knowledge of the intents and purposes of the various steps.

PCAL Sequence. Figure 5-13 illustrates the operations of the PCAL instruction. If the call is within the current segment (local label), only the steps shown on the left side of the diagram are performed. For calls outside the current segment, the steps on the right side are added.

The first step is to fetch the program label. From the PCAL instruction definition, we see that the label can be obtained from one of two places: from the TOS if N is zero, or from PL–N if N is not zero. This operation can be seen in the SCAL operation of figure 5-12, where the label is fetched from either the Segment Transfer Table, at PL–N, or from the TOS. The only difference is that PCAL puts the fetched label into temporary storage in the CPU, instead of directly into the P-register.

Thus, referring to figure 5-13, PCAL initially checks N to see if the label is on the TOS. If not (block 1), the label is fetched from PL–N and a check is made to see if that location is actually within the bounds of the Segment Transfer table. (N must be ⩽ STTL value in the PL location.) If out of STT bounds, an STT violation interrupt to segment 13 is incurred; otherwise, the PCAL sequence continues. If the label is on the TOS (block 2), the label is put into temporary storage in the CPU and S is decremented to delete the label from the stack. At this time, the CPU has the label but does not know whether it is local or external, or if it is valid.

The next step is to place a standard four-word stack marker onto the stack (block 3) and update the Q pointer by loading it with the content of S (block 4). Both Q and S are now pointing at the last word (delta Q) of the new stack marker.

Now the label is checked to see if it is a local label (bit 0 = 0). If it is, the sequence goes directly to block 11 (skip next seven paragraphs).

If the label is external (bit 0 = 1), bits 8 through 15 are checked to see if the segment number specified is valid. If the segment number does not have an entry in the Code Segment Table (two times segment number must be ⩽ CST Length, in first location of CST), or if the segment number specified is 0 (segment 0 is callable only by external interrupts), a CST violation interrupt to segment 13 is incurred. Otherwise, the PCAL sequence continues.

Next, absolute addresses for PB and PL are calculated from the CST entry and loaded into these two registers (block 5). The CST entry is fetched from CSTP + 2* segment number (bits 8 through 15 of the label). The second word of the two-word CST entry is an absolute address for PB (could be a secondary memory address if the segment is absent from primary memory). The first word contains the length (÷ 4) of the called segment in bits 4 through 15. The value for PL is calculated by adding PB + 4* length –1. The P-register is initially set equal to PB at this time; as explained later, execution may begin at this value of P.

Block 6 sets the privileged mode bit in the Status register if the mode bit in the CST entry indicates privileged mode, or if the caller was executing in privileged mode (i.e., if the privileged mode bit in Status already was set). (Although not shown, the Reference bit in the CST is set at this time, for statistical purposes.)

Block 7 stores bits 8 through 15 of the label into bits 8 through 15 of the Status register. This indicates to the system that we are now operating in the called segment.

A check is then made to see if the called segment is absent, by checking bit 0 of the first word of the CST entry. If it is absent, a four-word stack marker is pushed onto the stack, then the label; external interrupts are disabled; PB, PL, and P are established from the CST entry for segment 14, and Status is updated accordingly. This starts execution of the Absence segment (block 8). Otherwise, a similar test is made on the trace bit in the CST entry. Likewise, a stack marker and the label are pushed onto the stack, external interrupts are disabled, and control is transferred to segment 16, the Trace segment (block 9). If neither of these tests is affirmative, PCAL execution continues.

The next check is to see if bits 1 through 7 of the label are 0. These bits specify which STT entry in the target segment contains the desired local label. Since a value of 0 would point at the STTL word in PL, the value of 0 is specially defined to indicate that P should start at PB (as set four paragraphs back). Thus only one more check is necessary before execution of the procedure may begin, and that is to see if the segment is callable (bit 1 of the STTL word in the PL location must be 0); if it is uncallable, control is transferred to segment 15, the STT Entry Uncallable segment (block 10). Control is transferred by pushing the label onto the stack, disabling external interrupts, establishing PB, P, and PL from the CST entry for segment 15, and updating the Status register.

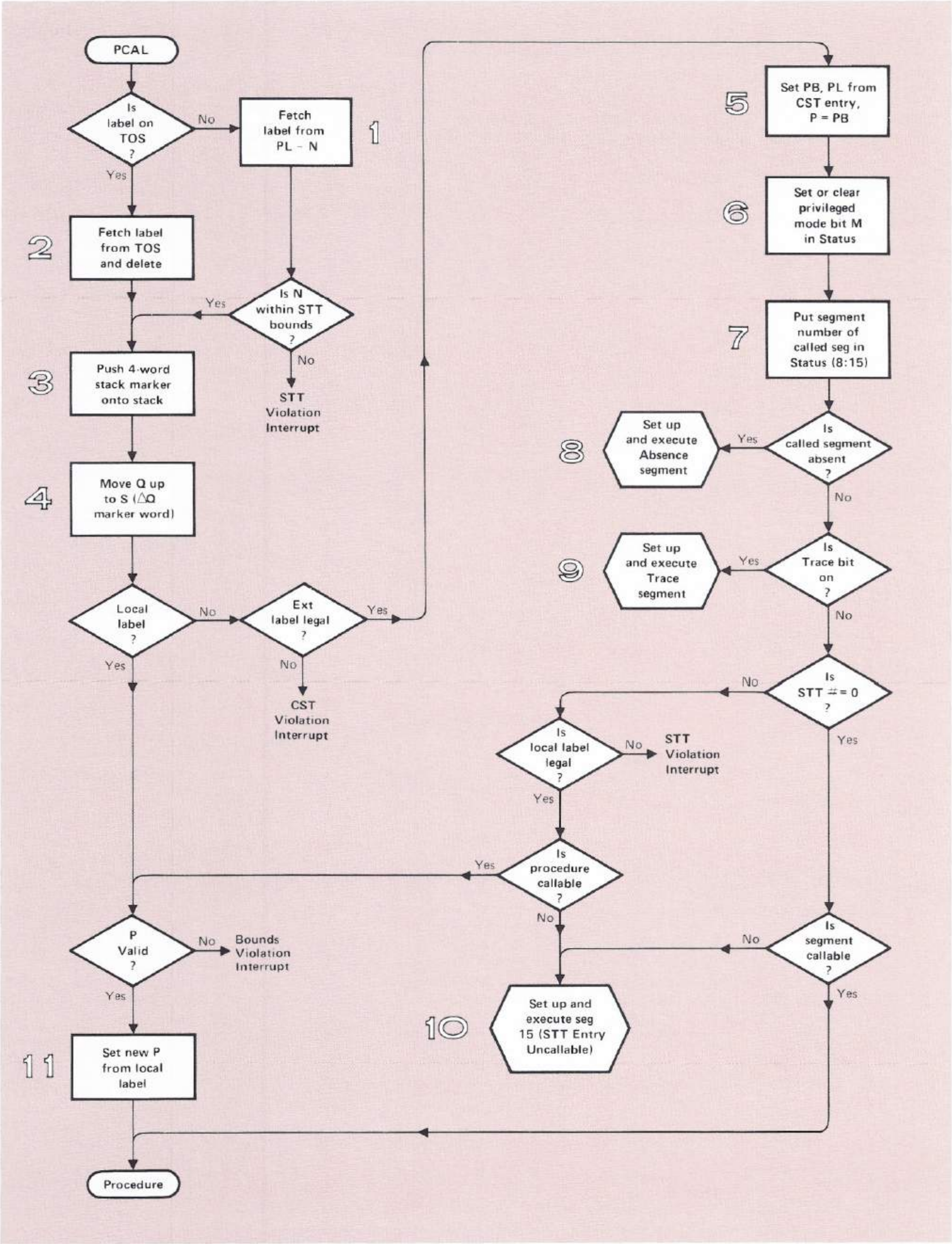

PCAL
Is label on TOS ?
No
Yes
Fetch label from PL - N
1
2
Fetch label from TOS and delete
Is N within STT bounds ?
Yes
No
STT Violation Interrupt
3
Push 4-word stack marker onto stack
4
Move Q up to S (ΔQ marker word)
Local label ?
No
Yes
Ext label legal ?
Yes
No
CST Violation Interrupt
5
Set PB, PL from CST entry, P = PB
6
Set or clear privileged mode bit M in Status
7
Put segment number of called seg in Status (8:15)
Is called segment absent ?
Yes
No
8
Set up and execute Absence segment
Is Trace bit on ?
Yes
No
9
Set up and execute Trace segment
Is STT # = 0 ?
No
Yes
Is local label legal ?
No
STT Violation Interrupt
Yes
Is procedure callable ?
Yes
No
Is segment callable ?
No
Yes
10
Set up and execute seg 15 (STT Entry Uncallable)
P Valid ?
No
Bounds Violation Interrupt
Yes
11
Set new P from local label
Procedure

Figure 5-13. Operations of PCAL

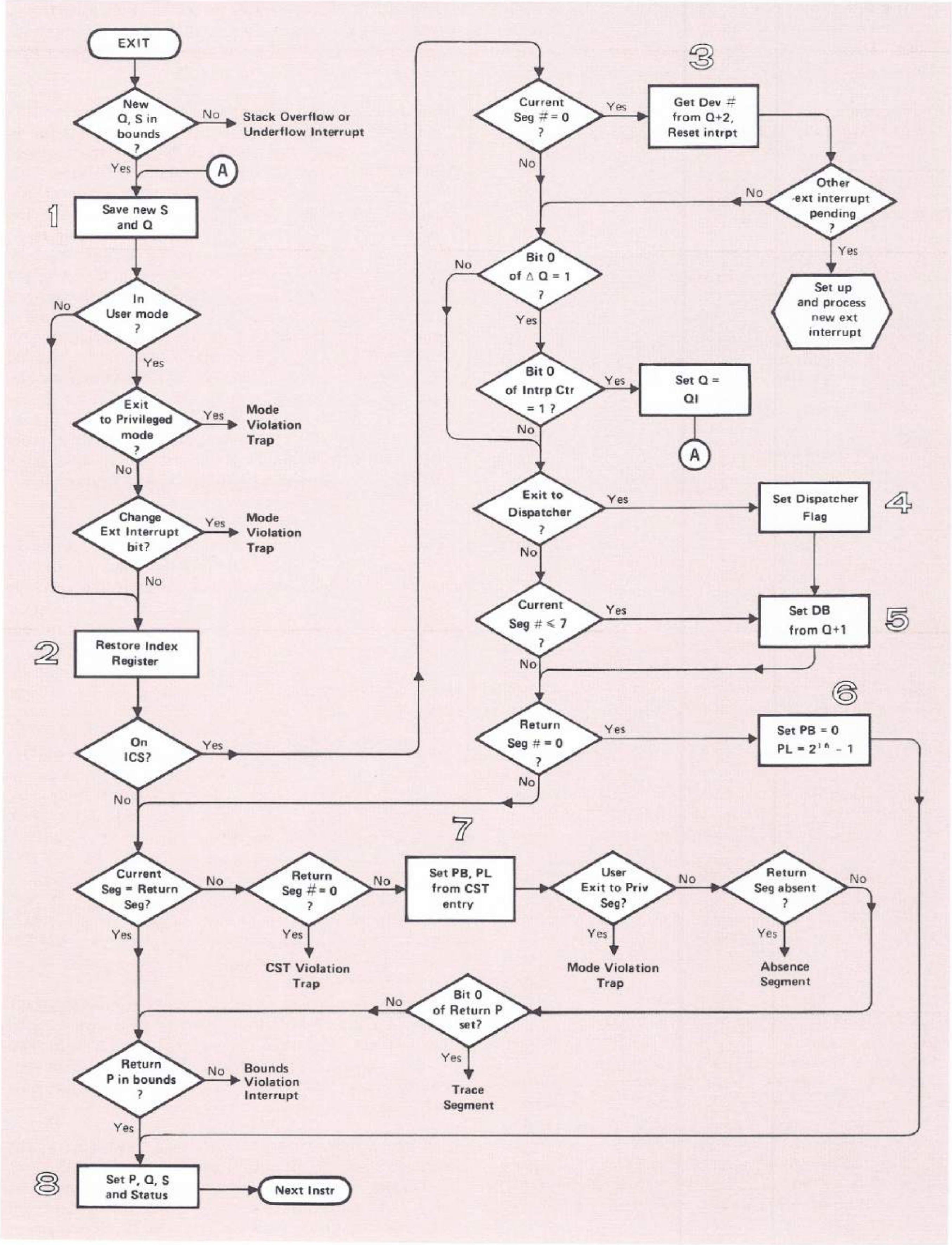


Figure 5-14. Operations of EXIT

Assuming that bits 1 through 7 of the external label are not 0, the value so indicated will point to one entry in the Segment Transfer Table. If it does not (i.e., if the value exceeds the STTL value), or if the entry pointed to is not a local label (i.e., if bit 0 = 1), there will be an STT Violation interrupt to segment 13. But if the label is valid, it is then checked to see if the procedure is callable by checking bit 1 (must be 0). If it is not callable, control is transferred to segment 15, the STT Entry Uncallable segment (block 10). If the local label indicates that the procedure is callable, the PCAL sequence continues.

Block 11 sets the P-register to the starting address of the procedure. The CPU at this point has a local label, whether it is in the same segment as the PCAL or in a segment external to the calling segment. The value for P is calculated by adding the contents of bits 2 through 15 of the local label to the contents of PB. As a final check, this value for P is checked to see that it does not exceed PL (Bounds Violation interrupt to segment 11 if it does). The resultant absolute value is then loaded into the P-register, and the location so indicated is fetched and execution of the procedure begins.

EXIT Sequence. Figure 5-14 illustrates the operations of the EXIT instruction. The diagram breaks down into three major functional sequences as follows: 1) Only the operations down the left side of the diagram are used when the exit occurs on any stack except the Interrupt Control Stack, and when the return is not to another segment. 2) A major branch to blocks 3 through 6 occurs if exiting from a routine that uses the Interrupt Control Stack. 3) A major branch to block 7 occurs if the return is to some segment other than the current one. The detailed sequence follows.

The first step is to check if the new values for Q and S are within bounds. In all cases, the new value for Q must be less than or equal to the Z-register content; otherwise there will be a Stack Overflow interrupt to segment 3. For user mode (but not privileged mode), Q must also be greater than or equal to the DB-register content; otherwise there will be a Stack Underflow interrupt to segment 13. Likewise, the new value for S must always be less than or equal to Z and, for user mode only, must also be greater than or equal to DB. The new value for S is Q-N-4, computed from the old Q. The new values for Q and S are saved temporarily within the CPU (block 1).

Next, two checks are made for illegal operations in user mode. The first check assures that an unprivileged user cannot accidentally or deliberately exit to the privileged mode. A Mode Violation trap to segment 17 is incurred if Status register bit 0 is a "0" (meaning, exit from user mode) and the corresponding bit in the status word of the stack marker does not match. The second check incurs the same trap if the user has changed the state of bit 1 (Enable/Disable External Interrupts) in the status word of the marker. Neither of these checks is made in privileged mode.

Assuming that no errors have occurred to this point, the sequence now restores the Index value from the stack marker to the Index register (block 2). Following this, a check is made to see if the current stack is the Interrupt Control Stack (ICS). If it is not, the sequence skips the next four paragraphs.

When exiting from a routine that has been using the ICS, the first check is to determine if the routine was for an external interrupt. This check is "yes" if the Segment Number field of the status register indicates Segment 0. If this field is not 0, the sequence skips the rest of this paragraph. For any external interrupt routine exit, the active state of the device's interrupt logic must be reset, since we can now assume that the request has been serviced. Thus an RIL (Reset Interrupt Level) signal is sent to the de.ice (block 3) and a check is made to see if the device's Interrupt Active flip-flop actually did reset. If it did not, a serious I/O error is indicated and the system will halt, with the SYSTEM HALT light on and all interrupts disabled. Otherwise, the next check is made which determines if any other external interrupts are pending. If so, the device number of the highest priority interrupt request is put on the stack in place of the former device number, and processing of this new interrupt begins; the existing stack marker is not changed. If no other external interrupts are pending, the EXIT sequence continues.

Bit 0 of $\Delta Q$ is now checked to see if it is a 1, which in effect asks if the Dispatcher was interrupted. If not, the remainder of this paragraph is skipped. If the check is affirmative, bit 0 of the Interrupt Counter (fixed location 7) is checked to see if it is a 1, which in effect asks if the routine being exited has requested a Dispatcher abort. If the answer is yes, Q is set to QI and the sequence goes back to block 1. Otherwise the sequence continues to the next paragraph.

A check is then made to see if we are exiting from the last routine to use the ICS. In such a case, control must be passed to the Dispatcher. This condition is tested by checking the content of the delta Q word of the stack marker. If the word is all zero, the Dispatcher Flag is set (block 4), indicating a return to the Dispatcher. The Dispatcher DB value from Q+1 is loaded into the DB-register (block 5). The next check determines whether the current segment is one which could have altered DB (segments 0 through 7). If this test is affirmative, the appropriate DB value from Q+1 is loaded into the DB-register (block 5).

If we are returning to a partly completed external interrupt routine, PB and PL are returned to their respective extreme values (block 6), with $PB = 0$ and $PL = 2^{16}-1$. The sequence would then proceed directly to block 8 (skip next two paragraphs.) Otherwise, the sequence continues with the following paragraph.

A check is now made to see if the return is to some segment other than the current one. If not, several checks involved in changing segments can be bypassed (remainder of this paragraph). If a return to segment 0 is indicated, there will be a CST Violation interrupt to segment 13, since segment 0 is undefined. Assuming this error does not exist, the next step (block 7) is to use the Status information in the stack marker to fetch the CST entry for the segment we are returning to. The CST entry gives both an absolute value for PB (second word of the entry) and a PB+ displacement for computing an absolute value for PL. These values are loaded into the PB-register and PL-register respectively. Next, a check is made to see if a user mode exit is attempting to return to an uncallable segment (bit 1 of its CST entry is a "1"). If so, there will be a Mode Violation trap to segment 17. Otherwise, bit 0 of the CST entry is then tested. If this bit is a "1", the segment being returned to is absent. In this case, the N parameter from the EXIT instruction is pushed onto the stack, external interrupts are disabled and control is transferred to segment 14, the Absence segment. If the absence-test is negative, the trace bit (bit 0) in the return-P word of the current stack marker is checked to see if it is a "1". If so, the parameter N from the EXIT instruction is pushed onto the stack, external interrupts are disabled and control is transferred to segment 16, the Trace segment. Otherwise, the EXIT sequence continues.

The final check determines that the new value for P (which is calculated by adding the PB relative displacement from the old stack marker to the PB-register content) does not exceed the PL value. If it does, there will be a Bounds Violation interrupt to segment 11.

The sequence finishes by loading the values for P, Q, S, and Status into their respective registers (block 8). The next instruction pointed to by P is fetched for execution.

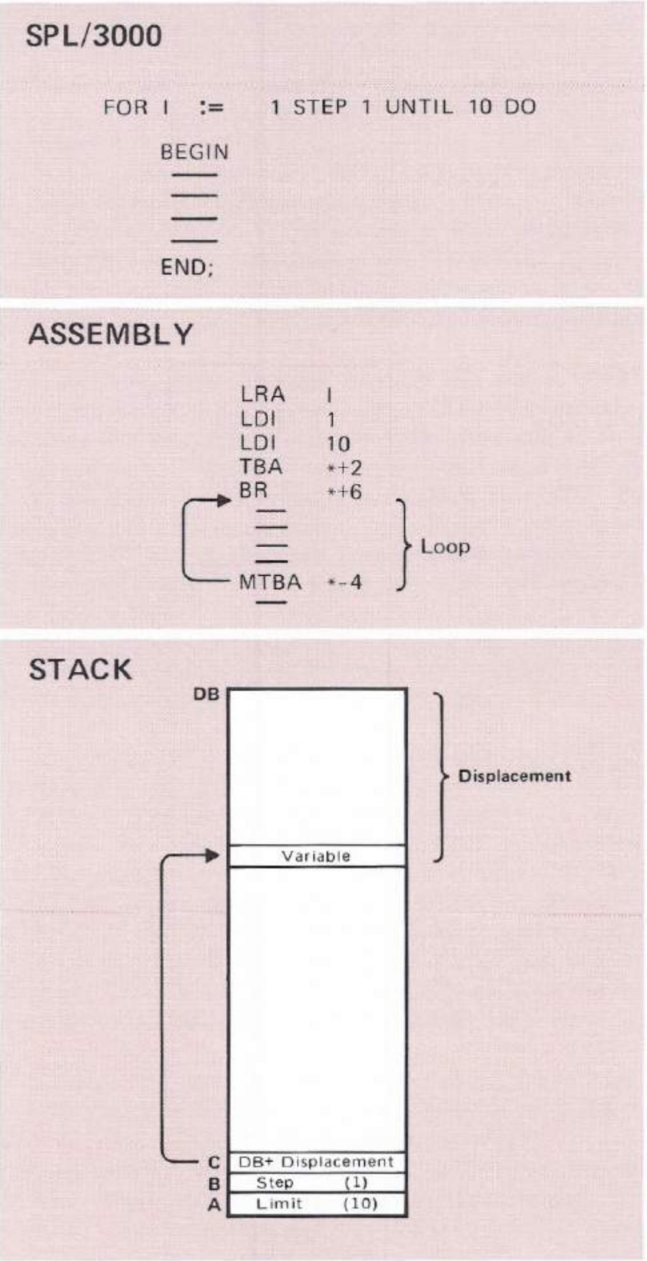
**17** TBA, MTBA, TBX, MTBX. These four instructions perform essentially the same function, and that is to provide a simple mechanism for loop repetition, loop counting, and loop exit, all in one instruction. The differences are that:

a. For TBA and MTBA, the variable is located in the stack; for TBX and MTBX the variable is located in the Index register.

b. For TBA and TBX, modification of the variable is assumed to have been done earlier in the loop, whereas MTBA and MTBX automatically modify the variable as part of their execution function.

With these differences understood, one of the instructions may be taken as a typical example for discussion. Figure 5-15 illustrates one use of MTBA, which is to execute the SPL/3000 FOR statement. As shown, the intent is to vary the value I from 1 to 10 while repeating a certain procedure ten times. (The TBA at the beginning is used to test if the loop is to be executed zero times in the general FOR statement.)

In assembly form, three instructions would be used to initialize the stack. The LRA I instruction puts the DB+ displacement for the variable onto the stack (C), and LDI 1 and LDI 10 push the values 1 and 10 (or octal 12) onto the stack to specify the step increment (B) and limit (A) respectively. The loop is then entered. (If the loop control instruction at the end were TBA or TBX, one of the instructions in the loop would add B to the variable.)



```
FOR I  :=   1 STEP 1 UNTIL 10 DO
   BEGIN
    —
    —
    —
   END;
```

```
LRA    I
LDI    1
LDI    10
TBA    *+2
BR     *+6
 —
 —
 —
MTBA   *-4
 —
```

Figure 5-15. Example of Loop Control with MTBA

The last instruction of the loop is MTBA, which checks to see if the variable has exceeded the limit. If it has not, control is transferred back (four locations in this example) to the beginning of the loop. The range is P ± 255. At the end of the final loop, MTBA increments the variable to 11, thus exceeding the limit and causing the next instruction in line to be fetched. The three words on the TOS relating to this loop are automatically deleted. The FOR statement has now been executed.

Values for the limit, step, and variable may be negative (two's complement) as well as positive. If step is negative (bit 0 = 1), exit from the loop will occur when the variable becomes smaller (more negative) than the limit, which may be either a positive or a negative number.

## I/O SYSTEM OVERVIEW

The purpose of the *I/O system* is to perform actual physical input/output operations for the *file system* of the MPE/3000 operating system. The user normally does not interact directly with the I/O system — only indirectly via the file system. Thus all I/O operations are normally invisible to the user. However, privileged users may access the I/O system directly, and users with real-time capability may bypass both the file system and the I/O system for direct access to specific devices. See figure 6-1.

This section of the manual presents a generalized description of the I/O system as accessed via the file system. Direct access by privileged or real-time users requires a deeper level of familiarity than is presented here.

### FILE SYSTEM OPERATION

Figure 6-2 illustrates the function of the I/O system in the overall handling of files. Hardware elements are shown on the right and software elements are shown on the left. The I/O system, as shown, is part hardware and part software. Several peripheral devices are shown connected to the I/O system, each of which has some capability for handling files — entering files, storing files, or both. Of particular interest in this discussion are the files stored on disc. (Several physical disc units might be used.) Each disc file is broken up into one or more *extents,* which in turn are composed of some number of *blocks.* When the file system causes the I/O system to transfer data to or from the disc, it does so one block at a time. The blocks are further subdivided into records and then into individual words. When the file system processes user file requests, it does so on the basis of records.

The memory management routine is also shown in figure 6-2 (dotted line) since it frequently makes its own requests to the I/O system. Memory management calls I/O in order to make drivers and code segments present in main memory.

In typical operation, the user's process might make a file request such as FREAD to the file system (1). The file system reads the record named in the request (2) and transfers the record to the stack associated with the user's process (3). Note that in this example, no input/output has taken place. This is because the named record is already present in a buffer in main memory.

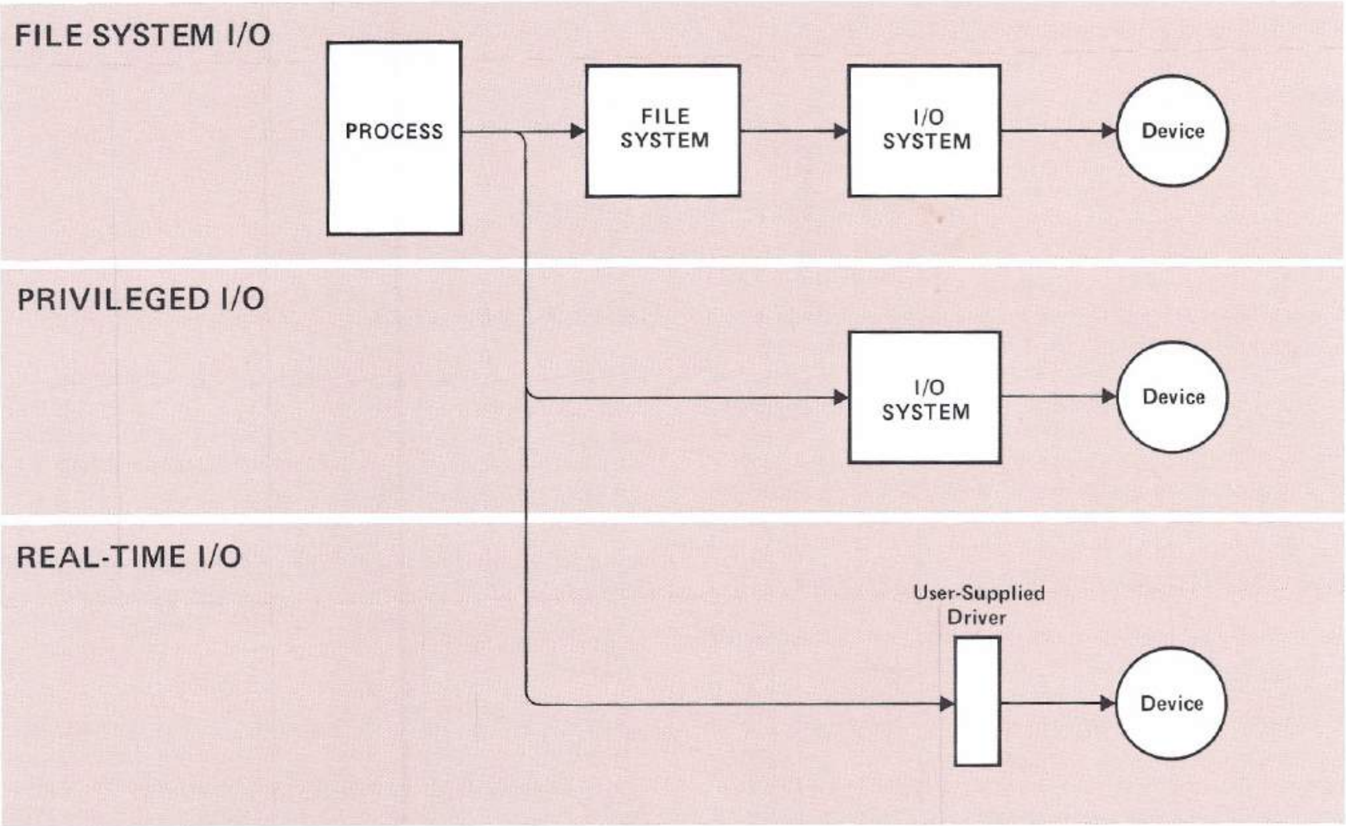


Figure 6-1. Basic I/O Access Methods

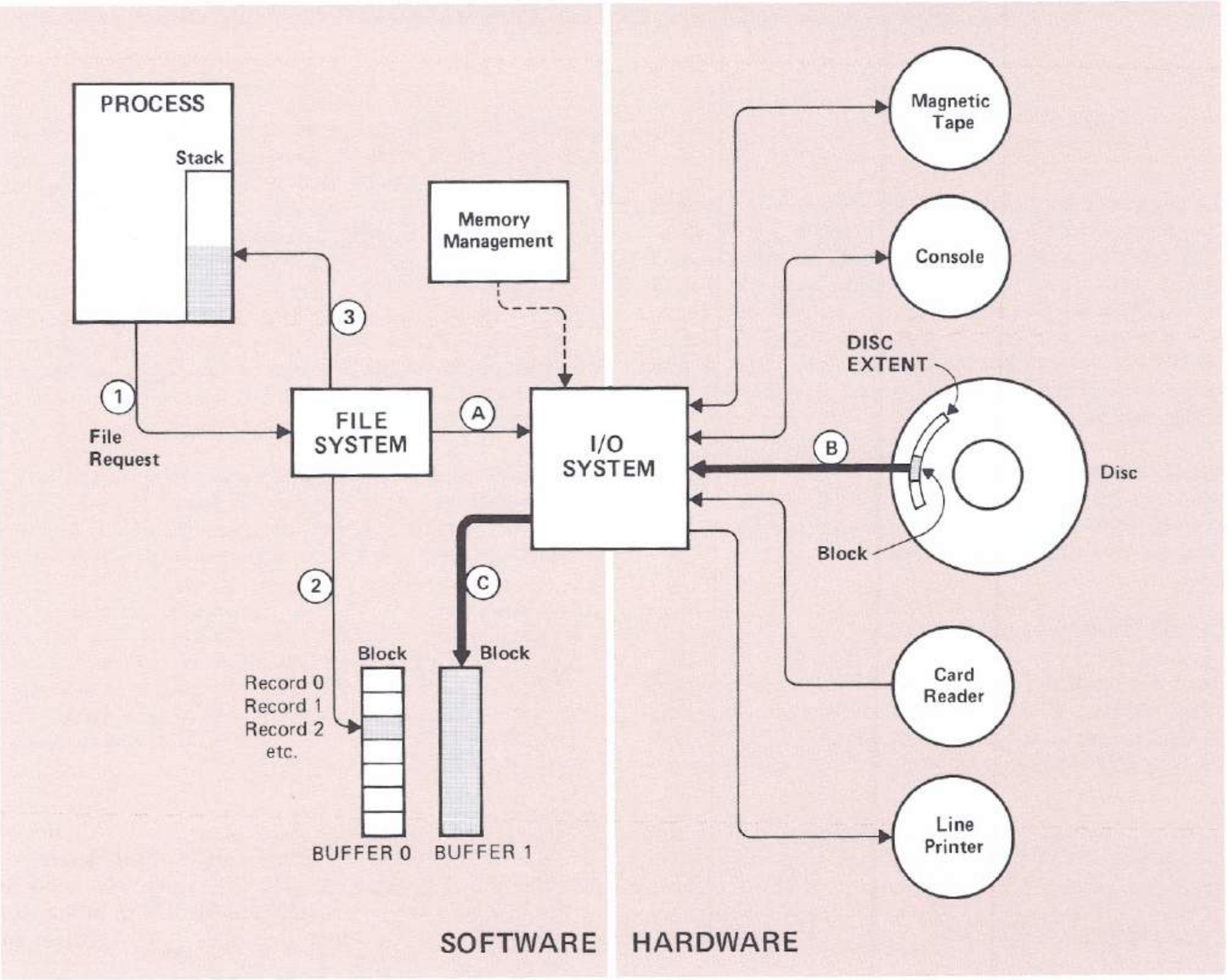


Figure 6-2. File System Basic Operation

Assume another case in which the requested record is not present. In this case, the file system makes a request to the I/O system (A) to read the block containing the particular record. The I/O system accordingly reads this block from the disc (B) and loads it into one of the buffers allocated to the named file (C). (When a user opens a file, he specifies how many buffers should be allocated for that file; however he cannot access the buffers directly — only by naming records within files.) The file system can now complete the request by reading the requested record to the stack.

Note that in none of the preceding operations did the user's process specify a device. An actual I/O operation may or may not have occurred, and the user is completely unaware of such occurrence. However, as described in the MPE/3000 Reference Manual, the operating system does permit devices to be specified, either as a class name or a logical device number. This would permit, for example, inputting or outputting files via a specific terminal, card reader, or line printer.

## DEFINITION OF TERMS

The preceding discussion presented a broad overview to show the relationship of the I/O system to the file system and peripheral devices. The following descriptions will concentrate on the block labeled "I/O System" in figure 6-2, explaining this area in greater detail.

Figure 6-3 illustrates some of the important elements of the I/O. system. This picture is by no means complete, but rather is intended to define the chain of linkages that are basic to the I/O system.

As shown in figure 6-3, a *device controller* is the hardware I/O interface, typically consisting of one or two interface cards. Depending on particular controllers, the device controller may drive only one peripheral (such as a terminal) or may be capable of driving several peripherals (such as disc units).

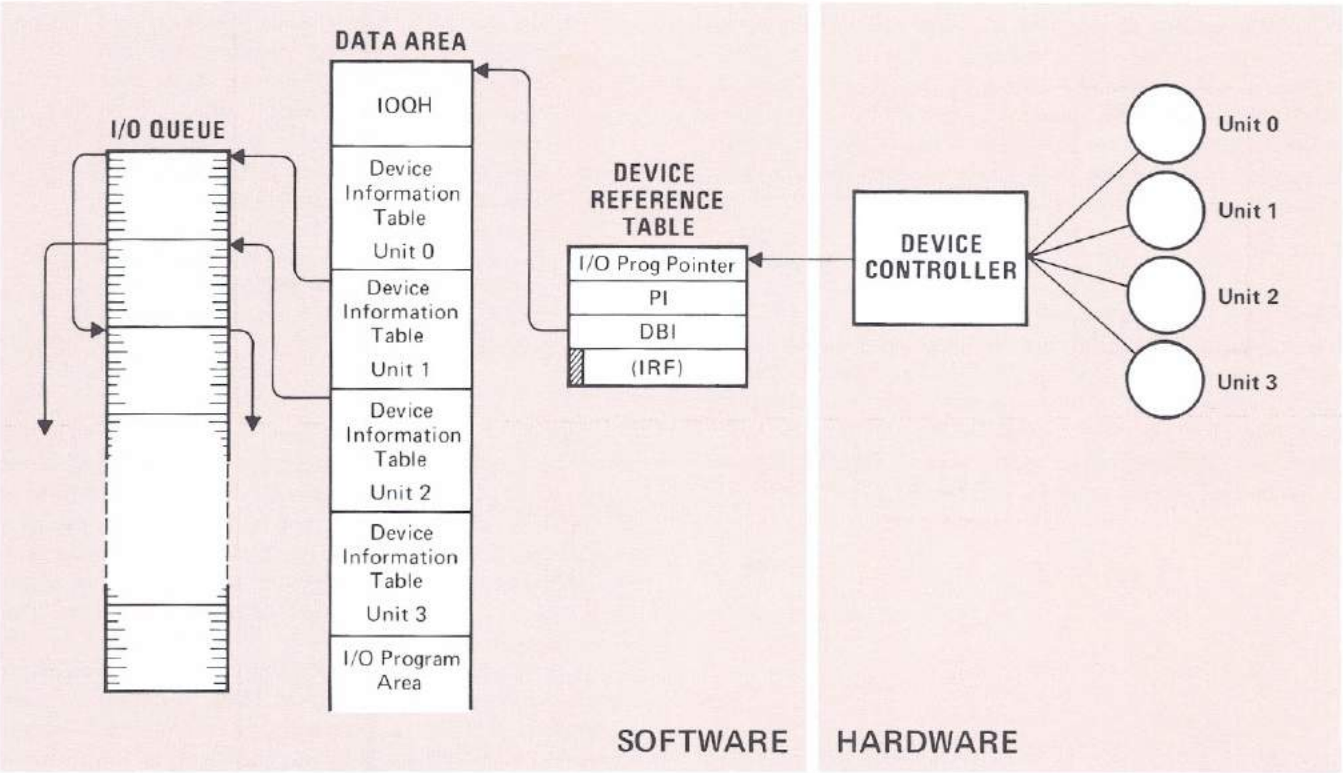


Figure 6-3. Fundamental Elements of I/O System

For each device controller there is a four-word entry in the Device Reference Table. (The Device Reference Table will be defined shortly.) The third word of this table entry contains a pointer to a data area uniquely associated with that table entry.

The data area consists of an *I/O Queue Head* (IOQH), one or more *Device Information Tables* (depending on how many units the device controller is driving), and an I/O program area. The IOQH contains CST and STT values for defining the location of the driver routines associated with that particular device controller. Along with various other information, the IOQH also defines how many Device Information Tables are present, and how long each one is.

The Device Information Table contains information relevant to one physical I/O device, and is differently configured for each type of device. In each case, however, the first word of this table points to an entry in the *I/O Queue* (IOQ) when a request is being made.

The I/O Queue is a single table (only one in the system) containing a fixed number of entries having a fixed number of words per entry. If there are no I/O requests pending in the system, none of the Device Information Table entries will be pointing to the IOQ. In this case, all elements of the IOQ are unused, and the first word of each element points to the first word of the next element. Thus, all unused elements are linked together. Assume, then, that the file system makes a request to use unit 1 of the device controller shown in figure 6-3. The I/O system will unlink the first free element in the IOQ and fill it with information pertaining to the request (including buffer address and logical device number).

Figure 6-3 assumes that the next request is for unit 2 (uses the next available element), followed by a second request for unit 1. This second request for unit 1 causes the first word of the initial request to point to the next unused element, which is then filled with information pertaining to the second request. Thus it can be seen that eventually the IOQ will contain a queue of requests for unit 1, a separate queue for unit 2, and so on, plus a linked list of free elements.

Then an I/O process is dispatched to execute the request. An I/O program will then be run on a device, using the request parameters given in the IOQ. When the request is fulfilled, the IOQ element is returned to the free list.

Note that the IOQ only establishes the priority of requests for each device, on a first-in first-out basis. Questions of priority in dispatching I/O processes (i.e., which queue) are resolved by the Dispatcher. Once several device controllers are running I/O programs, priority conflicts are resolved by hardware service priority.

Figure 6-4 illustrates the *Device Reference Table* (DRT). The DRT consists of a number of four-word entries corresponding to the number of device controllers present in the system. It is located in fixed memory locations beginning at octal address 14. (Locations 0 through 13 are allocated to other purposes; see table 4-1.) The upper limit for the table is location 1777, which thus limits the maximum number of four-word entries to 253 (decimal).

Since each DRT entry is always four words in length, it is convenient for the hardware to map device numbers to DRT addresses simply by multiplying by four. (Left-shift device number two binary places.) Thus the entry for device number 3 begins at octal location 14 (i.e., $3_8$ x 4 = $14_8$). Since the DRT begins at location 14, device number 3 is the lowest device number. Devices 0, 1, and 2 do not exist.

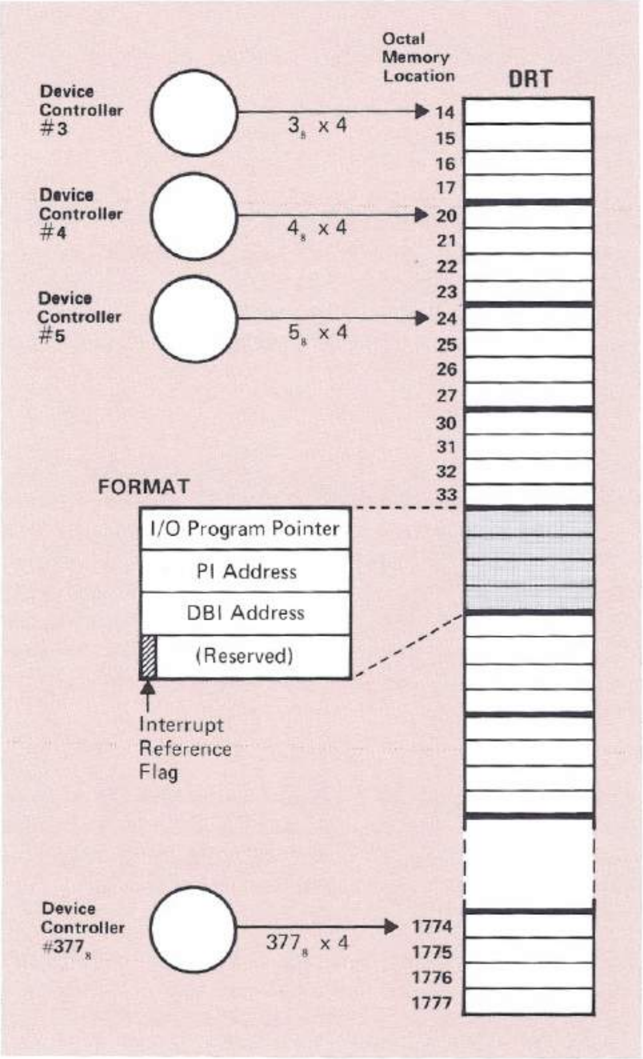


Figure 6-4. Device Reference Table

Note: The *device number* associated with a particular DRT entry defines a device controller or multiplexer channel, and not necessarily an actual device. Remember also that some controllers, identified by one "device number", are capable of driving several physical units. Individual identification of actual devices is made by *logical device numbers*. The logical device number is the value used by the file system in requesting I/O, and the I/O system software performs the logical to physical device number translation.

The format of a DRT entry is also shown in figure 6-4. The first three words are absolute addresses and the fourth word contains a bit for use as an interrupt flag. The first word is the *I/O Program Pointer*, which initially points to the first word of the I/O program for the associated controller, and (for multiplexed channel devices) is updated to point at the next program word as the I/O program progresses. The second word (PI) points to the starting address of the interrupt program for the associated controller. (Interrupts are discussed in Section VII.) The third word (DBI) points to the data area for the associated controller. The Interrupt Reference Flag (fourth word) is discussed in Section VII, Interrupt System; it is primarily a user feature and is not used by the I/O system.

## I/O INSTRUCTIONS

There are five I/O instructions in the HP 3000 instruction set. These are:

| | |
|---|---|
| SIO | Start I/O |
| RIO | Read I/O |
| WIO | Write I/O |
| TIO | Test I/O |
| CIO | Control I/O |

These instructions are fully defined in Section V under the heading "I/O and Interrupt Instructions". The distinction to note here is that the SIO instruction is used in conjunction with an I/O program, and the remaining four are not. That is, the SIO instruction commands a device controller to begin executing its associated I/O program, which effects a block transfer of data between an I/O device and memory. This is termed an "SIO transfer" mode. The other four instructions, on the other hand, transfer only one word per instruction, between the device and the top-of-stack in the CPU. This is a "direct transfer" mode, and is used primarily with terminal devices. In this manual, direct I/O is usually treated separately from normal SIO operations, due to these differences.

For additional information on transfer modes, refer to the "Transfer Modes" heading in Section VIII.

This page intentionally blank.

Section VI continues on the following page.

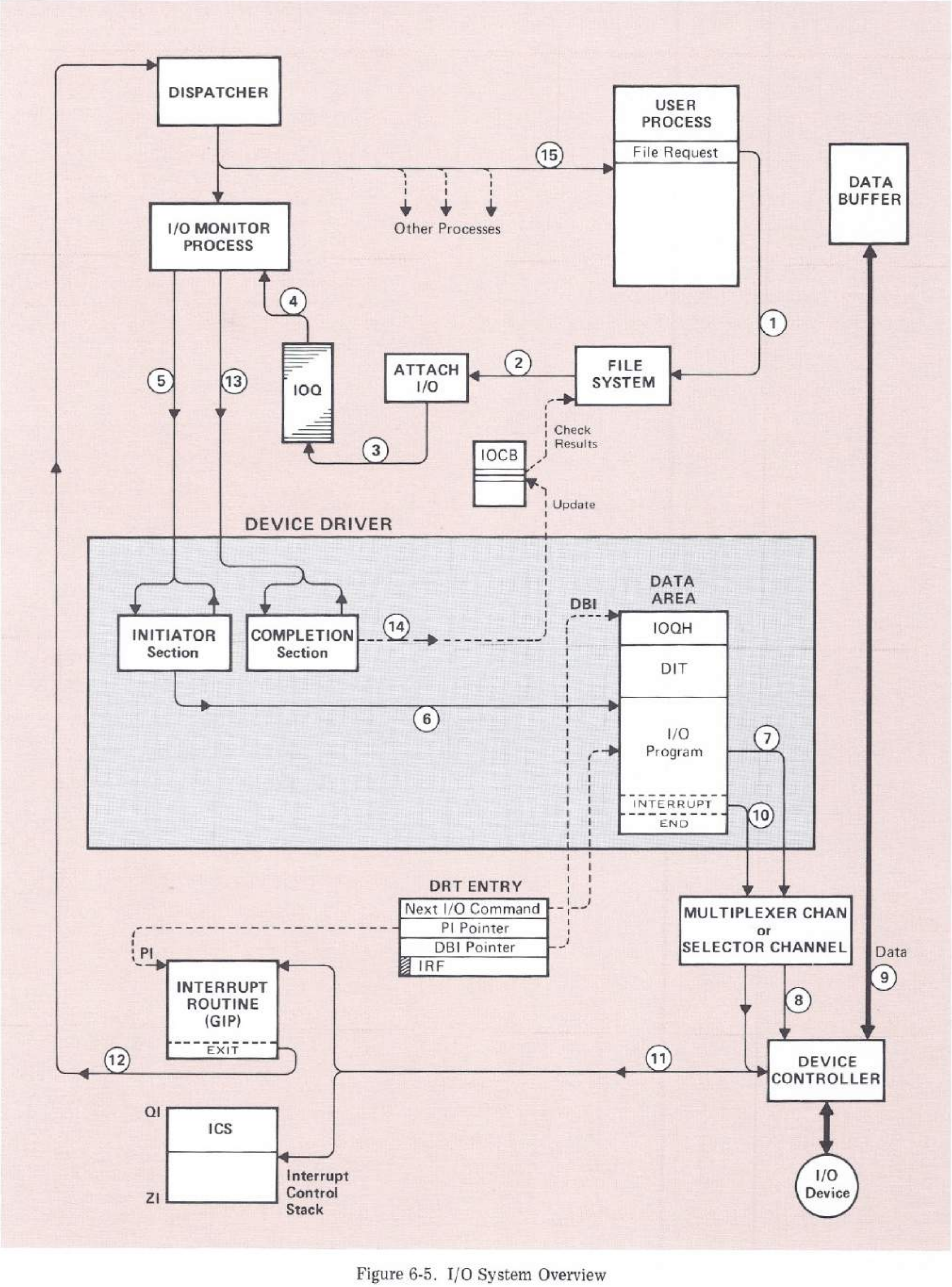


Figure 6-5. I/O System Overview

## GENERAL I/O OPERATION

Figure 6-5 is a general overview of the operations of the I/O system. (Does not apply to direct I/O devices.) To provide a complete sequence of operations, it will be assumed that the file request will result in a need for physical I/O to be performed; as stated earlier, this will not always be the case. The sequence of operations is as follows.

① An executing user process generates a file request to the file system.

② The file system tests the validity of the request and calls the *Attach I/O* (ATTIO) intrinsic. This is the entry point to the I/O system, implied by the first and second examples of figure 6-1.

③ Attach I/O inserts the request parameters in the I/O Queue for the requested device.

④ When all earlier requests for the device have been completed, and when the *I/O Monitor Process* has highest priority among all other processes, the I/O Monitor Process begins execution for this request.

⑤ The I/O Monitor process ensures that the data buffer for the file is present and frozen in memory. It then issues a PCAL to the initiator section of the *device driver*, passing the request parameters to that routine.

Note: A device driver normally consists of three parts: an initiator section, a completion section, and a data area. With multiple data areas, one driver may drive several devices.

⑥ The initiator section assembles the I/O program (using the request parameters), issues an SIO instruction to the device controller, and exits back to the I/O Monitor Process. The SIO instruction initializes the DRT to point at the starting location of the I/O program.

⑦ ⑧ The I/O program issues commands via a multiplexer or selector channel to the device controller, on demand by the channel.

⑨ The device controller, on receiving a read or write command from the I/O program, transfers a block of data to or from the data buffer. The length of the block is specified by the I/O command.

⑩ On completion of the data transfer, the I/O program commands the device controller to request an interrupt. The I/O program then ends.

⑪ The device controller causes a CPU interrupt to an interrupt routine, which tells the I/O Monitor Process that an interrupt has occurred.

Note: There are currently two interrupt routines for external interrupts. One is the General Interrupt Processor (GIP) for all types of devices except terminals, and the other is the Terminal Interrupt Processor (TIP). Other interrupt routines may exist, depending on the requirements of newly developed interfaces.

⑫ The interrupt routine (or the last routine to use the Interrupt Control Stack — see ICS definition in Section VII) exits to the Dispatcher. It also may awaken the related I/O process if necessary.

⑬ When the I/O Monitor Process is again dispatched, it recognizes that an interrupt has occurred and accordingly calls the completion section of the device driver.

⑭ The completion section checks the results of the transfer. If necessary, it may initiate additional transfers by telling the I/O Monitor Process to call the initiator section again. Otherwise, it updates the *I/O Control Block* with information regarding results of the original request. The file system may then check these results. The I/O Control Block is a table of doubleword entries, with one entry for each I/O request. Each entry contains a transmission log (number of words or bytes transferred), logical I/O status, and the process number of the process to activate upon I/O completion.

⑮ When the user process is again dispatched, return is made to a point following the file request, depending on whether blocked or unblocked I/O was specified. (Refer to discussion on blocked/ unblocked I/O later in this section.)

## DIRECT I/O OPERATION

The operations for direct I/O involve considerably more software overhead than the operations for the SIO transfer mode. This is due to the varied nature of the terminal devices that use direct I/O, and also to the fact that the system must respond to commands entered via the terminal as well as to file requests affecting that terminal.

In addition, the operation is complicated by such factors as speed sensing, error sensing, whether the device is synchronous or asynchronous, whether the device is capable of reading or writing or both, what controls exist, and which mode or modes the device is capable of. Also, the log-on sequence is handled by an entirely different set of routines than those used for data handling.

Thus, the sequences described in the following paragraphs present only a broad generalization of direct I/O terminal operations. The sequences given should not be construed as

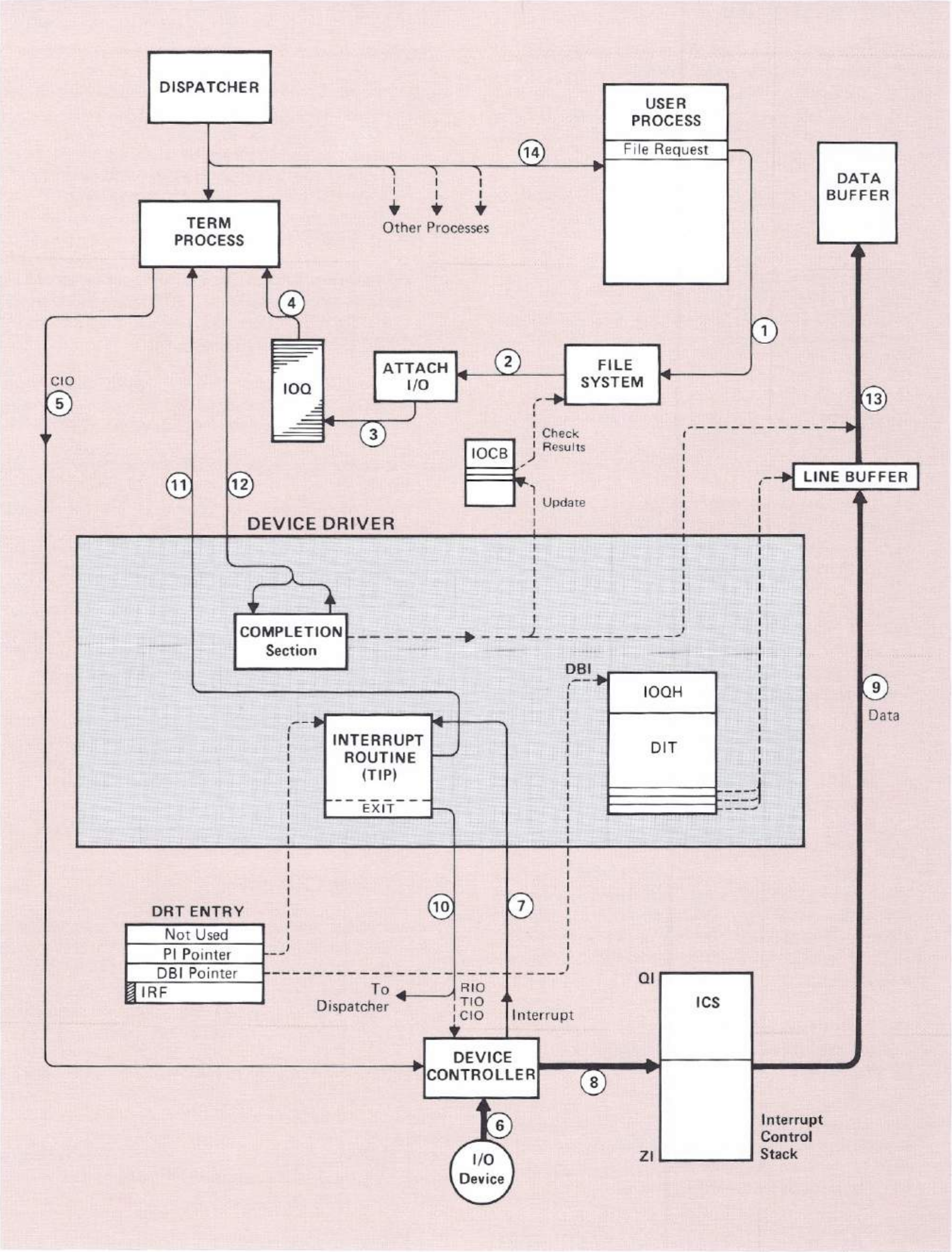


Figure 6-6. Direct Read for Terminal Devices

representing any particular device or even a "typical" device. It will be assumed that the log-on sequence has been accomplished.

Figures 6-6 and 6-7 illustrate the handling of data via direct I/O terminal devices. Figure 6-6 shows input (read) operations and figure 6-7 shows output (write) operations.

In comparison with figure 6-5, note that there is no I/O program in the data area; instead, the interrupt routine performs the functions of an I/O program. The interrupt routine, in this case, is part of the device driver.

Note also that direct read uses no initiation section and direct write uses no completion section. Also: no multiplexer or selector channel is involved.

One element not previously present is the *line buffer*. The line buffer consists of a number of *buffer tanks*, which are pointed to by address words in the Device Information Table for a particular terminal. A sufficient number of these tanks is used to accommodate the line or record length of the associated device. Data is transferred between the line buffer and the device (via the Interrupt Control Stack) on a character-by-character basis. Data is transferred between the line buffer and the data buffer on a record basis. This scheme conserves main memory space by allowing the data buffer to be absent on disc while the comparatively slow terminal device is transferring individual characters.

DIRECT READ. The sequence of operations for direct read, illustrated in figure 6-6, is as follows. Again, it will be assumed that the file request does require a physical read from the terminal.

① The executing user process generates a file request to the file system.

② The file system tests the validity of the request and calls the Attach I/O intrinsic.

③ Attach I/O inserts the request parameters in the I/O Queue for the requested device. Unlike the general (SIO) case, which uses a first-in/first-out queue for the requests, terminal requests are analyzed for relative importance and are then inserted into an appropriate place in the queue. The factors involved in assessing request importance are: mode (standard, escape, break, and console), and request type (standard, soft, and hard).

④ When all higher priority requests for the terminal have been completed, and when the *TERM process* has highest priority among all other processes, the TERM process begins execution for this request. (There is one TERM process for each terminal device controller.)

⑤ The TERM process enables interrupts and links together a sufficient number of buffer tanks to accommodate the request. It then issues a CIO (Control I/O) instruction directly to the device controller to enable read interrupt. TERM then exits to the dispatcher.

⑥ The device controller enables the device to read a character. When a key is pressed, the device returns the character to the controller.

⑦ On receipt of the character, the device controller causes the CPU to interrupt to the interrupt routine for terminals, TIP (Terminal Interrupt Processor).

⑧ TIP issues an RIO instruction to the device controller. This causes the character to be loaded onto the Interrupt Control Stack, and also causes a command to be issued to the device to read the next character. TIP now checks the character on the ICS to see if it is a data character or a control character.

⑨ If the character on the ICS is a data character, it is transferred by TIP to the line buffer. If it is a control character, TIP performs the appropriate control function.

⑩ TIP exits to the Dispatcher and the sequence repeats back to step 7 until the entire record has been read.

⑪ When TIP detects a CR character (Carriage Return), TIP sets a bit in the Device Information Table to signify that the record is complete, then exits back to the TERM process.

⑫ The TERM process, after checking the Device Information Table, issues a PCAL to the completion section of the device driver.

⑬ The completion section transfers the content of the line buffer to the data buffer. Then the transmission log in the I/O Control Block is updated and the completion section exits back to the TERM process.

⑭ TERM releases the buffer tanks and goes to sleep. The Dispatcher then returns control to the user process. To read another record, the file system must make another I/O request to Attach I/O.

DIRECT WRITE. The sequence of operations for direct write, illustrated in figure 6-7, is as follows:

① The executing user process generates a file request to the file system.

② The file system tests the validity of the request and calls the Attach I/O intrinsic.

③ Attach I/O inserts the request parameters in the I/O Queue for the requested device.

④ When all higher priority requests for the terminal have been completed, and when the TERM process has highest priority among all other processes, the TERM process begins execution for this request.

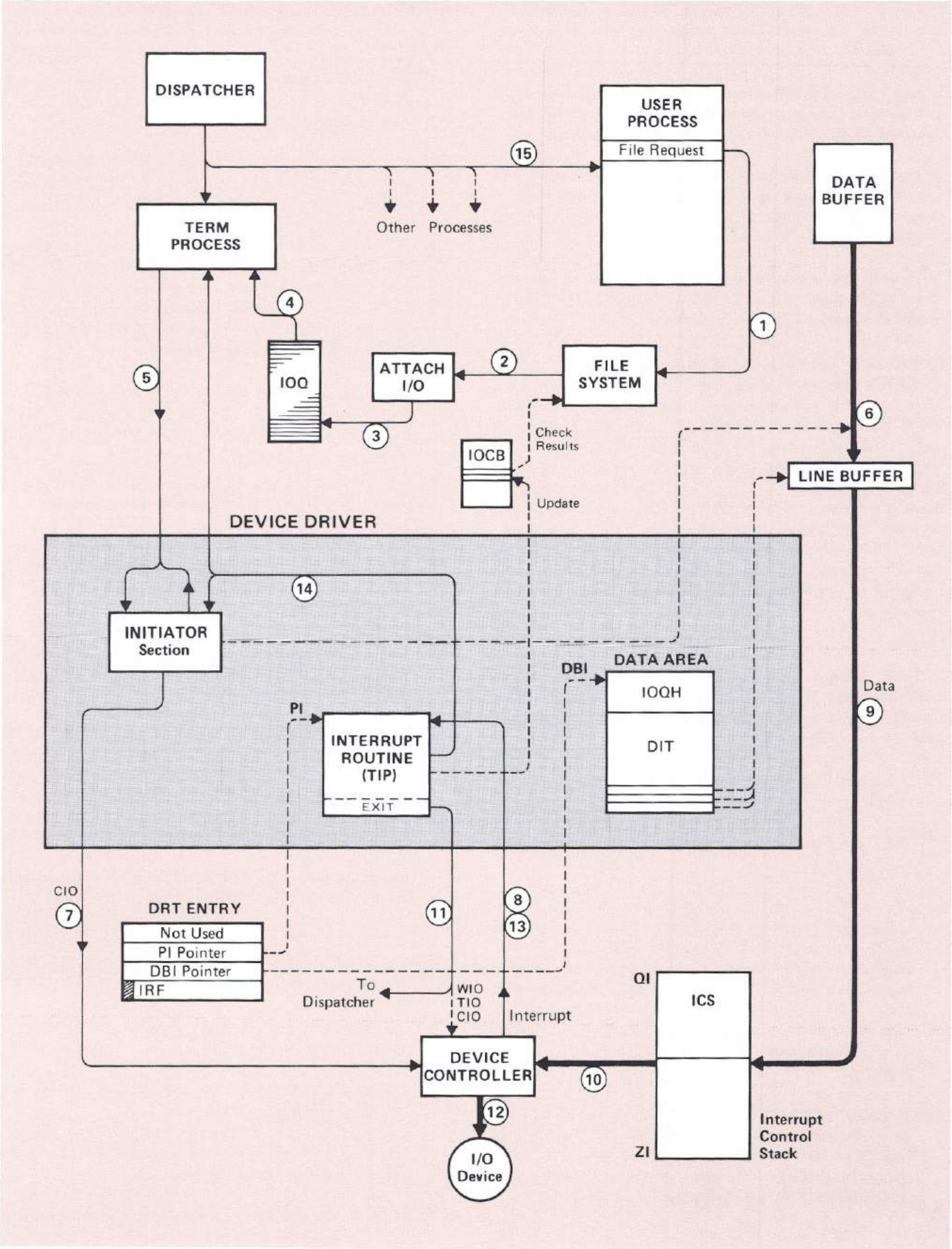


Figure 6-7. Direct Write for Terminal Devices

⑤ The TERM process enables interrupts and links together a sufficient number of buffer tanks to accommodate the request. TERM then issues a PCAL to the initiator section of the device driver.

⑥ The initiator transfers one line (maximum of 132 bytes) from the data buffer to the line buffer.

⑦ The initiator issues a CIO (Control I/O) instruction to the device controller to enable write interrupt and exits back to the TERM process.

⑧ The device controller causes the CPU to interrupt to TIP, the Terminal Interrupt Processor.

⑨ TIP transfers a byte to the ICS. If the byte is a control character, TIP does the control function and gets the next byte from the line buffer. If it is a data character, proceed to 10.

⑩ TIP executes a WIO instruction, transferring the character from the ICS to the device controller.

⑪ TIP then exits to the Dispatcher, while hardware takes control from this point.

⑫ The device controller writes the character out to the device.

⑬ On completion of the write, the device controller generates another interrupt to TIP. The sequence repeats back to step 9 until all characters in the record have been written out to the terminal.

⑭ When TIP detects a CR character (Carriage Return) in step 9, TIP checks a counter to see if this was the last line. If not, TIP calls the initiator again repeating back to step 6. If this was the last line, TIP exits back to the TERM process, which disables interrupts, releases the buffer tanks, and goes to sleep.

⑮ The Dispatcher then returns control to the user process.

## BLOCKED/UNBLOCKED I/O

At the conclusion of all three of the preceding operating sequences (general I/O, direct read, and direct write), control is returned to the user process on completion of I/O. While the I/O operation was in progress, the user process may have been suspended at that point to await I/O completion *(blocked I/O)*, or may have continued to execute while periodically checking for I/O completion *(unblocked I/O)*. The choice of blocked or unblocked I/O is made in the call to ATTIO. (The file system nearly always uses unblocked I/O.)

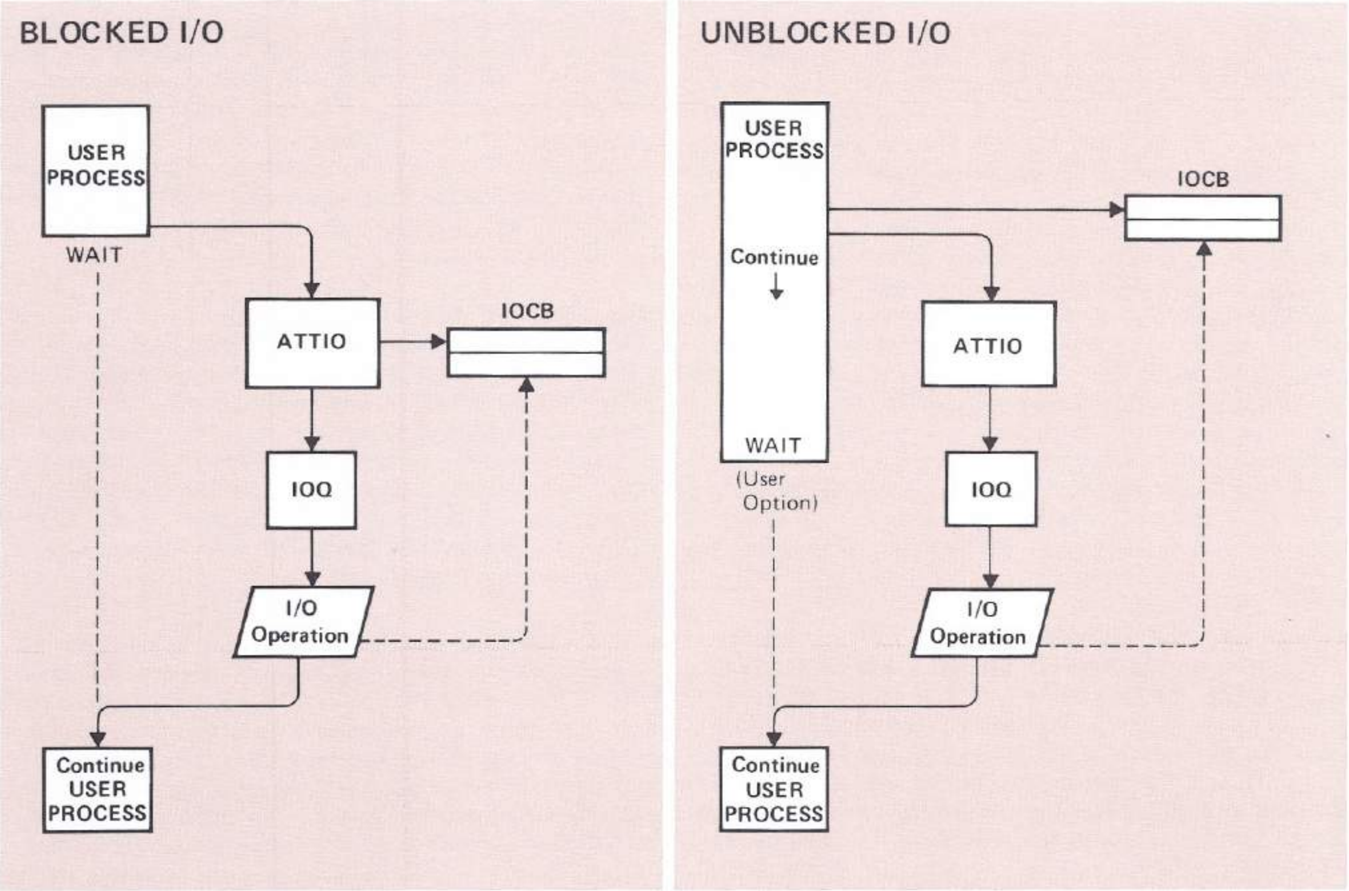


Figure 6-8. Blocked and Unblocked I/O

The following paragraphs discuss the characteristics of blocked and unblocked I/O. Refer to figure 6-8. ("User" implies privileged user.)

BLOCKED I/O. As shown in figure 6-8, the user process goes into an I/O wait substate as soon as the I/O request is given. Since an I/O Control Block will be provided automatically, it is not the user's responsibility to provide one.

The user process remains in the wait substate while the I/O operations proceed. The ATTIO intrinsic creates an I/O Control Block entry for this request. Then the request is entered into the I/O Queue and is ultimately processed via the hardware I/O system. At the end of the I/O operation, the results of the transfer are entered into the IOCB. Control is then returned to the user process, along with the contents of the IOCB (to the top of the stack). ATTIO then deletes the IOCB entry for this request.

The user process now continues to execute from the point following the I/O request.

UNBLOCKED I/O. In the case of unblocked I/O, also illustrated in figure 6-8, the user process must initially provide the I/O Control Block. (Privileged capability is assumed.) The process must also specify the action to be taken on completion of I/O: either no action or awaken the process if in an I/O wait substate. This specification (like the blocked/unblocked I/O choice) is made in the call to ATTIO.

The process may then, after calling ATTIO, continue to execute, and may generate other unblocked I/O requests. It is the responsibility of the process to synchronize all unblocked requests and to check the contents of the associated IOCB entries for I/O completion. The process also has the capability to put itself into the I/O wait substate, and to change the I/O completion action for any unblocked request at any time. Obviously, however, the process should not specify "no action" for all unblocked requests and then go into the I/O wait substate; there is no way to recover from this hanging situation. At least one request must specify "awaken process".

While the process continues to execute, ATTIO enters the request into the I/O queue, and hardware processing of the request begins. At the end of the I/O operation, the results of the transfer are entered into the user-provided IOCB. Then the completion action bit is examined. If "awaken process" is specified, the process will be awakened if it has put itself into the I/O wait substate, as shown in figure 6-8. If "no action" is specified, presumably the process has continued to execute without any wait, or will be awakened by some other process. In any case, the process checks for I/O completion by checking the contents of the IOCB.

## HARDWARE I/O SYSTEM

As evident from the preceding overview of I/O operations, the hardware portion of the I/O system bears a large measure of the responsibility in the execution of an I/O request. That is, when software passes control to hardware, the hardware assumes full control from that point while the software goes on to other business.

The remainder of this section describes the hardware I/O system. (For a more detailed explanation of the hardware logic, refer to Section VIII.)

### HARDWARE ELEMENTS

Separately identifiable hardware elements are: the I/O Processor (IOP), multiplexer channel, selector channel, device controller, and peripheral device. With reference to figure 6-9, the following paragraphs define the basic functions of each of these elements.

The *I/O Processor* has three basic functions, relating to the three different transfer modes illustrated in figure 6-9. In the case of direct I/O, the IOP executes the direct I/O instructions (RIO, WIO, TIO, CIO, SIN and SMSK), transferring data, device status, and control information between the CPU and a device controller. In the case of programmed I/O via a multiplexer channel, the IOP transfers I/O program words between memory and the multiplexer channel, and data between memory and the controller. In the case of programmed I/O via a selector channel, the IOP passes initialization information to the device controller; the IOP does not become involved in any part of the I/O program execution. The IOP also interrupts the CPU on behalf of the device controllers.

The *multiplexer channel* acts as a switch to enable one of 16 device controllers to transfer one word of data to or from memory via the IOP, then to allow another controller – based on priority – to perform its transfer. At all times, the multiplexer channel contains the current I/O program word for each of the 16 device controllers. To accomplish this, the multiplexer channel has a 16-location solid-state memory to contain the 16 I/O program words, and is responsible for updating the contents and fetching the next I/O program word when necessary.

The *selector channel* also acts as a switch but in a manner different from a multiplexer channel. Whereas the multiplexer switches between controllers on demand, based on hardware priority, the selector channel maintains the connection for one controller until it has completed the I/O program. Thus only one I/O program is current at a given time for one channel. Another major difference, as shown in figure 6-9, is that the selector channel accesses memory directly for data and I/O program word transfers, rather than indirectly through the I/O Processor. These features permit a very high speed data transfer rate.

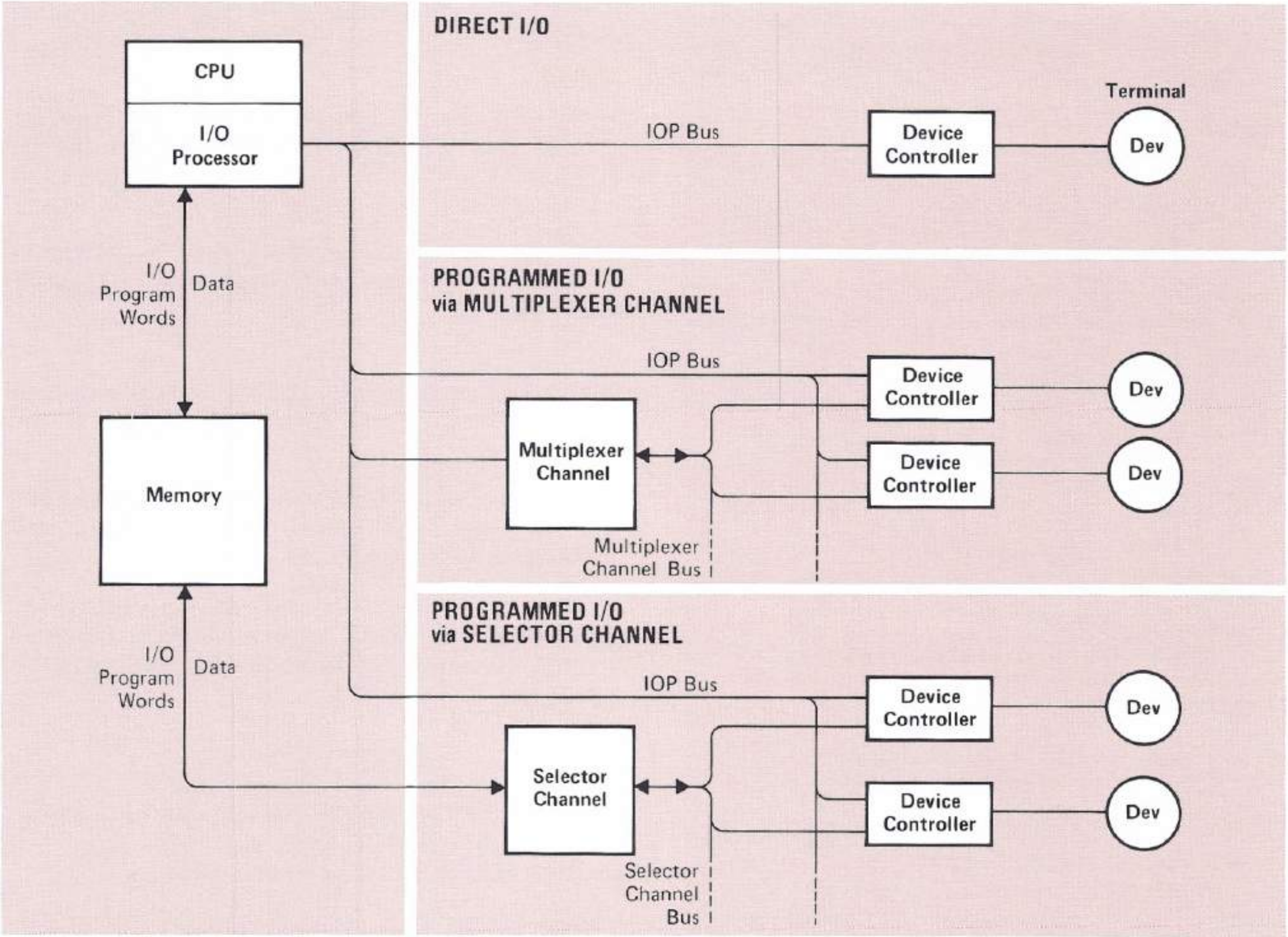


Figure 6-9. Hardware I/O Elements

The device controller is the interface between a peripheral device and the computer system. Its primary function is to translate programmed I/O commands from a multiplexer or selector channel (or direct I/O commands from the I/O Processor) to the unique signals required to control a particular device. When an I/O program is in execution, the device controller responds to and requests service from the channel. The device controller also generates interrupts when required by some device condition or by direct or programmed command.

The *peripheral device* receives output data for storage or display, or supplies input data to the computer. In general, one device controller controls one peripheral device; however, some controllers are capable of controlling several devices.

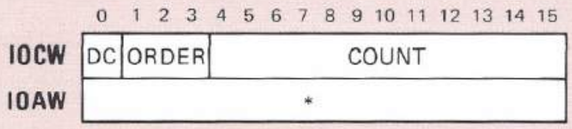
## I/O PROGRAMMING

The I/O program, as shown earlier in figure 6-5, is a part of a device driver and is uniquely assembled for each I/O request from the file system. Once the driver issues an SIO instruction to the requested device controller, the hardware I/O system begins to execute the I/O program — independently of the CPU. The CPU is then free to continue processing in parallel with the I/O operations.

The following paragraphs define the elements of an I/O program and describe the actions occurring after the SIO instruction is issued to the hardware.

I/O PROGRAM WORD. Figure 6-10 illustrates the format of the *I/O program word.* Two computer words are used to accommodate the 32-bit word length. The first word is designated as the *I/O Command Word,* or IOCW, and the second word is designated as the *I/O Address Word,* or IOAW. The IOAW does not necessarily always contain an address, as indicated in the figure.

*Data chaining* occurs for WRITE and READ orders if bit 0 of the IOCW is a "1". This bit may be a "1" for a WRITE order followed by a WRITE or for a READ order followed by a READ. This will permit the hardware to treat the counts of each order as a continuous chained count, without reinitializing for each order. The DC bit should be "0" for all other orders.



| | | |
|---|---|---|
| *Data Address | for | WRITE<br>READ |
| *Jump Address | for | JUMP |
| *Control Information | for | CONTROL |
| *Device Status | for | SENSE<br>END |
| *Residue Count | for | RETURN RESIDUE |
| *Unused | for | INTERRUPT |

DC = Data Chain
ORDER = I/O command code
COUNT = Word Count

**ORDER CODES**

| | | |
|---|---|---|
| 000 | JUMP, | Conditional if IOCW(4) = 1<br>Unconditional if IOCW(4) = 0 |
| 001 | RETURN RESIDUE | |
| 010 | INTERRUPT | |
| 011 | END, | with interrupt if IOCW(4) = 1<br>without interrupt if IOCW(4) = 0 |
| 100 | CONTROL | |
| 101 | SENSE | |
| 110 | WRITE | |
| 111 | READ | |

Figure 6-10. I/O Program Word Format

The count field of the IOCW contains a two's complement negative count value for WRITE and READ orders. The count is a word count, independent of the particular recording format (bytes, words, or records). For a CONTROL order, these 12 bits are used for control information in addition to the 16 control bits in the IOAW (a total of 28 bits).

The eight I/O orders are defined as follows:

JUMP. If bit 4 of the IOCW is a "1", a conditional jump of I/O program control is made to the address given by the IOAW at the discretion of the device controller. If bit 4 of the IOCW is a "0", an unconditional jump is made.

RETURN RESIDUE. This causes the residue of the count to be returned to the IOAW. The residue is obtained from the multiplexer or selector channel.

INTERRUPT. This causes the device controller to interrupt the CPU.

END. End of the I/O program. If bit 4 of the IOCW is a "1", the device controller also interrupts the CPU. Returns device status to the IOAW.

CONTROL. This causes transfer of a 16-bit control word in the IOAW to the device controller, as well as the 12-bit count field.

SENSE. This causes transfer of a 16-bit status word from the device controller to the IOAW.

WRITE. This causes "count" words of data to be transferred between main memory and the device, starting at the address given by the IOAW.

READ. This causes "count" words of data to be transferred between the device and main memory, starting at the address given by the IOAW.

TYPICAL I/O PROGRAM OPERATION. Figure 6-11 shows the sequence of operations occurring as the result of an SIO instruction. The sequence is as follows.

① The SIO instruction, decoded by the CPU, fetches the
② device number given at S-K in the stack, and puts the TOS into the first word of the DRT as the I/O program pointer.

③ SIO then loads the device number into the eight least
④ significant bits of the IOP Control Register, and loads an SIO command into bits 1, 2, and 3.

⑤ The I/O Processor issues the SIO command to the device controller, and execution by the hardware begins. The CPU is now free to continue execution elsewhere.

⑥ On demand from the multiplexer channel, the I/O Processor obtains the program pointer from the Device Reference Table. (The selector channel obtains the program pointer directly, not via the IOP.) As shown earlier (figure 6-4), the address is obtained by multiplying the device number by four. The program pointer is the first word of the four-word DRT entry.

⑦ The program pointer points to the first double word of the I/O program. The pointer is updated to point at each I/O program double word as the program progresses. (The selector channel, to minimize memory fetches, copies the pointer value into a register and updates the pointer internally; the multiplexer channel, however, updates the pointer directly in the DRT.)

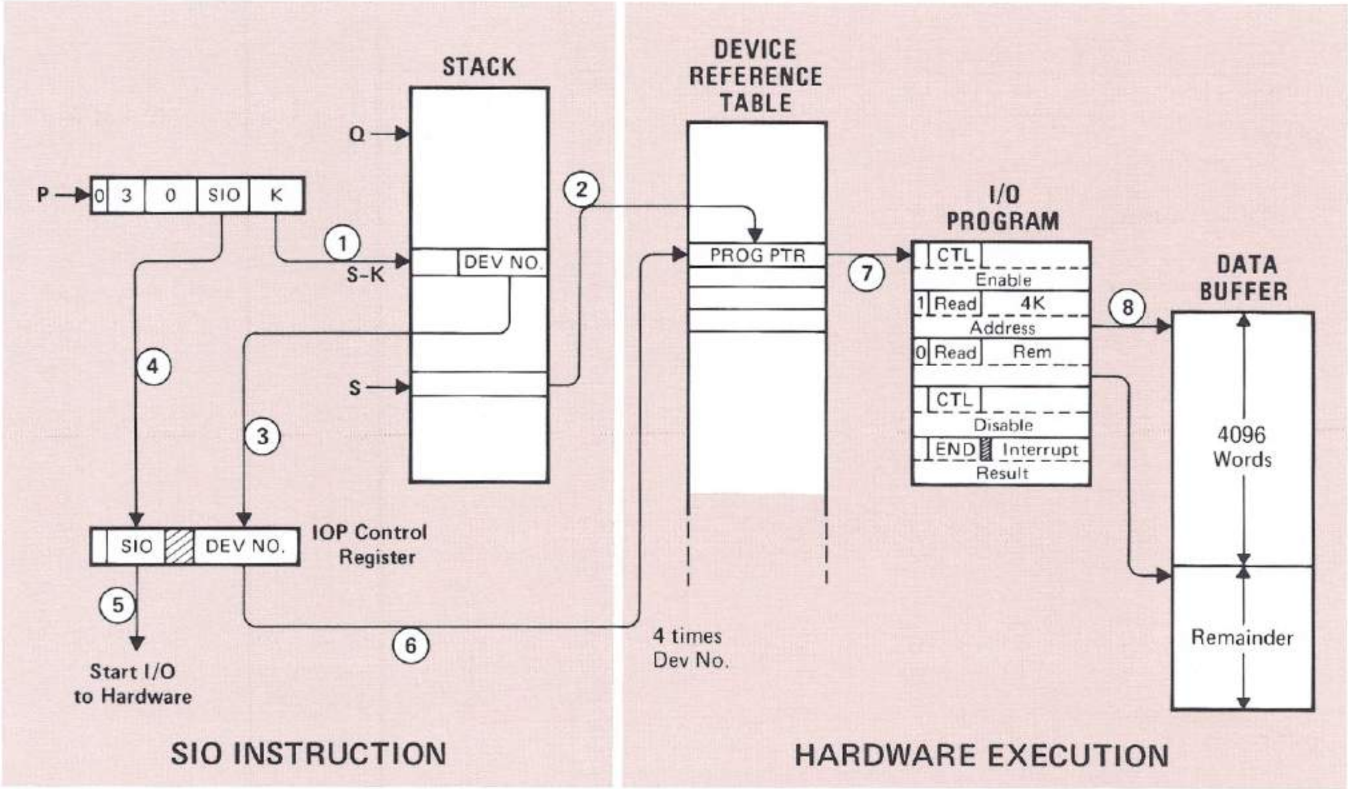


Figure 6-11. I/O Program Operation

(8) The sample I/O program is assumed to operate as follows. The first double word contains a CONTROL order which enables the hardware I/O subsystem for this device number. The second double word contains a READ order, which causes the subsystem to read 4096 words (or 8192 bytes) into the data buffer whose starting location is given in the IOAW word. Since the data chaining bit is on, the next (third) double word is also a READ order, which specifies the remaining count required to fulfill the I/O request. (Additional READ orders could be given for larger requests.) The IOAW may specify a buffer area contiguous to the first 4096-word buffer if desired, or in another part of memory if a *scatter read* is desired.

When the transfer is complete, the fourth double word, a CONTROL order, turns off the I/O subsystem. The final double word contains an END order, which obtains the result of the transfer (device status) and loads it into the IOAW; the END order then generates an interrupt to inform the software that the transfer is complete.

At the completion of an I/O program, the selector channel returns the current program pointer value to the DRT. The multiplexer does not take any special action since it updates the DRT after each order fetch.

The interrupt system is designed to conform with the basic architectural scheme of the HP 3000. Thus, interrupt routines are called and exited in a manner resembling the way that procedures are called and exited. An interrupt is therefore an implicit PCAL (vs. explicit PCAL instruction). Also, code and data domains are kept separate.

The primary difference is that the calling operations are performed by a microprogrammed *Interrupt Handler* rather than by the PCAL instruction. For exit, however, the same EXIT machine instruction is used as for exiting from a procedure.

The first 16 entries in the Code Segment Table (CST) are always devoted to defining the code domains of interrupts. CST entries 1 through 16 define the code for internal interrupts and CST entry 0 is the default segment number for external interrupts. Table 7-1 lists all interrupts according to code segment numbers.

Code segmentation for external interrupts is performed by the Device Reference Table. The default segment number of 0 is retained in the Segment Number field of the Status register while processing external interrupts. This tells both hardware and software that an external interrupt is being processed.

The "parameter" is a value that is derived by the Interrupt Handler, and passes to the interrupt routine relevant information about the interrupt — such as to identify the source or type of an error.

Before discussing the various interrupt types, the Interrupt Control Stack will first be defined, since it will be referred to frequently throughout the succeeding descriptions.

## INTERRUPT CONTROL STACK

The *Interrupt Control Stack* (ICS) is a single stack, unique to one CPU, which is used in common by all external interrupts and some of the internal interrupts ("ICS type"). When only minimal data is to be handled by an interrupt routine, the data is processed on the ICS. (Otherwise, the separate data area defined in the DRT must be used for data.) Use of a common stack also permits efficient nesting of interrupt routines, via stack markers.

The ICS has a permanent stack marker (set up by the operating system) which is used for exiting to the Dispatcher. This guarantees that the final routine to use the ICS will always exit to the Dispatcher. Figure 7-1 illustrates the format of the Dispatcher marker on the ICS.

Note that, unlike the standard four-word stack marker, the Dispatcher marker contains five words. As will be explained later, all markers on the ICS (as well as the marker left on the previous stack before switching to the ICS) include a fifth word to save the current value of DB. The reason for saving DB is that all external interrupts automatically alter DB (to the DBI value); also, interrupt routines for ICS-type internal interrupts may also change DB. The EXIT instruction restores DB.

The delta Q location of the Dispatcher marker always contains a "0" word. Since the Dispatcher does not change Q until a new process is dispatched, a specific value for Q is not needed. Instead, the "0" value tells the hardware not to delete the marker when exiting from the ICS, and to set the *Dispatcher Flag* in the CPU. The Dispatcher Flag is set whenever an exit is made from the ICS to the Dispatcher, and remains set while the Dispatcher is executing. It is cleared when the Dispatcher completes its execution, or is aborted by another interrupt.

The segment-number field of the Status word permanently points to the CST entry for the Dispatcher, and the "P - PB" word permanently points to the starting point in the Dispatcher code segment. The EXIT instruction uses these values for transferring control to the Dispatcher.

The locations preceding the Dispatcher marker comprise the ICS global area, which contains operating system information set up or to be acknowledged by the Dispatcher. The location following the Dispatcher marker is used for the parameter by those interrupts that do pass a parameter (refer to table 7-1). Note that since ICS-type interrupts use a five-word marker, the parameter is found in location Q+2, rather than the usual Q+1 location.

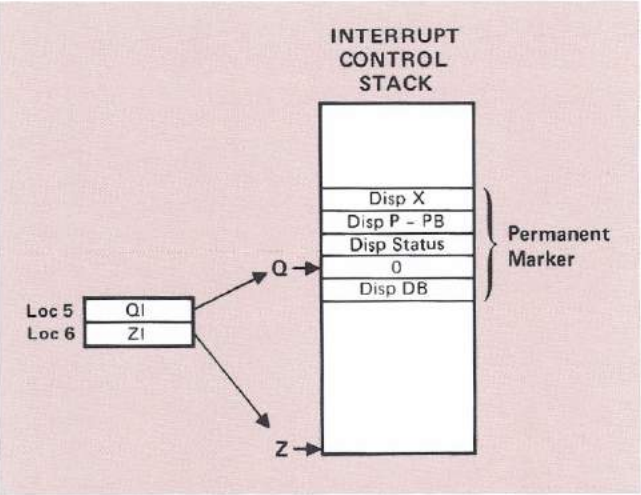


Figure 7-1. Dispatcher Marker on ICS

Table 7-1. Interrupts and Traps

| *CODE SEG | TYPE | | | PARAMETER |
|---|---|---|---|---|
| 0 | External Interrupts (via DRT) | | On ICS at Q+2 | Device No. |
| 1 | Power Fail | Internal ICS-type | | |
| 2 | Power On | | | |
| 3 | Stack Overflow | | | |
| 4 | Module Interrupt | | | Module No. |
| 5 | Console Interrupt | | | CPU No. |
| 6 | Cold Load | | | |
| 7 | Unassigned | | | |
| 10 | Unassigned | | | |
| 11 | Module Error | | On Current Stack at Q+1 | |
| | Illegal Address | | | 1000* |
| | Bounds Violation | | | 2000 |
| | Non-responding Module | | | 4000 |
| 12 | Parity Error | | | |
| | Data Parity Error | | | 10000 |
| | Memory Address Parity Error | | | 20000 |
| | System Parity Error | | | 40000 |
| 13 | Miscellaneous Error | | | |
| | Stack Underflow | | | 1 |
| | CST violation | | | 2 |
| | STT violation | | | 3 |
| 14 | Code Segment Absence | | | |
| | If PCAL | | | Label |
| | If EXIT | | | N |
| 15 | STT Entry Uncallable | | | Label |
| 16 | Trace | | | |
| | If PCAL | | | Label |
| | If EXIT | | | N |
| 17 | Traps | | | |
| | Integer Overflow | User Traps | | 1 |
| | Floating Point Overflow | | | 2 |
| | Floating Point Underflow | | | 3 |
| | Integer divide by 0 | | | 4 |
| | Floating Point divide by 0 | | | 5 |
| | Mode violation | System Traps | | 6 |
| | Unimplemented instruction | | | 7 |

*Octal numbers

A hardware *ICS Flag* is set in the CPU whenever a switch is made to the ICS from any other stack. The ICS Flag remains set until another process is dispatched and the ICS is no longer the current stack.

Figure 7-1 also shows the delimiting of the ICS by QI and ZI ("interrupt" Q and Z). These values are given in fixed memory locations 5 and 6 for CPU number 1 or locations 11 and 12 (octal) for CPU number 2, if used. The QI value points to the delta Q location of the Dispatcher marker on the ICS. The ZI value points to the ICS stack limit.

Privileged software may gain access to the ICS by loading an absolute value into the Z-register which is equal to ZI. This is accomplished by a SETR Z instruction with a ZI – DB relative value on the top of the stack. (SETR Z will add DB to the relative value before checking if the result is equal to ZI.) This action will set the ICS Flag. Note: S and Q must also be set to appropriate ICS values by the same SETR instruction, or a stack overflow is likely to occur.

## INTERRUPT TYPES

Interrupts may be divided into two basic types: *external interrupts*, which are controlled signals from the I/O system, and *internal interrupts*, which typically are unexpected signals caused by certain hardware conditions or programming violations.

The HP 3000 system characteristics necessitate splitting each of these two basic types, resulting in the following four types:

- External interrupts (from standard I/O devices)
- IRF-type external interrupts (from non-standard I/O devices, using Interrupt Reference Flag)
- ICS-type internal interrupts (using Interrupt Control Stack)
- Non-ICS internal interrupts

A *standard device* is one which is known to, and controllable by, the file system of the MPE/3000 operating system. A *non-standard device* is independent of the file system (such as a real-time device), and uses the *Interrupt Reference Flag* (IRF) to make its interrupt known to the operating system.

Figure 7-2 compares the overall operations of all four interrupt types. Taking a general view of this figure, note that operations proceed mostly left-to-right. For example, external interrupts begin by triggering some actions in hardware, then the interrupt processing environment is set up in software, and finally there is an exit to the *Dispatcher*. The Dispatcher is the part of the operating system which schedules the execution of processes.

Note that three of the four types of interrupts exit to the Dispatcher. (The same three use the Interrupt Control Stack for a data domain.) The action of exiting to the Dispatcher, instead of returning to the point of interrupt, permits the operating system to re-evaluate process priorities. Remember that in the case of external interrupts (refer back to figure 6-2), the process that caused the I/O request has been inactive while the hardware I/O system is transferring data. Another process would be running at the time of the interrupt. The interrupt essentially means: re-activate the monitor process so the I/O request can be completed. Since the Dispatcher decides which process is activated next, the Dispatcher is the logical point of return.

Note: It is assumed here that only one interrupt is being processed. As will be shown later, interrupt routines can be interrupted by other interrupts, and the exit to the Dispatcher occurs only when making the final exit on the Interrupt Control Stack.

For IRF-type interrupts, the meaning of the interrupt is even simpler — i.e., activate a certain process (via the Dispatcher). For ICS-type internal interrupts, the Dispatcher return is still necessary, since the priorities of processes may have changed while operating on the Interrupt Control Stack, as a consequence of functions performed thereon.

One of the important features of the Dispatcher is that it can be aborted at any time during its operation. Thus if a new interrupt arrives while the Dispatcher is in operation, the new interrupt can immediately be handled without having to restore any particular conditions. This feature maintains the fast interrupt response of the system in multi-interrupt situations.

For non-ICS internal interrupts, an exit to the Dispatcher is not necessary since these operate on the current user's stack. The exit can be made directly back to the point of interrupt.

Figure 7-2 also shows that the interrupt routine code is accessed via the Device Reference Table for external interrupts (both types) and via the Code Segment Table for internal interrupts (both types). This is because internal interrupts are processed by specifically assigned code segments which must always be present in main memory (or the system will halt). External interrupts, on the other hand, are device-related and so code segmentation is done by the Device Reference Table. Accordingly, an internal interrupt routine is an *interrupt code segment* and an external interrupt routine is called the *interrupt receiver code*.

All external interrupt routines, by definition, execute in privileged mode. The Interrupt Handler automatically sets

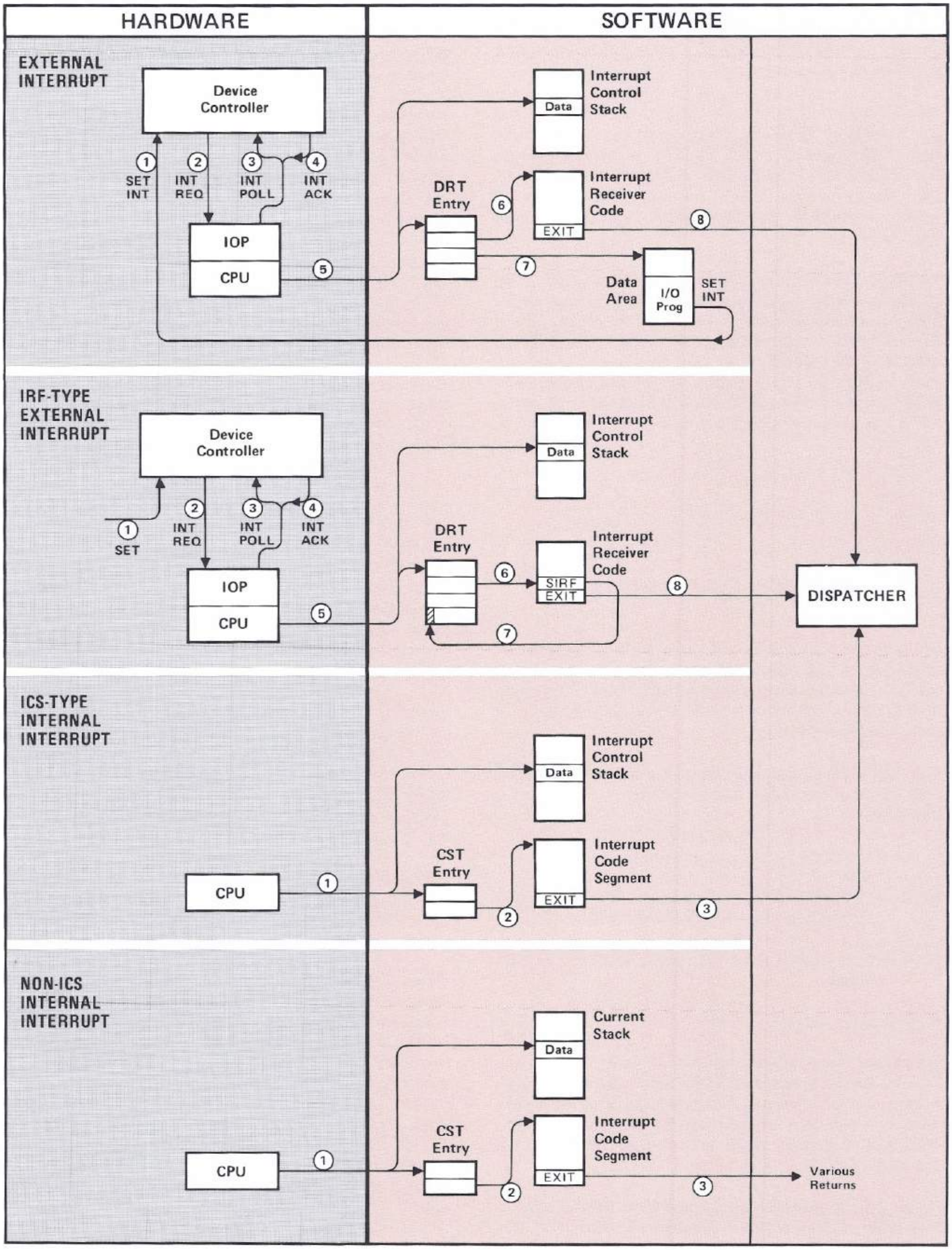


Figure 7-2. Interrupt System Overview

the Mode bit in the Status register to the privileged mode state before transferring control to the interrupt routine. For internal interrupts, however, the mode bit in the CST entry determines which mode is to be used during execution.

All external interrupt routines are entered with the external interrupt system enabled. All internal interrupt routines are entered with the external interrupt system disabled.

The following paragraphs individually describe each of the four interrupt types. Only a brief introductory description is given at this point. Detailed operating sequences are given later in this section.

## EXTERNAL INTERRUPTS

External interrupts interface external events to software processes. Referring to figure 7-2 (top example), the overall operation is as follows:

(1) At or near the end of an I/O program, the device controller decodes a SET INT (Set Interrupt) command, which causes the controller to set its Interrupt Request flip-flop. This action corresponds to step 10 in the I/O System Overview, figure 6-5.

Note: The device controller's Interrupt Request flip-flop can also be set by an SIN instruction decoded by the CPU. However, such action is more commonly used in diagnostic routines than in conventional I/O operations.

(2) The setting of the Interrupt Request flip-flop causes the device controller to issue an INTREQ (Interrupt Request) signal to the I/O Processor – provided that a previously issued mask permits requests from this controller. (Masks will be discussed later.)

(3) The I/O Processor issues a poll (INTPOLL) to activate the highest-priority request. (There may be more than one request.)

(4) The device controller returns an acknowledgement (INTACK), along with its device number.

(5) The IOP requests the CPU to set up the interrupt environment. The initial steps are to set up the data segment registers to point at the Interrupt Control Stack (after saving the user's environment on his own stack) and to fetch the device's DRT entry.

(6) The address in the second word of the DRT entry is loaded into the P-register, thus transferring control to the interrupt receiver code.

(7) The information in the data area for this device (pointed to by the third word of the DRT) is updated by the interrupt receiver. This information will tell the I/O monitor process that the initiator section of the device driver has done its work, and the completion section should be called.

(8) The interrupt receiver exits to the Dispatcher. This action corresponds to step 12 in the I/O System Overview, figure 6-5.

## IRF EXTERNAL INTERRUPTS

IRF-type external interrupts provide an external means of activating a process. One action of this interrupt is to set the Interrupt Reference Flag associated with a particular device controller. It is then up to the operating system to connect the process to the device and begin execution. Referring to the second example in figure 7-2, the overall operation is as follows:

(1) An external event sets the Interrupt Request flip-flop in the device controller. This could be as simple as a contact closure.

(2) The device controller issues an Interrupt Request to the I/O Processor, provided that a previously issued mask permits requests from this controller.

(3) The I/O Processor issues an interrupt poll to activate the highest-priority request.

(4) The device controller returns an acknowledgement along with its device number.

(5) The IOP causes the CPU to switch to the ICS (after saving the user's environment on his own stack), and to fetch the device's DRT entry.

(6) The address in the second word of the DRT entry is loaded into the P-register, thus transferring control to the interrupt receiver code.

(7) An SIRF (Set Interrupt Reference Flag) instruction in the interrupt receiver sets the IRF bit (bit 0) of the fourth word in the DRT entry.

(8) The interrupt receiver exits to the Dispatcher. (The association of the external interrupt and a specified process is performed by the Dispatcher, using software table information and the SIRF bit of the DRT.)

## ICS INTERNAL INTERRUPTS

ICS-type internal interrupts operate on the Interrupt Control Stack, and the interrupt code for each separate interrupt is permanently allocated in code segments 1 through 7 (see table 7-1). Referring to the third example in figure 7-2, the overall operation is as follows:

① An internal hardware condition (due to power failure, stack overflow, module interrupt, or console interrupt) causes the CPU to switch to the ICS (after saving the user's environment on his own stack), and to fetch the Code Segment Table entry for the specific interrupt (1 through 7).

② The absolute address in the second word of the CST entry is loaded into the PB-register, and execution begins with P = PB. This transfers control to the appropriate segment.

③ After processing the interrupt, the code segment exits to the Dispatcher.

### NON-ICS INTERNAL INTERRUPTS

The non-ICS type interrupts operate on the current user's stack. The interrupts caused by this type are generally caused by errors in a user's process, or by the system while executing a user's process. Code segments 10 through 17 are permanently allocated for processing these interrupts. Referring to figure 7-2, the overall operation is as follows:

① An error in the execution of a user's process causes the CPU to save the user's environment on his own stack and to fetch the CST entry for the specific interrupt (10 through 17).

② The absolute address in the second word of the CST entry is loaded into the P-register, thus transferring control to the interrupt segment.

③ After processing the interrupt, the code segment exits back to the point of interrupt in the user's process. In the case of serious errors, another procedure may be called to abort the offending process, or, for irrecoverable errors, a system halt will occur.

## EXTERNAL INTERRUPT PROCESSING

Before discussing the sequence of operations for external interrupts, there are three important factors that need to be considered. These are: interrupt priorities, the interrupt mask, and interrupt program pointers.

### INTERRUPT PRIORITIES

Servicing of external interrupts is done in descending order of priority. That is, the highest priority interrupt is serviced first. A higher priority interrupt can always interrupt the processing of a lower one.

The interrupt priority of a device is completely independent of the device number and interrupt masking. It is determined by the device's logical proximity to the IOP on the interrupt poll line. The interrupt poll is wired at system configuration time from one device controller to another, using twisted-pair clip-on wires. An illustration of how an interrupt poll might be wired is shown in a later section (figure 8-7). The routing of the interrupt poll is determined by the desired interrupt priorities of the device controllers, and is completely independent of other parameters.

Each device controller therefore has a distinct priority level in relation to all other controllers. The maximum number of controllers, and hence interrupt levels, is 253.

### INTERRUPT MASK

The *mask* is a word of 16 bits which, when transmitted to the I/O system, will enable or disable the interrupt request logic of certain groups of device controllers, according to the bit pattern of the word. A logic "1" in a given bit position will enable the corresponding group of interrupts; a logic "0" will disable the group.

The mask is originally created as a word on the top of the stack. From there it is transmitted to all device controllers simultaneously by a SMSK (Set Mask) instruction. Each device controller is wired (at system configuration time) to respond to one particular bit in the mask word. Thus when the mask is transmitted by SMSK, the Mask flip-flop in each device controller will either set or reset according to the value of the bit to which the controller is sensitive.

Assuming that all controllers do accept the new mask, the SMSK instruction will also load the mask word into the Mask register in the CPU. This makes it possible to check the value of the existing mask at any time by reading it to the top of the stack by a RMSK (Read Mask) instruction. Note that RMSK is actually reading a copy of all the mask bits and not the real Mask bits in all the devices. RMSK is a non-privileged instruction; SMSK, however, is a privileged instruction.

If there is a hardware failure on the Power Bus Terminator card and it does not issue a Mask Return signal, then the mask word is retained on the TOS, the Condition Code is set to CCL to indicate an I/O error, and the external interrupt system is disabled. Note that the state of the individual Mask flip-flops on the device controllers are in an unknown state in this case.

Figure 7-3 illustrates an example of interrupt masking. In this example, if bit 2 of the transmitted mask word is a

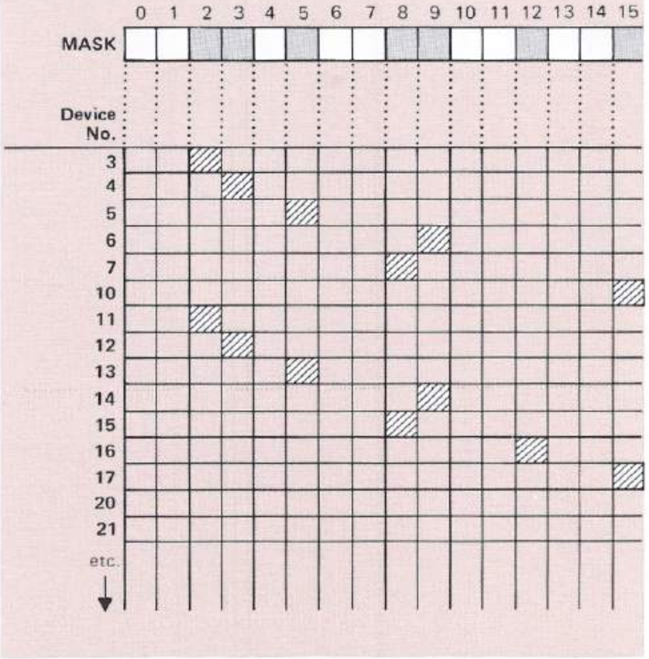


Figure 7-3. Interrupt Masking

logic "1", interrupts will be permitted from devices 3 and 11; if bit 2 is a "0", devices 3 and 11 will not be able to make interrupt requests. Similarly, bit 3 controls the interrupts from devices 4 and 12, bit 5 controls the interrupts from devices 5 and 13, and so on.

## INTERRUPT PROGRAM POINTER

The Device Reference Table was defined in Section VI. As stated then, the second word of each DRT entry contains the interrupt program pointer. This is an absolute address pointing to the start of the interrupt routine associated with a particular device controller. See figure 7-4. Note that several controllers could point to the same routine.

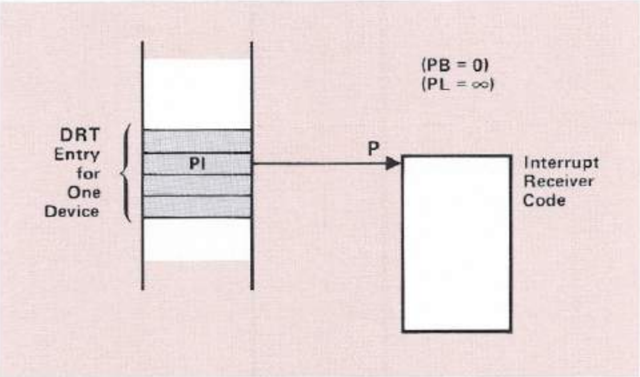


Figure 7-4. Interrupt Program Pointer

## SEQUENCE OF OPERATIONS

Figures 7-5 and 7-6 illustrate the sequence of operations for processing external interrupts. Basically, we are narrowing the scope of the overall I/O operation to focus on just the portion that establishes the interrupt processing environment on receipt of an external interrupt. In previous figures, this corresponds to steps 11 and 12 in figure 6-5, and to steps 5, 6 and 7 in figure 7-2.

Figure 7-5 shows how control is transferred from the point of interrupt in a user's code segment to the start of the interrupt receiver code. Also shown is the transfer of the data domain from the current user's stack to the interrupt control stack. Figure 7-6 shows how a second interrupt is handled and how exit is made from the interrupt routines.

The following paragraphs describe the sequence of operations, step by step. Note first the Dispatcher marker in the Interrupt Control Stack; the contents are not detailed since they were discussed under a previous heading. Note also that all operations are under control of the hardware-implemented Interrupt Handler until control is transferred to the interrupt receiver code in software.

The initial assumption is that the current process is operating at point P in some user's code when the CPU recognizes an external interrupt. The CPU thereupon passes control to the Interrupt Handler.

(1) The first action of the Interrupt Handler is to push into memory any TOS elements of the current user's data that are in CPU registers. This takes a maximum of four memory cycles if all four registers are full.

(2) Next, a normal four-word stack marker is pushed onto the user's stack, plus the absolute value of DB that is currently in use. (DB may not necessarily point to a location within the stack, such as if a system intrinsic had been called at the time of the interrupt.) This action preserves most of the user's environment; the current value of S will be preserved later (refer to step 5). Incidentally, DL is never changed by an interrupt.

(3) The Interrupt Handler now goes to location 5 (assuming CPU #1) and loads the QI value into the Q-register. This points Q at the delta Q location of the permanent Dispatcher marker. (As explained previously, this location contains a value of 0.)

(4) The content of location 6 is next fetched and the value of ZI is loaded into the Z-register. This establishes the stack limit for the Interrupt Control Stack. (The ICS Flag is also set by this action in hardware.)

(5) The user's current absolute value of S is stored into location Q-5 on the Interrupt Control Stack. (Of course, S will by this time be pointing at the top word of the marker on the user's stack.) Later, when control is passed to the Dispatcher in step 18, the Dispatcher will convert S to a relative value by subtracting the DB

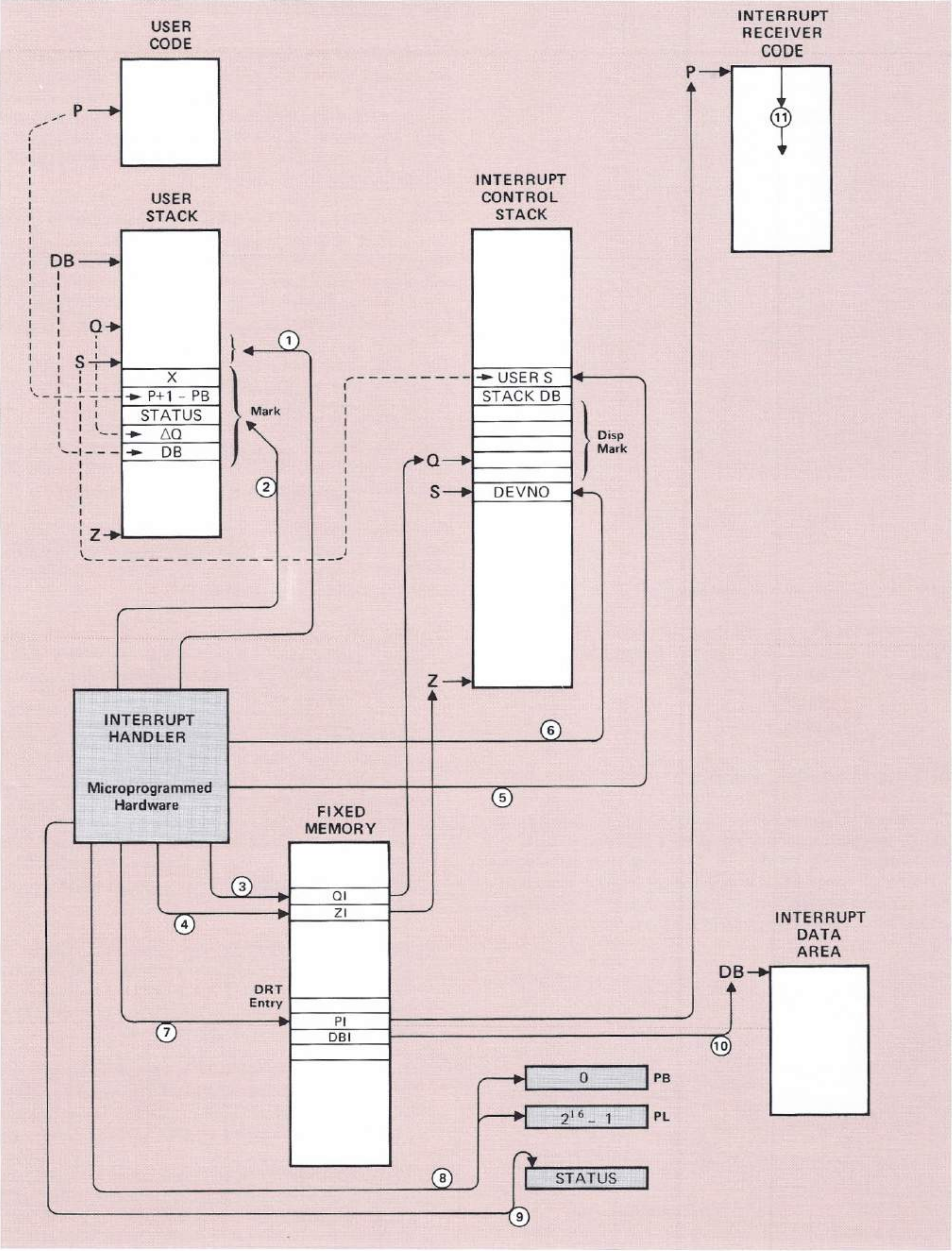


Figure 7-5. External Interrupt Operations

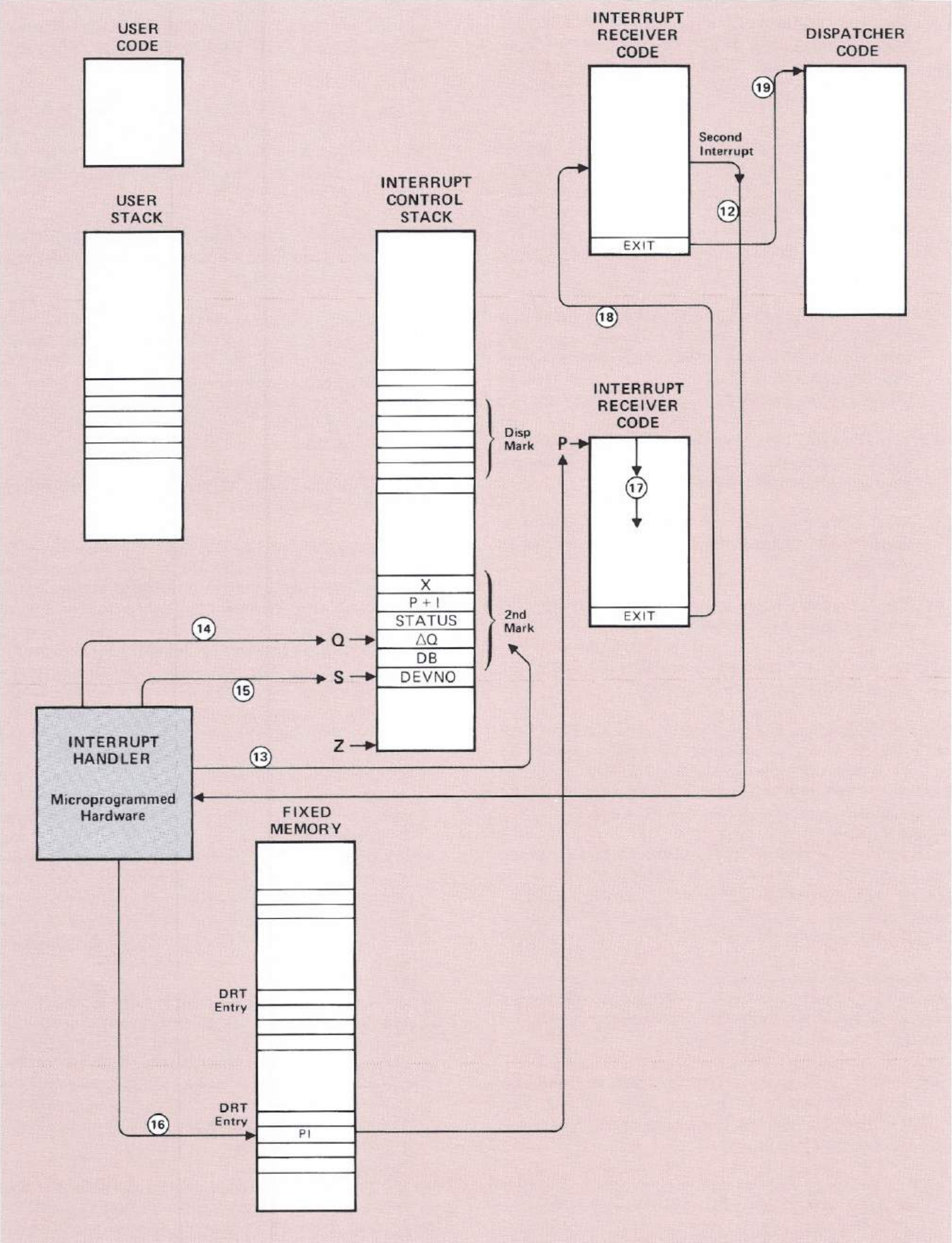


Figure 7-6. Second External Interrupt

value of the user's stack, and will store the result in the user's environment. (The "stack DB" is always saved in ICS location Q–4 by the Dispatcher, prior to launching a process — in case DB is altered during the process.) This action will complete the preservation of the user's pre-interrupt environment.

⑥ The S pointer is set to point at location Q+2, and the device number of the interrupting device is stored into this location. The CPU obtains this number from the Interrupt Address register in the I/O Processor. At this point, the Interrupt Control Stack is fully delimited by register values, and is ready for handling interrupt data.

⑦ The Interrupt Handler now uses the device number on the ICS to form an address for fetching the second word of the DRT entry for that device. The content of that location (PI) is loaded into the P-register, thus pointing to the start of the interrupt receiver code.

⑧ The PB- and PL-registers are set to their respective extreme values (PB = 0, PL = $2^{16}$ - 1) since they are not used for delimiting interrupt code.

⑨ Bit 0 of the Status register is set to a "1" so that, as required, the interrupt receiver code will execute in privileged mode.

⑩ The DB-register is set to the value of DBI, the third word in the device's DRT entry.

⑪ The CPU now fetches the instruction at P and begins executing the interrupt receiver code.

The following steps, relating to figure 7-6, list the actions occurring if a second interrupt (of higher priority, of course) is received while processing the first interrupt. Assuming a still higher priority, another interrupt could interrupt the second routine in the same manner as described below. This example shows how several levels of interrupts can be nested on the Interrupt Control Stack. Since the ICS is common to all external interrupts, no further switching of environments is necessary for additional interrupts. As mentioned in Section I, this reduces the interrupt response time by about two microseconds.

If, however, the second interrupt did not occur before completing the processing of the first interrupt, the sequence of operations would skip from this point (step 11) to step 19. The sequence continues as follows:

⑫ The CPU recognizes a second interrupt while executing the interrupt receiver code for the first interrupt. The CPU therefore again passes control to the Interrupt Handler.

⑬ The Interrupt Handler pushes into memory any TOS elements that are in CPU registers, and pushes the usual five-word marker onto the ICS. The fifth word of this marker is the DB value that is currently in the DB-register at the time of interrupt.

⑭ The Q-register is updated to point at the delta Q word of the new marker. The delta Q value is the number of locations back to the delta Q word of the Dispatcher marker.

Note: Unlike the first interrupt, subsequent interrupts do not store S into Q–5 at this point (see step 5). Such action would overlay one of the variables associated with the previous interrupt.

⑮ The S-register is updated to point at location Q+2, and the device number of the second interrupting device is stored into that location.

⑯ The Interrupt Handler uses the device number to fetch the second word of the DRT entry for that device. The content of that location (PI) is loaded into the P-register, thus pointing to the start of the interrupt receiver code for the second device. Also (not shown), the DBI value from the new DRT entry is loaded into the DB-register.

⑰ The CPU now fetches the instruction at P and begins executing the interrupt receiver code.

⑱ Assuming there are no other higher priority interrupts, the interrupt routine for the second device runs to completion and then exits to the point of interrupt in the interrupt routine for the first device. The exit, as usual, is made via the stack marker. Note that since PB = 0 while operating on the ICS, the "return P" (second word) of the marker is an absolute as well as a relative address value. The Q value is restored to the previous setting, pointing to the delta Q word of the Dispatcher marker. The S pointer is moved back to the location just preceding the second stack marker. The N field of the EXIT instruction must always be 0 for exiting from interrupt routines, so that none of the variables associated with the previous routine will be deleted by the act of moving S back. One of the actions of the EXIT instruction is to issue a Reset Interrupt command to the interrupting device controller, which clears the interrupt active condition and unblocks the interrupt poll line to lower priority devices. (The device number is obtained from location Q+2.)

⑲ The interrupt receiver code for the first interrupt now runs to completion and an exit is made to the Dispatcher. Again, the EXIT instruction issues a Reset Interrupt command to the device controller. In this case, however, the stack marker is not deleted, and the hardware sets the Dispatcher Flag to signify that the Dispatcher is now executing. This completes the sequence of operations.

If another external interrupt should occur while the Dispatcher is executing, the interrupt is treated in a slightly different way. If the CPU recognizes an interrupt while the Dispatcher Flag is set (from step 19), the sequence effectively repeats steps 12 through 17 with the added actions that, in step 14, bit 0 of ΔQ is set to 1 (indicating a Dispatcher interrupt) and the Dispatcher Flag is cleared. Then the interrupt receiver code can optionally set bit 0 of the Interrupt Counter (fixed location 7) to a 0 if no processing is required (e.g., in the case of character interrupts from a terminal, which should not require aborting the Dispatcher), or to a 1 if it wishes to process the interrupt (such as for a carriage return interrupt). In either case, when the EXIT instruction is given, the sequence goes to step 19. As explained in the EXIT instruction commentary, EXIT will allow the Dispatcher to continue execution from the point of interrupt if bit 0 of the Interrupt Counter is clear, or aborts the Dispatcher (i.e., eliminates progress made prior to the interrupt by setting Q = QI, thus eliminating the interrupt stack marker and all Dispatcher data) if bit 0 of the Interrupt Counter is a 1. In the latter case, the Dispatcher will be restarted.

If, however, the Dispatcher is allowed to run to completion, the CPU will clear the Dispatcher Flag when the Dispatcher sets the Z-register to some value other than ZI. (This is one of the last actions of the Dispatcher.)

## IRF INTERRUPT PROCESSING

Normally, a process calls I/O which in turn causes interrupts. *IRF interrupts*, however, reverse the situation — that is, the interrupt calls a process. This is necessary because the devices that use the Interrupt Reference Flag are not known to the file system, which normally handles all I/O requests. Thus the IRF interrupt must inform the Dispatcher of its occurrence, so that the Dispatcher can activate the process which is associated with the interrupting device. That process may then use the device directly, bypassing the file system.

In order to maintain some control over non-standard devices, the operating system has control of "arming" the IRF bits in the DRT entries.

The operating sequence for an IRF interrupt begins as a normal external interrupt, subject to priority and masking. Thus the first eleven steps are exactly the same as described for external interrupt processing (i.e., all of the steps shown in figure 7-5). That part of the sequence takes the operation up to the point of beginning the execution of the interrupt receiver code. The sequence then continues as follows, with reference to figure 7-7.

⑫ An SIRF instruction in the interrupt receiver code sets (to logic "0") the IRF bit in the DRT entry for the interrupting device controller. If the bit already was in the "0" state, the SIRF instruction is treated as a NOP; the next instruction (EXIT) causes an exit to the Dispatcher with no further effects.

⑬ Assuming that the IRF bit was previously armed (i.e., in the "1" state), the SIRF execution continues by incrementing the *Interrupt Counter* in location 7 (assuming CPU #1). At a later time, this counter will tell the Dispatcher how many IRF interrupts are pending.

⑭ An EXIT instruction resets the Interrupt Active flip-flop of the interrupting device controller, and causes an exit to the Dispatcher via the Dispatcher marker on the ICS.

⑮ The Dispatcher checks the content of the Interrupt Counter and, if non-zero, activates each process corresponding to the DRT's which have the IRF bit reset. The IRF bits are then set again to the "1" state.

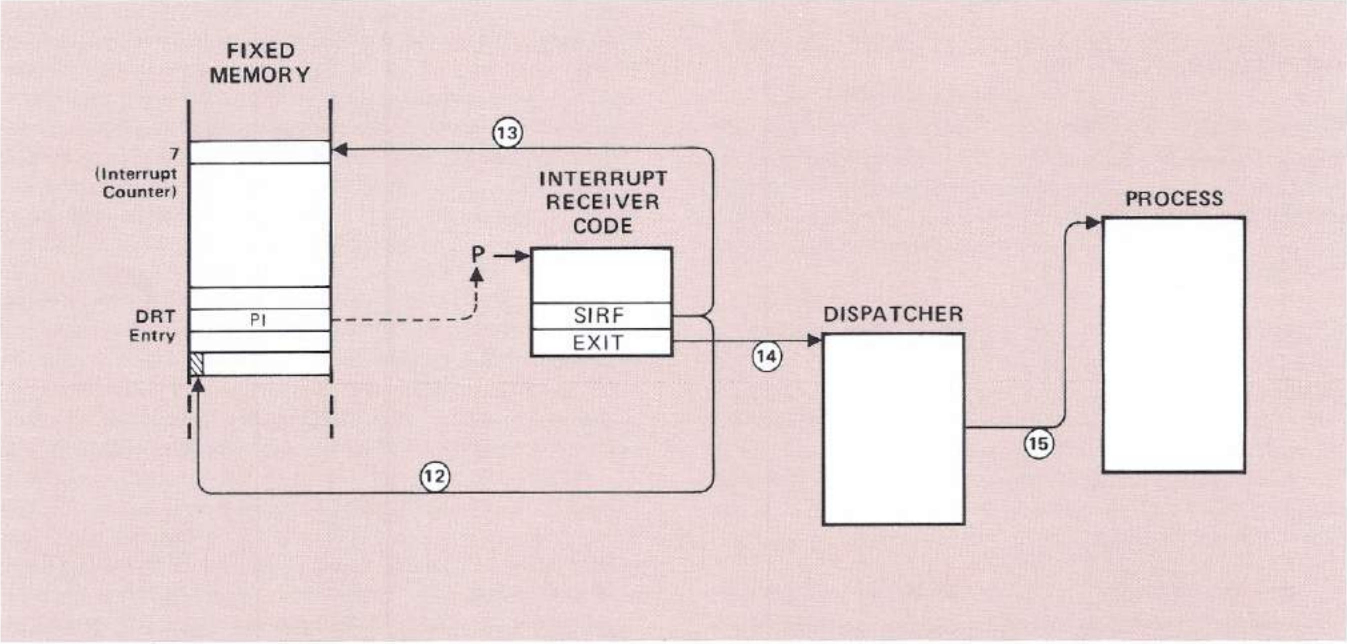


Figure 7-7. IRF Interrupt Processing

# INTERNAL INTERRUPT PROCESSING

As listed earlier in table 7-1, there are 25 internal interrupts, including seven user-related traps. These 25 interrupts are processed by the first 15 dedicated code segments (numbered 1 through 17 octally). Several of the segments process more than one specific interrupt; two of the segments are presently unassigned. Note that all of the user and system traps enter one code segment, number 17.

When internal interrupts are being processed, all external interrupts are disabled. Internal interrupts therefore have higher "priority". Among internal interrupts, however, there is no priority structure (except in the case of simultaneous interrupts); any internal interrupt may interrupt the processing of any other. If multiple interrupts occur simultaneously, they stack their markers in the following order, and are therefore serviced in the reverse order: integer overflow, system parity error, memory address parity error, data parity error, non-responding module, bounds violation, illegal address, module interrupt, external interrupt, console interrupt, and power fail.

A module error interrupt while processing a module error, or a parity error interrupt while processing a parity error, are considered to be irrecoverable errors, resulting in a system halt.

In most cases, the Interrupt Handler loads a parameter onto the stack. The parameter (listed in table 7-1) passes information regarding the interrupt from the hardware to the interrupt processing software. In some cases, the parameter is simply an interrupt identification number; in other cases, the parameter gives specific information, such as a program label, to the interrupt routine.

## GENERAL DESCRIPTIONS

POWER FAIL. Code segment 1 does the interrupt processing for the *Power Fail interrupt*. This routine saves the software status in a format suitable for automatic restart, making use of the finite time between the detection of a power failure and the loss of usable power (approximately 10 milliseconds).

POWER ON. The Power On segment (code segment 2) is entered either by an initial power turn-on, or by an automatic restart following a power failure — if automatic restart is enabled by a panel switch. (The computer will halt on restoration of power if automatic restart is disabled.) Assuming that automatic restart is enabled, the Power On segment will set up the software environment and pass control to the operating system.

STACK OVERFLOW. A stack overflow results from attempting to stack more data than can be contained on the current stack (SM > Z). This condition will result in an interrupt to segment 3, which processes the interrupt. The system makes the decision whether to abort the current process or to expand the stack.

MODULE INTERRUPT. A module interrupt occurs when a CPU receives a transmission from a system module (hardware) from which it is not expecting a transmission. The offending module number (FROM code) is passed to segment 4 as a parameter. The interrupt routine may then attempt to identify the source of the error and take appropriate action. The interrupt is disabled if external interrupts are also disabled (by bit 1 of Status = 0).

CONSOLE INTERRUPT. The *console interrupt* is the console operator's method of getting the attention of the operating system prior to entering an operator command. Segment 5 processes this interrupt, and passes the CPU number as a parameter. In order to protect the Dispatcher from random console interrupts during its final SETR, EXIT sequence, console interrupts are disabled when the Dispatcher Flag is set and the external interrupts are disabled (by bit 1 of Status = 0). Console interrupts are also disabled during system halt and power fail.

COLD LOAD. The *cold load* operation does not use an internal interrupt. It is therefore an exception to the present general discussion of internal interrupts. A "cold load interrupt" is listed as the sixth internal interrupt only in order to obtain a dedicated code segment number (6) for the routine which "brings up" the operating system. But, here again, the designation, code segment 6, is largely fictitious, since the CST entry for that segment has no significance and is not used. Instead, PB is set to 0, PL is set to $2^{16}-1$, and P is indicated by fixed memory location 1. The way cold load operates is roughly as follows: Pressing the COLD LOAD switch causes the CPU to start its cold load microprogram, which begins by reading the operator-set switches on the panel. The switches will have been set to indicate the cold load device number and an 8-bit control byte. The microprogram generates a five-word I/O program beginning at the DRT entry locations for the specified device, and then issues an SIO instruction to that device and goes into a waiting loop to wait for an external interrupt from that device. Meanwhile the I/O Processor causes the device controller to begin executing the five-word I/O program. This program reads in a 32-word bootstrap loader (a larger I/O program), which in turn reads in still larger blocks (e.g., 128 words) which eventually accomplish the loading of all required fixed memory locations. This includes overlaying the previously used DRT locations with normal DRT entries. Finally, the I/O program causes the device controller to generate the external interrupt that the CPU has been waiting for, and ends. The CPU then proceeds to initialize the registers for execution of code segment 6, with the ICS as the data domain. (PB, DB, and DL all set to 0, PL to $2^{16}-1$, Z to ZI, Q to QI, S to Q+1, and P to the content of fixed memory location 1.) The Status register is set to 140006, octal, to indicate privileged mode, enable external interrupts, and indicate segment number 6. The CPU then halts. When RUN is pressed, segment 6 will execute, setting up the operating conditions for the operating system (software tables, linkages, etc.) Once this is complete, the system is in full operation.

MODULE ERROR. Segment 11 processes three different module errors: illegal address, bounds violation, and non-responding module. An identification number (1000, 2000, or 4000, octal) is passed to the routine to identify which type of error occurred. An illegal address is caused by attempting to address a memory location beyond the limit of the physical main memory. A bounds violation interrupt is caused by attempting to address locations outside of a specified program domain or data domain; refer to "Bounds Checking" in Section III. A non-responding module interrupt occurs when the CPU requests information from some other module and that information is not received in a reasonable length of time (a preset time in the order of 4.6 milliseconds).

PARITY ERROR. Segment 12 processes three different *parity errors*: data parity error, memory address parity error, and system parity error. These are indicated by the parameters 10000, 20000, and 40000 (octal) respectively. In general, parity checking is done by the receiving module; there are exceptions, however. Also, only those parity errors that result in an interrupt will be discussed here. (Parity checking is also performed on transmissions between an IOP or Selector Channel and memory; any errors, however, result in a transfer error signal to the affected device controller, rather than a CPU interrupt.) A data parity error interrupt is caused only by the CPU on receiving a data word from memory that has erroneous parity. (Parity should be odd.) A memory address parity error interrupt is caused only by a memory module on receiving an address word from the CPU that has erroneous parity. (Memory will ignore any read or write request accompanying the erroneous address word.) A system parity error interrupt is caused by either the CPU or a memory module on receiving a combination of FROM bits, TO bits, and MOP bits (total of nine bits, including parity) that produces erroneous parity.

MISCELLANEOUS ERRORS. Three different kinds of errors are processed by segment 13. These are: stack underflow, CST violation, and STT violation, indicated by parameter values 1, 2, and 3 respectively. A stack underflow interrupt is caused by an attempt to move SM below DB. This might result from deleting too much information from the stack, or from using the SETR or SUBS instructions incorrectly. (See definition under "Bounds Checking" in Section III.) A CST (Code Segment Table) violation interrupt is caused by calling a non-existent code segment, or by attempting to exit to a non-existent code segment. An STT (Segment Transfer Table) violation interrupt is caused by attempting to call a procedure in an external segment or the local segment through a non-existent STT entry, or if the STT entry in a called external segment is not a local label.

CODE SEGMENT ABSENCE. An interrupt to segment 14 is generated whenever an attempt is made to call or return to a segment that is not present in main memory. The PCAL and EXIT instructions perform the appropriate tests, by checking bit 0 of the first word in the CST entry for the external segment. Segment 14 invokes the memory management part of MPE/3000, which is then responsible for making the absent segment present in main memory. For PCAL, the parameter passed to the segment 14 routine is the external program label, so that the routine will know which segment to make present from the disc. For EXIT, the parameter is the value N, since the EXIT instruction is not fully executed when the interrupt to segment 14 occurs; thus segment 14 must preserve the N value so that S can be pointed at the correct location when execution resumes. (The absence segment is not invoked by interrupts; the absence of an interrupt segment will cause a system halt if that interrupt occurs. The halt also occurs if any segment with a CST number less than octal 20 is absent.)

STT ENTRY UNCALLABLE. An interrupt to segment 15 is generated by a PCAL instruction if attempting to call a segment which has been declared to be uncallable. A segment is uncallable if bit 1 of the STTL word in its PL location is a logic "1".

TRACE. An interrupt to segment 16 is generated by a PCAL instruction when calling an external segment whose CST entry has the *Trace* bit set ("1"). The interrupt is also generated by an EXIT instruction when exiting via a stack marker in which the Trace bit (bit 0 of the return-P word) is set. Thus the Trace segment can collect information on calls outside of the local segment, such as the time taken to execute code segments, etc. (Note: trace interrupts do not occur for interrupts or on exiting from external interrupt routines.)

TRAPS. Segment 17 handles the processing of seven *traps*. Five of these are user traps, which are caused by arithmetic errors, and two are system traps, which are caused by attempted illegal use of privileged mode or unimplemented instructions. Each trap is identified by a parameter which is placed on the stack by the Interrupt Handler; see table 7-1. The five user traps are controlled by the "User Traps Enable/Disable" bit (bit 2) in the Status register; see figure 3-6 in Section III. If the traps are disabled by this bit when an error occurs, the Overflow bit in Status will be set in lieu of the trap; no explicit error identification is given. If, however, the traps are enabled, the interrupt to segment 17 will occur.

The two system traps, on the other hand, are always active — i.e., not subject to the enable/disable bit in the Status register. The mode violation trap includes illegal use of privileged instructions, user exit to privileged mode, and any alteration of the "External Interrupts Enable/Disable" bit (bit 1) of the Status word (checked during EXIT). The unimplemented instruction trap is incurred by any attempt to execute an instruction for which there is no valid code in the machine instruction set.

## SEQUENCE FOR ICS TYPE

Figure 7-8 illustrates the sequence of operations for processing ICS type internal interrupts. The figure shows

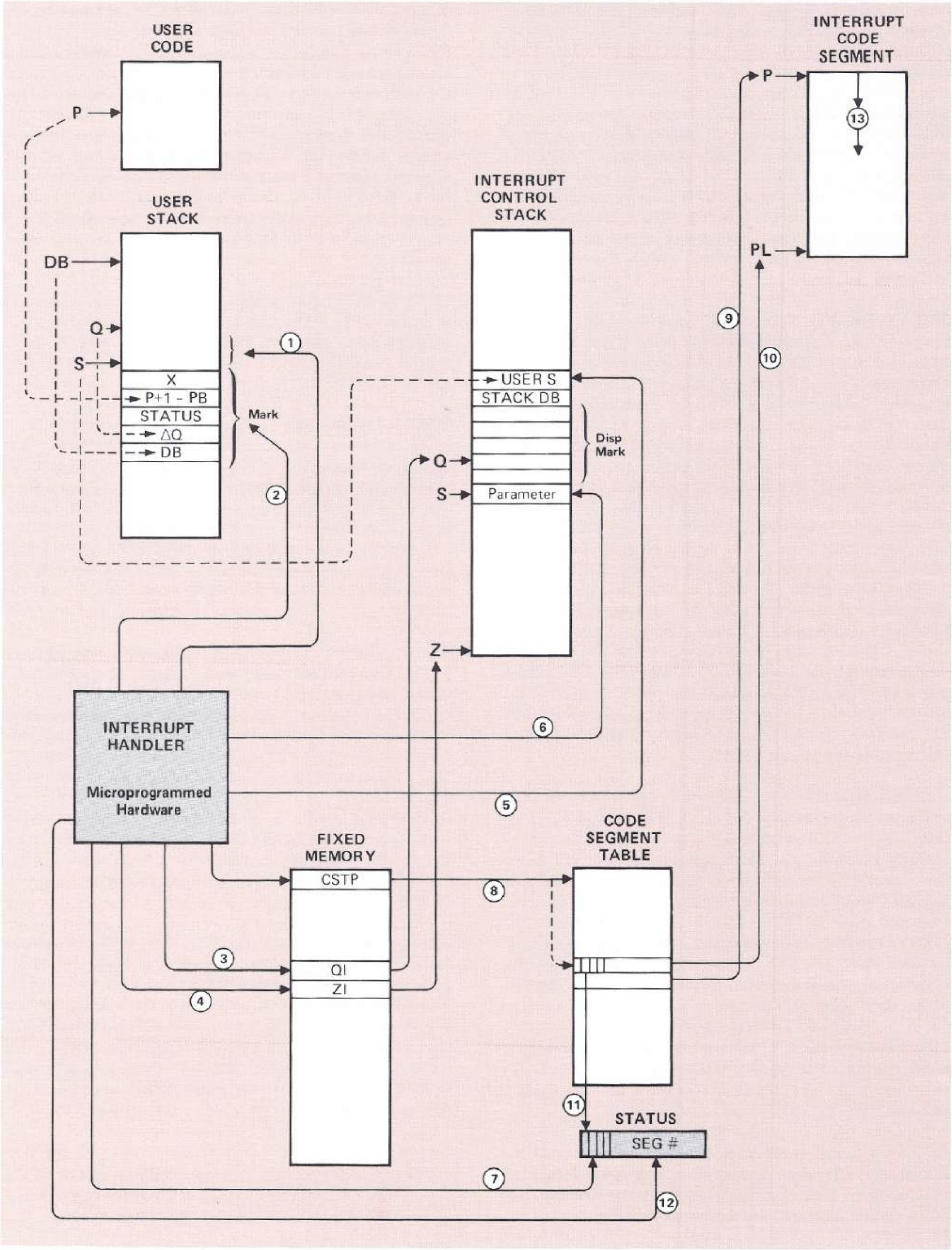


Figure 7-8. ICS Internal Interrupt Operations

how control is transferred from the point of interrupt in the user's code to the start of the interrupt code segment, and how the data domain is switched from the user's stack to the Interrupt Control Stack.

The initial assumption is that the current process is executing at point P in the user's code when an interrupt condition occurs. The CPU then passes control to the Interrupt Handler. The sequence is then as follows:

(1) The Interrupt Handler pushes into memory any TOS elements that are in CPU registers. This takes a maximum of four memory cycles if all four registers are full.

(2) Next, a normal four-word stack marker is pushed onto the user's stack, plus the value of DB that is currently in use.

(3) The QI value is fetched from location 5 and is loaded into the Q-register. This points Q at the delta Q location of the permanent Dispatcher marker.

(4) The ZI value is fetched from location 6 and is loaded into the Z-register. This establishes the stack limit for the Interrupt Control Stack.

(5) The user's current value of S is stored into location Q-5 on the ICS. (Up to this point the operation has been identical to the sequence of operations for external interrupts, described earlier; the actions now begin to differ.)

(6) A parameter, if any, is now pushed onto the ICS. Only the Module and Console interrupts use a parameter.

(7) External interrupts are disabled by clearing ("0") bit 1 of the Status register.

(8) The Interrupt Handler next fetches the Code Segment Table Pointer from fixed memory location 0. Using this value, indexed by two times the code segment number of the specific interrupt, the Interrupt Handler then fetches the relevant CST entry.

(9) The absolute address in the second word of the fetched CST entry is loaded into the P- and PB-registers. (Unlike external interrupts, which set PB to 0, an internal interrupt provides a value for PB and starts P at that location — as does PCAL.)

(10) Using the code segment length value in the first word of the CST entry, PL is established relative to PB.

(11) The mode bit in the CST entry (bit 1 of the first word) is transferred into the mode bit (bit 0) of the Status register. This determines the mode of execution for the segment, privileged mode or user mode. Also, external interrupts and user traps are disabled.

(12) The code segment number is loaded into the Status register to indicate which segment is executing.

(13) Lastly, the CPU fetches the instruction at P and begins executing the interrupt code segment.

Additional ICS type internal interrupts could occur before exiting from the interrupt code segment, and they would be stacked on the ICS in a manner similar to that shown in figure 7-6. If there are any external interrupts, either suspended on the ICS or waiting for priority, they will be processed after all internal interrupts have been processed. (However, external interrupts can interrupt internal interrupt routines if the software re-enables the external interrupt system.) After all internal and external interrupts using the ICS have been processed, an exit to the Dispatcher will occur, as described for steps 17 and 18 of figure 7-6.

## SEQUENCE FOR NON-ICS TYPE

Figure 7-9 illustrates the processing of non-ICS type internal interrupts. As shown in the figure, the Interrupt Control Stack is not used; the interrupt code segment will operate on the user's stack.

Assume that the user is executing at point P when an interrupt condition occurs. The CPU passes control to the Interrupt Handler, and the sequence is then as follows:

(1) Any TOS elements that are in CPU registers are pushed into memory.

(2) A normal four-word stack marker is pushed onto the user's stack.

(3) The parameter is pushed onto the stack.

(4) External interrupts are disabled by clearing ("0") bit 1 of the Status register.

(5) The Interrupt Handler next fetches the Code Segment Table Pointer from fixed memory location 0. Using this value, indexed by two times the code segment number of the specific interrupt, the Interrupt Handler then fetches the relevant CST entry.

(6) The absolute address in the second word of the fetched CST entry is loaded into the P- and PB-registers.

(7) Using the code segment length value in the first word of the CST entry, PL is established relative to PB.

(8) The mode bit in the CST entry (bit 1 of the first word) is transferred into the mode bit (bit 0) of the Status register. This determines the mode of execution for the segment, privileged mode or user mode.

(9) The code segment number is loaded into the Status register to indicate which segment is executing.

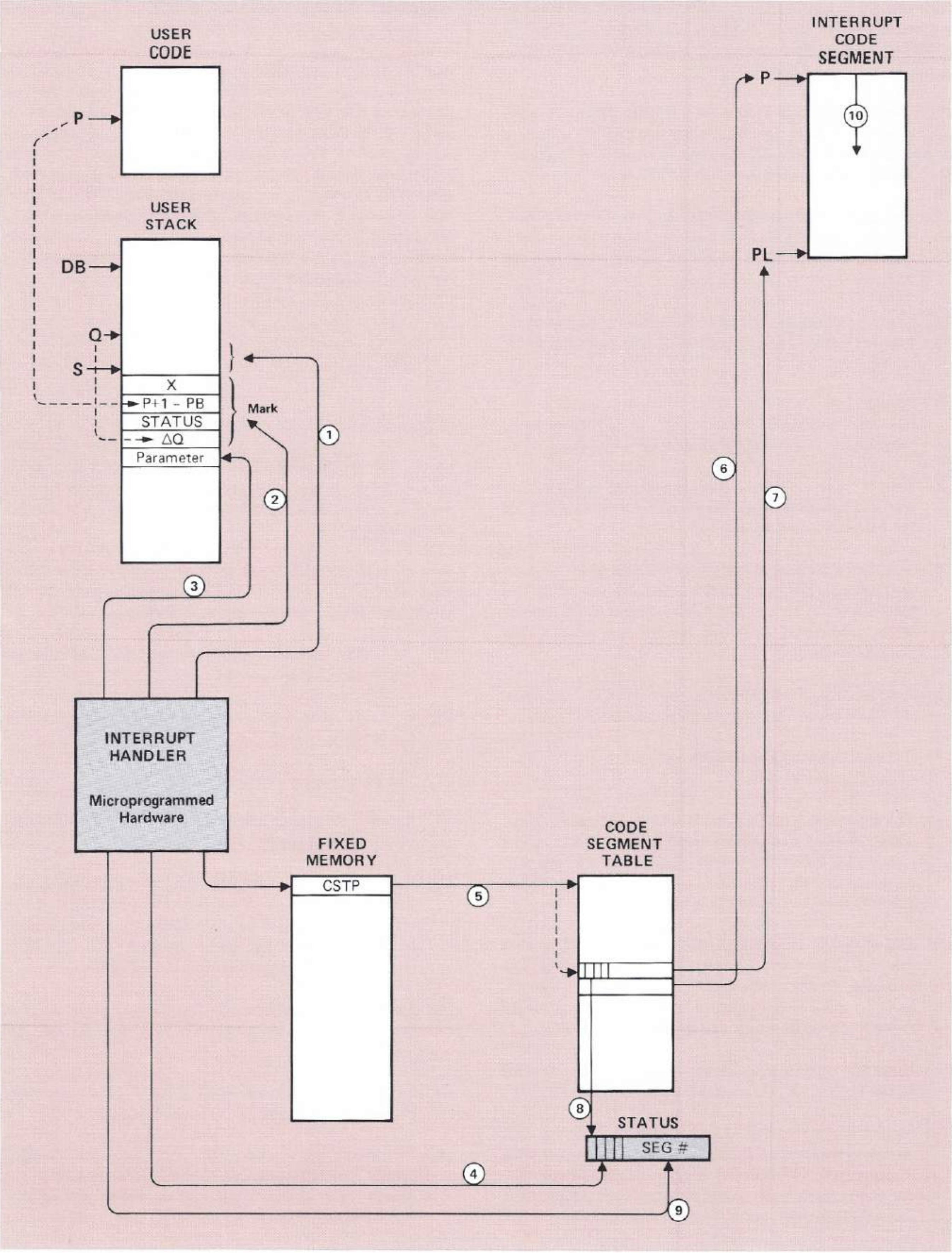


Figure 7-9. Non-ICS Internal Interrupt Operations

⑩ The CPU fetches the instruction at P and begins executing the interrupt code segment.

If an ICS type internal interrupt should interrupt the processing of a non-ICS type, control will not revert to the non-ICS routine until there is an exit to the Dispatcher from the ICS. The user's process, in that case, will have to contend with other processes for priority.

## INTERRUPT HANDLER

The Interrupt Handler is a microprogram (actually a set of microprograms) permanently stored within a read-only memory in the CPU. The CPU periodically checks for the existence of a waiting interrupt condition, which is stored in one of several bit positions in a dedicated CPU register (CPX1 or CPX2), and then transfers control to the Interrupt Handler.

The purpose of the Interrupt Handler is to save the interrupted environment and transfer control to the interrupt routine in software. The suspended environment is saved in a format that is ready to resume execution.

The descriptions which follow are essentially a summary of the preceding portion of this section. A flowchart will be used as a basis for discussion, with the assumption that the reader understands the physical operations that have been previously described.

Figures 7-10 and 7-11 illustrate the operations performed by the Interrupt Handler. Generally, the sequence begins with the START block at the top left corner and ends with the EXECUTE block at the bottom right corner. There are exceptions for cold load, power on (if automatic restart is disabled), the "run" interrupt, halt mode interrupts (mostly single-cycle operations), parity errors or module errors while executing the respective parity error or module error routines, and disabled traps.

As shown proceeding down the left side of figure 7-10 (A through F), a series of tests is made to identify the basic type of error. Sub-tests G through K provide further identification. (Note that tests for Absence, Trace, and STT Entry Uncallable interrupts are not included, since the EXIT and PCAL microprograms handle those three interrupts.) The following descriptions are given in the sequence of basic tests, A through F.

### ICS TYPE

If the waiting interrupt is determined to be of the ICS type (A), an additional test (G) is made to see if it is a Power-On interrupt. If not, skip the remainder of this paragraph. If so, a further check is made to see if auto-restart is enabled. If not, the microprogram jumps to the halt loop. If auto-restart is enabled, the data segment registers are set up for operation on the ICS (block 1). That is, Q is set to QI, Z to ZI, and S to the content of QI–5 (where S was saved by the power fail interrupt). The DB- and DL-registers are initially cleared. Many of these register settings are largely arbitrary, simply to provide standardized initial conditions for the Power On routine. Once this is done, the operation proceeds to the "Transfer Control" sequence (refer to that heading).

Assuming that the interrupt is not a Power-On, the sequence continues as follows (referring to blocks 2, 3, 4, and 5). First (block 2), a standard four-word stack marker is pushed onto the current stack. Next (block 3), the current DB value is pushed onto the stack. This is followed by a test to see if the interrupt occurred while operating on the ICS. If not, the user's value of S is saved in location QI–5 of the ICS (block 4), and the operation proceeds to block 5; if so, block 4 is bypassed and an additional test is made to see if the interrupt occurred while executing in the Dispatcher. If so, the Dispatcher is aborted by establishing the ICS again (Q = QI, Z = ZI, and S = QI + 1), as set by block 5. Otherwise block 5 is bypassed, since the registers are already set for ICS operation. In any case, the operation now proceeds to the "Transfer Control" sequence (refer to that heading).

### MISCELLANEOUS ERROR

If the waiting interrupt is determined to be a miscellaneous error (B), the only operation to occur before the "Transfer Control" sequence is to push a standard four-word stack marker onto the current stack (block 6). Miscellaneous errors include stack underflow, CST violation and STT violation.

### TRAPS

If the waiting interrupt is the result of a trap (C), it is tested to see if it is an arithmetic trap (H). If not, (i.e., a system trap), the remainder of this paragraph is skipped, since system traps are not subject to the enable/disable bit. For arithmetic traps, the state of the enable/disable bit in the Status register is checked. If traps are enabled, the remainder of this paragraph is skipped. If disabled, the Overflow bit in Status is set (block 7), and the next instruction is fetched and executed; no further interrupt handling operations occur.

The next action is to push a standard four-word stack marker onto the current stack (block 6). Following this, the "Transfer Control" sequence begins.

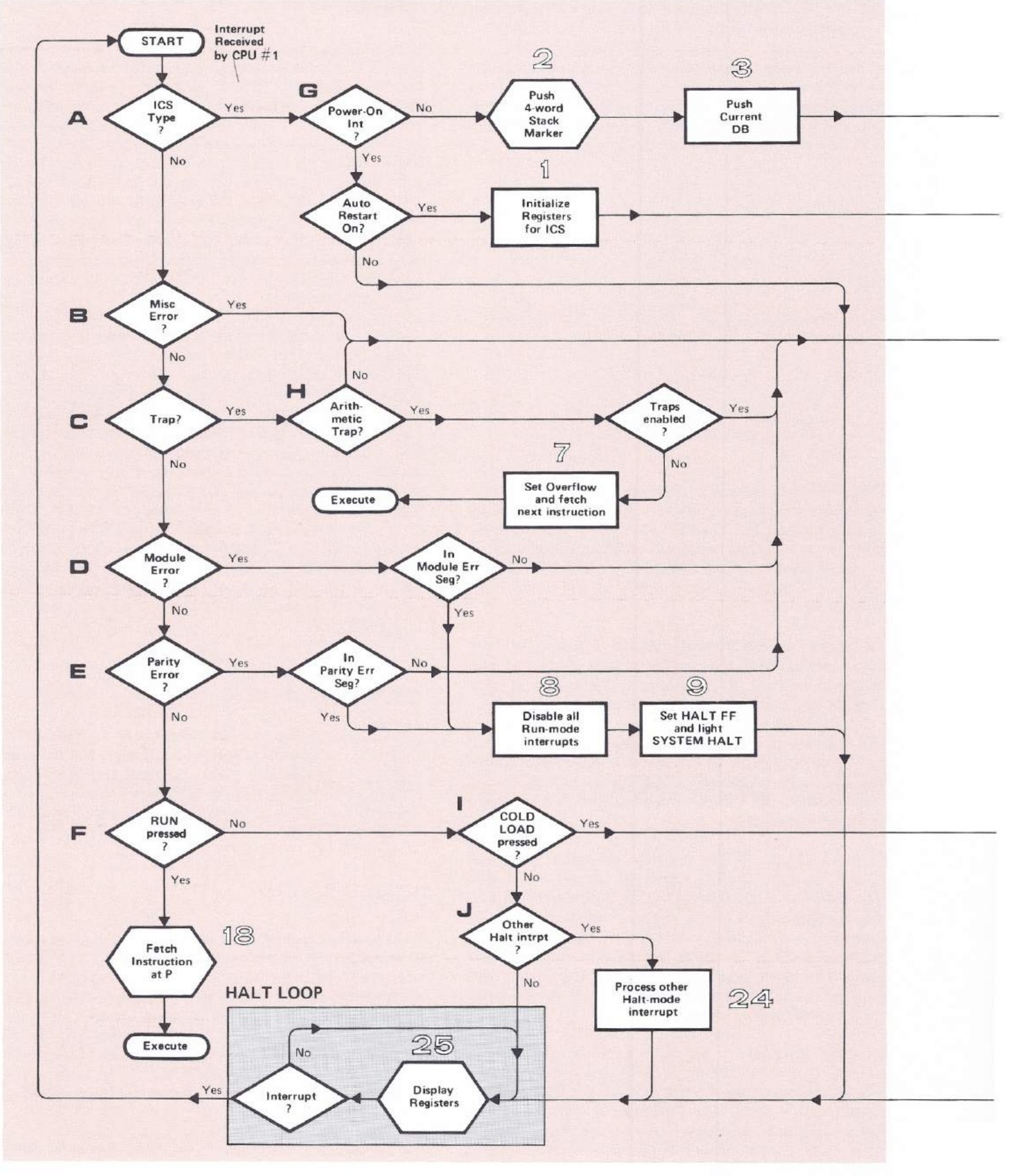


Figure 7-10. Interrupt Handler, Part A

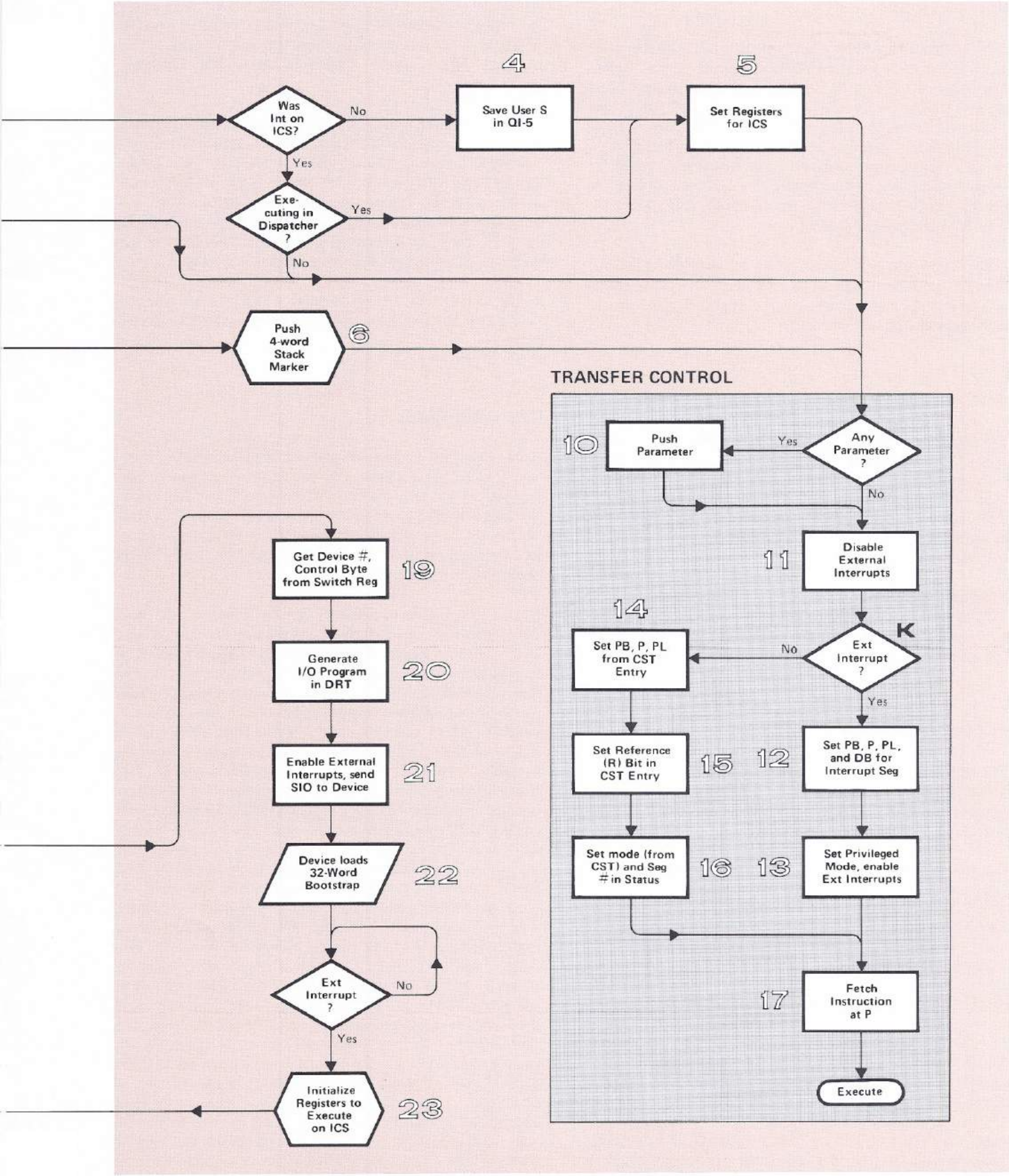

4
5
Was Int on ICS?
No
Save User S in QI-5
Set Registers for ICS
Yes
Exe-cuting in Dispatcher ?
Yes
No
Push 4-word Stack Marker
6
TRANSFER CONTROL
10
Push Parameter
Yes
Any Parameter ?
No
11
Disable External Interrupts
Get Device #, Control Byte from Switch Reg
19
14
K
Set PB, P, PL from CST Entry
No
Ext Interrupt ?
Yes
Generate I/O Program in DRT
20
Set Reference (R) Bit in CST Entry
15
12
Set PB, P, PL, and DB for Interrupt Seg
Enable External Interrupts, send SIO to Device
21
Set mode (from CST) and Seg # in Status
16
13
Set Privileged Mode, enable Ext Interrupts
Device loads 32-Word Bootstrap
22
Ext Interrupt ?
No
Yes
17
Fetch Instruction at P
Execute
Initialize Registers to Execute on ICS
23

Figure 7-11. Interrupt Handler, Part B

## MODULE ERROR

If the waiting interrupt is a module error (D), a test is made to see if the error occurred while executing in the Module Error segment. If not, the remainder of this paragraph is skipped. If the error did occur while executing in the Module Error segment, the error will halt the computer. All run-mode interrupts are disabled (block 8), meaning that the CPU will respond only to those interrupts caused by pressing operator panel switches. The Halt flip-flop is set and the SYSTEM HALT light on the panel is lit (block 9). The CPU then enters the halt loop.

If the error did not occur while in the Module Error segment, a standard four-word stack marker is pushed onto the current stack (block 6), and the operation proceeds to the "Transfer Control" sequence.

## PARITY ERROR

If the waiting interrupt is a parity error (E), a test is made to see if the error occurred while executing in the Parity Error segment. If not, the remainder of this paragraph is skipped. If so, the error will halt the computer. All run-mode interrupts are disabled (block 8), the Halt flip-flop is set, and the SYSTEM HALT light is lit (block 9). The CPU then enters the halt loop.

If the error did not occur while in the Parity Error segment, a standard four-word stack marker is pushed onto the current stack (block 6), and the operation proceeds to the "Transfer Control" sequence.

## TRANSFER CONTROL

The "Transfer Control" sequence completes most of the operations described above. Basically, this sequence simply transfers control to the appropriate interrupt routine software.

The first action is to test whether any parameter is to be passed to the routine. If so, the appropriate parameter (see table 7-1) is pushed onto the current stack (block 10). (Depending on how this sequence was entered, the current stack is either the ICS or a user stack.) If no parameter is required, block 10 is bypassed.

Next, all external interrupts are disabled while the code segment registers are being set up (block 11). In the case of internal interrupts, the external interrupts will remain disabled on completion of this sequence; the interrupt routine software will eventually re-enable the external interrupts.

Next a test is made to see if the interrupt is an external interrupt (K). If not, the remainder of this paragraph is skipped. For external interrupts (block 12), P is set to 0, PL to $2^{16}-1$, and P to the PI value given in the DRT entry for the interrupting device. Also DB is set to the DBI value. Next (block 13), the mode bit in the Status register is set to privileged mode and external interrupts are re-enabled. The sequence for external interrupts skips the following paragraph.

For internal interrupts (block 14), PB and PL are set from the values given in the CST entry for the appropriate interrupt code segment; P is set equal to PB. Next (block 15), the Reference bit in the CST entry is set if it has not already been set. The Status register is updated (block 16) by setting the mode bit to the same state as the mode bit in the CST entry, and loading the segment number of the interrupt code segment into bit positions 8 through 15.

Finally (block 17), the instruction in the location pointed to by P is fetched, and execution of the software routine begins.

## RUN/LOAD/HALT

If the interrupt is none of the run-mode interrupts mentioned above, it is a halt-mode interrupt — i.e., one caused by pressing a switch on the operator panel. The first check (F) tests if the RUN pushbutton was pressed. If so, the CPU fetches the instruction in the location specified by the current address in the P-register (block 18), and begins execution.

If the interrupt is not due to the RUN switch, the next check (I) tests if the COLD LOAD pushbutton was pressed. If not, the remainder of this paragraph is skipped. The cold load sequence is represented by blocks 19 through 23. The first action (block 19) is to read the manually-set content of the Switch register; this will consist of an eight-bit device number and an eight-bit control byte. Next (block 20), a five-word I/O program is loaded into the memory locations beginning at the DRT locations for the input device. (This is an arbitrary starting point for beginning the bootstrap loading operation; the DRT locations will be overlaid with correct DRT information at a later time by the loading software.) Next (block 21), external interrupts are enabled and an SIO command is issued to the input device. The device controller then begins executing the five word I/O program, which includes a command to read 32 words comprising a bootstrap loader (block 22). On completion of the five-word program, the device controller continues executing into the 32-word program, while the CPU waits for an interrupt from the device. One of the commands in the I/O program will be an interrupt command. This will cause the device controller to generate an external interrupt, and the sequence then continues to block 23. Block 23 initializes the data segment registers to operate on the ICS (Q = QI, Z = ZI, S = QI + 1) and sets P to the content of location 1. Location 1 will by this time have a cold-load address for P, and locations 2 and 3 will be used during the cold load operation; all three of these locations will later be overlaid with correct system information as indicated in an earlier section (table 4-1). The CPU now goes into the halt loop and waits for RUN to be pressed.

If the interrupt was not due to RUN or COLD LOAD being pressed, the Interrupt Handler checks what other halt-mode interrupt occurred. Examples are: single instruction switch, load register switch, display memory switch, etc. The appropriate operation is performed (block 24), and the CPU goes into the halt loop. The halt loop is also entered if no cause is found.

The halt loop consists of displaying the registers (block 25), and checking for any interrupt to occur. When an interrupt does occur, control is transferred to the start of the Interrupt Handler.

# FUNCTIONAL OPERATION SECTION VIII

This section describes the logic operation of the system hardware. The complexity of the hardware precludes any detailed discussion of logic cards in this reference manual. Instead, the descriptions given here are highly simplified, based mostly on block diagrams. If further details are required, the reader must refer to the maintenance documentation.

Brief descriptions of the following units are given:

a. the bus system
b. the Central Processor Unit (CPU)
c. the Module Control Unit (MCU)
d. a typical memory module
e. the Input/Output Processor (IOP)
f. the Multiplexer Channel
g. the Selector Channel

In addition, sequences of operations for CPU transfers to and from memory are given, as well as I/O transfers by way of both the Multiplexer Channel and the Selector Channel.

As much as possible, correct nomenclature has been applied. A list of mnemonics and abbreviations used in this section is given in table 8-1, at the end of this section.

## BUS SYSTEM

The *bus system* is a network of data and control lines which are necessary to effect the transfer of data between modules and between I/O devices and memory. Figures 8-1 and 8-2 show, respectively, the electrical and physical configuration of the system buses. Figure 8-1 represents a four-module system, consisting of a CPU/IOP module, two primary memory modules, and a high-speed channel module. The I/O system includes two Multiplexer Channels, although there may be any practical number; each Multiplexer Channel can accommodate up to 16 device controllers. The Port Controller is shown with two Selector Channels, each of which can accommodate up to eight device controllers. There may be additional Port Controllers, each of which will be assigned a module number.

CENTRAL DATA BUS. All communications and transfers of data between modules occur by way of the *central data bus*. This bus consists of a 50-conductor flat cable which connects together each Module Control Unit (MCU) and each Port Controller in the system. See both figures 8-1 and 8-2. (Figure 8-2 does not illustrate the central data bus terminator cards, which are attached to each end of the bus.)

IOP BUS. The I/O Processor (IOP) is connected to every device controller in the system by the *IOP bus*. As explained later, Multiplexer Channels are also connected to this bus. The IOP bus provides the means for the IOP (in one direction) to send control signals and control words to any device controller and (in the reverse direction) to accept interrupts from the device controllers. For multiplexed SIO devices, all data transmissions also occur via the IOP bus. For high-speed devices on a selector channel, data transmissions occur via the IOP bus only for the direct I/O instructions (RIO, WIO, CIO, and TIO).

SELECTOR CHANNEL BUS. The *selector channel bus* (one per Selector Channel) provides the communication path for a Selector Channel to select one of up to eight devices for transmission. Data transmissions on the channel bus, occurring as a result of an SIO instruction, are by block transfer (data burst). Only one device on any channel can be selected at a time, and it will monopolize the channel until the device's I/O program is finished. The Port Controller, however, can service all four channels simultaneously, on a word-by-word basis.

MULTIPLEXER CHANNEL BUS. With a few minor differences in signal nomenclatures, the *multiplexer channel bus* is virtually identical to the selector channel bus. This allows certain device controllers, such as high-speed discs, to be connected interchangeably to either bus. The difference is that data transmissions are under control of the Multiplexer Channel instead of a Selector Channel. All data transmissions, in this case, are via the IOP bus and are multiplexed among the devices on a word-by-word basis. (The equivalent data lines on the channel bus are used as service request lines on the multiplexer channel bus.)

POWER BUS. The *power bus*, unlike the flat cable signal buses discussed above, is a rigid printed circuit board. Terminal strips on the right side of each board (in figure 8-2) accept the power wires from the power supply, which is mounted to the rear of the cabinet. However, some I/O bus lines and the system clock are also routed along the power bus, as indicated by the small flat cables shown attached to the power bus in the figure.

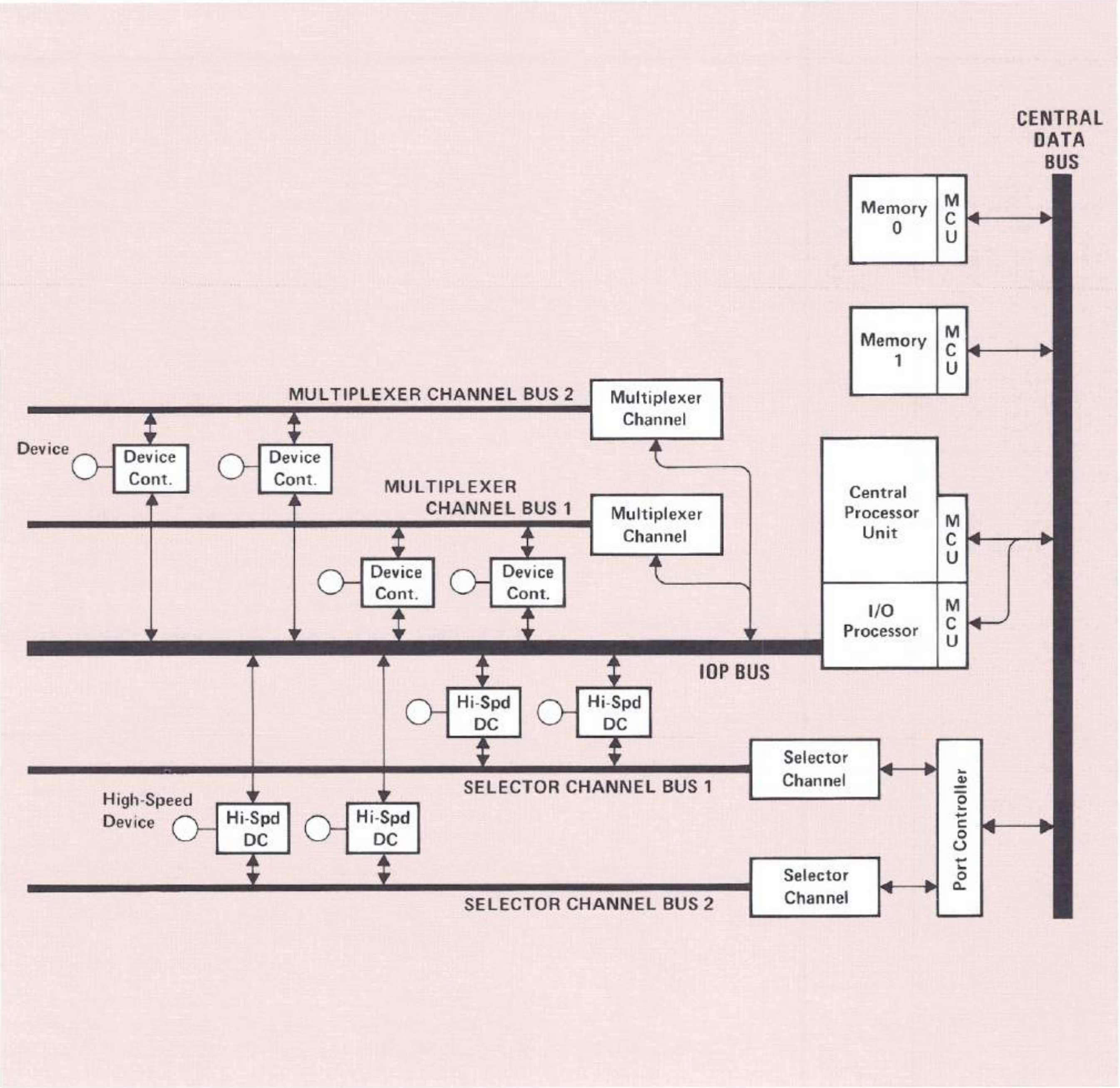


Figure 8-1. The Bus System

## CENTRAL PROCESSOR UNIT

The CPU portion of the CPU/IOP module logically consists of three sections, as shown in figure 8-3. The Instruction Decoder receives an instruction word from memory and translates it into a microprogram starting address; the microprogram is then read out of ROM (read-only memory) and is decoded into a set sequence of control signals. The Processor Registers include 20 flip-flop registers that can be loaded from the U-bus (i.e., output of Arithmetic Logic) and read onto the R-bus and/or S-bus (inputs to Arithmetic Logic). The Arithmetic Logic basically executes various functions (add, subtract, etc.) on the R- and S-bus inputs, with or without a shift, and outputs the result to either the CPU Output Register (for transmission out of the module) or the U-bus (for storage in one of the internal registers).

CPU elements identified in figure 8-3 are briefly described in the following paragraphs. (Figure 8-4, shown facing figure 8-3 in order to show MCU interconnections, will be discussed later.)

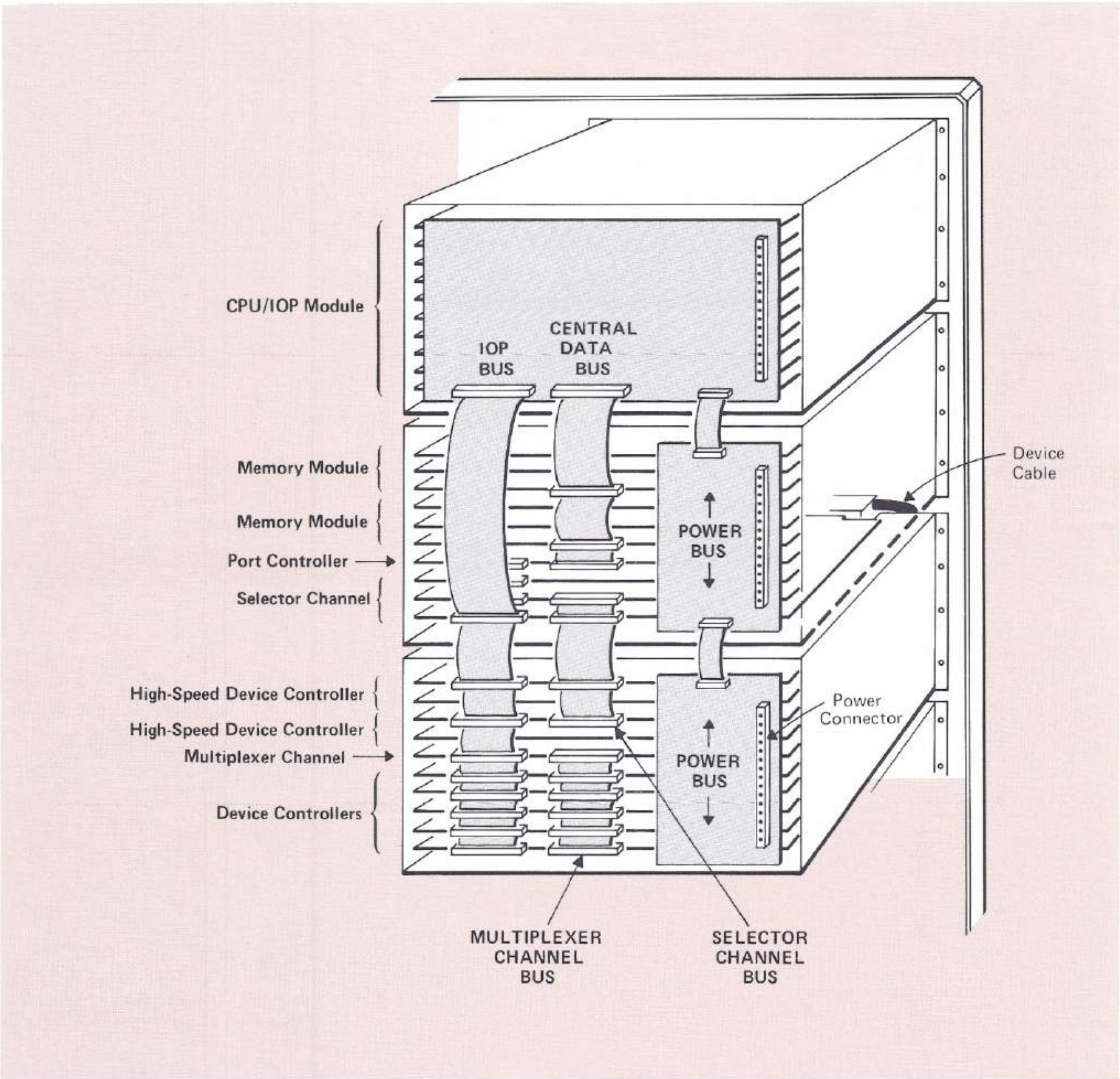


Figure 8-2. System Buses, Rear View

## INSTRUCTION DECODER

NEXT INSTRUCTION REGISTER. The *Next Instruction Register* is loaded with an instruction from memory by a procedure which is described under the heading, Central Data Bus Transmissions.

CURRENT INSTRUCTION REGISTER. The *Current Instruction Register* contains the instruction that is currently being executed. It is loaded from the Next Instruction Register by a NEXT signal from the microprogram. The reason for having two instruction registers is so that one instruction can be executing while another is being fetched from memory.

LOOK-UP TABLE. The *Look-Up Table*, together with a preliminary address generator, provides two stages of decoding to produce a microprogram starting address from the instruction bits in the Current Instruction Register.

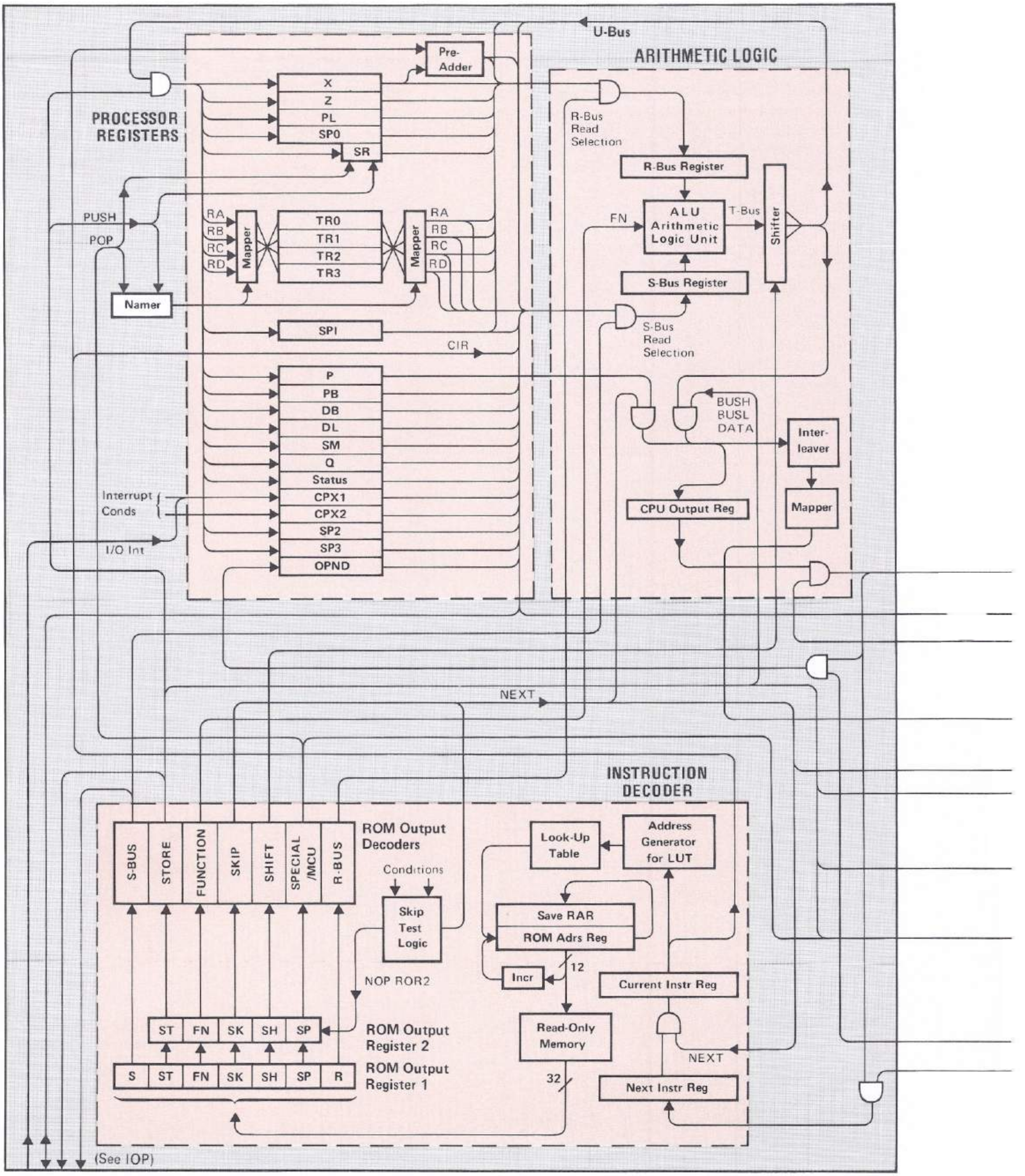


Figure 8-3. Central Processor Unit

ROM ADDRESS REGISTER. The *ROM Address Register* (RAR) supplies the address of each microprogram word to the Read-Only Memory. It is given a starting address from the Look-Up Table and is thereafter automatically incremented every 175 nanoseconds until the end of the microprogram for that instruction is reached. However, during the execution of a microprogram, the RAR contents may be forced to some other address value, such as for a microprogram jump or jump-to-subroutine. Although not shown in figure 8-3, the ROM Address Register can be loaded from the ROM Output Registers, the U-bus, and the Hardware Maintenance Panel. When the microprogram does a jump-to-subroutine, the current ROM address is saved in the Save ROM Address Register; upon return from the subroutine, the saved value is loaded back into the ROM Address Register.

READ-ONLY MEMORY. The *Read-Only Memory* (ROM) accepts 12-bit addresses from RAR and outputs the 32-bit microinstruction words of a *microprogram* to the ROM Output Registers. There is at least one microprogram in ROM for each machine instruction. For example, instructions which affect the top-of-stack will first call a microprogram routine to check that there are enough filled or vacant top-of-stack registers to carry out the operation; then, after possibly one or more memory transfers to adjust the stack, the microprogram for the instruction may begin.

ROM OUTPUT REGISTERS. There are two *ROM Output Registers* (ROR), numbered 1 and 2. The 32-bit output from ROM is loaded into ROR1 on each clock cycle (175 nanoseconds). On the next clock cycle, five of the seven fields of the microinstruction word are transferred from ROR1 to ROR2 (while ROR1 is receiving the next microinstruction word). Thus it takes two cycles to initially "fill the pipeline", but thereafter ROR2 receives a new microinstruction word on each successive cycle. The reason for having two ROM Output Registers is so that the S and R fields can be decoded in advance of the rest of the word. Thus S- and R-bus selection will have occurred, and the selected data will be ready and waiting at the S- and R-bus Register outputs by the time the rest of the word is decoded from ROR2.

ROM OUTPUT DECODERS. Each field of the ROM output word is separately decoded. The S-bus field selects one of 31 registers (or sets of lines) to be loaded into the S-bus Register. (Only 22 are shown in figure 8-3.) The Store field selects one of 22 registers (not all shown) in which to store the U-bus data. In addition, the PUSH and central data bus

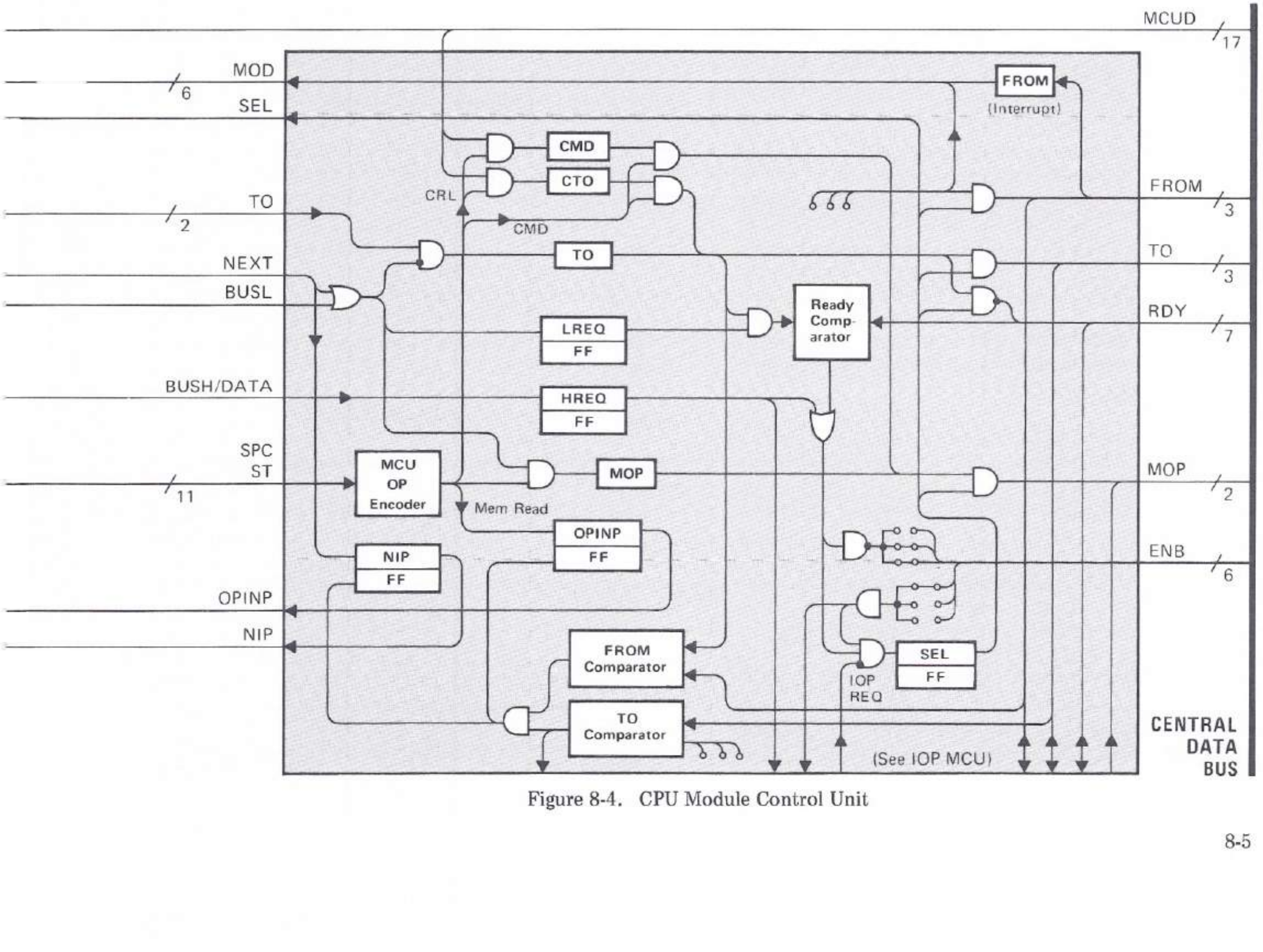


Figure 8-4. CPU Module Control Unit

request signals also come from this field. The Function field specifies the function that the Arithmetic Logic Unit is to perform on the two operands in the R- and S-bus Registers. The Skip field determines what condition shall be tested for a possible skip; if the condition is met (e.g., U-bus positive/negative, odd/even, zero/non-zero, overflow set, etc.). ROR2 is caused to execute a NOP (no operation), effectively skipping one microinstruction word. Other signals, such as NEXT, also come from the Skip field. The Shift field specifies how the T-bus data will be shifted onto the U-bus (right one, left one, straight through, etc.). The Special field has many varied uses including the generation of POP and memory opcode signals. The R-bus field selects one of 15 processor registers (or the U-bus) for loading into the R-bus Register.

## PROCESSOR REGISTERS

There are 22 processor registers, as shown in figure 8-3. These registers may be selectively loaded from the U-bus (except the Operand register, which is loaded from the central data bus, and the interrupt condition registers CPX1 and CPX2), and selectively read into the R- and/or S-bus Registers. Figure 8-3 groups together those registers that are similarly read out. For example, the X-, Z-, PL-, SP0, and SR-registers may be read out only to the R-bus Register. TR0 through TR3 and SP1 registers may be read out to either the R- or S-bus Registers. The P-, PB-, (etc.) through OPND Registers may be read out only to the S-bus Register.

The purposes of most of the processor registers are more appropriately discussed elsewhere in this manual, and so will not be discussed here. However, a few of these registers are not accessible from outside the CPU, and so are shown nowhere else except in figure 8-3. For example, the four *scratch pad registers*, SP0, SP1, SP2, and SP3, are used only by the ROM microprograms. These registers are available to the microprograms for holding temporary values, such as to contain the middle word during triple-word shifts (while the R- and S-bus Registers contain the most and least significant words, respectively).

The logic consisting of the namer, two mappers, the four TR registers (TR0 through TR3) and the SR-register, is designated as the Top-of-stack Register Renamer, or simply the *renamer*. This logic permits fast access to the top-of-stack elements by renaming the registers when stack elements are added or deleted (rather than transferring data from register to register). The ROM microprograms know the top-of-stack elements (when in the CPU) only by the names RA (top), RB, RC, and RD. The namer includes a two-bit naming register to tell the mappers which of the four Top-of-stack Registers (TR0 through TR3) is "RA", and "RB", etc. This two-bit naming register is decremented each time a stack element is added (PUSH) and incremented each time a stack element is deleted (POP). To keep track of how many elements are in the TR registers, the three-bit SR-register is incremented by PUSH and decremented by POP, in step with the naming register. When the SR-register count is zero, there are no elements in the TR registers; this would tell a ROM microprogram not to look for RA in the CPU, and that one or more memory fetches may be required.

The *pre-adder* is used to gain a speed increase for instructions which use or perform computations on bits in the Current Instruction Register. For example, when executing indexed memory reference instructions, the proper displacement field of the Current Instruction Register is pre-added to the contents of the X-register. Thus the final absolute address can be computed in only one cycle by adding the output of the pre-adder to the contents of the base register (P, DB, Q, or S).

## ARITHMETIC LOGIC

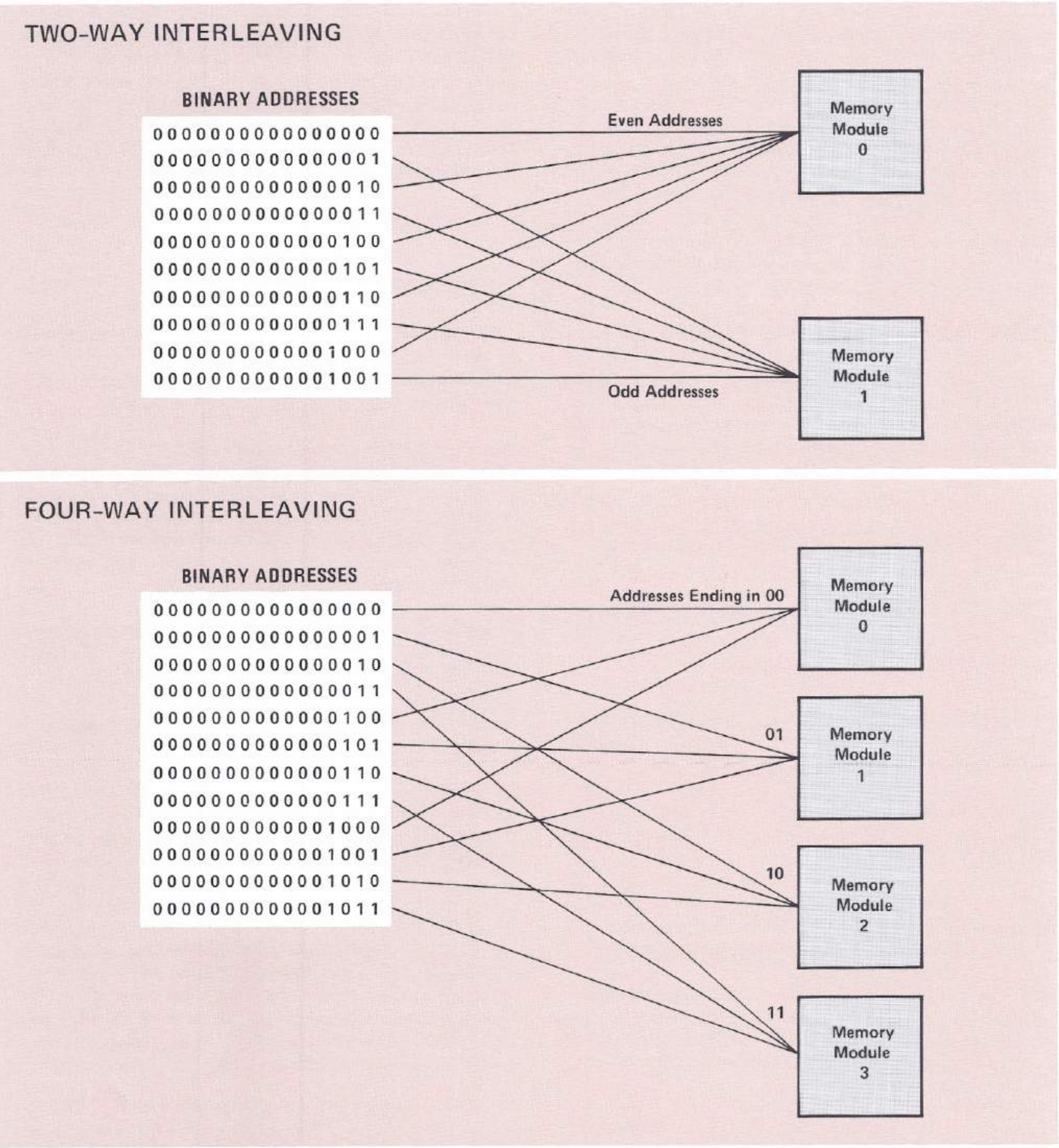
The foregoing discussions have touched on about half of the arithmetic logic blocks. The following paragraph summarizes these blocks; this will be followed by descriptions of the previously unmentioned blocks.

The R-bus and S-bus read selection circuits, under control of the ROM R- and S-bus fields, read one of the processor registers (or a set of bus lines) into the R-bus and S-bus Registers. Then, under control of the ROM Function field, the Arithmetic Logic Unit performs an arithmetic or logical function on the R- and S-bus operands. And under control of the Shift field, the result on the T-bus is transferred either directly or shifted onto the U-bus.

CPU OUTPUT REGISTER. There are actually two *CPU Output Registers*, though both are represented as a single register for simplicity in figure 8-3. These registers are used as buffers for sending information to memory. If the ROM Store field specifies DATA, BUSH (Bus High), or BUSL (Bus Low), the U-bus is loaded into the CPU Output Register. If the ROM Skip field specifies NEXT (to fetch next instruction), the P-register is loaded into the CPU Output Register. When the transmission to memory occurs (by a procedure described later under the heading, Central Data Bus Transmissions), a SEL (Select) signal reads the buffer contents out to the central data bus.

INTERLEAVER. The *interleaver* is a circuit which provides mechanical switches for the user to select one of two memory interleaving schemes. Memory interleaving causes the memory transmissions for consecutive addresses to be directed alternately between two memory modules or, for four-way interleaving, rotationally among four memory modules. This is illustrated in figure 8-5. Note that for two-way interleaving, all even addresses are directed to one module and all odd addresses to the other module. For four-way interleaving, the two least significant bits of the address are used for module selection.

The advantage of interleaving is that, if sequential addresses are being accessed, it allows memory cycles to overlap; that is, a second transmission to memory may be made while the



## BIT EXCHANGES

Two 64K-byte (32K-word) Modules: 0 ←→ 15

Four 32K-byte (16K-word) Modules: 0 ←→ 14 and 1 ←→ 15

Figure 8-5. Memory Interleaving

first is still going through its memory cycle in another module. Logically, interleaving is accomplished by exchanging address bits as shown in the lower box of figure 8-5. In the CPU, the interchanging is done only to obtain a module number; the address sent out on the central data bus is unmodified. In the memory module, the address word itself is altered by the specified bit exchange, so that all memory locations can be filled. (Obviously, if a module used only even-numbered addresses, only half of its locations could be used.) An incidental effect of the bit exchange is that addresses which were originally consecutive will not be adjacent within the memory module; this is of no consequence in the operation of the system.

MAPPER. The *mapper* (i.e., memory mapper) examines the three most significant bits of each address word from the output of the interleaver, and outputs a module number on the TO lines which corresponds to that address. Jumpers or switches are used to configure the mapper appropriately for the quantity and sizes of memory modules existent in the system.

## MODULE CONTROL UNIT

Each module gains access to the central data bus by way of its *Module Control Unit*, or MCU. The Module Control Unit in each module may be a dedicated card, distributed on several cards, or located on a small part of one card. However, they all perform essentially the same function, and that is to establish the priority of transmissions on the central data bus.

Figure 8-4 illustrates in simplified form the logic of the CPU Module Control Unit. This MCU is representative, and will be used as an example in the following discussions.

Since the purpose of the MCU is to effect bus transmissions, the logic is best described by following the sequence of operations involved in different types of bus transmissions. Refer to the next major heading, Central Data Bus Transmissions.

## CENTRAL DATA BUS TRANSMISSIONS

The procedures discussed under this heading describe how an instruction is fetched, how an operand is fetched and stored, and how a module is given an operation code. Figures 8-3 and 8-4 are used as references throughout these procedures.

As mentioned before, the diagrams are simplified, and so not all features are shown. For example, none of the error checking logic is shown, nor is the gating that prevents undesired simultaneous operations. Flip-flop resets are not shown unless they are particularly significant, and clock inputs are not shown at all. Functionally similar logic has been combined in some cases, whereas in actual fact some circuits are duplicated in the interest of speed.

### TO FETCH NEXT INSTRUCTION

CPU TRANSMIT. The first step in fetching an instruction is to send an address to memory and tell memory what to do with that address (read contents and send back to CPU). The following three paragraphs describe this step.

When a NEXT micro-order is decoded from the ROM Skip field, a NEXT signal loads the contents of the P-register (address of instruction to be fetched) into the CPU Output Register. NEXT also transfers the Next Instruction Register contents into the Current Instruction Register (CIR). The CPU may proceed to execute the CIR contents while the following operations are in progress.

The objective now is to refill the Next Instruction Register. Assuming that the transmission may proceed, NEXT sets the LREQ (Low Request) flip-flop in the MCU. (The difference between *low request* and *high request* is that low request always checks to see if the destination module is ready to receive a transmission; high request assumes that the destination module is expecting the transmission, so readiness is not checked.) By this time, the MCU Operation Decoder has encoded the appropriate *memory opcode* (MOP), which is now in the MOP register. The memory opcode is a two-bit code which tells memory what to do when it receives bus data. The four possible codes are NOP (No Operation), CW (Clear/Write), RR (Read/Restore), and RNW (Read/No Write). In this case the memory opcode is RR. NEXT locks this code in the MOP register, and sets the NIP (Next In Process) flip-flop. Setting NIP "opens" the Next Instruction Register, so that it will load all central data bus transmissions until told to stop (by resetting NIP, later). NEXT also locks the TO register, which now contains the destination module number from the mapper.

The LREQ signal reads the TO register contents into the Ready Comparator, which checks the RDY (Ready) line from the intended destination to see if that module is ready to receive. If not, nothing further happens until the RDY line is true. The output of the Ready Comparator (through a set of changeable jumpers) pulls low on the Enable (ENB) line for this module number. Since each module cannot transmit unless all ENB lines of higher priority modules are high, this pulling low on one ENB line disables all lower priority modules (those with higher module numbers). Provided that no higher priority module has pulled low on its ENB line to this module (through a second set of jumpers), and provided the I/O Processor is not requesting the bus, the output of the Ready Comparator now sets the Select

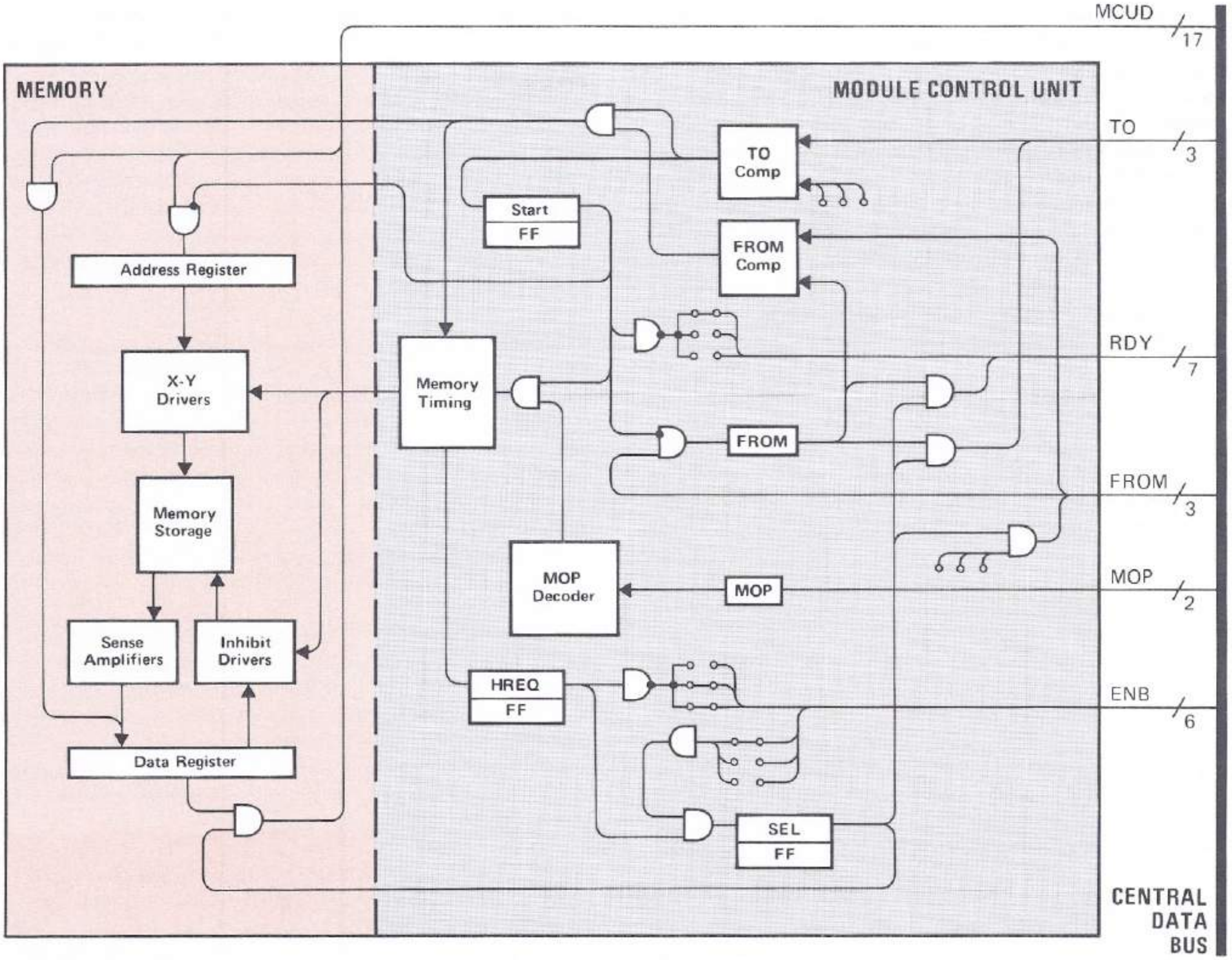


Figure 8-6. A Typical Memory Module

(SEL) flip-flop. The SEL signal reads out the CPU Output Register contents to the central data bus, as well as the TO and FROM module numbers and the memory opcode. SEL also pulls low on the destination module's RDY line for one cycle, so that other modules will not assume the memory module is ready before memory has a chance to pull the RDY line low itself on the next cycle.

MEMORY RECEIVE AND TRANSMIT. The next step in the process is for memory to receive the address from the bus, read the contents of the addressed location, and transmit the contents back to the CPU. The following two paragraphs describe this step. See figure 8-6.

The TO Comparator identifies the code on the TO lines as its own module number and sets a Start flip-flop. The Start signal locks the address word from the bus into the address register, and locks the FROM bits into the FROM register. The Start signal also keeps the module's RDY line pulled low (the CPU had pulled it low temporarily in the preceding cycle), and together with the decoded memory opcode begins the read/write memory cycle. The X-Y drivers begin to read the contents of the addressed memory location into the data register, via the sense amplifiers. Meanwhile, after a fixed delay, the MCU begins the process of requesting access to the bus by setting the HREQ flip-flop. (Since memory transmits only to modules that are expecting the transmission, only high requests are used.) The HREQ signal pulls low its ENB line to lower priority modules and, provided no higher priority module has pulled low on its ENB to this module, sets the Select flip-flop.

By this time, the memory location contents are in the data register, and the SEL signal reads the contents out to the central data bus. SEL also reads out the wired FROM code and the TO code (which is simply the saved FROM code, since transmission is back to the CPU).

CPU RECEIVE. The last step in the process is for the CPU to receive the instruction word, which is now on the central data bus, and load it into the Next Instruction Register. The following paragraph describes this step. Refer back to figures 8-3 and 8-4.

The TO Comparator identifies the code on the TO lines as its own module number, and gives a true output. Also, the FROM Comparator identifies the transmission as the one it was waiting for by comparing the saved TO register contents with the FROM lines of the bus; it therefore also gives a true output. (If the FROM code is not the expected one, it is loaded into the FROM register, and the bus information is processed as an interrupt from the identified module.) The two true outputs together reset the NIP flip-flop. The Next Instruction Register, which up until now has been freely loading all bus transmissions into itself, is now inhibited from further loading, since it now contains the expected next instruction.

## TO FETCH AN OPERAND

The procedure for fetching an operand from memory is very similar to the procedure for fetching an instruction. The main differences are that the initiating signals are different, and the receiving register is the Operand (OPND) Register rather than the Next Instruction Register. The following descriptions are therefore somewhat abbreviated, primarily giving the overall flow of information. Refer back to the preceding descriptions if further logical details are necessary.

CPU TRANSMIT. The process of sending an address to memory begins when a BUSL (Bus Low) signal from the ROM Store field loads the U-bus contents into the CPU Output Register and sets the LREQ flip-flop. The MCU Operation Decoder gives a memory opcode to the MOP register and sets the OPINP (Operand in Process) flip-flop. The OPND register now begins to load all bus transmissions. The LREQ signal causes the Ready Comparator to check if the destination module is ready and, if so, enters the priority structure. When priority allows (ENB present), the Select flip-flop is set, causing the address in the CPU Output Register to be read out to the central data bus.

MEMORY RECEIVE AND TRANSMIT. The memory module, after recognizing its TO code and setting the Start flip-flop, locks the address from the bus into the address register. The Start signal, together with the decoded memory opcode, initiates the reading of the addressed location into the data register. Meanwhile, the HREQ flip-flop is set and priority is established. When ENB is present, the Select flip-flop is set causing the operand, now in the data register, to be read out to the central data bus. The saved FROM code is used to identify the destination (TO) as the CPU module.

CPU RECEIVE. The TO and FROM Comparators together cause the OPINP flip-flop to reset, thus locking the operand from the bus into the OPND register.

## TO STORE AN OPERAND

Storing an operand in memory involves much the same logic operations that were discussed in the preceding fetch transmissions. The main difference here is that instead of being a round trip, CPU to memory and then memory to CPU, there are two consecutive transmissions from CPU to memory. The first transmission is the address, the second is the operand. The following paragraphs, again condensed to illustrate the overall flow of information, describe these transmissions.

CPU ADDRESS TRANSMIT. A BUSL signal from the ROM Store field loads the U-bus contents into the CPU Output Register and sets the LREQ flip-flop. The MCU Operation Decoder gives a memory opcode to the MOP register; in this case the opcode is Clear/Write rather than Read/Restore as in the previous cases. (Neither NIP nor OPINP flip-flops are set.) After checking if the destination module is ready and ENB is present, the LREQ signal causes the Select flip-flop to be set. This reads out the address to the central data bus.

MEMORY RECEIVE. The memory module, after recognizing its TO code and setting the Start flip-flop, locks the address from the bus into the address register. The Start signal, together with the decoded memory opcode, causes a "clear" half-cycle. The Start flip-flop remains set, and the FROM, MOP and address registers remain locked. Also the RDY line remains low, so no other modules may send a new address to this memory module.

CPU DATA TRANSMIT. The CPU, meanwhile, has put the operand on the U-bus, and a DATA signal from the ROM Store field loads it into the CPU Output Register. The DATA signal also sets the HREQ flip-flop. (Destination readiness does not need to be checked, since memory is expecting a data transmission from this module.) After priority checks, the HREQ signal sets the Select flip-flop, which reads out the operand to the central data bus. (The memory opcode is NOP, since memory is already holding the appropriate opcode.)

MEMORY RECEIVE. In the memory module the TO Comparator recognizes its TO code and the FROM Comparator verifies transmission from the correct module. The true outputs from both of these comparators cause the operand from the bus to be loaded into the data register, and additionally cause the memory timing to proceed with the second half of the clear/write memory cycle. This causes the operand to be stored into the addressed location.

## TO COMMAND A MODULE

The instruction set includes an instruction, CMD, which permits privileged executive programs to issue commands directly to a module (assuming the module is equipped to handle such commands). When programmed, the CMD instruction takes a 16-bit word from the top of the stack and sends it to a module whose module number (and two-bit opcode) are given in another word in the stack. (See CMD instruction definition.) The logic operations involved in this type of transfer are described in the following paragraph.

A BUSH signal from the ROM Store field loads the word containing the opcode and intended module number into the CPU Output Register. The TO code is the CPU's own module number so that, after select occurs, the CPU transmits to itself. The five effective bits from the CPU Output Register are loaded by a CRL (Control) signal into the CMD (Command) and CTO (Command TO) registers in the MCU. The CPU, meanwhile, has read the top-of-stack word onto the U-bus, and a BUSL signal from the ROM Store field loads this word into the CPU Output Register. A CMD signal from the MCU Operation Decoder enables the CMD and CTO registers to be read out when select occurs, rather than MOP and TO respectively. Thus when the Select flip-flop is set, the 16-bit word in the CPU Output Register is transmitted to the module specified by CTO, with the CMD opcode on the MOP lines.

# I/O SYSTEM

The remainder of this section deals with components of the input/output system. Before proceeding with detailed descriptions, an overall view of the I/O system will be presented. First, an overall discussion of I/O priorities is given, followed by a summary of data routes and a comparison of basic transfer modes. Figures 8-7, 8-8, and 8-9 are used as the bases of these discussions.

## I/O PRIORITIES

There are two types of priority to be considered in the I/O system: interrupt priority and service priority. That is, the ability of a device to interrupt the CPU is based on a priority structure that is separate and distinct from the priority structure that handles service requests.

Figure 8-7 partially illustrates the priority structure, showing the use of "polls" to establish priority. (This figure is a modified copy of figure 8-1.)

The *interrupt poll* determines the priorities of all I/O interrupts. As shown in figure 8-7, the interrupt poll originates in the I/O Processor and is wired in series through every device controller in the system. The proximity to the I/O Processor on this line determines the interrupt priority of each controller. The desired wiring sequence is dependent on system configuration. Physically, the interrupt poll is a twisted-pair wire (signal and ground) connected into and out of each unit at INT POLL IN and INT POLL OUT terminals. Functionally, the interrupt poll is an I/O Processor response to a received Interrupt Request (INTREQ line in the IOP bus). The poll propagates through each non-requesting unit and stops at the first requesting unit it encounters. That unit will then return INTACK (Interrupt Acknowledge) and its device number to the I/O Processor. The I/O Processor accordingly generates an interrupt signal to the CPU. When the CPU is ready to process the interrupt, it will use the device number saved in the I/O Processor (Interrupt DEVNO register) to refer to the device.

Service priority, unlike the simple series-linked structure of interrupt priority, is determined in two steps. For Multiplexer Channel devices, the first level determines the priority among two or more Multiplexer Channels. The second level determines the priority of each device controller associated with that Multiplexer Channel. Figure 8-7 shows only the first-level determination of priority among Multiplexer Channels by means of a *data poll*; the remaining priority determination is by logic which has not been detailed in the figure. The data poll operates very much like the interrupt poll. That is, when the I/O Processor receives a Service Request, it sends out a data poll. The first requesting Multiplexer Channel encountered by the poll stops propagation of the poll, and proceeds to specify the kind of service required. Since priority is therefore determined by proximity to the I/O Processor, the poll is wired through each Multiplexer Channel in the desired priority sequence.

The second-level priority determination for Multiplexer Channel devices is by a *service request number*. Since each Multiplexer Channel can handle 16 device controllers, there are 16 service request numbers (0 through 15). Each device controller associated with a given Multiplexer Channel is uniquely wired by a jumper to connect to one of these 16 numbers. This, then, gives the device controller a specific priority level. Service request number 0 is highest priority; 15 is lowest priority.

(The service request number has no association with the device number. It is simply a convenient means by which a Multiplexer Channel can communicate with and assign priorities to its set of device controllers.)

For high-speed device controllers, the Port Controller determines the first level of priority. Selector Channel 1 has highest priority and Selector Channel 4 has lowest priority. The second-level determination is a simple preemptive process: the first device to be given an SIO instruction, on a particular channel, will have exclusive use of that channel until its I/O program is finished. No further SIO instructions for devices connected to that channel can be honored until that time.

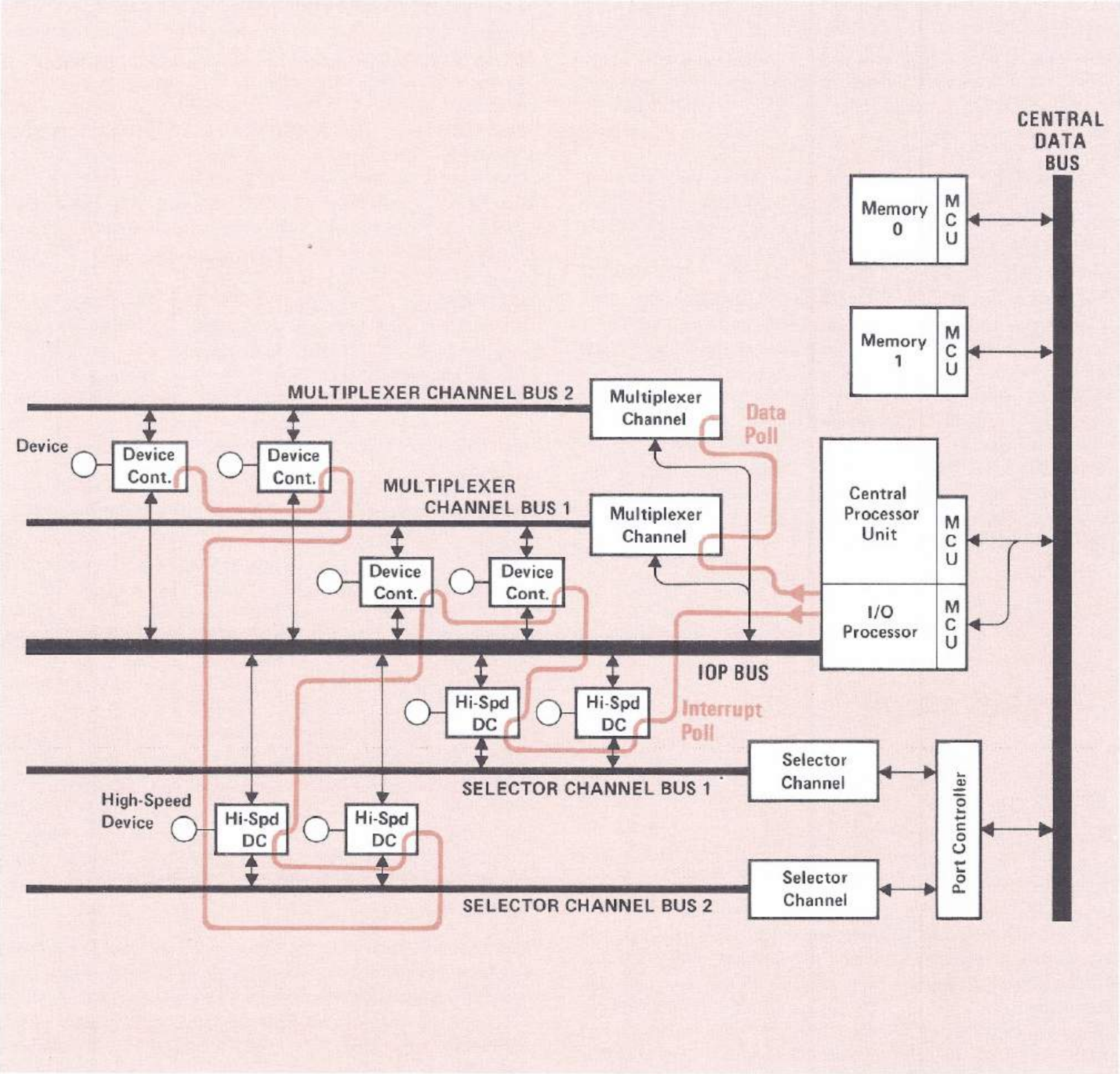


Figure 8-7. Interrupt Poll and Data Poll

## I/O DATA ROUTES

Figure 8-8 illustrates data transfer routes for both low-speed and high-speed devices and for both direct I/O and SIO type instructions. The ten blocks represent one of each type of unit (one low-speed device controller, one Multiplexer Channel, one memory module, etc.), and correspond to the ten simplified logic diagrams presented in this section.

MULTIPLEXER CHANNEL DEVICE. For direct I/O instructions, information is transferred to or from the top of the stack in the CPU via the I/O Processor and IOP bus. The information could be device status (for TIO or rejected SIO, RIO, or WIO), control information (CIO), or data (RIO or WIO). For SIO operation, data is transferred to and from memory by way of the central data bus, I/O Processor, and IOP bus.

SELECTOR CHANNEL DEVICE. For direct I/O instructions, the data route is the same as for Multiplexer Channel devices: to or from the top of the stack in the CPU via the I/O Processor and IOP bus. For SIO operation, data is transferred to and from memory by way of the central data bus, Port Controller, Selector Channel, and channel bus.

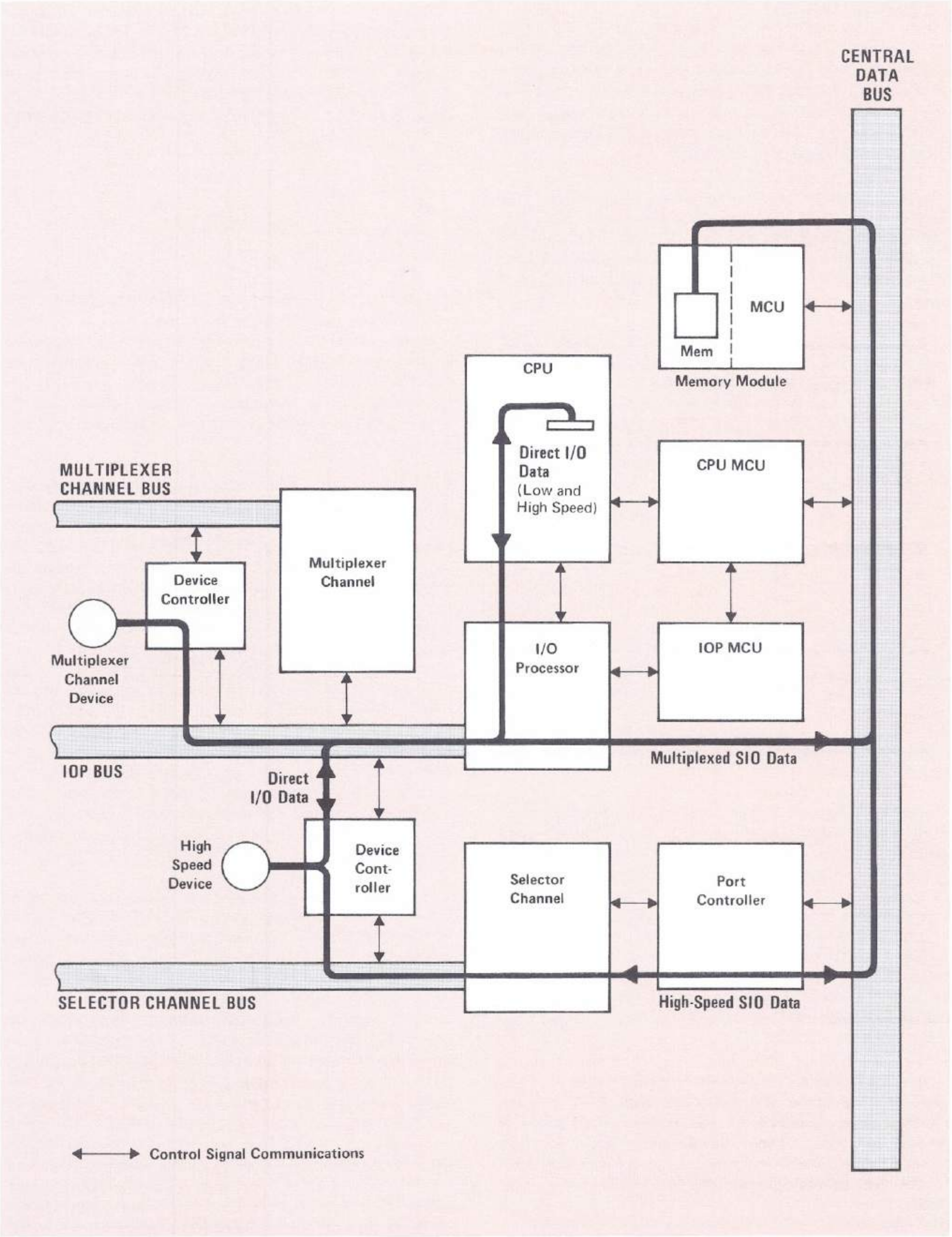


Figure 8-8. Input/Output Data Routes

## TRANSFER MODES

There are three basic modes of data transfer. One, direct I/O, is relatively uncomplicated, consisting of the transfer of a single word (per CPU instruction) between the CPU and a device controller; the Multiplexer Channel and Selector Channel are not involved. Direct I/O operation will be described at the end of this section.

The other two transfer modes are SIO-type transfers. That is, the CPU gives the I/O system a command to "start I/O" for a particular device, and the I/O system proceeds to execute an I/O program for that device. The program, which resides in memory, controls the input and output of data.

Specifically the two SIO modes are: moderate-speed transfers via the Multiplexer Channel, and high-speed transfers via the Selector Channel. Figure 8-8 illustrates the difference in data routes for these two modes; however, the significant difference is in the sequencing of transfers for multiple device controllers. The following paragraphs describe the differences between a multiplexer and a selector, with reference to figure 8-9.

MULTIPLEXER. A *multiplexer* transfers data from many sources on an apparently simultaneous basis. Thus it is the function of the Multiplexer Channel to perform one discrete operation for one device controller (such as to transfer one word to or from memory), and then check to see which device controller has highest priority for the next discrete operation.

Referring to figure 8-9, note that the Multiplexer Channel includes a 16-cell solid-state memory. Each location in this memory corresponds to one of the 16 device controllers connected to the multiplexer channel bus, and at all times it contains the information required to execute the next operation for that device. Typically this would be the current I/O program word. When a particular device controller is selected for service, the stored word is read out to a set of registers and the Multiplexer Channel proceeds to execute the indicated operation. Then the information is updated for the next anticipated operation and is stored back in the memory cell.

The overall Multiplexer Channel operating sequence is as follows. Each time a device controller requires a new I/O program word, it causes the Multiplexer Channel to fetch an address from the Device Reference Table (1) and loads it into its solid-state memory location. (Some other operation for another device could be interleaved after each of these steps.) Then (2), the I/O program doubleword is fetched and loaded into the same memory location. This I/O program word is then read out (3), control signals are issued to the device controller (4), and the updated operation information is stored back into the memory cell (5). If the device controller was commanded to transfer data, it issues a service request when it is ready (6), causing another read-out of the stored information (7) and a transfer of data (8); updated operation information is re-stored (9). Steps 6 through 9 are repeated for each word transferred.

SELECTOR. A *selector* transfers data from many sources in a data block manner. That is, it locks onto one device controller until I/O program for that device is completed. Then a check is made to see which device controller has highest priority for the next block transfer. Since only one I/O program will be in progress as long as a particular device is selected, the selector is designed to facilitate very high speed transfers.

As shown in figure 8-9, the Selector Channel uses double-buffering for both data and I/O program words. For data, this permits device/channel transfers to overlap channel/memory transfers. For I/O program words, this permits the next program word to be fetched from memory while the current word is active. Both of these features contribute to the speed capability. In addition, the necessity to repeatedly fetch a DRT entry for the address of the current I/O program word (as is done by the Multiplexer Channel) is eliminated by including a Program Counter in the Selector Channel. The Program Counter is loaded with the initial address contained in the DRT, but is thereafter incremented (or altered for jumps) internally in the Selector Channel. To provide software compatibility with Multiplexer Channel transfers, the final value of the Program Counter is automatically re-stored in the DRT at the end of the program. Software cannot distinguish whether the transfer occurred by way of the Multiplexer Channel or the Selector Channel.

The overall Selector Channel operating sequence is as follows. When the device controller is commanded by the CPU to "start I/O", it causes the Selector Channel to fetch the starting address of the I/O program from the Device Reference Table (A). This address is used to fetch an I/O program doubleword (B) and load it into either the active control registers or, during order prefetch, into the buffers (C). The Program Counter is incremented after each fetch. Control signals are issued to the device controller (D), and (E) if the command is a "read", the device controller reads data into buffer A (or buffer B if A is full); if the command is a "write", the device controller writes data from buffer A (or buffer B if A is empty). Meanwhile (F), the Selector Channel attempts to keep both buffers full for output or both empty for input, by transmissions to or from memory. At the end of the block transfer, the next I/O program word is fetched (repeat back to step B). At the end of the I/O program, the Selector Channel stores its Program Counter contents into the Device Reference Table (G).

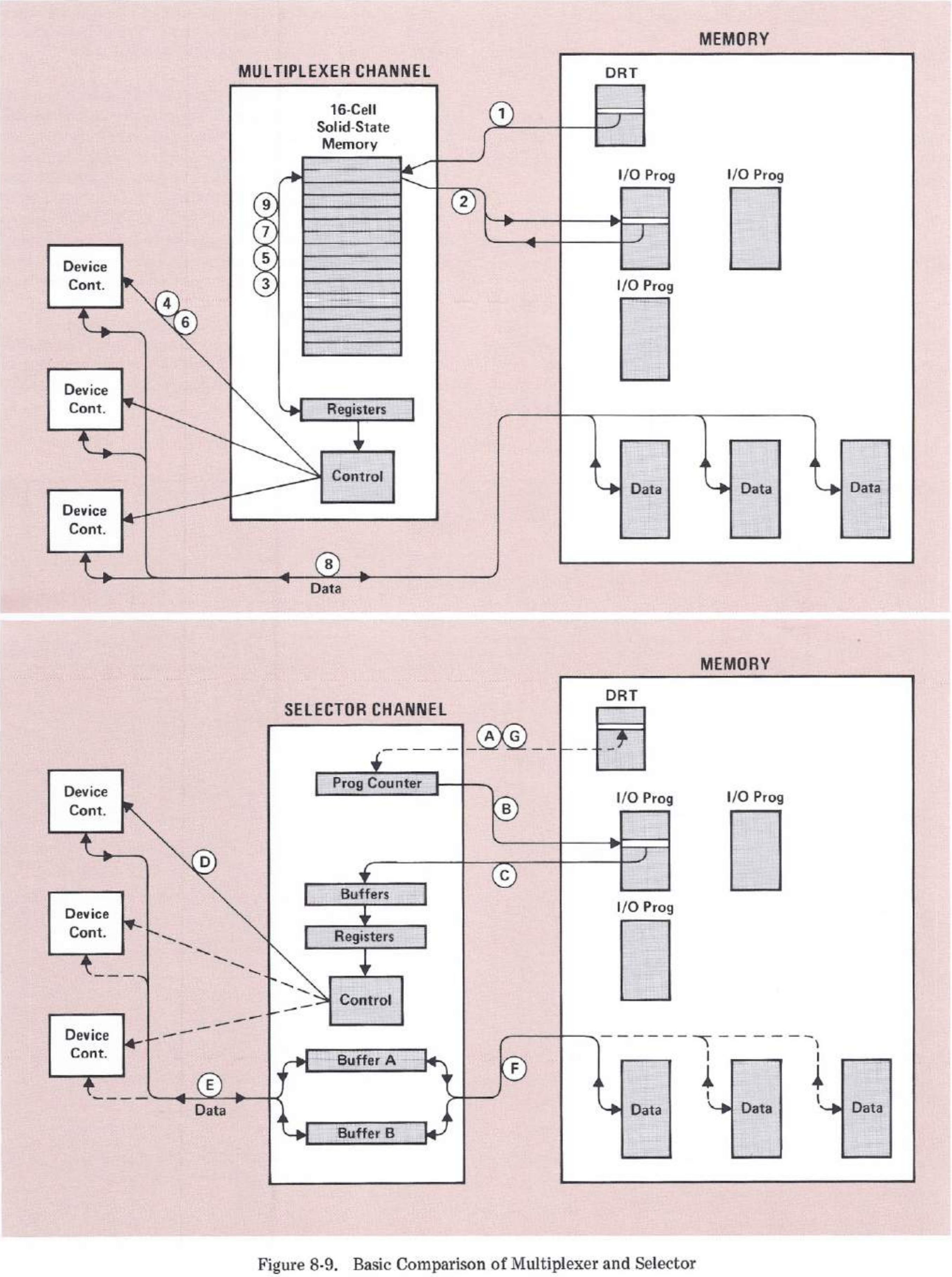


Figure 8-9. Basic Comparison of Multiplexer and Selector

# I/O PROCESSOR

Figure 8-10 is a simplified logic diagram of the I/O Processor portion of the CPU/IOP module. The signal lines at the left of the diagram are the IOP bus. The lines at the top connect to the CPU (see figure 8-3). Figure 8-11, shown facing figure 8-10 in order to show MCU interconnections, will be discussed later under the heading IOP Module Control Unit.

## IOP LOGIC

Basically, the functions of the I/O Processor are to: 1) execute direct I/O instructions and pass the results to the CPU, and 2) transfer data and I/O program words between memory and device controllers, so that the CPU may continue to execute other instructions without further intervention. The operations performed by the I/O Processor will be seen throughout the remainder of this section, when actual transfer sequences are discussed. The following paragraphs describe the blocks identified in figure 8-10.

IOP CONTROL REGISTER. This register receives the I/O instruction information, which has been combined by the CPU into a single word. The instruction code from the code segment has been translated into a 3-bit command (IOCMD). This can now be read out onto the IOCMD lines of the IOP bus. The device number has been obtained from the stack, and can now be read out on the DEVNO lines of the IOP bus. The SO bit (Service Out) tells the addressed device to accept and respond to the accompanying information. (The device controller must return SI, Service In.)

IOP CONTROL. This block represents sequencing logic for transfers between the device and memory, and between the device and the CPU. Each of the lines shown entering or leaving this block will be discussed later when transfer sequences are described.

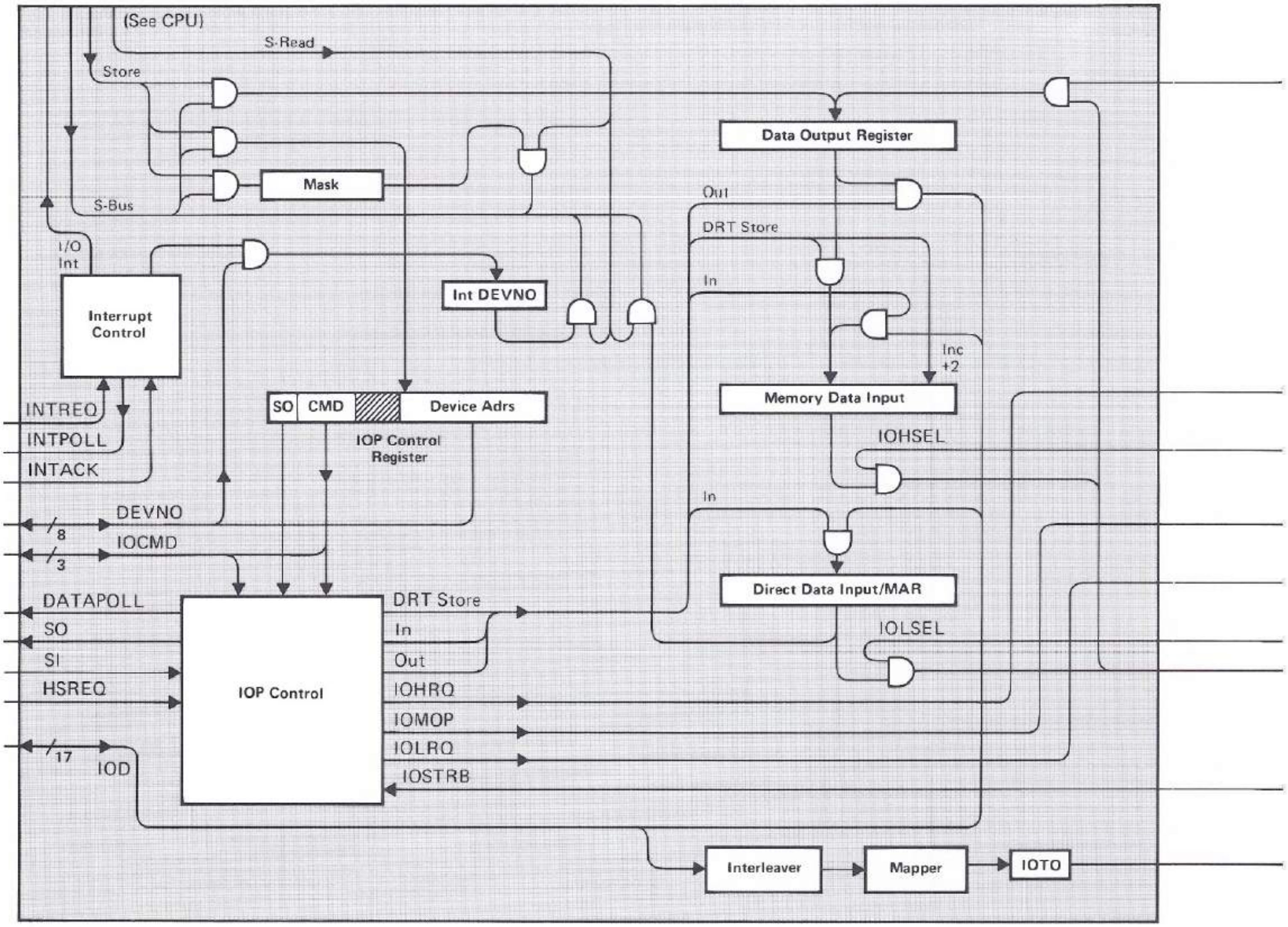


Figure 8-10. I/O Processor

INTERRUPT CONTROL. The interrupt control logic accepts an Interrupt Request (INTREQ) from the device controllers on the IOP bus, interrogates the device controllers with INTPOLL to find the highest-priority request, and, when Interrupt Acknowledge (INTACK) is received, loads the device address into the Interrupt DEVNO register. It then issues an interrupt (I/O Int) signal to the CPU.

INT DEVNO. The Interrupt DEVNO register holds the device number of the interrupting device so that, upon command, the CPU may read the contents onto its S-bus for interrupt processing.

DATA OUTPUT REGISTER. There are actually two Data Output Registers, one for memory data received from the central data bus, and one for direct data received from the S-bus of the CPU. For simplicity figure 8-10 combines the two registers into one. Signals from IOP Control can either read the contents out onto the IOP bus (OUT), or transfer the contents into the Memory Data Input register (for re-storing a DRT entry).

DATA INPUT REGISTERS. There are two input registers. The Memory Data Input register is used for sending data to memory via the central data bus. This register is loaded either from the IOP bus (In) or, for DRT entry re-storing, from the Data Output Register. When doing a DRT store, the Memory Data Input register is incremented by two before the transfer is made. The second input register may be used either as a Direct Data Input register or as a Memory Address register (MAR). It is loaded from the IOP bus. When direct I/O is being executed, the register contents are read onto the CPU S-bus. When addressing memory, the register contents are read out to the central data bus.

INTERLEAVER AND MAPPER. These circuits are the same as described earlier for the CPU, under the Arithmetic

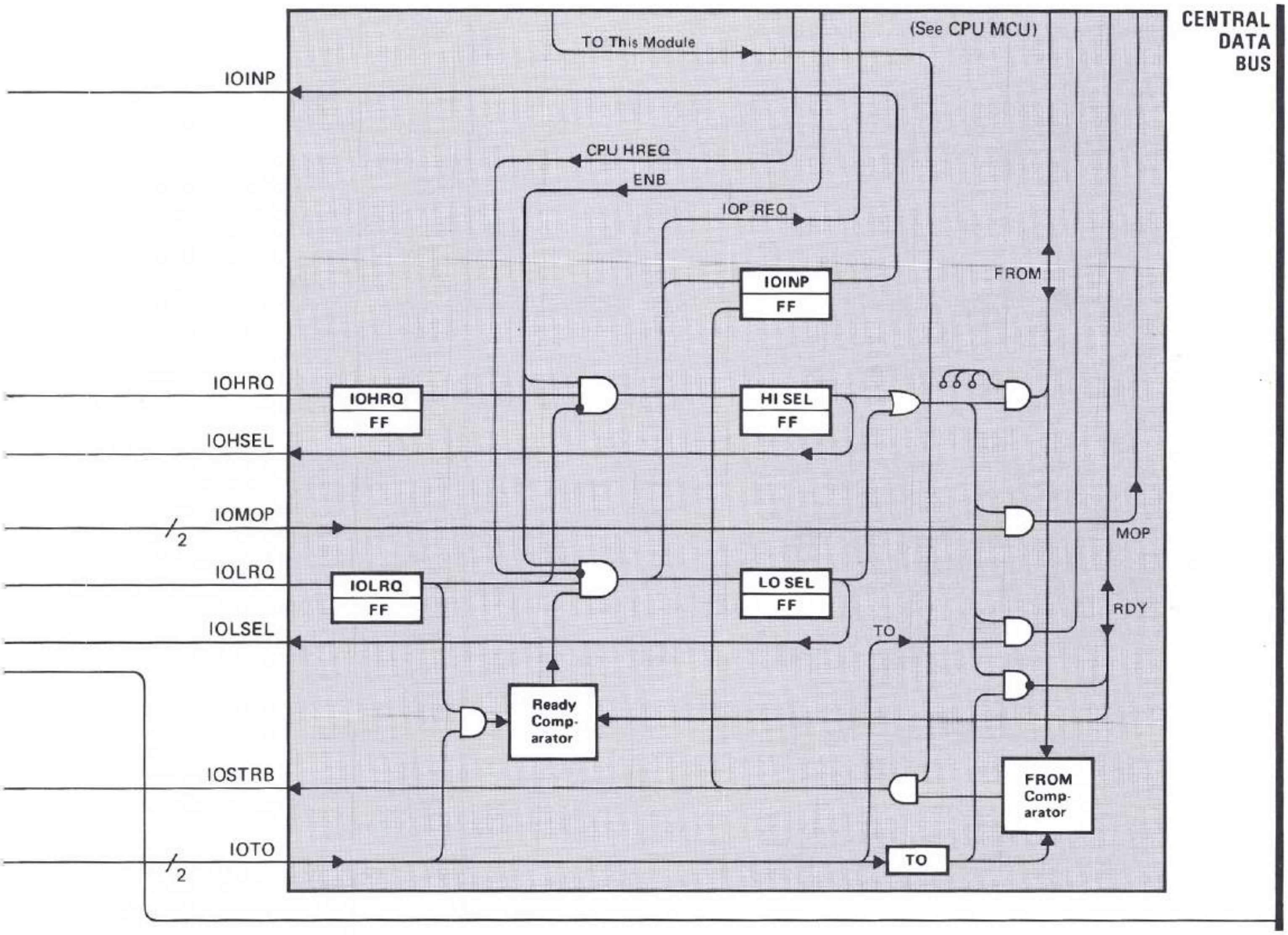


Figure 8-11. IOP Module Control Unit

Logic heading. The purpose of these circuits is to derive an appropriate module number when transmitting to memory. The memory module number for each transmission is loaded into the IOTO register.

### IOP MODULE CONTROL UNIT

Figure 8-11 illustrates the Module Control Unit for the I/O Processor. This MCU is a simplified form of the CPU MCU discussed earlier. Both of these MCUs, in fact, are physically located on the same printed-circuit card. They operate basically in parallel, but not independently. Since both MCUs share the same access to the central data bus, it is necessary to resolve priority when both IOP and CPU simultaneously attempt to use the bus.

Priority is resolved such that all IOP requests take precedence over CPU requests, except that a CPU high request takes precedence over an IOP low request. This exception means simply that the CPU is in the middle of a transfer, having sent an address to memory, and the high request is an attempt to follow up by sending the data. The CPU low request, on the other hand, represents the beginning of a transfer (attempt to send an address) and so is of lesser importance.

Note the logic in figures 8-11 and 8-4 which accomplishes this priority resolution. In figure 8-11, the IOP REQ is generated when either a low request (IOLRQ) or a high request (IOHRQ) is about to set one of the Select flip-flops (LO SEL or HI SEL). This signal, in figure 8-4, inhibits the CPU's Select flip-flop from being set. Note, however, that a CPU HREQ signal from the CPU can inhibit IOLRQ from generating the IOP REQ signal.

The IOINP flip-flop provides a function similar to the NIP and OPINP flip-flops in the CPU MCU. IOINP (I/O In Process) is set when a request sets LO SEL, if the memory opcode (MOP) is Read/Restore. When data is returned from memory the FROM Comparator in figure 8-11 checks that the transmission is from the same memory module that the address was sent to (by comparing with the contents of the TO register). Also, the TO Comparator in figure 8-4 checks that the transmission is to "this module". Together, the outputs of these two comparators generate an IOSTRB (I/O Strobe) signal which resets the IOINP flip-flop. This causes the IOP to lock the Data Output Register, since it now contains the correct information from the central data bus. IOSTRB also tells IOP Control that the data is ready for output via the IOP bus.

The Ready Comparator checks if a destination module is ready, so that an I/O low request can set the Low Select (LO SEL) flip-flop. Setting the LO SEL flip-flop causes the contents of the Data Input Register, FROM, TO, and MOP to be read out onto the central data bus for transmissions to memory.

## MULTIPLEXER CHANNEL

As explained earlier under the heading of Transfer Modes, the purpose of the Multiplexer Channel is to execute the I/O programs of up to 16 devices on a multiplexed (word-by-word) basis. All data transfers for these 16 devices are also multiplexed on a word-by-word basis. A wired-in service request number in each device controller determines its priority in being serviced.

Figures 8-12 and 8-13 show, in simplified form, the logic which accomplishes this purpose. Figure 8-13 is the Multiplexer Channel and figure 8-12 shows one device controller connected to the multiplexer channel bus (top of diagram). The IOP bus runs across the bottom of both diagrams and connects to the I/O Processor at the right. (See figure 8-10.)

The following descriptions, which refer to these two figures, describe the major operations that were outlined briefly under the Transfer Modes heading.

### INITIALIZE

When the CPU encounters an SIO instruction the CPU, under control of its SIO microprogram, outputs a command word to the IOP Control Register. (See figure 8-10.) The I/O Processor, in turn, relays this information to the device controller (figure 8-12) via the IOP bus. Note in figure 8-12 that the device number on the bus (DEVNO) is compared with the internal wired device number. A true result, together with the SO (Service Out) signal from the I/O Processor, enables the IOCMD (I/O Command) to be decoded. The IOCMD in this case is SIO which, when decoded, sets the SR (Service Request) flip-flop.

The Service Request is sent via the multiplexer channel bus to the Multiplexer Channel. (Since an SIO to a controller temporarily inhibits service requests from all other controllers, the only controller requesting is the one receiving the SIO.) The Priority Encoder then issues a 4-bit binary code which corresponds to the Service Request line number. The binary code is used as a "RAM Address", to enable one of the 16 locations in the solid-state memory. The solid-state memory consists of three separate "RAMs", or *Random-Access Memories*, one each for the IOCW and IOAW parts of the I/O program doubleword, and one to specify the "state" (or next operation) — in this case a DRT fetch. The IOCW is contained in the Order RAM (16 bits), the IOAW is contained in the Address RAM (16 bits), and the state is contained in the state RAM (4 bits). Each of the 16 addressable locations therefore contains 36 bits.

For the initialize operation, the State RAM location for the requesting device is forced to the condition required for a DRT fetch. Once this is done, the Multiplexer Channel returns SI (Service In) to the I/O Processor.

## DRT FETCH

The Service Request received at the Multiplexer Channel from the device controller causes HSREQ (the Multiplexer Channel's service request) to be issued to the I/O Processor, and also sets the SR Latch. Any of the 16 SR inputs can set this latch and generate HSREQ; however, only the highest priority request will be honored by the Priority Encoder.

When the I/O Processor receives HSREQ, it issues DATAPOLL to all Multiplexer Channels. The highest priority Multiplexer Channel stops the propagation of the poll (since SR Latch is set), and its transfer logic is enabled. First, the contents of the addressed RAM location are loaded into the State, Address, and Order Registers. The State bits tell the transfer logic to send out a command to the device controller via the multiplexer channel bus, along with the service request number signal (which is returned on the same line used for Service Request) and CHANSO (Channel Service Out). This command tells the device controller to read out its device number to the IOP bus.

Note: The approximately 20 command and response lines shown as part of the multiplexer channel bus have not been individually identified, as they represent greater detail than is required at this level of discussion.

The device controller, for a DRT fetch, reads out its device number (Shifted DEVNO) onto the IOD lines. Instead of being read onto the eight least significant lines of the bus (8 through 15), the number is read onto lines 6 through 13, which is left-shifted by two bits. This effectively multiplies the number value by four, thus automatically providing the correct address for that device's DRT entry. (Remember that each device uses four locations in the DRT.)

Meanwhile, the Multiplexer Channel is returning an SI (Service In) response to the I/O Processor, along with an IOCMD (I/O Command) which tells the I/O Processor to accept the address existing on the IOD lines, and that a DRT fetch from that address is required.

Now the I/O Processor proceeds to fetch the DRT entry, as follows. (Reference can be made to figures 8-10 and 8-11.) The I/O Processor issues IOLRQ to its MCU, with an appropriate MOP to read memory. When Select occurs, the address is transmitted to memory, and when memory returns the DRT entry contents, IOSTRB loads the word into the Data Output register. The contents of this register are then read out onto the IOD lines, and SO is issued.

On receiving SO, the Multiplexer Channel loads the DRT word into the Address RAM, re-stores the Order register contents into the Order RAM, and sets the State RAM to the condition required for an I/O program word fetch.

The I/O Processor, meanwhile, transfers its copy of the DRT word from the Data Output register to the Data Input register, increments it by two, and sends it back to the DRT in memory. (This is an anticipatory move, as the Address RAM presently contains the desired address for the next operation; the incremented address in the DRT will not be used until the next DRT fetch.)

At this point the DRT fetch operation is complete. Some other operation for another device could be interleaved here.

## I/O PROGRAM WORD TRANSFERS

Each I/O program word consists of two words in memory, the IOCW (I/O Command Word) and the IOAW (I/O Address Word). Therefore two memory transfers are required. The first transfer is to fetch the IOCW. Depending on the order that the IOCW contains, the second transfer may be either a fetch or a store. The differences will be pointed out in the following descriptions.

IOCW FETCH. The SR flip-flop in the device controller is still set from the previous procedure, so HSREQ is still present at the I/O Processor. The I/O Processor therefore issues a new DATAPOLL. The SR Latch in the Multiplexer Channel, which had reset on the trailing edge of the previous SO, has become set again, since the SR input was still present at the next clock. Thus DATAPOLL is stopped from further propagation, and the transfer logic is enabled again.

Again, the contents of the addressed RAM location are loaded into the State, Address, and Order registers. The state specifies an IOCW fetch, so the transfer logic reads out the contents of the Address Register and issues SI and IOCMD ("transfer from memory") to the I/O Processor. The address now on the IOD lines is the word previously fetched from the DRT, indicating the address of the I/O program word.

The I/O Processor loads the address into the Memory Address Register (MAR) and issues IOLRQ to its MCU, with MOP (Read/Restore). The MCU, when priority allows, transmits the address to memory. When memory returns the IOCW, IOSTRB loads this word into the Data Output Register in the I/O Processor. The I/O Processor then reads the word out to the IOD lines and issues SO.

On receiving SO, the Multiplexer Channel loads the IOCW into the Order RAM. (If the order is Control, the Multiplexer Channel issues a command through the multiplexer channel bus, so that the device controller may also load the IOCW into its Control register.) The contents of the Address Register, incremented by one, are re-stored in the Address RAM, and the next state (fetch or store IOAW) is stored in the State RAM.

At this point the IOCW fetch is complete. Some other operation for another device could be interleaved here.

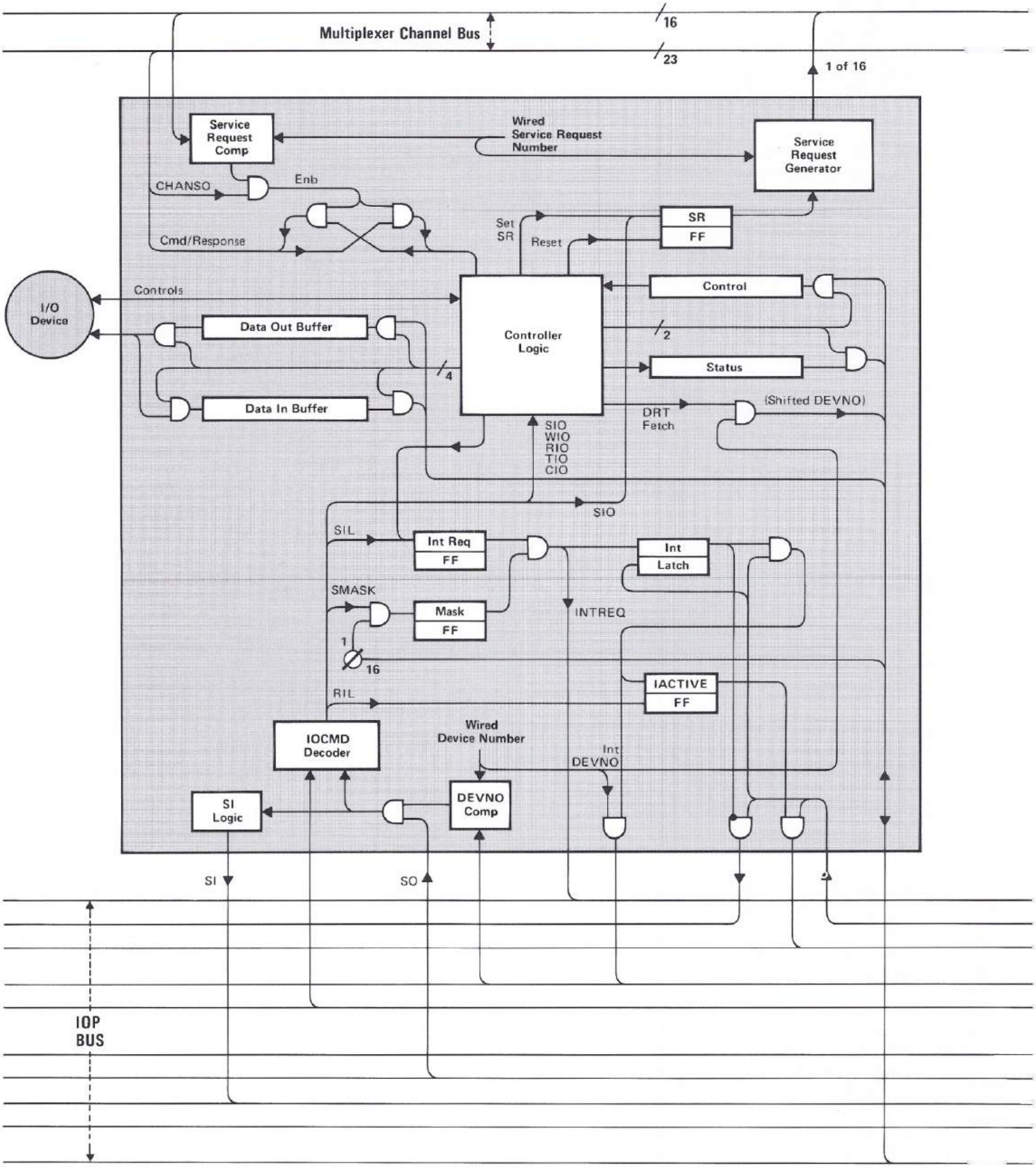


Figure 8-12. Multiplexer Channel Device Controller

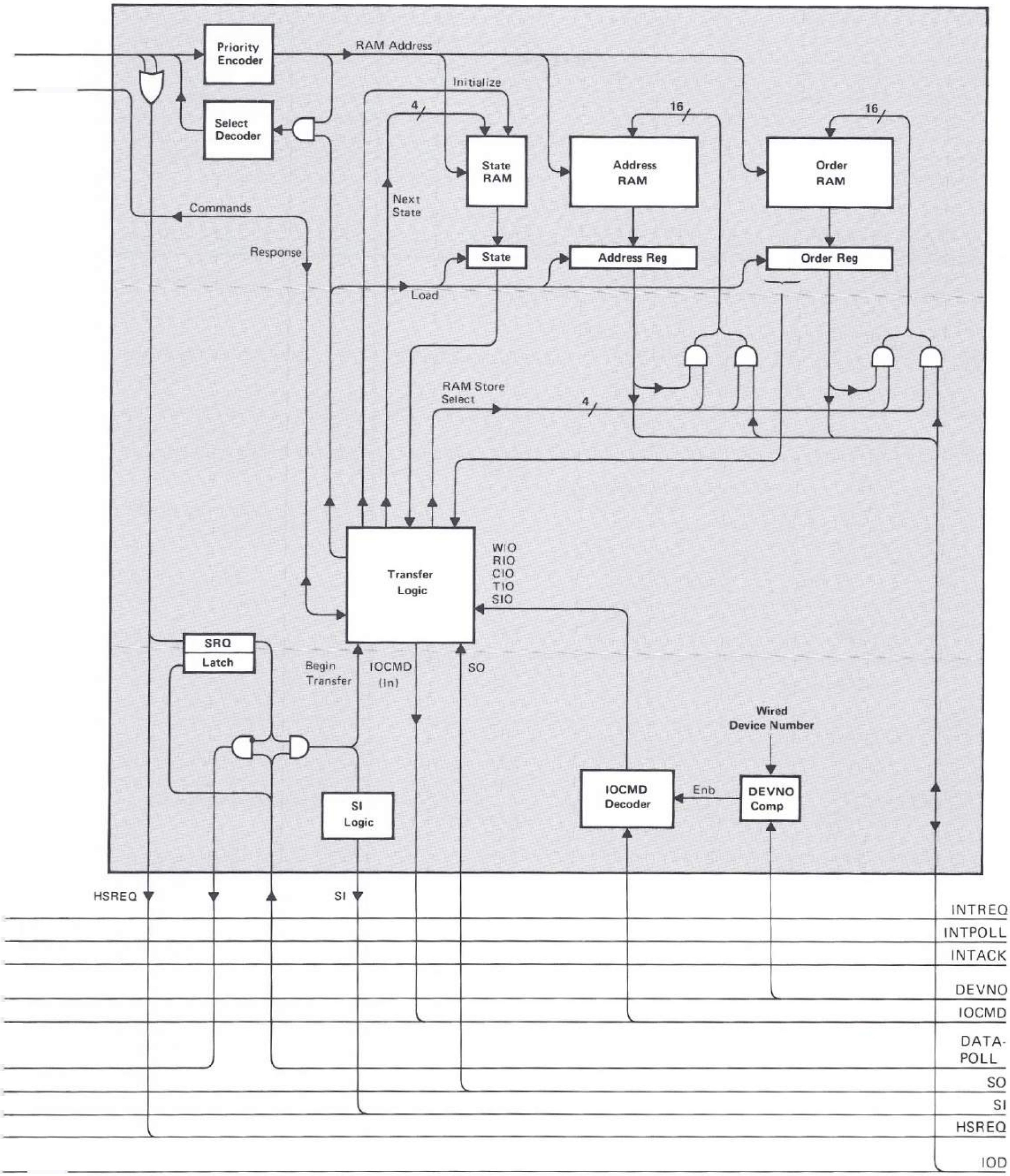


Figure 8-13. Multiplexer Channel

The next operation, transfer of the IOAW, begins the same way for each of the orders. That is, SR to the Multiplexer Channel causes HSREQ to the I/O Processor. The I/O Processor returns a DATAPOLL which enables the Multiplexer Channel to load the addressed RAM location into the State, Address, and Order Registers. The action after this point varies, depending on the order that the IOCW contains. The following paragraphs describe each of the various courses of action.

IOAW FETCH. The Read, Write, Jump, Control, and Interrupt orders each cause an IOAW fetch. However, the action taken on receiving the IOAW varies in each case, as will be pointed out.

The IOAW fetch begins by reading out the contents of the Address Register (incremented on the trailing edge of DATAPOLL in the IOCW fetch procedure) to the IOD lines. The Multiplexer Channel also issues SI and IOCMD ("transfer from memory") to the I/O Processor. The I/O Processor, in turn, issues IOLRQ with MOP to its MCU to request a memory read.

When memory returns the contents of the addressed location, IOSTRB loads it into the Data Output Register in the I/O Processor. The I/O Processor then reads out the contents of this register to the IOD lines and issues SO. For Read, Write and Jump orders, the Multiplexer Channel will store the word (IOAW) into the Address RAM. For a Control order, the Multiplexer Channel issues a command via the multiplexer channel bus to tell the device controller to load the word into its Control register. For an Interrupt order, the fetched information is loaded into the Address RAM but is disregarded.

In addition, for Read, Write, and conditional Jump, a command is sent to the device controller to specify conditions for the next action. For Read, the "in-transfer" condition is set. For Write, the "out-transfer" condition is set. For conditional jump, the controller is given the choice of setting or not setting the "jump met" condition. If "jump met" is true in the next DRT fetch sequence (or if an unconditional jump was given), a store operation (instead of fetch) will occur. That is, the Multiplexer Channel will cause the contents of the Address Register to be sent to the I/O Processor, which will increment the value by two before storing in the DRT. (The Address RAM already contains the correct jump address, so a DRT "fetch" is not necessary.)

IOAW STORE. The Sense, End, and Return Residue orders each cause an IOAW store operation. This operation begins as the Multiplexer Channel reads the incremented contents of the Address Register out to the IOD lines and issues SI with a "transfer-to-memory" IOCMD.

The I/O Processor Loads this address into its Memory Address Register (MAR) and issues IOLRQ to its MCU with a "Clear/Write" MOP. The ensuing central data bus transmission prepares memory for receiving data.

Meanwhile, the I/O Processor has issued SO to the Multiplexer Channel to ask for data. Depending on the current order, the Multiplexer Channel either gates the Order Register contents out to the IOD lines (Return Residue order) or issues a command to the device controller, telling it to read its Status register contents out (Sense or End orders). When either action occurs, SI is returned to the I/O Processor, which causes the I/O Processor to load the IOD information into its Memory Data Input register.

The I/O Processor then proceeds to transmit this information to memory by issuing IOHRQ to its MCU. When the transmission occurs, the appropriate information will be stored into the IOAW location of the I/O program doubleword.

NEXT OPERATION. At this point (after the IOAW fetch or store), the I/O program word transfer is complete. In addition, all orders except Read and Write (i.e., Control, Sense, Return Residue, End, Jump, and Interrupt) are fully executed. The next operation for any of these orders (except End, which terminates the program) is to return to the DRT fetch operation.

For Read or Write, however, a data transfer is indicated. Procedures for data transfers are next described.

## DATA TRANSFERS

Data transfers are very similar to the I/O program word transfers described above, in that the basic operation is to fetch or store information using a memory address that has been put in the Address RAM by a previous operation. (For I/O program word transfers, the previous operation was the DRT fetch; for data transfers, the previous operation is the I/O program word transfer.)

The main difference is that the data transfer is device-initiated. That is, when the device is ready for a transfer, it so informs its device controller, which then issues a Service Request to the Multiplexer Channel. Another difference is that the word count and memory address contained in the Order and Address Registers must be incremented during each word transfer.

Each data transfer consists of two distinct steps: the transfer of an address to memory, and the transfer of data to or from that address. The first step is the same for either output or input, and is described first. Output and input data transfers are then separately described, followed by the end-of-transfer operations.

ADDRESS TRANSFER. When the device sets the device controller's SR flip-flop, the SR signal to the Multiplexer Channel generates an HSREQ signal to the I/O Processor.

The I/O Processor returns DATAPOLL, which enables the Multiplexer Channel to begin its transfer. First, the addressed RAM location is read out to the State, Address, and Order Registers. Then the Address Register contents are read out to the IOD lines. Also SI and an appropriate IOCMD ("transfer to memory" or "transfer from memory") are sent to the I/O Processor.

The I/O Processor loads the address into its Memory Address Register (MAR) and issues IOLRQ to its MCU, with a "Read/Restore" or a "Clear/Write" MOP. When priority allows, the MCU will transmit the address to memory.

Meanwhile, the Multiplexer Channel resets the device controller's SR flip-flop, via the multiplexer channel bus, and increments the Address and Order Registers.

OUTPUT TRANSFER. When memory returns a data word, IOSTRB loads the word into the Data Output Register in the I/O Processor. The I/O Processor then reads the contents of this register out to the IOD lines and issues SO. On receiving SO, the Multiplexer Channel issues a command to the device controller via the multiplexer channel bus, telling the controller to load the word on the bus into the controller's Data Out Buffer. The device controller returns SI to the I/O Processor and proceeds to output the word to the device.

Meanwhile, the Multiplexer Channel re-stores the contents of the State, Address, and Order Registers into the RAM location, and the output data transfer is complete. Some other operation for another device could be interleaved here. Otherwise, the entire data transfer procedure repeats.

INPUT TRANSFER. As the input data transfer procedure begins, memory is expecting the data. The procedure begins when the I/O Processor sends SO to the Multiplexer Channel to ask for data. On receiving SO, the Multiplexer Channel issues a command to the device controller via the multiplexer channel bus, telling the device controller to read the contents of its Data In Buffer out to the IOD lines. When the controller does so, it also sends an SI response, which causes the I/O Processor to load the data into its Memory Data Input register. The I/O Processor then issues IOHRQ to its MCU, with a "Clear/Write" MOP, thus causing a data transmission to memory via the MCU bus.

Meanwhile, the Multiplexer Channel re-stores the contents of the State, Address, and Order Registers into the RAM location, and the input data transfer is complete. Some other operation for another device could be interleaved here. Otherwise, the entire data transfer procedure repeats.

END OF TRANSFER BY WORD COUNT. If the word count rolls over while incrementing (during the address transfer sequence), then in the data transfer sequence the Multiplexer Channel will issue a command which will reset the "in-transfer" or "out-transfer" condition in the device controller. Also an End-of-Transfer (EOT) signal accompanies the last command from the Multiplexer Channel to read or write. The controller logic will therefore not transfer any more data to or from the device. It will, however, issue one more SR.

In the Multiplexer Channel, the transfer logic sets the next state to "DRT fetch", when re-storing the RAMs at the end of the final data transfer. When the Multiplexer Channel receives the SR from the device controller, and when priority conditions are satisfied, a new DRT fetch procedure will begin. This advances the I/O program to the next IOCW.

END OF TRANSFER BY DEVICE. On termination of a transfer by a device, the controller will issue an SR to the Multiplexer Channel. When the Multiplexer Channel responds with the select code and CHANSO, the device controller returns a "device end" signal. This causes the Multiplexer Channel to initiate a DRT fetch, thus advancing the I/O program to the next IOCW.

## INTERRUPTS

Each device controller has its own device number and is able to generate an interrupt on being given an Interrupt command by the I/O Processor. The interrupt logic for a device controller is shown in figure 8-12.

As explained earlier in this manual, each device number can be assigned to an interrupt mask group. If the mask bit for that group is not set, no interrupt from that device can occur. Note in figure 8-12 that setting the Mask flip-flop will allow the Interrupt Request flip-flop to set the Interrupt Latch. The conditions that set the Mask flip-flop are: 1) that the I/O Processor has issued an IOCMD of SMASK (Set Mask); 2) that the mask word given on the IOD lines includes a true bit corresponding to the single bit that is wired to the Mask flip-flop input. Several device controller cards may have their Mask flip-flop wired to the same IOD line; thus these cards form one interrupt mask group.

An interrupt is initiated either by a CPU instruction (SIN, Set Interrupt), for any device number, or by an I/O program order (device controllers only). A SIN instruction causes the I/O Processor to issue an IOCMD of SIL (Set Interrupt Level) with the appropriate DEVNO, which sets the Interrupt Request flip-flop. An Interrupt order causes the Multiplexer Channel to issue "set interrupt" command to the device controller via the multiplexer channel bus; the controller logic then directly forces the Interrupt Request flip-flop to set. From either cause, setting the Interrupt Request flip-flop will result in an INTREQ signal to the I/O Processor and (only if the Mask flip-flop is set) the setting of the Interrupt Latch.

When the I/O Processor receives INTREQ and is ready to process the request, it returns INTPOLL to determine the highest priority request. The first set latch encountered by the poll stops further propagation of the poll, and is then permitted to set the Interrupt Active (IACTIVE) flip-flop. This causes the interrupt device number to be sent to the I/O Processor via the DEVNO lines. An Interrupt Acknowledge signal (INTACK) is also sent, telling the I/O Processor to load DEVNO into its Interrupt DEVNO register.

When the I/O Processor has the device number, it issues an I/O Int signal to the CPU, so that interrupt processing may begin when the CPU is ready.

## SELECTOR CHANNEL

As explained earlier under the heading of Transfer Modes, a Selector Channel operates only one I/O program, and transfers blocks of data, for only one device at a time.

However, since there may be several Selector Channels operating in the system, it would be advantageous to first study the Port Controller, to see how each channel gains access to the central data bus. Then, following the Port Controller discussion, the complete operating sequences for a Selector Channel will be given.

### PORT CONTROLLER

The Port Controller provides four ports to the central data bus for I/O program and data transfers between Selector Channels and memory. Figure 8-14 is a simplified logic diagram of the Port Controller. Note that only one-fourth of the logic is shown; the logic for the three remaining ports is identical to the one shown. The signal and data lines on the left of the diagram represent a portion of the port controller bus. (The port controller bus contains four sets of signal lines—one set for each channel—and one set of data lines which is shared by all four channels.) The bold line on the right is the central data bus. Connection points for the other three sets of logic are marked by X's.

The Port Controller is assigned a module number, like other system modules, by jumper wiring of the module number and Enable (ENB) inputs and outputs. The module number (2, 3, 4, 5, or 6) gives the Port Controller a specific transmission priority among the other system modules.

A Selector Channel requiring transfer of a word to or from memory presents the Port Controller with a request for a Clear/Write or a Read/Restore operation, respectively, along with the memory module number (0, 1, 2, or 3) to which the address will be sent.

A Clear/Write operation consists of a Low Request (LREQ) for an address transfer followed by a Low Select (LSEL) of that address from the channel to memory, via the central data bus; then a High Request (HREQ) for a data transfer followed by a High Select (HSEL) of that data to memory, via the central data bus. A Read/Restore operation consists of a LREQ for an address transfer followed by a LSEL of the address to the bus and memory; then a "wait" for a return transfer of data to the Port Controller from the module to which the address was sent. This return transfer of data is indicated to the Selector Channel by the STRB (Strobe) signal.

Priority is resolved among the four ports in the Port Controller on the following basis: Low Requests, with the desired destination module ready, are granted first to channel 1, next to channel 2, next to channel 3, and last to channel 4. A High Request for any channel takes precedence over all Low Requests. Since a High Request is set immediately when a Low Request for a CW operation is granted, there can be at most one High Request pending at a time.

To maximize the transfer rate, each port can accept a request from its channel before a previously requested transfer has completed; i.e., after LSEL but before or during HSEL or STRB. This second request cannot be granted by a LSEL, however, until the first transfer is complete.

The Clear/Write sequence is as follows. A CWREQ on the request lines to the port sets the LREQ flip-flop and sets the MOP flip-flop to the CW state. The TO lines from the channel are clocked into TO Register A, and the content is then compared with the Ready line (RDY) for that module. When the destination is ready, and ENB is present, and the port has priority, the LSEL and HREQ flip-flops are set. LSEL clocks the content of TO Register A into TO Register B, and gates the address from the channel to the central data bus along with TO (= TO Reg A), FROM (= wired module number), and MOP (= CW). LSEL also pulls low on the destination's RDY line. Then, when ENB is present, the HSEL flip-flop is set. HSEL gates data from the channel to the central data bus, along with TO (= TO Reg B), FROM (= wired module number), and MOP (= NOP).

The Read/Restore sequence is as follows. A RRREQ on the request lines to the port sets the LREQ flip-flop and sets the MOP flip-flop to the RR state. The TO lines from the channel are clocked into TO Register A, and the content is then compared with the RDY line for that module. When the destination is ready, and ENB is present, and the port has priority, the LSEL flip-flop is set. LSEL clocks the content of TO Register A into TO Register B, and gates the address from the channel to the central data bus along with TO (= TO Reg A), FROM (= wired module number), and

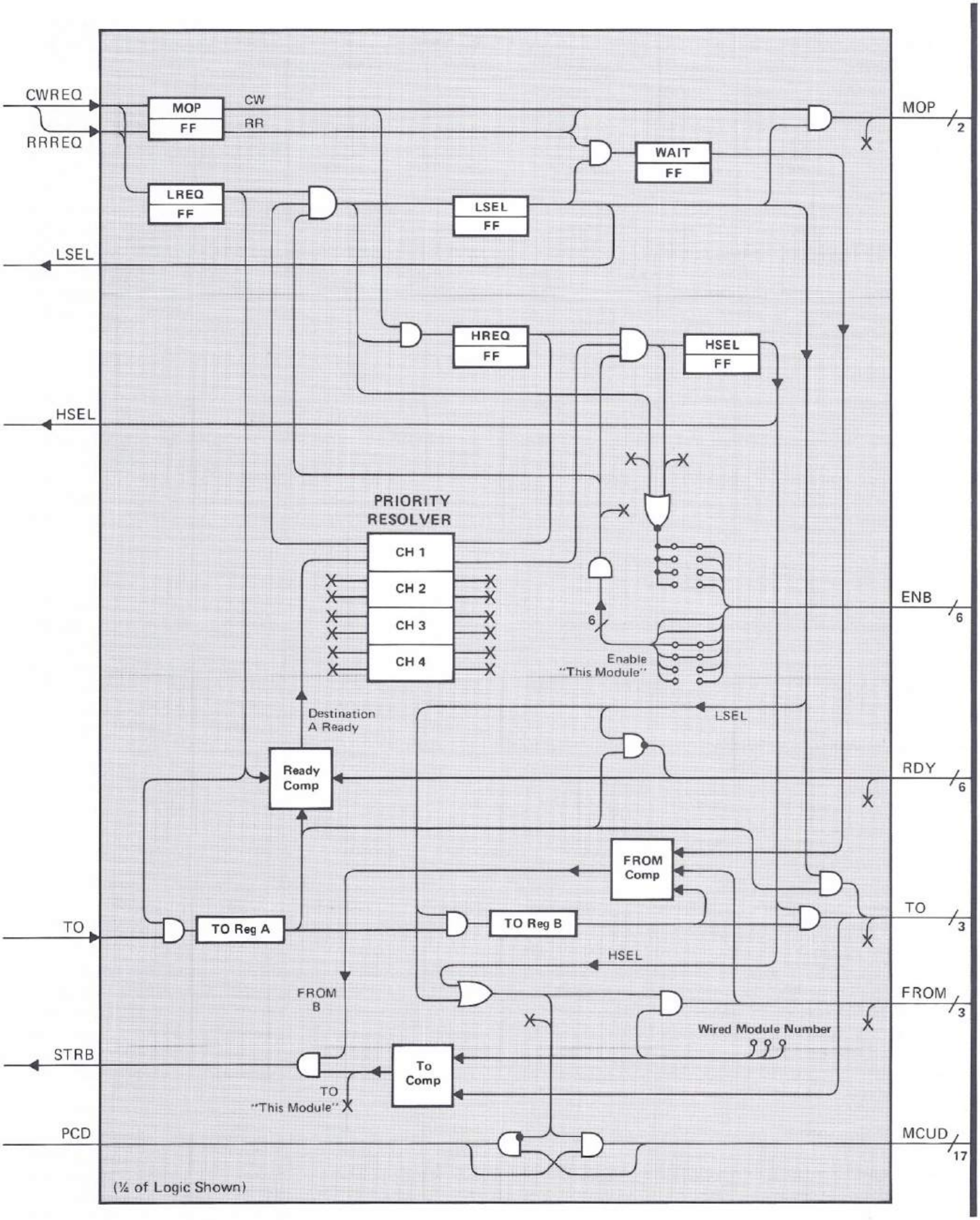


Figure 8-14. Port Controller

MOP (= RR). LSEL also sets the Wait flip-flop. Then, when returning data is present on the bus, the TO (= This Module) and FROM (= TO Reg B) comparisons match, and a STRB signal is sent to the channel. This tells the channel to accept the data on the port controller data (PCD) lines.

## INITIATOR SEQUENCE

The following procedures describe how the Selector Channel's program counter is initialized, as the first step in executing an I/O program for one device.

Refer to figures 8-15 and 8-16, which show simplified logic diagrams of, respectively, a high-speed device controller and a Selector Channel. The selector channel bus, which is similar to the multiplexer channel bus in purpose, is shown originating at the Selector Channel. It is routed to all controllers on this channel, although only one is shown. The selector channel bus differs from the multiplexer channel bus in that it uses 16 lines for transfer of control, status, and data words between device controller and channel; the corresponding lines on the multiplexer channel bus are used as service request lines for up to 16 devices.

The IOP bus (not shown) connects to the device controller as indicated at the top of figure 8-15. Except for the SI signal, the Selector Channel has no connections to the IOP bus. Physically, the SI line is routed to the Selector Channels by way of the power bus (see figure 8-2).

The initiator sequence begins when the CPU encounters an SIO instruction. The CPU, under control of its SIO microprogram, outputs a command word to the IOP Control Register. (See figure 8-10.) This initial command is a TIO (Test I/O), the purpose of which is to see if there is already an I/O program active on the channel. The I/O Processor issues the TIO with SO and DEVNO on the IOP bus. The device controller compares DEVNO with its internal wired device number and a true comparison, with SO, causes the controller to return SI to the I/O Processor with a 16-bit status word on the IOP bus. The CPU microprogram obtains this status word from the I/O Processor and checks to see that bit 0, the "SIO OK" bit, is true. This bit will be true if the device is ready and the channel is inactive. Assuming that the SIO OK bit is true, the CPU microprogram outputs an SIO command to the IOP Control Register, and the I/O Processor issues the SIO command to the controller.

Again, the DEVNO on the bus is compared with the internal wired device number (see figure 8-15), and the true result, with SO, enables the IOCMD (I/O Command) to be decoded. The IOCMD is now SIO which, when decoded, issues a Request (REQ) signal to the channel control logic. The channel then returns SI (Service In) to the I/O Processor as an acknowledgment response. From now on (except for processing an interrupt), the I/O Processor is not involved. The data gating logic routes all data transmissions to the DATA lines of the selector channel bus, rather than the IOD lines of the IOP bus.

The REQ signal, sent to the Selector Channel via the channel bus, is accompanied by a 16-bit word on the DATA lines. This 16-bit word identifies the requesting device by device number (eight bits).

When the Selector Channel receives REQ from the device controller, it sets the control logic to "active". The device address is then loaded into the DEVNO register. The Selector Channel is now exclusively reserved for that device. Furthermore, only this controller will respond to CHANSO (Channel Service Out) from the Selector Channel.

The Selector Channel now reads out the device number from the DEVNO register, and requests a memory transfer by issuing a Read/Restore Request to the Port Controller. The Port Controller checks if memory is ready and, when ENB is present, sets LSEL. The LSEL signal is returned to the Selector Channel, where it reads the device number (shifted left by two bits to be the DRT entry address) onto the PCD lines. LSEL also reads out the TO, FROM, and MOP codes in the Port Controller, thus effecting an address transmission to memory.

When memory returns the DRT contents, the Port Controller issues STRB to the Selector Channel. Since the channel control logic is expecting a DRT word, it loads the bus data into the I/O Program Counter. The contents of the I/O Program Counter will hereafter be used to address the individual locations of the I/O program, and so no further DRT fetches are necessary. Program execution will occur as a result of "fetch" and "execute" sequences, next described.

## FETCH SEQUENCE

Fetching an I/O program doubleword requires two memory fetches. Unlike the Multiplexer Channel, which examines the IOCW to determine what to do about the IOAW (fetch it, store into it, or gate it out to the device controller), the two memory fetches always occur. The different operations for the various types of I/O orders are accomplished in the execute sequence.

The fetch sequence begins with the Selector Channel reading out the contents of the I/O Program Counter, and requesting a memory transfer (Read/Restore Request to Port Controller). When the Port Controller has obtained transmit priority, it returns LSEL, transmitting the I/O Program Counter contents to memory as an address. (The Counter is immediately incremented.)

When memory returns the IOCW from the addressed location, the Port Controller issues STRB to the Selector Channel. The channel control logic, which is expecting the IOCW, loads the word into the IOCW Active Register. Then the I/O Program Counter is again read out with another memory transfer request. The Port Controller transmits this address to memory, and the I/O Program Counter is again incremented. Then, when memory returns the IOAW from the addressed location, the Selector Channel loads the word into the IOAW Active Register, and at this point the fetch sequence is complete.

The channel control logic can now examine the order. If the order specified in the IOCW is Read or Write, and if data chaining is also specified, a pre-fetch sequence is enabled. This operation is the same as the fetch sequence described in the preceding two paragraphs, except that the returned data is loaded into the IOCW Buffer and IOAW Buffer instead of the IOCW and IOAW Active Registers. An additional condition for the pre-fetch sequence is that data transfers take precedence; i.e., pre-fetch will occur only when both Input Buffers A and B are empty (for Read) or both Output Buffers A and B are full (for Write).

Then, when the Read or Write order finishes, due either to word count rollover or to a "device end" condition (see Read and Write execute sequences), the IOCW/IOAW Buffers are read into the IOCW/IOAW Active Registers. The data transfer can thus continue uninterrupted. If the new IOCW specifies further data chaining, another pre-fetch is initiated to refill the buffers.

## EXECUTE SEQUENCES

A separate description is given below for the execute sequence of each of the eight I/O orders. In each case except End (which terminates the I/O Program), operation returns to the fetch sequence following the completion of the execute sequence, in order to fetch the next I/O program word.

SENSE. The Selector Channel issues a "P STATUS STB" signal to the device controller, with CHANSO, via the channel bus. The device controller accordingly reads the contents of its Status register onto the channel DATA lines and returns CHAN ACK (Channel Acknowledge). On receipt of CHAN ACK, the Selector Channel loads the Status information into one of the two input buffers, and prepares for a memory transfer. First the contents of the I/O Program Counter are decremented by one. This is necessary because the Status word must be stored in the IOAW location for the current order, whereas the fetch sequence has incremented the I/O Program Counter to point at the next word. Once this is done, the contents of the I/O Program Counter and the input buffer containing the status word are read out to the channel PCD gates (but not gated out yet). Also, the I/O Program Counter contents are decoded by the interleaver and mapper to derive a TO code. (See earlier discussions of these circuits under the Arithmetic Logic heading of the CPU discussion.) A Clear/Write Request to the Port Controller requests a transmission to memory, and when the Port Controller returns LSEL, the address from the I/O Program Counter is sent to memory and the Counter is incremented. An HSEL from the Port Controller (which follows immediately unless ENB has been preempted by a higher-priority module) then reads out the Status word to the PCD lines and sends it to memory. This stores Status in the IOAW location.

RETURN RESIDUE. The function of the Return Residue order is to send the current contents of the Residue Register (which reflects the results of the most recent Read or Write order) to the IOAW location of the current I/O program word. The device controller is not involved. To begin the procedure, the channel control logic decrements the I/O Program Counter (for the same reason described in the preceding paragraph). The contents of the I/O Program Counter and the Residue Register are then read out to the PCD gates, while a Clear/Write Request and a mapped TO code are issued to the Port Controller. When the Port Controller returns LSEL, the address from the I/O Program Counter is sent to memory. When HREQ sets the HSEL flip-flop, the word count from the Residue Register is sent to memory. This stores the residue in the IOAW location.

INTERRUPT. The channel control logic issues a "P SET INT" signal to the device controller, with CHANSO, via the selector channel bus. The device controller returns CHAN ACK and sets its Interrupt Request flip-flop. Provided the Mask flip-flop is set, the device controller issues INTREQ to the I/O Processor via the IOP bus. When the I/O Processor returns INTPOLL, the device number is sent to the I/O Processor, along with INTACK. On receipt of INTACK, the I/O Processor generates an interrupt signal to the CPU.

JUMP. The Jump order may be specified to be either conditional or unconditional. It is the function of an unconditional jump or a successful conditional jump to transfer the contents of the IOAW Buffer (the jump address) to the I/O Program Counter. (The IOAW Buffer and IOAW Active Register contain identical contents at this time.) In the case of a conditional Jump order, the Selector Channel issues a "set jump" command to the device controller, with CHANSO, via the channel bus. The device controller returns a true or false "jump met" signal. If the jump is not met, operation returns to the fetch sequence. If the jump is met, and for an unconditional Jump order, the channel control logic gates the contents of the IOAW Active Register into the I/O Program Counter. Thus subsequent orders will be fetched and executed from a new I/O program area.

CONTROL. The Control order routes both the IOCW and the IOAW to the device controller. The Selector Channel

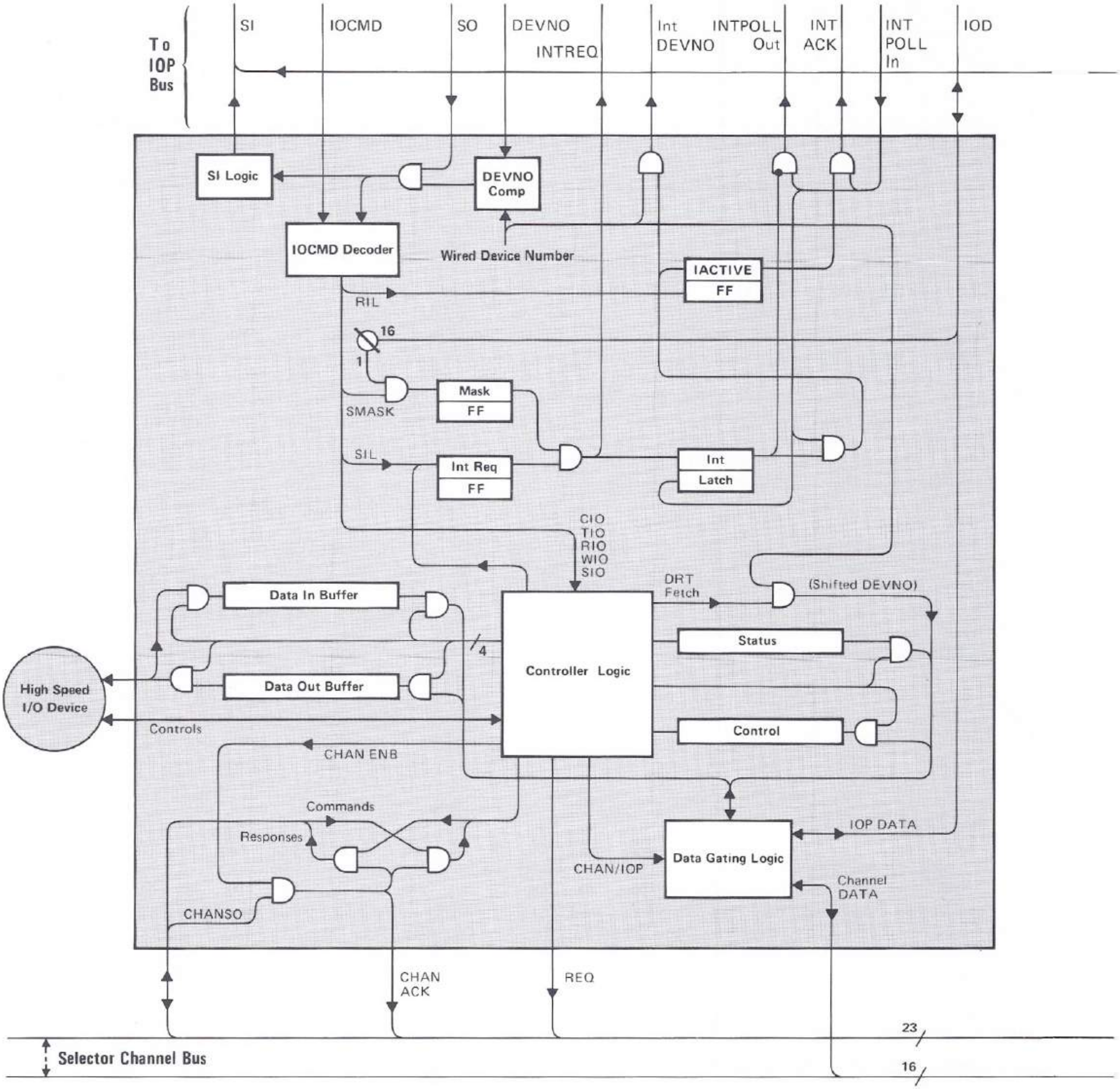


Figure 8-15. High-Speed Device Controller

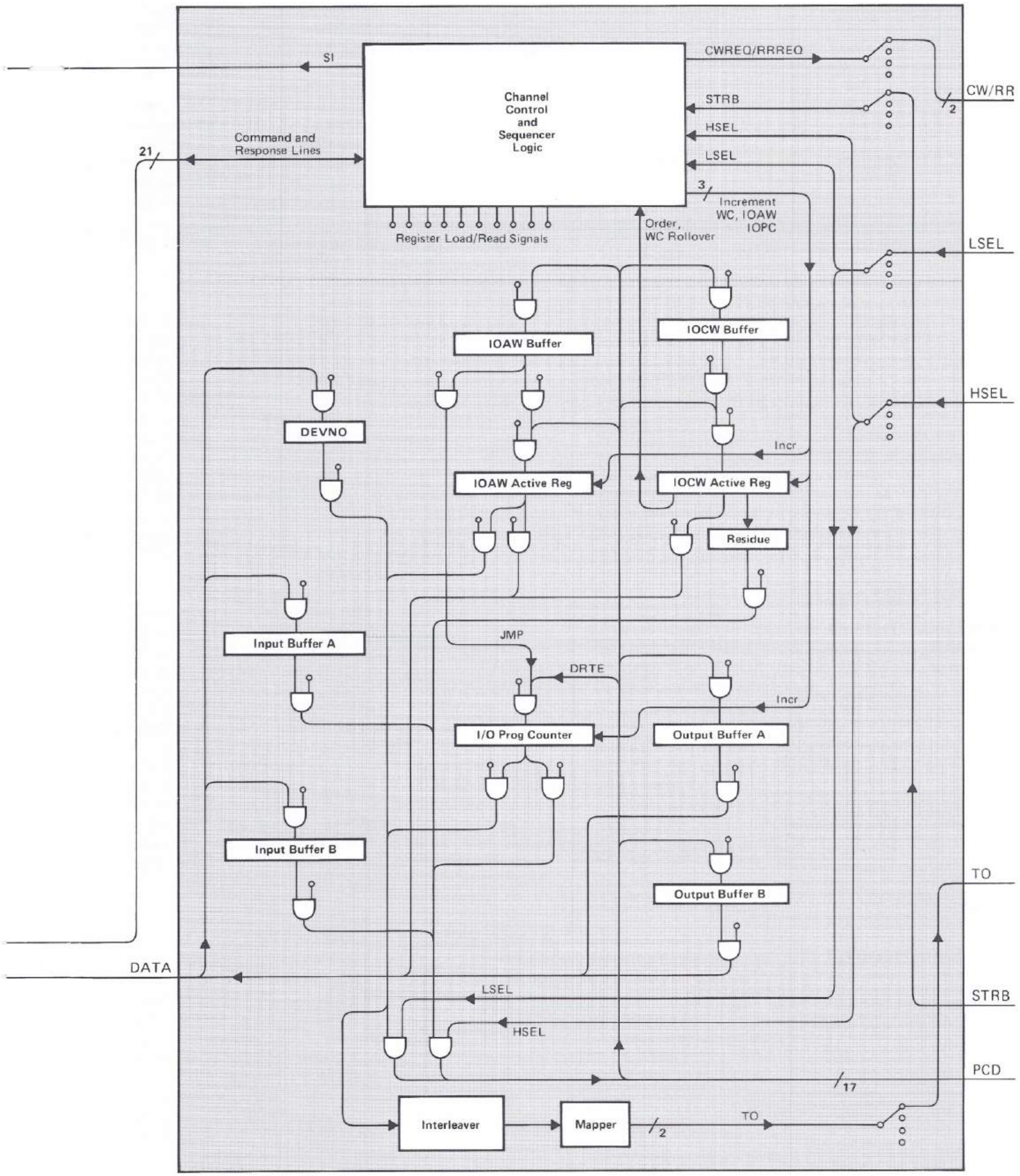


Figure 8-16. Selector Channel

first reads out the contents of the IOCW Active Register to the channel DATA lines and issues a "PCMDI" signal, with CHANSO, for the device controller to load the DATA word. The device controller accordingly loads the word into its Control register, and then issues a request (CHAN SR) back to the Selector Channel to send the second word. The Selector Channel reads out the contents of the IOAW Active Register to the DATA lines and issues a second command ("P CONT STB"), with CHANSO, for the device controller to load this new word. When the device controller has done so, and is ready for the next order, it returns the appropriate response (another CHAN SR) signal to the Selector Channel.

READ. The Read order causes a block of data to be transferred from the device to memory. The block size in words is specified in two's complement form by the word count (IOCW bits 4 through 15) and the absolute starting address in memory is specified by the IOAW. While the block transfer is in progress, there are two separate, simultaneous operations taking place: the device-to-channel transfer and the channel-to-memory transfer. The following two paragraphs separately describe these two operations. To begin the Read execute sequence, the Selector Channel issues CHANSO to the controller. When the controller returns CHAN ACK, the Selector Channel issues the initial RD NXT WD ("Read Next Word") with CHANSO still asserted. When CHANSO is removed, both the Selector Channel and the controller are set to the "in-transfer" condition to enable data transfers.

After the device has read a word and the controller is ready to transfer it to the channel, it sends CHAN SR (Channel Service Request) to the channel. The channel issues "P READ STB" and CHANSO, causing the device controller to read its Data In Buffer onto the channel DATA lines and to return CHAN ACK. On receiving CHAN ACK, the Selector Channel loads the data into either Input Buffer A or Input Buffer B (depending on which is empty), increments the word count in the IOCW Active Register, and re-issues RD NXT WD. The above transfer sequence repeats for each data word until the device controller asserts DEV END to terminate the block, or until the word count rolls over. In either case, the channel sends EOT ("End of Transfer") to the controller and, if not data chaining, clears the "in-transfer" condition. A CHAN SR from the controller is required to resume program execution.

Meanwhile, the Selector Channel attempts to keep both Input Buffers empty by transmitting their contents to memory. The control logic for the A and B buffers ensures that data is transmitted to memory in the same sequence as received from the device. To accomplish a memory transfer, the Selector Channel enables the IOAW Active Register for use as a memory address, enables Input Buffer A or B for use as a data word, and sends a Clear/Write Request and a mapped TO code to the Port Controller. When the port returns LSEL, the IOAW is gated onto the bus as an address to memory, and the IOAW is incremented to point to the next data location. When the port returns HSEL, the Input Buffer is gated onto the bus to be stored in the addressed memory location. The preceding operation (this paragraph) repeats until the Read order completes, via a DEV END or word count rollover, and all input data has been sent to memory.

If the data chaining bit in the IOCW Active Register is true, the next order pair will have been prefetched when possible during the block data transfer. When the Read order completes, the prefetched order pair will be transferred from the IOCW/IOAW buffers to the active registers without the need for a normal fetch sequence. Data input can thus continue for the next block with minimum interruption. If the data chaining bit is not set, the Read termination will be followed by a normal fetch sequence.

WRITE. The Write order causes a block of data to be transferred from memory to the device. The block size in words is specified in two's complement form by the word count (IOCW bits 4 through 15) and the absolute starting address of the block in memory is specified by the IOAW. While the block transfer is in progress, there are two separate, simultaneous operations taking place: the memory-to-channel transfer and the channel-to-device transfer. The following two paragraphs separately describe these two operations. To begin the Write execute sequence, the Selector Channel issues CHANSO to the controller, and when the controller returns CHAN ACK, both the Selector Channel and the controller are set to the "out-transfer" condition to enable data transfers.

Meanwhile, the Selector Channel proceeds with a memory fetch and will attempt to keep both output buffers full. The control logic for the A and B Output Buffers ensures that data is transmitted to the device in the same sequence as it was fetched from memory. To accomplish a memory fetch, the Selector Channel enables the IOAW Active Register for use as a memory address and sends a Read/Restore Request (RRREQ) and a mapped TO code to the Port Controller. When the port returns LSEL, the IOAW is gated onto the bus as an address to memory, and the IOAW is incremented to point to the next data location. When the port returns STRB, the data on the bus from memory is loaded into an empty output buffer. The preceding operation (this paragraph) repeats until the Write order completes (by either a DEV END or word count rollover).

When the controller is ready to accept a data word from the channel, it sends CHAN SR. The channel issues CHANSO and "P WRITE STB" and gates Output Buffer A or B onto the channel DATA lines. The controller returns CHAN ACK, causing the channel to remove P WRITE STB, increment the word count, and remove CHANSO in that order. The device controller uses the removal of P WRITE STB to latch the data word from the channel DATA lines. The above transfer sequence (this paragraph) repeats for each data word sent to the device controller, until the device controller asserts DEV END to prematurely terminate the block or until the word count rolls over. In either case, the Selector Channel sends EOT ("End of Transfer") to the controller and, if not data chaining, clears the out-transfer condition. To resume program execution, a new CHAN SR from the controller is required by the Selector Channel.

If the data chaining bit (IOCW bit 0) is true, the next order pair will have been prefetched when possible during the block transfer. When the Write order completes, the prefetched order pair will be transferred from the IOCW/IOAW buffers to the active registers without the need for a normal fetch sequence. Data output to the controller can thus continue for the next block with minimum interruption. If the data chaining bit is not set, termination of the Write order will be followed by a normal fetch sequence.

END. The execute sequence for the End order begins by duplicating the operations of a Sense order, obtaining the controller's status word and storing it in the IOAW location in the I/O program. Additionally, if IOCW bit 4 is true, a "P SET INT" signal is also issued to the controller; see Interrupt order description. Then the channel proceeds to store the contents of its I/O Program Counter into the device's DRT location. As explained earlier, this is to maintain compatibility with I/O programs run via a Multiplexer Channel. The Selector Channel enables its DEVNO register, shifted left two bits, as a memory address, enables the I/O Program Counter for use as data, and sends a Clear/Write Request (CWREQ) and a mapped TO code to the Port Controller. When the port returns LSEL, the shifted device number is gated out as the DRT address, and when the port returns HSEL, the I/O Program Counter content is gated out to the bus as data. This completes all operations for the I/O program. The channel control logic resets to the inactive condition, thus allowing another program for the same or another device to be initiated via that channel.

## DIRECT I/O OPERATION

In addition to the SIO modes of transfer, described under the Multiplexer Channel and Selector Channel headings of this section, a direct I/O mode is also provided. In this mode, the CPU may transfer information directly to or from a device controller, without involving memory, Multiplexer Channel, or Selector Channel.

The CPU has four instructions for direct I/O communication. These are: TIO (Test I/O), CIO (Control I/O), RIO (Read I/O), and WIO (Write I/O). In each case, one word is transferred for each instruction, either to or from the top of the stack in the CPU. The following paragraphs describe the operation for each of these four instructions. Figures 8-10 and 8-12, the I/O Processor and a device controller, may be used as references.

TIO. The Test I/O instruction obtains the contents of the device controller's Status register and loads it into the CPU's current top-of-stack register (RA). When the CPU encounters a TIO instruction, its TIO microprogram loads a command word into the IOP Control Register in the I/O Processor. The I/O Processor then issues a TIO IOCMD to the device addressed by the DEVNO code, along with SO. The addressed device is therefore enabled to accept and decode the command, and accordingly reads the contents of its Status register onto the IOD lines, with SI. On receipt of SI, the I/O Processor loads the Status word into the Direct Data Input register and informs the CPU that the word is present (by means of a flag signal not shown in figure 8-10). The CPU then issues a read signal which reads the contents of the Direct Data Input register to the S-bus. From there, the status word is routed to the current RA register.

RIO. The operations for the Read I/O instruction begin by performing a TIO to the controller (as above) to check the Read/Write OK status bit. If status is acceptable, the same sequence is repeated except that the OUTCMD is RIO and data is transferred from the Data In Buffer rather than the Status register.

CIO. The Control I/O instruction obtains a control word from the top-of-stack register (RA) and sends it to the device controller's Control register. When the CPU encounters a CIO instruction, its CIO microprogram loads the RA contents into the Data Output Register, and then issues a command word to the IOP Control Register in the I/O Processor. The command word causes a CIO IOCMD to be issued to the device controller addressed by the DEVNO code, along with SO. At the same time, the contents of the Data Output Register are read out onto the IOD lines. When the device controller decodes the IOCMD it loads the word on the IOD lines into its Control register, and returns SI to the I/O Processor. On receiving SI, the I/O Processor returns a flag signal to the CPU, indicating completion of the instruction.

WIO. The operations for the Write I/O instruction begin by performing a TIO to the controller (as above) to check the Read/Write OK status bit. If status is acceptable, the remaining operations for the Write I/O instruction are the same as for CIO, except that the information sent is a data word, the IOCMD is WIO instead of CIO, and the information is loaded into the Data Out Buffer instead of the Control register.

Table 8-1. Mnemonics and Abbreviations

| | |
|---|---|
| Adrs | Address |
| ALU | Arithmetic Logic Unit |
| BUSH | Bus High |
| BUSL | Bus Low |
| CHAN ACK | Channel Acknowledge |
| CHAN DATA | Channel Data |
| CHANSO | Channel Service Out |
| CHAN SR | Channel Service Request |
| CIO | Control Input/Output |
| Comp | Comparator |
| Cont | Controller |
| CPU | Central Processor Unit |
| CRL | Control |
| CTO | Command TO |
| CW | Clear/Write |
| CWREQ | Clear/Write Request |
| DC | Device Controller/Data Chain |
| DEV END | Device End |
| DEVNO | Device Number |
| DATAPOLL | Data Poll |
| DRT | Device Reference Table |
| DRTE | Device Reference Table Entry |
| ENB | Enable |
| EOT | End-of-Transfer |
| HI SEL | High Select |
| HREQ | High Request |
| HSEL | High Select |
| HSREQ | High Service Request |
| INTACK | Interrupt Acknowledge |
| IACTIVE | Interrupt Active |
| Incr | Increment |
| Int DEVNO | Interrupt Device Number |
| IOAW | I/O Address Word |
| IOCMD | I/O Command |
| IOCW | I/O Control Word |
| IOD | I/O Data |
| IOHRQ | I/O High Request |
| IOINP | I/O In Process |
| IOLRQ | I/O Low Request |
| IOMOP | I/O Memory Opcode |
| IOP | Input/Output Processor |
| IOSTRB | I/O Strobe |
| INTPOLL | Interrupt Poll |
| INTREQ | Interrupt Request |
| JMP | Jump |
| LO SEL | Low Select |
| LREQ | Low Request |
| LSEL | Low Select |
| LUT | Look-Up Table |
| MAR | Memory Address Register |
| MCU | Module Control Unit |
| Mem | Memory |
| MOP | Memory Opcode |
| NIP | Next In Process |
| NOP | No Operation |
| OPINP | Operand In Process |
| OPND | Operand |
| P CONT STB | Programmed Control Strobe |
| P READ STB | Programmed Read Strobe |
| P SET INT | Programmed Set Interrupt |
| P STATUS STB | Programmed Status Strobe |
| P WRITE STB | Programmed Write Strobe |
| PCD | Port Controller Data |
| Prog | Program |
| RAM | Random Access Memory |
| RAR | ROM Address Register |
| RDY | Ready |
| Reg | Register |
| REQ | Request |
| RIL | Reset Interrupt Level |
| RIO | Read Input/Output |
| RNW | Read/No Write |
| ROM | Read-Only Memory |
| ROR | ROM Output Register |
| RR | Read/Restore |
| RRREQ | Read/Restore Request |
| SEL | Select |
| SI | Service In |
| SIL | Set Interrupt Level |
| SIN | Set Interrupt |
| SIO | Start Input/Output |
| SMASK | Set Mask |
| SO | Service Out |
| SR | Service Request |
| STRB | Strobe |
| TIO | Test Input/Output |
| WC | Word Count |
| WIO | Write Input/Output |

# APPENDIX

# INDEX OF TERMS

# INDEX OF TERMS

# OCTAL ARITHMETIC

## ADDITION

### TABLE

| 0 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|
| 1 | 02 | 03 | 04 | 05 | 06 | 07 | 10 |
| 2 | 03 | 04 | 05 | 06 | 07 | 10 | 11 |
| 3 | 04 | 05 | 06 | 07 | 10 | 11 | 12 |
| 4 | 05 | 06 | 07 | 10 | 11 | 12 | 13 |
| 5 | 06 | 07 | 10 | 11 | 12 | 13 | 14 |
| 6 | 07 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

### EXAMPLE

```
Add:   3677   octal
     + 1331   octal
      (111-)  carries
       5230   octal
```

## MULTIPLICATION

### TABLE

| 1 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|
| 2 | 04 | 06 | 10 | 12 | 14 | 16 |
| 3 | 06 | 11 | 14 | 17 | 22 | 25 |
| 4 | 10 | 14 | 20 | 24 | 30 | 34 |
| 5 | 12 | 17 | 24 | 31 | 36 | 43 |
| 6 | 14 | 44 | 30 | 36 | 44 | 52 |
| 7 | 16 | 35 | 34 | 43 | 52 | 61 |

### EXAMPLE

```
Multiply:   657   octal
          ×  54   octal
            3274
           4153
           45024  octal
```

(Reminder: add in octal)

## COMPLEMENT

To find the two's complement form of an octal number. (Same procedure whether converting from positive to negative or negative to positive.)

### RULE

1. Subtract from the maximum representable octal value.
2. Add one.

### EXAMPLE

Two's complement of $556_8$ :

$$\begin{array}{r} 177777 \\ -\ 000556 \\ \hline 177221 \\ +\ 1 \\ \hline 177222_8 \end{array}$$

# OCTAL/DECIMAL CONVERSIONS

## OCTAL TO DECIMAL

TABLE

| OCTAL | DECIMAL |
|---|---|
| 0- 7 | 0- 7 |
| 10-17 | 8-15 |
| 20-27 | 16-23 |
| 30-37 | 24-31 |
| 40-47 | 32-39 |
| 50-57 | 40-47 |
| 60-67 | 48-55 |
| 70-77 | 56-63 |
| 100 | 64 |
| 200 | 128 |
| 400 | 256 |
| 1000 | 512 |
| 2000 | 1024 |
| 4000 | 2048 |
| 10000 | 4096 |
| 20000 | 8192 |
| 40000 | 16384 |
| 77777 | 32767 |

EXAMPLE

Convert $463_8$ to a decimal integer.

$$\begin{aligned} 400_8 &= 256_{10} \\ 60_8 &= 48_{10} \\ 3_8 &= \underline{3_{10}} \\ & \quad 307 \text{ decimal} \end{aligned}$$

## DECIMAL TO OCTAL

TABLE

| DECIMAL | OCTAL |
|---|---|
| 1 | 1 |
| 10 | 12 |
| 20 | 24 |
| 40 | 50 |
| 100 | 144 |
| 200 | 310 |
| 500 | 764 |
| 1000 | 1750 |
| 2000 | 3720 |
| 5000 | 11610 |
| 10000 | 23420 |
| 20000 | 47040 |
| 32767 | 77777 |

EXAMPLE

Convert $5229_{10}$ to an octal integer.

$$\begin{aligned} 5000_{10} &= 11610_8 \\ 200_{10} &= 310_8 \\ 20_{10} &= 24_8 \\ 9_{10} &= \underline{11_8} \\ & \quad 12155_8 \end{aligned}$$

$\uparrow$

(Reminder: add in octal)

## NEGATIVE DECIMAL TO TWO'S COMPLEMENT OCTAL

TABLE

| DECIMAL | 2's COMP |
|---|---|
| -1 | 177777 |
| -10 | 177766 |
| -20 | 177754 |
| -40 | 177730 |
| -100 | 77634 |
| -200 | 177470 |
| -500 | 177014 |
| -1000 | 176030 |
| -2000 | 174040 |
| -5000 | 166170 |
| -10000 | 154360 |
| -20000 | 130740 |
| -32768 | 100000 |

EXAMPLE

Convert $-629_{10}$ to two's complement octal.

$$\begin{aligned} -500_{10} &= 177014_8 \\ -100_{10} &= 177634_8 \\ -20_{10} &= 177754_8 \quad \text{(Add in octal)} \\ -9_{10} &= \underline{177767_8} \\ & \quad 176613_8 \end{aligned}$$

For reverse conversion (two's complement octal to negative decimal):

1. Complement, using procedure on facing page.
2. Convert to decimal, using OCTAL TO DECIMAL table.

# MATHEMATICAL EQUIVALENTS

## $2^{\pm n}$ IN DECIMAL

| $2^n$ | $n$ | $2^{-n}$ |
|---|---|---|
| 1 | 0 | 1.0 |
| 2 | 1 | 0.5 |
| 4 | 2 | 0.25 |
| 8 | 3 | 0.125 |
| 16 | 4 | 0.0625 |
| 32 | 5 | 0.03125 |
| 64 | 6 | 0.01562 5 |
| 128 | 7 | 0.00781 25 |
| 256 | 8 | 0.00390 625 |
| 512 | 9 | 0.00195 3125 |
| 1 024 | 10 | 0.00097 65625 |
| 2 048 | 11 | 0.00048 82812 5 |
| 4 096 | 12 | 0.00024 41406 25 |
| 8 192 | 13 | 0.00012 20703 125 |
| 16 384 | 14 | 0.00006 10351 5625 |
| 32 768 | 15 | 0.00003 05175 78125 |
| 65 536 | 16 | 0.00001 52587 89062 5 |
| 131 072 | 17 | 0.00000 76293 94531 25 |
| 262 144 | 18 | 0.00000 38146 97265 625 |
| 524 288 | 19 | 0.00000 19073 48632 8125 |
| 1 048 576 | 20 | 0.00000 09536 74316 40625 |
| 2 097 152 | 21 | 0.00000 04768 37158 20312 5 |
| 4 194 304 | 22 | 0.00000 02384 18579 10156 25 |
| 8 388 608 | 23 | 0.00000 01192 09289 55078 125 |
| 16 777 216 | 24 | 0.00000 00596 04644 77539 0625 |
| 33 554 432 | 25 | 0.00000 00298 02322 38769 53125 |
| 67 108 864 | 26 | 0.00000 00149 01161 19384 76562 5 |
| 134 217 728 | 27 | 0.00000 00074 50580 59692 38281 25 |
| 268 435 456 | 28 | 0.00000 00037 25290 29846 19140 625 |
| 536 870 912 | 29 | 0.00000 00018 62645 14923 09570 3125 |
| 1 073 741 824 | 30 | 0.00000 00009 31322 57461 54785 15625 |
| 2 147 483 648 | 31 | 0.00000 00004 65661 28730 77392 57812 5 |
| 4 294 967 296 | 32 | 0.00000 00002 32830 64365 38696 28906 25 |

## $10^{\pm n}$ IN OCTAL

| $10^n$ | $n$ | $10^{-n}$ |
|---|---|---|
| 1 | 0 | 1.000 000 000 000 000 000 00 |
| 12 | 1 | 0.063 146 314 631 463 146 31 |
| 144 | 2 | 0.005 075 341 217 270 243 66 |
| 1 750 | 3 | 0.000 406 111 564 570 651 77 |
| 23 420 | 4 | 0.000 032 155 613 530 704 15 |
| 303 240 | 5 | 0.000 002 476 132 610 706 64 |
| 3 641 100 | 6 | 0.000 000 206 157 364 055 37 |
| 46 113 200 | 7 | 0.000 000 015 327 745 152 75 |
| 575 360 400 | 8 | 0.000 000 001 257 143 561 06 |
| 7 346 545 000 | 9 | 0.000 000 000 104 560 276 41 |
| 112 402 762 000 | 10 | 0.000 000 000 006 676 337 66 |
| 1 351 035 564 000 | 11 | 0.000 000 000 000 537 657 77 |
| 16 432 451 210 000 | 12 | 0.000 000 000 000 043 136 32 |
| 221 411 634 520 000 | 13 | 0.000 000 000 000 003 411 35 |
| 2 657 142 036 440 000 | 14 | 0.000 000 000 000 000 264 11 |
| 34 327 724 461 500 000 | 15 | 0.000 000 000 000 000 022 01 |
| 434 157 115 760 200 000 | 16 | 0.000 000 000 000 000 001 63 |
| 5 432 127 413 542 400 000 | 17 | 0.000 000 000 000 000 000 14 |
| 67 405 553 164 731 000 000 | 18 | 0.000 000 000 000 000 000 01 |

# MATHEMATICAL EQUIVALENTS

## $2^x$ IN DECIMAL

| $x$ | $2^x$ | $x$ | $2^x$ | $x$ | $2^x$ |
|---|---|---|---|---|---|
| 0.001 | 1.00069 33874 62581 | 0.01 | 1.00695 55500 56719 | 0.1 | 1.07177 34625 36293 |
| 0.002 | 1.00138 72557 11335 | 0.02 | 1.01395 94797 90029 | 0.2 | 1.14869 83549 97035 |
| 0.003 | 1.00208 16050 79633 | 0.03 | 1.02101 21257 07193 | 0.3 | 1.23114 44133 44916 |
| 0.004 | 1.00277 64359 01078 | 0.04 | 1.02811 38266 56067 | 0.4 | 1.31950 79107 72894 |
| 0.005 | 1.00347 17485 09503 | 0.05 | 1.03526 49238 41377 | 0.5 | 1.41421 35623 73095 |
| 0.006 | 1.00416 75432 38973 | 0.06 | 1.04246 57608 41121 | 0.6 | 1.51571 65665 10398 |
| 0.007 | 1.00486 38204 23785 | 0.07 | 1.04971 66836 23067 | 0.7 | 1.62450 47927 12471 |
| 0.008 | 1.00556 05803 98468 | 0.08 | 1.05701 80405 61380 | 0.8 | 1.74110 11265 92248 |
| 0.009 | 1.00625 78234 97782 | 0.09 | 1.06437 01824 53360 | 0.9 | 1.86606 59830 73615 |

## $n \log_{10} 2$, $n \log_2 10$ IN DECIMAL

| $n$ | $n \log_{10} 2$ | $n \log_2 10$ | $n$ | $n \log_{10} 2$ | $n \log_2 10$ |
|---|---|---|---|---|---|
| 1 | 0.30102 99957 | 3.32192 80949 | 6 | 1.80617 99740 | 19.93156 85693 |
| 2 | 0.60205 99913 | 6.64385 61898 | 7 | 2.10720 99696 | 23.25349 66642 |
| 3 | 0.90308 99870 | 9.96578 42847 | 8 | 2.40823 99653 | 26.57542 47591 |
| 4 | 1.20411 99827 | 13.28771 23795 | 9 | 2.70926 99610 | 29.89735 28540 |
| 5 | 1.50514 99783 | 16.60964 04744 | 10 | 3.01029 99566 | 33.21928 09489 |

## MATHEMATICAL CONSTANTS IN OCTAL SCALE

| | | |
|---|---|---|
| $\pi = (3.11037\ 552421)_{(8)}$ | $e = (2.55760\ 521305)_{(8)}$ | $\gamma = (0.44742\ 147707)_{(8)}$ |
| $\pi^{-1} = (0.24276\ 301556)_{(8)}$ | $e^{-1} = (0.27426\ 530661)_{(8)}$ | $\ln \gamma = -(0.43127\ 233602)_{(8)}$ |
| $\sqrt{\pi} = (1.61337\ 611067)_{(8)}$ | $\sqrt{e} = (1.51411\ 230704)_{(8)}$ | $\log_2 \gamma = -(0.62573\ 030645)_{(8)}$ |
| $\ln \pi = (1.11206\ 404435)_{(8)}$ | $\log_{10} e = (0.33626\ 754251)_{(8)}$ | $\sqrt{2} = (1.32404\ 746320)_{(8)}$ |
| $\log_2 \pi = (1.51544\ 163223)_{(8)}$ | $\log_2 e = (1.34252\ 166245)_{(8)}$ | $\ln 2 = (0.54271\ 027760)_{(8)}$ |
| $\sqrt{10} = (3.12305\ 407267)_{(8)}$ | $\log_2 10 = (3.24464\ 741136)_{(8)}$ | $\ln 10 = (2.23273\ 067355)_{(8)}$ |

# CONSOLIDATED CODING TABLE

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | STACK OPCODES ALL 64 STACK OPS MAY BE USED IN EITHER POSITION (STACK OP A OR B) | | | | 00 | | | NOP | | | 40 | | | DEL | |
| | | | | | 01 | | | DELB | | | 41 | | | ZROB | |
| | | | | | 02 | | | DDEL | | | 42 | | | LDXB | |
| | | | | | 03 | | | ZROX | | | 43 | | | STAX | |
| | | | | | 04 | | | INCX | | | 44 | | | LDXA | |
| | | | | | 05 | | | DECX | | | 45 | | | DUP | |
| | | | | | 06 | | | ZERO | | | 46 | | | DDUP | |
| | | | | | 07 | | | DZRO | | | 47 | | | FLT | |
| | | | | | 10 | | | DCMP | | | 50 | | | FCMP | |
| | | | | | 11 | | | DADD | | | 51 | | | FADD | |
| | | | | | 12 | | | DSUB | | | 52 | | | FSUB | |
| | | | | | 13 | | | MPYL | | | 53 | | | FMPY | |
| | | | | | 14 | | | DIVL | | | 54 | | | FDIV | |
| | | | | | 15 | | | DNEG | | | 55 | | | FNEG | |
| | | | | | 16 | | | DXCH | | | 56 | | | CAB | |
| | | | | | 17 | | | CMP | | | 57 | | | LCMP | |
| | | | | | 20 | | | ADD | | | 60 | | | LADD | |
| | | | | | 21 | | | SUB | | | 61 | | | LSUB | |
| | | | | | 22 | | | MPY | | | 62 | | | LMPY | |
| | | | | | 23 | | | DIV | | | 63 | | | LDIV | |
| | | | | | 24 | | | NEG | | | 64 | | | NOT | |
| | | | | | 25 | | | TEST | | | 65 | | | OR | |
| | | | | | 26 | | | STBX | | | 66 | | | XOR | |
| | | | | | 27 | | | DTST | | | 67 | | | AND | |
| | | | | | 30 | | | DFLT | | | 70 | | | FIXR | |
| | | | | | 31 | | | BTST | | | 71 | | | FIXT | |
| | | | | | 32 | | | XCH | | | 72 | | | SPARE | |
| | | | | | 33 | | | INCA | | | 73 | | | INCB | |
| | | | | | 34 | | | DECA | | | 74 | | | DECB | |
| | | | | | 35 | | | XAX | | | 75 | | | XBX | |
| | | | | | 36 | | | ADAX | | | 76 | | | ADBX | |
| | | | | | 37 | | | ADXA | | | 77 | | | ADXB | |
| 01 | SUB OPCODE 1 | | | X | 00 | | | ASL | | | SHIFT COUNT L | | | | |
| | | | | X | 01 | | | ASR | | | " | " | " | | |
| | | | | X | 02 | | | LSL | | | " | " | " | | |
| | | | | X | 03 | | | LSR | | | " | " | " | | |
| | | | | X | 04 | | | CSL | | | " | " | " | | |
| | | | | X | 05 | | | CSR | | | " | " | " | | |
| | | | | X | 06 | | | SCAN | | | RESERVED | | | | |
| | | | | I | 07 | | | IABZ | | +/- | P RELATIVE DISPLACEMENT | | | | |
| | | | | X | 10 | | | TASL | | | SHIFT COUNT L | | | | |
| | | | | X | 11 | | | TASR | | | " | " | " | | |
| | | | | I | 12 | | | IXBZ | | +/- | P RELATIVE DISPLACEMENT | | | | |
| | | | | I | 13 | | | DXBZ | | +/- | " | " | | " | |
| | | | | I | 14 | | | BCY | | +/- | " | " | | " | |
| | | | | I | 15 | | | BNCY | | +/- | " | " | | " | |
| | | | | X | 16 | | | TNSL | | | RESERVED | | | | |
| | | | | Y | 17 | | | SPARE | | | | | | | |
| | | | | X | 20 | | | DASL | | | SHIFT COUNT L | | | | |
| | | | | X | 21 | | | DASR | | | " | " | " | | |
| | | | | X | 22 | | | DLSL | | | " | " | " | | |
| | | | | X | 23 | | | DLSR | | | " | " | " | | |
| | | | | X | 24 | | | DCSL | | | " | " | " | | |
| | | | | X | 25 | | | DCSR | | | " | " | " | | |
| | | | | I | 26 | | | CPRB | | +/- | P RELATIVE DISPLACEMENT | | | | |
| | | | | I | 27 | | | DABZ | | +/- | " | " | | " | |
| | | | | I | 30 | | | BOV | | +/- | " | " | | " | |
| | | | | I | 31 | | | BNOV | | +/- | " | " | | " | |
| | | | | X | 32 | | | TBC | | | BIT POSITION | | | | |
| | | | | X | 33 | | | TRBC | | | " | " | | | |
| | | | | X | 34 | | | TSBC | | | " | " | | | |
| | | | | X | 35 | | | TCBC | | | " | " | | | |
| | | | | I | 36 | | | BRO | | +/- | P RELATIVE DISPLACEMENT | | | | |
| | | | | I | 37 | | | BRE | | +/- | " | " | | " | |
| 02 | SUB OPCODE 2 | | | 00 | MOVE OPS | | | 0 | MOVE | | PB/DB | RESERVED | | SDEC | |
| | | | | | | | | 1 | MVB | | " | " | | " | |
| | | | | | | | | 2 | MVBL | | 0 | " | | " | |
| | | | | | | | | 2 | SCW | | 1 | " | | " | |
| | | | | | | | | 3 | MVLB | | 0 | " | | " | |
| | | | | | | | | 3 | SCU | | 1 | " | | " | |
| | | | | | | | | 4 | MVBW | | N | A | U | " | |
| | | | | | | | | 5 | CMPB | | PB/DB | RESERVED | | SDEC | |

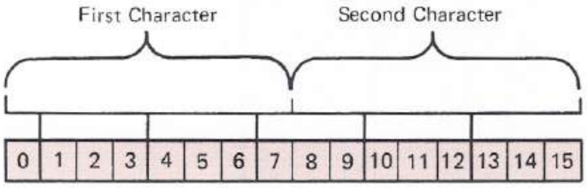
# CONSOLIDATED CODING TABLE

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 02 | SUB OPCODE 2 | | | 00 | MINI OPS | | | 14 | | RSW | | | RESERVED | | 0 |
| | | | | | | | | 14 | | LLSH | | | RESERVED | | 1 |
| | | | | | | | | 15 | | PLDA | | | RESERVED | | 0 |
| | | | | | | | | 15 | | PSTA | | | RESERVED | | 1 |
| | | | | 01 | SPARE | | | | | | | | | | |
| | | | | 02 | LDI | | | | | IMMEDIATE OPERAND N | | | | | |
| | | | | 03 | LDXI | | | | | | " | " | " | | |
| | | | | 04 | CMPI | | | | | | " | " | " | | |
| | | | | 05 | ADDI | | | | | | " | " | " | | |
| | | | | 06 | SUBI | | | | | | " | " | " | | |
| | | | | 07 | MPYI | | | | | | " | " | " | | |
| | | | | 10 | DIVI | | | | | | " | " | " | | |
| | | | | 11 | PSHR | | | | DB | DL | Z | STA | X | Q | S |
| | | | | 12 | LDNI | | | | | IMMEDIATE OPERAND N | | | | | |
| | | | | 13 | LDXN | | | | | | " | " | " | | |
| | | | | 14 | CMPN | | | | | | " | " | " | | |
| | | | | 15 | EXF | | | | BEGINNING BIT # | | | | | # OF BITS | |
| | | | | 16 | DPF | | | | " | " | " | | | " " " | |
| | | | | 17 | SETR | | | | DB | DL | Z | STA | X | Q | S |
| 03 | SUB OPCODE 3 | | | 00 | SPECIAL OP | | | 00 | | SPARE | | | | | |
| | | | | | | | | 01 | | PAUS | | | K FIELD | | |
| | | | | | | | | 02 | | SED | | | " " | | |
| | | | | | | | | 03 | | XCHD | | | " " | | |
| | | | | | | | | 04 | | SMSK | | | " " | | |
| | | | | | | | | 05 | | RMSK | | | " " | | |
| | | | | | | | | 06 | | XEQ | | | " " | | |
| | | | | | | | | 07 | | SIO | | | " " | | |
| | | | | | | | | 10 | | RIO | | | " " | | |
| | | | | | | | | 11 | | WIO | | | " " | | |
| | | | | | | | | 12 | | TIO | | | " " | | |
| | | | | | | | | 13 | | CIO | | | " " | | |
| | | | | | | | | 14 | | CMD | | | " " | | |
| | | | | | | | | 15 | | SIRF | | | " " | | |
| | | | | | | | | 16 | | SIN | | | " " | | |
| | | | | | | | | 17 | | HALT | | | " " | | |
| | | | | 01 | SCAL | | | | | | STT ENTRY # N | | | | |
| | | | | 02 | PCAL | | | | | | " | " | " " | | |
| | | | | 03 | EXIT | | | | | | | N FIELD | | | |
| | | | | 04 | SXIT | | | | | | | " " | | | |
| | | | | 05 | ADXI | | | | | | | " " | | | |
| | | | | 06 | SBXI | | | | | | | " " | | | |
| | | | | 07 | LLBL | | | | | | PL- DISPLACEMENT N | | | | |
| | | | | 10 | LDPP | | | | | | P+ DISPLACEMENT N | | | | |
| | | | | 11 | LDPN | | | | | | P- DISPLACEMENT N | | | | |
| | | | | 12 | ADDS | | | | | | IMMEDIATE OPERAND N | | | | |
| | | | | 13 | SUBS | | | | | | " | " | " | | |
| | | | | 14 | TSBM | | | | | | DB+ DISPLACEMENT N | | | | |
| | | | | 15 | ORI | | | | | | IMMEDIATE OPERAND N | | | | |
| | | | | 16 | XORI | | | | | | " | " | " | | |
| | | | | 17 | ANDI | | | | | | " | " | " | | |
| 04 | | LOAD | | X | I | | | PDQS ADDRESS MODE & DISPLACEMENT | | | | | | | |
| 05 | | TBA | | 0 | 0 | 0 | +/- | | P RELATIVE DISPLACEMENT | | | | | | |
| | | MTBA | | 0 | 1 | 0 | +/- | | " | " | " | | | | |
| | | TBX | | 1 | 0 | 0 | +/- | | " | " | " | | | | |
| | | MTBX | | 1 | 1 | 0 | +/- | | " | " | " | | | | |
| | | STOR | | X | I | 1 | | DQS ADDRESS MODE & DISPLACEMENT | | | | | | | |
| 06 | | CMPM | | X | I | | | PDQS | " | " | " | " | | | |
| 07 | | ADDM | | X | I | | | " | " | " | " | " | | | |
| 10 | | SUBM | | X | I | | | " | " | " | " | " | | | |
| 11 | | MPYM | | X | I | | | " | " | " | " | " | | | |
| 12 | | INCM | | X | I | 0 | | DQS | " | " | " | " | | | |
| | | DECM | | X | I | 1 | | " | " | " | " | " | | | |
| 13 | | LDX | | X | I | | | PDQS | " | " | " | " | | | |
| 14 | | BR | | X | I | 0 | +/- | | P RELATIVE DISPLACEMENT | | | | | | |
| | | BR | | X | 1 | 1 | DQS ADDRESS MODE (INDIRECT) & DISPLACEMENT | | | | | | | | |
| | | BCC | | I | 0 | 1 | CCG | CCE | CCL | +/- | P RELATIVE DISPLACEMENT | | | | |
| 15 | | LDB | | X | I | 0 | | DQS ADDRESS MODE & DISPLACEMENT | | | | | | | |
| | | LDD | | X | I | 1 | | " | " | " | " | " | | | |
| 16 | | STB | | X | I | 0 | | " | " | " | " | " | | | |
| | | STD | | X | I | 1 | | " | " | " | " | " | | | |
| 17 | | LRA | | X | I | | | PDQS | " | " | " | " | | | |

# CHARACTER CODES

| ASCII Character | First Character Octal Equivalent | Second Character Octal Equivalent |
|---|---|---|
| A | 040400 | 000101 |
| B | 041000 | 000102 |
| C | 041400 | 000103 |
| D | 042000 | 000104 |
| E | 042400 | 000105 |
| F | 043000 | 000106 |
| G | 043400 | 000107 |
| H | 044000 | 000110 |
| I | 044400 | 000111 |
| J | 045000 | 000112 |
| K | 045400 | 000113 |
| L | 046000 | 000114 |
| M | 046400 | 000115 |
| N | 047000 | 000116 |
| O | 047400 | 000117 |
| P | 050000 | 000120 |
| Q | 050400 | 000121 |
| R | 051000 | 000122 |
| S | 051400 | 000123 |
| T | 052000 | 000124 |
| U | 052400 | 000125 |
| V | 053000 | 000126 |
| W | 053400 | 000127 |
| X | 054000 | 000130 |
| Y | 054400 | 000131 |
| Z | 055000 | 000132 |
| a | 060400 | 000141 |
| b | 061000 | 000142 |
| c | 061400 | 000143 |
| d | 062000 | 000144 |
| e | 062400 | 000145 |
| f | 063000 | 000146 |
| g | 063400 | 000147 |
| h | 064000 | 000150 |
| i | 064400 | 000151 |
| j | 065000 | 000152 |
| k | 065400 | 000153 |
| l | 066000 | 000154 |
| m | 066400 | 000155 |
| n | 067000 | 000156 |
| o | 067400 | 000157 |
| p | 070000 | 000160 |
| q | 070400 | 000161 |
| r | 071000 | 000162 |
| s | 071400 | 000163 |
| t | 072000 | 000164 |
| u | 072400 | 000165 |
| v | 073000 | 000166 |
| w | 073400 | 000167 |
| x | 074000 | 000170 |
| y | 074400 | 000171 |
| z | 075000 | 000172 |
| 0 | 030000 | 000060 |
| 1 | 030400 | 000061 |
| 2 | 031000 | 000062 |
| 3 | 031400 | 000063 |
| 4 | 032000 | 000064 |
| 5 | 032400 | 000065 |
| 6 | 033000 | 000066 |
| 7 | 033400 | 000067 |
| 8 | 034000 | 000070 |
| 9 | 034400 | 000071 |
| NUL | 000000 | 000000 |
| SOH | 000400 | 000001 |
| STX | 001000 | 000002 |
| ETX | 001400 | 000003 |
| EOT | 002000 | 000004 |
| ENQ | 002400 | 000005 |

| ASCII Character | First Character Octal Equivalent | Second Character Octal Equivalent |
|---|---|---|
| ACK | 003000 | 000006 |
| BEL | 003400 | 000007 |
| BS | 004000 | 000010 |
| HT | 004400 | 000011 |
| LF | 005000 | 000012 |
| VT | 005400 | 000013 |
| FF | 006000 | 000014 |
| CR | 006400 | 000015 |
| SO | 007000 | 000016 |
| SI | 007400 | 000017 |
| DLE | 010000 | 000020 |
| DC1 | 010400 | 000021 |
| DC2 | 011000 | 000022 |
| DC3 | 011400 | 000023 |
| DC4 | 012000 | 000024 |
| NAK | 012400 | 000025 |
| SYN | 013000 | 000026 |
| ETB | 013400 | 000027 |
| CAN | 014000 | 000030 |
| EM | 014400 | 000031 |
| SUB | 015000 | 000032 |
| ESC | 015400 | 000033 |
| FS | 016000 | 000034 |
| GS | 016400 | 000035 |
| RS | 017000 | 000036 |
| US | 017400 | 000037 |
| SPACE | 020000 | 000040 |
| ! | 020400 | 000041 |
| " | 021000 | 000042 |
| # | 021400 | 000043 |
| $ | 022000 | 000044 |
| % | 022400 | 000045 |
| & | 023000 | 000046 |
| ' | 023400 | 000047 |
| ( | 024000 | 000050 |
| ) | 024400 | 000051 |
| * | 025000 | 000052 |
| + | 025400 | 000053 |
| , | 026000 | 000054 |
| - | 026400 | 000055 |
| . | 027000 | 000056 |
| / | 027400 | 000057 |
| : | 035000 | 000072 |
| ; | 035400 | 000073 |
| < | 036000 | 000074 |
| = | 036400 | 000075 |
| > | 037000 | 000076 |
| ? | 037400 | 000077 |
| @ | 040000 | 000100 |
| [ | 055400 | 000133 |
| \ | 056000 | 000134 |
| ] | 056400 | 000135 |
| Δ | 057000 | 000136 |
| — | 057400 | 000137 |
| ` | 060000 | 000140 |
| { | 075400 | 000173 |
| \| | 076000 | 000174 |
| } | 076400 | 000175 |
| ~ | 077000 | 000176 |
| DEL | 077400 | 000177 |



# ALPHABETICAL INDEX OF INSTRUCTIONS

HEWLETT hp PACKARD

Printed in U.S.A.