HEWLETT hp PACKARD

HP 3000 Series II/III Computer System

System Reference Manual

HP 3000 Series II/III Computer Systems

System Reference Manual

HEWLETT PACKARD

5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA 95050

Part No. 30000-90020

Printed in U.S.A. 7/78

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

Page	Date
i to ii	7/78
iii to iv	1/79
v to ix	7/78
1-1	7/78
1-2 to 1-3	1/79
2-1 to 2-10	7/78
3-1 to 3-66	7/78
4-1 to 4-32	7/78
5-1 to 5-20	7/78
6-1 to 6-24	7/78
7-1 to 7-40	7/78
I-1 to I-8	7/78

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

Second Edition . . . . . July 1978
Update Package No. 1 . . . . . Jan 1979

CONTENTS

Section I — Page

INTRODUCTION

Purpose of Manual 1-1
System Descriptions 1-1

Section II — Page

SYSTEM FEATURES

Central Processing Unit (CPU) 2-2
- Separate Code and Data 2-2
- Data Stacks 2-2
- Variable-Length Segmentation 2-4
- Registers 2-5
- Instructions 2-6
- Microcode 2-6

Main Memory 2-7
Input/Output 2-7
- Direct I/O (DIO) 2-8
- Programmed I/O (SIO) 2-8
  - Multiplexer Channel 2-8
  - Selector Channel 2-8

Device Reference Table (DRT) 2-8
Data Service and Interrupt Priorities 2-8
Interrupt System 2-9
Peripherals 2-9

Section III — Page

THEORY OF OPERATION

Introduction 3-1
System-Level Description 3-1
CPU General Information 3-2
- Pipeline 3-2
- Data Formats 3-4
  - Byte Format 3-6
  - Logical Format 3-6
  - Single-Word, Fixed-Point Format 3-6
  - Double-Word, Fixed-Point Format 3-6
  - Floating-Point Format 3-6
  - Long Floating-Point Format 3-7
- Instruction Formats 3-8
  - General Format 3-9
  - Stack Op 3-9
  - Shift 3-9
  - Branch 3-9
  - Bit Test 3-11
  - Move 3-11
  - Special 3-11
  - Immediate 3-11
  - Field 3-11
  - Register Control 3-11
  - Program Control 3-11
  - I/O and Interrupt 3-11
  - Loop Control 3-11
  - Memory Address 3-11
- Status Word Format 3-12

CPU Registers 3-15
- Code Segment Registers 3-15
  - PB Register 3-15
  - PB-Bank Register 3-15
  - P Register 3-15
  - PL Register 3-15
- Data Segment Registers 3-15
  - DL Register 3-15
  - DB Register 3-15
  - DB-Bank Register 3-15
  - Q Register 3-15
  - SM Register 3-17
  - SR Register 3-17
  - S Pointer 3-17
  - Z Register 3-17
  - Stack Bank Register 3-17
- Other CPU Registers 3-17
  - Index Register 3-17
  - Status Register 3-17

Privileged Mode 3-18
Addressing Conventions 3-18
- Memory Addressing 3-18
- Indirect Addressing 3-20
  - Code Indirect 3-20
  - Data Indirect 3-21
- Indexing 3-21
  - Code Indexing 3-21
  - Data Indexing 3-21
- Byte Addressing 3-24
  - Direct 3-25
  - Direct, Indexed 3-25
  - Indirect 3-25
  - Indirect, Indexed 3-25
- Double-Word Indexing 3-25
- Bounds Checking 3-25
  - Program Transfer 3-25
  - Program References 3-26
  - Data References 3-26
  - Stack Overflow 3-26
  - Stack Underflow 3-26
- Access to DB- Area 3-28
  - Word Addressing 3-28
  - Byte Addressing 3-28

Block-Level Description 3-29
- Bus System 3-29
  - Central Data Bus 3-31
  - IOP Bus 3-31
  - Selector Channel Bus 3-31
  - Multiplexer Channel Bus 3-31
  - Power Bus 3-31
- Overview 3-31
  - Unconditional Jump 3-32
  - Conditional Jump 3-32
- Central Processor Unit 3-32
  - Next Instruction Register 3-33
  - Current Instruction Register 3-33
  - CMUX and CMUX Control 3-33
  - Mapper and Mapper Control 3-33
  - Look Up Table ROM 3-33
  - V-Bus MUX and V-Bus Control 3-34
  - ROM Address Register 3-34
  - Save Register 3-35

CONTENTS (continued)

Read-Only Memory 3-35
ROM Output Registers 3-35
Microcode Jumps 3-35
S-Bus Field Decoder 3-36
Store Field Decoder 3-36
Function Field Decoder 3-36
Skip Field Decoder 3-36
Shift Field Decoder 3-36
Special Field Decoder 3-36
MCU Option Field Decoder 3-61
R-Bus Field Decoder 3-61
Processor Registers 3-61
Renamer 3-61
Top-of-Stack Registers 3-62
Index Register 3-62
Stack Limit Register 3-62
Program Limit Register 3-62
Scratch Pad 0 Register 3-62
Scratch Pad 1 Register 3-62
Stack Register 3-62
Program Base Register 3-62
Data Limit Register 3-62
Stack Memory Register 3-63
Data Base Register 3-63
Q Register 3-63
Scratch Pad 2 Register 3-63
Scratch Pad 3 Register 3-63
Process Clock Register 3-63
Program Counter Register 3-63
Operand Register 3-63
Status Register 3-64
Counter Register 3-64
Overflow 3-65
Carry 3-65
Condition Code Logic 3-65
Pre-Adder 3-65
R-Bus Register 3-65
S-Bus Register 3-65
Arithmetic Logic Unit 3-65
Shifter 3-65
Decimal Corrector 3-65
ACOR 3-66
DCOR 3-66
CPX1 Register 3-66
CPX2 Register 3-66

Section IV Page
MEMORY SEGMENTATION
Introduction 4-1
Virtual Memory 4-1
Processes 4-1
Procedures 4-3
System Library 4-3
Memory Management 4-3
Code Segmentation 4-4
Data Segmentation 4-5
Main Memory Organization 4-6
Basic Table Structures 4-7
Code Segment Table and Extension 4-7
Data Segment Table 4-9
Code Segment Linkage 4-10
Stack Operation 4-14
Examples of Stack Usage 4-19
Basic Arithmetic 4-20
Procedure Calls 4-21
Recursion 4-26
Main Program Call 4-27
Test for Zero 4-31
First Recursive Call 4-31
Successive Recursions 4-31
First Exit 4-31
First Recursive Exit 4-32
Successive Exits 4-32

Section V Page
INPUT/OUTPUT SYSTEM
Introduction 5-1
File System Operation 5-2
Definition of Terms 5-2
I/O Instructions 5-5
General I/O Operation 5-6
Direct I/O Operation 5-9
Direct Read 5-9
Direct Write 5-11
Blocked/Unblocked I/O 5-13
Blocked I/O 5-13
Unblocked I/O 5-13
Hardware I/O System 5-14
Hardware Elements 5-14
I/O Processor 5-14
Multiplexer Channel 5-15
Selector Channel 5-16
Device Controller 5-16
Peripheral Device 5-16
I/O Programming 5-16
I/O Program Word 5-16
Typical I/O Program Operation 5-18

Section VI Page
INTERRUPT SYSTEM
Interrupt System Overview 6-1
Interrupt Control Stack (ICS) 6-2
Interrupt Types 6-4
External Interrupts 6-4
ICS Internal Interrupts 6-6
Non-ICS Internal Interrupts 6-6
External Interrupt Processing 6-7
Interrupt Program Pointer 6-7
Sequence of Operations 6-7
Internal Interrupt Processing 6-12
General Descriptions 6-13
Bounds Violation 6-13
Illegal Memory Address 6-13
Non-Responding Module 6-13

System Parity Error 6-13
Address Parity Error 6-13
Data Parity Error 6-13
Module Interrupt 6-14
Power Fail 6-14
Unimplemented Instruction 6-14
STT Violation 6-14
CST Violation 6-14
DST Violation 6-14
Stack Underflow 6-14
Privileged Mode Violation 6-14
Stack Overflow 6-14
Integer Overflow 6-14
Floating-Point Overflow 6-15
Floating-Point Underflow 6-15
Integer Divide by Zero 6-15
Floating-Point Divide by Zero 6-15
Extended Precision Floating-Point Overflow 6-15
Extended Precision Floating-Point Underflow 6-15
Extended Precision Floating-Point Divide by Zero 6-15
Decimal Overflow 6-15
Invalid ASCII Digit 6-15
Invalid Decimal Digit 6-16
Invalid Word Count 6-16
Result Word Count Overflow 6-16
Decimal Divide by Zero 6-16
Absent Code Segment 6-16
Trace 6-16
STT Entry Uncallable 6-16
Absent Data Segment 6-16
Power On 6-16
Cold Load 6-16
Sequence for ICS Type Internal Interrupts 6-17
Sequence for Non-ICS Type Interrupts 6-19
Interrupt Handler 6-19
DISP Instruction 6-21
Pseudo Enabling/Disabling the Dispatcher 6-23
IXIT Instruction 6-23

Section VII Page
FUNCTIONAL OPERATION
Introduction 7-1
Central Processor Unit 7-1
Module Control Unit 7-1
Central Data Bus Transmissions 7-5
Fetch Next Instruction 7-5
CPU Address Transmit 7-5
Memory Receive and Transmit 7-6
CPU Receive 7-6
Fetch an Operand 7-6
CPU Address Transmit 7-6
Memory Receive and Transmit 7-7
CPU Receive 7-7
Store an Operand 7-7
CPU Address Transmit 7-7
Memory Receive 7-7
CPU Data Transmit 7-7
Memory Receive 7-8
Command a Module 7-8
I/O System 7-9
I/O Priorities 7-9
I/O Data Routes 7-10
Multiplexer Channel Device 7-10
Selector Channel Device 7-10
Transfer Modes 7-10
Multiplexer Channel 7-11
Selector Channel 7-11
I/O Processor 7-12
IOP Logic 7-12
I/O Command 7-12
IOP Control 7-12
Interrupt Control 7-12
INT DEVNO 7-12
Data Output Registers 7-13
Data Input Registers 7-13
IOP Module Control Unit 7-13
Initialize 7-13
DRT Fetch 7-14
I/O Program Word Transfers 7-15
IOCW Fetch 7-15
IOAW Fetch 7-16
IOAW Store 7-16
Next Operation 7-17
Data Transfers 7-17
Address Transfer 7-17
Output Transfer 7-17
Input Transfer 7-18
End of Transfer by Word Count 7-18
End of Transfer by Device 7-18
Interrupts 7-18
Selector Channel 7-19
Port Controller 7-19
Initiator Sequence 7-20
Fetch Sequence 7-21
Execute Sequences 7-22
Sense 7-22
Interrupt 7-22
Jump 7-22
Control 7-23
Set Bank 7-23
Read 7-23
Return Residue 7-24
Write 7-24
End 7-25
Direct I/O Operation 7-25
TIO 7-25
RIO 7-26
CIO 7-26
WIO 7-26

ILLUSTRATIONS

Title	Page
2-Bay Computer System	1-2
3-Bay Computer System	1-2
Series III (23435A) 1- and 2-Bay Computer Systems	1-3
HP 3000 Series II and III Hardware Organization	2-1
Subprogram's View of the Data Stack	2-4
Typical Computer System	3-2
Data Formats	3-5
Floating-Point Data Representation	3-8
Instruction Groups	3-10
Status Word Format	3-13
CPU Registers	3-16
Memory Addressing Modes	3-19
Examples of Indirect Addressing	3-22
Examples of Indexing	3-23
Examples of Byte Addressing	3-24
Addressing and Stack Bounds	3-27
Access to DB- Area	3-28
Bus System Block Diagram	3-30
Code Sharing and Data Privacy	4-2
Segmented Memory Management	4-4
Code Segment Evolution	4-5
Basic Data Structures	4-8
Code Segment Table Entry Format	4-9
Data Segment Table Entry Format	4-10
Procedure Calls Within and Between Code Segments	4-11
Formats Associated with Code Segments	4-13
Stack Registers and One Stack	4-15
Top of Stack in the CPU	4-16
Stack Mark Chain	4-17
Standard 4-Word Stack Marker Format	4-19
Basic Arithmetic Stack Operations	4-20
Declaring and Calling a Procedure	4-22
Executing a Simple Procedure	4-23
Example of Recursive Procedure	4-28
Recursive Calls	4-29
Recursive Exits	4-30
Basic I/O Access Methods	5-3
File System Basic Operation	5-3
Fundamental Elements of I/O System	5-4
Device Reference Table	5-6
I/O System Overview	5-7
Direct Read for Terminal Devices	5-10
Direct Write for Terminal Devices	5-12
Blocked and Unblocked I/O	5-14
Hardware I/O Elements	5-15
I/O Program Word Format	5-17
I/O Program Operation	5-18
Dispatcher Marker on ICS	6-3
Interrupt System Overview	6-5
Device Controller	6-8
First Level External Interrupt	6-9
Second Level Interrupt	6-10
ICS Internal Interrupt Operations	6-18
Non-ICS Internal Interrupt Operations	6-20
Interrupt Handler	6-21
IXIT Instruction	6-24
Simplified Logic Diagram, Plan View	7-2
Central Processor Unit (CPU) Simplified Logic Diagram	7-3
Model Control Unit (MCU) Simiplified Logic Diagram	7-27
Memory Module Simplified Logic Diagram	7-29
Input/Output Processor (IOP) Simplified Logic Diagram	7-31
Interrupt Poll and Data Poll	7-32
Input/Output Data Routes	7-33
Basic Comparison of Multiplexer and Selector Channels	7-34
Multiplexer Channel and Device Controler Simplified Logic Diagram	7-35
Multiplexer Channel Simplified Logic Diagram	7-36
Port Controller Simiplified Logic Diagram	7-37
Device Controller on a Selector Channel Simplified Logic Diagram	7-38
Selector Channel Simiplified Logic Diagram	7-39
Direct I/O Device Controller Simplified Logic Diagram	7-40

TABLES

Title	Page
CPU Features	2-3
Machine Registers	2-5
Main Memory Features	2-7
Condition Codes	3-14
Bounds Checks	3-26
S-Bus Field Code Definitions	3-37
Store Field Code Definitions	3-40
Function Field Code Definitions	3-44
Skip Field Code Definitions	3-50
Shift Field Code Definitions	3-53
Special Field Code Definitions	3-54
MCU Option Field Code Definitions	3-57
R-Bus Field Code Definitions	3-59
Top-of-Stack Namer Relationships	3-61
Low Main Memory Location Reservations	4-7
Recursive Program	4-27
Interrupt Types	6-1

SECTION I
INTRODUCTION

1-1. PURPOSE OF MANUAL

This manual is the reference document for the Hewlett-Packard HP 3000 Series II and III Computer Systems. Except where specified, the content of this manual applies equally to the Series II and Series III Computer Systems. The sections of this manual are arranged in a building block sequence for the reader who is unfamiliar with the computer systems. Specifically, this manual contains seven sections arranged as follows:

- Section I Provides an introduction to the HP 3000 Series II and Series III Computer Systems.
- Section II Describes the prominent features of the systems.
- Section III Provides a system-level description of the principles of operation for complete systems and a detailed block-level theory of operation description of the central processor units.
- Section IV Describes the memory segmentation schemes used in the HP 3000 Series II and III, including an in-depth description of the stack and stack operations.
- Section V Describes the I/O system as accessed via the file system.
- Section VI Describes system interrupts and trap processing.
- Section VII Provides a functional operation description of the systems.

1-2. SYSTEM DESCRIPTIONS

The HP 3000 Series II and Series III Computer Systems are general purpose computers with true multiprogramming and multilingual capabilities. They can simultaneously handle many interactive and batch operations; each in any of several programming languages. The systems feature a hardware stack architecture, variable-length code segmentation in a hardware-assisted virtual memory scheme, user protection, dynamic storage allocation, and integrated hardware/software design. The hardware and software work together in an interrelated manner with the hardware performing many operations conventionally performed by software. The computer systems use a common Multiprogramming Executive (MPE) operating system. MPE is a general purpose, disc-based software system that supervises the processing of user programs. MPE relieves the user of many program control, input/output, and other housekeeping responsibilities by monitoring and controlling the compilation, run preparation, loading, execution, and output of user programs. MPE also controls the order in which programs are executed and allocates the hardware and software resources they require.

The computer systems are configured with semiconductor memory. Series II memory sizes range from 64K words to 256K words (K = 1024). Series III memory sizes range from 128K words to 1024K words.

The Series II Computer System is available in two hardware configurations; a two-bay system (Model 6) and a three-bay system (Model 8). The Series III Computer System is offered in two product models, 32421A and 23435A, each of which is available in two hardware configurations. The Series III Computer System, product number 32421A, is available in the standard two-bay system and an optional three-bay (Option 200) system. Rack layouts for the two- and three-bay systems are shown in figures 1-1 and 1-2, respectively. The Series III Computer System, product number 23435A, is also available in two hardware configurations, a standard one-bay system and an optional two-bay (Option 200) system. The rack layouts for this product are shown in figure 1-3.

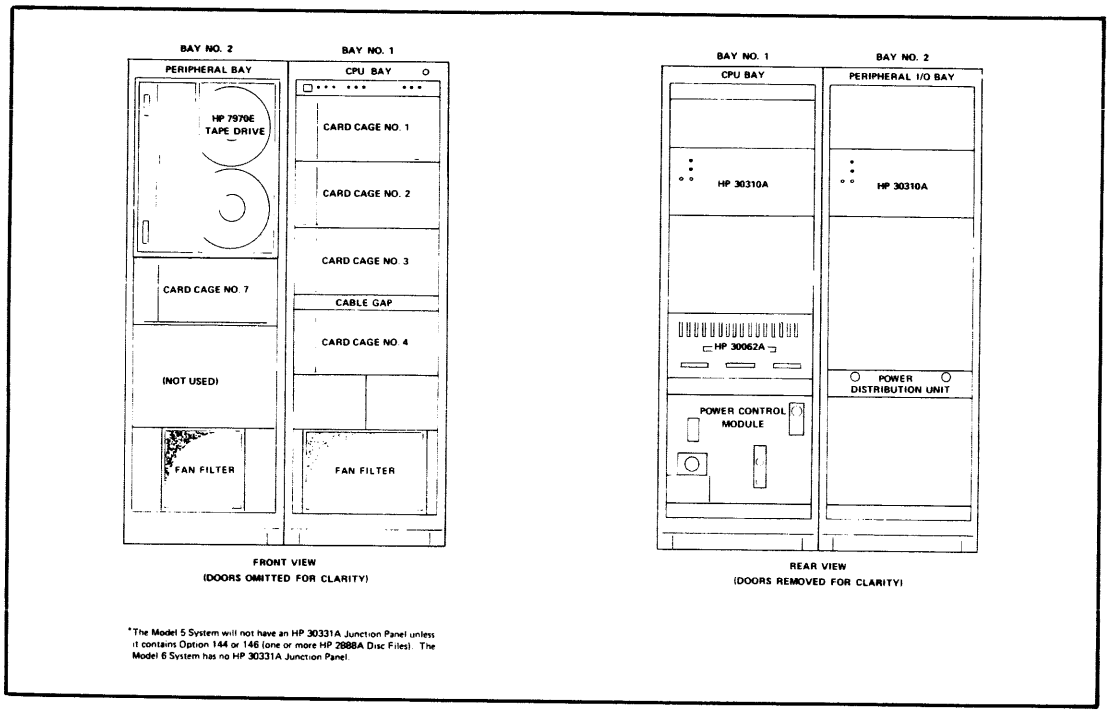


Figure 1-1. 2-Bay Computer System

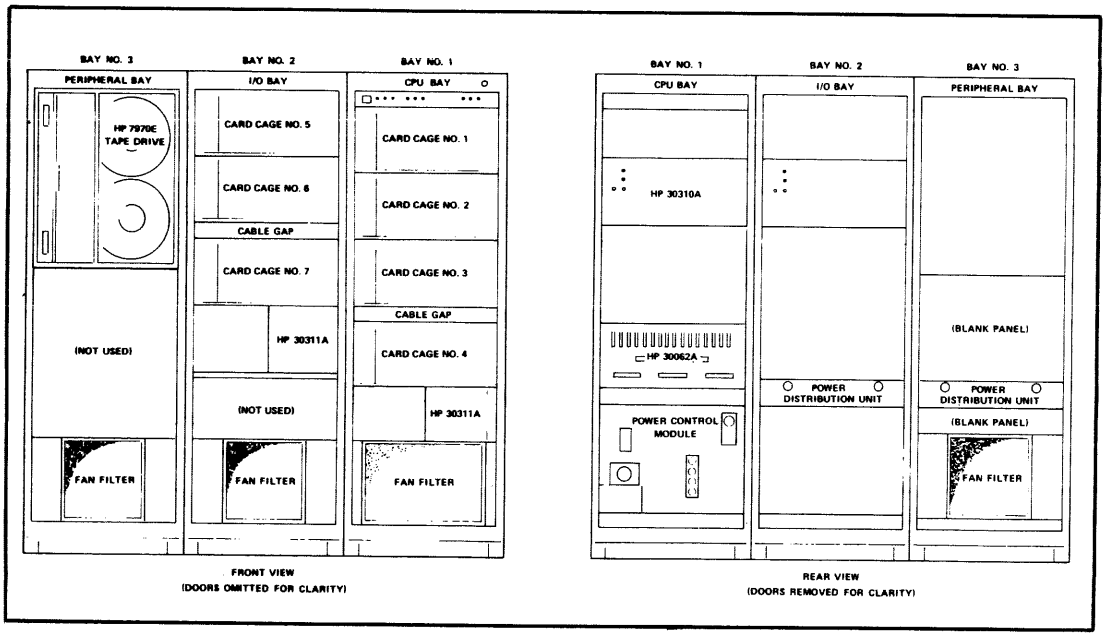
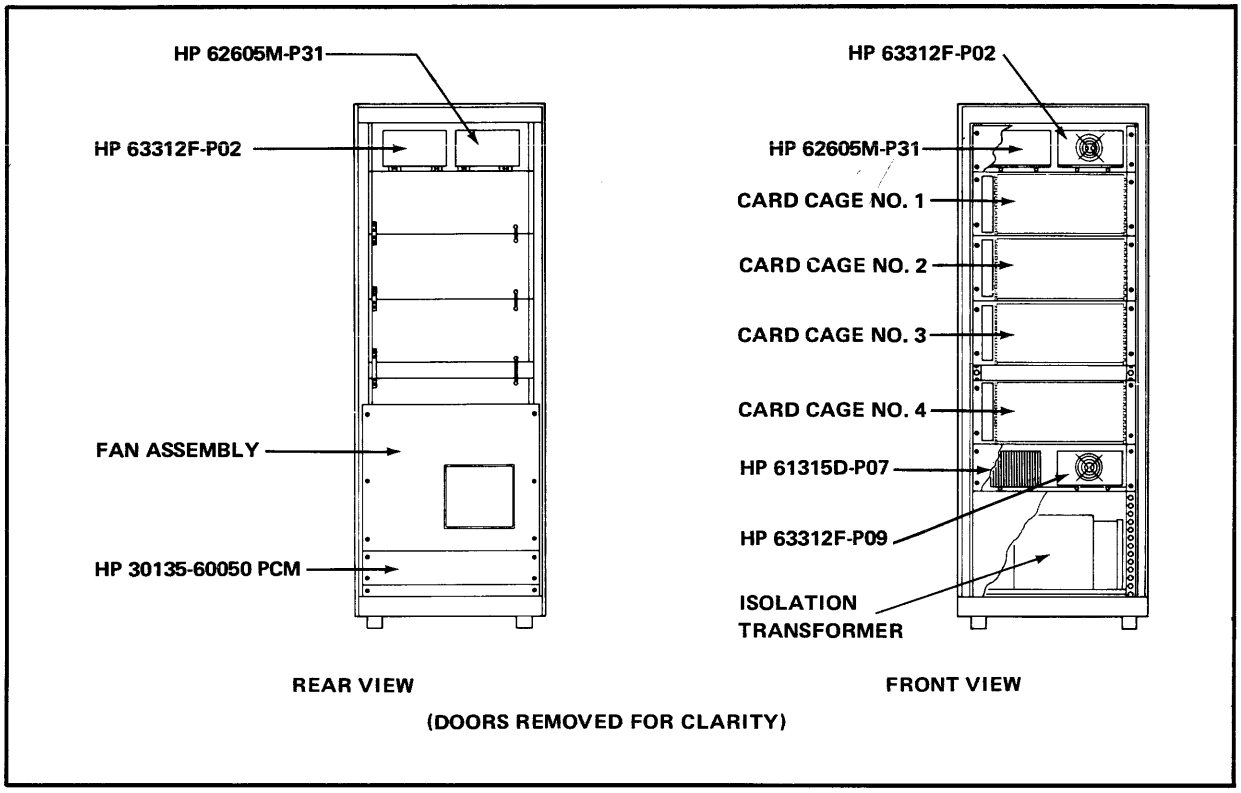


Figure 1-2. 3-Bay Computer System



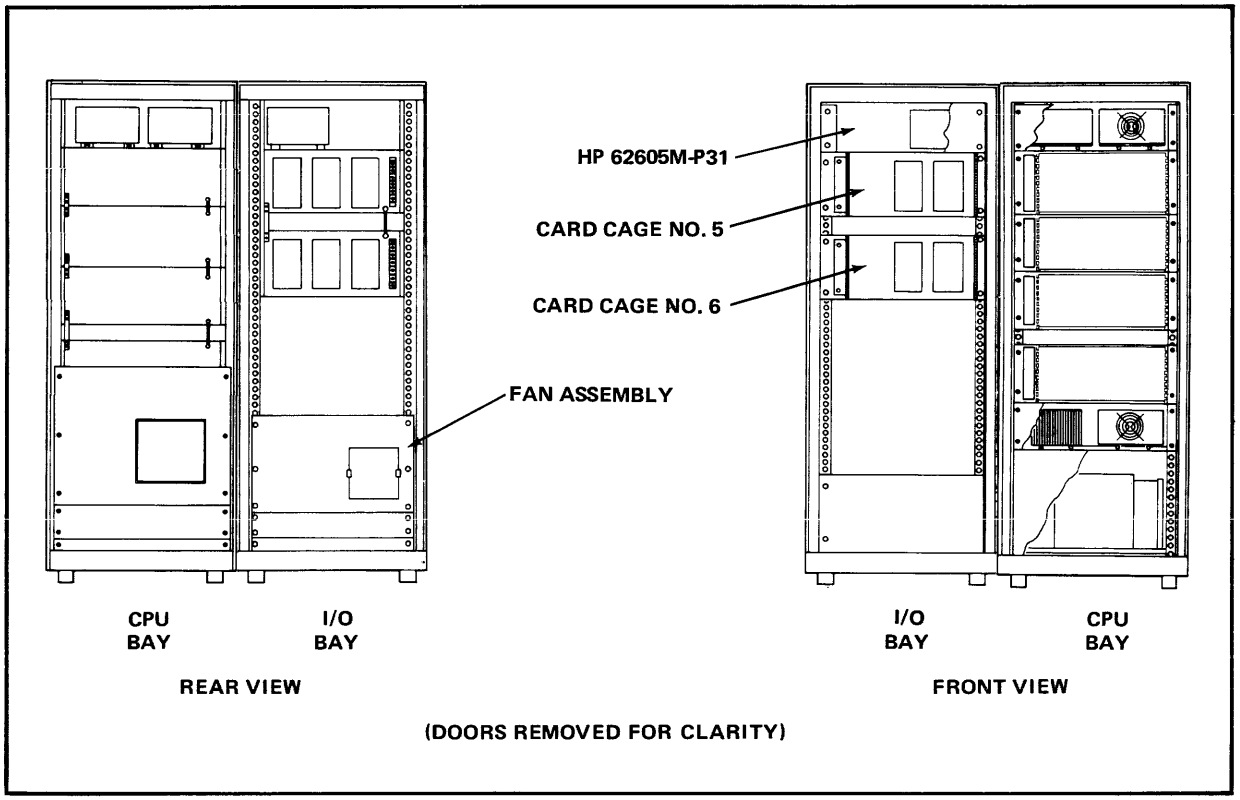


Figure 1-3. Series III (23435A) 1-Bay and 2-Bay Computer Systems

SYSTEM FEATURES

SECTION II

This section summarizes the architectural features of the HP 3000 Series II and III Computer Systems. The design of the hardware provides an efficient and powerful foundation upon which the software is built.

The hardware elements of the computer systems are organized as illustrated in figure 2-1. Communication between modules occurs over the Central Data Bus. The Central Processing Unit (CPU) and the Input/Output Processor (IOP), although independent of one another, share a common module address. This is resolved by giving the IOP a higher priority in the case where both processors concurrently request use of the bus. The CPU is controlled by a specially designed microprocessor to allow a great deal of flexibility in the machine instruction set. The computer systems also employ high-speed, semiconductor memory modules which use automatic fault detection and correction with no loss in performance.

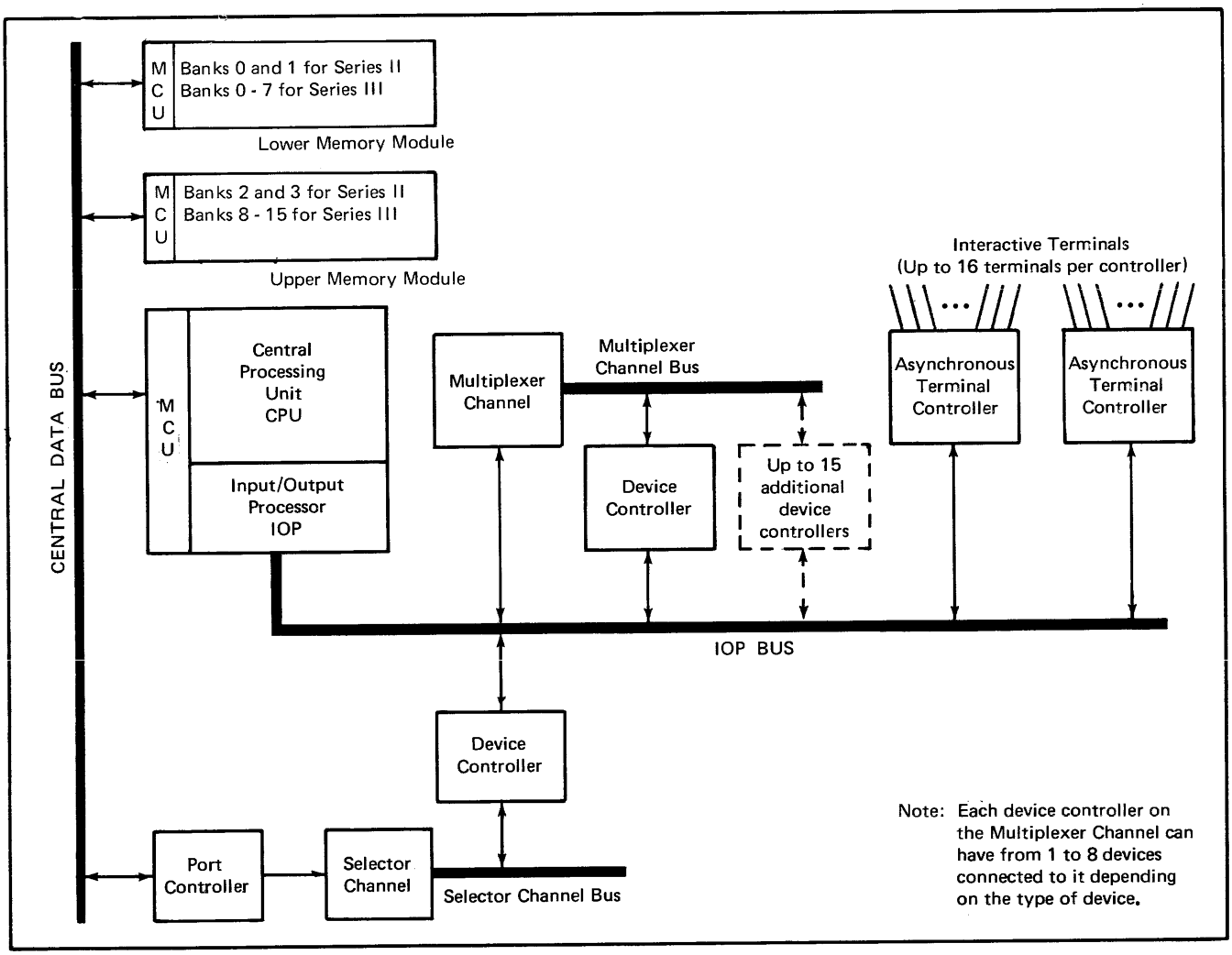


Figure 2-1. HP 3000 Series II and III Hardware Organization

The basic structure of independent modules organized around a Central Data Bus permits high-speed internal data rates. 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 hardware reconfiguration.

The presence of a separate IOP Bus which is totally dedicated to input/output data transfers means that the computer systems can always respond immediately to the needs of I/O devices regardless of what transfers are currently in progress between the various system modules. It also means that many I/O operations can be handled concurrently with CPU, main memory, and high-speed Selector Chanel operations.

Data may be transferred directly between main memory and high-speed peripheral devices in block mode by way of high-speed Selector Channels connected to the Central Data Bus. For lower speed devices, data may be multiplexed on a word-by-word basis by way of the IOP and a Multiplexer Channel. In both cases the I/O channels execute in parallel with CPU operations. Direct control of devices attached to the IOP bus is also possible through the use of the CPU's direct I/O instructions.

The configuration of hardware modules and peripheral devices is easily changed to accommodate system expansion.

2-1. CENTRAL PROCESSING UNIT (CPU)

The significant features of the CPU are listed in table 2-1.

2-2. SEPARATE CODE AND DATA

Code consists of the executable instructions and unchanging constants of a program or subprogram. As the code executes, the manipulated values are referred to as data. Code and data are maintained in strictly separate domains by the compilers and the system and cannot be intermixed (with the exception, however, 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. This means that main memory can be used with optimum efficiency since one copy of heavily used programs is shared by many users concurrently. Re-entrancy and stack-structured data (discussed in the next paragraph) together make possible subprogram recursion (a subprogram calling itself), which is essential for efficient compilers and system software. Also, since code is non-modifiable, exact copies of all active code are always retained on disc, thus allowing code to be overlayed without having to first write it back out to the disc.

2-3. DATA STACKS

The data for each user is organized into a *data stack*. In general, a *stack* is a storage area where the last item stored in is usually the first item taken out. In actual use, however, programs have direct access to all elements in the stack by specifying addresses relative to several CPU registers (the DB, S, and Q registers). The stack structure provides an efficient mechanism for parameter passing, dynamic allocation of temporary storage, efficient evaluation of arithmetic expressions, and recursive subprogram calls. In addition, it enables rapid context switching to establish a new environment on subprogram calls and interrupts. All features of the stack (including the automatic transferring of data to and from the CPU registers and checking for stack overflow and stack underflow) are implemented in the hardware.

Table 2-1. CPU Features

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
- Automatic restart after power failure
- Central Data Bus
- Bus parity checking
- Concurrent CPU and I/O operations

INSTRUCTIONS

- 209 powerful instructions
- All instructions except stack operations are 16 bits in length (stack operations may be packed two per word)
- 16- and 32-bit integer arithmetic
- 32- and 64-bit floating point arithmetic
- 28-digit packed decimal arithmetic
- Special instructions that optimize the efficiency of the operating system

When programming in a high-level language such as COBOL or RPG, all manipulation of the stack is done for the user automatically by the language processor. The user can, however, manipulate the stack directly by writing subprograms in SPL (Systems Programming Language for the HP 3000).

Figure 2-2 illustrates the general structure of a data stack as viewed from a subprogram. The white area represents filled locations, all containing valid data, while the shaded area represents available unfilled locations. The stack area is delimited by the locations defined as DB (Data Base) and S (Stack pointer). The addresses DB and S are retained in dedicated CPU registers. The Q-minus relative addressing area contains the parameters passed by the calling program. The area between Q and S contains the subprogram's local and temporary variables and intermediate results.

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 where the most recently written statement is farther down the page than previous (older) ones.

The area from S+ 1 to Z (the shaded area) is available for adding more elements to the stack. When a data word is added to the stack, it is stored in the next available location and the S pointer is automatically 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 in the undefined area.

To refer to recently stacked elements of data, S-minus relative addressing is used. Under this convention, 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. The others are DB-plus relative addressing and Q-minus and Q-plus relative addressing (the Q-register separates the data of a calling program or subprogram from the data of a called subprogram).

Since the top four elements of the stack are the most frequently used, there are four CPU registers (RA, RB, RC, and RD) which may at various times contain up to four of the topmost stack elements. The use of CPU registers in this way increases the execution speed of stack operations by reducing the number of memory references needed when manipulating data at or near the top of the stack. These registers are implicitly accessed by many of the machine instructions and whenever the stack locations S, S-1, S-2, or S-3 are specifically referenced. Data stacks are expanded automatically by the operating system during execution up to a maximum size of 64K bytes.

2-4. VARIABLE-LENGTH SEGMENTATION

Variable-length segmentation of code and data is used to facilitate multiprogramming. This method, in comparison with paging schemes, minimizes "checkerboard" waste of memory resources due to

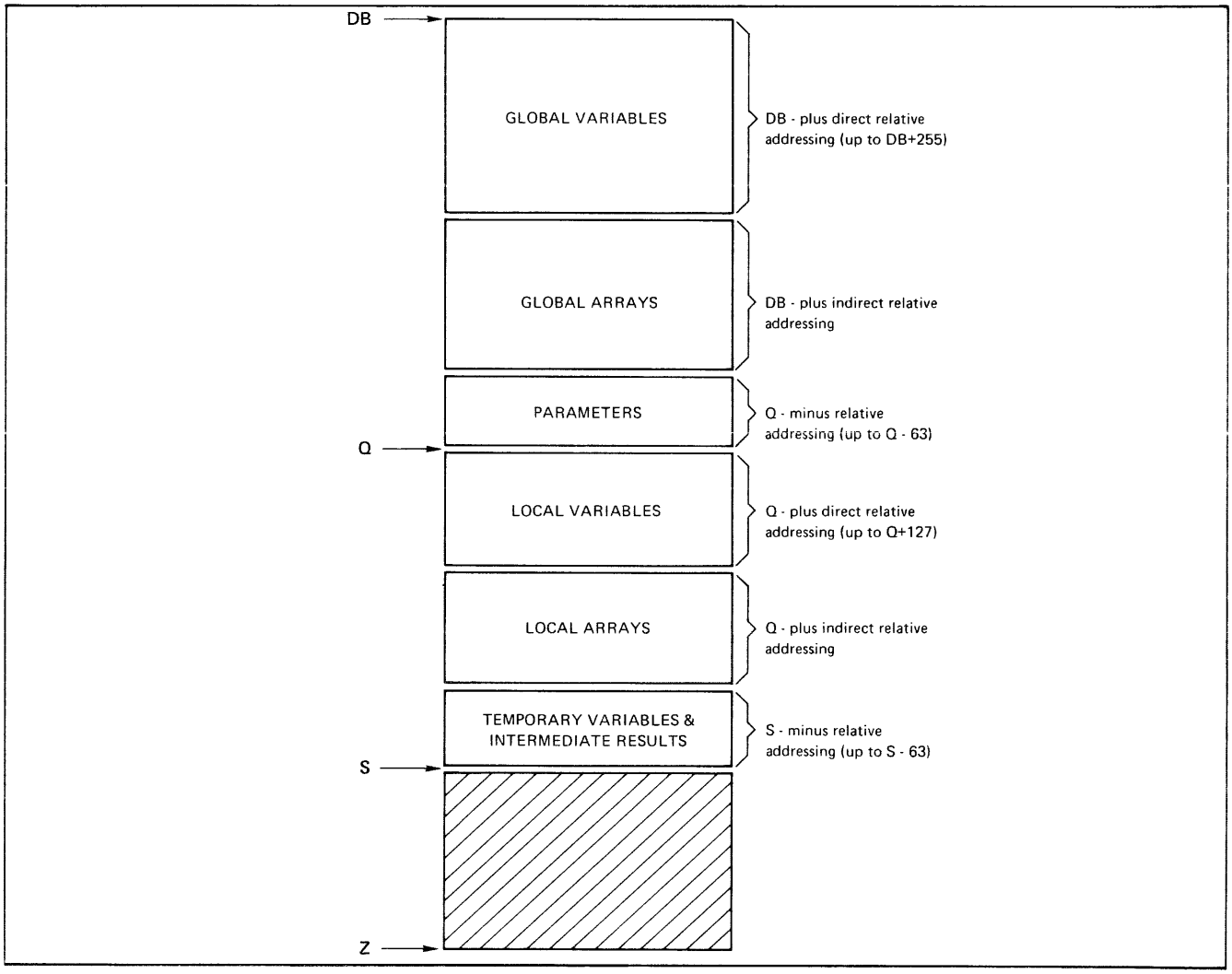


Figure 2-2. Subprogram's View of the Stack

internal fragmentation. It also makes it possible for the operating system to deal with logical instead of physical entities. This means, for example, that a particular subprogram can always be contained within one segment rather than arbitrarily divided between two physical pages, thus minimizing the amount of swapping that need be done while executing that subprogram. The location and size of all executing code segments is maintained in a *Code Segment Table* while the location and size of all associated data segments is maintained in a *Data Segment Table*. These tables are known to both hardware and software. Software uses them for dynamic memory management by the operating system. Hardware uses them to perform references and transfers between segments and to make sure that all segments required for current execution are present in main memory. Code segments may be up to 32,760 bytes in length. Data segments may be up to 65,528 bytes in length.

Segments are stored on disc and are brought into main memory only when needed. This design results in a virtual memory environment which appears to be many times larger than the maximum size of the physical main memory.

2-5. REGISTERS

The architecture of the computer systems employs a set of 38 special purpose registers rather than a smaller set of general purpose registers. Each register is included in the system to efficiently perform a specific function.

All addressing of code and data is done relative to hardware address registers. Thus, by simply changing the base addresses in these registers, segments are dynamically relocatable in memory. The few instances where absolute addresses are required are privileged operations handled by the operating system. The registers are summarized in table 2-2.

Table 2-2. Machine Registers

REGISTER	FUNCTION	REGISTER	FUNCTION
PB P PL PB-Bank	Code Segment Pointers	SWCH PCLK	Switch Register Program Clock Register
CIR NIR	Current Instruction Register Next Instruction Register	SP0 SP1 SP2 SP3 CTR ABS-Bank CPX1 CPX2 MOD	Scratch Pad, Flag, and Interrupt Registers
DL DB Q SM SR Z DB-Bank S-Bank	Stack Pointers		
		IOA IOD	I/O Registers
RA RB RC RD	Top of Stack Registers	ACOR DCOR OPND	Memory Address and Data Registers
X STA	Index Register Status Register	RAR SAVE	Firmware Address Registers

2-6. INSTRUCTIONS

There are 209 unique instructions for the computer systems, including instructions for performing 4-word extended precision floating point and 28-digit packed decimal arithmetic. Many of these instructions have multiple actions, several addressing modes (DB-plus, S-minus, Q-plus, Q-minus, P-plus, and P-minus), indirect addressing, and/or indexing which give a high complexity-to-instruction ratio. Code compression is achieved through the use of no-address (stack) instructions which implicitly use the content of the stack registers as operands. All instructions except the 63 stack operations are in 16-bit format; the stack operations may be packed two per 16-bit word to further enhance the code density.

A complete set of arithmetic instructions provides integer (16-bit two's complement), double integer (32-bit two's complement), logical (16-bit positive integer), 28-digit packed decimal (BCD coded digits packed two per byte), floating-point (32 bits including a 23-bit precision mantissa), and extended precision floating-point (64 bits including a 55-bit precision mantissa) arithmetic.

Other instructions are designed to facilitate string processing, subprogram linkage, and loop control.

Certain special instructions are designated as *privileged*, meaning that they were designed specifically for use by the operating system. They may, however, be used by programs which the installation permits to run in privileged mode. Some of these special instructions, such as the DISP instruction for entry to the MPE Dispatcher, instructions for enabling/disabling process switching, and instructions for data transfers between data segments, contribute greatly to the efficiency of the operating system.

WARNING

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in the privileged mode.

2-7. MICROCODE

The entire instruction set of the computer systems is in the form of microcoded operations in read-only memory. This microcode is executed by a microprocessor in response to machine instructions fetched into the CPU. By allowing microprogrammed hardware to execute certain repetitive functions, such as subprogram linkage, string processing, and buffer transfers (traditionally software-implemented), the amount of code and execution times are greatly reduced. In addition to the instruction set, other system functions have been microprogrammed, including the interrupt handler, a cold-start loader, the saving of critical environment information on power failure, automatic restart upon restoration of power, and a set of microdiagnostics that can be invoked from the front panel of the systems. All of these features are standard in the HP 3000 Series II and III.

The microprogrammed instructions routinely check for addressing bounds violations during execution and automatically interrupt to error handling routines if violations occur. These memory protection checks are usually overlapped with the operand fetch and therefore do not slow execution.

2-8. MAIN MEMORY

The significant features of main memory are listed in table 2-3.

Table 2-3. Main Memory Features

- High-speed, semiconductor, random access memory
- Automatic fault detection and single bit fault correction
- Seven memory sizes ranging from 64K words to 256K words for Series II and six memory sizes ranging from 128K words to 1024K words for Series III
- Rechargeable battery packs to maintain memory data for a minimum of 40 minutes during power failure

The computer systems use high-speed, semiconductor, random access memory which provides single-bit fault correction and some double-bit fault detection. The parity system is retained with this feature, thus maintaining integrity over the various busses. Series II main memory is available in 64K, 96K, 128K, 160K, 192K, 224K, and 256K word configurations (K = 1024). Series III main memory is available in 128K, 256K, 384K, 512K, 768K, and 1024K word configurations. Due to the modular design, any system can be easily expanded from one memory size to another. When there are more than 128K words of memory in the Series II system, the memory is divided into two modules; 128K words in the first module and the remainder in the second. When there are more than 512K words of memory in the Series III system, the memory is divided into two modules; 512K words in the first module and the remainder in the second. These modules can operate concurrently, which improves execution time. The word length transmitted over the bus is 17 bits — 16 bits of data (one word or two bytes), and one parity bit. For Series II memory modules, the word length is expanded to 21 bits — 16 bits of data and five bits for automatic fault detection and correction. For Series III memory modules, the word length is expanded to 22 bits — 16 bits of data and six check bits.

Operating power for the memory modules is supplied by rechargeable battery packs in the semiconductor memory power supplies. When the power supply input voltage is removed, battery power is available for a minimum of 40 minutes to maintain memory data.

The memory modules interface with other system modules by way of the Central Data Bus. The other system modules may request transfers of data to or from the memory modules on that bus. Communication between the various modules on the Central Data Bus is controlled by Module Control Units (MCU's), one for each module.

2-9. INPUT/OUTPUT

All user access to input/output devices is by way of the device-independent MPE file system. All location of data, buffering, data transfers, and deblocking are handled automatically by MPE. When you ask to read a named file, you are only implicitly specifying the actual device and/or disc address of the file; the file system determines the explicit device and/or address for you and performs the read. At another level, when you ask the file system for a certain type of device by specifying a device class name (e.g., magnetic tape, line printer, etc.), the file system takes care of allocating an actual device for you. Users who must have actual contact with specific devices may address them directly. Below this simple, flexible interface is a powerful and carefully balanced hardware/software input/output system.

All devices can be operated concurrently (within system bandwidth). Peripherals that fail are taken off-line from the operating system by operator command.

There are two distinct means of implementing I/O: *direct I/O (DIO)* and *programmed I/O (SIO).*

2-10. DIRECT I/O (DIO)

Direct I/O allows for single-word transfers of data, status, or control information between a device connected to the IOP bus and the top of the user's data stack.

2-11. PROGRAMMED I/O (SIO)

Programmed I/O can be used with devices on either the Selector Channel (high-speed devices) or the Multiplexer Channel (medium- to low-speed devices). With programmed I/O, the CPU simply issues a Start I/O (SIO) instruction to the Device Controller. The Device Controller, in turn, initiates the SIO program for the particular device which then runs under the control of the Selector Channel or under the control of the Multiplexer Channel and the IOP. The SIO program uses a unique set of commands (optimized for I/O operations) to transfer information between main memory and the external device. Both the Selector Channel and the Multiplexer Channel (via the IOP) have direct access to main memory. The SIO program and CPU processing run concurrently until the appropriate I/O command terminates the device transfer (this I/O command can also cause an interrupt signal to be sent to the CPU, thus informing the CPU that the I/O task is complete).

2-12. MULTIPLEXER CHANNEL. Each Multiplexer Channel handles up to 16 Device Controllers. The Multiplexer Channel, in conjunction with the IOP, allows the Device Controllers connected to it to run concurrently, interleaving their transfers on a word-by-word basis. By multiplexing, cumulative data rates of up to 1,038,000 bytes per second (inbound) and 952,000 bytes per second (outbound) are possible. Data from the Multiplexer Channel is supplied directly to the IOP for transfer to main memory via the Central Data Bus.

2-13. SELECTOR CHANNEL. Selector Channel data transfers bypass the IOP completely to provide transfer rates of up to 1.9 million bytes per second for a single device. All Selector Channel data transfers are performed in block mode and the data is supplied directly to main memory via the Central Data Bus.

2-14. DEVICE REFERENCE TABLE (DRT)

Device Controllers are identified by a *device number* which is used to access the *Device Reference Table (DRT)*. The DRT is a table known to both hardware and software and it contains, among other things, a pointer to the start of the SIO program for each Device Controller. Since there can be a maximum of 125 entries in this table, the HP 3000 Series II may have up to 125 Device Controllers in its I/O system (the actual limitation is the 7-bit I/O address bus). Certain Device Controllers may control several devices. In such cases, each device attached to the controller is addressed separately using a unit number assigned when the device is installed.

2-15. DATA SERVICE AND INTERRUPT PRIORITIES

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 Controller 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 interrupt or data service priority chains, 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 but a low data service priority.

2-16. INTERRUPT SYSTEM

The interrupt system provides for up to 125 external interrupt levels. When interrupts occur, the microprogrammed interrupt handler 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 enviroment and transfers control to the interrupt routine.

External 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, thus allowing higher priority devices to interrupt lower priority devices.

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

2-17. PERIPHERALS

Mass storage devices consist of moving-head disc drives. These units provide storage capacities from 15 to 120 million bytes and data transfers of nearly one megabyte per second. On-line storage can be greatly expanded by adding additional disc drives to the system.

Low cost magnetic tape units are available in 9-channel models with recording densities of 800 or 1600 bpi.

Card readers are available that operate at 200 or 600 cards per minute.

An HP 2640B CRT Console is supplied as standard equipment for the system console.

Line printers are available with operating speeds of 165, 200, 240, 300, 436, 600, 925, or 1800 lines per minute. The various line printers provide 132-column print lines, using 64, 96, or 128 characters.

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

An interface is available which allows connection of a CalComp Series 500, 600, or 700 (excluding models 745 and 748) Plotter to the system.

Also available are an Asynchronous Terminal Controller and synchronous interfaces for data communications. There may be up to two Asynchronous Terminal Controllers connected to the system, each controlling up to 16 asynchronous terminals.

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

THEORY OF OPERATION

SECTION III

3-1. INTRODUCTION

This section contains a system-level description of the principles of operation for the HP 3000 Series II and III Computer Systems, a block-level and a functional theory of operation for the central processor unit (CPU), and additional CPU information such as architecture, data formats, and instruction formats. Since I/O and interrupts are extensive subjects, they are treated in separate sections. Brief descriptions of the following units are provided:

- Bus system
- Central Processor Unit (CPU)
- Module Control Unit (MCU)
- Typical memory module
- Input/Output Processor (IOP)
- Multiplexer (MUX) Channel
- Selector Channel

In addition, sequences of operations for CPU transfers to and from memory are given.

3-2. SYSTEM-LEVEL DESCRIPTION

A block diagram of a computer system is shown in figure 3-1. The diagram does not show a complete system but a typical system containing a Multiplexer Channel and I/O interface and device (I/O subsystem). Direct I/O devices are not shown on the diagram, nor are additional Multiplexer Channels and their associated I/O subsystems. The following components, therefore, comprise the typical system as depicted:

- Memory subsystem.
- Central processor unit (CPU).
- Input/output processor (IOP).
- Multiplexer channel.
- I/O subsystem (interface PCA and I/O device), such as line printer or card reader.

The memory subsystem contains the I/O drivers that are executed by the CPU, I/O programs that are transferred by the IOP to the Multiplexer Channel, and the Device Reference Tables (DRT). The I/O drivers contain I/O instructions such as Read I/O (RIO), Write I/O (WIO), Start I/O (SIO), Test I/O (TIO), and Control I/O (CIO). The DRT begins in memory location octal 14 and contains a maximum of 125 four-word entries. Each table entry corresponds to a unique I/O device number. The first word of each entry contains the address of the next I/O program instruction for the device.

When the CPU executes an I/O instruction (TIO, CIO RIO, WIO, or SIO), direct commands are sent through the IOP and IOP bus to the addressed I/O subsystem. When the I/O subsystem accepts a direct command, it returns an acknowledge signal and executes the command.

If the CPU executes an SIO instruction, the IOP, in conjunction with the Multiplexer Channel, assumes control of the I/O subsystem and the CPU is free to continue processing other functions. The IOP transfers an I/O program, one instruction at a time, from the memory to the Multiplexer Channel. The Multiplexer Channel then controls the operation of the I/O subsystem.

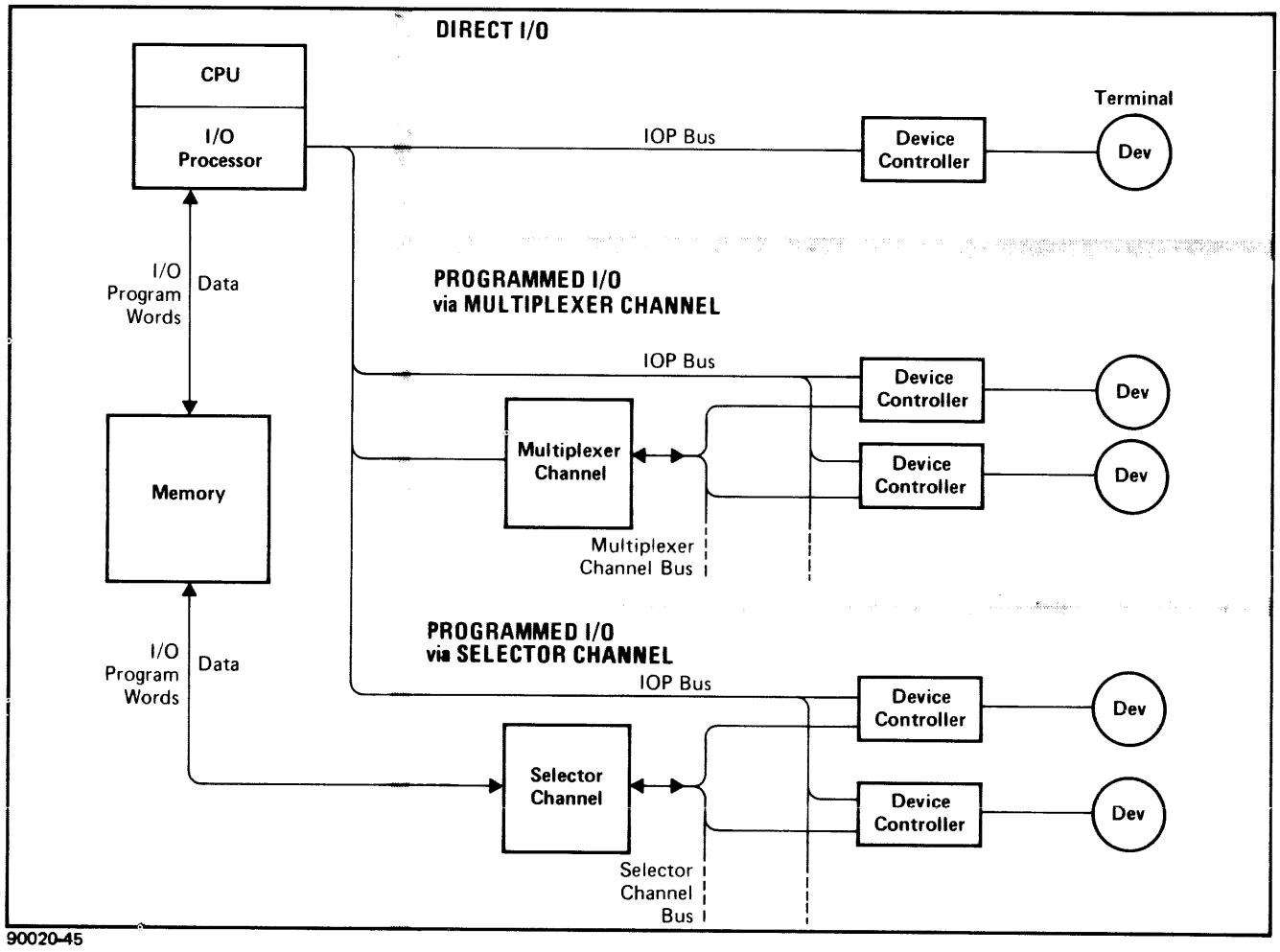


Figure 3-1. Typical Computer System

3-3. CPU GENERAL INFORMATION

3-4. PIPELINE

There are two *pipelines* in the CPU, the *data* pipeline and the *microcode* pipeline. The *microcode* pipeline consists of the V-Bus Mux, ROM, ROR1, and ROR2. The *data* pipeline consists of the store logic, registers, R- and S-Bus logic, arithmetic logic unit (ALU), shifter, and decimal corrector. Because the data pipeline is controlled by the microcode pipeline they will be discussed together.

The general operation of the data circuitry is to pick up two operands from two registers and input them to the ALU where a mathematical calculation can be accomplished. The result is then output on

the T-Bus where the result can either be shifted (shift left 1, shift right 1, or swap bytes with or without clearing either byte) or decimal arithmetic corrected. This final result is then put on the U-Bus and stored in any one of the registers or input to the ALU a second time for further calculations. This entire operation is specified by one microcode instruction which, in effect, takes 175 nanoseconds to execute. To achieve this time, the computer systems use a data and microcode pipeline which have the end effect of one data calculation (stated another way, one microcode instruction) completed every clock cycle (175 nanoseconds).

Consider, for the moment, microcode instructions from consecutive ROM addresses being executed (no microcode jumps or interrupts). To give the data time to propogate through the entire data circuitry, the data is stepped through in two steps. The first step is to read the operands from the two source registers to the input lines for the R- and S-Bus Registers. The logic is given one clock cycle (175 nanoseconds) to accomplish this. The second step is for the data to go through the ALU, shifter or decimal corrector, and store logic and be on the input to the selected store register. The logic is given a second clock cycle to complete this operation.

To cause this to happen the microcode instruction must be executed in two steps. During the first step the instruction is held in ROR1 and effectively the only two fields being decoded during this clock cycle are the *R-* and *S-Bus* fields. These two fields cause the R- and S-Bus logic to select the correct registers for the two operands and gate the operations to the input of the R- and S-Bus registers.

The same clock pulse edge that clocks the operands into the R- and S-Bus registers also:

- Clocks the microcode instruction into ROR2
- Clocks the next microcode instruction into ROR1
- Clocks the final result of the previous microcode instruction into the register specified by its store field.

During the clock cycle period that the microcode instruction is held in ROR2, the *function* field of that instruction is specifying what calculation is to be accomplished by the ALU. The *shift* field of the instruction is specifying what the shifter/decimal corrector circuitry (depending on which of these two circuits is specified by the function field) is to be accomplishing. The *store* field is specifying to the store logic which register to select to gate the final result appearing on the U-Bus. On the next clock edge the now completed microcode instruction will be discarded by loading the next microcode instruction in ROR2. The final result of the instruction that was executed is clocked into the register specified by the store field of that instruction.

During the clock cycle, the instruction that progressed through the microcode pipeline is in ROR2. The R- and S-Bus fields of ROR (which contain the next instruction to be executed) are being decoded and the clock edge that moves this next instruction from ROR1 to ROR2 also clocks the new operands into the R- and S-Bus registers. Thus, it can be seen that on the leading edge of each clock the following operations occur simultaneously:

- A new microcode instruction is clocked into ROR1
- The instruction in ROR1 is clocked into ROR2
- The operands specified by the R- and S-Bus fields of the instruction that has been in ROR1 are clocked into the R- and S-Bus registers.

- The final result of the operands specified by the instruction that has been in ROR2 is clocked into the destination register specified by the store field of that instruction.

This gives the net result of one microcode instruction being executed per clock cycle and thus, one arithmetic calculation accomplished each clock cycle.

Two fields of the microcode instruction that have not been discussed are the *skip* and *special* fields. The *special* field controls the hardware to accomplish such operations as setting condition codes, popping the stack and incrementing and decrementing the SR register. For a full list of operations specified by this field and the results of these operations, refer to the *Microprogram Listings Manual.*

The *skip* field specifies a test condition such as the status of internal flags; the contents of the SR register as compared to zero, two, three, or four; and operand results that appear on the T-Bus as compared to zero, non-zero, odd, and even. For a complete list of test conditions that can be specified by this field refer to the Microprogram Listings Manual. When the condition being tested is true, the microcode instruction at the next ROM address is not executed. If the current instruction is a microcode jump instruction, however, the jump will be executed only if the condition being tested is true. In the case where the next microcode instruction is not to be executed, the skip condition is tested while the microcode instruction is in ROR2. This means that the instruction to be skipped is in ROR1. The clock pulse edge that moves the instruction to be skipped from ROR1 to ROR2 also sets the NOP2 flip-flop. This tells the decode circuitry to cause the ALU to add, the shift field will be forced to a pass function, and the store field will not be decoded. However, the operands specified by the R- and S-Bus fields of the instruction to be skipped were clocked into the R- and S-Bus registers so that the contents of the U-Bus at the end of the NOP clock cycle will be the sum of the contents of the source registers.

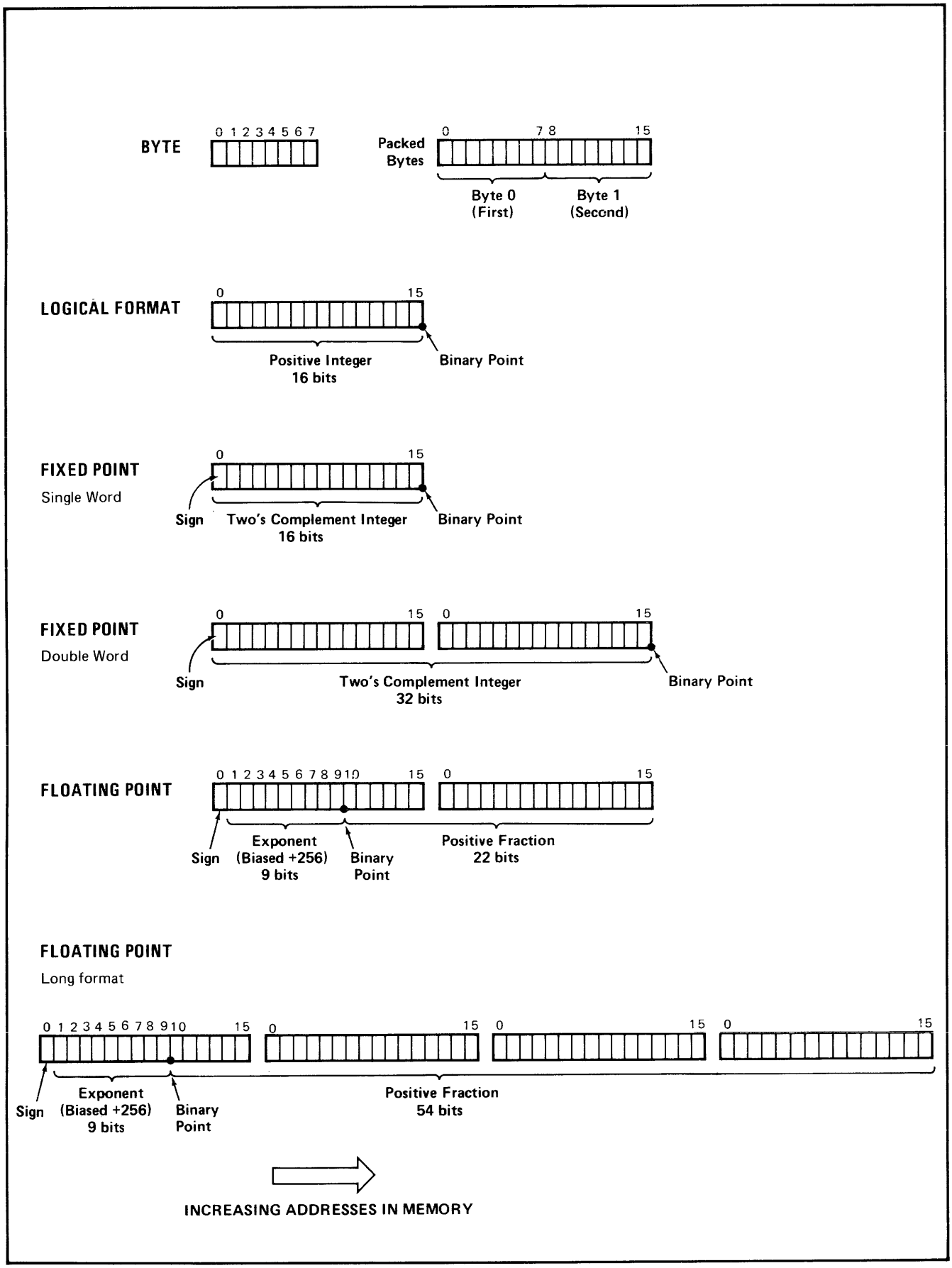
3-5. DATA FORMATS

There are six different data formats (see figure 3-2) that are processable by the computer system's instruction set:

- Byte format
- Logical format
- Single-word, fixed-point format
- Double-word, fixed-point format
- Floating-point format
- Long floating-point format

NOTE

Definitions of the machine instructions referenced in the following paragraphs are contained in the *Machine Instruction Set Reference Manual.*



2152-31

Figure 3-2. Data Formats

3-6. 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 3-2 shows the basic byte format, 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 paragraph 3-48, "Addressing Conventions".) When consecutive bytes are addressed in memory with ascending addresses, the high order byte (bits 0 through 7) of a packed word is accessed first and the low order byte (bits 8 though 15) second.

3-7. 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 from 0 through + 65,535, decimal. The instruction set provides six instructions for logical arithmetic: LCMP, LADD, LSUB, LMPY, LDIV, and NOT. In logical 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. All other respects 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 subtraction 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^{17}. For multiplication (LMPY), overflow cannot occur and the Carry bit has a special meaning (see the definition of the LMPY instruction in the *Machine Instruction Set Reference Manual*). For division (LDIV), the Overflow (instead of 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).

3-8. SINGLE-WORD, 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 multiplication and division, Carry is not used; Overflow indicates that the result cannot be contained in 15 bits plus sign.

3-9. DOUBLE-WORD, FIXED-POINT FORMAT. The double-word, fixed-point format is the same as the single-word format described in the preceding paragraph except that two words are linked together to form a 32-bit double-word 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 eight instructions for double-word integer arithmetic: DCMP, DADD, DSUB, DMUL, DDIV, 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 the definition of the multiply long instruction in the *Machine Instruction Set Reference Manual*). The operands for MPYL and the divisor and quotient for DIVL are single-word.

3-10. 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.

The floating-point format used by the computer system has some special features which are illustrated separately in figure 3-3. 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 3-3 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 1×2^0.

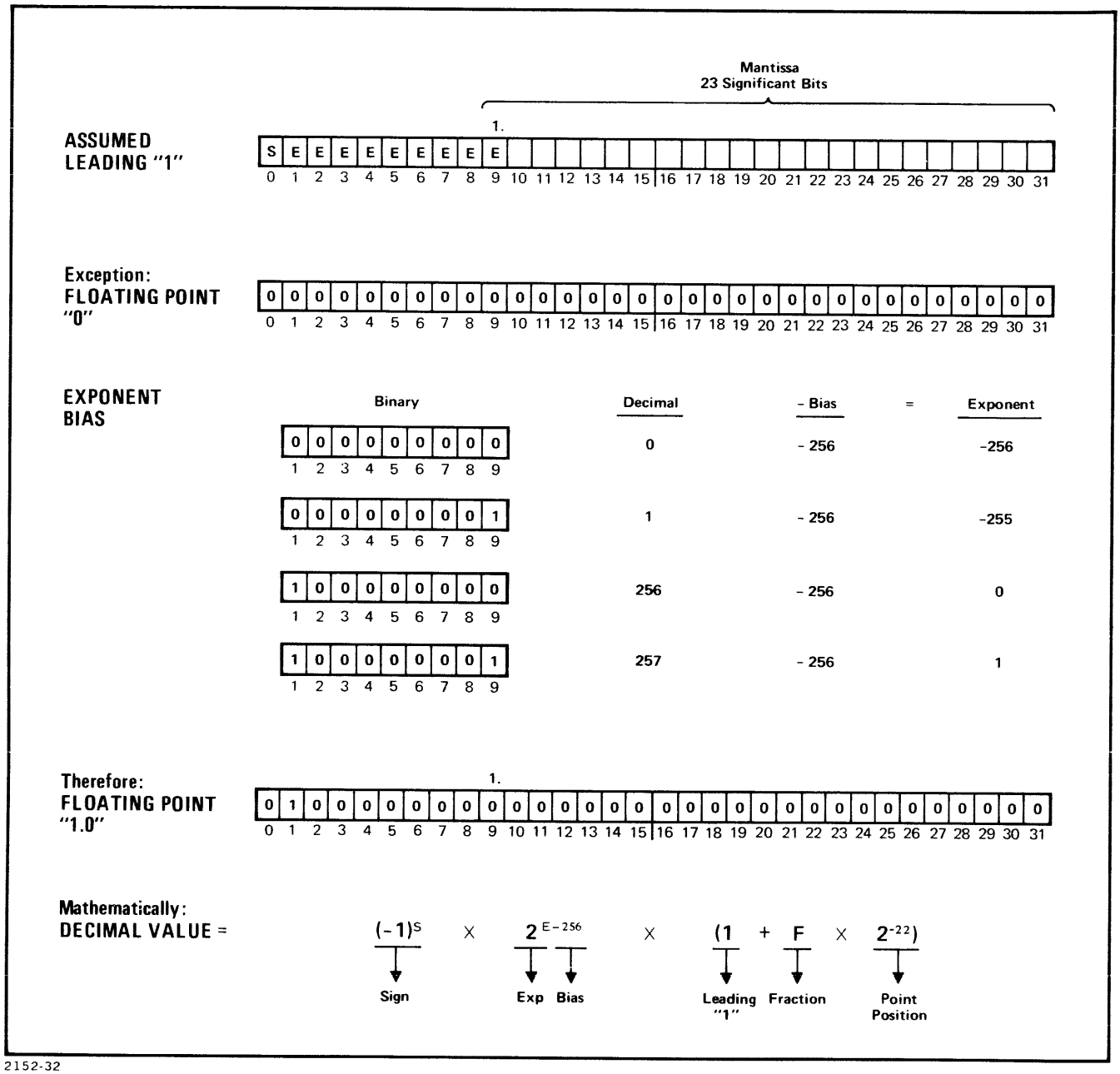
Figure 3-3 also shows the mathematical equation for computing the value of a floating-point number represented by the above conventions. (The exception: zero is defined as: S = E = F = 0.)

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 the definitions of these instructions in the *Machine Instruction Set Reference Manual*).

3-11. LONG FLOATING-POINT FORMAT. The long floating-point format is the same as the standard format described above except that 32 fraction bits are added to the right of the second word. With only this change, the information given in figure 3-3 is also valid for this format. (Note that the "point position" modifier in the equation becomes 2^{-58} instead of 2^{-22}.)

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 double-word or triple-word quantity onto the stack, the least significant word will be on the TOS.



2152-32

Figure 3-3. Floating Point Data Representation

3-12. INSTRUCTION FORMATS

The instruction set has been designed for maximum efficiency of bit usage in the instruction word. Therefore, the instruction formats do not necessarily fall neatly into rigid field boundaries. There are 23 distinct formats used by the instruction set. Refer to the *Machine Instruction Reference Set Manual* for additional information.

In addition to the instruction formats, there are 13 instruction groups (see figure 3-4), as follows:

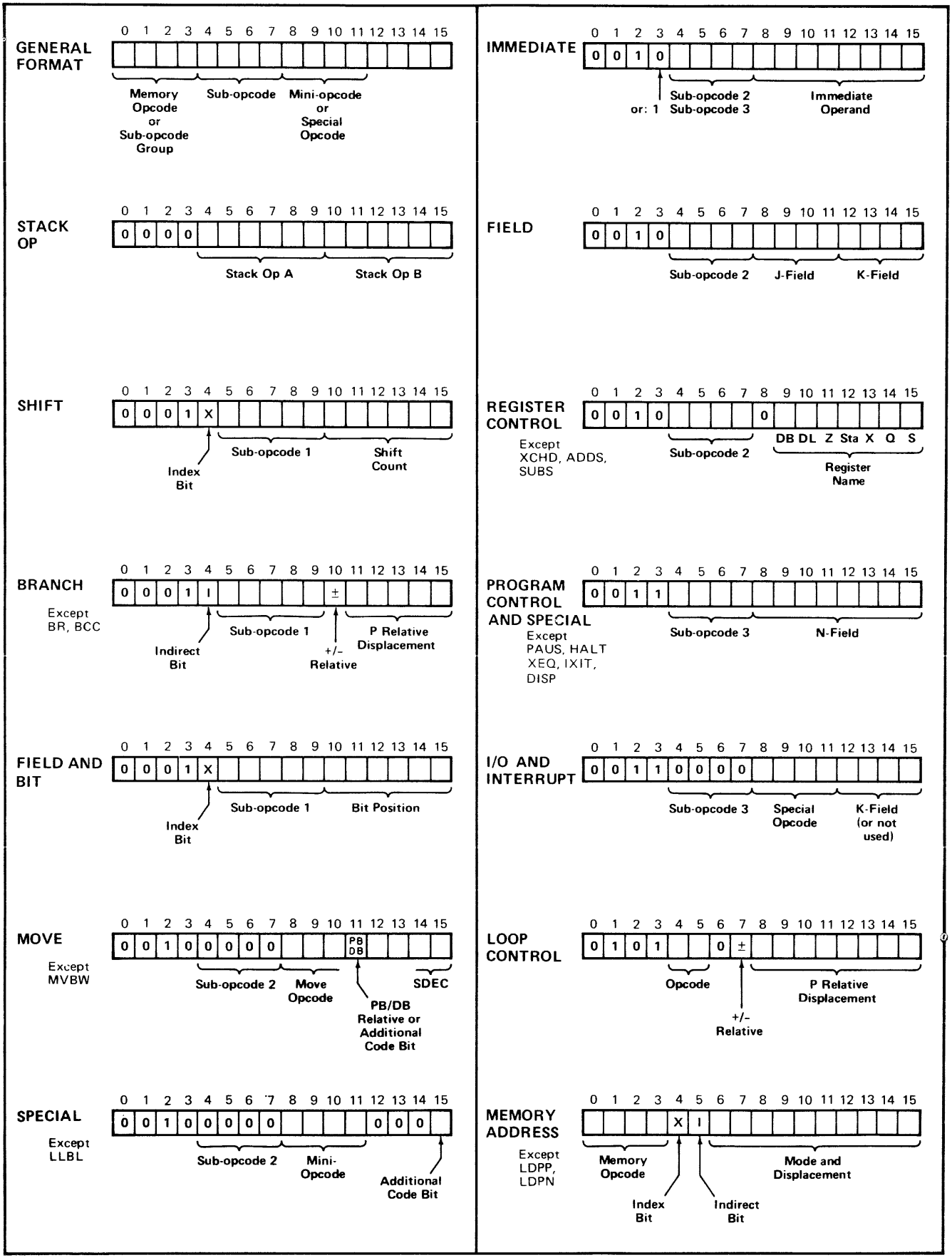
- Stack Op Format
- Shift Format
- Branch Format
- Bit Test Format
- Move Format
- Special Format
- Immediate Format
- Field Format
- Register Control Format
- Program Control Format
- I/O and Interrupt Format
- Loop Control Format
- Memory Address Format

3-13. GENERAL FORMAT. The first format in figure 3-4 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".

3-14. 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.

3-15. 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.

3-16. 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 though 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" = −).



2152-33

Figure 3-4. Instruction Groups

3-17. 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 specified bit position is modified by the addition of the index register contents if the Index bit is set (bit 4 = "1").

3-18. MOVE. The move group of instructions accounts for twelve 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.

3-19. SPECIAL. The special group uses four mini-opcodes. The mini-opcode group is, 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.

3-20. 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.

3-21. 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 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.

3-22. 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.

3-23. 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).

3-24. 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.

3-25. 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.

3-26. MEMORY ADDRESS. The memory address instruction format uses bits 0, 1, 2, and 3 to encode specific instructions. Bits 6 through 15 give both an addressing mode and a displacement. (Refer to paragraph 3-48, "Addressing Conventions".) 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.

3-27. STATUS WORD FORMAT

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 figure 4-11 in Section IV.)

Figure 3-5 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.

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, IXIT, 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.

Bit 2 is used to enable or disable user traps. 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) contains 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 the *Machine Instruction Set Reference Manual.*) 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 1 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 in the *Machine Instruction Set Reference Manual.*

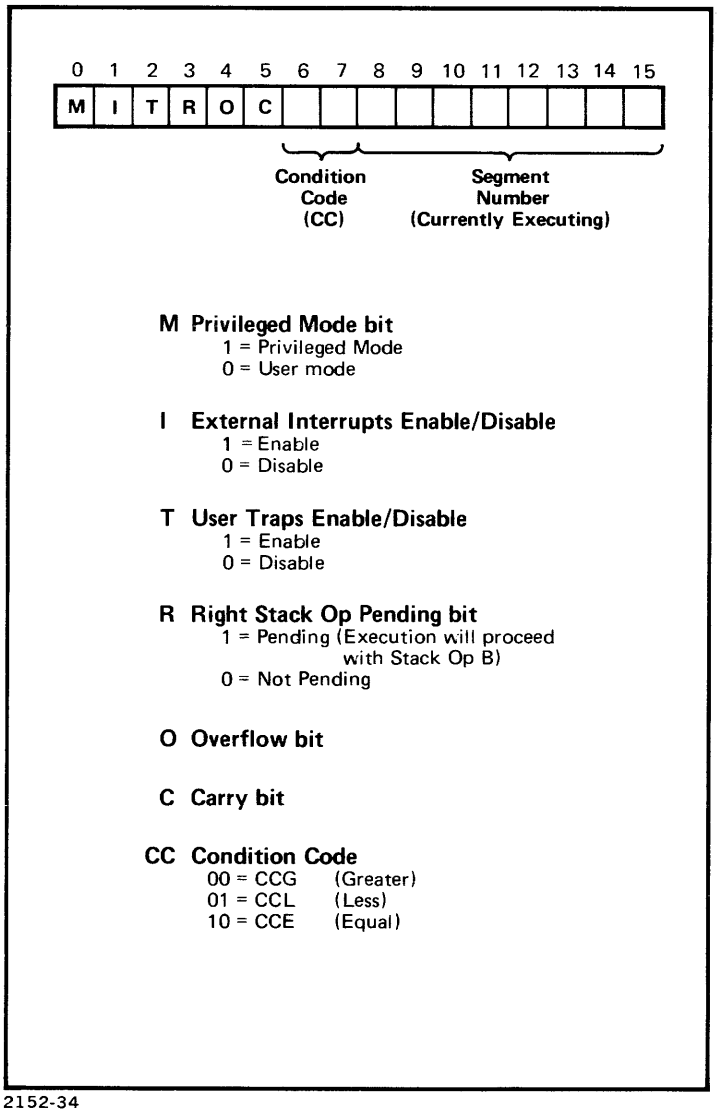


Figure 3-5. Status Word Format

Bits 6 and 7 are used for the Condition Code. Although several instructions make special use of the Condition Code, the Condition Code typically indicates the state of an operand (or a comparison result with two operands). The operand may be a word, byte, double word, or triple word, 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.

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 character (all others) CCE (10) if alphabetic (uppercase 101 - 132, lowercase 141 - 172)
CCC sets CC	=	CCG (00) if operand 1 > 2 CCL (01) if operand 1 < 2 CCE (10) if operand 1 = 2
CCD sets CC	=	CCG (00) if device not ready CCL (01) if non-responding Device Controller CCE (10) if responding Device Controller and/or device ready

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.

Pattern D for the Condition Code, designated as CCD, is used with I/O instructions. (Not all I/O instructions use the condition codes.) The Condition Code is set to 00 if the device is not ready. This condition is usually caused by the device being busy. This code is used only with the instructions which will (WIO and RIO) or could (SIO) require data to be moved. The Condition Code is set to 01 if the Device Controller does not respond. Some examples of what could cause this are power off the device or controller, problems with the device or controller, or waiting for a response to an interrupt request (this would be used with a Controller Processor). The Condition Code is set to 10 if the device or controller responded and the instruction completed properly.

3-28. CPU REGISTERS

Since the computer system's 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 3-6, 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.

3-29. CODE SEGMENT REGISTERS

3-30. 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.

3-31. PB-BANK REGISTER. The PB-bank register is a 2-bit register (4-bit register for Series III) associated with the PB register to define in which bank of 64K words the code segment resides.

3-32. 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 in user or privileged mode.

3-33. 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.

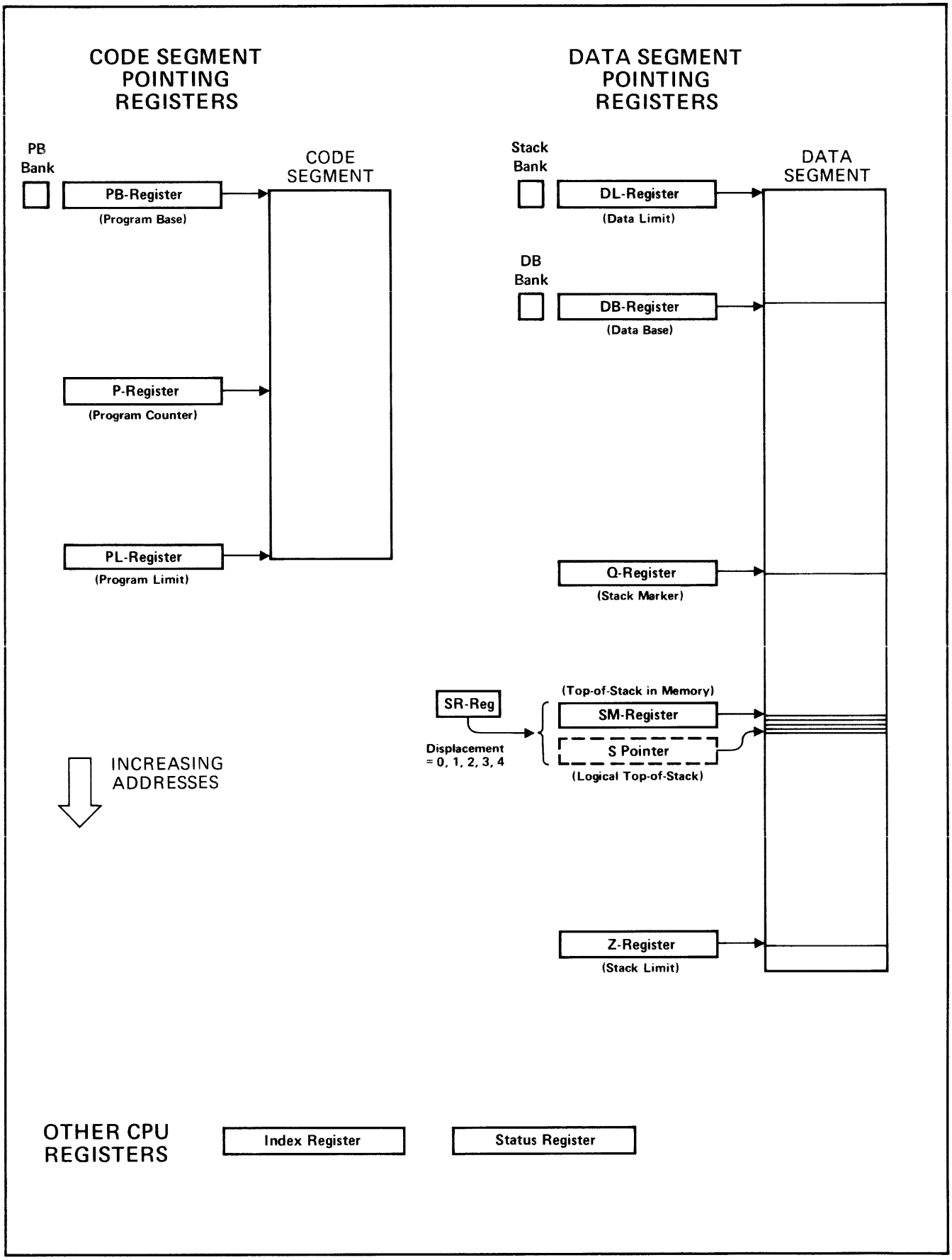
3-34. DATA SEGMENT REGISTERS

3-35. 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.

3-36. 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.

3-37. DB-BANK REGISTER. The DB-bank register is a 2-bit register (4-bit register for Series III) associated with the DB register to define the bank in which the stack, or split stacks, reside. (See paragraph 3-70 for a description of split stacks.)

3-38. 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).



2152-35

Figure 3-6. CPU Registers

3-39. 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).

3-40. 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.

3-41. S POINTER. The S pointer defines the logical top of the stack. The S pointer is not a physical register but rather is logically composed 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.

3-42. 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 several 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.)

3-43. STACK BANK REGISTER. The stack bank register is a 2-bit register (4-bit register for Series III) used with the DL, Q, S, and Z registers to define the 64K word bank in which the stack resides. Due to the split-stack feature, the stack bank is not necessarily equal to the DB-bank. (See paragraph 3-70 for a description of split stack.)

3-44. OTHER CPU REGISTERS

Two CPU registers not associated with code or data segments are the index register and the status register. These are described in the following paragraphs.

3-45. 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.

3-46. 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", paragraph 3-27.)

3-47. PRIVILEGED MODE

The computer systems have the capability of operating in either privileged mode or user mode, and are 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 privileged instructions and to call segments that have been declared *uncallable*. 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. A privileged mode user, however, can use the computer as as if he were the operating system itself.

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.

WARNING

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in the 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 in Section IV.

3-48. ADDRESSING CONVENTIONS

3-49. MEMORY ADDRESSING

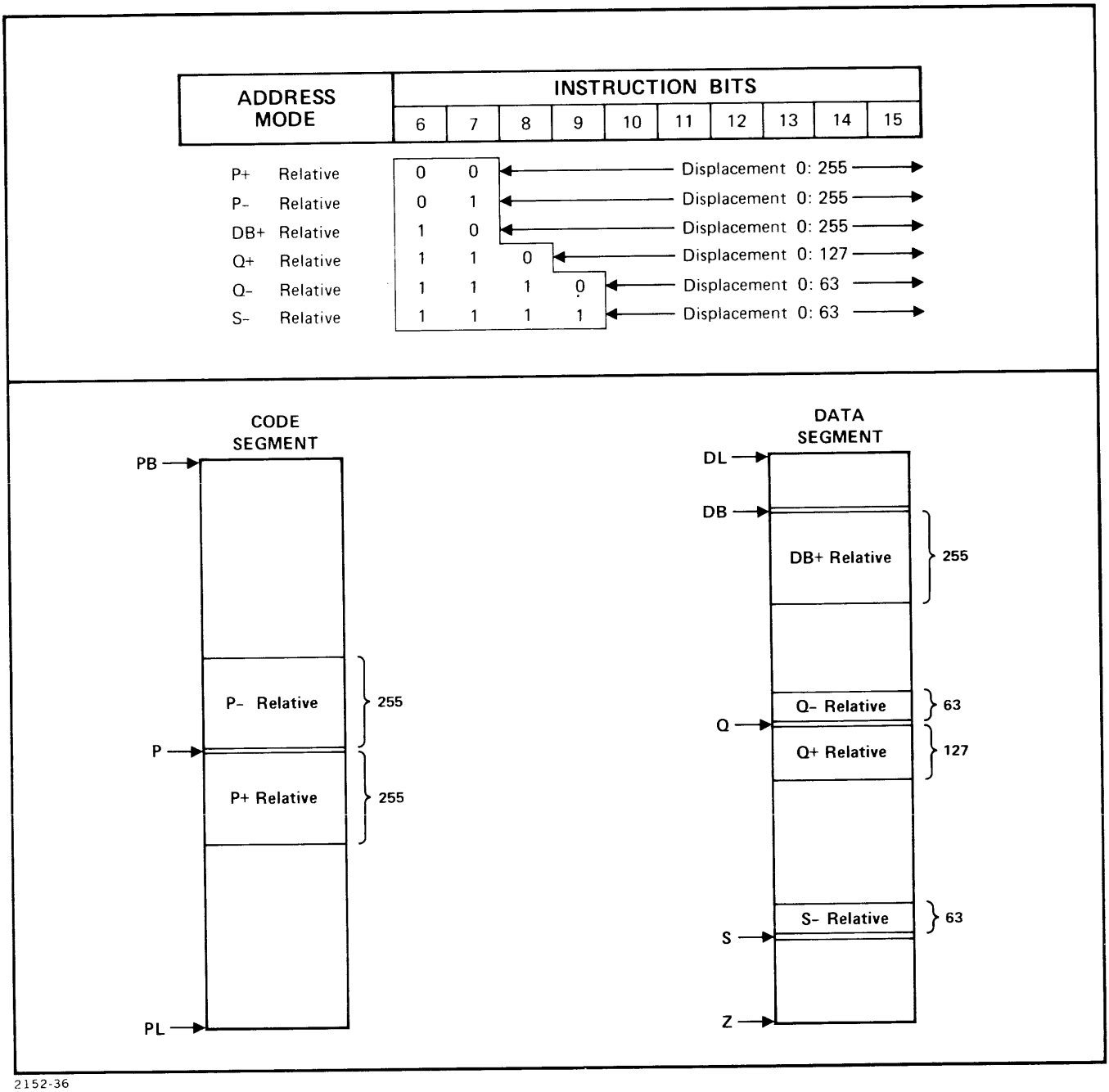
Earlier, in figure 3-4, the format for memory address instructions was shown to employ bits 6 through 15 for "mode and displacement". The following paragraphs explain and illustrate the six memory addressing modes and the respective displacement ranges. Refer to figure 3-7.

The system uses *relative addressing* almost exclusively. (Only privileged instructions, including the I/O group and PLDA, PSTA, MABS, LSEA, SSEA, LDEA, SDEA, 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 3-7, 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."

The ranges of displacement for the various modes of relative addressing also are shown in figure 3-7. (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.



2152-36

Figure 3-7. Memory Addressing Modes

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 taken care of automatically 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 the *Machine Instruction Set Reference Manual* 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, instructions 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 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).

3-50. INDIRECT ADDRESSING

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 3-8.

For memory address instructions, indirect addressing is specified by bit 5 of the instruction word ("1" indicates indirect to be used). For the branch instructions (excluding BR), indirect addressing is specified by bit 4. See figure 3-4.

3-51. CODE INDIRECT. Figure 3-8 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.

3-52. DATA INDIRECT. The first of the three examples in figure 3-8 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.

3-53. 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 3-9 shows some examples of indexing. Unlike figure 3-8, this figure does not illustrate all combinations of cases. Figure 3-9 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.

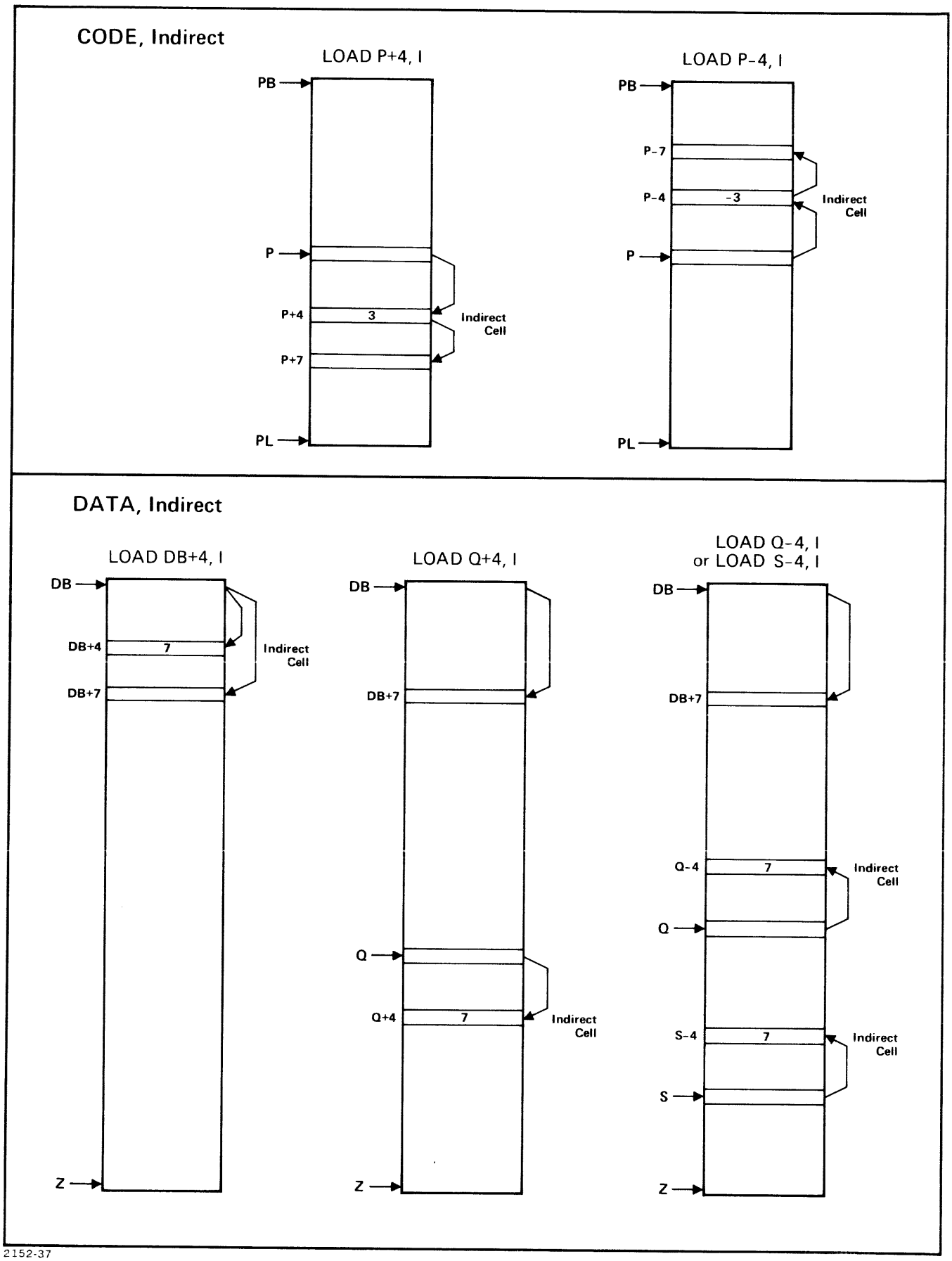
3-54. CODE INDEXING. The first example in figure 3-9 shows the actions occurring for an assumed instruction "LOAD P+ 4, X". The displacement, + 4, would by itself point to location P+ 4; however, by adding the index of 5 to the displacement, the location P+ 11 (octal) is addressed. It is the content 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).

3-55. 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.



2152-37

Figure 3-8. Examples of Indirect Addressing

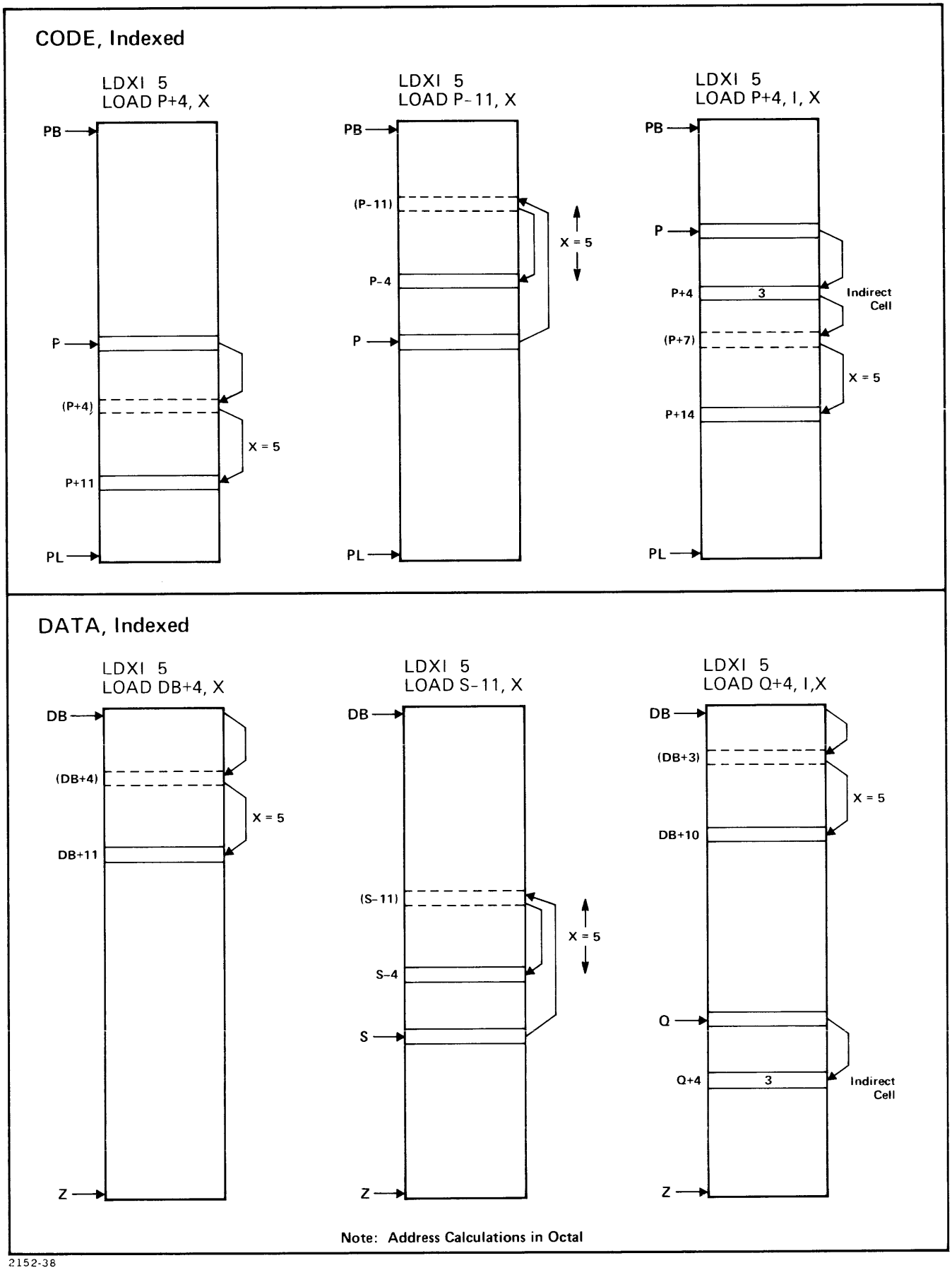


Figure 3-9. Examples of Indexing

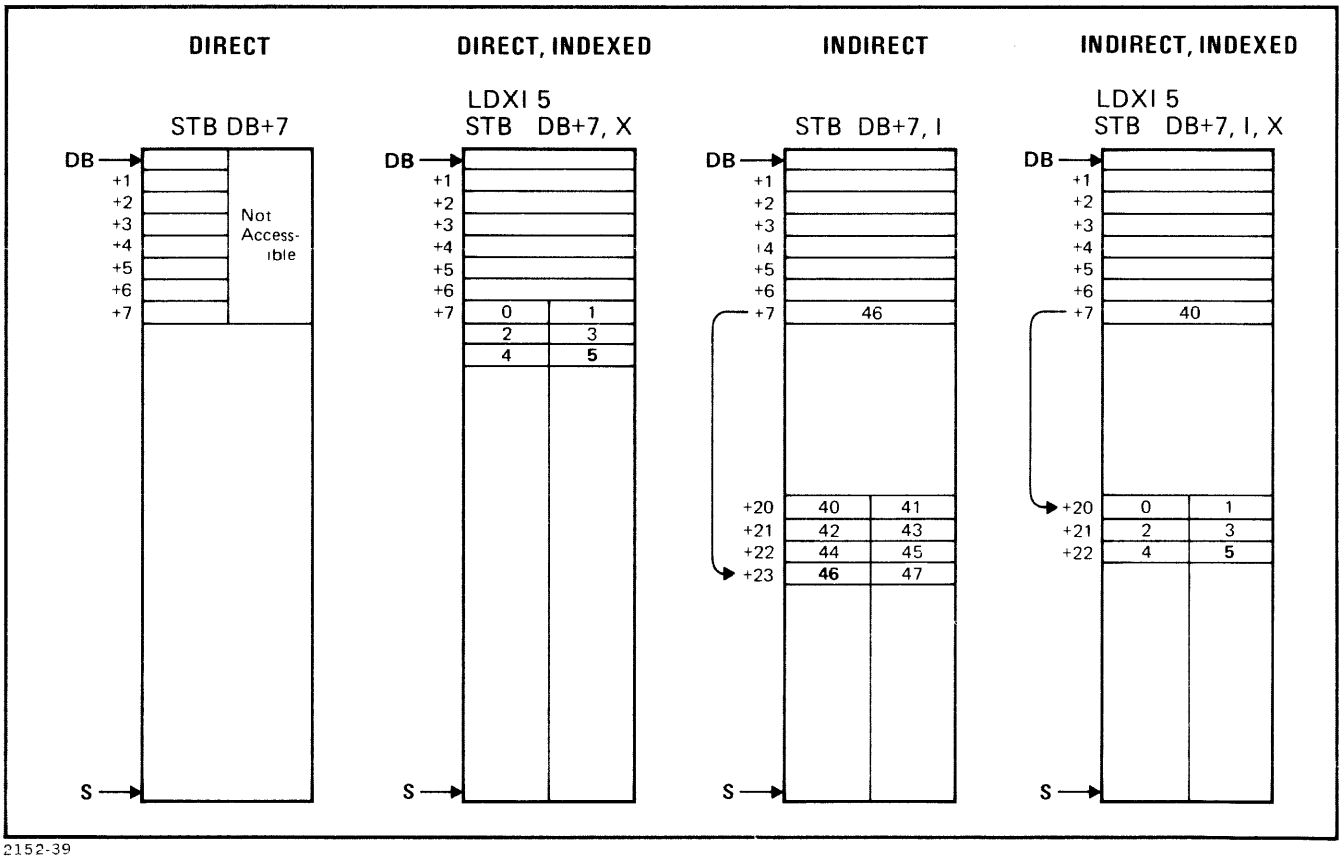
The final example illustrates data indexing combined with indirect addressing. Again, post-indexing is applied. The example instruction is "LOAD Q+ 4, I, X". The displacement, + 4, points to the indirect cell at Q+ 4, which contains the value 3. Since indirect addresses for data are always DB+ relative, 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).

3-56. 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 central processor unit 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 3-10 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.



2152-39

Figure 3-10. Examples of Byte Addressing

3-57. 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 3-10, a "STB DB+ 7" instruction would store a byte from the TOS into the left byte of the DB+ 7 location.

3-58. 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.)

3-59. 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 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.)

3-60. 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 3-10 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 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 paragraph 3-68, "Access to DB- Area".

3-61. DOUBLE-WORD INDEXING

Two memory address type instructions, LDD and STD, permit double-word 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 double word in a double-word array.

3-62. BOUNDS CHECKING

The central processor unit 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 the *Machine Instruction Set Reference Manual* define the checks that are made; however the lack of such mention does not necessarily imply that no checks are made. (Sufficient checks are made so that a non-privileged user cannot adversely affect other users or the operating system.)

The following paragraphs summarize the basic bounds checks that occur for the applicable instruction types. Refer to table 3-2 and figure 3-11.

3-63. 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 also applies when branching indirect via the stack, 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 1.

3-64. 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; 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 1.

3-65. 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 1.

3-66. 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 128 locations beyond Z which could be filled with stack related data. A stack overflow causes an interrupt to segment 1, which, under the discretion of the operating system, may extend the stack limit.

3-67. 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 1. Users can access the area between DL and DB by indirect addressing or indexing, as long as SM does not become less than DB. Although the hardware does address arithmetic modulo 64K, code segments and data stack may not cross memory bank boundaries. This restriction is handled by the operating system.

Table 3-2. Bounds Checks

CHECK	DEFINITION	MODE
Program Transfer	$PB \leqslant E \leqslant PL$	Privileged, User
Program References	$PB \leqslant E \leqslant PL$	User only (except moves)
Data References	$DL \leqslant E \leqslant S$	User only
Stack Overflow	$SM > Z$	Privileged, User
Stack Underflow	$SM < DB$	User only
E = Effective Address of Memory Reference		

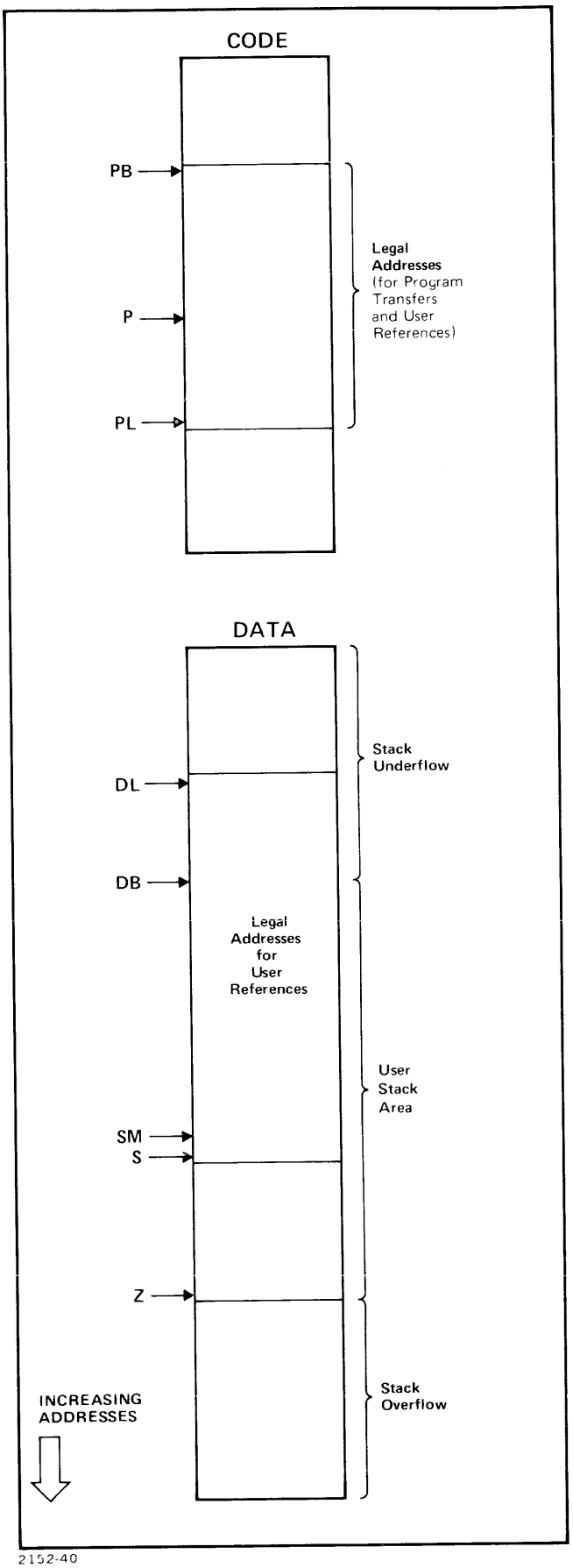


Figure 3-11. Addressing and Stack Bounds

3-68. ACCESS TO DB- AREA

Both privileged and user modes have access to the data area between DB and DL through indirect addressing and indexing. The privileged mode additionally has direct access by the privileged move instructions MVBL and MVLB. Figure 3-12 illustrates the technique of indirect addressing to access this area, using both word and byte examples.

3-69. WORD ADDRESSING. The left part of figure 3-12 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.)

3-70. BYTE ADDRESSING. The right part of figure 3-12 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),

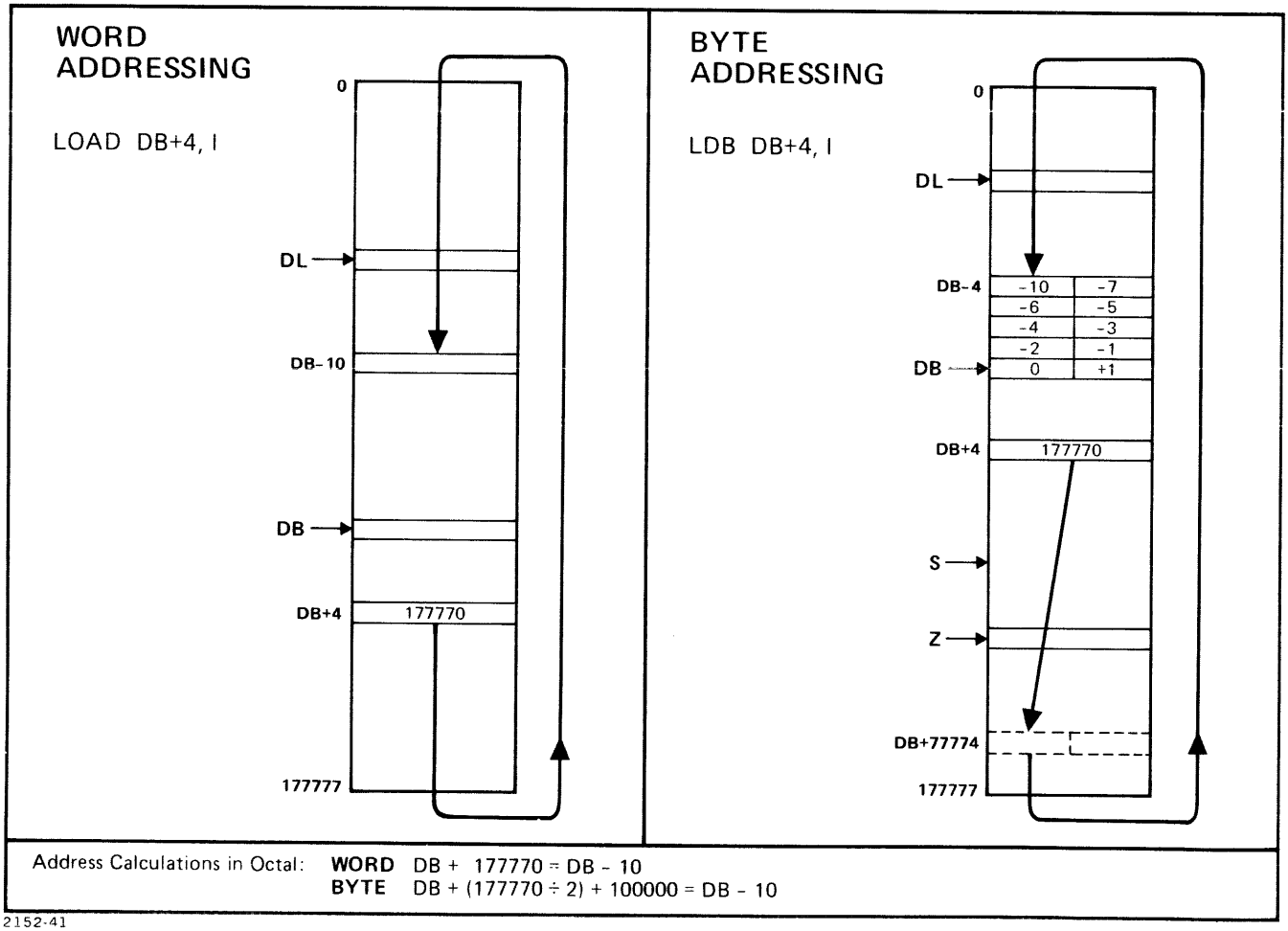


Figure 3-12. Access to DB- Area

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 1. In privileged mode, the result of the second test is ignored; execution continues even if out of bounds, using the second referenced byte.

3-71. BLOCK-LEVEL DESCRIPTION

The block-level description divides the computer system into groups and describes the operation of each group. The groups are:

- Bus System
- Central Processor Unit (CPU)
- Memory Module
- Device Controller
- Module Control Unit (MCU)
- Port Controller
- Input/Output Processor (IOP)
- Multiplexer Channel (MUX)
- Selector Channel (SC)

3-72. 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. Figure 3-13 represents a system that consists of a CPU/IOP Module, two Memory Modules, and a Port Controller. While only two Multiplexer Channels are shown, there may be any practical number. Each Multiplexer Channel can accommodate up to 16 Device Controllers. The Port Controller is shown with a maximum configuration of two Selector Channels. Each Selector Channel can accommodate up to eight High Speed Device Controllers.

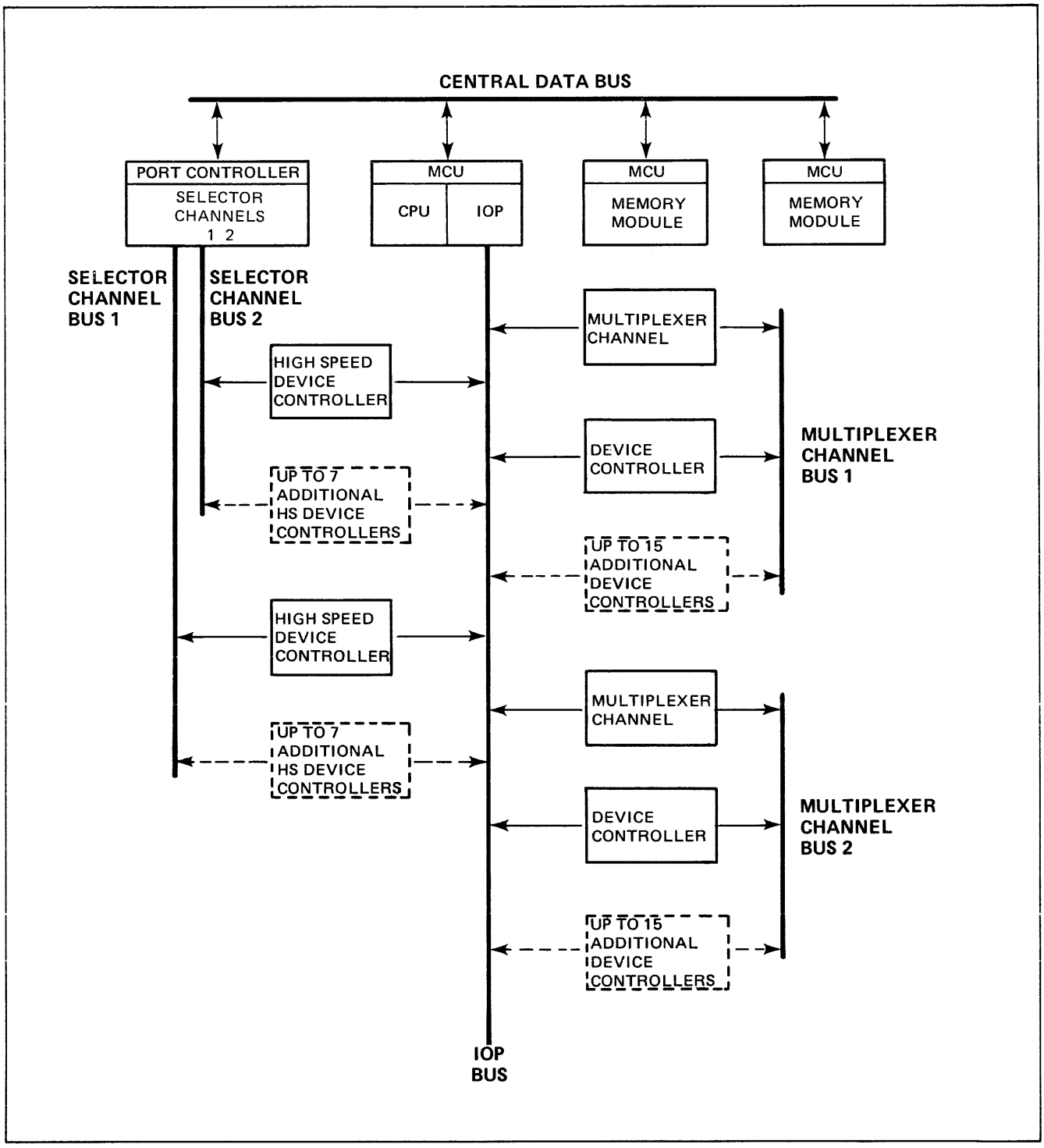


Figure 3-13. Bus System Block Diagram

3-73. 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 each Module Control Unit (MCU) and each Port Controller in the system.

3-74. 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.

3-75. 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 each 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 both channels simultaneously, on a word-by-word basis.

3-76. MULTIPLEXER CHANNEL BUS. With a few minor differences in signal nomenclatures, the Multiplexer Channel bus (also called the MUX channel bus) is virtually identical to the Selector Channel bus. This allows certain devices, 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 Selector Channel bus are used as service request lines on the Multiplexer Channel bus.)

3-77. POWER BUS. The power bus (not shown), unlike the flat cable signal buses discussed above, is a rigid printed circuit board. Terminal strips on the right side of each board accept the power wires from the power supply, which is mounted to the rear of the cabinet. However, some I/O bus lines and clocks are also routed along the power bus.

3-78. OVERVIEW

Operation of the Central Processor Unit (CPU) is controlled by the software set of instructions and the microprogram.

The CPU requests an instruction from memory via the Module Control Unit (MCU). The instruction is received over the central data bus and is loaded into the next instruction register (NIR). The instruction is clocked into the current instruction register (CIR) and then into the CMUX. If the pipeline has not been filled (see "Pipeline", paragraph 3-4), the NIR output not only goes into CIR, but also goes direct to the CMUX, thus saving one clock pulse.

Ten bits of the CMUX output go to the instruction mapper and 8 bits go to the mapper control. The 8-bit output of the mapper goes to the look up table (LUT) read-only memory (ROM). The LUT ROM produces a 12-bit microprogram starting address from the instruction received from memory and also the following 8 special use bits: SRP0, SRP1, and SRP2 to the SR preadjust adder; Z, PC0, PC1, and W to the preadder control (W also goes to the mapper control); and the JULI bit to the BMUX and CMUX control circuits.

The 12-bits of address from the LUT are fed to the VBUS MUX. The output of the VBUS MUX is 16-bits to the ROM and increment circuit. The 16-bits to the ROM is the starting address for the microcode instruction provided no special conditions, such as stack pre-adjustment, are needed. The 32-bit ROM output is clocked into ROR1.

At the same time that the ROM is being accessed, the address is sent to the increment circuit. On the same clock pulse that clocks the ROM output to ROR1, the address-plus-one is put into the address register (RAR). The output of RAR goes back to the VBUS MUX. When, on the next clock cycle, the incremental address goes to ROM, the new microcode instruction goes to ROR1, and the original microcode instruction goes to ROR2. The pipeline is packed, functioning, and incrementing one step at a time through the microcode.

3-79. UNCONDITIONAL JUMP. An unconditional jump is a jump that occurs without regard to the data. If the microcode calls for an unconditional jump, a jump target address is selected out of the shift, special, and R-bus fields of the microcode instruction in ROR1 (ROR2 if previous microcode line contained a data-dependent skip condition) and fed back to the VBUS MUX so that the new ROM microcode instruction is sent to ROR1. The target address goes to the increment circuit, is incremented by one, and the new target address + 1 is stored in RAR awaiting the next clock cycle when it can be clocked into the VBUS MUX for consecutive addressing of the microcode instructions.

3-80. CONDITIONAL JUMP. The microcode instruction calling for a jump comes out of the ROM and into ROR1 which decodes the R- and S-fields. The R- and S-field information is sent through the R- and S-bus logic and is waiting at the inputs of the R- and S-bus registers. On the next clock cycle, the jump instruction goes to ROR2 and the R- and S-field data is clocked through the R- and S-bus registers. The T-bus data is loaded from ROR2 to feed the ROM so that on the next clock cycle the address of the jump-to microcode instruction goes to ROM. As the new instruction goes to ROR2, the jump-to address + 1 goes into RAR and the operation resumes stepping through the microcode from there.

3-81. CENTRAL PROCESSOR UNIT

The central processor module determines the basic characteristics of the system hardware. The central processor module is divided into three major sections, Central Processor Unit (CPU), Input/Output Processor (IOP), and Module Control Unit (MCU). The MCU is shared by the CPU and IOP.

The CPU portion of the CPU/IOP module logically consists of three sections (see figure 7-2 in Section VII).

- Microprocessor
- Processor Registers
- Arithmetic Logic Unit (ALU)

The microprocessor receives an instruction word from memory and translates it into a microprogram starting address. The microprogram is then read out of ROM and is decoded into a set sequence of control signals. The processor registers are 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 unit executes various functions (add, subtract, etc.) on the R- and S-bus inputs, with or without a shift, and outputs the result to either of the CPU registers (for transmission out of the module) or to the U-bus (for storage in one of the internal registers).

A set of simplified logic diagrams for the HP 3000 Series II and III Computer Systems is contained in Section VII (see figures 7-2 through 7-5 and 7-9 through 7-14).

3-82. NEXT INSTRUCTION REGISTER. The next instruction register (NIR) is a 16-bit register (see figure 7-2). The NIR register is loaded with an instruction from memory. The register provides storage for the instruction to be executed immediately following the current instruction. This allows an instruction to be fetched concurrently with the execution of the current instruction. The NIR register is loaded by an NIP signal from the MCU operation decoder. The NIP signal is generated as a result of a skip field code NEXT as described in the skip field code definitions (see table 3-6) or the MCU option NIR as described in the MCU option field code definitions (see table 3-9).

3-83. CURRENT INSTRUCTION REGISTER. The current instruction register (CIR) is a 16-bit register containing the instruction currently being executed by the CPU. The CIR is loaded from the NIR by a NIRTOCIR signal from the next logic. The NIRTOCIR signal is generated as a result of a skip field code NEXT as described in the skip field code definitions (see table 3-6) or the cycle after a special field code CCPX (see table 3-8) with U-bus bit 0 being a logic 1.

If the pipeline has not been filled (see "Pipeline", paragraph 3-4), the NIR not only goes into CIR, but also goes direct to the CMUX. One clock pulse is saved. Two instruction registers, NIR and CIR, allow one clock pulse to be memory fetching while the other is still executing an instruction. Instruction translation is accomplished from CIR two clock cycles after the execution has begun until execution is completed unless it is the right instruction of a stack-op (interrupting between stack-op instructions not being considered). In the case of a Right Stack-Op instruction, the entire translation is accomplished from CIR. The controlling factor concerning the execution of the Right Stack-Op instruction is the BMUX control circuitry.

3-84. CMUX AND CMUX CONTROL. The next control circuitry along with the CMUX control circuitry controls the CMUX multiplexer to specify whether the contents of NIR or CIR go into the mapper circuitry.

3-85. MAPPER AND MAPPER CONTROL. The mapper circuitry is fed by the outputs of the CMUX control and the mapper control circuitry. The resulting 8-bit output of the mapper circuitry addresses a specific location in the look up table (LUT) ROM.

3-86. LOOK UP TABLE ROM. The look up table ROM (LUT ROM) outputs a 12-bit address and 8 control bits. The 12-bit address is then fed to the V-bus MUX. The 16-bit output of the V-bus MUX addresses the initial microcode that will start the accomplishment of the CIR instruction. The 8 control bits (SRP0, SRP1, SRP2, Z, PC0, PC1, W, and JLUI) specify the mode of addressing being utilized for memory reference instructions. The SRP0, SRP1, and SRP2 bits define how many hardware top-of-stack registers must be valid before execution of the instruction can actually begin.

Data bit 0 in the look up table ROM is the W-bit (bits 1 through 11 of the LUT contain the starting address of the microprogram for the instruction to be executed). When a new instruction is to be executed, the W-bit is read out of the LUT and stored in the W-bit register. The W-bit has different meanings for different instructions (the bit has a fixed, known value for every instruction).

a. For STACKOPS (CIR (0:3) = %00), the W-bit has no meaning, it is set equal to logic 1 merely for convenience.

b. For SUBOP 1 (CIR (0:3) = %01) instructions:

(1) The W-bit is set to logic 1 for instructions regarding P-relative addresses (some branches). In this case, CIR (10) is treated as a sign bit for the P-relative displacement in CIR (11:15). This bit controls the function of the pre-adder (add or subtract) such that a positive number or a negative number may be obtained from it.

(2) The W-bit is set equal to logic 0 for shift-type instructions. In this case, the pre-added output is CIR (10:15), a 6-bit shift count, with zeros in all other bit positions.

c. For SUBOP 2 (CIR (0:3) = %02) instructions, the W-bit controls the function of the pre-adder. In all cases, the input to the pre-adder is CIR (8:15). When the W-bit is logic 0 the pre-adder is set to the ADD function. Since the second input to the pre-adder is logic 0 (no indexing), the output is -CIR (8:15) (=317 - CIR (8:15)), a negative number.

d. For SUBOP 3 (CIR (0:3) = %03) instructions there are two cases:

(1) For SPECOP 00 (CIR (0:3) = %03), the W-bit is set to logic 0. This forces the pre-adder to the ADD function. In addition, CIR (12:15) only is applied to the pre-adder input. The output, therefore, is the K-field, CIR (12:15).

(2) For SPECOP 01 through 17 (CIR (4:7) = %01 - %17), the W-bit causes the same action as in SUBOP 2, explained in paragraph c, above.

e. SUBPO %04 through %17 (CIR (0:3) %04 - %17) instructions, in general, reference an operand in memory. The operations necessary to obtain the effective address of this operand are common to most of the instructions, and therefore one microprogram is used for this calculation. When one of these instructions is to be executed, it maps (through the LUT) to this microprogram to obtain the operand address. When this is done, the instruction then jumps to the microprogram that executes the specific instruction called for and the W-bit now becomes effective. The W-bit is set to logic 1. When the foregoing address calculation routine has been completed, a micro-operation (JLUI) in the ROM skip field is executed. If the instruction does not specify indirect addressing (or if one level of indirect addressing has been completed), the execution of JLUI forces a microprogram jump to an address contained in the LUT. Since the contents of CIR have not changed, the LUT would normally still be pointing to the address of the foregoing address calculation routine, and an infinite loop would result. The W-bit, however, now modifies the LUT entry address to a different (but related) address. This LUT address contains the microprogram address of the desired instruction to be executed.

3-87. V-BUS MUX AND V-BUS CONTROL. One of 9 inputs to the V-Bus MUX is selected by the V-Bus Control to be fed through the V-Bus MUX to become a 16-bit address to the ROM. The 16-bit address is also fed to an incrementing circuit which increments the address by 1 and feeds the output to the ROM address register (RAR).

3-88. ROM ADDRESS REGISTER. The ROM address register (RAR) is a 16-bit register which holds the address of the next microinstruction to be executed if no preempting conditions (Interrupt, Jump, etc.) occur. The RAR is loaded with the ROM address incremented by 1. After loading, it is automatically incremented every 175 nanoseconds by the increment logic until reaching the end of the ROM microprogram for that instruction. The RAR input is normally loaded from the incremented ROM address unless a repeat is specified, at which time the contents of RAR do not change until the repeat is terminated.

In addition to the 12-bit output from the LUT ROM, RAR can be loaded from the ROM output register rank 2 (ROR2), by a JMPGATE signal generated in response to a function field code Jump (JMP) or Jump To Subroutine (JSB), by the interrupt logic due to an interrupt or power failure, or from the U-bus in response to a RAR store specified, or from the Hardware Maintenance Panel.

3-89. SAVE REGISTER. When a Jump to Subroutine (JSB) is decoded by the function field decoder (see table 3-5), a JSB1 signal is generated and the contents of the RAR are loaded into the save register until a Return from Subroutine (RSB) is decoded by the skip field decoder. The RSB signal loads the contents of the save register back into the V-bus MUX and from there into the ROM which continues executing the microprogram with the microinstruction following the JSB.

3-90. READ-ONLY MEMORY. The read-only memory (ROM) accepts 16-bit addresses from the V-bus MUX and outputs 32-bit microinstructions of a microprogram to the ROM output registers. The ROM contains 4096 (7777 octal), 32-bit microinstruction words. Each instruction generally calls several microinstructions from the ROM. For example, instructions which affect the top of stack (TOS) 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 one or more memory transfers to adjust the stack, remaining microinstructions called by the instruction will begin. Updated addresses for succeeding microinstructions called by the instruction are furnished to the ROM every 175 nanoseconds by the RAR.

3-91. ROM OUTPUT REGISTERS. There are two ROM output registers, ROR1 and ROR2. The 32-bit output from ROM is loaded into ROR1 on each clock cycle (175 nanoseconds). On the next clock cycle, six 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. Two ROM output registers allow the S- and R- fields to be decoded in advance of the rest of the word. Thus S- and R-bus selection will have occurred in ROR1, and the selected data will be ready and waiting on the U-bus by the time the rest of the word is decoded from ROR2. Each field of the ROM output word is separately decoded (see tables 3-3 through 3-10). The *S-bus field* selects one of 31 registers (or sets of lines) to be loaded into the S-bus register. The *store field* selects one of 29 registers in which to store the U-bus data. 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 and CTL bus request signals. The *R-bus field* selects one of 15 processor registers (or sets of lines) for loading into the R-bus register.

3-92. MICROCODE JUMPS. Microcode jumps can be taken from either ROR1 or ROR2. The jumps will be taken from ROR1 only under the condition that the jump has a skip code of unconditional and the instruction in ROR2 meets one or more of the following conditions:

- Is cancelled by NOP2.
- Is a ROM Immediate type of instruction without a data-dependent skip.
- Contains a NOP skip function.
- Contains a non-data-dependent skip test (skip codes 14 - 27, 32 - 34) which is not met or if an ROR1 jump has just been completed.

All other microcode jumps will be executed from ROR2.

An unconditional jump is a jump that occurs without regard to the data. (See paragraph 3-79.)

Jumps executed from ROR2 because none of the fast-jump (see above) conditions were present, and conditional jumps which are always executed from ROR2 behave as follows:

- NOT TAKEN — next line in sequence executed on next clock.
- NON-DATA-DEPENDENT TAKEN — one overhead clock required (NOP2 effective) before target line executed.
- DATA-DEPENDENT TAKEN — two overhead clocks required (FREEZE, NOP2) before target line executed.

Execution of jumps in ROR2 inhibit any fast jumps from ROR1 being executed. Hence, if there are two consecutive lines of microcode containing jumps, the jump in ROR2 will be taken and the jump in ROR1 will be ignored. (See paragraph 3-80.)

The 32-bit microinstruction word from the ROM output registers is divided into seven fields, each field containing from three to five bits. Each field, when decoded, produces a set of microcode signals which control the operation of the CPU.

3-93. S-BUS FIELD DECODER. The S-bus field decoder (bits 0:4) selects one of 32 registers (or sets of lines) to be loaded into the S-bus register. S-bus field code definitions are shown in table 3-3.

3-94. STORE FIELD DECODER. The U-bus store decoder (bits 5:9) selects one of the registers or other destination for the U-bus data. Store field code definitions are shown in table 3-4.

3-95. FUNCTION FIELD DECODER. The function field decoder (bits 10:14) specifies the function to be performed by the arithmetic logic unit (ALU) on the two operands in the R- and S-bus registers. Function field code definitions are shown in table 3-5.

3-96. SKIP FIELD DECODER. The skip field decoder (bits 15:19) determines what condition will 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.), ROM output register 2 (ROR2) will execute a no-operation (NOP), effectively skipping one microinstruction word. The skip field also specifies the condition under which a Jump (JMP) or Jump to Subroutine (JSB) will be executed if coded in the microinstruction. Other signals, such as NEXT which calls the next instruction from memory, are also decoded from the skip field. Skip field code definitions are shown in table 3-6.

3-97. SHIFT FIELD DECODER. The shift field decoder (bits 20:22) specifies how the T-bus data will be shifted (right one, left one, straight through, etc.). In addition, the shift field generates the scratch pad 1 register and scratch pad 3 register shift signals in conjunction with the function field. The shift field code definitions are shown in table 3-7.

3-98. SPECIAL FIELD DECODER. The special field decoder (bits 23:27) has varied uses such as generating the memory operation code signals and the POP signal, which moves the stack elements up one location such that the second element (S minus 1) becomes the top of stack. The special field code definitions are shown in table 3-8.

Table 3-3. S-Bus Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
(blank)	11111	The S-bus register is loaded with all zeros.
CC (Condition Code)	10111	The CC S-bus field code is used to retrieve the condition code (CC) portion of the status word for use with certain conditional branch instructions. When executed, bits 6 and 7 of the status word are loaded into bits 8 and 9 of the S-bus register and if both of these bits are zeros, S-bus register bit 7 becomes a one. All other S-bus register bits become zeros.
CIR (Current Instruction Register)	00000	The 16-bit output of the current instruction register (CIR) is loaded into the S-bus register.
CPX1	00100	CPX1, a collection of 16 special signals, is loaded into the S-bus register.
CPX2	00110	CPX2, a collection of 16 special signals, is loaded into the S-bus register.
CTRH (Counter High)	01101	The 6-bit content of the counter (CNTR) register is loaded into bits 4 thru 9 of the S-bus register. All other S-bus register bits become zeros.
CTRL (Counter Low)	01100	The 6-bit content of the counter (CNTR) register is loaded into bits 10 thru 15 of the S-bus register. All other S-bus register bits become zeros.
DB (Data Base)	10101	The 16-bit content of the data base (DB) register is loaded into the S-bus register.
DL (Data Limit)	11100	The 16-bit content of the data limit (DL) register is loaded into the S-bus register.
IOA (I/O Address)	01001	The 8-bit content of the interrupt device number (IDN) register is loaded into bits 8 thru 15 of the S-bus register. Bits 0 thru 7 of the S-bus register become zeros.
IOD (I/O Data)	01010	The 16-bit content of the direct input data (DID/MUXMA) register in the IOP is loaded into the S-bus register.
MOD (Module Number)	00101	The MOD S-bus field code provides the CPU with two pieces of information. When executed, the 4-bit content of the interrupt module number (IMN) register is loaded into bits 4 thru 7 of the S-bus register. Also, if the CPU is CPU1, bit 13 of the S-bus register becomes a 1. If CPU2 (Series II only), bit 12 of the S-bus register becomes a 1. These bits are used to fetch the correct Q1 and Z1 entries in the code segment table. All other bits of the S-bus register become zeros.
NOP (No Operation)	11111	No operation.
OPND (Operand)	10110	The 16-bit content of the operand (OPND) register is loaded into the S-bus register. An attempt to execute an OPND while an MCU operand directed operation is in progress results in a CPU freeze until the MCU operation is complete.

Table 3-3. S-Bus Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
P (Program Counter)	10000	The 16-bit content of the program counter (P) register is loaded into the S-bus register.
PADD (Pre-Adder)	00010	The 16-bit output of the pre-adder (PADD) is loaded into the S-bus register.
PB (Program Base)	11110	The 16-bit content of the program base (PB) register is loaded into the S-bus register.
PCLK (Process Clock)	01011	The Process Clock, PCLK, is placed in the S-bus register.
Q (Stack Marker Pointer)	10001	The 16-bit content of the stack marker pointer (Q) register is loaded into the S-bus register.
QDWN (Stack Marker Pointer Down)	01000	The QDWN S-bus field code is used to put the content of the lowest valid TOS register in the S-bus register. During execution, TNAME becomes the sum of NAME and SR(1:2) and the S-bus register is loaded as follows: If TNAME = 00 then S-BUS := TR3S If TNAME = 01 then S-BUS := TR0S If TNAME = 10 then S-BUS := TR1S If TNAME = 11 then S-BUS := TR2S To preserve stack integrity, a DCSR (Decrement SR) code is normally executed. Due to the pipeline effect, a TOS reference in the store field of the preceding microinstruction also uses the above described TNAME.
RA	11011	The RA S-bus field code is used to read the content of the first TOS register (location S). SR must be greater than zero.* During execution, TNAME becomes NAME and the S-bus register is loaded as follows: If TNAME = 00 then S-BUS := TR0S If TNAME = 01 then S-BUS := TR1S If TNAME = 10 then S-BUS := TR2S If TNAME = 11 then S-BUS := TR3S
RB	11010	The RB S-bus field code is used to read the content of the second TOS register (location S-1). SR must be greater than 1.* During execution, TNAME becomes NAME and the S-bus register is loaded as follows: If TNAME = 00 then S-BUS := TR1S If TNAME = 01 then S-BUS := TR2S If TNAME = 10 then S-BUS := TR3S If TNAME = 11 then S-BUS := TR0S
RBR (Read Bank Register)	00011	Read bank register onto S-bus (14:15 for Series II and 12:15 for Series III). The S-bus bits 0 - 13 are zeroed. The bank register to be read is specified in the MCU field. Execution of the Special field is inhibited.

Table 3-3. S-Bus Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
RC	11001	The RC S-bus field code is used to read the content of the third TOS register (location S-2). SR must be greater than 2.* During execution, TNAME becomes NAME and the S-bus register is loaded as follows: If TNAME = 00 then S-BUS := TR2S If TNAME = 01 then S-BUS := TR3S If TNAME = 10 then S-BUS := TR0S If TNAME = 11 then S-BUS := TR1S
RD	11000	The RD S-bus field code is used to read the content of the fourth TOS register (location S-3). SR must be equal to 4.* During execution, TNAME becomes NAME and the S-bus register is loaded as follows: If TNAME = 00 then S-BUS := TR3S If TNAME = 01 then S-BUS := TR0S If TNAME = 10 then S-BUS := TR1S If TNAME = 11 then S-BUS := TR2S
SBUS	01111	The SBUS code causes the S-bus register content to remain unchanged.
SM (Stack Memory)	10011	The 16-bit content of the stack memory (SM) register is loaded into the S-bus register.
SP1 (Scratch Pad 1)	00001	The 16-bit content of the scratch pad 1 (SP1) register is loaded into the S-bus register.
SP2 (Scratch Pad 2)	11101	The 16-bit content of the scratch pad 2 (SP2) register is loaded into the S-bus register.
SP3 (Scratch Pad 3)	10101	The 16-bit content of the scratch pad 3 (SP3) register is loaded into the S-bus register.
STA (Status)	10100	The 16-bit status word is loaded into the S-bus register.
SWCH	00111	The 16-bit content of the switch register is loaded into the S-bus register.
UBUS	01110	The 16-bit U-bus data word is loaded into the S-bus register. The U-bus data is established by the preceding microinstruction.

*True only if RA:RD are being used as part of the stack. RA:RD often are used by the microprogram as scratch pad registers when not used otherwise.

Table 3-4. Store Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
NOP (No Operation)	11111	No Operation.
BSP0 (Bus to Scratch Pad Ø)	00101	The store field code BSP0 stores the U-bus into ACOR or DCOR, depending on the MCU field option selected and in SP0. Disables the special field and enables the MCU options, one of which must be used.
BSP1 (Bus to Scratch Pad 1)	00100	Same as BSP0 except SP1 is used.
BUS	00111	Same as BSP0 except none of the scratch-pad registers are used.
CTRH (Counter High)	01111	The store field code CTRH stores U-bus bits 4 thru 9 in the counter (CNTR) register bits 0 thru 5.
CTRL (Counter Low)	01110	The store field code CTRL stores U-bus bits 10 thru 15 in the counter (CNTR) register bits 0 thru 5.
DB (Data Base)	10011	The store field code DB stores the 16-bit U-bus word in the data base (DB) register.
DL (Data Limit)	11100	The store field code DL stores the 16-bit U-bus word in the data limit (DL) register.
IOA (I/O Address)	00001	The store field code IOA sends the command on the U-bus, bits 5 through 7, to the device whose address is on the U-bus in bits 8 through 15. U-bus, bit 0 = 1, sends a service-out signal to the device.
IOD (I/O Data)	00010	The store field code IOD stores the 16-bit word currently on the U-bus in the direct output data (DOD) register.
MREG (Memory Register)	00011	The store field code MREG is used to store data in an address that lies in a TOS register (i.e., S ≥ E > SM where S = SR + SM). Prior to executing MREG, the value E minus S is placed in the SP1 register. During execution, TNAME becomes the sum of NAME and SP1 (14:15) and the TOS registers are loaded as follows: If TNAME = 00 then TR0 := U-BUS If TNAME = 01 then TR1 := U-BUS If TNAME = 10 then TR2 := U-BUS If TNAME = 11 then TR3 := U-BUS Due to the pipeline effect, a TOS register referenced in the R- or S-bus field of the following microinstruction assumes the above described TNAME.
P (Program Count)	10000	The store field code P stores the 16-bit U-bus word in the program counter (P) register.
PB (Program Base)	11110	The store field code PB stores the 16-bit U-bus word in the program base (PB) register.

Table 3-4. Store Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
PCLK (Process Clock)	00000	The Process Clock, PCLK, is placed in the S-bus register.
PL (Program Limit)	01001	The store field code PL stores the 16-bit U-bus word in the program limit (PL) register.
PUSH	01000	The store field code PUSH effectively moves all stack elements down one location and loads the U-bus word on the top of stack. To maintain stack integrity, SR must be less than four. When PUSH is executed, TNAME becomes NAME and the TOS registers are loaded as follows: If TNAME = 00 then TR3 := U-BUS If TNAME = 01 then TR0 := U-BUS If TNAME = 10 then TR1 := U-BUS If TNAME = 11 then TR2 := U-BUS To complete the operation, NAME is decremented and SR is incremented.
Q (Stack Marker Pointer)	10001	The store field code Q stores the 16-bit U-bus word in the stack marker pointer (Q) register.
QUP (Stack Marker Pointer Up)	01011	The store field code QUP effectively inserts the U-bus word into the stack at location SM plus one. For example, if stack locations S and S minus one are in the TOS registers and location S minus two is the first stack element in memory (location SM), execution of QUP places the U-bus word in a TOS register at stack location S minus two. The first stack element in memory (location SM) becomes S minus three. To maintain stack integrity, the SR register must be incremented (special field code INSR) indicating the addition of a TOS register element. Normally, SR should be less than four. When the store field code QUP is executed, TNAME becomes the sum of NAME and SR and the TOS registers are loaded as follows: If TNAME = 00 then TR0 := U-BUS If TNAME = 01 then TR1 := U-BUS If TNAME = 10 then TR2 := U-BUS If TNAME = 11 then TR3 := U-BUS Due to the pipeline effect, a TOS register referenced in the R- or S-bus fields of the following microinstruction assumes the above described TNAME.

Table 3-4. Store Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
RA	11011	The store field code RA stores the U-bus word in the first TOS register (location S). SR must be greater than zero.* During execution of RA, TNAME becomes NAME and the TOS registers are loaded as follows: If TNAME = 00 then TR0 := U-BUS If TNAME = 01 then TR1 := U-BUS If TNAME = 10 then TR2 := U-BUS If TNAME = 11 then TR3 := U-BUS
RAR (ROM Address Register)	10111	The store field code RAR stores bits 0 thru 15 of the U-bus in ROM address register bits 0 thru 15. The intent of this code is to force the processor to a new microprogram address specified by the U-bus word. Execution of the RAR code requires three microcycles. The first loads the ROM address register and the next two are NOPs allowing the ROM output registers (ROR1 and ROR2) to be loaded with the new microinstruction.
RB	11010	The store field code RB stores the U-bus word in the second TOS register (location S minus one). SR must be greater than one.* During execution of RB, TNAME becomes NAME and the TOS registers are loaded as follows: If TNAME = 00 then TR1 := U-BUS If TNAME = 01 then TR2 := U-BUS If TNAME = 10 then TR3 := U-BUS If TNAME = 11 then TR0 := U-BUS
RC	11001	The store field code RC stores the U-bus word in the third TOS register (location S minus two). SR must be greater than two.* During execution of RC, TNAME becomes NAME and the TOS registers are loaded as follows: If TNAME = 00 then TR2 := U-BUS If TNAME = 01 then TR3 := U-BUS If TNAME = 10 then TR0 := U-BUS If TNAME = 11 then TR1 := U-BUS
RD	11000	The store field code RD stores the U-bus word in the fourth TOS register (location S minus three). SR must be equal to four.* During execution of RD, TNAME becomes NAME and the TOS registers are loaded as follows: If TNAME = 00 then TR3 := U-BUS If TNAME = 01 then TR0 := U-BUS If TNAME = 10 then TR1 := U-BUS If TNAME = 11 then TR2 := U-BUS
SBR (Stack Bank Register)	00110	The store field code SBR stores the U-bus bits (14:15 for Series II and 12:15 for Series III) in the bank register specified in the MCU field code. Execution of the special field is inhibited.

*True only if RA:RD are being used as part of the stack, RA:RD often are used by the microprogram as scratch pad registers when not used otherwise.

Table 3-4. Store Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
SM (Stack Memory Pointer)	10010	The store field code SM stores the U-bus word in the stack memory (SM) register.
SP0 (Scratch Pad 0)	01101	The store field code SP0 stores the U-bus word in the scratch pad 0 (SP0) register.
SP1 (Scratch Pad 1)	01100	The store field code SP1 stores the U-bus word in the scratch pad 1 (SP1) register.
SP2 (Scratch Pad 2)	11101	The store field code SP2 stores the U-bus word in the scratch pad 2 (SP2) register.
SP3 (Scratch Pad 3)	10101	The store field code SP3 stores the U-bus word in the scratch pad 3 (SP3) register.
STA (Status)	10100	The store field code STA stores the U-bus word in the status register.
X (Index)	10110	The store field code X stores the U-bus word in the index (X) register.
Z (Stack Limit Pointer)	01010	The store field code Z stores the U-bus word in the stack limit pointer (Z) register.

Table 3-5. Function Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
ADD	11111	The content of the R-bus register is added to the content of the S-bus register and the result is placed on the T-bus.
ADDO (Add-Enable Overflow)	11011	The content of the R-bus register is added to the content of the S-bus register and the result is placed on the T-bus. Carry and overflow are modified in the status register and CCA is set on the T-bus.
AND	00111	The content of the R-bus register is logically "anded" with the content of the S-bus register and the result is placed on the T-bus.
BNDT (Bounds Test)	01101	The function field code BNDT is used to perform a bounds test of an address. Execution of this code results in the content of the R-bus register minus the content of the S-bus register being placed on the T-bus. If RRZ, RLZ, LRZ, LLZ is specified, then BNDT does a "CAD" instead of a "SUB." The R- and S-bus fields are coded so that this result is a negative number (CARRY = 0) if a bounds violation occurs. If the CPU is not operating in the priviledged mode (STATUS(0) = 0), and a bounds violation occurs, a microjump to ROM address 0003 is executed. If no violation has occurred (CARRY = 1) or the CPU is operating in the priviledged mode (STATUS(0) = 1), the next microinstruction will be executed in the usual manner.
CAD (Complement and Add)	01110	The content of the R-bus register is added to the one's complement of the content of the S-bus register and the result is placed on the T-bus. If the S-bus register contains all zeros, CAD results in the R-bus register content minus 1 on the T-bus.
CADO (Complement and Add-Enable Overflow)	01010	The content of the R-bus register is added to the one's complement of the content of the S-bus register and the result is placed on the T-bus. Carry and overflow are modified in the status register and the condition code is set to CCA on the T-bus data.
CAND (Complement-And)	00101	The R-bus register content is logically "anded" with the complement of the S-bus register content and the result is placed on the T-bus.
CRS (Circular Shift)	11010	The R-bus register content is added to the S-bus register content and the result is placed on the T-bus. The T-bus is then circular shifted one place right or left as specified in the shift field (SR1 or SL1) and placed on the U-bus.
CTSD (Controlled Shift Double)	10111	The function field code CTSD adds the contents of the R-bus register and the S-bus register, puts the result on the T-bus, and performs a double word shift of the T-bus and a scratch pad left or right as specified by the shift field code (SR1 or SL1). The type of shift is determined by the content of the current instruction register (CIR) as follows: If CIR(7) = 1 then circular shift If CIR(7:8) = 01 then logical shift If CIR(7:8) = 00 then arithmetic shift The most significant word is on the T-bus. If a left shift is specified, scratch pad 1 (SP1) contains the least significant word. If a right shift is specified, scratch pad 3 (SP3) contains the least significant word. Regardless of the direction of the shift, both SP1 and SP3 are shifted left and right respectively.

Table 3-5. Function Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
CTSS (Controlled T-bus Shift Single)	11100	The R-bus register content is added to the S-bus register content and the result is placed on the T-bus. The T-bus is then shifted left or right as specified by the shift field code (SR1 or SL1). The type of shift is determined by the content of the current instruction register as follows: If CIR(7) = 1 then circular shift If CIR(7:8) = 01 then logical shift If CIR(7:8) = 00 then arithmetic shift
DCAD (Decimal Add)	11110	The contents of the R- and S-bus registers are added and the results are placed into the decimal correction adder. The decimal corrector adder output is placed on the U-bus.

DVSB (Divide-Subtract)	01000	The function field code DVSB performs the subtract, shift, and test necessary to execute a divide algorithm. The R- and S-bus fields of the microinstruction are coded so that initially the 16-bit divisor is in the S-bus register and the most significant 16-bits of the dividend are in the R-bus register. The least significant 16-bits of the dividend are in the SP1 register. Both divisor and dividend must be positive numbers upon execution of the DVSB code and Flag 2 (F2) must be 0 (cleared). An SL1 code in the shift field of the microinstruction directs the left shift of the T-bus. The follcwing algorithm is then executed repeatedly to perform the complete divide.
		
		TBUS := RBUS - SBUS;
		UBUS(0:14) := TBUS(1:15);
		If ALU carry or F2=1 then
		BEGIN
		RREG(0:14) := UBUS(0:14);
		RREG(15) := SP1(0);
		SP1(0:14) := SP1(1:15);
		SP1(15) := 1
		F2 := TBUS(0);
		END
		else
		BEGIN
		RREG(0:14) := RREG(1:15);
		RREG(15) := SP1(0);
		SP1(0:14) := SP1(1:15);
		SP1(15) := 0
		F2 := RREG(0);
		END
		
		For example, after 17 executions of the above algorithm, a 16-bit quotient is contained in the SP1 register and the remainder times 2 is contained in the R-bus register. When the remainder is unloaded from the R-bus register, it is shifted right one place (divided by 2).

Table 3-5. Function Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
INC (Incremented Add)	11101	The R-bus register content is added to the S-bus register content plus 1. The result is placed on the T-bus.
INCO (Incremented Add-Enable Overflow)	11001	The R-bus register content is added to the S-bus register content plus 1, the result placed on the T-bus, and the carry and overflow is modified in the status register. The condition code is set to CCA on the T-bus data.
IOR (Inclusive OR)	10110	The content of the R-bus register is logically inclusively "ored" with the content of the S-bus register and the result is placed on the T-bus.
JMP (Jump)	01100	The JMP function field code directs a micro-jump to the ROM address (jump target) specified by bits 20 thru 31 of the ROM output register if the skip field condition is met (a condition must be specified). The R-bus, shift, and special field decoders are disabled and the U-bus and T-bus become the S-bus register content.
JSB (Jump to Subroutine)	00100	The JSB function field code directs a micro-subroutine jump to the ROM address specified by bits 20 thru 31 of the ROM output register if the skip field code condition is met. If the condition is met and the JSB is executed, the save register is loaded with the address of the line following the JSB and is used as a return address at the subroutine end (see function field code RSB). During execution of the JSB, the R-bus, shift, and special field decoders are disabled and the T-bus and U-bus become the S-bus register content.

MPAD (Multiply-Add)	11000	The function field code MPAD performs the add, shift, and test necessary to execute a multiply algorithm. The R-bus field of the microinstruction is coded so that initially the 16-bit multiplicand is in the R-bus register. The S-bus field code is UBUS which is initially all zeros. Scratch pad 3 contains the 16-bit multiplier. Both multiplier and multiplicand must be positive numbers upon execution of the MPAD code. An SR1 code in the shift field directs the right shift of the T-bus. The following algorithm is executed repeatedly to perform a complete multiply.
		
		T-BUS := R-REG plus S-REG;
		U-BUS(1:15) := T-BUS(0:14);
		U-BUS(0) := ALUcarry;
		If SP3(15) = 1 then
		BEGIN
		S-REG := U-BUS;
		SP3(1:15) := SP3(0:14);
		SP3(0) := T-BUS(15);
		END
		else

Table 3-5. Function Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
		BEGIN S-REG(1:15) := S-REG(0:14); SP3(1:15) := SP3(0:14); SP3(0) := S-REG(15); END After 16-executions of the above algorithm, the result is a 32-bit word with the most significant 16-bits in the S-bus register and the least significant 16-bits in scratch pad 3.
PNLR (Panel Read)	10000	The PNLR function field code allows the auxiliary control panel to select and display a CPU register. This code appears in the microprogram during execution of HALT and PAUSE routines. When PNLR is executed, the ROM output register (ROR1) R-bus and S-bus fields are disabled. The maintenance panel interface supplies these field codes which put the content of the selected register in the associated (R- or S-bus) register. The T-bus and U-bus become the R-bus register content plus the S-bus register content (one of which will be zeros). The auxiliary control panel completes this operation by displaying the U-bus as the selected register.
PNLS (Panel Store)	10001	The PNLS function field code allows the auxiliary control panel to load a CPU register with the content of one of its switch registers. This code is part of the halt mode interrupt micro-routine for servicing a maintenance panel interrupt. When PNLS is executed, the ROM output register (ROR2) store field is disabled and the maintenance panel interface card supplies the store field code respective of the selected CPU register. A SWCH S-bus field code causes the S-bus register to be loaded with the content of the selected auxiliary control panel switch register. The T-bus and U-bus become the R-bus register content (zeros) plus the S-bus register content and at the end of the cycle, the selected register is loaded with the U-bus data.
QASL	00000	The QASL function field code causes a four register arithmetic shift left of the U-bus, scratch pad 3, scratch pad 1, and the R-bus register containing the most, next most, next least, and least significant word respectively. Shift Left One code (SL1) is required in the shift field. The sign bit is preserved. T-BUS := S-REG; U-BUS(0) := T-BUS(0); U-BUS(1:14) := T-BUS(2:15); U-BUS(15) := SP3(0); SP3(0:14) := SP3(1:15); SP3(15) := SP1(0); SP1(0:14) := SP1(1:15); SP1(15) := R-REG(0); R-REG(0:14) := R-REG(1:15); R-REG(15) := 0

Table 3-5. Function Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
QASR	00001	The QASR function field code causes a four register arithmetic shift right of the U-bus, scratch pad 3, scratch pad 1, and the S-bus register containing the most, next most, next least, and least significant words respectively. Shift right one code (SR1) is required in the shift field. The sign bit is propagated. T-BUS := R-REG; U-BUS(0:1) := T-BUS(0); U-BUS(2:15) := T-BUS(1:14); SP3(0) := T-BUS(15); SP3(1:15) := SP3(0:14); SP1(0) := SP3(15); SP1(1:15) := SP1(0:14); S-REG(0) := SP1(15); S-REG(1:15) := S-REG(0:14);
REPC (Repeat Until Condition)	10100	The REPC function field code causes the next microinstruction to be executed repeatedly until the skip field condition of that microinstruction is met. During execution the T-bus becomes the R-bus register content plus the S-bus register content. The REPC code is decoded from ROR2 and at that time disables the RAR increment function and the ROR1 load function and sets the Repeat FF. The RAR then contains the address of the microinstruction following the one to be repeated and ROR1 contains the microinstruction to be repeated. The next cycle loads ROR2 and executes the microinstruction to be repeated. As long as the Repeat FF remains set, the content of ROR1 and ROR2 does not change and is executed each cycle. When the skip field condition is met, the Repeat FF is cleared, the pipeline is filled correctly, and the next microinstruction is fetched in the usual manner.
REPN (Repeat N Times)	10101	The REPN function code operates in the same manner as the REPC code described for the preceding label. The difference is that REPN loads a repeat counter register with the content of the microinstruction skip field. Bits 1 thru 5 of the counter become ROR2 bits 5 thru 19; bit 0 of the counter becomes a 1. The counter content is then the two's complement of the number of repeats to be performed. To utilize the counter, the repeated microinstruction contains a special field code INCTR (Increment Counter) and a skip field condition CTRM (Counter Maximum).
ROM	10011	The function field code ROM loads the R-bus register with a 16-bit constant obtained from the microinstruction. The ROM code is decoded from ROR1, loading the R-bus register with bits 16 thru 31 of ROR1. The T-bus then becomes the R-bus register content plus the S-bus register content. The R-bus, shift, special, and skip field decoders are disabled by the ROM code.
ROMI (ROM Inclusive)	10010	The function field code ROMI loads the R-bus register with a 16-bit constant obtained from the microinstruction. The ROMI code is decoded from ROR1, loading the R-bus register with ROR1 bits 16 thru 31. The T-bus then becomes the R-bus register content inclusive "ored" with the S-bus register content. The R-bus, shift, special, and skip field decoders are disabled by the ROMI code.

Table 3-5. Function Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
ROMN (ROM And)	00011	The function field code ROMN loads the R-bus register with a 16-bit constant obtained from the microinstruction. The ROMN code is decoded from ROR1, loading the R-bus register with ROR1 bits 16 thru 31. The T-bus becomes the R-bus register content logically "anded" with the S-bus register content. The R-bus, shift, special, and skip field decoders are disabled by the ROMN code.
ROMX (ROM Exclusive)	00010	The function field code ROMX loads the R-bus register with a 16-bit constant obtained from the microinstruction. The ROMX code is decoded from ROR1, loading the R-bus register with ROR1 bits 16 thru 31. The T-bus becomes the R-bus register content exclusive "ored" with the S-bus register content. The R-bus, special, shift, and skip field decoders are disabled by the ROMX code.
SUB (Subtract)	01111	The content of the S-bus register is subtracted from the content of the R-bus register and the result is placed on the T-bus.
SUBO (Subtract-Enable Overflow)	01011	The content of the S-bus register is subtracted from the content of the R-bus register and the result is placed on the T-bus. Carry and overflow are modified in the status register and condition code CCA is set on the T-bus data.
UBNT (Unconditional Bounds Test)	01001	The function field code UBNT is used to perform an unconditional bounds test of an address. Execution of this code results in the content of the R-bus register minus the content of the S-bus register being placed on the T-bus. If RRZ, RLZ, LRZ, LLZ is specified, then UBNT does a "CAD" instead of a "SUB." The R-bus and S-bus field are coded so that this result is a negative number (CARRY = 0) if a bounds violation occurs. The response to a bounds violation is a micro-jump to ROM address 0003. If no violation occurs (CARRY = 1), the next microinstruction is executed in the usual manner.
XOR (Exclusive OR)	00110	The content of the R-bus register is exclusive "ored" with the content of the S-bus register and the result is placed on the T-bus.

Table 3-6. Skip Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
BIT6	00101	The skip field code BIT6 sets the NOP2 FF if bit 6 of the U-bus word is a logic 1.
BIT8	00110	The skip field code BIT8 sets the NOP2 FF if bit 8 of the U-bus word is a logic 1.
CRRY (Carry)	01000	The skip field code CRRY sets the NOP2 FF if the ALU carry out is a logic 1.
CTRM (Counter Max)	11011	The skip field code CTRM sets the NOP2 FF if the counter contains all ones.
EVEN	00010	The skip field code EVEN sets the NOP2 FF if the U-bus word is an even number (U-bus bit 15 is a logic 0).
F1 (Flag 1)	01100	The skip field code F1 sets the NOP2 FF if Flag 1 FF is set.
F2 (Flag 2)	01110	The skip field code F2 sets the NOP2 FF if Flag 2 FF is set.
F3 (Flag 3)	11100	The skip field code F3 sets the NOP2 FF if Flag 3 FF is set.
INDR (Indirect)	10100	The skip field code INDR sets the NOP2 FF if the Indirect Bit FF is set and the indirect signal is a logic 1.
JLUI	11001	The skip field code JLUI causes a microjump to the ROM address specified by the LUT (look up table) providing the indirect condition (skip field INDR code) is not met. If the indirect condition is met the microjump is not executed.
NCRY	01001	The skip field code NCRY sets the NOP2 FF if the carry out from the ALU is zero.
NEG	01011	The skip field code NEG sets the NOP2 FF if the U-bus word is a negative number (U-bus bit 0 is a logic 1).

Table 3-6. Skip Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
NEXT	11101	Terminates current instruction and initiates the sequence necessary to begin execution of the next instruction. If stackop A has just been executed and stackop B is not a NOP, then the hardware executes stackop B. Otherwise the action shown in the timing figure below takes place (a, b, c, d, e, f are equal length CPU clock cycles): a b c d e f ... Mem. Sel. cycle DATA→NIR NIR→LUT ... NEXT: BUSL, RWP; Issue LOREQ; LUT→ VBUS→ ROM→ RANK1; NIR→CIR — NOP2: P+1→P Select cycle. RANK1→ RANK2 (if memory reference, force PADD, BASE to R, S-BUS Reg's.) — execute 1st line of micro code of new instr. Time periods a, b, c (if present), and d occur in the currently executing instruction. "a" and "b" must occur before "d" for maximum execution speed – otherwise a CPU freeze will occur at "d". "a" and "b" result from the "next instruction prefetch" of the current instruction. "c" may or may not be present depending on the length of the current instruction. "d" is the last line of the current instruction. It initiates a "next instruction prefetch", transfers (NIR) to CIR, and applied the address on the VBUS (normally using the LUT output) to the ROM input. The ROM word at this address is stored in RANK1. In addition, the NOP2 FF is set. "e" is used to increment the P-reg., transfer RANK1 to RANK2, and if the new instruction is a memory-reference type, load the R- and S-BUS reg's. with the Pre-adder output and the proper base register. This is also the "select" cycle for the "next instr. prefetch" if there is no MCU conflict. During "f", the first line of the new instruction is executed. The above is the normal sequence of operation of NEXT. This sequence is modified in the event an interrupt is pending or the micro code line is "... DATA NEXT". "NEXT" also clears F1, F2, F3, CNTR, Subroutine Flag FF, and the ABS-BANK reg.
NF1 (Not Flag 1)	01101	The skip field code NF1 sets the NOP2 FF if Flag 1 FF is cleared.
NF2 (Not Flag 2)	01111	The skip field code NF2 sets the NOP2 FF if Flag 2 FF is cleared.
NOFL (Not Overflow)	00111	The skip field code NOFL sets the NOP2 FF if the ALU overflow bit is not a logic 1. Causes conditional jump and JSB to be two-cycle instructions.
NOP (No Operation)	11111	

Table 3-6. Skip Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
NPRV (Not Privileged)	10110	The skip field code NPRV sets the NOP2 FF if the privileged mode bit (status word bit 0) is zero.
NSME (Not Same)	00100	The skip field code NSME sets the NOP2 FF if all bits of the T-bus are not the same.
NZRO (Not Zero)	00001	The skip field code NZRO sets the NOP2 FF if the T-bus word is not equal to zero.
ODD	00011	The skip field code ODD sets the NOP2 FF if the U-bus word is an odd number (U-bus bit 15 is a logic 1).
POS (Positive)	01010	The skip field code POS sets the NOP2 FF if the U-bus word is a positive number (U-bus bit 0 is a logic 0).
RSB (Return from Subroutine)	11000	The skip field code RSB causes a microjump to the ROM address contained in the save register.
SR4 (SR=4)	10010	The skip field code SR4 sets the NOP2 FF if the SR register content is equal to four.
SRL2 (SR<2)	10101	The skip field code SRL2 sets the NOP2 FF if the SR register content is less than two.
SRL3 (SR<3)	10111	The skip field code SRL3 sets the NOP2 FF if the SR register content is less than three.
SRN4 (SR Not 4)	10011	The skip field code SRN4 sets the NOP2 FF if the SR register content is not four.
SRNZ (SR Not Zero)	10001	The skip field code SRNZ sets the NOP2 FF if the SR register content is not zero.
SRZ (SR Zero)	10000	The skip field code SRZ sets the NOP2 FF if the SR register content is equal to zero.
TEST	11010	The skip field code TEST sets the NOP2 FF if any enabled interrupt is pending.
UNC (Unconditional)	11110	The skip field code UNC and/or unconditional JMP's set the NOP2 FF.
ZERO	00000	The skip field code ZERO sets the NOP2 FF if the T-bus word is equal to zero.

Table 3-7. Shift Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
(blank)	111	The T-bus word is placed directly on the U-bus.
LLZ (Left to Left and Zero)	001	The shift field code LLZ places the left byte of the T-bus in the left byte of the U-bus and zeros in the right byte of the U-bus.
LRZ (Left to Right and Zero)	000	The shift field code LRZ places the left byte of the T-bus in the right byte of the U-bus and zeros in the left byte of the U-bus.
RLZ (Right to Left and Zero)	101	The shift field code RLZ places the right byte of the T-bus in the left byte of the U-bus and zeros in the right byte of the U-bus.
SWAB (Swap Bytes)	110	The shift field code SWAB places the left byte of the T-bus in the right byte of the U-bus and the right byte of the T-bus in the left byte of the U-bus.
RRZ (Right to Right and Zero)	100	The shift field code RRZ places the right byte of the T-bus in the right byte of the U-bus and zeros in the left byte of the U-bus.
SL1 (Shift Left 1)	010	The shift field code SL1 shifts the T-bus one place left onto the U-bus. Refer to the function field code descriptions for the action taken when used with function field codes CRS, CTSD, CTSS, DVSB, and TASL.
SR1 (Shift Right 1)	011	The shift field code SR1 shifts the T-bus logically one place right onto the U-bus. Refer to the function field code descriptions for the action taken when used with function field codes CRS, CTSD, CTSS, MPAD, and TASR.

Table 3-8. Special Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
(blank)	11111	No special field operation.
CCA (Condition Code A)	11110	The special field code CCA sets the condition code bits to CCL (01) if the T-bus word is less than zero (T(0) = 1), CCE (10) if the T-bus word is equal to zero (Signal T = 0 is true), or CCG (00) if the T-bus word is greater than zero (T(0) = 0 and signal T = 0 is false).
CCB (Condition Code B)	00000	The special field code CCB sets the condition code to CCL (01) if bits 8 thru 15 of the U-bus form a special ASCII character, CCE (10) if an alphabetic ASCII character, or CCG (00) if a numeric ASCII character.
CCE (Condition Code E)	11101	The special field code CCE sets the condition code bits to CCE (10).
CCG (Condition Code G)	11100	The special field code CCG sets the condition code bits to CCG (00).
CCL (Condition Code L)	11011	The special field code CCL sets the condition code bits to CCL (01).
CCPX (Clear CPX1)	00001	Clears the interrupt status register bits as specified by the true bits on the U-bus. U-Bus bit 0 Halt 1 Run 2 System Halt 3 (Unused) Bits 4 (MSB) through 7 (LSB) code the following functions: Octal Code 0 NOP 1 Clear BNDV 2 Clear Illegal Address 3 Clear CPU Timer 4 Clear System Parity Error 5 Clear Address Parity Error 6 Clear Data Parity Error 7 Clear Module Interrupt 10 Clear External Interrupt 11 Power Fail Turn-Off Interrupt 16 Reverse System Parity 17 Reverse MCUD Parity 8 Diagnostic NIRTOCIR 9 (Unused) 10 Diagnostic Set CPX1 (Bits 1:8) 11 Clear ICS Flag 12 Clear DISP Flag

Table 3-8. Special Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
		13 (Unused) 14 Diagnostic Freeze 15 Clear Panel FF's
CCRY (Clear Carry)	10101	The special field code CCRY clears carry in the status register.
CCZ (Condition Code Zero)	11010	The special field code CCZ sets the condition code bits to CCE (10) if the T-bus word is equal to zero (signal T = 0 true) or CCG (00) if the T-bus word is not equal to zero (signal T = 0 false).
CF1 (Clear Flag 1)	10010	The special field code CF1 clears CPU Flag 1 FF.
CF2 (Clear Flag 2)	10001	The special field code CF2 clears CPU Flag 2 FF.
CF3 (Clear Flag 3)	00111	The special field code CF3 clears CPU Flag 3 FF.
CLIB (Clear Indirect Bit)	01110	At the end of the cycle, CLIB sets the Indirect Bit FF which masks the indirect line until a NEXT or JLUI option in the skip field is encountered.
CLO (Clear Overflow)	11001	The special field code CLO clears the status word overflow bit.
CLSR (Clear SR)	00010	The special field code CLSR clears the SR register. This is an asynchronous reset. No other SR operation is allowed during that time.
CTF (Set Carry to Flag 1)	00110	The special field code CTF stores the ALU carry bit in the Flag 1 FF.
DCSR (Decrement SR)	01001	The special field code DCSR decrements the content of the SR register by a count of one.
FHB (Flag to High Bit)	01101	The special field code FHB transfers the content of the Flag 1 FF to bit 0 of the U-bus.
HBF (High Bit to Flag 1)	01100	The special field code HBF transfers the content of U-bus bit 0 to the Flag 1 FF.
INCN (Increment Name)	01010	The special field code INCN increments the content of the name register by a count of one.
INCT (Increment Counter)	01011	The special field code INCT increments the content of the counter register by a count of one.
INSR (Increment SR)	01000	The special field code INSR increments the content of the SR register by a count of one.
LBF (Low Bit to Flag 2)	01111	The special field code LBF transfers the content of U-bus bit 15 to the Flag 2 FF.

Table 3-8. Special Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
NOP (No Operation)	10111	No operation.
POP	10111	The special field code POP moves the stack elements up one location such that the second element of the stack (S minus one) becomes the top element (S), etc. The previous top of stack element is lost. When executed, this is accomplished by decrementing the SR register and incrementing the name register.
POPA (Pop setting CCA)	10110	The special field code POPA functions the same as special field code POP with the addition that the condition code is set to CCL (01) if the T-bus word is less than zero, CCE (10) if the T-bus word is equal to zero, or CCG (00) if the T-bus word is greater than zero.
SCRY (Set Carry Bit)	10100	The special field code SCRY sets Carry in the status register.
SDFG (Set Dispatcher Flag)	00101	The special field code SDFG sets the dispatcher flag (bit 12 of interrupt status register CPX1).
SF1 (Set Flag 1)	10011	The special field code SF1 sets CPU Flag 1 FF.
SF2 (Set Flag 2)	10000	The special field code SF2 sets CPU Flag 2 FF.
SF3 (Set Flag 3)	00011	The special field code SF3 sets CPU Flag 3 FF.
SIFG (Set Interrupt Stack Flag)	00100	The special field code SIFG sets the interrupt flag (bit 11 of interrupt status register CPX1).
SOV (Set Overflow)	11000	The special field code SOV sets the status word overflow bit.

Table 3-9. MCU Option Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
ABS (Absolute)	00000	The MCU option code ABS specifies the Absolute bank register which may be read into S-bus (14:15 for Series II and 12:15 for Series III) with "RBR" or stored from U-bus (14:15 for Series II and 12:15 for Series III) with "SBR". This bank register is used as a scratch pad bank register by the micro code.
CMD (Command)	00010	The MCU option code CMD enables the bus options (BUS, BSP0, and BSP1) in the store field to store the U-bus into the address CPU output register, ACOR, and to initiate a low-request command. When "selected", the ACOR is output to the MCU-bus, and the command and module number ("TO" lines) are obtained from the CRL register.
CRL (Control)	00001	The MCU option code CRL enables the store field bus options (BUS, BSP0, and BSP1) to load the S-bit CRL register from the U-bus as follows: CRL(0:1) := U-BUS(10:11) Command CRL(2:4) := U-BUS(13:15) Address. The CPU freezes until any pending MCU requests are completed.
DATA	10001	The MCU option code DATA enables the store field bus options (as in CRL) to store the U-bus into DCOR, and initiates a "high-request" command.
DB (DB - Relative)	10000	The MCU option code DB functions the same as ABS except that it specifies the DB-bank register used with DB - relative addressing.
DPOP (Data - POP)	10010	The DPOP MCU option code functions the same as DATA and also pops the stack.
NIR (Next Instruction Register)	01001	The MCU option code NIR enables the store field bus options (as in CRL) to store the U-bus into DCOR, and initiates a "high-request" command. On the following "select" cycle, DCOR is read into the MCU bus and stored in the CPU NIR register.
OPND (Operand Register)	11001	Same as NIR except that the MCU bus is stored in the CPU OPND register.
PB (PB - relative addressing)	01000	Same as ABS, except that it specifies the PB-bank register used in PB-relative addressing.
RND (Returned Data)	10100	The MCU option code RND enables the store field bus options (as in CRL) to store the U-bus in ACOR and initiate a "low-request" command. The DB-bank register generates the module number used to initiate a data fetch from memory. The returned data is loaded into the NIR register.
RNP	10111	Same as RND, except that the PB-bank register generates the module number.
RNS	11100	Same as RND, except that the stack-bank register generates the module number.
ROA	00111	Same as RND, except that the ABS-bank register generates the module number and the returned data is loaded into the OPND register.

Table 3-9. MCU Option Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
ROD	10111	Same as RND, except that the DB-bank register generates the module number and the returned data is loaded into the OPND register.
ROND	10011	Same as RND, except that the returned data is stored in both the NIR and OPND registers.
RONP	01011	Same as RNP, except that the returned data is stored in both the NIR and OPND registers.
RONS	11011	Same as RNS, except that the returned data is stored in both the NIR and OPND registers.
ROP	01111	Same as RNP, except that the returned data is stored in the OPND register.
ROS	11111	Same as RNS, except that the returned data is stored in the OPND register.
ROSA	00101	Same as ROA, except that the addressed word is set to all 1's in memory during the same, non-interruptable, memory cycle. (Series II only.)
ROSD	10101	Same as ROSA, except that DB-bank is used. (Series II only.)
S	11000	Same as ABS, except that the stack-bank register is specified and is used with DB, Q, or S-relative addressing.
WRA	00110	The MCU option code WRA enables the store field bus options (as in CRL) to store the U-bus in ACOR and initiate a "low-request" command. The ABS-bank register generates the module number used to initiate a data store into memory. On the "select" cycle, the addressed memory module interprets the MCU bus data as an address and goes "busy". The module stays busy until it receives the data to be stored (normally sent on the following cycle with a microcode BUS DATA instruction) and completes the "write" cycle, or until its timer runs down.
WRD	10110	Same as WRA, except that the DB-bank register generates the module number.
WRS	11110	Same as WRA, except that the stack-bank register generates the module number.

Table 3-10. R-Bus Field Code Definitions

LABEL AND NAME	FIELD CODE	DESCRIPTION
NOP (No Operation)	1111	No Operation.
MREG	0011	The MREG R-bus field code is used to fetch a memory element that happens to lie in a TOS register (i.e., E is greater than SM). Prior to executing MREG, the value S minus E must be placed in the SP1 register. During execution of MREG, TNAME becomes the sum of NAME and SP1(14:15) and the R-bus register is loaded as follows: If TNAME = 00 then R-BUS := TR0R If TNAME = 01 then R-BUS := TR1R If TNAME = 10 then R-BUS := TR2R If TNAME = 11 then R-BUS := TR3R Due to the pipeline affect, a TOS register referenced in the store field of the preceding microinstruction assumes the above TNAME.
PADD (Pre-Adder)	0100	The 16-bit output of the pre-adder is loaded into the R-bus register.
PL (Program Limit)	0000	The 16-bit content of the program limit (PL) register is loaded into the R-bus register.
RA	1011	The RA R-bus field code is used to read the content of the first TOS register (location S). SR must be greater than 0.* During execution, TNAME becomes NAME and the R-bus register is loaded as follows: If TNAME = 00 then R-BUS := TR0R If TNAME = 01 then R-BUS := TR1R If TNAME = 10 then R-BUS := TR2R If TNAME = 11 then R-BUS := TR3R
RB	1010	The RB R-bus field code is used to read the second TOS register (location S-1). SR must be greater than 1.* During execution, TNAME becomes NAME and the R-bus register is loaded as follows: If TNAME = 00 then R-BUS := TR1R If TNAME = 01 then R-BUS := TR2R If TNAME = 10 then R-BUS := TR3R If TNAME = 11 then R-BUS := TR0R
RBUS	0101	The RBUS R-bus field code causes the R-bus register to remain unchanged.
RC	1001	The RC R-bus field code is used to read the third TOS register (location S2). SR must be greater than 2.* During execution, TNAME becomes NAME and the R-bus register is loaded as follows: If TNAME = 00 then R-BUS := TR2R If TNAME = 01 then R-BUS := TR3R If TNAME = 10 then R-BUS := TR0R If TNAME = 11 then R-BUS := TR1R

*True only if RA:RD are being used as part of the stack. RA:RD often are used by the microprogram as scratch pad registers when not used otherwise.

Table 3-10. R-Bus Field Code Definitions (Continued)

LABEL AND NAME	FIELD CODE	DESCRIPTION
RD	1000	The RD R-bus field code is used to read the fourth TOS register (location S-3). SR must be equal to 4.* During execution, TNAME becomes NAME and the R-bus register is loaded as follows: If TNAME = 00 then R-BUS := TR3R If TNAME = 01 then R-BUS := TR0R If TNAME = 10 then R-BUS := TR1R If TNAME = 11 then R-BUS := TR2R
SP0 (Scratch Pad 0)	1101	The 16-bit content of the scratch pad 0 (SP0) register is loaded into the R-bus register.
SP1 (Scratch Pad 1)	1100	The 16-bit content of the scratch pad 1 (SP1) register is loaded into the R-bus register.
SR (Stack Register)	0001	The 3-bit content of the stack (SR) register is loaded into the R-bus register bits 0 thru 2. R-bus register bits 3 thru 15 become zeros.
UBUS	1110	The 16-bit U-bus data word is loaded into the R-bus register. The U-bus data is established by the preceding microinstruction.
X (Index)	0110	The 16-bit content of the index (X) register is loaded into the R-bus register.
XC (X Conditional)	0111	The XC R-bus field code is used with indexed memory addressing. If the index bit of the current instruction (CIR bit 4) is zero, the R-bus register is loaded with zeros, otherwise the R-bus register is loaded with the 16-bit content of the X-register.
Z (Stack Limit)	0010	The 16-bit content of the stack limit (Z) register is loaded into the R-bus register.

*True only if RA:RD are being used as part of the stack. RA:RD often are used by the microprogram as scratch pad registers when not used otherwise.

3-99. MCU OPTION FIELD DECODER. The MCU option field decoder (bits 23:27) uses the same bits as the special field. (The special field is disabled and the MCU option field is enabled when executing a store field code BUS, BSP0, and BSP1.) MCU option field codes initiate transfers to or from memory and transfers from ACOR to the operand, next instruction, or command registers via the central data bus. MCU option field code definitions are shown in table 3-9.

3-100. R-BUS FIELD DECODER. The R-bus field decoder (bits 28:31) selects one of 16 registers (or sets of lines) for loading into the R-bus register. R-bus field code definitions are shown in table 3-10.

3-101. PROCESSOR REGISTERS. The processor registers may be selectively loaded from the U-bus (except the operand, I/O address, I/O direct data in, CPX1 and CPX2 registers) and selectively read into the R- and/or S-bus registers. Figure 7-3 groups 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. SP1 register may be read out to either the R- and S-bus register. 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 in Section VII, so will not be discussed here. 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 (the R- and S-bus registers contain the most and least significant words, respectively).

3-102. RENAMER. The logic consisting of the namer, adder, three 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 TR0 through TR3 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. (see table 3-11). 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 the TOS in the CPU, and that one or more memory fetches may be required.

The adder combines the outputs of the namer, SR register, and scratch pad 1, and generates the TNAME (0:1) signals for the TOS mappers.

The TOS mappers use the TNAME code to control access to the TOS registers. The TNAME code signifies which of the TOS registers (TR0, TR1, TR2, or TR3) is RA, RB, RC, and RD.

Table 3-11. Top-of-Stack Namer Relationships

TNAME	=	00	01	10	11
RA	=	TR0	TR1	TR2	TR3
RB	=	TR1	TR2	TR3	TR0
RC	=	TR2	TR3	TR0	TR1
RD	=	TR3	TR0	TR1	TR2

3-103. TOP-OF-STACK REGISTERS. The top-of-stack registers consist of eight 16-bit registers designated TR0R:TR3R and TR0S:TR3S. The two groups of registers always contain the same data, i.e., TR0R = TR0S, TR1R = TR1S, etc. The registers contain up to four of the top elements of the current data stack. The TOS registers are read by R-bus field codes RA, RB, RC, RD, and MREG and S-bus field codes RA, RB, RC, RD, and QDWN as described in the field code definitions. The TOS registers are loaded by store field codes RA, RB, RC, RD, PUSH, and QUP as described in the field code definitions.

3-104. INDEX REGISTER. The index (X) register is a 16-bit register containing the index word to be used by memory reference instructions if indexing is specified. Certain other instructions (not memory reference) use the X register for parameters or addresses. The X register is read by R-bus field codes X and XC and loaded by store field code X as described in the field code definitions.

3-105. STACK LIMIT REGISTER. The stack limit (Z) register is a 16-bit register which contains an absolute address pointing to the top memory location available to the current data stack. There are locations (128 words) in the stack above the stack limit; however, these are reserved for stack markers in the event of an interrupt. The Z register is read by R-bus field code Z and loaded by store field code Z as described in the field code definitions.

3-106. PROGRAM LIMIT REGISTER. The program limit (PL) register is a 16-bit register which contains the absolute address of the upper location of the current program segment. The PL register is read by an R-bus field code PL and loaded by a store field code PL as described in the field code definitions.

3-107. SCRATCH PAD 0 REGISTER. The scratch pad 0 (SP0) register is a 16-bit register used by the CPU for storage of partial results during various CPU routines and as an address for memory transfers. The SP0 register is read by R-bus field code SP0 and loaded by store field codes SP0 and BSP0 as described in the field code definitions.

3-108. SCRATCH PAD 1 REGISTER. The scratch pad 1 (SP1) register is a 16-bit register used by the CPU to store partial results during various microprogram routines. The SP1 register can be shifted left and provides serial data input to bit 15 and output from bit 0. The SP1 register is read by R-bus field code SP1, loaded by store field code SP1, and shifted by function field codes CTSD, DVSB, and QASL as described in the field code definitions. In addition, SP1 can be read onto the S-bus by S-bus field code SP1 (code is not the same as R-bus code SP1).

3-109. STACK REGISTER. The stack register (SR) counter is a three-bit register that provides the number of TOS registers that are currently in use. The SR counter works in conjunction with the name register to locate and access any of the top four elements of the data stack. The SR counter is read by R-bus field code SR and modified by store field code PUSH and special field codes INSR, DCSR, POPA, CLSR and POP, as described in the field code definitions.

3-110. PROGRAM BASE REGISTER. The program base (PB) register is a 16-bit register that contains the absolute address of the bottom location of the current program segment. The PB register is read by S-bus field code PB and loaded by store field code PB as described in the field code definitions.

3-111. DATA LIMIT REGISTER. The data limit (DL) register is a 16-bit register which contains the absolute address of the bottom useable location in the current data stack. The DL register is read by S-bus field code DL and loaded by store field code DL as described in the field code definitions.

3-112. STACK MEMORY REGISTER. The stack memory (SM) register is a 16-bit register which contains the absolute address of the top element of the data stack in memory. Depending upon the number of TOS registers in use (reflected by the contents of the SR register), this address can be from zero to four locations below the actual top of stack. The SM register is read by S-bus field code SM and loaded by store field code SM as described in the field code definitions.

3-113. DATA BASE REGISTER. The data base (DB) register is a 16-bit register and is one of the stack limit registers. The DB register contains the absolute address of the first location of directly addressable storage in the current data stack. The DB register is read by S-bus field code DB and loaded by store field code DB as described in the field code definitions.

3-114. Q REGISTER. The Q register is a 16-bit stack marker register which contains the absolute address of the current stack marker being used within the data stack. The Q register is read by S-bus field code Q and loaded by store field code Q as described in the field code definitions.

3-115. SCRATCH PAD 2 REGISTER. The scratch pad 2 (SP2) register is a 16-bit register used by the CPU to store partial results during various microprogram routines. The SP2 register is read by S-bus field code SP2 and loaded by store field code SP2 as described in the field code definitions.

3-116. SCRATCH PAD 3 REGISTER. The scratch pad 3 (SP3) register is a 16-bit register used by the CPU to store partial results during various microprogram routines. The SP3 register can be shifted right and provides serial data input to bit 0 and output from bit 15. The SP3 register is read by S-bus field code SP3, loaded by store field code SP3, and shifted by function field codes CTSD, MPAD, and TASR as described in the field code definitions.

3-117. PROCESS CLOCK REGISTER. The process clock (PCLOCK) register is a 16-bit counter. The process clock is loaded and read by software instructions and is continuously incremented as long as the CPU is not executing on the Interrupt Control Stack (ICS FLAG= 0) or halted. The clocking interval is 1.001 ms. The maximum range of the clock before rollover is approximately 65.5 seconds.

3-118. PROGRAM COUNTER REGISTER. The program counter (P) register is a 16-bit register which contains the absolute address of the next program instruction to be fetched from memory. During execution of a skip field code NEXT, PB-bank and P are used to select a memory module and prefetch the instruction following the one which is about to be executed. The P register is read by S-bus field code P and loaded by store field code P as described in the field code definitions.

3-119. OPERAND REGISTER. The operand (OPND) register is a 16-bit register that provides storage for data read from memory by the CPU. The operand register is loaded by an OPINP signal from the Operand In Process (OPINP) flip- flop in the MCU operation decoder as a result of MCU options OPND, RNWA, RWA, and RWAN. The operand register is read by an RDOPND signal from the S-bus decoder as a result of an S-bus field code OPND as described in the field code definitions.

When the CPU freezes for an operand, the operand from memory goes directly to the S-bus logic as well as into the operand register. It is then loaded into the S-bus register to await CPU operation.

3-120. STATUS REGISTER. The status register is a 16-bit register which indicates the current status of the CPU hardware. The function for each bit is as follows:

a. Bit 0. Privileged mode bit.

1 = Privileged mode.
0 = User mode

b. Bit 1. External interrupts.
Includes module interrupts and console interrupts (if in DISPATCH).

1 = Enable.
0 = Disable.

c. Bit 2. User traps.

1 = Enable.
0 = Disable.

d. Bit 3. Stack op B.

1 = Pending.
0 = Not pending.

e. Bit 4. Overflow bit.

f. Bit 5. Carry bit.

g. Bits 6:7. Condition code.

00 = CCG
01 = CCL
10 = CCE

11 = Note that on this implementation of the CPU, a "11" stored in bits 6:7 acts as though the condition code is both CCE and CCL. This should not be used and is presented for information only.

h. Bit 8:15. Current executing code segment number.

The status register is read by S-bus field code STA and loaded by store field code STA as described in the field code definitions. Status bits also are affected by function field codes CADO, SUBO, INCO, ADDO; and special field codes CCB, SCRY, CCRY, POPA, SOV, CLO, CCZ, CCL, CCG, CCE, and CCA.

3-121. COUNTER REGISTER. The counter (CNTR) register is a six-bit register used as a repeat counter by the CPU. The two's complement of the desired count is loaded into the CNTR register; the CNTR register is then incremented for each repeated execution until it contains all ones as indicated by a Counter Maximum (CTRM) code from the skip field. The CNTR register is affected or referenced by S-bus field codes CTRI and CTRH, function field code REPN, store field codes CTRL and CTRH, special field code INCT, and skip field code CTRM as described in the field code definitions.

In addition to the above function, the CNTR register saves the content of the SR register when the CPU is put in the halt mode. This is useful because the halt microroutine pushes all data stack elements into memory thus forcing the SR register content to zero. The CNTR register can then be displayed to show what the content of the SR register was just prior to the halt.

3-122. OVERFLOW. The Overflow flip-flop controls the overflow bit (bit 4) of the status word, and stores the state of the Overflow signal from the ALU, when OFCENB signal is true. The Overflow flip-flop is set by special field code SOV (Set overflow) and cleared by the special field code CLO (Clear Overflow).

3-123. CARRY. The Carry flip-flop controls the carry bit (bit 5) of the status word, and stores the state of the Carry signal from the ALU, when OFCENB signal is true. The Carry flip-flop is set by special field code SCRY (Set Carry) and cleared by the special field code CCRY (Clear Carry).

3-124. CONDITION CODE LOGIC. The condition code logic controls the condition code. Bits 6 and 7 of the status word are used for the condition code. Although several instructions make use of the condition code, the condition code typically indicates the state of an operand (or a comparison result with two operands). The operand may be a word, byte, double word, or triple word, 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 special interpretations, there are basic patterns for interpreting these codes. The three patterns are shown in table 3-1.

3-125. PRE-ADDER. The pre-adder is used to gain a speed increase for instructions which use or perform computations on bits in the CIR. For example, when executing indexed memory reference instructions (and not indirect), the proper displacement field of the CIR 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 (PB, DB, Q or Z).

3-126. R-BUS REGISTER. The R-bus register is a 16-bit register that provides buffer storage between the R-bus and the ALU. The R-bus register can be shifted left one bit position (see function field code QASL). The R-bus register is loaded from the R-bus. (Refer to R-bus field code definitions.)

3-127. S-BUS REGISTER. The S-bus register is a 16-bit register that provides buffer storage between the S-bus and the ALU. The S-bus register can be shifted right one bit position (refer to function field code QASR). The S-bus register is loaded from the S-bus. (Refer to S-bus field code definitions.)

3-128. ARITHMETIC LOGIC UNIT. The arithmetic logic unit (ALU) combines the R- and S-bus data and generates functions that are divided into two modes (or groups): arithmetic functions and logic functions. The output of the ALU is placed on the T-bus to go to either the T-bus shifter or the decimal corrector.

3-129. SHIFTER. The 16-bit output of the ALU function generator (the combined R- and S-bus data) is applied to the T-bus shifter. All shifts and rotates of the T-bus (left shift, right shift, right-left swap, etc.) are executed as directed by the shift field decoder. The output of the T-bus shifted is placed on the U-bus to be stored in one of the U-bus registers.

3-130. DECIMAL CORRECTOR. The decimal corrector takes the ALU output (already added) and adds "6" to each group of 4 bits, and generates carries to the next group as appropriate to yield a correct decimal addition. The shifter is disabled. Each group of 4 bits in the source operands must be in the range of from 0 to 9. If an invalid digit is detected during the add cycle, overflow will be true.

3-131. ACOR. The address computer output register (ACOR) is a 16-bit register that functions as a memory address buffer between the U-bus and the central data bus.

3-132. DCOR. The data computer output register (DCOR) is a 16-bit register that functions as a buffer for memory bound data and operand address transfer between the U-bus and the central data bus.

3-133. CPX1 REGISTER. The CPX1 register provides 16 bits that are used to monitor the system run mode interrupt status. When a run mode interrupt is experienced, the CPU reads the CPX1 register and checks the content for the cause of the interrupt. The S-bus field code CPX1 reads the CPX1 register and the special field code CCPX affects the CPX1 register as described in the field code definitions. The following is a list of the CPX1 register contents with the significance of each bit:

Bit 0: Integer overflow
Bit 1: Bounds violation
Bit 2: Illegal address
Bit 3: CPU timer
Bit 4: System parity error
Bit 5: Address parity error
Bit 6: Data parity error
Bit 7: Module interrupt
Bit 8: External interrupt
Bit 9: Power fail interrupt
Bit 10: 0
Bit 11: ICS flag
Bit 12: DISP flag
Bit 13: Emulator
Bit 14: I/O timer
Bit 15: Option present

3-134. CPX2 REGISTER. The CPX2 register provides 16 bits that are used to monitor the system halt mode interrupt status. When a halt mode interrupt occurs, the CPU reads the CPX2 register and checks its contents for the cause of the interrupt. The S-bus field code CPX2 reads the CPX2 register and the special field code CCPX affects the CPX2 register content as described in the field code definitions. The following is a list of the CPX2 register content with the significance of each bit:

Bit 0: Run switch
Bit 1: Dump switch
Bit 2: Load switch
Bit 3: Load register
Bit 4: Load address
Bit 5: Load memory
Bit 6: Display memory
Bit 7: Single instruction
Bit 8: Execute switch
Bit 9: Increment address
Bit 10: Decrement address
Bit 11: 0
Bit 12: 0
Bit 13: Inhibit PFARS
Bit 14: System halt
Bit 15: Run flip-flop

MEMORY SEGMENTATION

SECTION IV

4-1. INTRODUCTION

Many programs may concurrently access an HP 3000 Series II or Series III Computer System and share its main memory through the technique of *memory segmentation*. As part of its basic architecture, the system organizes programs into variable length segments of code and data which may be swapped in and out of main memory on demand. The programmer, therefore, works with a memory that appears to be many times larger than the actual physical size. In fact, a single program may exceed the main memory capacity, and still allow space for many other users on the same machine. The system is said to have a *virtual memory* since it can concurrently execute a total address space many times larger than its main memory capacity.

4-2. VIRTUAL MEMORY

Virtual memory is a very efficient memory management scheme which in addition to main (semiconductor) memory uses disc storage as secondary memory. Users' program code and data are divided into variable-length segments which reside in secondary memory. As a program is executing, only those segments of code and data which are required at a particular time actually reside in main memory; the other related segments remain on disc until they in turn are required. When a particular code segment is no longer needed, it is simply overlayed by another segment. This can be done because the code segments are non-modifiable and re-entrant. When the segment is needed again, it is simply copied from the disc on which the program resides. Since programs are copied into main memory directly from the disc on which they reside, they need not be copied prior to execution to a special "swapping disc." Data segments, however, are dynamic and their contents can change during execution. Therefore, when a particular segment is no longer needed, it is automatically copied back to the system disc (replacing the previous version of that segment on the disc) and the main memory space of that segment is then available for other segments. The process of transferring segments between secondary memory and main memory is referred to as *swapping*. In this way, many users may have very large programs executing concurrently with one another and with the operating system. The operating system itself and the various language and utility programs are handled in exactly the same manner. The virtual memory scheme uses a working set table to implement an advanced "segment trap frequency" replacement algorithm designed especially for the HP 3000 Series II and III Computer Systems.

4-3. PROCESSES

Programs are run on the basis of *processes* created and handled by the operating system. The process is not a *program* itself, but the unique execution of a program by a particular user at a particular time. *Thus, a process is an execution of a program*. 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*. A process consists of a Process Control Block (PCB) 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. Thus, while a *program* consists of data in a file, and instructions not yet executable, a *process* is an executing program with the data stack assigned. The code segments used by a process can be shared with other processes, but its data stack is private

(figure 4-1). For example, each user working on-line through the BASIC language is running his program under a separate process; all use the same code (the only copy of the BASIC interpreter in the system) but each has his own data stack.

Processes are invisible to the programmer. In other words, the programmer has no control over processes or their structure. The user with certain optional capabilities, however, can create and manipulate processes directly.

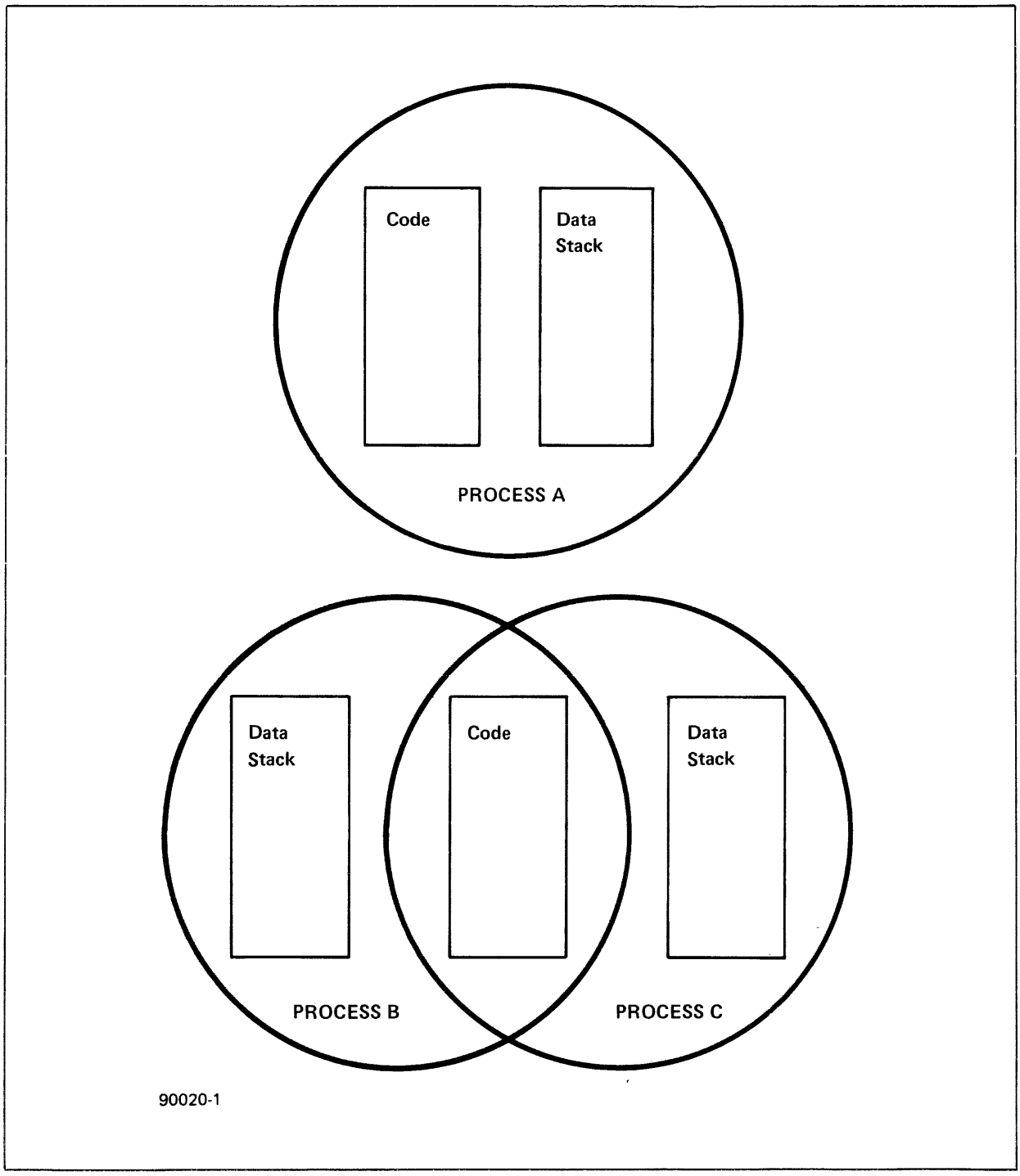


Figure 4-1. Code Sharing and Data Privacy

4-4. PROCEDURES

In the Multiprogramming Executive (MPE) operating system most individual programming operations are handled by unique sets of code known as procedures. (Others are handled by special programs or interrupt routines.) In Systems Programming Language (SPL), the language in which MPE is coded, a procedure is defined by a procedure declaration consisting of:

- A *procedure head*, containing the procedure name and type, parameter definitions, and other information about the procedure
- A *procedure body*, containing executable statements and data declarations local to the procedure.

As part of their function, many procedures also return parameter values to the processes that invoke them. The subprograms and subroutines of FORTRAN, COBOL, etc. are compiled into procedures automatically.

In SPL, each procedure is executed by a corresponding *procedure call*. When a procedure call is encountered, control is transferred to the procedure, which runs until an exit is encountered. Control is then returned to the statement following the call for that procedure.

In addition to the procedures provided by MPE, SPL allows the user to write procedures to suit his own purposes.

4-5. SYSTEM LIBRARY

The MPE system library is a collection of frequently-used routines that can be readily shared (among many users). A library code segment consists of one or more procedures. Some of these segments reside permanently in virtual memory, while others are stored elsewhere on disc and called into virtual memory as needed.

4-6. MEMORY MANAGEMENT

The memory management system is responsible for the allocation of main memory to executing processes. Program and Library segments which are needed for execution are automatically swapped into main memory from disc. A *segment* is the basic entity for swapping transfers. (See figure 4-2). When a segment in main memory falls into disuse, this is detected and such an unused segment becomes a candidate for automatic overlay by the system. An attempt by an executing process to access code or data not present in main memory will cause an *absence trap*. This internal interrupt (see Section VI) causes the memory management software to allocate main memory space for the missing segment. When the absent segment is swapped from the *disc memories* to the *main memory* (is said to become *present* in main memory), the executing process will continue.

The computer system's virtual memory management keeps track of segment usage. Frequently used segments may never be swapped, but will remain in main memory, while rarely used segments will be in disc memory most of the time. This results in higher efficiency and faster overall execution time.

At this point you should be visualizing a dynamic situation in which segments are being swapped rapidly between main memory and the disc memories, according to the demands of the executing programs. Also bear in mind that many users may be executing the same or different programs at any given time.

4-7. CODE SEGMENTATION

In the HP 3000 Series II and III Computer Systems, code is grouped into logical entities consisting of one or more procedures (subroutines). Each segment may be up to 32K bytes long (K = 1024). Programs may optionally be broken into multiple segments. Procedures or subroutines must be fully contained within one segment.

The steps in segment operation are illustrated in figure 4-3. First, source code is translated into a *User Subprogram Library (USL)*. By either embedding control statements in the source code, or by direct manipulation of the USL using the MPE Segmenter, the procedures in the USL can be organized into independent code segments. This is done automatically by some compilers.

The USL may be prepared into a *program file* for execution. Alternatively, segments in a USL may be added to *Segmented Libraries*. Segmented Libraries permit separate programs to share procedures. Segmented Library segments are swapped directly from the library file on disc, independent of which program absence trapped on the library procedure. (We are ignoring possible security restrictions on files which may be imposed through the operating system.) Procedures in segments in a program file are swapped directly from the program file on disc.

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 instruction in a code segment, and in no way is it possible to append to or add onto a code segment once it is executing from a program file or Segmented Library. It is this feature which ensures that all system code is *re-entrant*, meaning that any sequence of instructions can be in simultaneous execution by multiple users. All computer system procedures are potentially recursive.

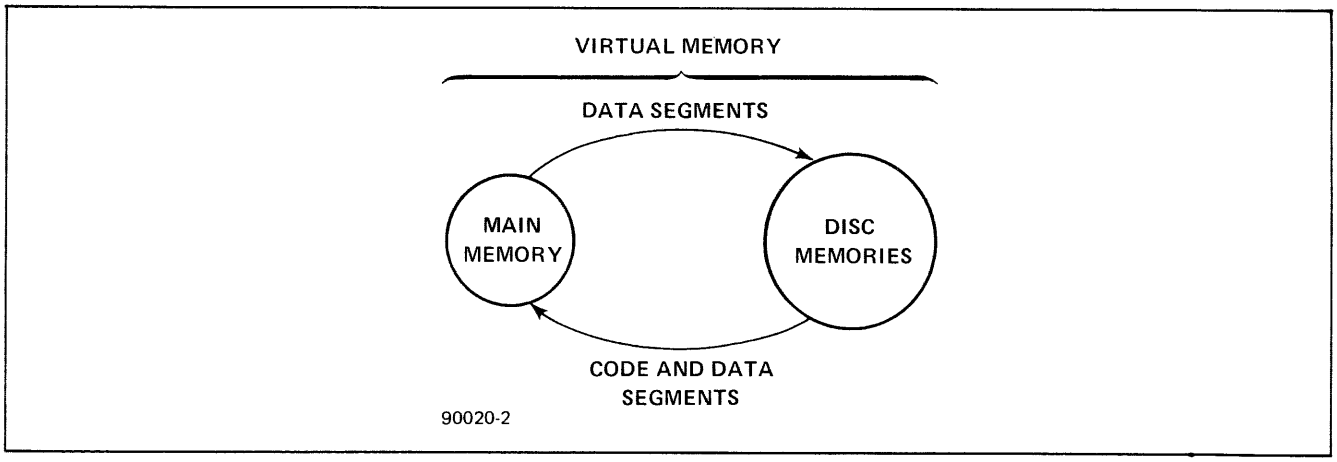


Figure 4-2. Segmented Memory Management

That system code segments are not modified during execution has specific advantages for the memory management system. Since all code segments are re-entrant, all are potentially shared by more than one user when present in main memory. For example, operating system services are provided in part by procedures (segments) contained in a *System Segmented Library (SSL)* shared by all programs requesting those services. Similarly, all users executing the same program file share the program's segments. In fact, only one copy of a segment needs to be in main memory, no matter how many users may be executing it concurrently. The result is an ability to handle more users in a given main memory capacity.

The second major advantage of code segment re-entrancy is that code segments are consequently *read only*. Thus, they need never be swapped from main memory back to disc, even when overlayed, because there is always an identical copy of the segment in the program file or library, from whence it may be fetched whenever needed. Hence, code is only swapped into main memory, never out, as is shown in figure 4-2. The resulting reduction in swap traffic is conducive to efficient management of main memory.

4-8. DATA SEGMENTATION

When a program file is selected for execution by a user, it is allocated a private, non-sharable *data segment*. Space for the data segment is reserved on the system disc and initialized as specified by information contained in the program file. The system maintains an up-to-date copy of all data segments on disc when they are not required in main memory.

The process (paragraph 4-3) is the basic executable entity on the system. In order for a process to execute, its data segment and code segment containing the procedure currently in execution by the CPU must be present in main memory.

The system is also capable of operating in a *split-stack* mode, whereby the DB register points to the current extra data segment, while the other stack registers continue to point to the stack data segment. (The CPU registers are defined in Section III.) This is particularly efficient for system routines with tables in system data segments. In split-stack mode, these data segments can be accessed relative to the DB-register while using the other stack registers for computation.

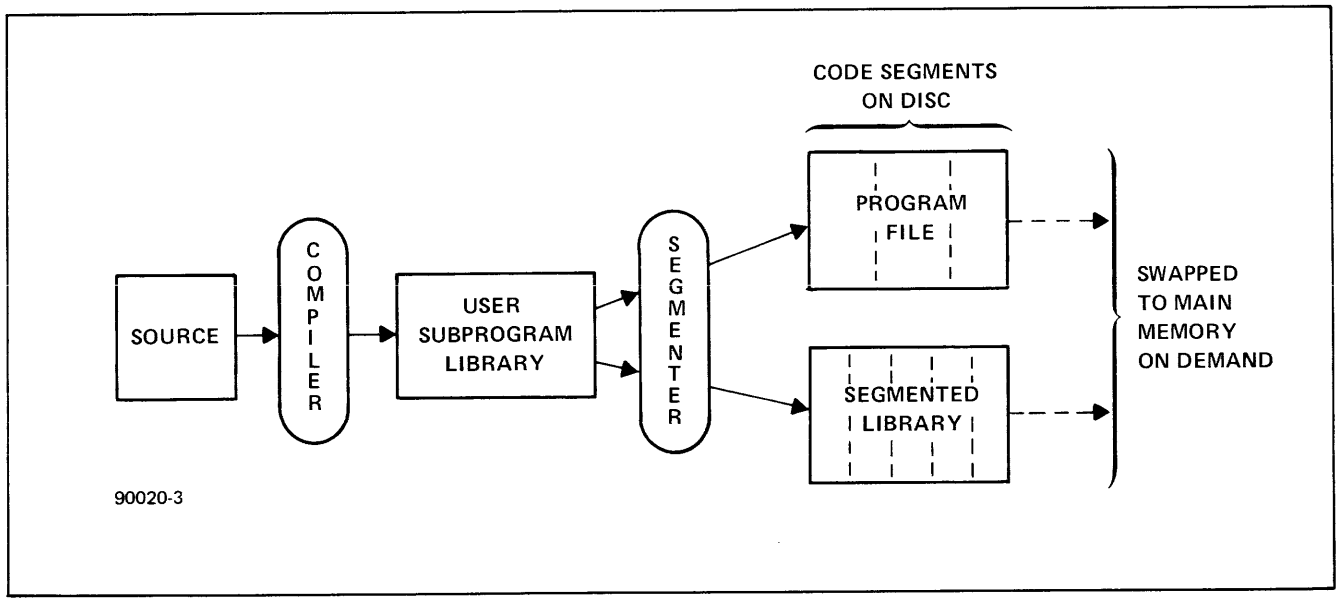


Figure 4-3. Code Segment Evolution

Data segments separate from the stack may be obtained dynamically during process execution. Data segments may also be expanded and contracted by the operating system as necessary. This includes system handling of the stack overflow internal interrupt (see Section VI), in which the data segment may automatically be expanded to a maximum of 32K words to accommodate the operation of the stack.

The system, in addition to the split-stack mode, contains instructions for data movement between data segments. These instructions cause an *absence trap* if either of the required data segments is not present. The operating system thus has efficient means for accessing very large address spaces outside of stacks, and can provide buffering and other data storage facilities, relieving the process of having to reserve space for these functions within the stack data segment. The addressable data space is thus effectively much larger than the 64K maximum data segment size.

4-9. MAIN MEMORY ORGANIZATION

The main memory is composed of up to four (Series II) or 16 (Series III) banks of high-speed, semiconductor, random access memory. Each bank contains 64K words, yielding a system main memory of up to 256K words (Series II) or 1024K words (Series III). The memory banks are grouped into two modules; a lower memory module and an upper memory module. See figure 2-1. The main memory has the following operating modes and specifications:

- WRITE: 700 nsec minimum cycle time.
- READ: 350 nsec access, 700 nsec cycle time.
- READ WRITE ONES (RW1): 350 nsec access, 1050 nsec cycle time. (Series II only.)
- NO OPERATION (NOP): 700 nsec cycle time.

Operating power for main memory's semiconductors, refresh circuits, and power fail circuits is supplied by a rechargeable battery pack in Semiconductor Memory Power Supply, part number 30311-60001. Each power supply can support 256K words of main memory on a Series II and 512K words on a Series III. Two power supplies are required to support any main memory greater than 256K words (Series II) or 512K words (Series III). In addition, the power supply maintains the battery charge. Power is available from the battery whenever AC line power is removed or lost and whenever the system is placed in the D.C. STANDBY operating mode. Battery power can be used for approximately 40 minutes (minimum) to maintain memory data.

The memory module(s) interface with the other system modules via the central (CTL) data bus. Other modules on this bus may request transfers of data to or from the memory module(s). Operation of main memory with the other modules on the CTL bus is controlled by a Module Control Unit (MCU) located within main memory.

4-10. BASIC TABLE STRUCTURES

The first few locations of main memory are reserved for system pointers as listed in table 4-1. Of particular interest to memory segmentation are those diagrammed in figure 4-4. Memory location 0 is set at system cold load to point to the location of the *Code Segment Table (CST)* (see (1) in figure 4-4). This table contains a single 4-word entry for each Segmented Library segment currently in use in the system.

Memory Location 1 ((2) in figure 4-4) points to the *Code Segment Table Extension (CSTX)* area allocated to the program being executed by the CPU. The Code Segment Table Extension is used to keep track of the code segments in the various program files being executed. Thus, the contents of memory location 1 will shift to point to various sections of the CST Extension as different programs are executed by the CPU. In figure 4-4, program X is currently being executed by user process A.

Memory Locations 2 and 3 are fixed at cold load time and point to the Data Segment Table (3) and Process Control Block Base (4), respectively. There is a four-word DST entry for each data segment in use in the system (see figure 4-6). There is a Process Control Block allocated to each process running in the system. The PCB entry for a process points to the DST entry for its stack data segment, but for simplicity this is not shown in figure 4-4.

Memory Location 4 is set by the software to point to the PCB of the currently executing process (5). The linkage from the PCB to the CST Extension area (6) is used to set memory location 1 2 . Notice that if, as illustrated by (7), processes B and C happen to be executing the same program, the program file segments will be shared. The CPU status register then points to the current segment of the current process holding CPU resources.

4-11. CODE SEGMENT TABLE AND EXTENSION

The Code Segment Table contains a list of code segments that are being referenced by executing programs. Its length is determined at system generation time. The actual number of entries in use at any time is a variable, limited only by the length of the table. Entries are dynamically allocated by the operating system as programs are loaded and unloaded. Each entry contains control information about the segment, and gives its length and starting address. (See figure 4-5).

Table 4-1. Low Main Memory Location Reservations

LOCATION	CONTENTS
0	Code Segment Table Base
1	Code Segment Table Extension
2	Data Segment Table Base
3	Process Control Block Base
4	Current Process Control Block
5	Interrupt Stack Base
6	Interrupt Stack Limit
7	Interrupt Mask
%10-%13	Reserved
%14-%777 (max.)	Device Reference and External Interrupt Table
%1000	System Global Table (Pointers to resident tables, etc.)

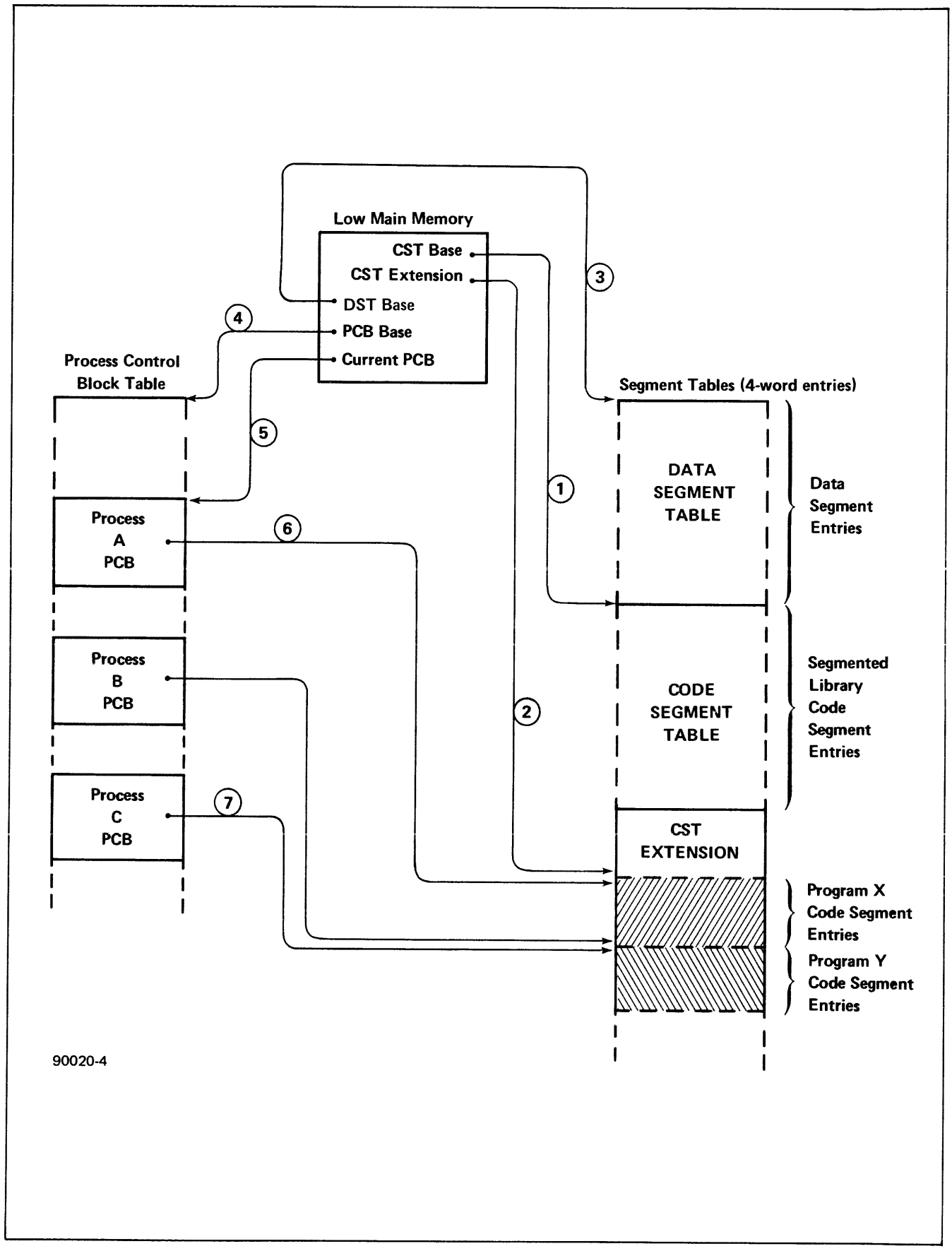


Figure 4-4. Basic Data Structures

The first %300 entries are reserved for Segmented Library segments. The CST entry for segment 0 contains control information. Segment 1 contains the routines needed to service internal interrupts (see Section VI). Segments 2 through 191 (%277) contain code such as service routines for external interrupts, system intrinsics, and library procedures.

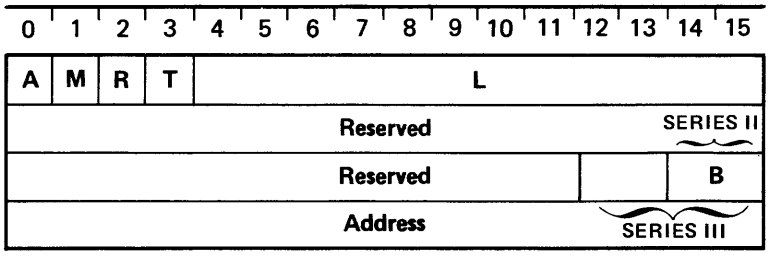
The remainder of the table entries fall in the CST Extension table and keep track of program segments. Each program may have up to 63 segments.

Accessing the table is done via the PCAL, EXIT, IXIT, and DISP instructions and is completely invisible to the user. For example, if on PCAL, an external segment number lies in the range 0<Seg.No.<%300, the entry is accessed via memory location 0. But should %300<Seg.No.<%400 then memory location 1 is used by the microcode to access the Extension area, using the Seg.No. %300 as the index into the block of CST Extension entries for that program.

4-12. DATA SEGMENT TABLE

The Data Segment Table contains a list of the various data segments currently in use by the operating system and user programs. These segments include I/O buffers, system and user process stacks, and extra data segments. Its length is determined at system generation time, and it contains a four-word entry for each segment (see figure 4-6). The actual number of entries in use at any one time is a variable, limited only by the length of the table. Entries are dynamically allocated by the system as programs are initiated or terminated, or special capability processes request or release additional data segments.

The data segment number zero is unused by the system so its DST entry is reserved for control information.



A = Absence Bit = 1 if segment is absent from main memory.
M = Mode Bit = 1 if segment executes in Privileged Mode (Code only).
R = Reference Bit = 1 if segment has been referenced (set by microcode).
T = Trace Bit = 1 if trace feature is used. Checked by PCAL instruction.
L = Length Field = segment length divided by 4.
B = Bank Address. Points to memory bank (if resident in main memory) in which segment resides.
ADDRESS = Absolute address of PB within B if the segment is present, otherwise the 3rd and 4th words contain the absolute disc address.

90020-5

Figure 4-5. Code Segment Table Entry Format

4-13. CODE SEGMENT LINKAGE

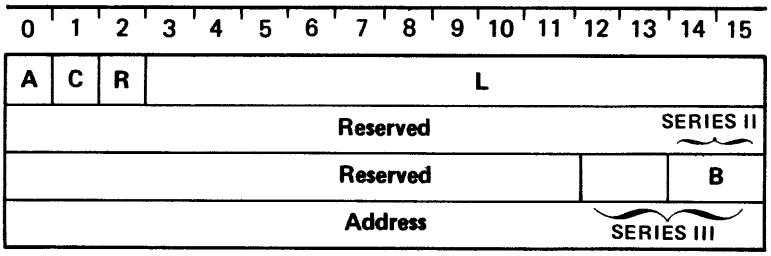
During the execution of one user process, there will usually be several code segments in memory and a single data segment. Assume that the current process presently has two code segments in memory, as shown in figure 4-7. (The data segment, not shown, is discussed later.)

The purpose of figure 4-7 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 MPE operating system to control the tables shown here. The Code Segment Table is used in the illustration.

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. If the segment is a program segment, location 1 is used.

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 four-word descriptor which includes the absolute address of the related segment and its length. (The format of CST entries is given in figure 4-5.) 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. Assume that one user is executing a process which requires code segments 22 through 25. Segments 22 and 23 are in main memory, since a reference has caused them to be brought in, whereas segments 24 and 25 are not presently needed, and therefore are absent on disc.

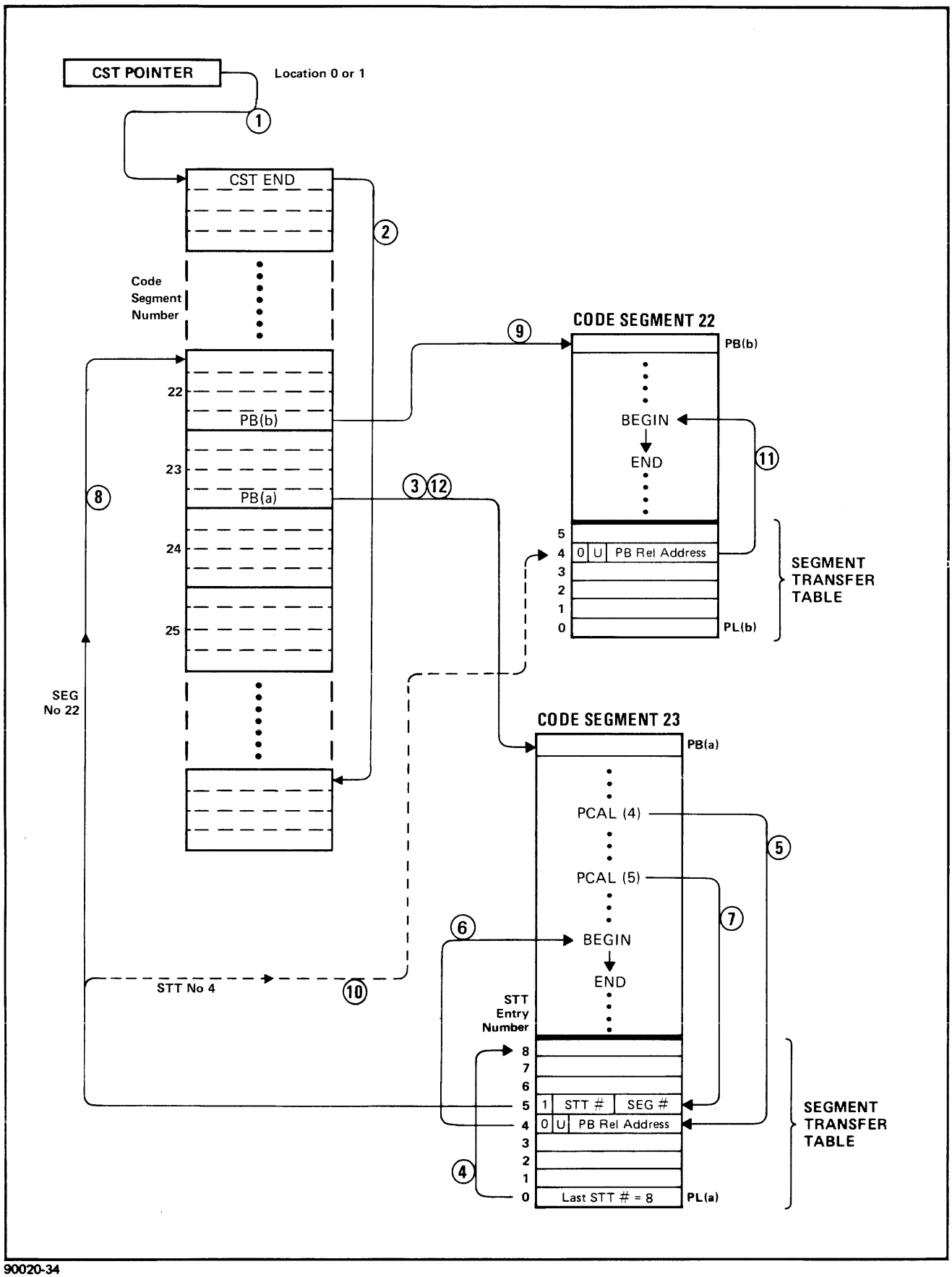
The process is currently executing instructions in segment 23. This means that the address value contained in the fourth word of CST entry 23 has been loaded into the PB register. Thus the PB register is pointing (3) at PB(a). The PL register, using a value derived from the segment length, is pointing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



A = Absence Bit = 1 if segment is absent from main memory.
C = Clean bit. Used to eliminate unnecessary output swapping.
R = Reference Bit = 1 if segment has been referenced (set by microcode).
L = Length Field = segment length divided by 4.
B = Bank Address. Points to memory bank (if resident in main memory) in which segment resides.
ADDRESS = Absolute Segment Address within given bank in third word of segment. If segment is absent, words 3 and 4 contain absolute disc address.

90020-6

Figure 4-6. Data Segment Table Entry Format



90020-34

Figure 4-7. Procedure Calls Within and Between Code Segments

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. These locations contain linking references for every *procedure call* in the Segment Transfer Table (STT) segment. A *procedure call* is an instruction which references a set of instructions elsewhere in the code segment. This set of instructions is structured as a procedure, to perform a standardized operation or computation and then return control to the instruction immediately following 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 4-8). This indicates (4) the number of the lest 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 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 (since the PCAL instruction uses PL- addressing, the instruction references cell PL-4). This location contains a local program label (see format in figure 4-8), 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.

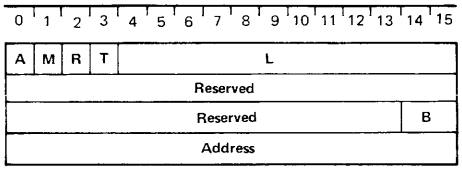
In this example there were no references outside the current segment. In the following example an external reference is made.

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 4-8). This implies that the called procedure is in some other segment. The contents of the label tell which segment, and also give 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 fourth 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 you transfer via the top-of-stack reference into another user's code segment (assuming that it is callable) you can do no harm by executing part of that other segment. You will render your own stack data meaningless, but you can in no way read or relocate the other user's code or data. The end result is completely unpredictable but would likely invoke one of many possible error traps.

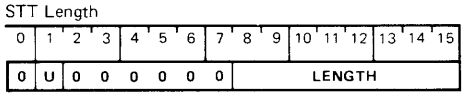
CODE SEGMENT TABLE Doubleword



A Absence Bit = 1 if segment is absent from main memory.
M Mode Bit = 1 if segment executes in Privileged Mode (Code only).
R Reference Bit = 1 if segment has been referenced (set by microcode).
T Trace Bit = 1 if trace feature is used. Checked by PCAL instruction.
L Length Field = segment length divided by 4.
B Bank Address. Points to memory bank (if resident in main memory) in which segment resides.
ADDRESS Absolute address of PB within B if the segment is present, otherwise the 3rd and 4th words contain the absolute disc address.

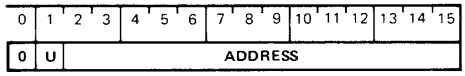
SEGMENT TRANSFER TABLE Words

STT Length



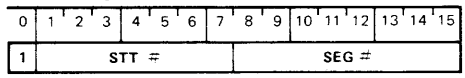
U Uncallable bit
LENGTH Maximum = 255 (Calls from external segments may reference only the first 127 entries, PL-1 thru PL-127.) (PL-0 = STTL.)

Local Program Label



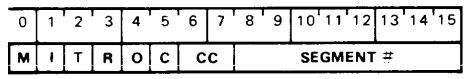
U Uncallable bit
ADDRESS PB relative, + only

External Program Label



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

STATUS Word



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

90020-7

Figure 4-8. 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 4-8), 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.

4-14. STACK OPERATION

Figure 4-9 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. 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 cause 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 4-10, 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 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.

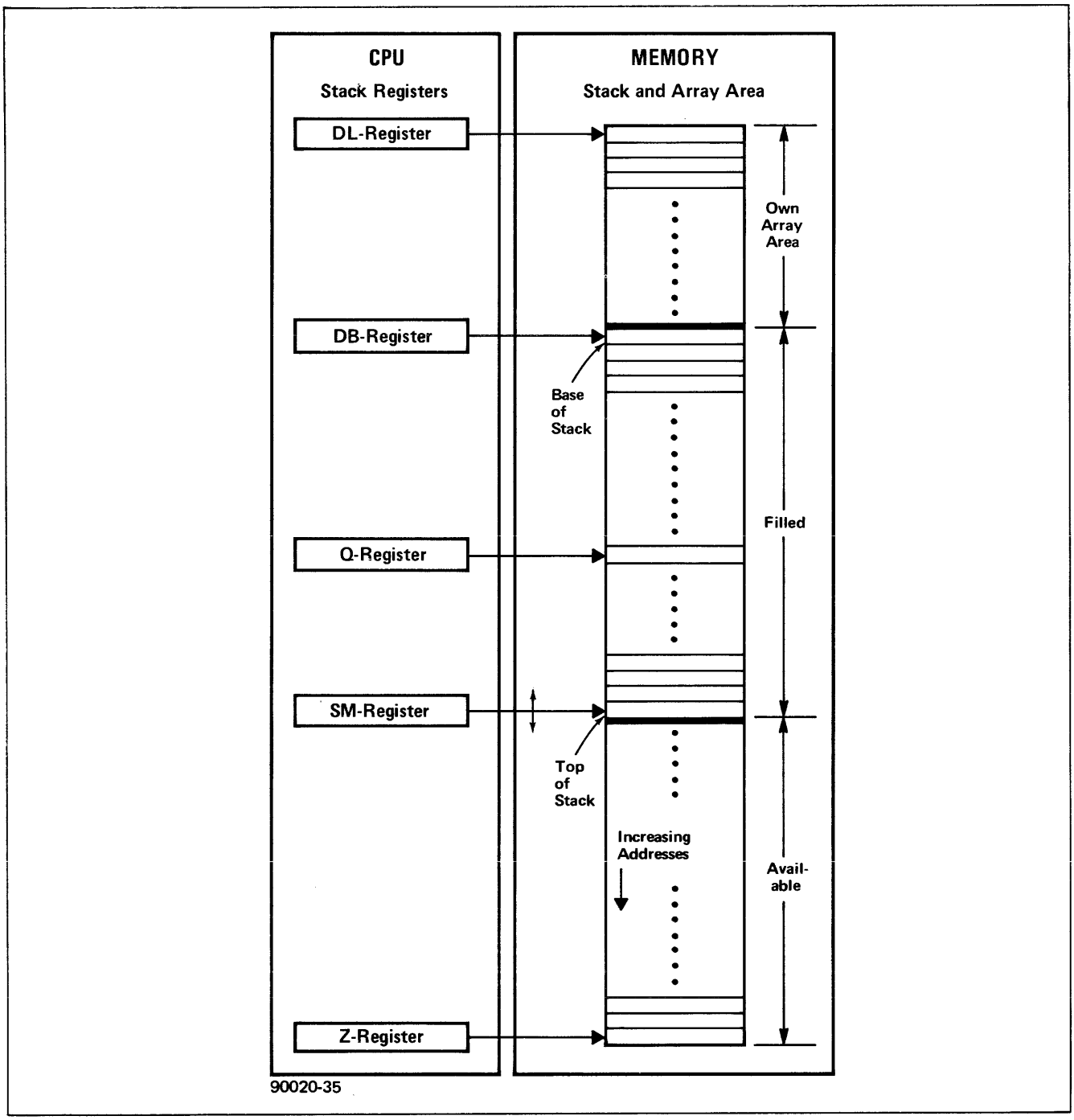


Figure 4-9. Stack Registers and One Stack

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.

The action of the Q register in marking the starting location for each procedure's data is shown in figure 4-11.

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

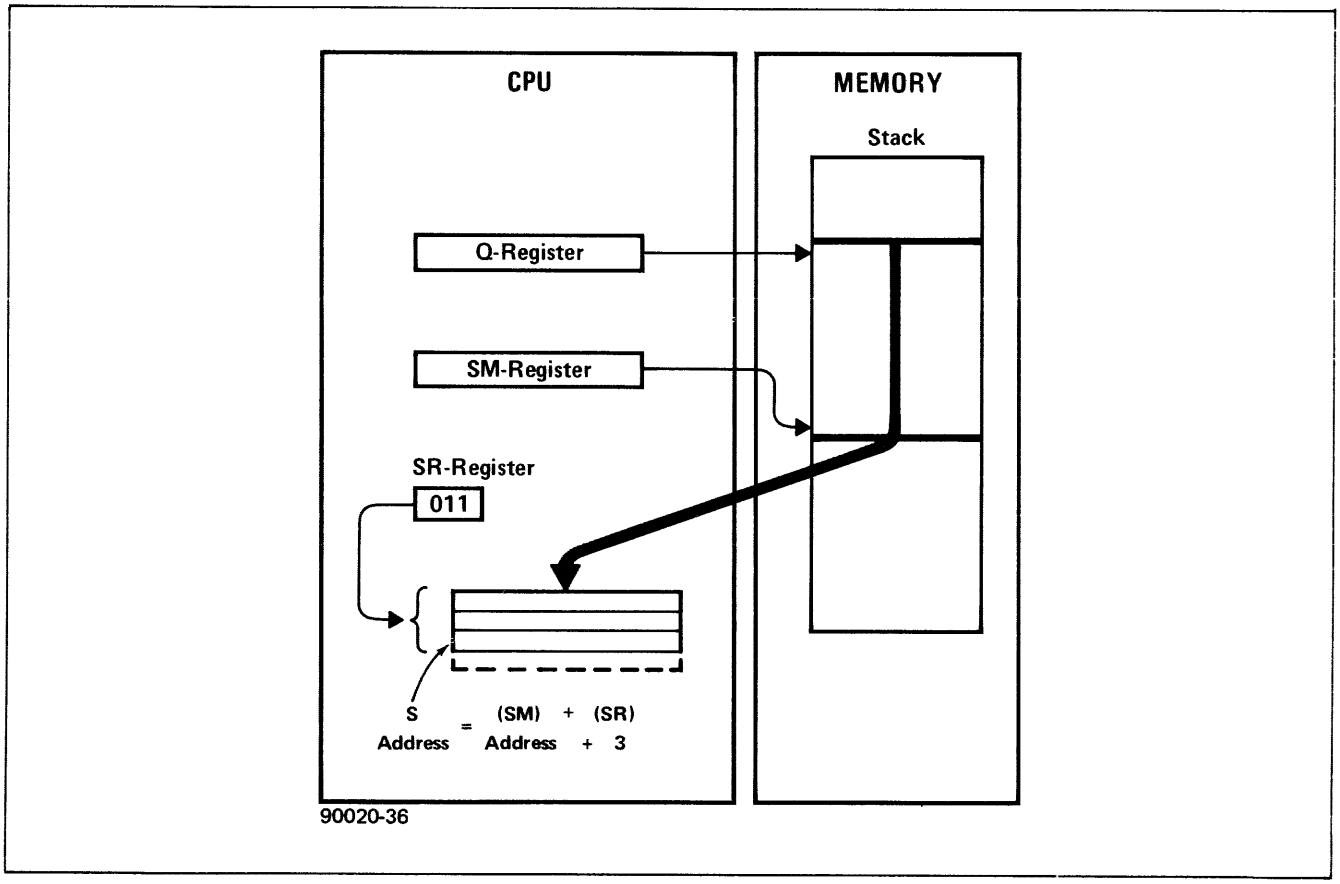


Figure 4-10. Top of Stack in the CPU

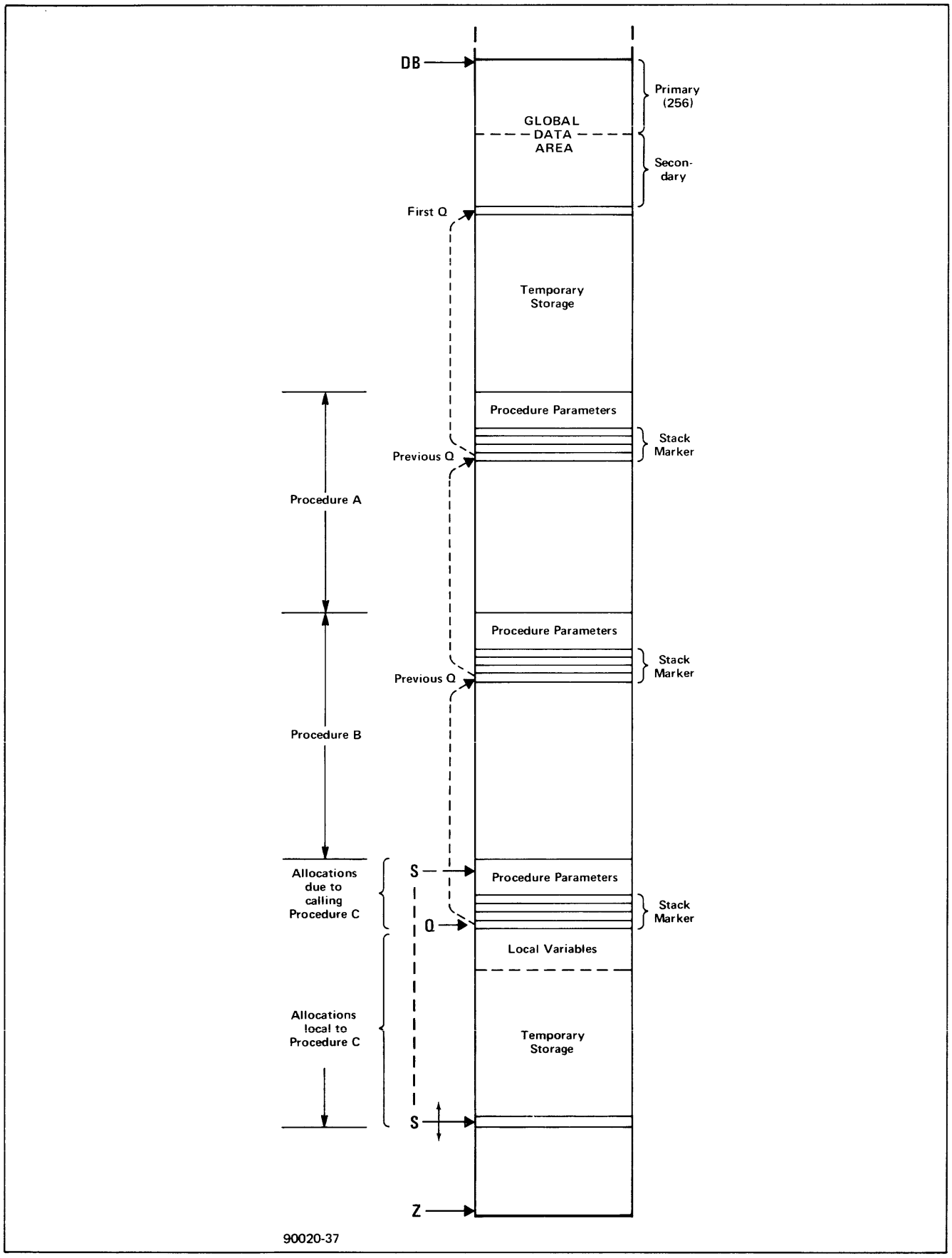


Figure 4-11. Stack Mark Chain

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. 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 marker chain virtually eliminates system overhead in keeping track of nested 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 either contains the value or is 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 also points at this location, since there is as yet no data on the stack. As the 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 are given in paragraph 4-15, "Examples of Stack Usage".)

At some time during execution of the code segment, it is assumed that procedure A is called. Accompanying the call are usually 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 4-12. The first word saves the current contents of the X register. The second word saves the return address for the code segment — 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. The fourth word is 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 (*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 answers computed by 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 the *Machine Instruction Set Reference Manual*).

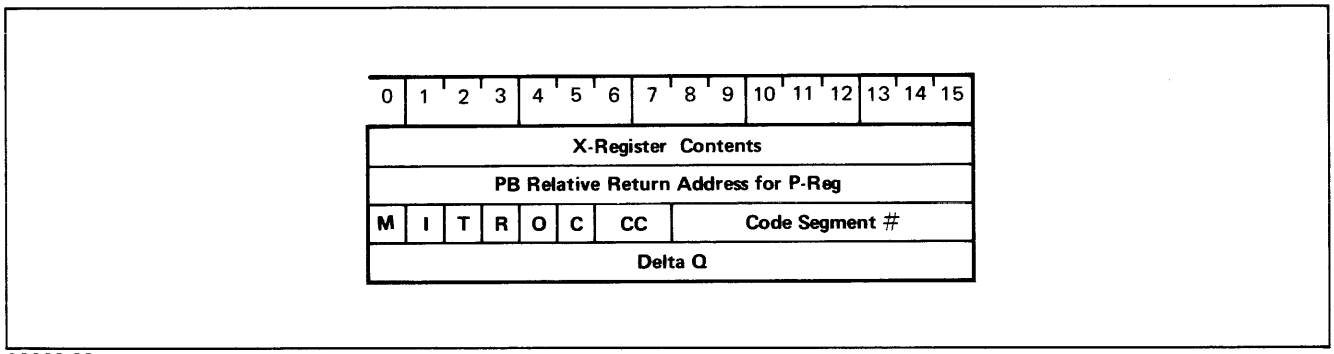
The sequence of events described in the preceding two paragraphs is repeated, until all procedure data and stack marks are eliminated, and only the final answer is on the top of the stack.

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*.

4-15. 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



90020-38

Figure 4-12. Standard 4-Word Stack Marker Format

nesting of procedures, and helps to make code re-entrant. Another major advantage, that it allows fast interrupt handling, is described in Section VI. The following examples are primarily designed to aid in understanding the stack concept.

4-16. BASIC ARITHMETIC

Figure 4-13 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.

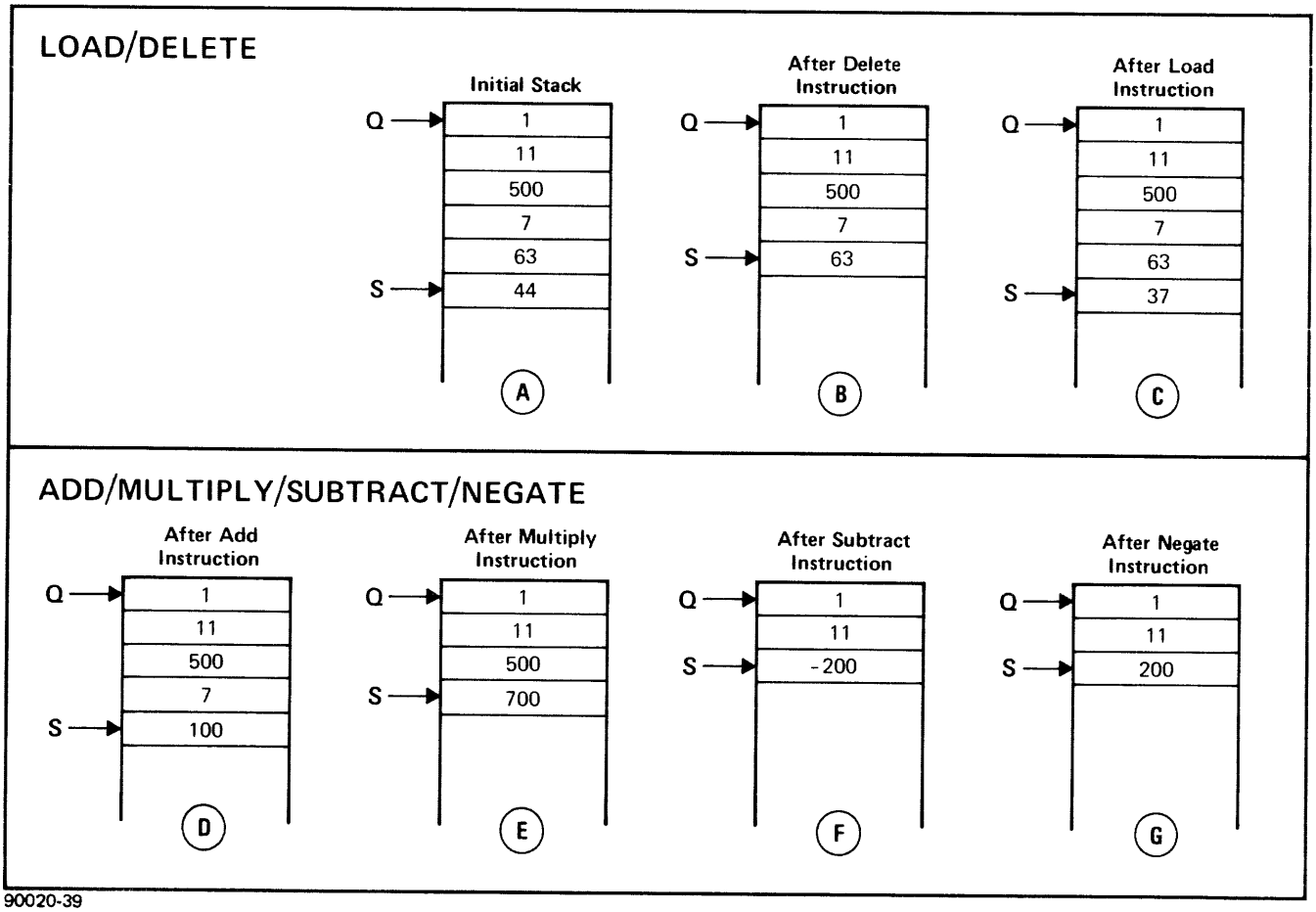


Figure 4-13. Basic Arithmetic Stack Operations

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 4-13 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 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.

4-17. PROCEDURE CALLS

Figures 4-14 and 4-15 illustrate the operations involved in a procedure call. Figure 4-14 shows programmatically how a procedure is set up and called, and figure 4-15 shows what happens to the stack when the procedure is called and executed.

SOURCE LANGUAGE

	1	BEGIN INTEGER ANSWER;
Procedure	2	INTEGER PROCEDURE QUOTIENT (J,K);
	3	VALUE J,K;
	4	INTEGER J,K;
	5	BEGIN
	6	QUOTIENT ← J/K;
	7	END;
Call	8	ANSWER ← QUOTIENT (25,10);
	9	END;

MACHINE LANGUAGE

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

MATHEMATICAL LANGUAGE

Procedure: ANSWER = J/K

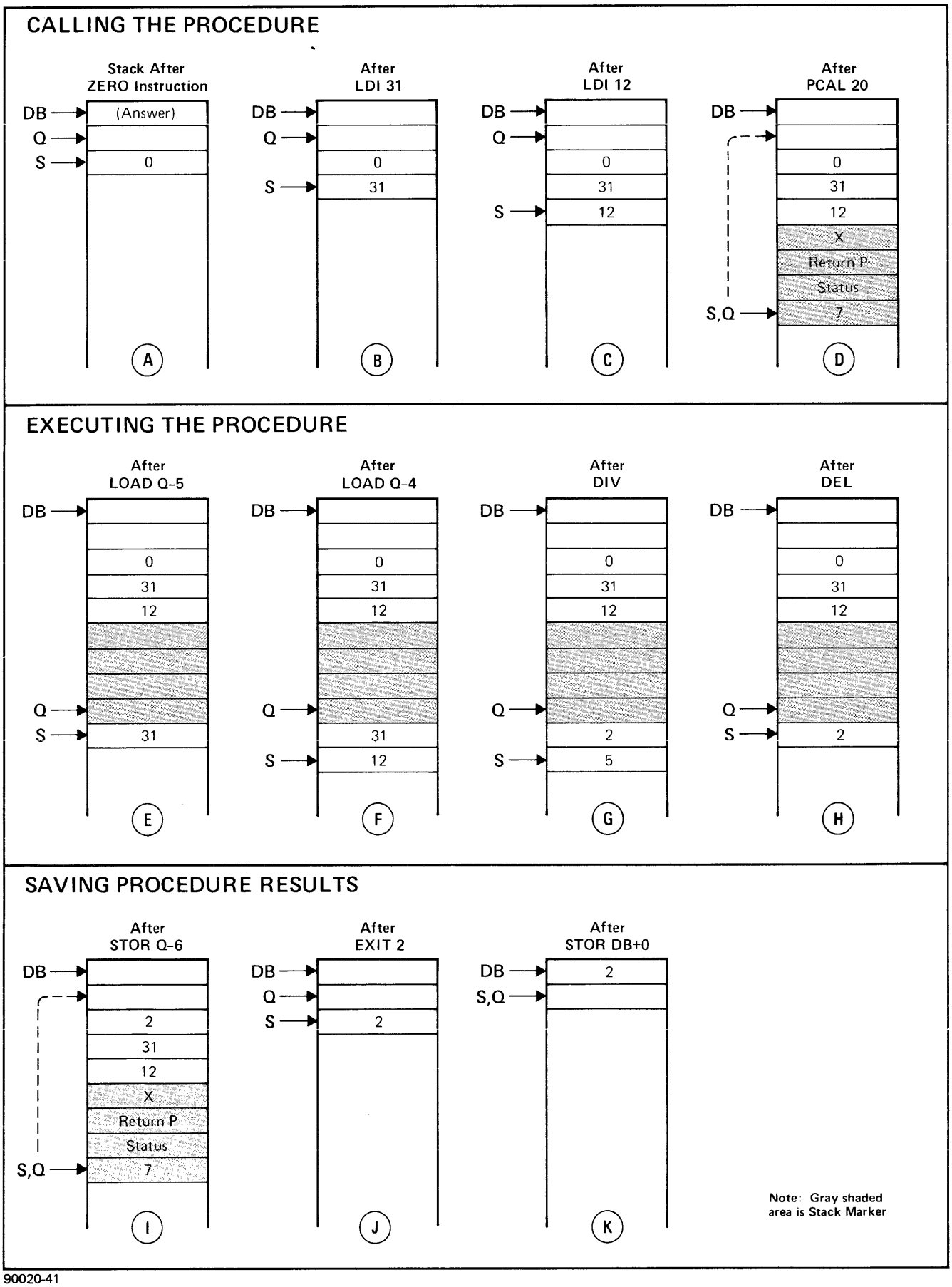
Call: Solve ANSWER for
J = 25 and K = 10

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

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

90020-40

Figure 4-14. Declaring and Calling a Procedure



90020-41

Figure 4-15. Executing a Simple Procedure

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 4-14), 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 4-14, 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 4-14 list two forms of the program that will accomplish the example procedure. The 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, but 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 line 10 through 14 to be generated, and line 8 causes lines 15 through 19 to be generated.

In order to explain the operation of the program in machine language, it is necessary to examine what is happening on the stack. Figure 4-15 is therefore referred to in the following discussions. Furthermore, to aid in visualizing the operations, they are described in chronological order; i.e., the machine language program begins execution 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 4-15). 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 4-11 and 4-9.

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 arrays are declared, DL is set coincident with DB, therefore DL is not shown.

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 4-14. 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 4-15). 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 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.

4-18. 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 4-14.) 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: $1 \times 2 \times 3 \times 4 = 24$.

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 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 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 4-16 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 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.

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

Figures 4-17 and 4-18 show the overall process of building up the stack by recursive calls, and then paring it down with recursive exits. These two figures are 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.

4-19. MAIN PROGRAM CALL

As before, the main program has already reserved global space for the final answer (Y) before 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 follow in that it contains return information to the outer block which called the present procedure. That is, the *return P* word is

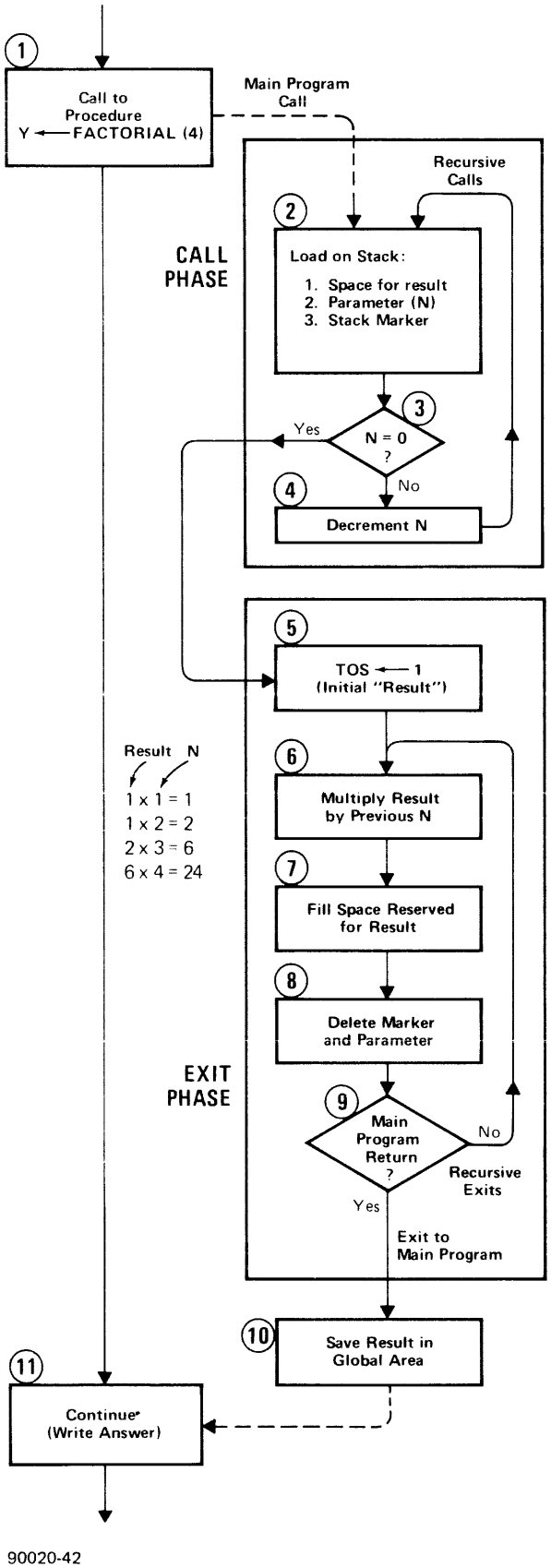
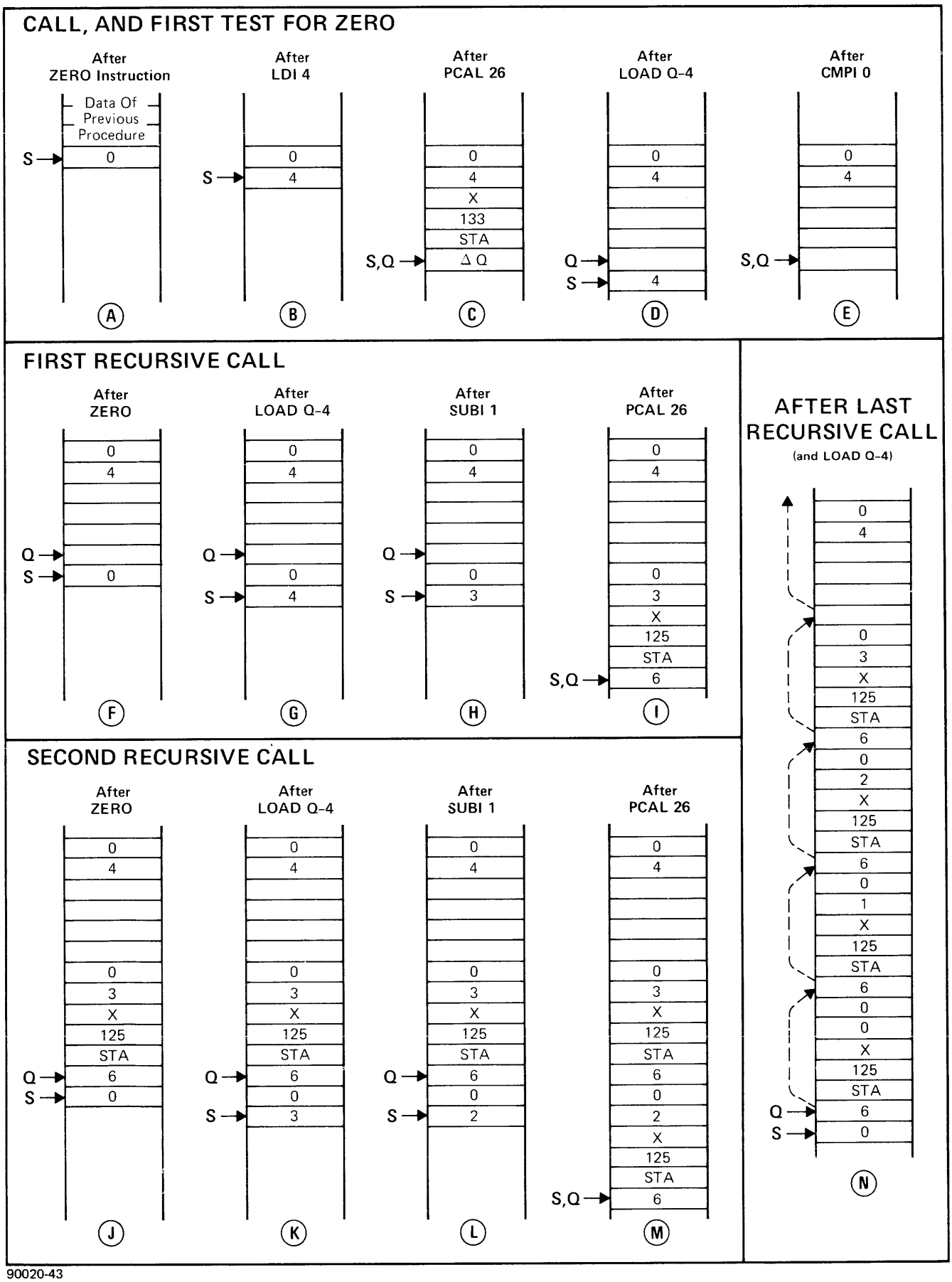


Figure 4-16. Example of Recursive Procedure



90020-43

Figure 4-17. Recursive Calls

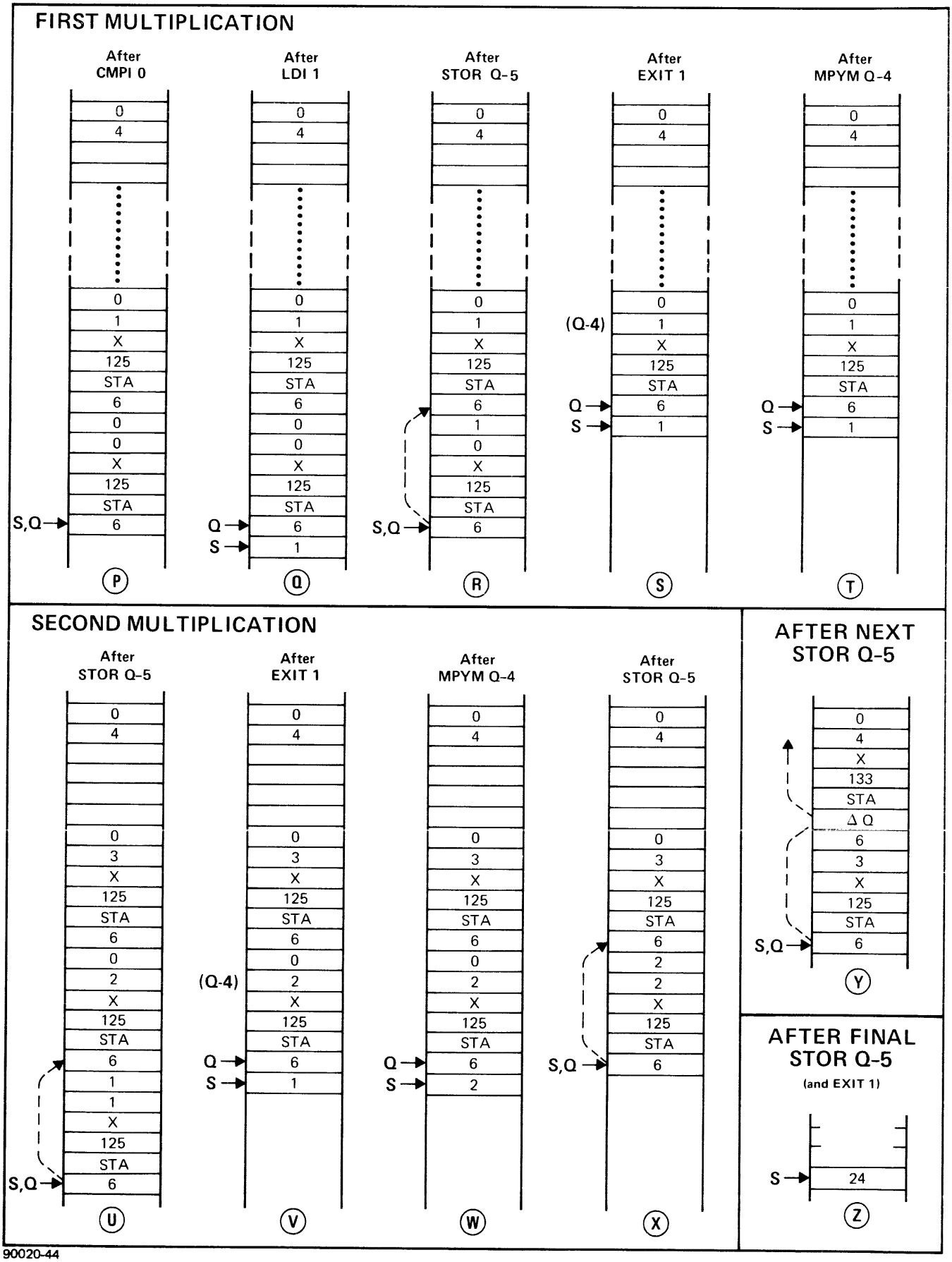


Figure 4-18. Recursive Exits

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.

4-20. 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 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.

4-21. 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, control is transferred 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.

4-22. 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 made again, resulting in another stack marker (M) which is identical to the one generated on the first recursion.

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.

4-23. 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 as 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 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 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 × 1 = 1 there is no apparent change from S to T, but in fact a multiplication has occurred.

4-24. 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.

4-25. SUCCESSIVE EXITS

After saving the result, the next EXIT 1 is encountered again, 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 × 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 × 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 A) 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.

INPUT/OUTPUT SYSTEM

SECTION V

5-1. INTRODUCTION

The general purpose of any computer system is to input, process, and output information. Under MPE, this information may be created and used by the operating system itself, by compilers or other systems, by user programs, or by users themselves. To handle all information in a uniform, efficient way, MPE treats it as groups of data called *files*. Specifically, a *file* is a collection of information or data identified by a name recognized by MPE. It may be helpful to think of a file in the traditional way that people in business and commerce have for many years — as a filing cabinet containing information of various kinds. Instead of a filing cabinet, of course, MPE uses media such as disc, cards, and tape for storing the information. On any of these media, a file may contain MPE commands, system or user programs, or data — alone or in any combination. For instance, a card file containing a batch job might include commands to compile, prepare, and run a payroll — processing program written in COBOL, plus the program itself. The resulting object program, brought into memory from a file on disc, might read input from another disc file that contains wage and salary information for all employees on the company payroll. The program might also write new output to this same file during updating operations.

Within a file, all information is organized into units of related data called *logical records* that for most applications are similar in form, purpose, and content. The records in the file can be arranged in almost any order — alphabetically, numerically, chronologically, by subject matter, and so forth. For example, in the payroll file, each logical record could contain the wage and salary data related to a particular employee. There would be one record for each employee, with the records arranged in alphabetical order according to each employee's last name. Returning to the file cabinet analogy, the logical record would correspond to a sheet of paper serving as the employee's payroll record, stored in the cabinet. The logical record is the smallest grouping of data that MPE can address directly; you specify its length when you create the file. Individual subsystems and user programs, however, also can recognize *fields* for data items within each record. For example, a payroll record for an employee might contain a *field* for each of the following items: the employee's name, social security number, marital status, gross pay, tax exemptions, individual deductions, and net pay. Beyond this, programs also can recognize and manipulate individual words, eight-bit bytes, and bits within a byte.

Data is transferred to and from files in units called *blocks*. These are the basic units that are physically transferred between main memory and the peripheral device on which the file resides. On disc and magnetic tape files, a block consists of one or more logical records; on files on other media, a block normally is equivalent to one logical record (unless you request input/output under the multi-record mode). In the file cabinet analogy, a block is equivalent to a file folder that contains one or more payroll records, removed from the cabinet and returned to it as a set. These records have no particular relationship to one another, except that they are handled most easily when kept together in the folder.

To summarize the interrelation of files, logical records, and blocks:

- A *file* is a collection of records treated as a unit and recognized by a name.
- A *logical record* is a collection of fields treated as a unit, residing in a file.
- A *block* is a group of one or more logical records transmitted to or from a file by an input/output operation.

The purpose of the I/O System, then, is to perform actual physical input/output operations for the file system of the MPE 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. See figure 5-1.

5-2. FILE SYSTEM OPERATION

Figure 5-2 illustrates the function of the I/O system in the overall handling of files. Software elements are shown on the left and hardware elements are shown on the right. The I/O system, as shown, is part software and part hardware. 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*. (Disc *extents* are composed of a 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. As noted previously, 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 5-2 (dotted line) since it frequently makes its own requests to the I/O system. Memory management calls the I/O system in order to bring code and data segments into main memory where they can be accessed by user processes.

In a typical operation, a user process might request the file system to read a file using the FREAD intrinsic (1). (See the MPE Intrinsics Reference Manual for a discussion of the FREAD intrinsic.) The file system reads the record named in the request (2) and transfers the record to the stack associated with the user 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 (BUFFER 0) in main memory.

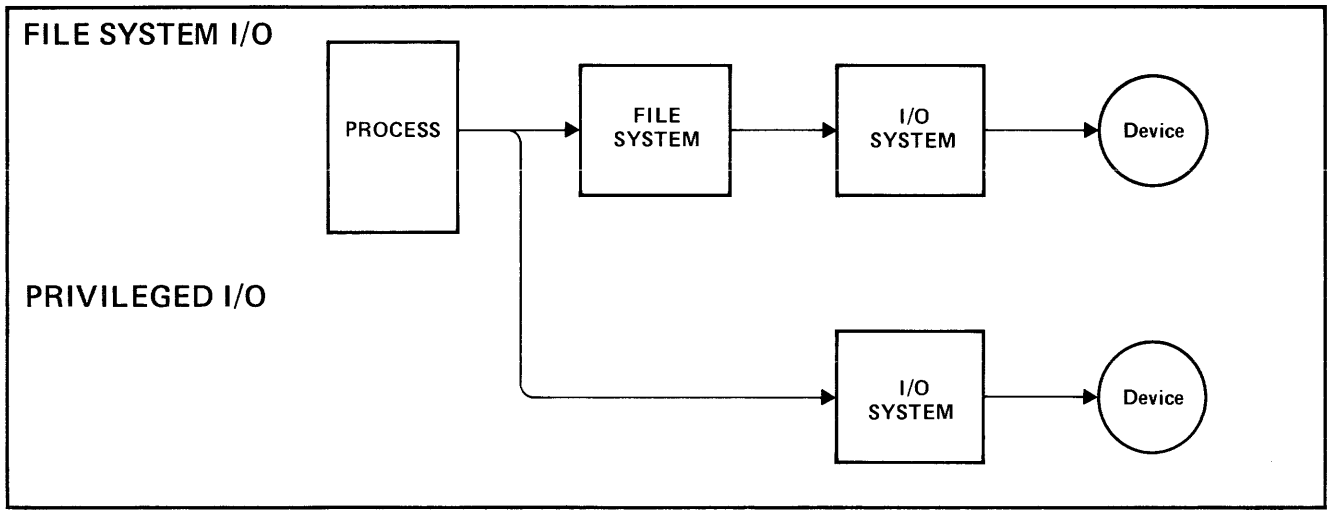
Assume another case in which the requested record is not present. In this case, the file system makes a request to the I/O system (A) to read the block containing the particular record. The I/O system accordingly reads this block from the disc (B) and loads it into one of the buffers (BUFFER 1) allocated to the named file (C). (When you open a file, you specify how many buffers should be allocated for that file; however, you 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 process specify a device. An actual I/O operation may or may not have occurred, and the user is completely unaware of such an occurrence. The operating system, however, allows devices to be specified either by class name or by a specific logical device number. This would permit, for example, inputting or outputting files via a specific terminal, card reader, or line printer.

5-3. DEFINITION OF TERMS

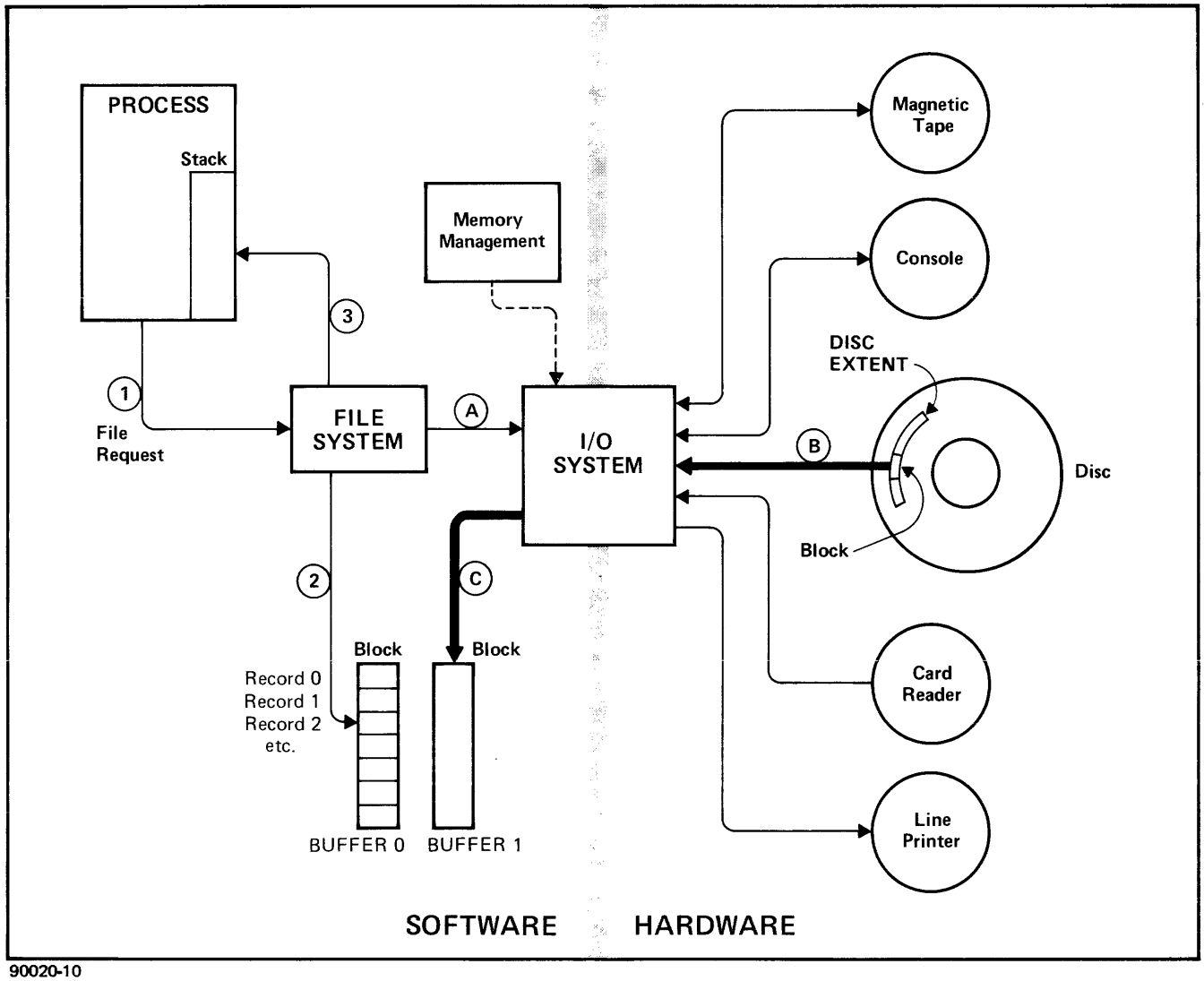
The preceding discussion presented a broad overview to show the relationship of the I/O system to the file system and peripheral devices. The following descriptions will concentrate on the detailed explanation of the block labeled "I/O System" (see figure 5-2).

A Device Controller (see figure 5-3) is the hardware I/O linkage between the CPU and I/O device. It typically consists of one or more logic cards. Depending on particular controllers, the Device Controller may drive only one peripheral (such as a card reader) or may be capable of driving several peripherals (such as disc units). Figure 5-3 illustrates some of the important elements of the I/O system. This figure is by no means complete, but rather is intended to define the chain of linkages that are basic to the I/O system.



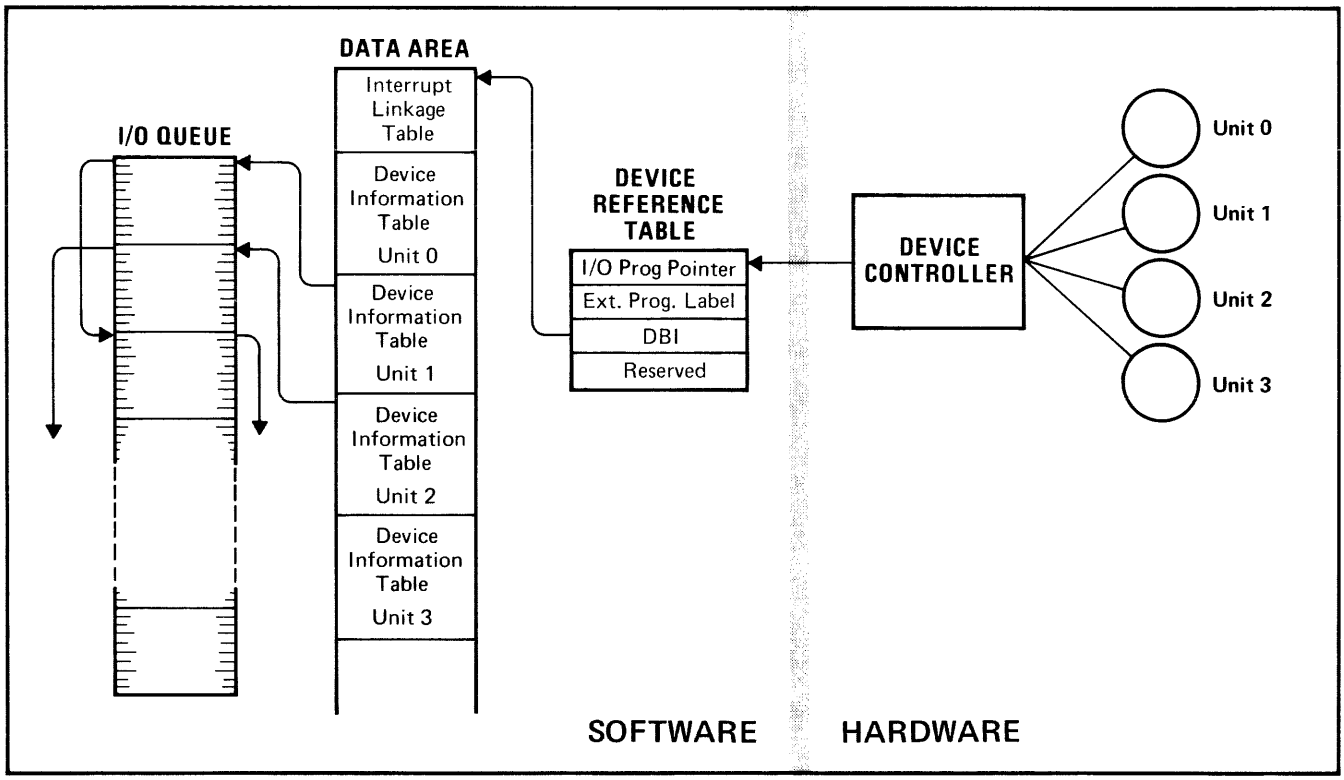
90020-9

Figure 5-1. Basic I/O Access Methods



90020-10

Figure 5-2. File System Basic Operation



90020-11

Figure 5-3. Fundamantal Elements of I/O System

For each Device Controller there is a four-word entry in the *Device Reference Table (DRT)*. The third word in this four-word table entry contains a pointer to a data area uniquely associated with that table entry.

The data area consists of an *Interrupt Linkage Table (ILT)*, one or more *Device Information Tables (DIT)* (depending on how many units the Device Controller is driving), and an *I/O program area*. Along with various other information, the *Driver Linkage Table (DLT)* contains *Code Segment Table (CST)* and *Segment Transfer Table (STT)* values for defining the location of the driver routines associated with that particular Device Controller.

The Device Information Table (DIT) contains information relevant to one physical I/O device, and is configured differently for each type of device. In each case, however, the third word of this table points to an entry in the *I/O Queue (IOQ)* when a request is being made.

The I/O Queue (IOQ) is a single table (only one per 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 entries of the IOQ are unused, and the second word of each entry points to the first word of the next entry. Thus, all unused entries are linked together. Assume, then, that the file system makes a request to use unit 1 of the Device Controller shown in figure 5-3. The I/O system will unlink the first free entry in the IOQ and fill it with information pertaining to the request (including buffer address and logical device number).

Figure 5-3 assumes that the next request is for unit 2 (uses the next available entry), 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 entry, 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 entries.

Next, an *I/O driver* is executed to initiate 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 entry 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 executing I/O drivers are resolved by the *Dispatcher*. (See the *MPE General Information Manual* for a description of the Dispatcher.) Once several Device Controllers are running I/O programs, priority conflicts are resolved by hardware service priority.

Figure 5-4 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. The DRT is located in fixed memory locations beginning at octal address 14. (Locations 0 through 13 are allocated to other purposes; see Section IV, table 4-1.) The upper limit for the table is location 777, which thus limits the maximum number of four-word entries to 125 (decimal).

Because 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., $(3 \times 4)_8 = 14_8$). Because the DRT begins at location 14, device number 3 is the lowest device number. (Devices 0, 1, and 2 do not exist.)

NOTE

> The *device number* associated with a particular DRT entry defines a Device Controller, and not necessarily an actual *physical* device. Remember also that some controllers, identified by one *device number*, are capable of driving several physical devices. Individual identification of *physical 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.

5-4. I/O INSTRUCTIONS

There are ten I/O instructions in the instruction set, as follows:

- CMD Send Command To Module, direct
- SIN Set Interrupt
- SED Set Enable/Disable External Interrupts
- SIO Start I/O
- RIO Read I/O, direct
- WIO Write I/O, direct
- TIO Test I/O, direct
- CIO Control I/O, direct
- SMSK Set Device Mask
- RMSK Read Device Mask

These instructions are fully defined in the *Machine Instruction Set Reference Manual* 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 nine 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 nine 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.

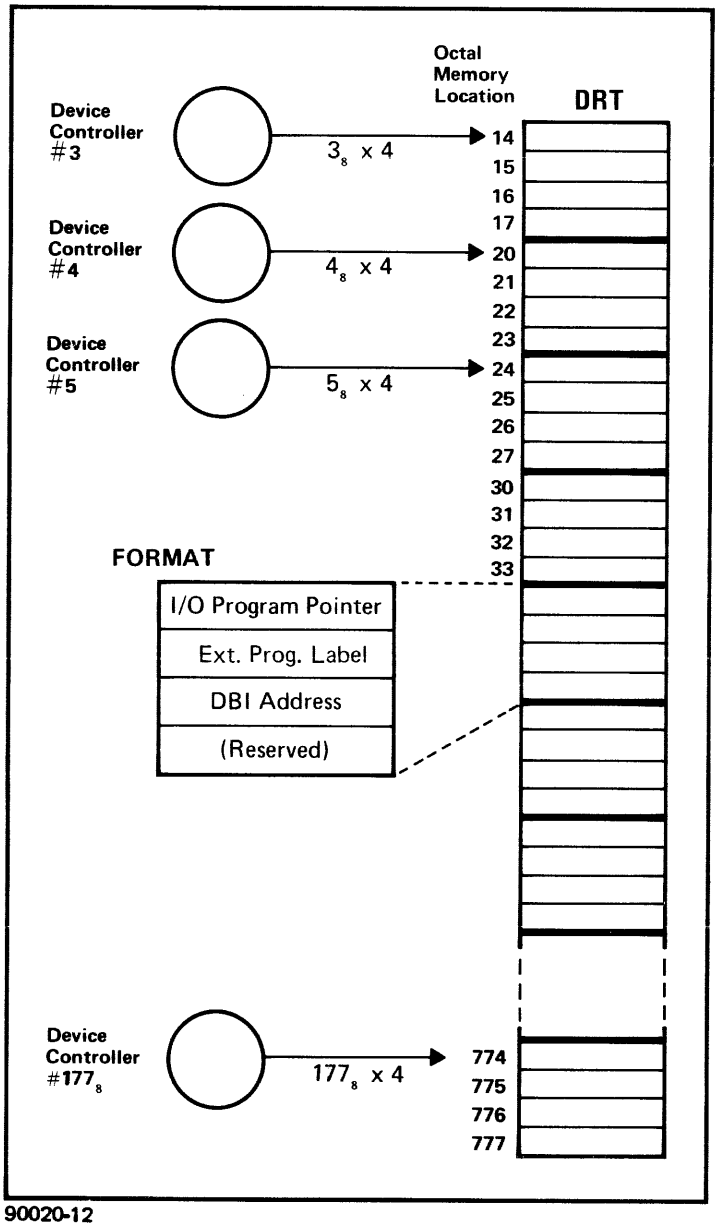
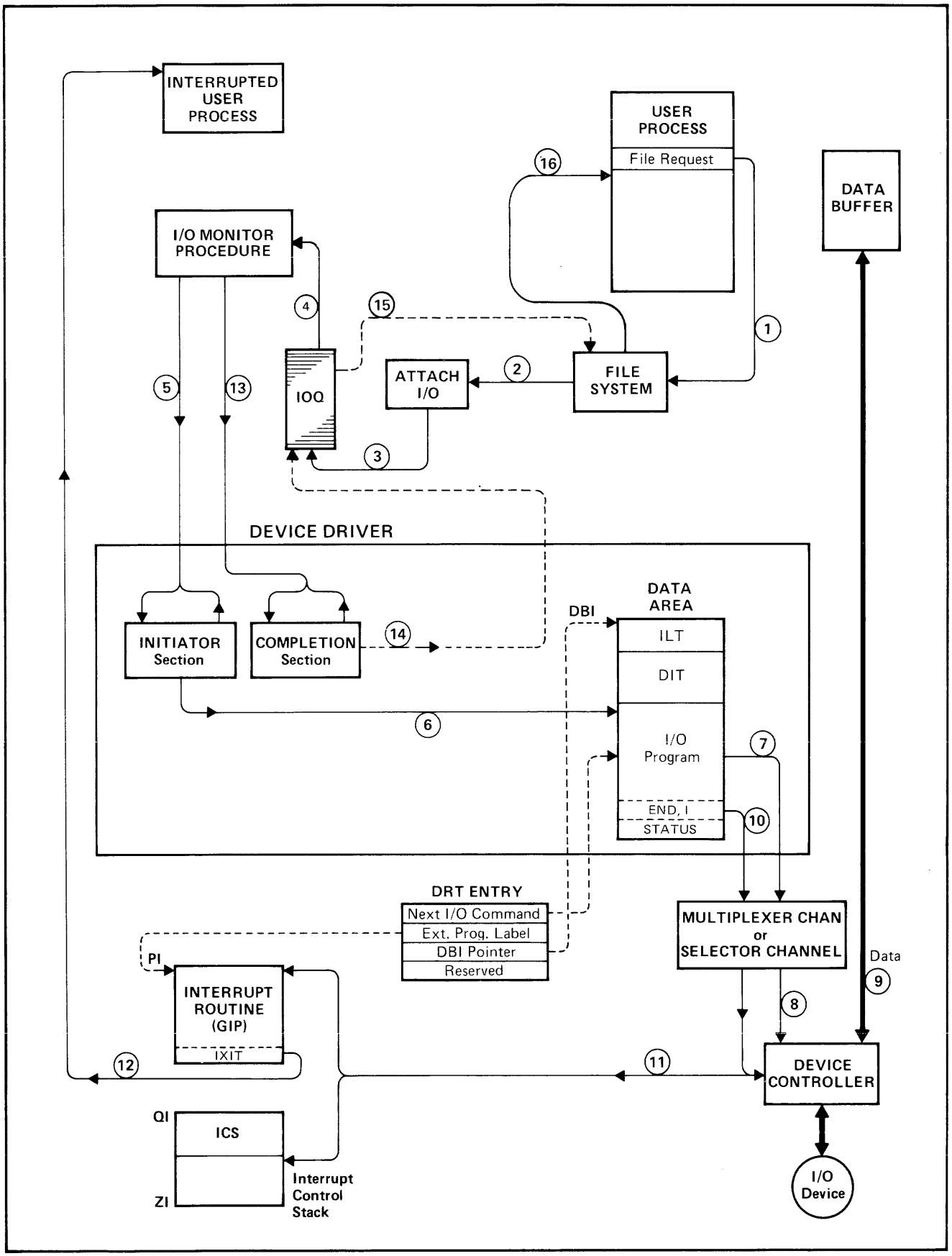


Figure 5-4. Device Reference Table

5-5. GENERAL I/O OPERATION

Figure 5-5 is a general overview of the operations of the I/O system for I/O transfers. (It does not apply to direct I/O devices.) To provide a complete sequence of operations, it is assumed that the file request will result in a need for physical I/O to be performed. (As stated earlier, this is not always the case.) The sequence of operations is as follows:



90020-13

Figure 5-5. I/O System Overview

NOTE

The numbers below correspond to the numbers in figure 5-5.

(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 (ATTACHIO) procedure. This is the entry point to the I/O system.

(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, the I/O Monitor Procedure begins execution for this request.

(5) The I/O Monitor ensures that the data buffer for the file is present in memory. It then issues a procedure call (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 one or more data areas. 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. The SIO instruction initializes the DRT to point at the starting location of the I/O program.

(7) The I/O program issues commands to the Multiplexer or Selector Channel.

(8) The Multiplexer Channel or Selector Channel enables the Device Controller.

(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 that an interrupt has occurred.

NOTE

There are several interrupt routines for external interrupts. For example, one is the *General Interrupt Processor (GIP)* for all types of devices except terminals, and another is the *Terminal Interrupt Processor (TIP)* for terminals.

(12) The interrupt routine (or the last routine to use the Interrupt Control Stack (ICS) - see ICS definition in Section VII) exits to the Interrupted User Process. It also may activate the related I/O process if necessary.

(13) When the I/O Monitor Procedure is executed again, 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 Queue with information regarding results of the original request. The file system may then check these results.

(15) When the user process is dispatched again, a return is made to a point following the file request, depending on whether blocked or unblocked I/O was specified. (Refer to "Blocked/Unblocked I/O", paragraph 5-9.)

5-6. 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.

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 representing any particular device or even a *typical* device. It is assumed that the log-on sequence has been accomplished.

Figures 5-6 and 5-7 illustrate the handling of data via direct I/O terminal devices. Figure 5-6 shows input (read) operations and figure 5-7 shows output (write) operations.

In comparison with figure 5-5, 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, and no Multiplexer Channel or Selector Channel is involved.

One element not previously present is the line buffer. The line buffer consists of a linked list of buffers, which are pointed to by an address word in the Device Information Table for a particular terminal. A sufficient number of these buffers 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. Thus, the line buffer reads *characters* from the device until the complete line (or *record*) is read, then transfers the *complete* line (or record) to the data buffer. 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.

5-7. DIRECT READ

The sequence of operations for direct read, illustrated in figure 5-6, is as follows. Again, it will be assumed that the file request requires a physical read from the terminal. The numbers below correspond to the numbers shown in figure 5-6.

(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 procedure.

(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 then are inserted into an appropriate place in the queue. (For example, factors such as whether the request is from the system console are considered.)

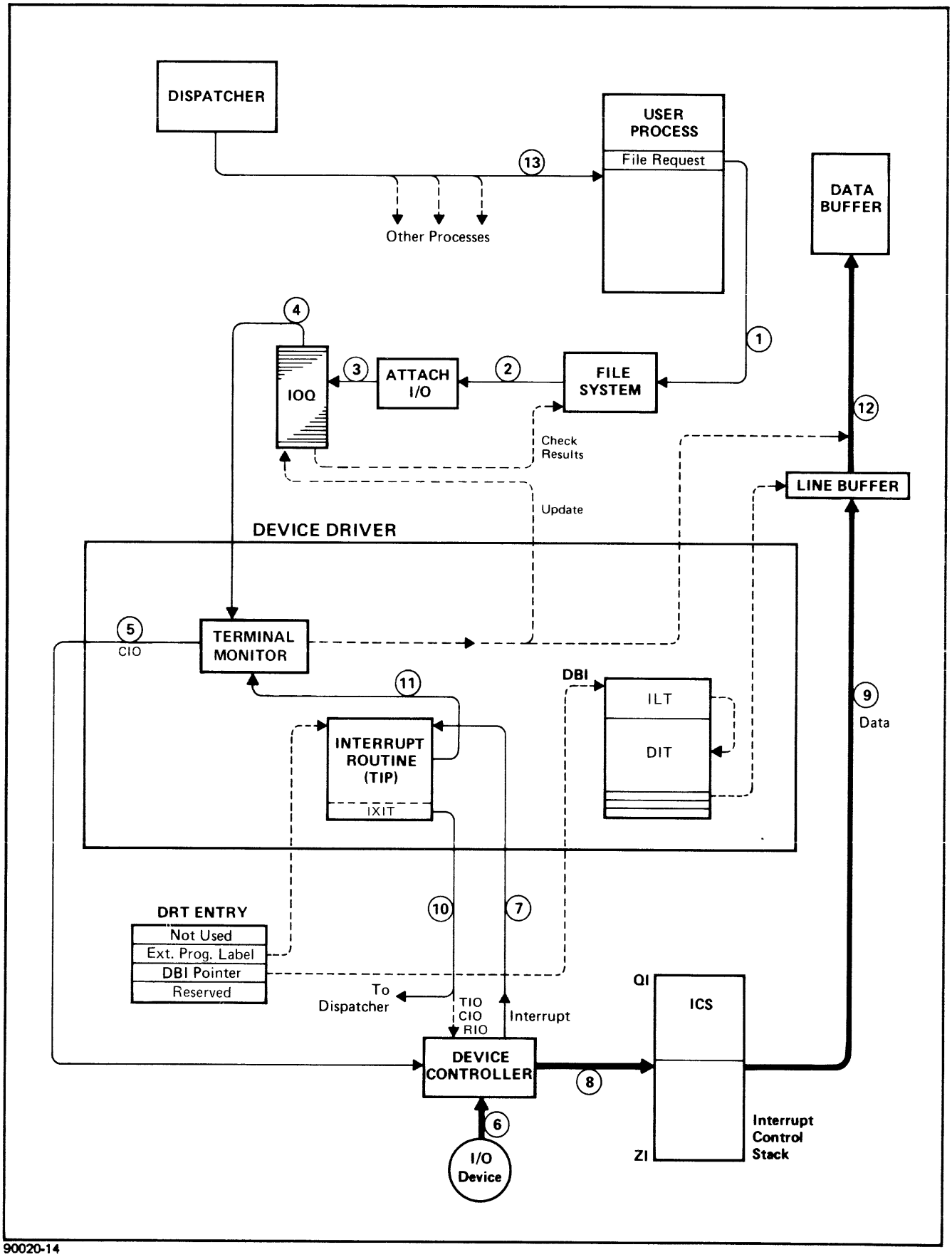


Figure 5-6. Direct Read For Terminal Devices

(4) When all higher priority requests have been completed, the TERM procedure begins execution for this request.

(5) The Terminal Monitor issues a CIO (Control I/O) instruction directly to the Device Controller, causing TIP to initiate the read operation.

(6) The Device Controller enables the device to transmit 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 a CPU 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 transmit the next character. TIP now checks the character on the ICS to see if it is a data character or a control character (such as a carriage return).

(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 a CR control character (Carriage Return) is detected, TIP sets a flag in the Device Information Table to signify that the record is complete, then causes the Terminal Monitor to be executed.

(12) The Terminal Monitor transfers the content of the line buffer to the data buffer and the transmission log in the IOQ is updated.

(13) The Terminal Monitor releases the line buffer and control is returned to the user process. To read another record, the file system must make another I/O request to Attach I/O.

5-8. DIRECT WRITE

The sequence of operations for direct write is illustrated in figure 5-7. The numbers below correspond to the numbers shown in figure 5-7.

(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 procedure.

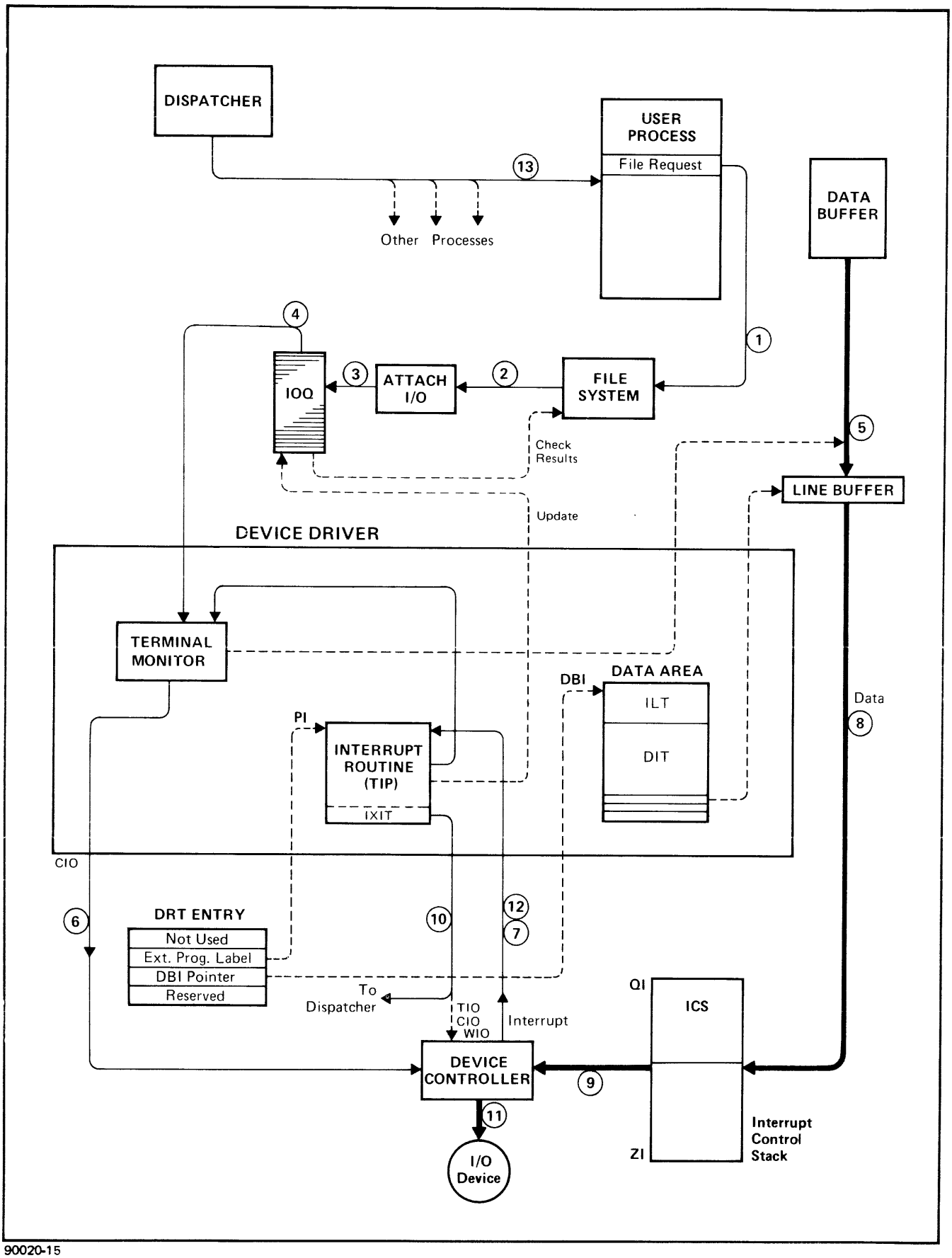
(3) Attach I/O inserts the request parameters in the I/O Queue for the requested device.

(4) When all higher priority requests have been completed, the Terminal Monitor begins execution for this request.

(5) The Terminal Monitor transfers the data from the data buffer to the line buffer.

(6) A CIO (Control I/O) instruction is issued to the Device Controller to initiate the write operation.

(7) The Device Controller causes the CPU to interrupt to TIP, the Terminal Interrupt Processor (TIP).



90020-15

Figure 5-7. Direct Write For Terminal Devices

(8) TIP transfers a character to the ICS.

(9) TIP executes a WIO instruction, transferring the character from the ICS to the Device Controller.

(10) TIP then exits to the Dispatcher, and hardware takes control from this point.

(11) The Device Controller writes the character out to the device.

(12) On completion of the write, the Device Controller generates another interrupt to TIP. The sequence repeats back to step 8 until all characters in the record have been written out to the terminal.

(13) The Dispatcher then returns control to the user process.

5-9. 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. While the I/O operation was in progress, the user process may have been suspended 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 ATTACHIO. (The file system nearly always uses unblocked I/O.) The following paragraphs discuss the characteristics of blocked and unblocked I/O.

5-10. BLOCKED I/O

As shown in figure 5-8, the user process goes into an I/O wait state as soon as the I/O request is given.

The user process remains in the wait state while the I/O operations proceed. 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 IOQ. Control is then returned to the user process. In the case of terminal writes, the operation is considered completed when the data has been transferred to the terminal buffers.

The user process now continues to execute from the point following the I/O request.

5-11. UNBLOCKED I/O

In the case of unblocked I/O, also illustrated in figure 5-8, privileged capability is assumed. The process must also specify the action to be taken on completion of I/O; either no action or reactivate the process if in an I/O wait state. This specification (like the blocked/unblocked I/O choice) is made in the call to ATTACHIO.

The process may then, after calling ATTACHIO, 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 for I/O completion. The process also has the capability to put itself into the I/O wait state, 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 state; there is no way to recover from this situation. At least one request must reactivate the process.

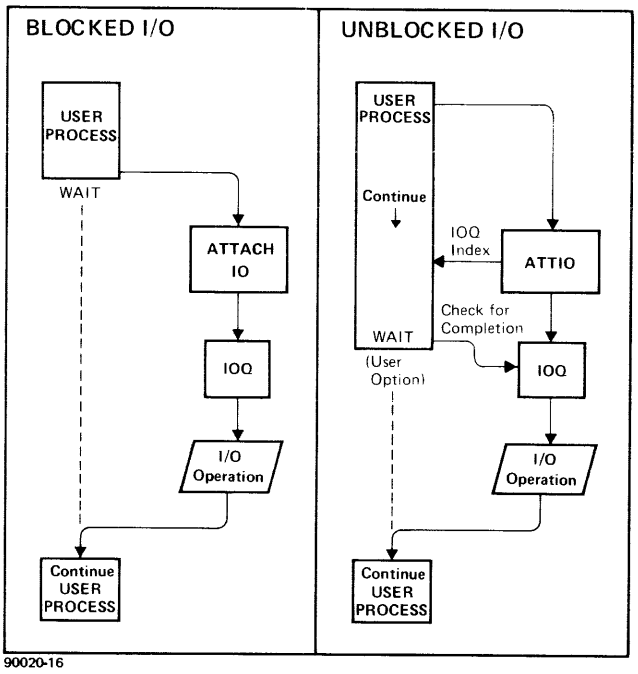


Figure 5-8. Blocked and Unblocked I/O

While the process continues to execute, ATTACHIO 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 IOQ. Then the completion action bit is examined. If *awaken* process is specified, the process will be reactivated if it has put itself into the I/O wait state, as shown in figure 5-8. If *no action* is specified, presumably the process has continued to execute without any wait, or will be reactivated by some other process. In any case, the process checks for I/O completion.

5-12. 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 performs other functions.

The remainder of this section describes the hardware I/O system.

5-13. HARDWARE ELEMENTS

The hardware I/O system consists of: the I/O Processor (IOP), Multiplexer Channel, Selector Channel, Device Controller, and peripheral device. The following paragraphs define the basic functions of each of these elements.

5-14. I/O PROCESSOR. The I/O Processor has three basic functions, relating to the three different transfer modes illustrated in figure 5-9:

1. For 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.

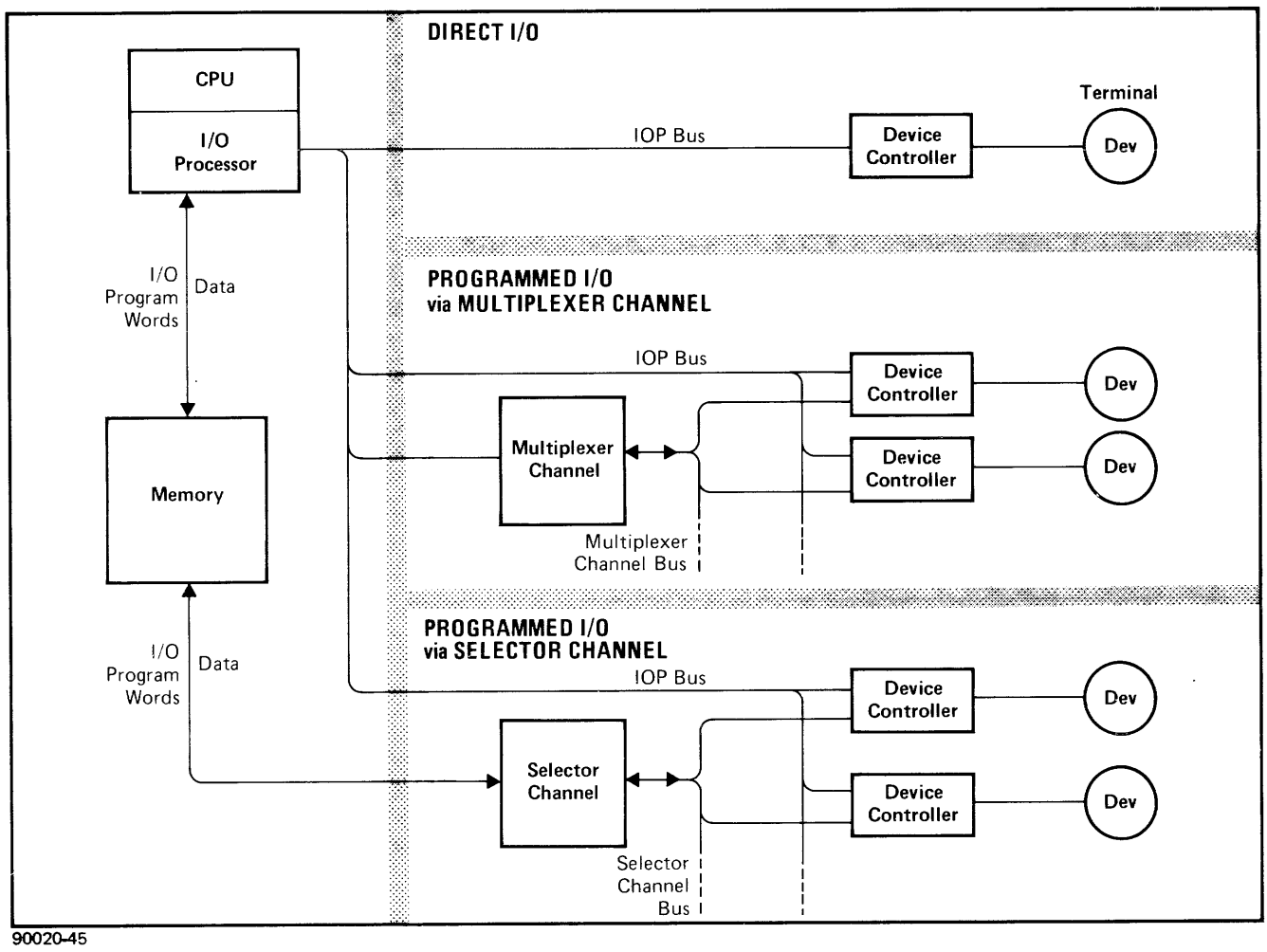


Figure 5-9. Hardware I/O Elements

2. For 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 Device Controller.

3. For 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.

In addition, the IOP interrupts the CPU on behalf of the Device Controllers.

5-15. MULTIPLEXER CHANNEL. The Multiplexer Channel acts as a switch to enable one of the 1 through 16 Device Controllers connected to it 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 doubleword 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 fetching the next I/O program doubleword when necessary.

5-16. SELECTOR CHANNEL. The Selector Channel also acts as a switch but in a manner different from a Multiplexer Channel. The Multiplexer Channel switches between Device Controllers on demand, based on hardware priority, whereas the Selector Channel maintains the connection for one Device 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 5-9, 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.

5-17. DEVICE CONTROLLER. The Device Controller is the hardware linkage between a perpheral device and the computer system. Its primary function is to translate programmed I/O commands from a Multiplexer Channel 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 Multiplexer Channel or Selector Channel. The Device Controller also generates interrupts when required by some device condition or by direct or programmed command.

5-18. PERIPHERAL DEVICE. The peripheral device receives output data for storage or display, or supplies input data to the computer. Usually, one Device Controller controls one peripheral device; however, some Device Controllers are capable of controlling several devices.

5-19. I/O PROGRAMMING

When 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.

5-20. I/O PROGRAM WORD. Figure 5-10 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.

The count field of the IOCW contains the least significant 12 bits of 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 nine 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.

- SET BANK. This defines the memory bank to be used for data transfers. The bank, for a given device, is set to zero when an SIO command is sent to the device.
- INTERRUPT. This causes the Device Controller to set its interrupt request flip-flop, causing an interrupt to 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 the 12-bit count field, and a 16-bit control word in the IOAW to the Device Controller.
- 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 from main memory to the device, starting at the address given by the IOAW.
- READ. This causes count words of data to be transferred from the device to main memory, starting at the address given by the IOAW.

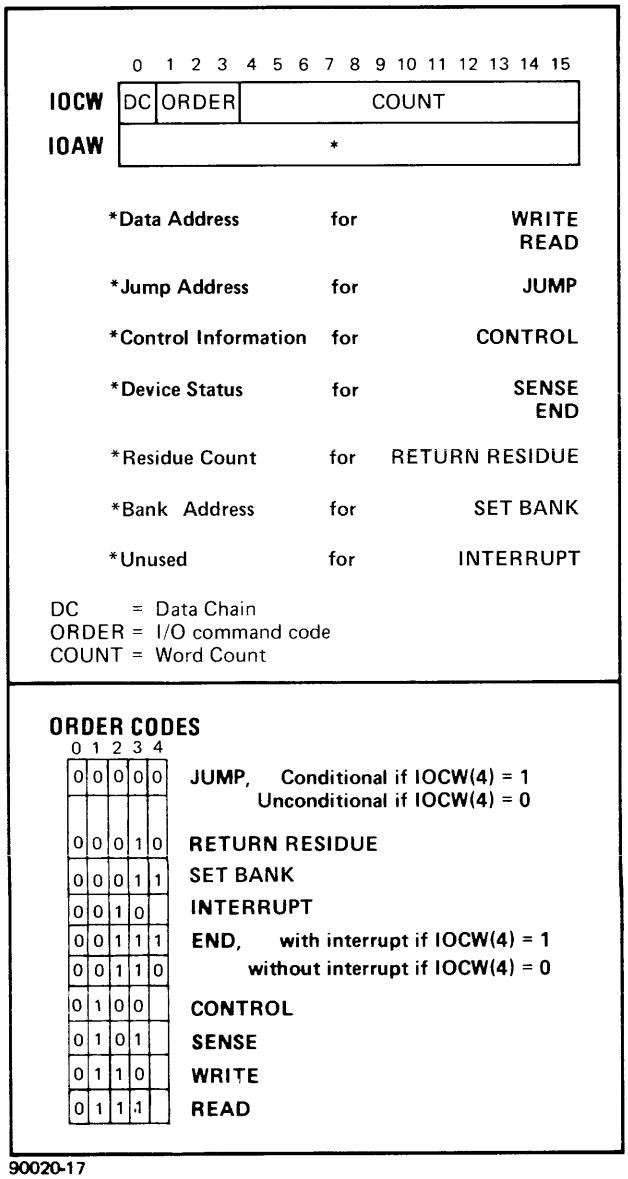


Figure 5-10. I/O Program Word Format

5-21. TYPICAL I/O PROGRAM OPERATION. Figure 5-11 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 device number given at S-K in the stack (2), and puts the top-of-stack (TOS) into the first word of its Device Reference Table entry as the I/O program pointer.

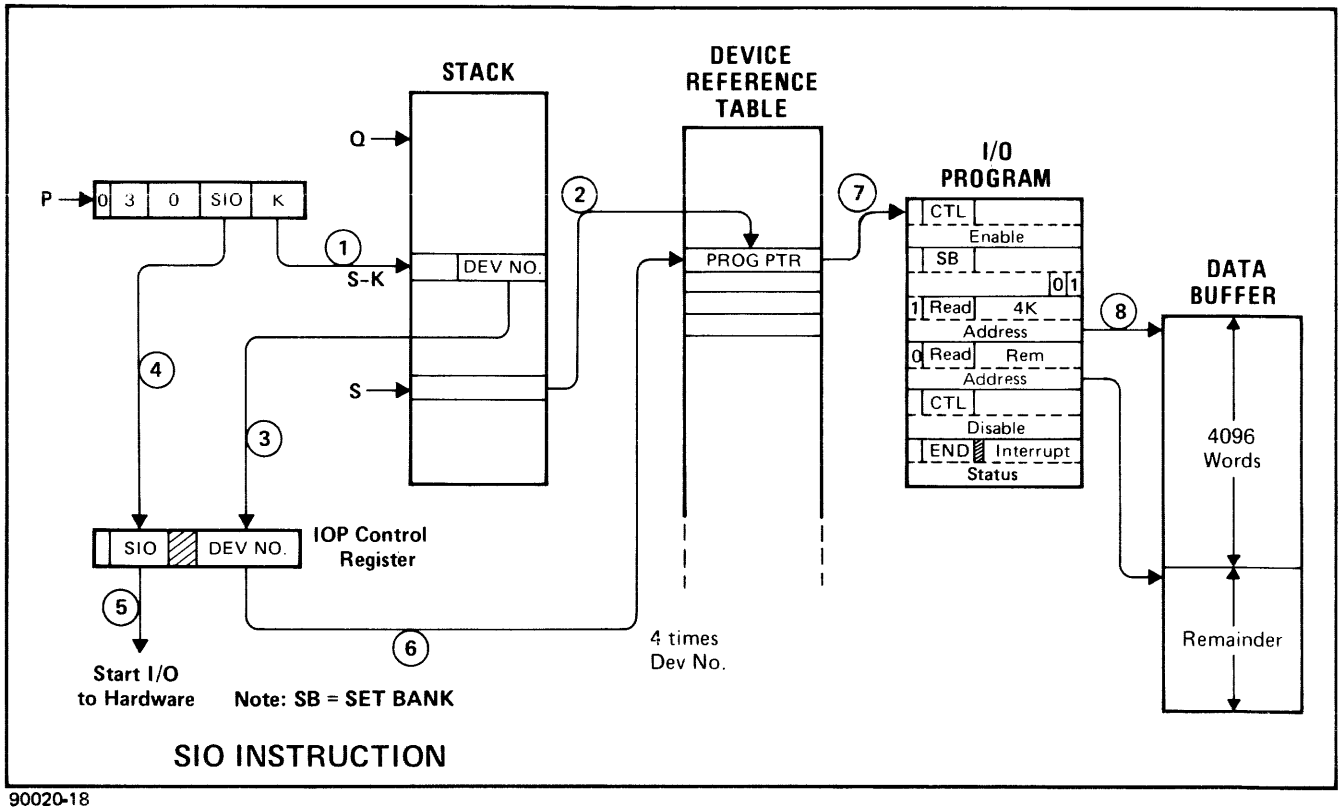
3 SIO then sends the device number to the IOP Control Register, and (4), sends an SIO command to the IOP.

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 5-4), 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 doubleword 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.)

8 The sample I/O program operates as follows. The first doubleword contains a CONTROL order which enables the hardware I/O subsystem for this device. The second doubleword 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) doubleword 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 will either specify an additional buffer area contiguous to the first 4096-word buffer if desired, or in another part of memory.



90020-18

Figure 5-11. I/O Program Operation

When the transfer is complete, the final doubleword contains an END order, which obtains the result of the transfer (device status) and loads it into the IOAW; the END order then tells the controller to generate 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 Channel does not take any special action since it updates the DRT after each order fetch.

INTERRUPT SYSTEM

SECTION VI

6-1. INTERRUPT SYSTEM OVERVIEW

The interrupt system conforms to the basic architectural scheme of the HP 3000 Series II and III Computer Systems. 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. an explicit PCAL instruction). Also, code and data domains are kept separate.

The primary differences are that the calling operations are performed by a microprogrammed *Interrupt Handler* rather than the PCAL instruction and, in some cases, the IXIT (Interrupt Exit) instruction is used for exiting the interrupt code instead of EXIT.

Code segment number 1 contains all internal interrupt procedures. Interrupt procedures for I/O devices may be in any segment other than segment number 1.

Table 6-1 lists the internal interrupts and traps with their corresponding entry numbers in the Segment Transfer Table (STT) of the internal interrupt code segment. The *parameter* is a value that is derived by the Interrupt Handler and which passes relevant information about the interrupt to the interrupt routine.

The Device Reference Table (DRT) contains a label for each entry, pointing to the interrupt procedure for each device. Bit 8 of the CPX1 register indicates an external interrupt. The *parameter* value for an external interrupt is the device number.

Before discussing the various interrupt types, the Interrupt Control Stack will be defined, since it will be referred to frequently throughout the succeeding descriptions.

Table 6-1. Interrupt Types

EXT. PROG. LABEL (%)	STT NO. (%)	INTERRUPT TYPE	PARAMETER*	EXECUTING STACK**
100401	1	Bounds Violation		
101001	2	Illegal Memory Address		
101401	3	Non-Responding Module		
102001	4	System Parity Error		ICS
102401	5	Address Parity Error		ICS
103001	6	Data Parity Error		ICS
103401	7	Module Interrupt	Module No.	ICS
104001	10	(Unused)		
104401	11	Power Fail		ICS
105001	12	(Unused)		
105401	13	(Unused)		
106001	14	(Unused)		
106401	15	(Unused)		
107001	16	(Unused)		
107401	17	(Unused)		
110001	20	Unimplemented Instruction		
110401	21	STT Violation		
111001	22	CST Violation		

Table 6-1. Interrupt Types (Continued)

EXT. PROG. LABEL (%)	STT NO. (%)	INTERRUPT TYPE	PARAMETER*	EXECUTING STACK**
111401	23	DST Violation		
112001	24	Stack Underflow		
112401	25	Privileged Mode Violation		
113001	26	(Unused)		
113401	27	(Unused)		
114001	30	Stack Overflow		ICS
114401	31	User Traps		
		a. Integer Overflow	%000001	
		b. Floating-Point Over.	%000002	
		c. Floating-Point Under.	%000003	
		d. Integer Divide by 0	%000004	
		e. Floating-Point Divide by 0	%000005	
		f. Ext. Prec. Floating-Point Overflow	%000010	
		g. Ext. Prec. Floating-Point Underflow	%000011	
		h. Ext. Prec. Floating-Point Divide by 0	%000012	
		i. Decimal Overflow	%000013	
		j. Invalid ASCII digit	%000014	
		k. Invalid Dec. digit	%000015	
		l. Invalid Source Word Count	%000016	
		m. Result Word Count Overflow	%000017	
		n. Decimal Divide by 0	%000020	
115001	32	(Unused)		
115401	33	(Unused)		
116001	34	(Unused)		
116401	35	(Unused)		
117001	36	(Unused)		
117401	37	Absent Code Segment		
		a. On PCAL	P-Label	
		b. On EXIT	N	
		c. On IXIT	0	
120001	40	Trace		
		a. On PCAL	P-Label	
		b. On EXIT	N	
		c. On IXIT	0	
120401	41	STT Entry Uncallable	P-Label	
121001	42	Absent Data Segment	DST No.	
121401	43	Power On		ICS
122001	44	Cold Load		ICS
		a. System I/O (SIO)	0	
		b. Direct I/O (DIO)	Label	

*Unless noted, the parameter is the External Program Label.
**Unless noted, Interrupts are serviced on the User Stack.

All User Traps (STT No. %31) are enabled by the User Traps bit in the Status register.

6-2. INTERRUPT CONTROL STACK (ICS)

The Interrupt Control Stack (ICS) is a single stack, unique to the 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 (Device Reference Table) must be used for data. The use of a common stack also permits efficient nesting of interrupt routines by using *stack markers*.

The ICS has a permanent stack marker, set up by the operating system which is used to enter the Dispatcher. Figure 6-1 illustrates the format of the Dispatcher marker on the ICS.

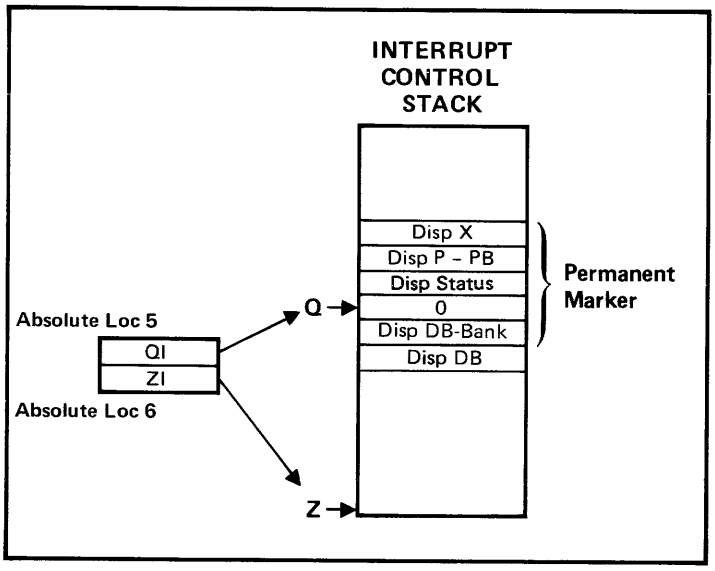


Figure 6-1. Dispatcher Marker on ICS

Note that, unlike the standard four-word stack marker, the Dispatcher marker contains six words. As will be explained later, all markers created because of an ICS type interrupt include two words to save the current value of DB and DB-bank. This information must be saved since all external interrupts automatically alter DB (to the DBI value from word 2 of the DRT) and DB-bank. Additionally, DB may be changed when processing an ICS-type internal interrupt.

The Delta Q location of the Dispatcher marker always contains a "0" word in bits 1-15 (Bit 0 can be set as described later). The value is zero because there is no previous marker on the ICS. The Dispatcher Flag is set whenever the Dispatcher is entered and remains set while the Dispatcher is executing. It is cleared when the Dispatcher completes its execution, or is interrupted.

The segment-number field of the status word (Disp Status) in the Dispatcher marker permanently points to the CST entry for the Dispatcher, and the P-PB (Disp P-PB) word permanently points to the starting point in the Dispatcher code segment. The DISP (Dispatch) 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.

Note that since ICS-type interrupts use a 6-word marker, the parameter is found in location Q+3, rather than the usual Q+1 location.

A hardware ICS Flag is set in the CPX1 register whenever a switch is made to the ICS from any other stack. The ICS Flag remains set until the ICS is no longer the current stack.

Figure 6-1 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 (Series II only), 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.

6-3. INTERRUPT TYPES

Interrupts may be divided into three basic types: external interrupts, which are signals from the I/O system, and two types of internal interrupts, which typically are unexpected conditions resulting from program execution. The three interrupt types are:

- External interrupts (from I/O devices)
- ICS-type internal interrupts (using the Interrupt Control Stack)
- Non-ICS internal interrupts

Figure 6-2 compares the overall operations of all three interrupt types. 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. The Dispatcher is the part of the operating system which schedules the execution of processes.

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.

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 three interrupt types. Only a brief introductory description is given at this point. Detailed operating sequences are given later in this section.

6-4. EXTERNAL INTERRUPTS

External interrupts interface external events to software processes. Referring to figure 6-2 (top example), the overall operation is as follows:

1. The Device Controller sets the interrupt flip-flop by one of the following:
   a. SIN (Set Interrupt) software instruction executed by the CPU, telling the device controller to interrupt.
   b. SET INT (Set Interrupt) command decoded by the Device Controller.
   c. END,I (End with Interrupt) command decoded by the Device Controller.
   d. The Device Controller detects an interruptable condition.
2. The setting of the Interrupt Request flip-flop causes the device controller to issue an INT REQ (Interrupt Request) signal to the I/O processor.

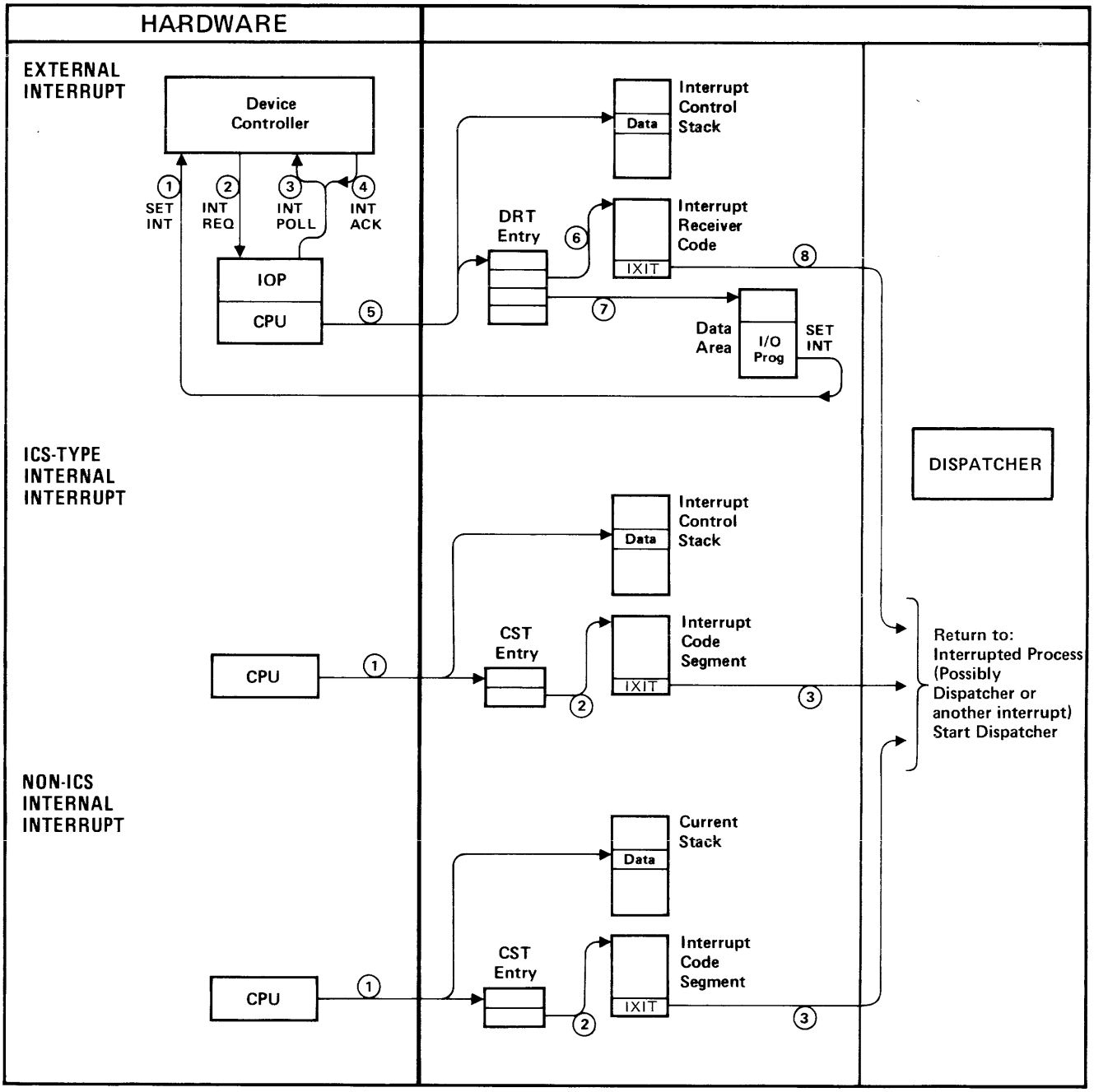


Figure 6-2. Interrupt System Overview

3. The I/O processor issues a poll (INT POLL) to activate the highest-priority request (There may be more than one request).

4. The Device Controller sets the Interrupt Active flip-flop, resets the Interrupt Request flip-flop, and sends the device number to the I/O processor.

5. The I/O processor examines the device number and:

   a. If the device number does not equal zero:

      1. Puts the device number into an I/O processor register.

2. Passes an interrupt signal to the CPU.

3. Turns on the external interrupt flag (Bit 8 in the CPX1 register).

4. Drops the INT POLL (Interrupt Poll) signal.

b. If the device number equals zero, disregards the interrupt signal and drops the INT POLL signal.

7. The I/O Processor 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.

8. The external program label in the second word of the DRT entry is used to get the CST entry for the interrupt receiver code which, in turn, is used to set the PB-bank, PB, and PL registers. The starting address for the interrupt receiver code is obtained from the STT entry pointed to by the external program label and is loaded into the P register, thus transferring control to the interrupt receiver code.

9. 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.

10. The interrupt receiver code IXITS, normally back to the interrupted process (which may be another interrupt or the Dispatcher). The interrupt receiver may also request a new dispatch by executing a DISP instruction. When an IXIT is executed by external interrupt code, a reset command is sent to the appropriate device.

6-5. 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 segment 1 (see table 6-1). Referring to the second example in figure 6-2, the overall operation is as follows:

1. A condition (such as power failure, stack overflow, or module interrupt) causes the CPU to switch to the ICS (after saving the user's environment on his own stack), by creating an External Program Label which points to a Segment Transfer Table entry in the internal interrupt segment (CST entry 1).

2. The PB and PL registers are set up based on CST entry 1.

3. The status register is set to Privileged Mode, Segment 1 with all other bits cleared (%100001).

4. The P register is set from the local label, reached via the STT entry in the External Program Label, thus transferring control to the internal interrupt code segment.

6-6. NON-ICS INTERNAL INTERRUPTS

The non-ICS type interrupts operate on the current user's stack. Referring to figure 6-2, the overall operation is as follows:

1. A special condition is detected and causes the CPU to save the user's environment on his own stack and to fetch the CST entry 1 by creating an External Program Label to the code for processing the interrupt.

2. The PB and PL registers are set up based on CST entry 1.

3. The status register is set to Prvileged Mode, Segment 1 with all other bits cleared (%100001).

4. The P register is set from the local label, reached via the STT entry in the External Program Label, thus transferring control to the internal interrupt code segment.

6-7. EXTERNAL INTERRUPT PROCESSING

Before discussing the sequence of operations for external interrupts, there are two important factors that need to be considered. These are *interrupt priorities* and *interrupt program pointers.*

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. 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. The routing of the interrupt poll is determined by the desired interrupt priorities of the Device Controllers, and is completely independent of other parameters.

6-8. INTERRUPT PROGRAM POINTER

The Device Reference Table was defined in Section V. As stated then, the second word of each DRT entry contains the interrupt program pointer. This is an external program label pointing to the start of the interrupt routine associated with a particular Device Controller. (See figure 6-3.) Note that several controllers could point to the same routine.

6-9. SEQUENCE OF OPERATIONS

Figures 6-4 and 6-5 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 5-5, and to steps 5, 6, and 7 in figure 6-2.

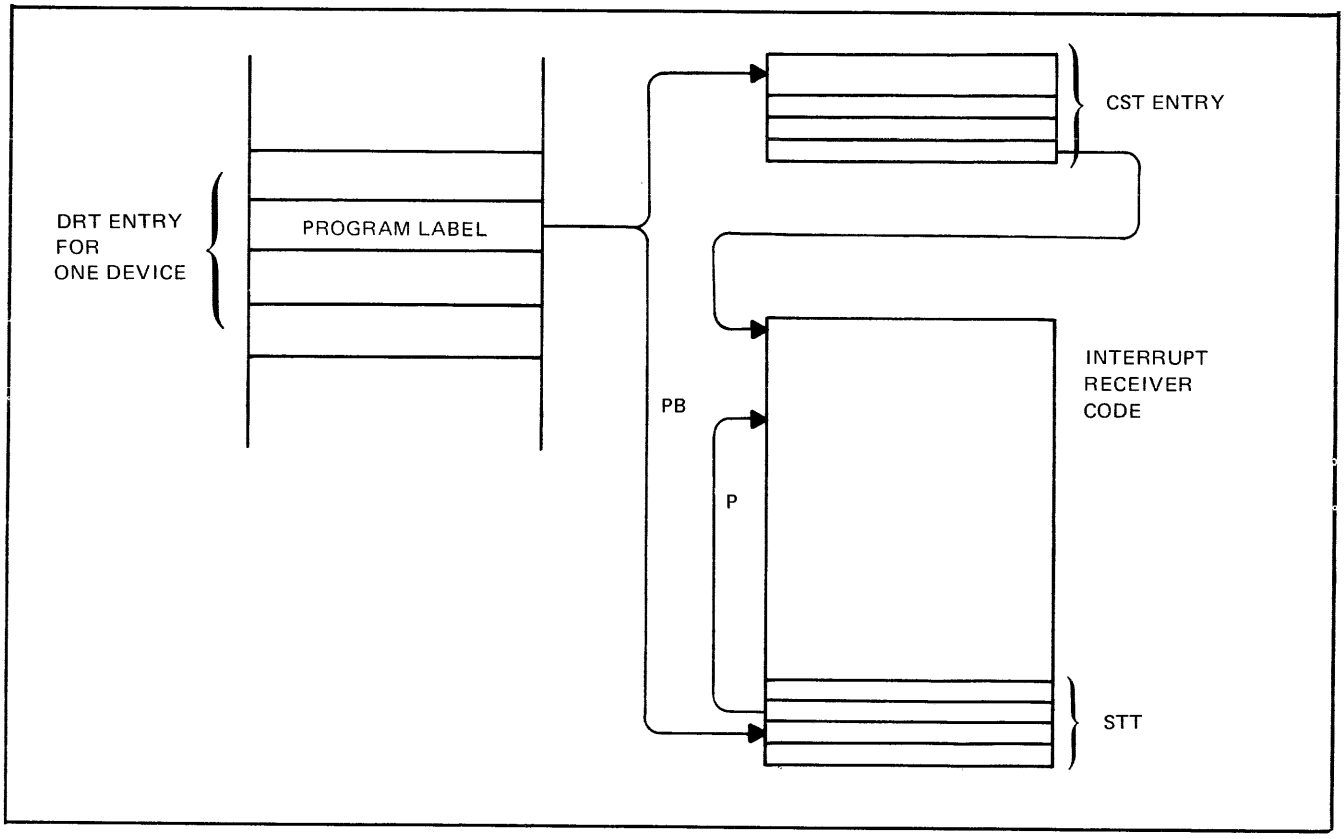


Figure 6-3. Device Controller

Figure 6-4 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 6-5 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. Next, a normal four-word stack marker is pushed onto the user's stack followed by the value of the user's DB-bank and the absolute value of DB that is currently in use. (DB may not necessarily point to a location within the user's stack, such as if a system intrinsic using a split stack 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 6).

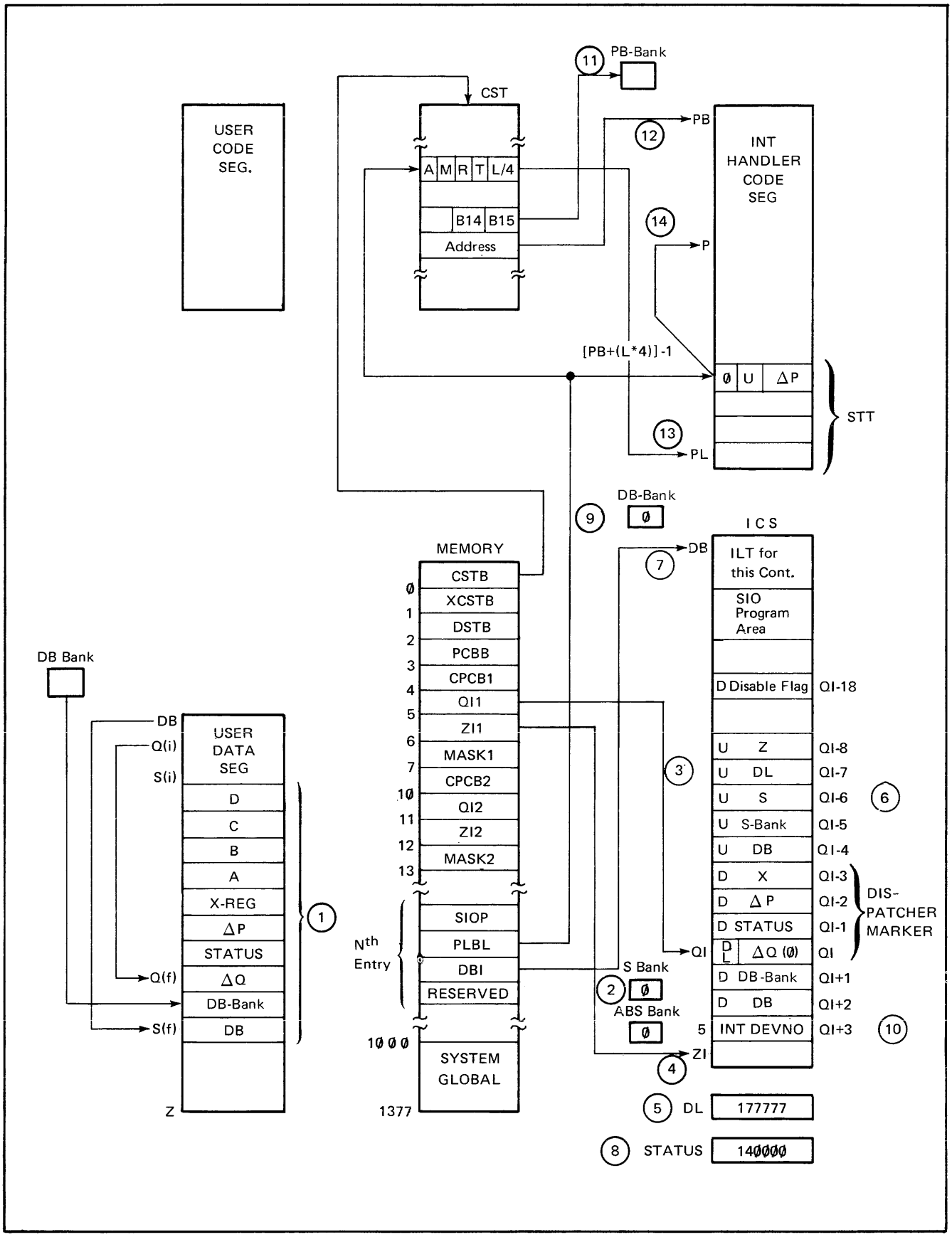


Figure 6-4. First Level External Interrupt

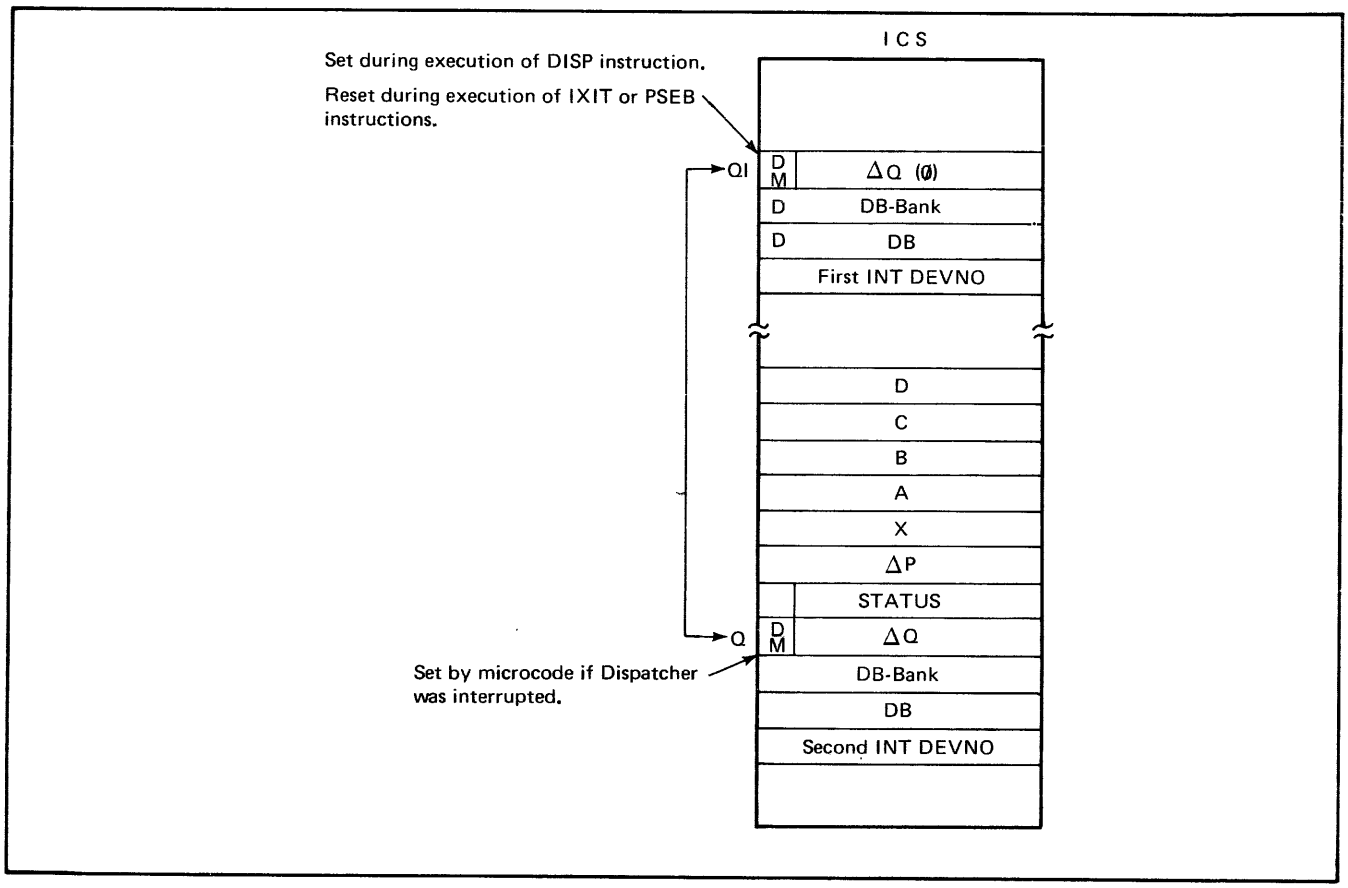


Figure 6-5. Second Level Interrupt or Dispatcher Interrupted

2. The S-bank register is set to 0 (the ICS is always in Bank 0).

3. The Interrupt Handler now goes to location 5 (assuming CPU 1) and loads the QI value into the Q register. This points at the Delta Q location of the permanent Dispatcher marker. (As explained previously, this location contains a value of 0.)

4. The contents of location 6 are 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 in the CPX1 register is also set.)

5. The DL register is set to the limit value of %177777.

6. The user's value of S relative to Stack DB (at QI-4) is calculated and stored in QI-6.

7. The CPU obtains the device number from the Interrupt Address register in the IOP and calculates the address of the DRT entry. DB is set to the DBI value in the third word of the Device Reference Table entry.

8. The status register is set to privileged mode, external interrupts enabled (%140000).

9. The DB-bank register is set to 0.

10. The S register is set to point at location Q+ 3 and the device number of the interrupting device is stored into this location. At this point, the Interrupt Control Stack is fully delimited by register values, and is ready for handling interrupt data.

11. The external program label for the interrupt receiver code is fetched from the second word of the DRT entry. The CST entry is obtained from the segment number in the external program label. Then, the PB bank register is set based on the CST entry.

12. The PB register is set based on the CST entry.

13. The PL register is set based on the CST entry.

14. The starting address of the interrupt receiver code is obtained from the STT entry pointed to by the external program label in the DRT entry. The interrupt receiver code segment number is placed in the status register. The P register is set to this value and the CPU fetches the instruction at P and begins executing the interrupt receiver code.

The following steps, relating to figure 6-5, list the actions occurring if a second interrupt of higher priority 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 additonal interrupts. This reduces the interrupt response time.

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 15) to step 21. 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. The sequence continues as follows:

15. The Interrupt Handler pushes into memory any TOS elements that are in CPU registers, and pushes the usual six-word marker onto the ICS. The fifth and sixth words are the values that are currently in the DB-bank and DB registers respectively at the time of the interrupt.

16. 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 previous marker.

NOTE

Unlike the first interrupt, subsequent interrupts do not store S into Q-6 at this point (see step 6) since such action would overlay one of the variables associated with the user who was first interrupted.

17. The CPU obtains the device number from the Interrupt Address register in the IOP and calculates the address of the DRT entry. DB is set to the DBI value in the third word of the Device Reference Table entry.

18. The S register is set to point at location Q+ 3 and the device number is stored into this location. At this point, the Interrupt Control Stack is fully delimited by register values and is ready for handling interrupt data.

19. The external program label for the interrupt receiver code is fetched from the second word of the DRT entry. The starting address of the interrupt receiver code is obtained from the STT entry pointed to by the external program label. The P register is set to this value and the CPU fetches the instruction at P and begins executing the interrupt receiver code.

20. Assuming there are no other higher priority interrupts, the interrupt routine for the second device runs to completion and then exits (using the IXIT instruction). The exit, as usual, is made via the stack marker. The return address is obtained from the stack marker, the Q register is restored back to the previous setting (using the Delta-Q value from the stack marker), pointing to the Delta-Q word of the Dispatcher marker. The S register is moved back to the location just preceding the second stack marker. One of the actions of the IXIT 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.

21. The interrupt receiver code for the first interrupt now runs to completion and an exit is made, usually back to the user process. Again, the IXIT instruction issues a Reset Interrupt command to the Device Controller. This completes the sequence of operations.

If an 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, the sequence effectively repeats steps 16 through 21 with the added actions that, in step 18, bit 0 of Delta Q is set to 1 (indicating a Dispatcher interrupt) and the Dispatcher Flag is cleared.

6-10. INTERNAL INTERRUPT PROCESSING

As listed in table 6-1, there are 35 internal interrupts including 14 user traps. These 35 interrupts are processed by the segment whose CST entry number is 1. Each interrupt has an entry in the Segment Transfer Table (STT) which points to the start of the code to process the interrupt. The user-related traps all share the same STT entry and the parameter value determines the processing to be performed.

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, and power fail.

In all cases, the Interrupt Handler loads a parameter onto the stack. The parameter (listed in table 6-1) 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.

6-11. GENERAL DESCRIPTIONS

6-12. BOUNDS VIOLATION. A bounds violation trap is caused by attempting to address locations outside of a specified program domain or data domain; refer to "Bounds Checking" in Section III.

6-13. ILLEGAL MEMORY ADDRESS. A memory address interrupt is caused by attempting to access a word of memory that does not physically exist on the system.

6-14. NON-RESPONDING MODULE. 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 on the order of 4.6 milliseconds).

6-15. SYSTEM PARITY ERROR. A parity error is detected on the 8-bit system information (TO, FROM, COMMAND) transmitted by the CPU to memory, or by memory to the CPU. This error will also be generated in the case where the CPU is waiting for data and a Memory-to-IOP transmission takes place with bad parity. In this case a *transfer error* is also sent to the requesting device. Note that the converse is also true (i.e., if the IOP is waiting for data and the CPU receives a transmission with bad system parity, a *transfer error* is sent to the requesting device). The above is the result of the CPU and IOP sharing the same module number. A system parity error also results if any module sends data with bad parity (not addresses) to memory.

6-16. ADDRESS PARITY ERROR. A parity error is detected by the memory on the 16-bit address word sent to it from any module. Upon detection of the error, the memory sends an appropriate error signal back to the CPU, and prevents the word addressed from being altered.

6-17. DATA PARITY ERROR. A parity error is detected by the CPU on the 16-bit data word sent to it from the memory.

When a parity error is detected on a memory transmission, the appropriate bit is set in CPX1 (the CPU status word for RUN-mode interrupts) and the instruction runs to completion (with the exception of certain interruptable instructions such as the group of move instructions). The result of the instruction is normally meaningless. If the parity error is due to a CPU *read* cycle (outgoing system or address information or incoming data), it is possible that the received data will be used by the CPU as an address for a following *write* cycle. In this case it would be possible to store erroneous data at some location. However, since bounds checking is done on the address, the worst that can happen is the destruction of a memory location in the current user's stack (assuming user mode — if in privileged mode, a system crash could occur). With single-bit error correcting memory, the probability of having a data parity error is very small.

6-18. 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 the interrupt routine as a parameter. The interrupt routine may then attempt to identify the source of the error and take appropriate action. The interrupt is disabled (when bit 1 of the Status register is 0). This interrupt can also be used as a flag between the CPU and another module for information swapping.

6-19. POWER FAIL. 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).

6-20. UNIMPLEMENTED INSTRUCTION. This system trap occurs when the CPU detects a bit pattern in the current instruction register which is not a valid instruction. This trap cannot be disabled by the User Traps Enable/Disable bit in the status register.

6-21. STT VIOLATION. A variety of conditions can cause this trap, as shown below:

1. The STT in an external program label is greater than the STT length (pointed to by PL) in the referenced segment. This error can occur while attempting to set up a new segment.

2. In the LLBL instruction, if the label which is fetched from PL-N is an internal label and N is greater than 127 (%177), the trap is invoked (this would require too large an STT number when creating the external label).

3. In PCAL, when setting up a new segment, if the STT number in the external program label points to an external program label in the new segment, the trap is invoked.

4. If (PL-N) in an SCAL instruction is an external label, the trap is invoked.

The STT Violation trap cannot be disabled.

6-22. CST VIOLATION. This trap is caused by an attempt to transfer to segment 0 or a segment number referenced through an external program label is greater than the CST length.

6-23. DST VIOLATION. The Data Segment Table segment number referenced by the MFDS, MTDS, or MDS instruction is greater than the number of entries contained in DSTL (the first word of the DST).

6-24. STACK UNDERFLOW. The process being exited is non-privileged and SM is less than DB. This might result from deleting too much information from the stack, or from using the SETR or SUBS instructions incorrectly.

6-25. PRIVILEGED MODE VIOLATION. This trap is caused by an attempt to execute a privileged instruction in user mode (that is, when bit 0 of the status register is 0). This violation also occurs in EXIT, if an attempt is made to exit from user to privileged mode, or if exiting from user mode and the external interrupts bit in the status word has been altered.

6-26. STACK OVERFLOW. A stack overflow results from attempting to stack more data than can be contained on the current stack (SM greater than Z). The system makes the decision whether to abort the process or to expand the stack.

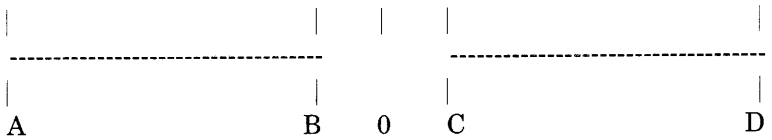
6-27. INTEGER OVERFLOW. An integer overflow occurs when the result of an integer operation (ADD, SUB, etc.) is outside the allowable range of integers which is −32768 to +32767.

6-28. FLOATING-POINT OVERFLOW. This trap occurs when the magnitude of the result of a two-word floating-point operation is larger than the largest representable floating-point number which is $1.157921 \times 10_{-77}$.

6-29. FLOATING-POINT UNDERFLOW. This trap occurs when the magnitude of the result of a two-word floating-point operation is less than the smallest representable positive number, which is $8.63617 \times 10_{-78}$, and is not equal to zero.

NOTE

Floating-point overflow and underflow can best be understood by referring to the chart below showing the range of valid numbers.



where

$A = -1.157921 \times 10_{77}$
$B = -8.63617 \times 10_{-78}$
$C = 8.63617 \times 10_{-78}$
$D = 1.157921 \times 10_{77}$

A number is valid if it is between A and B, C and D, or equal to 0.

6-30. INTEGER DIVIDE BY ZERO. This trap occurs when the divisor in a DIV, DIVI, DIVL, or LDIV instruction is equal to zero.

6-31. FLOATING-POINT DIVIDE BY ZERO. This trap occurs when the divisor in an FDIV instruction is equal to 0.

6-32. EXTENDED PRECISION FLOATING-POINT OVERFLOW. This trap occurs when the magnitude of the result of an extended precision floating-point operation exceeds the largest representable extended precision value which is $1.157920892373162 \times 10_{-77}$.

6-33. EXTENDED PRECISION FLOATING-POINT UNDERFLOW. This trap occurs when the magnitude of an extended precision floating-point operation is less than the smallest representable positive extended precision value (which is $8.636168555094445 \times 10_{-78}$) and is not zero.

6-34. EXTENDED PRECISION FLOATING-POINT DIVIDE BY ZERO. This trap occurs when the divisor in an extended precision divide operation is zero.

6-35. DECIMAL OVERFLOW. This trap occurs when a packed decimal result has too many significant digits for the specified storage size. Except for the NSLD instruction, and MPYD with actual result greater than 28 digits, when this occurs the low order digits of the result are stored; surplus high order digits are discarded.

6-36. INVALID ASCII DIGIT. This trap occurs when a decimal arithmetic instruction encounters an invalid ASCII digit.

6-37. INVALID DECIMAL DIGIT. This trap occurs when a decimal arithmetic instruction encounters an invalid packed decimal digit.

6-38. INVALID WORD COUNT. This trap occurs when a word count for a decimal instruction is less than zero or greater than six.

6-39. RESULT WORD COUNT OVERFLOW. This trap occurs when a digit count for a decimal instruction is less than zero or greater than 28.

6-40. DECIMAL DIVIDE BY ZERO. This trap occurs when an attempt is made to divide a decimal number by zero.

6-41. ABSENT CODE SEGMENT. The absence bit in the CST entry for the referenced segment is set to 1. This check is performed in PCAL, EXIT, IXIT, DISP, and the firmware Interrupt Handler. If during PCAL, the program label is passed as the parameter to the Interrupt Handler; if in EXIT, the number of words to be deleted from the stack is passed; and if in IXIT, a zero is passed.

6-42. TRACE. Non-local PCAL and external interrupts check the trace bit in the CST entry for the referenced segment. EXIT and IXIT check bit two of the return address in the marker stacked by PCAL or the external interrupt (this bit is set by the trace routine software if it is desired to trace exits). In either case, if the bit tested is 1, the trace routine is entered. For PCAL's and external interrupts, another marker is stacked first which is used by the EXIT from the trace routine. For EXIT and IXIT, no marker is stacked; hence, bit 0 of the return address of the last marker stacked (prior to EXITing from trace) must be cleared by software in the trace routine. Otherwise, an infinite trace loop could occur. Tracing segment 1 results in a system halt. Tracing external interrupts or the Dispatcher requires special software in the trace routine due to the differences in EXIT and IXIT.

6-43. STT ENTRY UNCALLABLE. The uncallable bit in a local label (or in PL if the STT number is 0) is set to 1. This label is referenced by a PCAL from another segment. This trap does not stack a new marker.

6-44. ABSENT DATA SEGMENT. The absence bit in the DST entry for the referenced segment is set to 1.

6-45. POWER ON. The Power On routine is entered either by an internal power turn-on, or by an automatic restart following a power failure when 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 routine will set up the software environment and pass control to the operating system.

6-46. COLD LOAD. Pressing the LOAD switch while simultaneously pressing the ENABLE 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 an eight-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 eight-word I/O program. This program reads in a 32-word bootstrap loader (a larger 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 1, with the ICS as the data

domain. The status register is set to 100001 (octal) to indicate privileged mode, and code segment 1. Then the CPU halts. When RUN is pressed, the cold-load routine in segment 1 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.

6-47. SEQUENCE FOR ICS TYPE INTERNAL INTERRUPTS

Figure 6-6 illustrates the sequence of operations for processing ICS type internal interrupts. The figure shows 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 the 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 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. Next, a normal four-word stack marker is pushed onto the user's stack followed by the value of the user's DB-bank and the absolute value of DB that is currently in use. (DB may not necessarily point to a location within the user's stack, such as if a system intrinsic using a split stack 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 6).

2. The S bank register is set to 0 (the ICS is always in Bank 0).

3. The Interrupt Handler now goes to location 5 (assuming CPU 1) and loads the QI value into the Q register. This points at the Delta-Q location of the permanent Dispatcher marker. (As explained previously, this location contains a value of 0.)

4. Next, the contents of location 6 is 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 in the CPX1 register is also set.)

5. The DL register is set to the limit value of %177777.

6. The user's value of S relative to stack DB (at QI-4) is calculated and stored in QI-6. (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.)

7. An external program label is created which points to segment 1, and whose STT number is a function of the type of interrupt (see table 6-1).

8. S is now set to Q+3 and a parameter is pushed onto the ICS at that location. Most ICS type internal interrupts pass the external program label; however, for example, a Module Interrupt passes the module number.

9. The current instruction is placed in the index register.

10. The interrupt condition is cleared.

11. PB-bank, PB and PL are set up based on CST entry number 1.

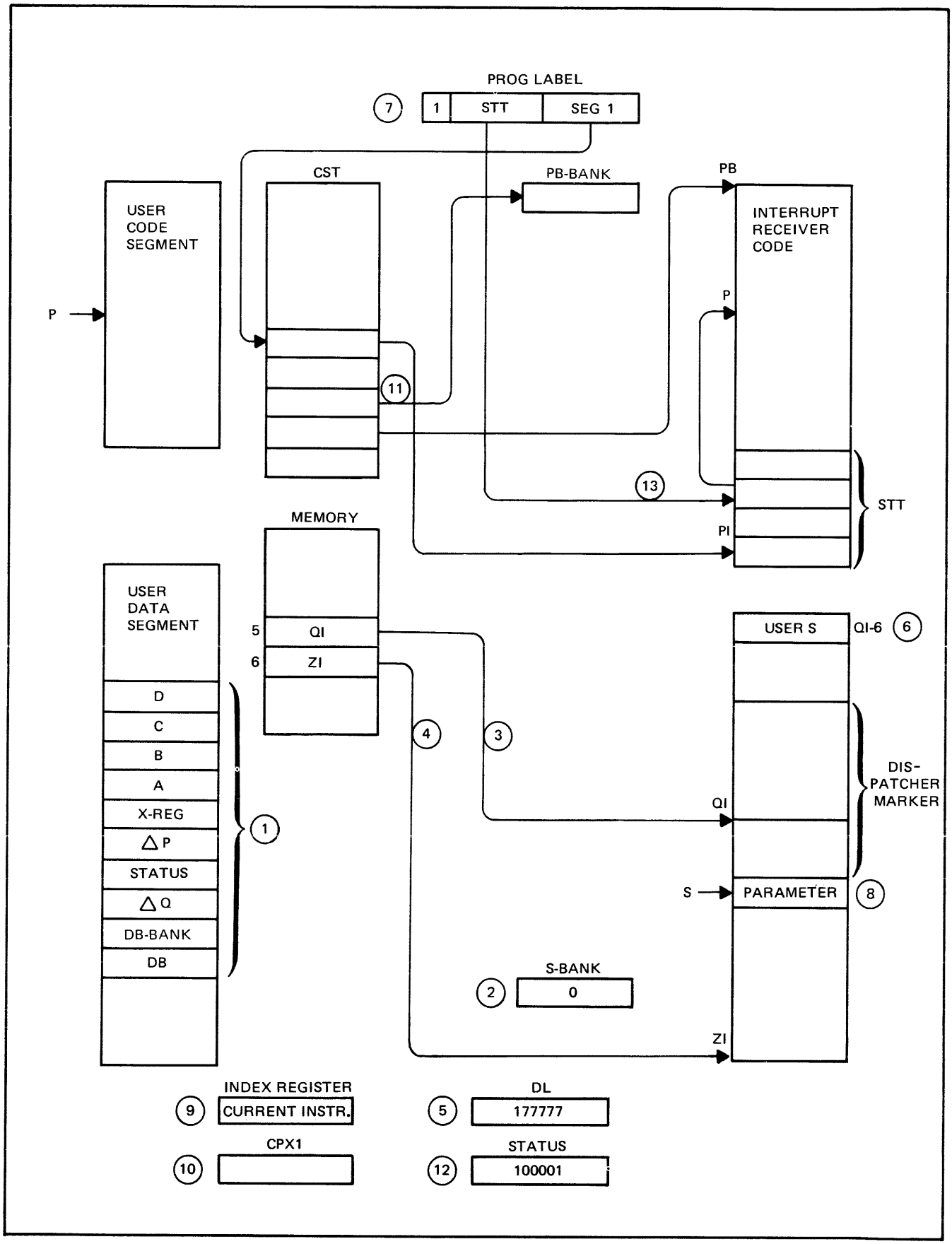


Figure 6-6. ICS Internal Interrupt Operations

12. The status register is set to privileged mode, segment 1 with all other bits cleared (%100001).

13. The starting address of the interrupt receiver code is obtained from the STT entry pointed to by the external program label. The P register is set to this value and the CPU fetches the instruction at P and begins executing the interrupt receiver code.

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 6-5. 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 back to the interrupted user will occur or the Dispatcher may be entered.

6-48. SEQUENCE FOR NON-ICS TYPE INTERRUPTS

Figure 6-7 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 (see table 6-1).

4. The current instruction is placed in the index register.

5. The Interrupt Handler generates an external program label to the interrupt receiver code in segment number 1.

6. The PB-bank, PB, and PL registers are set based on the CST entry.

7. The status register is set to privileged mode, segment 1 with all other bits cleared (%100001).

8. P is set from the local label, reached via the STT entry in the external program label, thus transferring control to the interrupt receiver code.

6-49. 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 that follow are essentially a summary of the preceding portion of this section. A flowchart is used as a basis for discussion.

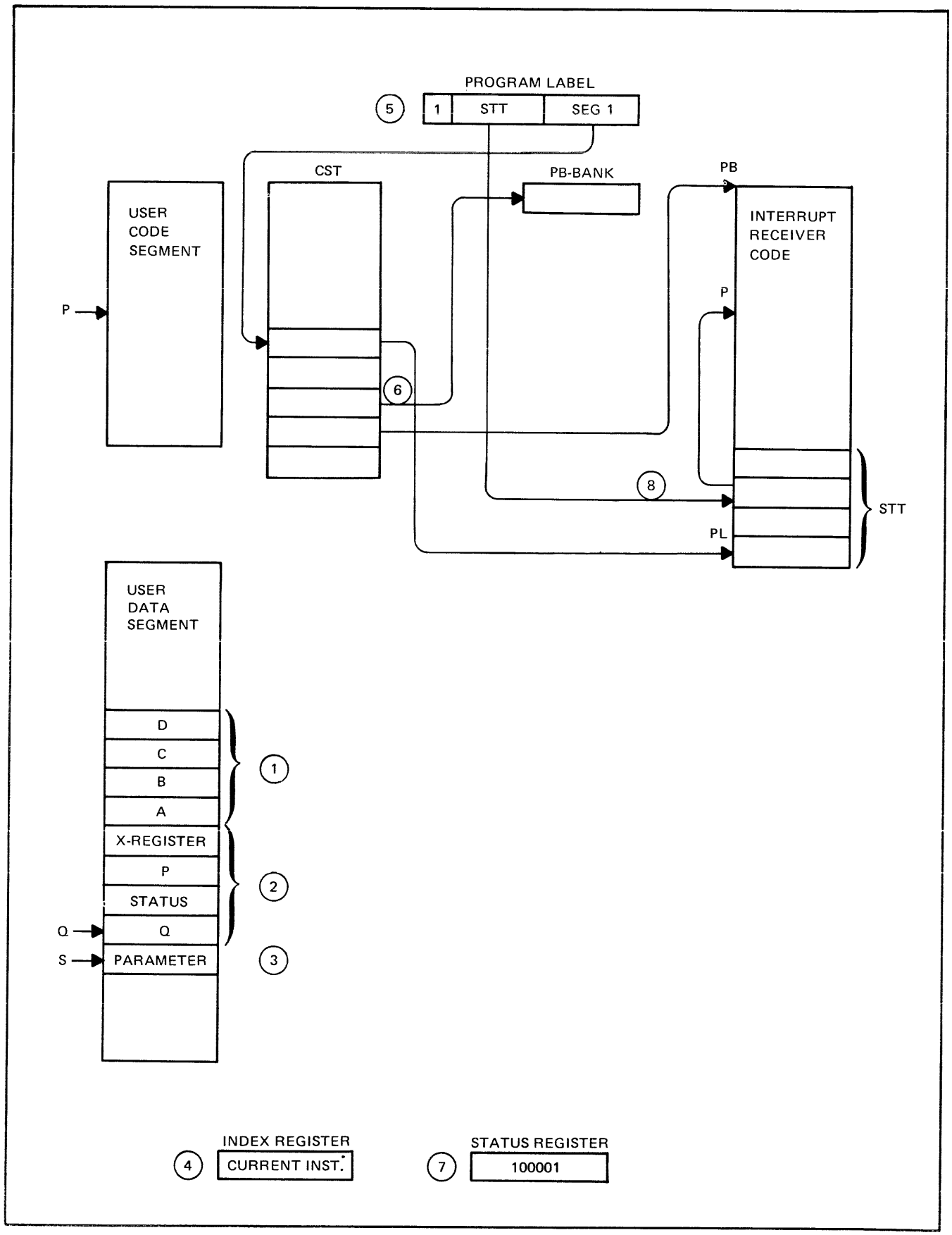


Figure 6-7. Non-ICS Internal Interrupt Operations

Figure 6-8 illustrates the operations performed by the Interrupt Handler. Generally, the sequence begins with the START block at the top left corner and ends with the Next CPU Instruction block at the bottom right corner.

6-50. DISP INSTRUCTION

The DISP instruction calls the Dispatcher which is a system process whose primary function is determining which active process will use the CPU and then transfer control to that process.

The Dispatcher can be called from a user program if in privileged mode. For example, the last instruction of a user process is a PCAL to a system process called TERMINATE' which, among other things, cleans up the CST, DST, and PCB entries for the user process. TERMINATE' then issues a DISP instruction. Some system error handling routines such as trap handlers may use it to call the Dispatcher after aborting the user program.

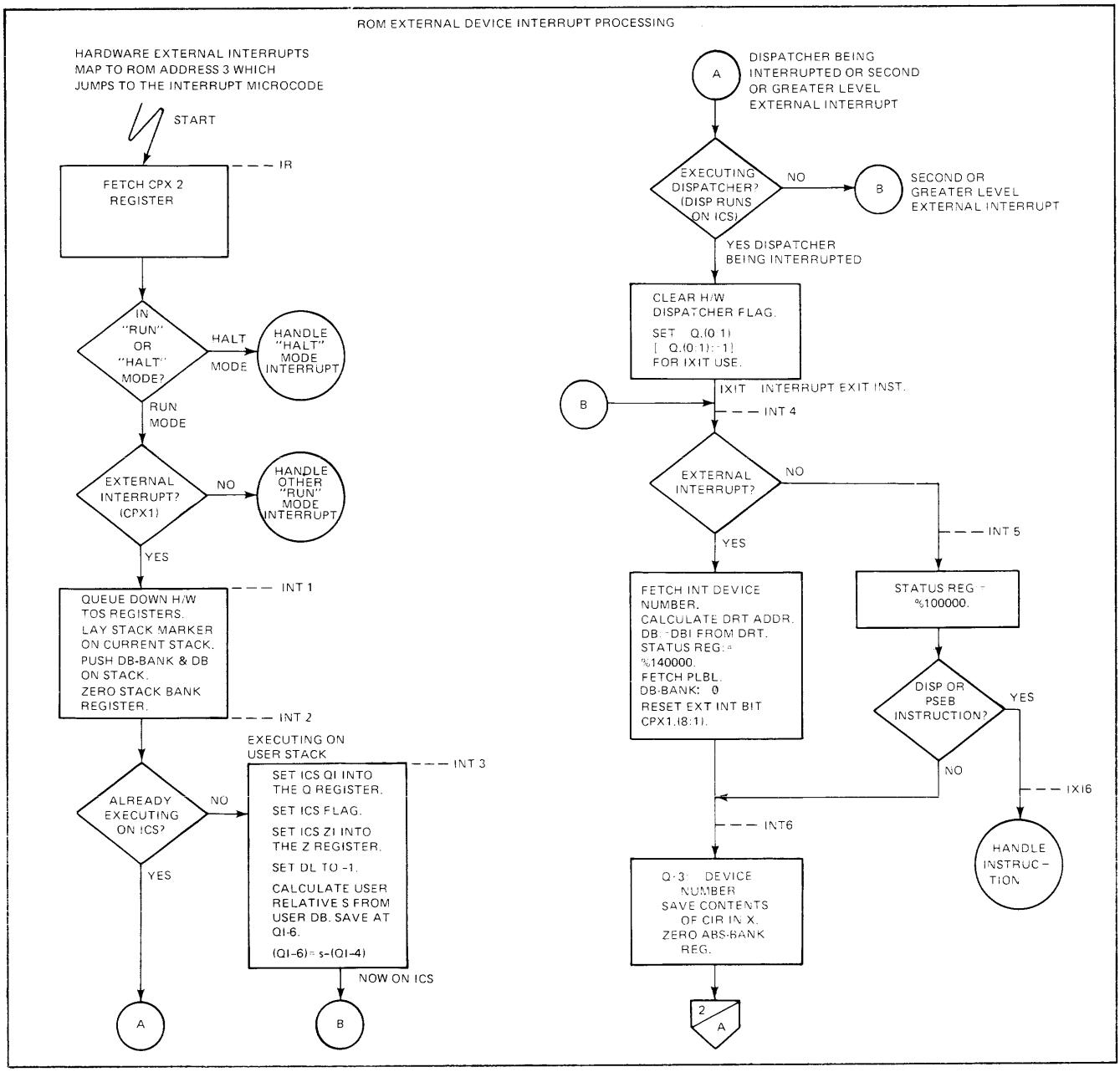


Figure 6-8. Interrupt Handler

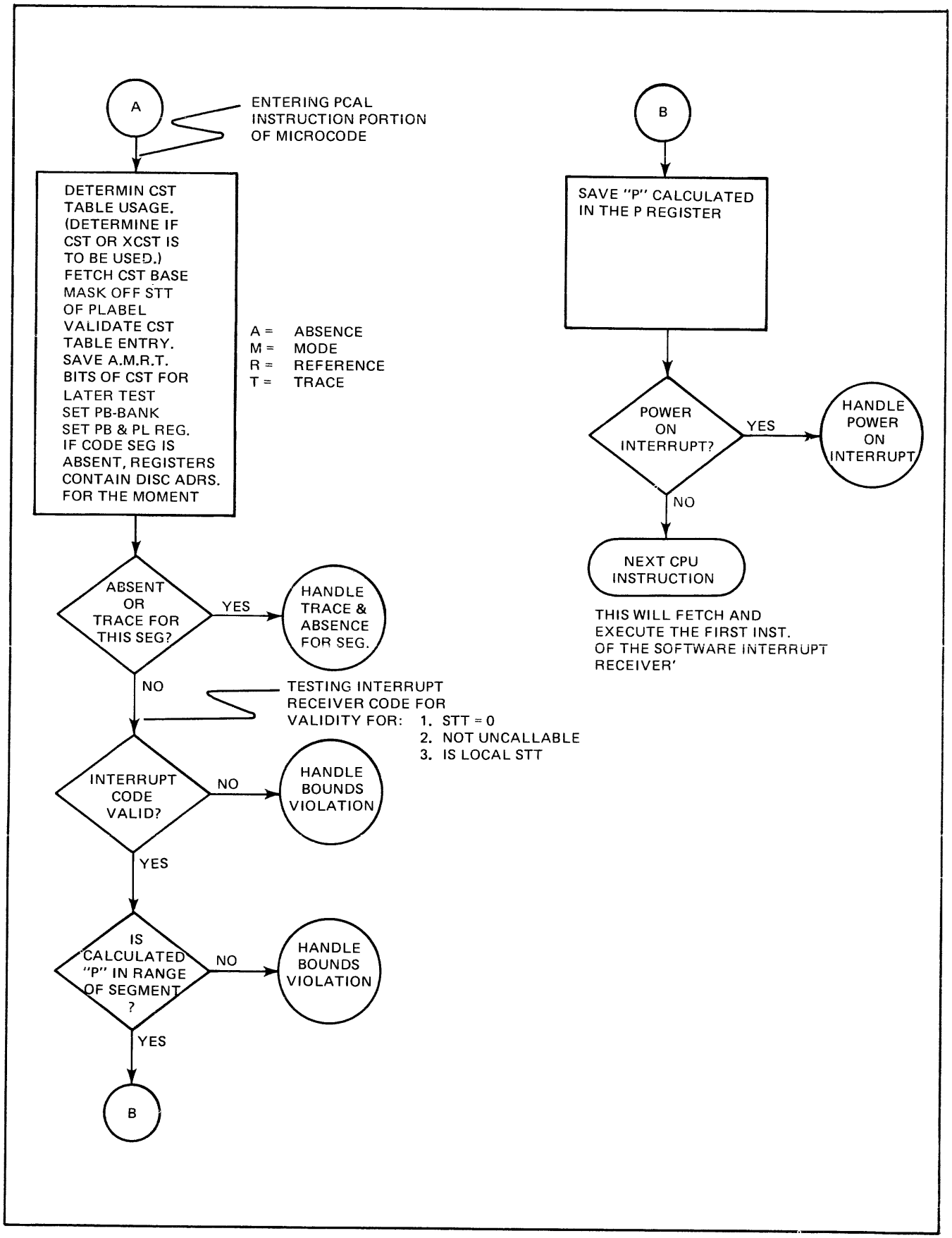


Figure 6-8. Interrupt Handler (Continued)

The DISP instruction can be executed by an Interrupt Handler after servicing all pending interrupts from a multiple Device Controller. In this case, the Dispatcher is not actually called, but instead a condition code of CCG is set and bit 0 of QI is set to instruct the IXIT instruction what to do. The next CPU instruction after the DISP instruction would then be executed and the Interrupt Handler would execute the IXIT instruction. The IXIT instruction then uses bit 0 of QI to determine which path to take.

All programs which use the DISP instruction must be prepared to handle the condition of the Dispatcher being pseudo disabled.

6-51. PSEUDO ENABLING/DISABLING THE DISPATCHER.

The PSDB (Pseudo Disable) and PSEB (Pseudo Enable) instructions are used to pseudo disable and enable the Dispatcher. The two instructions must be executed in pairs — for each disable, there must be a corresponding enable within the same process. The Dispatcher can be locked several levels deep with PSDB instructions, but must have one PSEB to unlock each level. A count is maintained in QI-18 for the number of disables which have not been unlocked.

These instructions are used to prevent a dispatch during critical sections of code and to avoid unnecessarily restarting the Dispatcher.

If the DISP instruction is executed and the Dispatcher is disabled (QI-18 is non-zero), then bit 0 of QI is set to 1 and the next CPU instruction is fetched. This bit is reset either by IXIT or PSEB when QI-18 becomes zero.

If QI-18 is already zero at the start of a PSEB instruction, a system halt will occur.

6-52. IXIT INSTRUCTION

Figure 6-9 is a simplified flowchart of the IXIT instruction. IXIT operates in either of two ways. The first (as denoted by (1) in the figure), is by the Dispatcher to transfer to a process being launched. The second (as denoted by (2) through (6) in the figure), is to exit from ICS type interrupt routines.

If the interrupt service routine is not in segment number 1, it is assumed to be an external interrupt routine and a Reset Interrupt is sent to the device whose device number is in Q+3.

If bit 0 of Q is zero and if Q= QI, the return is to the interrupted process (2). Otherwise the return is to a lower priority interrupt which was interrupted (3).

If bit 0 of Q is 1 and bit 0 of QI is zero, the return is to the Dispatcher which was interrupted (4).

If both bit 0 of Q and bit 0 of QI are 1, a DISP instruction has been executed and the request to start the Dispatcher is still pending. If QI-18 is zero, the Dispatcher is not disabled, QI is cleared, and a transfer is made to the Dispatcher's entry point (5) or (6). It doesn't matter whether a process (Q= QI) or the Dispatcher (Q not equal to QI) was interrupted. If QI-18 is non-zero, the Dispatcher is disabled and the DISP request cannot be carried out at this time. Instead, IXIT returns to the interrupted Dispatcher (Q not equal to QI (4a)), or to the interrupted process (Q = QI (2a)). The Start Dispatcher request is still pending, (bit 0 of QI is 1).

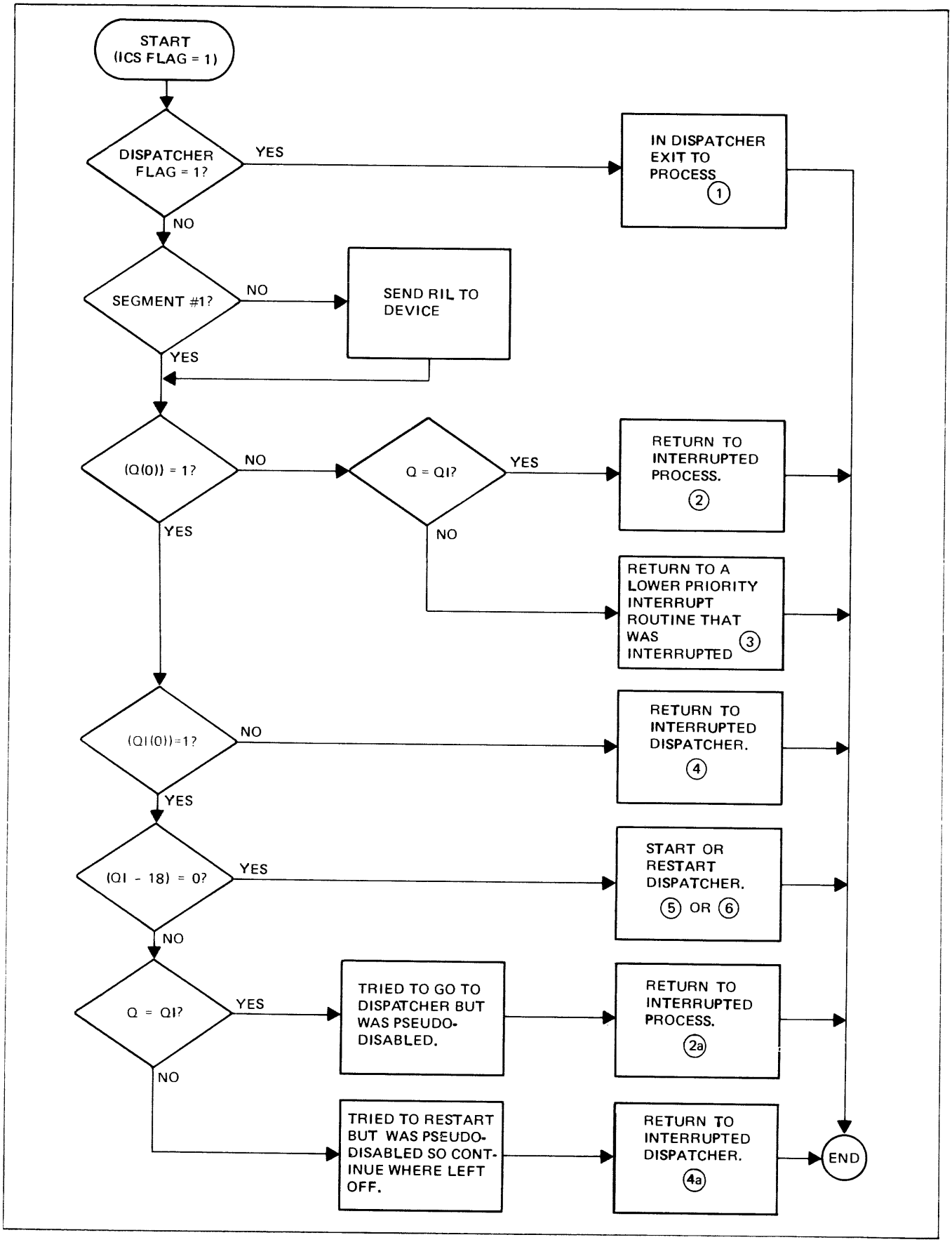


Figure 6-9. IXIT Instruction

FUNCTIONAL OPERATION

SECTION VII

7-1. INTRODUCTION

This section provides a functional description of the HP 3000 Series II and III Computer Systems. The following units are described:

- Central Processor Unit (CPU)
- Module Control Unit (MCU)
- Memory Module
- Input/Output Processor (IOP)
- Multiplexer Channel
- 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.

There are ten simplified logic diagrams for each system that, when combined, form a simplified logic diagram of the HP 3000 Series II or III Computer System. Figure 7-1 is a plan view showing where the individual diagrams fit into the whole.

7-2. CENTRAL PROCESSOR UNIT

The Central Processor Unit (CPU) was described in detail in Section III. The CPU simplified logic diagram, figure 7-2, is kept in this section, however, to be near the simplified logic diagrams of the other system modules.

7-3. MODULE CONTROL UNIT

Each module gains access to the central data bus (CTL Bus) by way of its Module Control Unit (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 the same function of interfacing with the other modules connected to the central data bus.

The Module Control Unit for the CPU (see figure 7-3) is representative of the other MCUs in the system. Because the purpose of the MCU is to effect bus transmission, the logic is best described dynamically by following the sequence of operations involved in different types of bus transmissions.

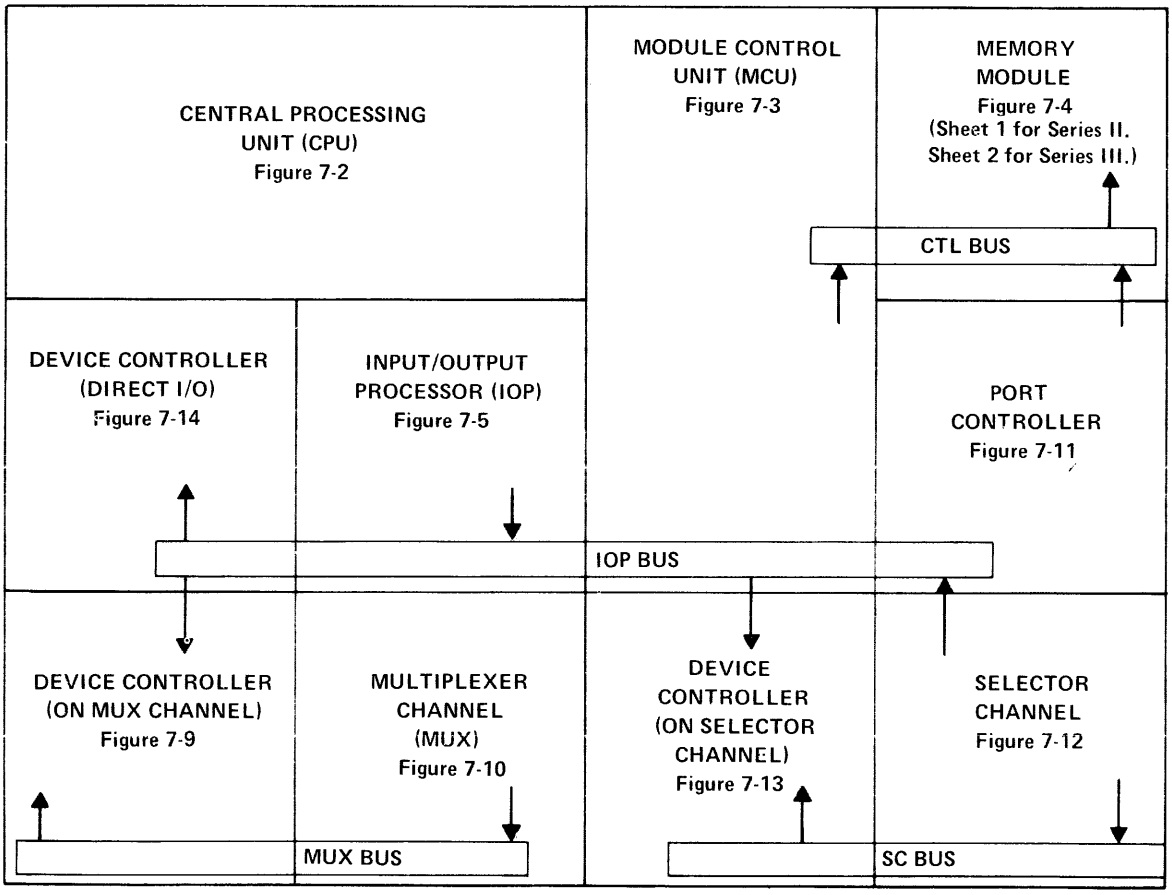
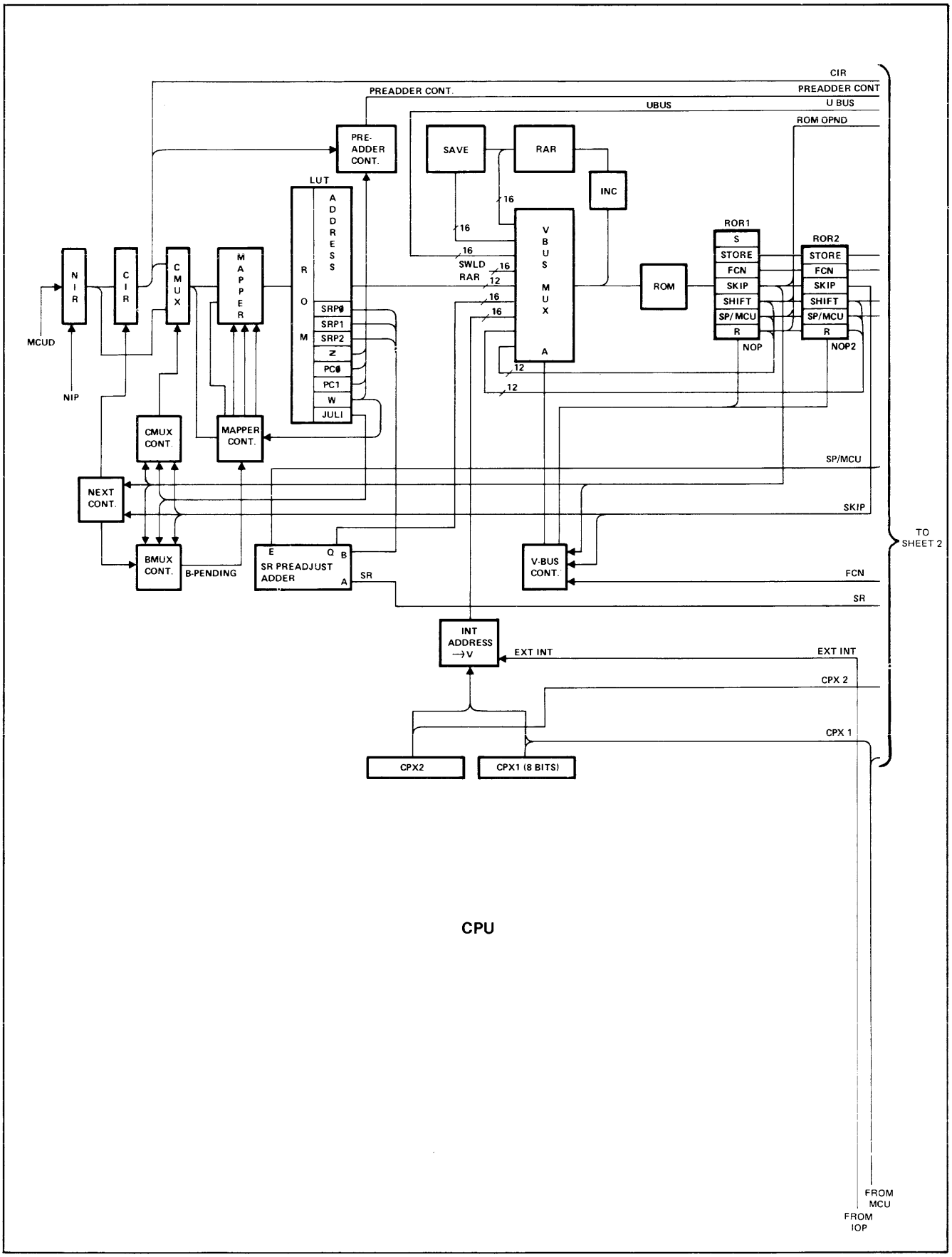


Figure 7-1. Simplified Logic Diagram, Plan View



CPU

Figure 7-2. Central Processor Unit (CPU) Simplified Logic Diagram (Sheet 1 of 2)

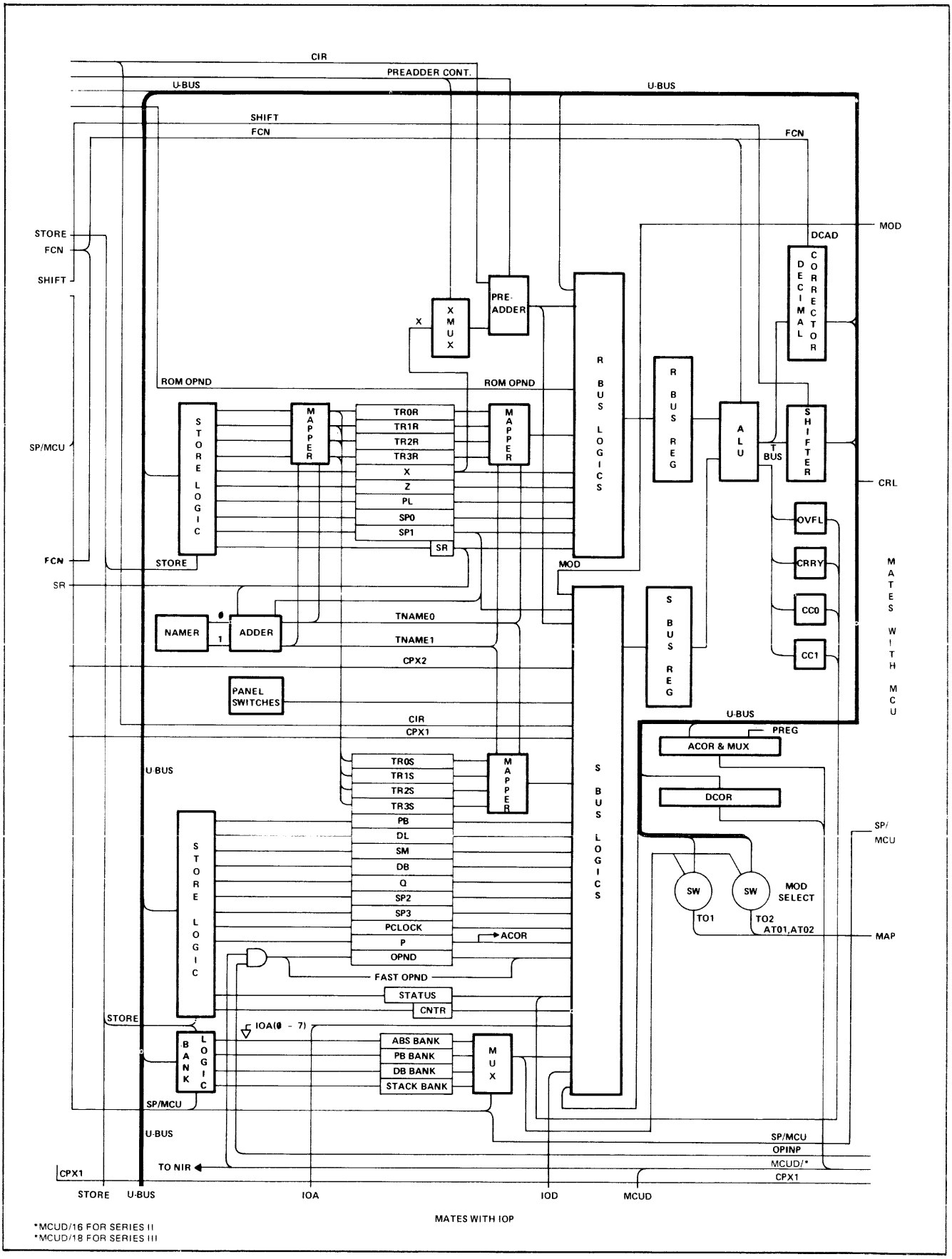


Figure 7-2. Central Processor Unit (CPU) Simplified Logic Diagram (Sheet 2 of 2)

7-4. CENTRAL DATA BUS TRANSMISSIONS

The following procedures are discussed as operations of the central data bus:

a. Fetch Next Instruction
   1. CPU Address Transmit
   2. Memory Receive and Transmit
   3. CPU Receive

b. Fetch an Operand
   1. CPU Address Transmit
   2. Memory Receive and Transmit
   3. CPU Receive

c. Store an Operand
   1. CPU Address Transmit
   2. Memory Receive
   3. CPU Data Transmit
   4. Memory Receive

d. Command a Module

7-5. FETCH NEXT INSTRUCTION

7-6. CPU ADDRESS TRANSMIT. The first steps in fetching an instruction are to send an address to memory and to tell memory what to do with that address.

When a NEXT instruction 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 ACOR register (see figure 7-2). NEXT also transfers the contents of the next instruction register (NIR) into the current instruction register (CIR). The CPU executes the CIR contents.

The objective now is to refill NIR while the instruction in CIR is being executed so as to implement the CPU instruction look-ahead feature. Assuming that the transmission may proceed, NEXT sets the Low Request (LREQ) flip-flop (see figure 7-3) in the Module Control Unit. (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 because a memory operation is being initiated, whereas high request assumes that the destination module is expecting the transmission of data to complete a memory write operation.) By this time, the MCU Encoder 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. There are four possible memory opcodes: No Operation (NOP), Write (W), Read (R), and, for Series II only, Read Write Ones (RW1). (For Series III, the two-bit RW1 code from the MOP register is decoded as a NOP.) NEXT locks the code in the MOP register and sets the Next In Process (NIP) flip-flop. Setting NIP *opens* the next instruction register so that all central data bus transmissions are gated to NIR until NIP is reset. NEXT also locks the TO register, which now contains the destination module number.

The LREQ signal reads the contents of the TO register into the Ready Comparator, which checks the Ready (RDY) line for 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 the Enable (ENB) line low for this module number. Since no module can transmit unless all ENB lines of higher priority modules are high, pulling the ENB line low disables all lower priority modules. Provided that no higher priority module has pulled its ENB line low 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 CPU Select (SEL) flip-flop. The SEL signal reads out the

ACOR contents to the central data bus, as well as TO and FROM module numbers and the memory opcode. SEL also pulls the destination module's RDY line low for one cycle, so that other modules will not assume that the memory module is ready before memory has a chance to pull the RDY line low itself on the next cycle.

7-7. MEMORY RECEIVE AND TRANSMIT. The next step in the process is for the memory module to receive the address from the bus, read the contents of the addressed location, and transmit the contents back to the CPU.

NOTE

For simplicity, the memory module's error detection, logging, and correction operations are not discussed in this manual. Detailed discussions for these operations are contained in the HP 3000 Series II/III System Service Manual, part number 30000-90018.

For Series II, the TO comparator (see figure 7-4, sheet 1 for Series II and sheet 2 for Series III) on the memory module identifies the code on the TO lines as its own module number and sets the Ready flip-flop. The Ready signal (RDY) locks the address word from the bus into the address register, and locks the FROM address into the from register. For Series III, the TO comparator sets the Ready flip-flop and the Address Latch flip-flop which locks the address word into the address register and the FROM address into the from register. The Ready signal also keeps the modules 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 memory cycle. The addressed memory location is read into the data register. Meanwhile, on the next clock edge, the MCU begins the process of requesting access to the bus by setting the Enable flip-flop. (Since memory transmits only to modules that are expecting the transmission, only high requests are used.) The Enable signal pulls its enable (ENB) line low to lower priority modules and, provided no higher priority module has pulled low on its ENB to tTis module, sets the Data Out flip-flop on the next clock edge.

The memory location contents are in the data register, and the Data Out signal reads the contents out to the central data bus. The Data Out also reads out the wired FROM code and TO code (which is simply the saved FROM code, since transmission is back to the CPU).

7-8. 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 in the CPU.

The TO comparator on the CPU MCU (see figure 7-3) 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 is 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 MOD register, and a module interrupt is generated to the CPU. The two true outputs together reset the NIP flip-flop. The next instruction register, which up until now has been loading all bus transmissions into itself, is now inhibited from further loading, because it now contains the expected next instruction.

7-9. 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 details are necessary.

7-10. CPU ADDRESS TRANSMIT. The process of sending an address to memory begins when a signal from the ROM Store field loads the U-bus contents into the ACOR register on the CPU and

sets the LREQ flip-flop on the CPU MCU (see figures 7-2 and 7-3). The MCU Operation Encoder gives a memory opcode to the MOP register and sets the OPINP (Operand in Process) flip-flop. The LREQ signal causes the Ready comparator to check if the destination module is ready and, if so, enters the priority. When priority allows (ENB present), the Select flip-flop is set, causing the address stored in ACOR to be read out to the central data bus.

7-11. MEMORY RECEIVE AND TRANSMIT. The memory module (see figure 7-4), after recognizing its TO code and setting the Ready flip-flop, locks the address from the bus into the address register. The Ready signal, together with the decoded memory opcode, initiates the reading of the addressed location into the data register. Meanwhile, the Enable flip-flop is set and priority is established. When the module has priority to use the bus on the next clock cycle, the Data Out 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.

7-12. 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.

NOTE

If the CPU is frozen awaiting the operand, the operand in addition to being loaded into the OPND register, is also loaded into the CPU S-bus register, thus saving one clock of instruction execution time.

7-13. 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 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.

7-14. CPU ADDRESS TRANSMIT. A signal from the ROM Store field loads the U-bus contents into the ACOR register on the CPU and sets the LREQ (low request) flip-flop on the CPU MCU (see figures 7-2 and 7-3). The MCU Operation Encoder gives a memory opcode to the MOP register; in this case the opcode is Write rather than Read as in the previous cases. (Neither the NIP nor OPIND flip-flops are set.) After checking to see if the destination module is ready and the Enable (ENB) signals are present, the LREQ signal sets the Select flip-flop which causes the address to be read out to the central data bus.

7-15. MEMORY RECEIVE. The memory module (see figure 7-4) after recognizing its TO code and setting the Ready flip-flop, locks the address from the central data bus into the address register. The FROM, MOP, and address registers remain locked, and the RDY line goes low so no other module can send a new address to this memory module.

7-16. 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 DCOR (see figure 7-2). The DATA signal also sets the High Request (HREQ) flip-flop on the CPU MCU (see figure 7-3). Destination readiness does not need to be checked, however, since memory is expecting a data transmission from this module. After priority checks, the HREQ signal sets the CPU 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.)

7-17. 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 start the memory write cycle. This causes the operand to be stored into the addressed location.

7-18. COMMAND A MODULE

The instruction set includes a Command instruction (CMD) which permits privileged mode 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 stack and sends it to a module whose module number (and two-bit opcode) are given in another word in the stack. (Refer to the *Machine Instruction Set Reference Manual* for the CMD instruction definition.)

WARNING

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in the privileged mode.

The ROM MCU field codes of Control (CRL) and Command (CMD), in that order, are used to effect the execution of the CPU instruction CMD.

When the hardware decodes a CRL in the MCU field, it gates bits 13 through 15 from the U-bus, through the TO MUX lines in the MCU, and these bits are clocked into the TO register (see figure 7-3). U-bus bits 10 and 11 are clocked into the MOP register. A following line of microcode then executes a CMD MCU function which gates that line's U-bus data into ACOR, and issues a Low Request (LREQ).

The MCU logic performs the normal sequence of checking for bus priority. Then the contents of the TO register are gated to the TO lines of the CTL bus, the contents of the FROM jumpers in the MOD register are gated to the FROM lines on the CTL bus, and the contents of the MOP register are gated to the MOP lines of the CTL bus. The contents of ACOR (see figure 7-2) are placed on the MCUD lines of the CTL bus. What affect the MOP and MCUD lines have on the addressed module depends on the design of the module.

When a module needs to communicate with the CPU, it cannot pass a data word to the MCUD lines because the MCU is not expecting the communication and will not gate the MCUD lines back. The CTL bus, however, since it does not use a handshake signal sequence, is monitoring the bus at all times and the CPU is able to detect that the module is trying to communicate. The hardware generates a module interrupt and sets CPX1 (bit 7). (Refer to Section VI for a description of interrupts.)

When the module has priority to use the CTL data bus, it places the CPU module number on the TO lines, CPU address on the FROM lines, and a value on the MOP lines. The MCU logic recognizes its own address, but because it is not expecting a value from the calling module, the FROM lines value is sent to bit positions 5, 6, and 7 of the MOD register and the MOP value is sent to bit positions 2 and 3 (see figure 7-3). The microcode will fetch these values from the MOD register and pass them to the interrupt handling CPU instruction.

To command the TO ME - FROM ME lines, the CPU instruction XEQ takes the 16-bit top-of-stack value and executes it as an instruction. In order to do this, the value must be placed in the next instruction register (NIR) before it can be gated to the current instruction register (CIR) to be executed. Because the CTL data bus is the only data path into NIR, the value is placed on the FROM lines of the CTL data bus with an address of TO the CPU (in essence, TO ME). The FROM lines will have the CPU's port number (FROM ME).

The TO ME - FROM ME action is caused by a code of NIR in the MCU field of a microcode instruction. When the MCU detects the NIR code, the High Request (CPU HREQ) flip-flop and the Next-In-Process (NIP) flip-flop (see figure 7-3) are set and the value placed on the U-bus is clocked into DCOR (see figure 7-2). The hardware places a No Operation (NOP) code in the MOP register and clocks the FROM jumper value through TO MUX into the TO register. The MCU, when granted priority to use the CTL data bus, gates the contents of the TO, MOP, and DCOR register onto the CTL data bus. Because the MCU for the CPU is monitoring the CTL data bus at all times, the TO comparator recognizes that the CPU is being addressed, and that the CPU is expecting the communication (determined by the fact that the values on the FROM lines match the contents of the TO register). The MCU, therefore, will accept the value on the MCUD lines, and, because the NIP flip-flop has been set, the value is clocked into NIR.

The same sequence as described for TO ME - FROM ME will occur if an Operand (OPND) code is decoded in the MCU field of a microcode instruction. The only difference is that the OPINP flip-flop on the MCU (see figure 7-3), instead of the NIP flip-flop, is set, and the value will be gated into the OPND register instead of NIR.

7-19. 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 is 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.

7-20. I/O PRIORITIES

There are two types of priority to be considered in the I/O system:

- Interrupt Priority
- Service Priority

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.

The *interrupt poll* determines the priorities of all I/O interrupts. The interrupt poll originates in the I/O Processor (see figure 7-5) 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-wire pair (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 (INT REQ line in the IOP bus). The poll propagates through each non-requesting unit and stops at the first requesting unit it encounters. That unit then returns INT ACK (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 uses 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 levels. 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 7-6 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.

NOTE

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. Although four Selector Channel ports are available (three for Series III), two Selector Channels is the maximum system configuration. 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 tha channel can be honored until that time.

7-21. I/O DATA ROUTES

Data transfer routes for both low- and high-speed devices and direct I/O are shown in figure 7-7. At least one of each type of unit (two low-speed Device Controllers, one high-speed Device Controller, one Multiplexer Channel, two memory modules, etc.) is shown. The routes shown in figure 7-7 are the normal input/output data routes.

7-22. 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.

7-23. SELECTOR CHANNEL DEVICE. For direct I/O instructions , the data route is the same as for Multiplexer Channel device: 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 Selector Channel bus.

7-24. 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 is 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 7-7 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 Channel and a Selector Channel.

7-25. MULTIPLEXER CHANNEL. 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 7-8, note that the Multiplexer Channel includes a 16-location solid-state memory. Each location in this memory corresponds to one of the 16 Device Controllers connected to the Multiplexer Channel bus. Each location 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 executes the indicated operation. Then the information is updated for the next anticipated operation and is stored back in the memory location.

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 load 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 location (5). If the Device Controller was commanded to transfer data, it issues a service request when it is ready (6), causing another readout of the stored information (7) and a transfer of data (8); updated operation information is restored (9) Steps (6) through (9) are repeated for each word transferred.

7-26. SELECTOR CHANNEL. A Selector Channel transfers data from many sources in a data block manner. That is, it locks onto one Device Controller until the 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 Channel is designed to facilitate very high speed transfers.

The Selector Channel uses double-buffering for both data and I/O program words (see figure 7-8). 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 restored 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).

7-27. I/O PROCESSOR

The Input/Output Processor (IOP) is part of the CPU/IOP module (see figure 7-5). The MCU is shared by the CPU and the IOP.

7-28. IOP LOGIC

The purpose of the Input/Output Processor (IOP) is to: 1) execute direct I/O instructions and pass the results to the CPU, and 2) to transfer data and I/O program words between memory and the Device Controllers, so that the CPU may continue to execute other instructions without further intervention.

7-29. I/O COMMAND. The I/O instruction information is combined by the CPU into a single word, placed on the U-bus, and sent through the IOA logic to the IOP bus (see figure 7-5). The instruction from the code segment has been translated into a three-bit command (IOCMD). The command 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 device numbers (DEVNO) lines of the IOP bus. The Service Out (SO) bit tells the addressed device, via the IOP control, to accept and respond to the accompanying information. (The Device Controller must return a Service In (SI) handshake signal.)

7-30. 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 is discussed later when transfer sequences are described.

7-31. 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 IOA register. An External Interrupt (EXTINT) signal is issued to the CPU.

7-32. INT DEVNO. The I/O address (IOA) register holds the device number of the interrupting device so that, upon command, the CPU can read the contents onto the S-bus for interrupt processing.

7-33. DATA OUTPUT REGISTERS. There are two data output registers, the IOP data out register for memory data received from the central data bus, and the IOD data out register for direct data received via the U-bus from the CPU. Signals from IOP control can either read the contents out onto the IOP bus, or transfer the contents into MUX for restoring a DRT entry.

7-34. DATA INPUT REGISTERS. There also are two input registers. The IOP data in register is used for sending data to memory via the central data bus. This register is loaded either from the IOP bus or, for DRT entry restoring, from the IOP data out register. When doing a DRT store, the IOP data in register is incremented by two before the transfer is made. The second input register, IOD data in, may be used either as a direct data input register or as a memory address register. 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.

7-35. IOP MODULE CONTROL UNIT

The Module Control Unit (see figure 7-3) contains MCUs for both the CPU and the IOP. The MCUs operate basically in parallel, but not independently. Since both MCUs share the same access to the central data bus, and also share the same module number, it is necessary to resolve priority when both the IOP and CPU simultaneously attempt to use the bus.

Priority is resolved such that 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 memory write operation, having sent an address to memory, and the high request is an attempt to follow up by sending the data. On the other hand, the CPU low request represents the beginning of a transfer (attempt to send an address) and any IOP request will have priority over the CPU low request.

An IOP Request (IOP REQ) signal is generated when either a low request (IOLREQ) signal (see figure 7-3) or a high request (IOHREQ) is about to set one of the select flip-flops (IOLSEL or IOHSEL). The IOP REQ signal inhibits the setting of the CPU Select flip-flop. However, a CPU HREQ signal will inhibit IOLRQ from generating the IOP signal.

When data is returned from memory the FROM comparator compares the data with the contents of the TO register to check that the transmission is from the same memory module to which the address was sent. The TO comparator also checks that the transmission is to *this module*. Together, the outputs of the two comparators generate an I/O Strobe (IOSTRB) signal which locks the IOP data out register (see figure 7-5), because it now contains the correct information from the central data bus. The IOSTRB also tells the IOP that the data is ready for output via the IOP bus.

The MCU Ready comparator checks to see if a destination module is ready, or that an I/O low request signal can set the I/O Low Select (IOLSEL) flip-flop. Setting the IOLSEL flip-flop causes the contents of the IOP data in register, FROM, TO, and MOP signals to be read out onto the central data bus for transmission to memory.

7-36. 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 7-5). The IOP relays this information

to the Device Controller (see figure 7-9) via the IOP bus. The device number (DEVNO) on the IOP bus is compared with the internally wired device number. A true result, together with the Service Out (SO) signal from the IOP, enables the I/O Command (IOCMD) to be decoded. The IOCMD in this case is SIO which, when decoded, sets the Service Request (SR) flip-flop.

The Service Request (SR) along with a Request (REQ) signal is sent via the MUX bus to the Multiplexer Channel (see figure 7-10). The SR and REQ cause the Multiplexer Channel, instead of the Device Controller, to return Service In (SI) and force a DRT Fetch to be the first operation performed for the Device Controller on the next Service Request from the Device Controller. An SIO to a Device Controller temporarily inhibits service requests from all other Device Controllers, therefore, the only Device Controller requesting is the one receiving the SIO command. The Priority Encoder/Select Decoder then issues a 4-bit binary code which corresponds to the SR line number. The binary code is used as a RAM address, to enable one of the 16 locations in the solid-state MUX memory. The solid-state memory contains separate Random-Access Memories (RAM)for each of 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, and an Auxiliary RAM containing the I/O order. 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 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 a Service In (SI) signal to the IOP, which in turn, causes the IOP to free the CPU to execute other instructions.

For Series II, the Auxiliary RAM uses bits 14 and 15 of the Set Bank I/O order on the IOD lines to send IOX to the Mod Select switches on the IOP (see figure 7-5). The Mod Select switches supply an IOTO signal to the MCU (see figure 7-3), where it is gated to memory (see figure 7-4, sheet 1) via the CTL data bus as the TO signal.

For Series III, the Auxiliary RAM uses bits 12 through 15 of the Set Bank I/O order on the IOD lines to send IOX (B12 and B13) to the CPU Mod Select switches (see figure 7-5) and to send IOX (B14 and B15) to memory (see figure 7-4, sheet 2) as part of the 18-bit memory address. The IOP Mod Select switches (figure 7-5) supply an IOTO signal to the MCU in a manner similar to the Series II, where it is gated to memory (see figure 7-4, sheet 2).

For both the Series II and Series III, the Multiplexer Channel will transmit a bank number of 0 unless actually moving data for a Read or Write order pair. In the following description, unless otherwise specified, the bank number will be considered zero.

7-37. DRT FETCH

The Service Request received at the Multiplexer Channel from the Device Controller (see figure 7-10) causes the Transfer/Control Logic to send a Multiplexer Channel service request (HSREQ) to the IOP and also sets the SR latch. Any of the 16 SR inputs can set this latch and generate an HSREQ signal; however, only the highest priority requests will be honored by the Priority Encoder.

The IOP, when it receives an HSREQ, issues a DATA POLL 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 address RAM location are loaded into the state, address, auxiliary, 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 SO (Service Out). This command tells the Device Controller to read out its device number to the IOP bus.

NOTE

Approximately 20 command and response lines shown as part of the Multiplexer Channel bus have not been individually identified; they represent greater detail than is required at this level of discussion.

The Device Controller, for a DRT fetch, reads out its device number (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 l3, 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 IOP proceeds to fetch the DRT entry (see figures 7-3 and 7-5). The IOP issues an IOLRQ to its MCU, with an appropriate MOP to read memory. When select occurs, the address is transmitted to memory. When memory returns the DRT entry contents, I/O Strobe (IOSTRB) loads the word into the IOP data out register. The IOP data out contents 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, restores 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.)

7-38. I/O PROGRAM WORD TRANSFERS

Each I/O program word consists of two words in bank 0 of main memory: the I/O Command Word (IOCW) and the I/O Address Word (IOAW). 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.

7-39. IOCW FETCH. The Service Request (SR) flip-flop in the Device Controller is still set from the previous procedure (DRT Fetch, see paragraph 7-36), so HSREQ is still present at the IOP. The IOP therefore issues a new DATA POLL. 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 DATA POLL is stopped from further propagation, and the Transfer/Control Logic is enabled again.

The contents of the Address 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 the 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 IOP issues an IOLRQ to the MCU. When priority allows, the MCU transmits the address to memory. When memory returns the IOCW,IOSTRB loads this word into the IOP data out register in the IOP. The IOP then reads the word out to the IOD lines and issues SO.

The Multiplexer Channel, on receiving SO, loads the IOCW into the Order RAM. If the order is Control, the Multiplexer Channel issues a command through the MUX bus so that the Device Controller will also load the IOCW into its control register. The contents of the address register/ counter is incremented by one and restored in the Address RAM. The next state, fetch or store IOAW, is stored in the State RAM.

The next operation, transfer of the IOAW, begins the same way for each of the orders. That is, SR to the Multiplexer Channel causes a HSREQ to be sent to the IOP. The IOP returns a DATA POLL which enables the Multiplexer Channel to load the addressed RAM location into the state, address, and order registers. Action after this point varies depending on the order that the IOCW tains.

7-40. IOAW FETCH. The Read, Write, Jump, Control, and Interrupt orders each cause an IOAW fetch. However, the action taken upon receipt of the IOAW is different in each case.

The IOAW fetch begins by reading the contents of the address register (incremented on the trailing edge of DATA POLL in the IOCW fetch procedure) to the IOD lines. The Multiplexer Channel also issues SI and the IOCMD "transfer from memory" to the IOP. The IOP issues IOLRQ with MOP to its MCU to request a memory read.

When memory returns the contents of the address location, IOSTRB loads it into the IOP data out register. The IOP then reads the contents of the IOP data out register to the IOD lines and issues SO. For Read, Write, Interrupt, 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 MUX 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.

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 Device 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.)

7-41. 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 out its status register contents (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.

7-42. 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, Set Bank, 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.

7-43. 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 a 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 (address to memory) 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.

7-44. 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 DATA POLL, which enables the Multiplexer Channel to begin its transfer. First, the addressed RAM location is read out to the state, address, auxiliary, and order registers. Then the address register contents are read out to the IOD lines and the auxiliary register to the IOX 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 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.

7-45. OUTPUT TRANSFER. When memory returns a data word, IOSTRB loads the word into the IOP data out 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 Device Controller to load the word on the bus into its 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 restores 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.

7-46. 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 Device Controller does this, 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 *Write* MOP, thus causing a data transmission to memory via the MCU bus.

Meanwhile, the Multiplexer Channel restores the contents of the state, address, auxiliary, 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.

7-47. 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 Device 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 restoring 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.

7-48. END OF TRANSFER BY DEVICE. On termination of a transfer by a device, the Device Controller issues an SR to the Multiplexer Channel The Multiplexer Channel responds with CHAN SO. The Device Controller returns a *device end* signal that causes the Multiplexer Channel to initiate a DRT fetch, thus advancing the I/O program to the next IOCW.

7-49. 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.

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. Setting the Mask flip-flop (see figure 7-9) allows the Interrupt Request flip-flop to set the Interrupt Latch flip-flop. The Mask flip-flop is set by the following conditions: 1) the IOP has issued an IOCMD of Set Mask (SMASK), and, 2) the mask word on the IOD lines include 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 forming 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 IOP to issue an IOCMD of SIL (Set Interrupt Logic) with the appropriate Device Number (DEVNO), which sets the Interrupt Request flip-flop (see figure 7-9). An Interrupt order causes the Multiplexer Channel to issue a Set Interrupt (SET INT) command to the Device Controller via the MUX bus. The Device Controller logic forces the Interrupt Request flip-flop to set. If the Mask flip-flop is set, setting the Interrupt Request flip-flop results in an Interrupt Request (INTREQ) signal to the IOP via the IOP bus. The INTREQ signal will cause the Interrupt Latch to set.

When the IOP (see figure 7-5) receives INTREQ and is ready to process the request, it returns INT POLL to determine the highest priority request. The first set latch encountered by the poll stops further propagation of the poll, which is then permitted to set the Interrupt Active (IACTIVE) flip-flop on the Device Controller (see figure 7-9). Setting the IACTIVE flip-flop causes the interrupt device number to be sent to the IOP via the DEVNO lines. An Interrupt Acknowledge (INTACK) signal also is sent, telling the IOP to load the DEVNO into its IOA register.

When the IOP has the non-zero device number, it issues an External Interrupt (EXTINT) signal to the CPU, so that the interrupt processing may begin when the CPU is ready.

7-50. SELECTOR CHANNEL

A Selector Channel operates only one I/O program and transfers blocks of data for only one device at a time (vs. the Multiplexer Channel which operates up to 16 devices).

Data is passed back and forth from the memory module, through the Central Data Bus, to the Port Controller, Selector Channel, and through the Selector Channel Bus to the Device Controller and the operating device (see figure 7-7).

Since there may be two Selector Channels operating in the system, the Port Controller will be described first to explain how each Selector Channel gains access to the central data bus.

7-51. PORT CONTROLLER

The Port Controller (see figure 7-11) provides four ports to the central data bus (port 2 is not available for use with Series III) for I/O programs and data transfers between Selector Channels and memory modules. Only one-fourth of the logic is shown in figure 7-11; logic for the remaining ports is identical to the one shown. The port controller bus contains five sets of signal lines — one set for each of the four Selector Channel ports — and one set of data lines which is shared by all four Selector Channel ports. It should be noted that although four Selector Channel ports are provided, two Selector Channels is the maximum system configuration.

For Series II, the Port Controller is assigned module number 4 which gives the Port Controller a transmission priority higher than the CPU/IOP module.

A Selector Channel requiring transfer of a word to or from memory presents the Port Controller with a request for a Write or a Read operation, respectively, along with the memory number (0, 1, 2, or 3) to which the address will be sent.

A Write operation consists of a Low Request (LREQ) for an address transfer followed by a Low Select (LSEL) of that address from the Selector 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 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. While one section of the Port Controller is waiting, another section could be instructing another part of memory to fetch or store a data word for another Selector Channel.

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 Selector Channel 1, next to Selector

Channel 2 (Series II only), next to Selector Channel 3, and last to Selector Channel 4. A High Request for any Selector Channel takes precedence over all Low Requests.

The Write sequence is as follows: A Write on the request lines to the Port Controller sets the LREQ flip-flop and sets the MOP flip-flop to the Write state. The TO lines from the Selector Channel are clocked into the TO register, and the content is then compared with the Ready (RDY) line for that module. When the destination is ready, the ENB is present, and the Port Controller has priority, the LSEL and HREQ flip-flops are set. LSEL gates the address from the Selector Channel to the CTL bus along with TO, FROM, and MOP. LSEL also pulls the destination's RDY line low. Then, when ENB is present, the HSEL flip-flop is set. HSEL gates data from the Selector Channel to the CTL bus, along with TO, FROM, and MOP.

The Read sequence is as follows: A Read on the request lines to the Port Controller sets the LREQ flip-flop and sets the MOP flip-flop to the Read state. The TO lines from the Selector Channel are clocked into the TO register, and the content is compared with the RDY line for that module. When the destination is ready, the ENB is present, and the Port Controller has priority, the LSEL flip-flop is set. LSEL gates the address from the Selector Channel to the CTL data bus along with TO, FROM, and MOP. LSEL also sets the Wait flip-flop. Then, when returning data is present on the bus, the TO and FROM comparisons match, and a STRB signal is sent to the Selector Channel to tell it to accept the data on the Port Controller data (PCD) lines.

7-52. INITIATOR SEQUENCE

The following procedure describes how the Selector Channel's program counter is initialized as the first step in executing an I/O program for one device.

The selector channel bus originates at the Selector Channel and is routed to all Device Controllers controlled by this Selector Channel. The selector channel bus is similar to the MUX bus in purpose, but differs in that it uses 16 lines for transfer of control, status, and data words between the Device Controller and Selector Channel, whereas the corresponding MUX bus lines are used as service request lines for up to 16 devices.

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 7-7). 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 Device Controller to return SI to the I/O Processor with a 16-bit status word on the IOP bus (see figure 7-12). 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 Selector 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 Device Controller.

Again, the DEVNO on the bus is compared with the internally wired device number (see figure 7-12), and the true result, with SO, enables the I/O Command (IOCMD) to be decoded. The IOCMD is now SIO which, when decoded, issues a Request (REQ) signal to the Selector Channel control logic. The channel then returns Service In (SI) to the IOP 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.

When the Selector Channel (see figure 7-13) receives a Request (REQ) from the Device Controller, it sets the control logic to *Active*. The Selector Channel then issues the Device Number Data base (DEVNO DB) to the Device Controller. The Device Controller gates the DEVNO — left shifted by two — onto the SR (Data) lines of the Selector Channel bus. The DRTE address is then loaded into the DEVNO DB register. The Selector Channel is now exclusively reserved for that device. Furthermore, only this Device Controller will respond to Channel Service Out (CHANSO) from the Selector Channel.

The Selector Channel now reads the device number from the DEVNO DB register, and requests a memory transfer by issuing a Read to the Port Controller (see figure 7-11). The Port Controller checks if memory is ready and, when Enable (ENB) is present, sets the LSEL flip-flop. The LSEL signal is returned to the Selector Channel (see figure 7-13), where it reads the DRTE address onto the PCD lines on the PC bus. 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 Selector 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, described next.

7-53. 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 read. 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 incremented immediately.)

When memory returns the IOCW from the addressed location, the Port Controller issues STRB to the Selector Channel. The Selector 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 Selector 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.

7-54. EXECUTE SEQUENCES

There are nine I/O orders. A separate description is given for the execute sequence of each of the I/O orders. In each case except END, which terminates the I/O program, operation returns to the fetch sequence following completion of the execute sequence, in order to fetch the next I/O program word. The nine I/O orders are:

- Sense
- Return Residue
- Interrupt
- Jump
- Control
- Set Bank
- Read
- Write
- End

7-55. SENSE. The Selector Channel issues a P STATUS STB signal to the Device Controller, with CHANSO, via the selector 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). The bank number becomes the TO address. A number is either loaded into the bank register or Bank 0 is picked up at the Bank Gate (see figure 7-13), gated through the MOD select switches, and sent as the TO address to the Port Controller via the TO lines of the port controller bus. A 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.

7-56. INTERRUPT. The Selector 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 INT REQ to the I/O Processor via the IOP bus. When the I/O Processor returns INT POLL, the device number is sent to the I/O Processor, along with INT ACK. On receipt of INT ACK, the I/O Processor generates an interrupt signal to the CPU.

7-57. 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.

7-58. CONTROL. The Control order routes both the IOCW and the IOAW to the Device Controller. The Selector Channel first reads out the contents of the IOCW active register to the channel DATA lines and issues a PCMD1 (Programmed Command One) 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.

7-59. SET BANK. When requesting a memory Read or Write (for data words only), an IOAW word goes into the Selector Channel (see figure 7-13) on the PCD lines and the two least significant bits (Series II) or four least significant bits (Series III) are loaded into the Bank Register by the Set Bank order. For Series II, the contents of the Bank Register (two bits) are gated through the Mod Select switches, placed on the PC bus, and sent to the Port Controller (see figure 7-11) to become the Memory Module's (see figure 7-4, sheet 1) TO signal on the CTL data bus. For Series III, two bits from the Bank Register (TO1-1 and TO1-2) are gated through the Mod Select switches (see figure 7-13) and Port Controller to become the Memory Module's TO signal in a manner similar to the Series II. Additionally for the Series III, two more bits from the Bank Register (PB14 and PB15) are applied back through the Port Controller via the PC bus (see figure 7-13) to become part of the Memory Module's 18-bit address (see figure 7-11 and figure 7-4, sheet 2) on the CTL data bus.

7-60. 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 describe these two operations. To begin the Read execute sequence, the Selector Channel issues CHANSO to the Device Controller (see figures 7-12 and 7-13). When the controller returns CHAN ACK, the Selector Channel issues the initial Read Next Word (RD NXT WD) 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 Channel Service Request (CHAN SR) 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 End of Transfer (EOT) 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 Write Request and a mapped TO code to the Port Controller.

When the Port Controller (see figure 7-11) 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 Controller 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.

7-61. 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 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.

7-62. 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 Request and the bank register as a TO address 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 pre-fetched 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.

7-63. 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 DB register, enables the I/O Program Counter for use as data, and sends a Write request and a TO= 0 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.

7-64. DIRECT I/O OPERATION

In addition to the SIO modes of transfer, there is a direct I/O (DIO) mode. In the direct I/O mode, the CPU transfers information directly to and from a Device Controller without involving Memory, Multiplexer Channel, or the Selector Channel (see figure 7-14). For each I/O instruction, one word is transferred either to or from the CPU top-of-stack. The CPU has the following four direct I/O instructions:

- Test I/O (TIO)
- Control I/O (CIO)
- Read I/O (RIO)
- Write I/O (WIO)

NOTE

Some Device Controllers cannot accept all DIO commands (see the specific subsystem manual). However, all Device Controllers will accept a TIO or CIO using bits 0 and/or 1. Bit 0, the standard control bit, causes a master clear of the subsystem. Bit 1 causes the subsystem interrupt logic to reset.

7-65. TIO

The Test I/O (TIO) instruction obtains the contents of the Device Controller's status register (see figure 7-9) and pushes it onto the top of the stack. When the CPU encounters a TIO instruction, its TIO microprogram sends a command word to the IOP Control circuit (see figure 7-5) in the I/O Processor (IOP). The IOP then issues a service out (SO) and a TIO command on the IOCMD lines via the IOP bus to the device addressed by the device number (DEVNO) code. The addressed device is therefore

enabled to accept and decode the command, and accordingly, reads the contents of the status register onto the IOD lines. SI is also issued which causes the IOP to load the Status word into the IOD data in register and informs the CPU that the word is present. The CPU then issues a read signal which reads the contents of the IOD data in register to the S-bus. From the S-bus, the status word is placed on the U-bus and pushed onto the stack.

7-66. RIO

The Read I/O (RIO) instruction begins by performing a TIO to the Device Controller (see paragraph 7-65) to check the Read/Write OK status bit (bit 1). If status is acceptable, the same sequence is repeated except that the command is RIO and data is transferred from the Data In Buffer rather than the Status register.

7-67. 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 micro-program loads the RA contents into the IOD data out register, and then issues a command word to the IOP. 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 IOD data data 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 signal to the CPU, indicating completion of the instruction.

7-68. WIO

The Write I/O instruction begins by performing a TIO to the controller (see paragraph 7-65) 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 Device Controller's Data Out Buffer instead of the control register.

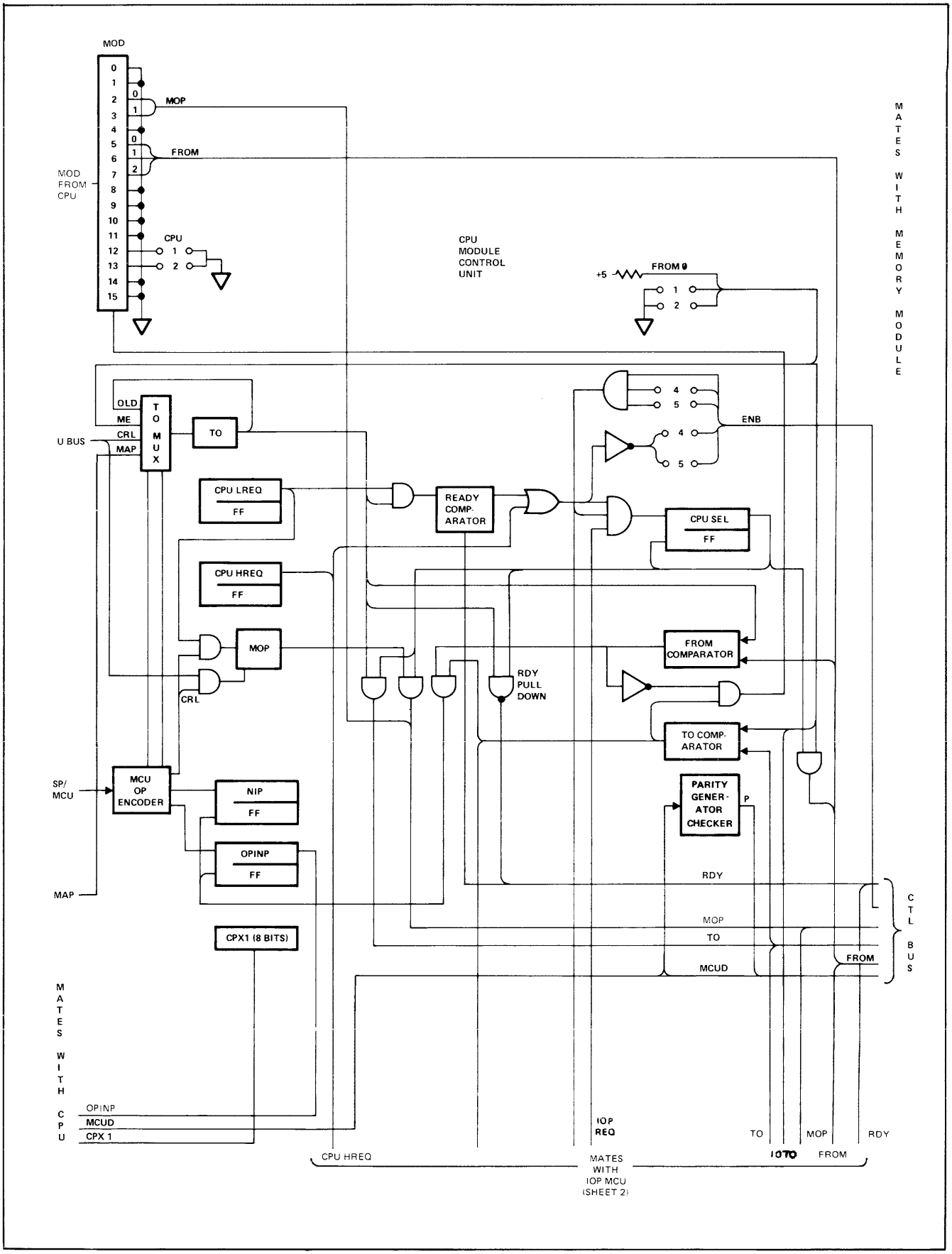


Figure 7-3. Module Control Unit (MCU) Simplified Logic Diagram (Sheet 1 of 2)

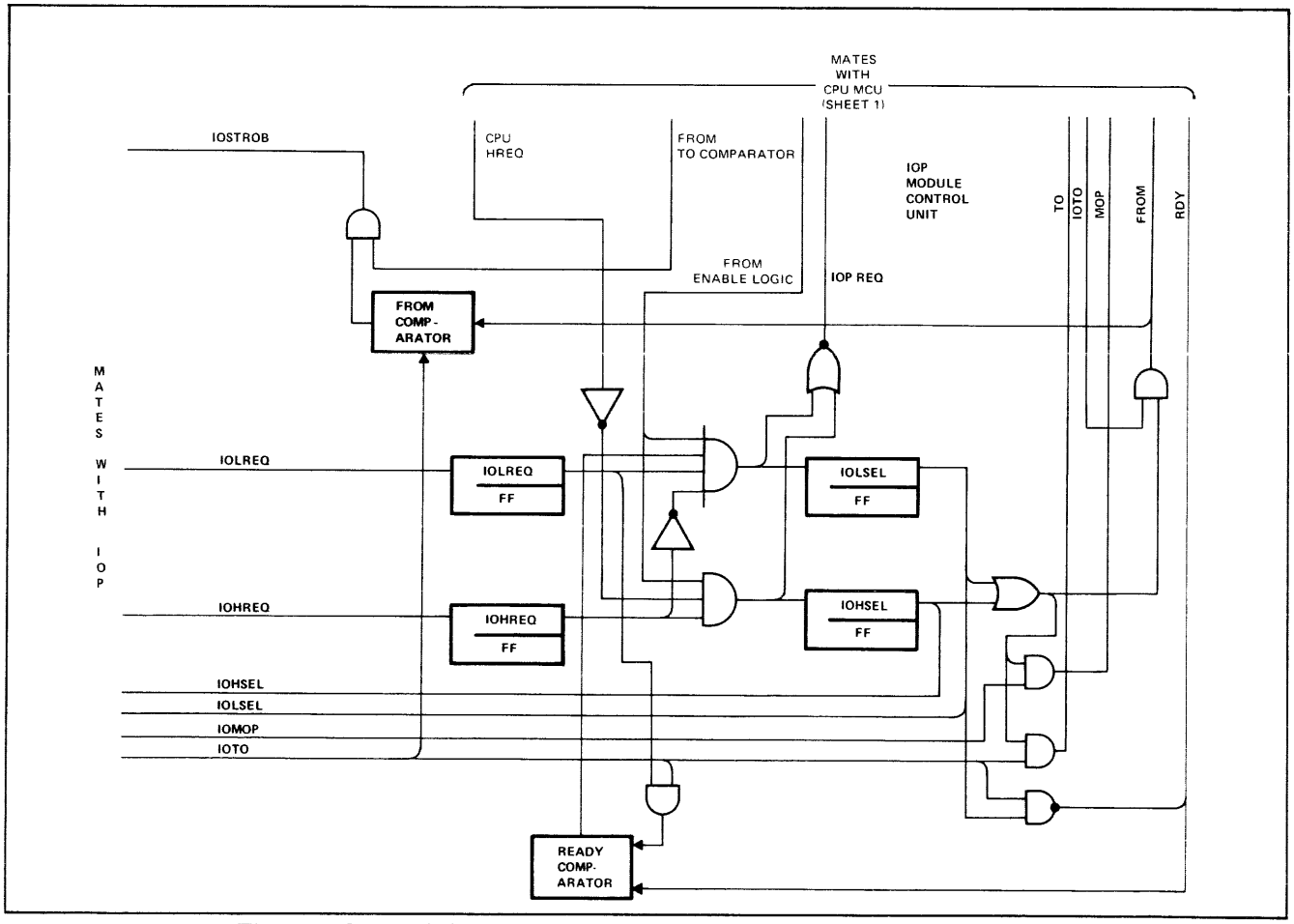


Figure 7-3. Module Control Unit (MCU) Simplified Logic Diagram (Sheet 2 of 2)

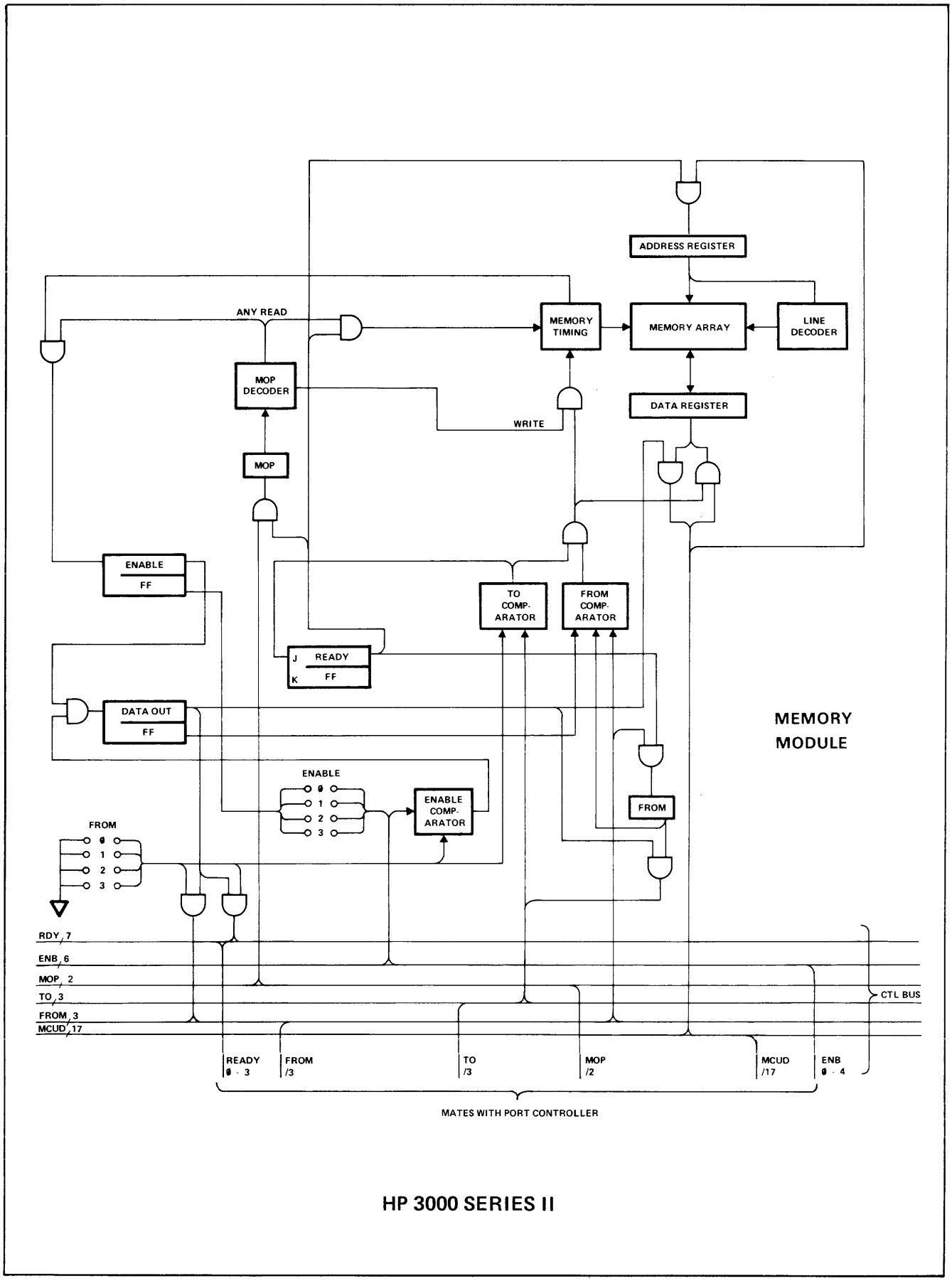
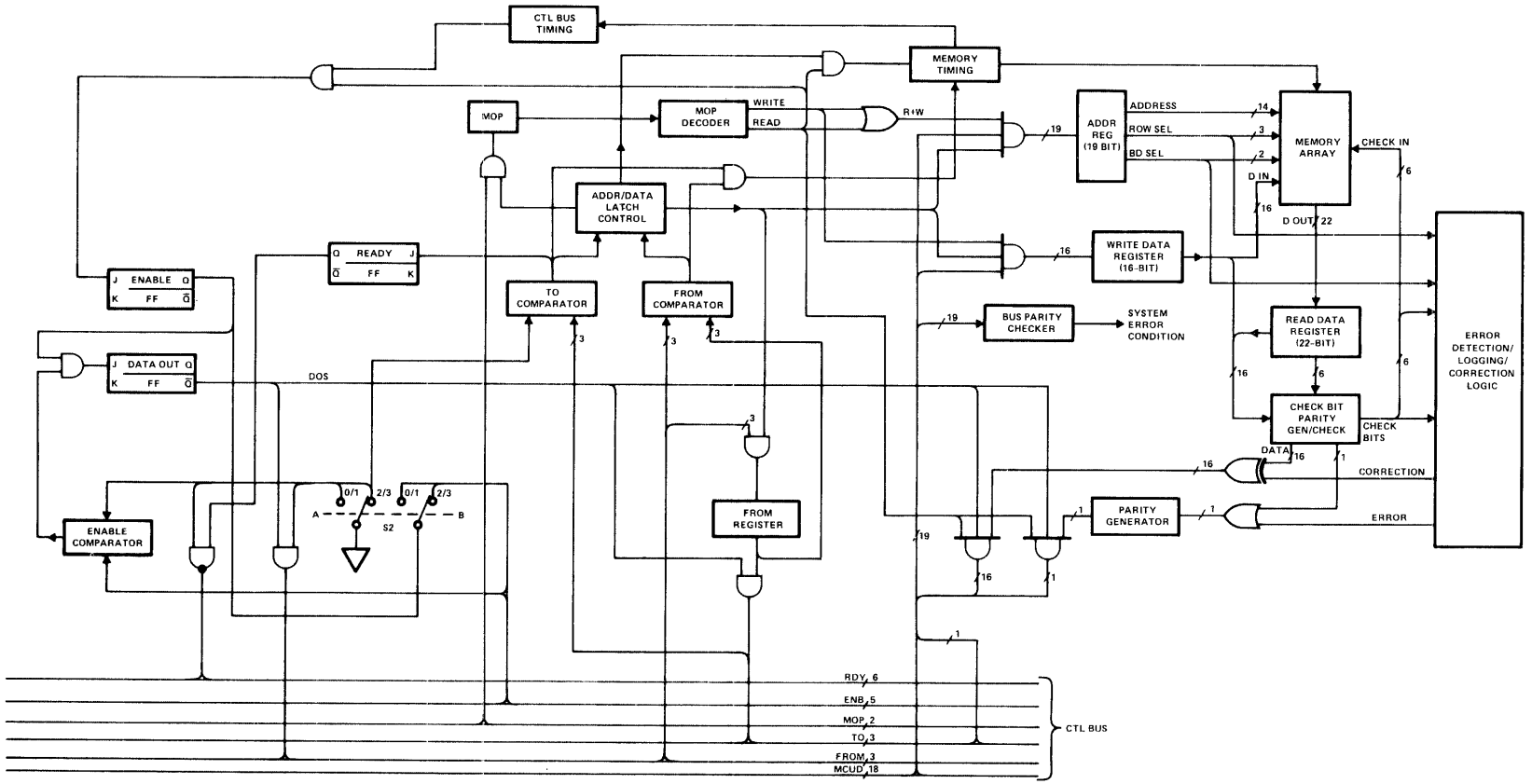
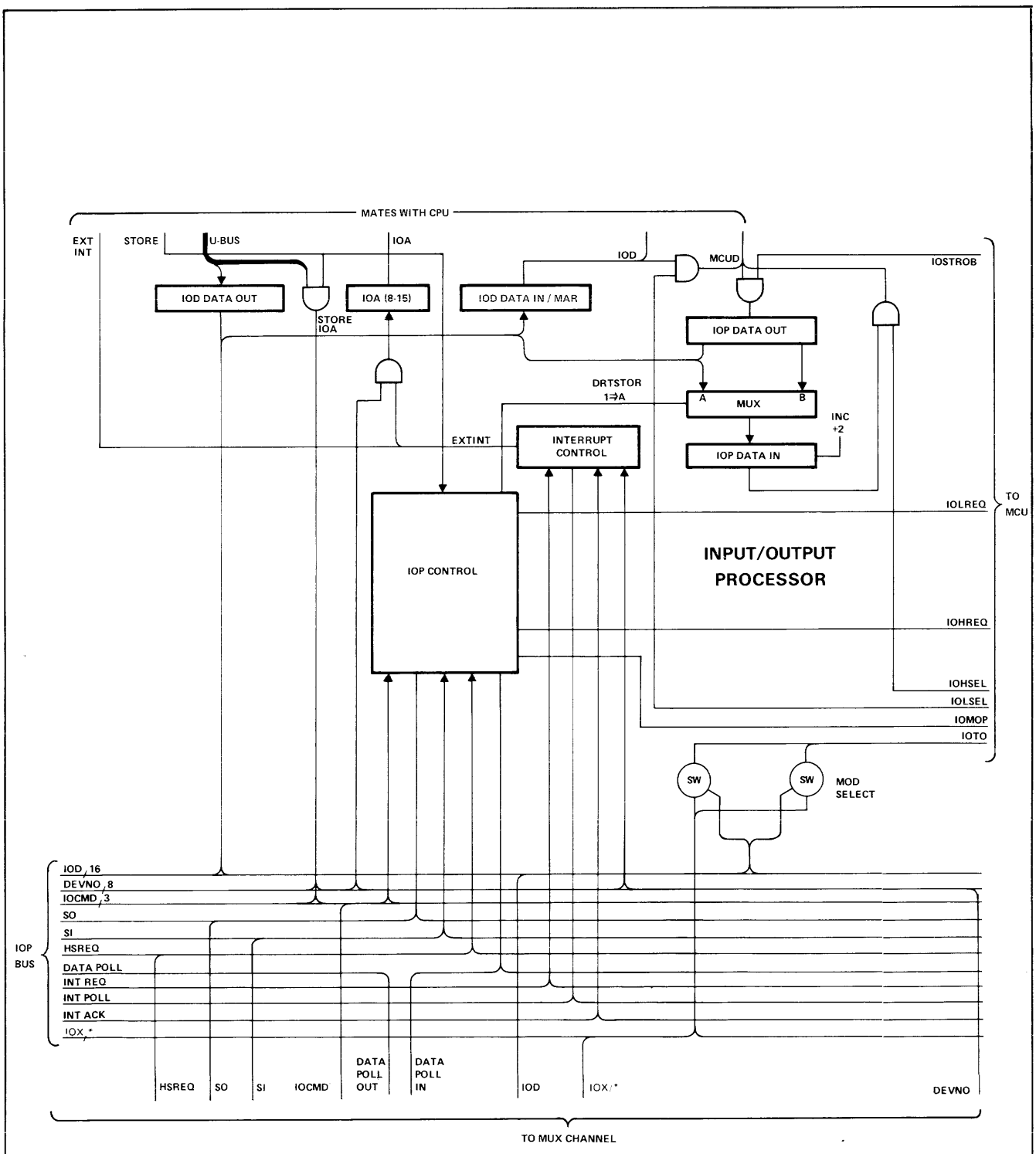


Figure 7-4. Memory Module Simplified Logic Diagram (Sheet 1 of 2)



HP 3000 SERIES III

Figure 7-4. Memory Module Simplified Logic Diagram (Sheet 2 of 2)



*IOX/2 FOR SERIES II
IOX/4 FOR SERIES III

Figure 7-5. Input/Output Processor (IOP) Simplified Logic Diagram

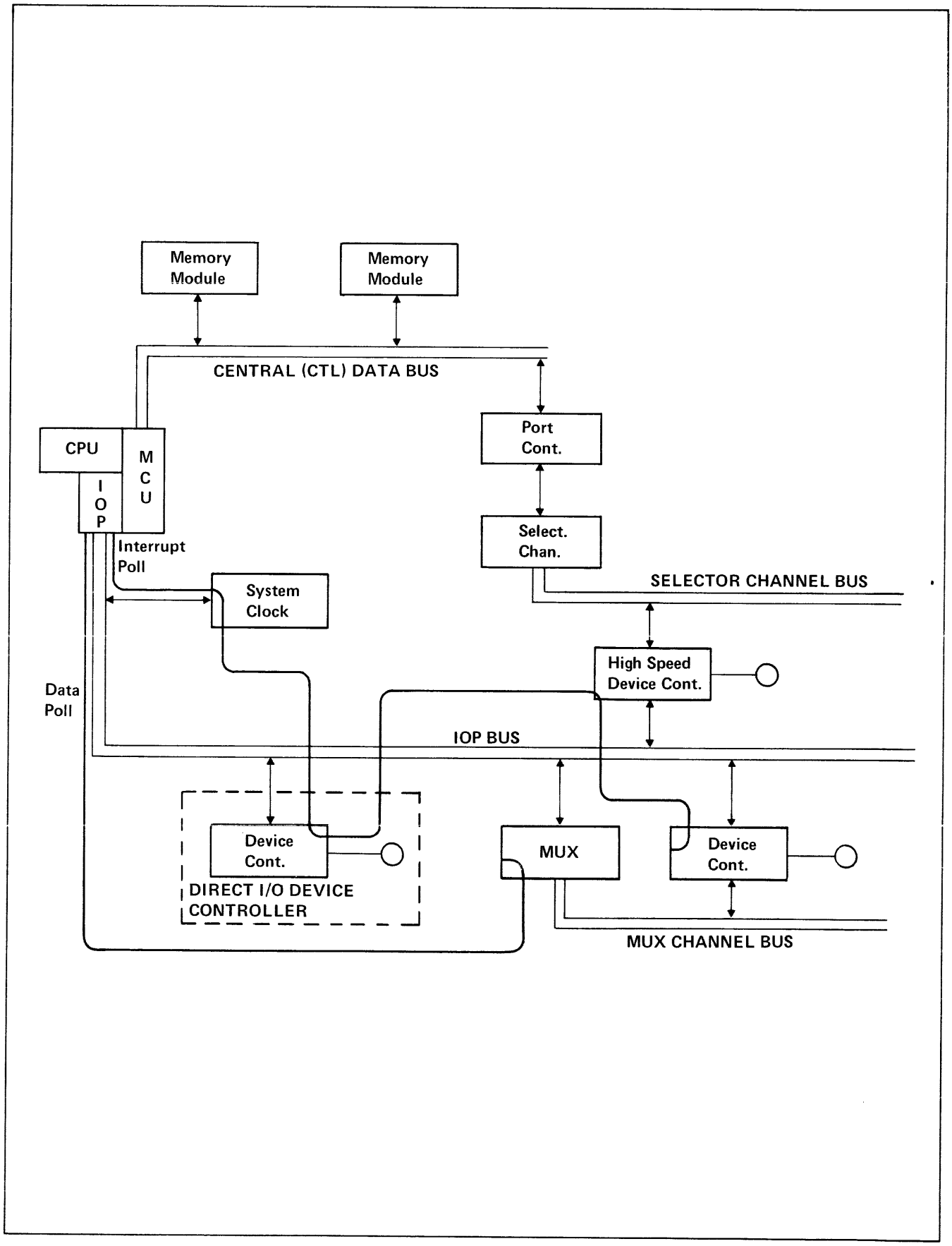


Figure 7-6. Interrupt Poll and Data Poll

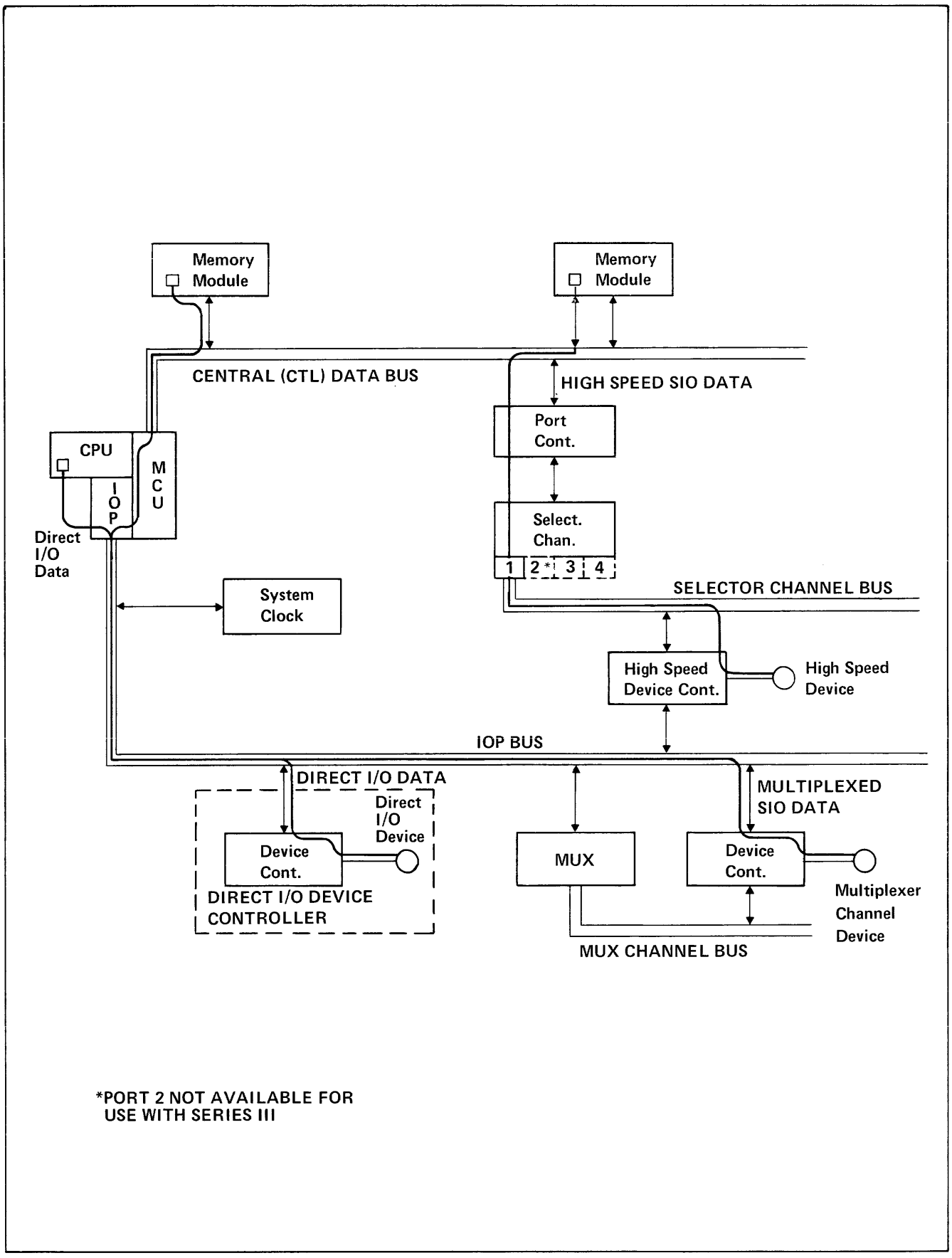


Figure 7-7. Input/Output Data Routes

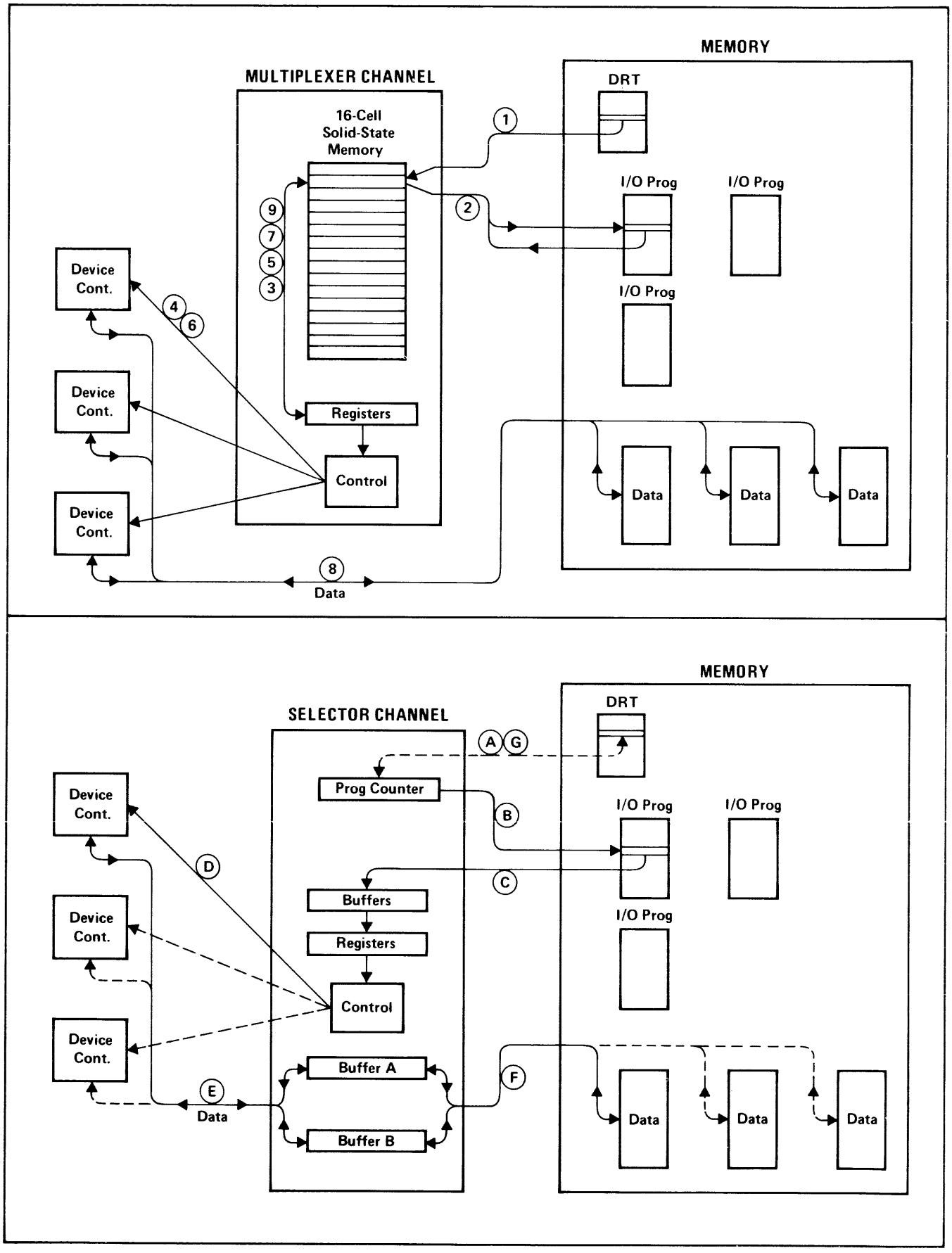


Figure 7-8. Basic Comparison of Multiplexer and Selector Channels

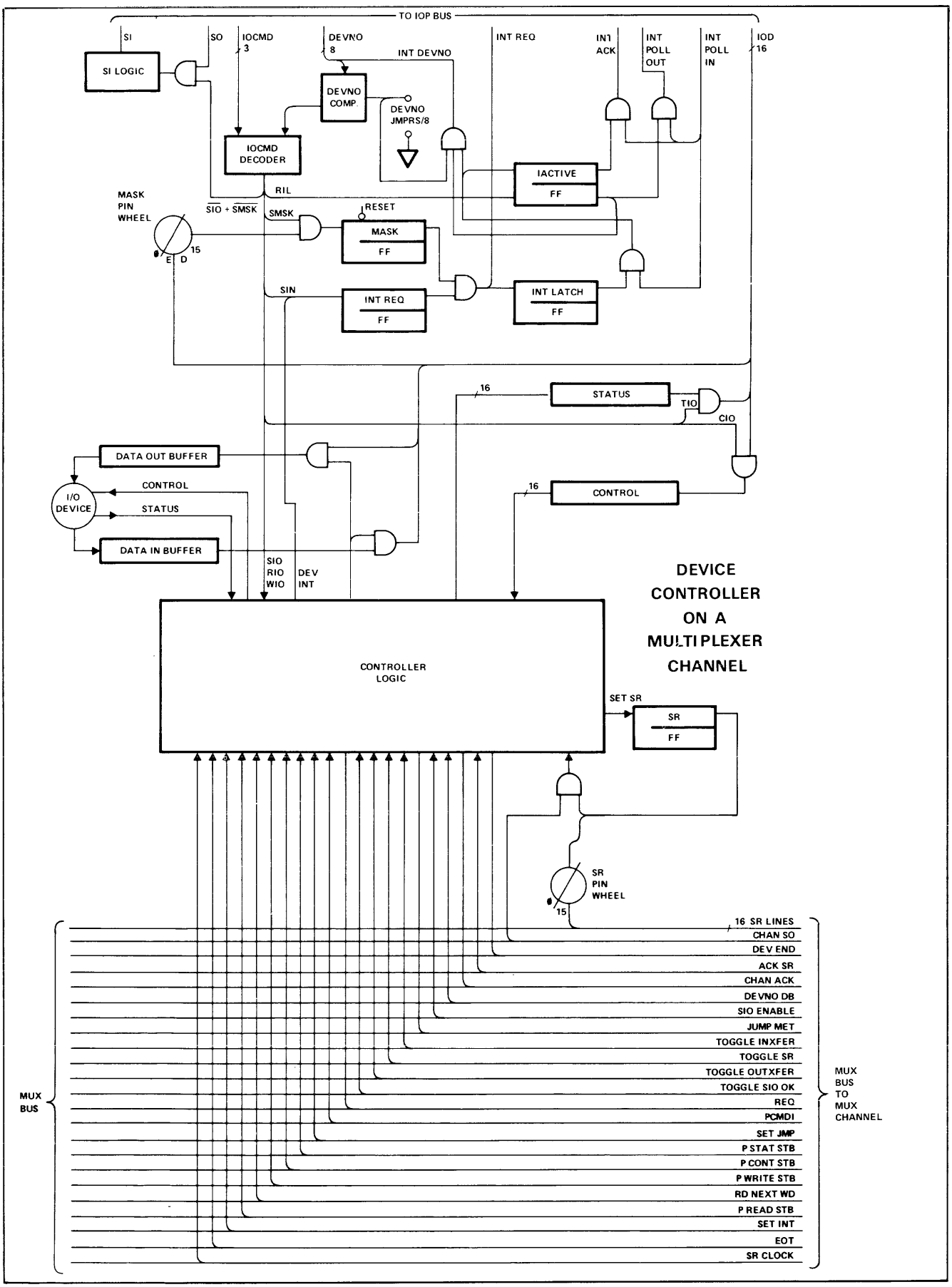


Figure 7-9. Multiplexer Channel and Device Controller Simplified Logic Diagram

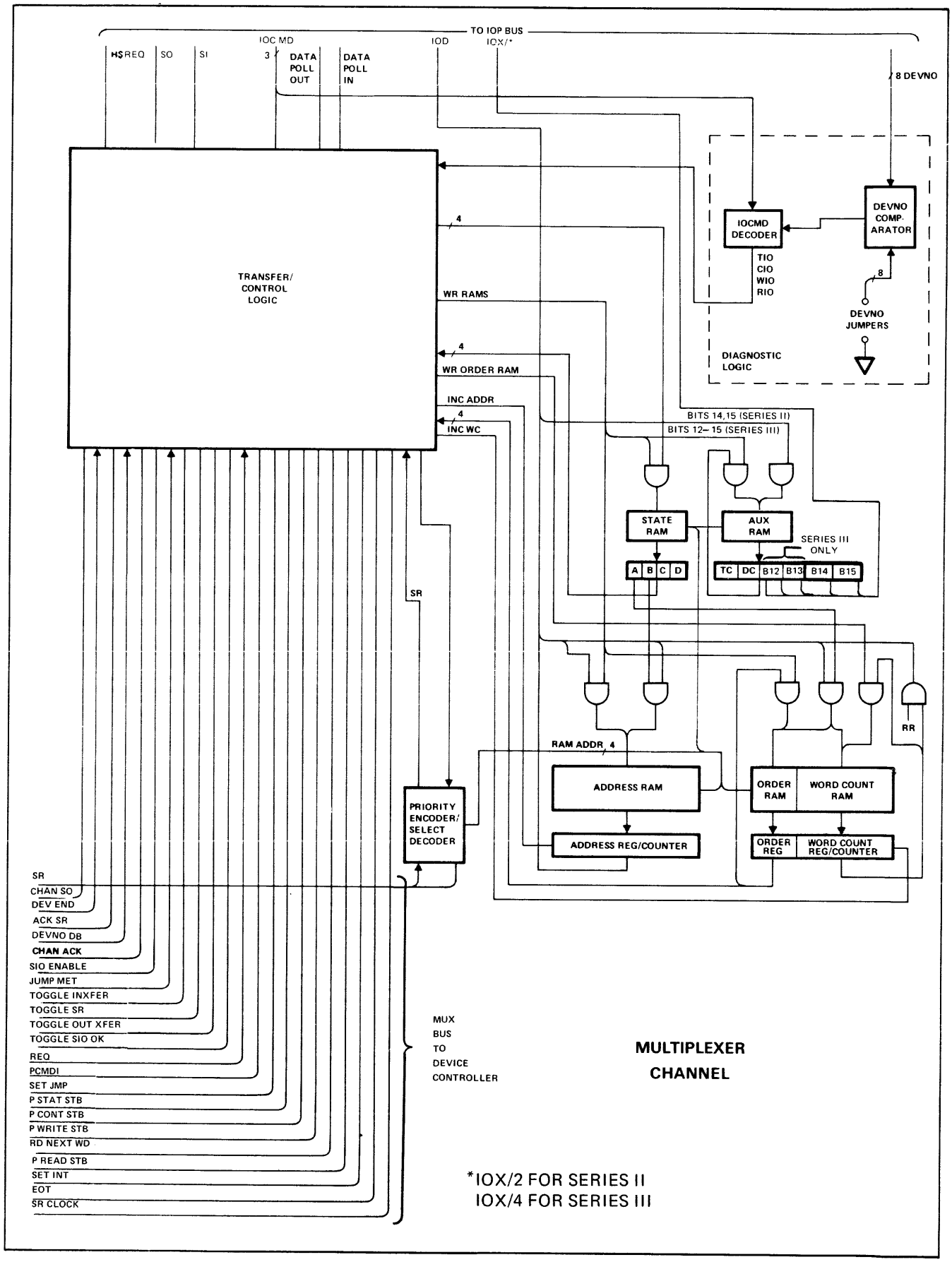


Figure 7-10. Multiplexer Channel Simplified Logic Diagram

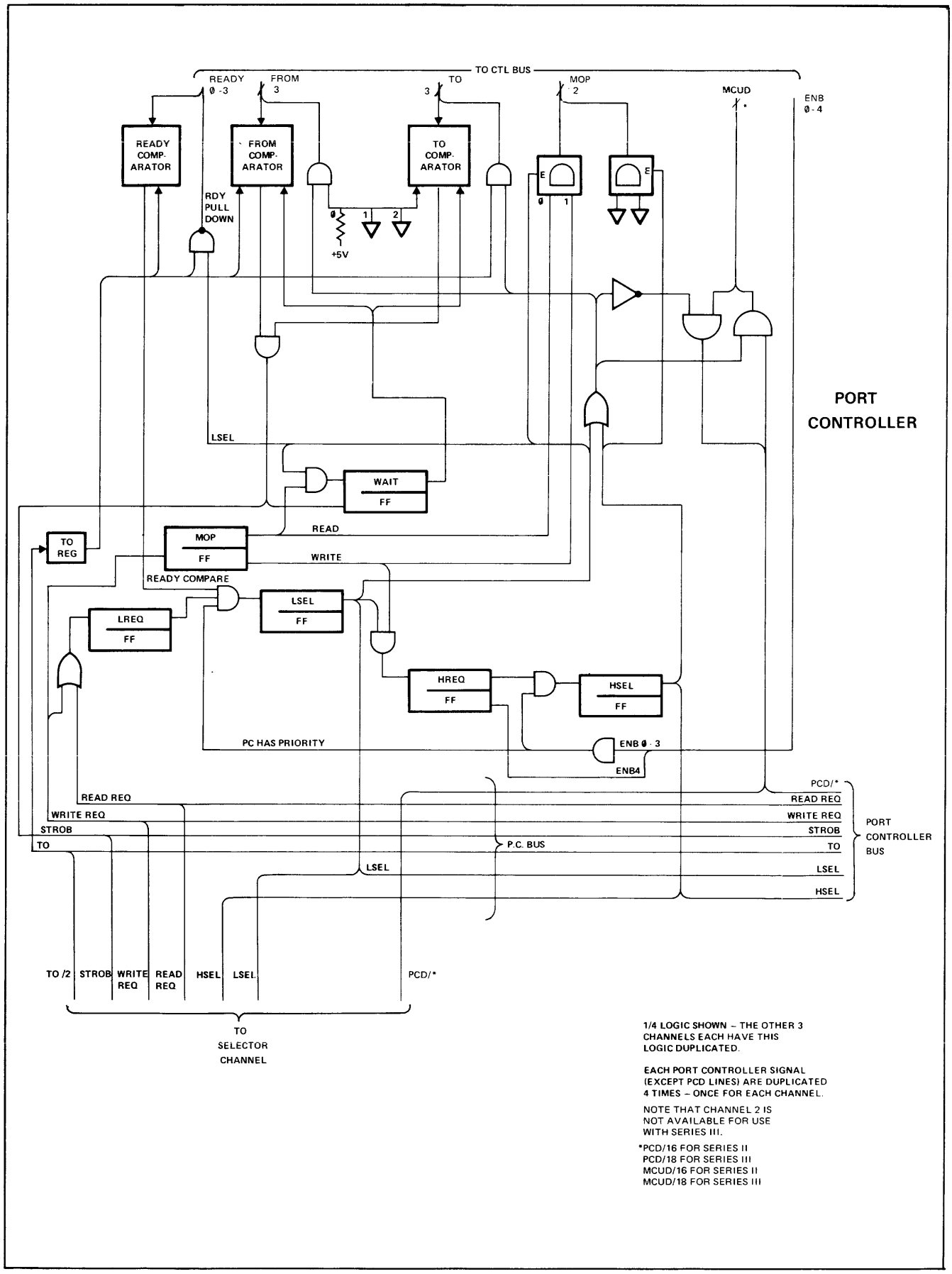


Figure 7-11. Port Controller Simplified Logic Diagram

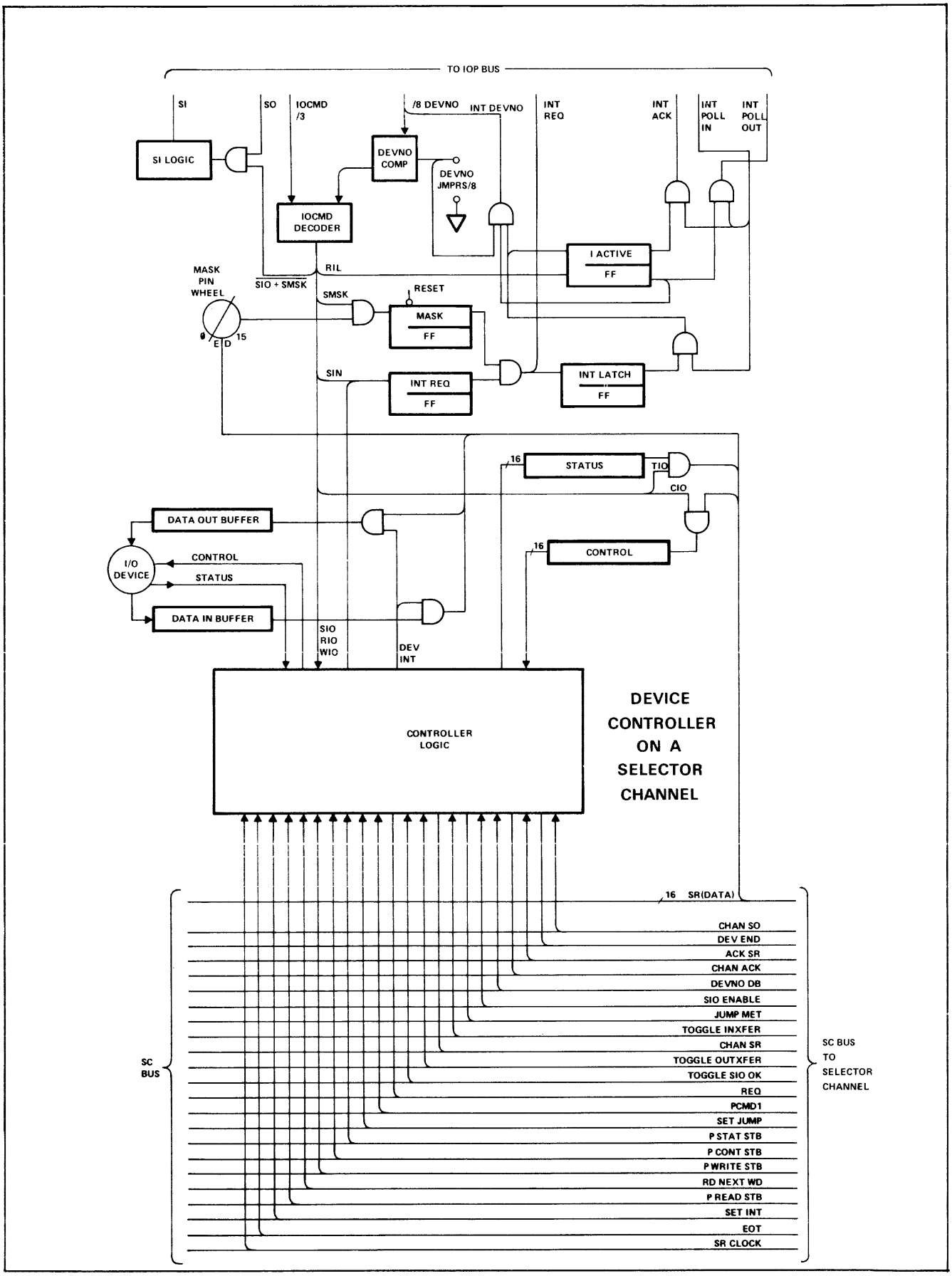


Figure 7-12. Device Controller on a Selector Channel Simplified Logic Diagram

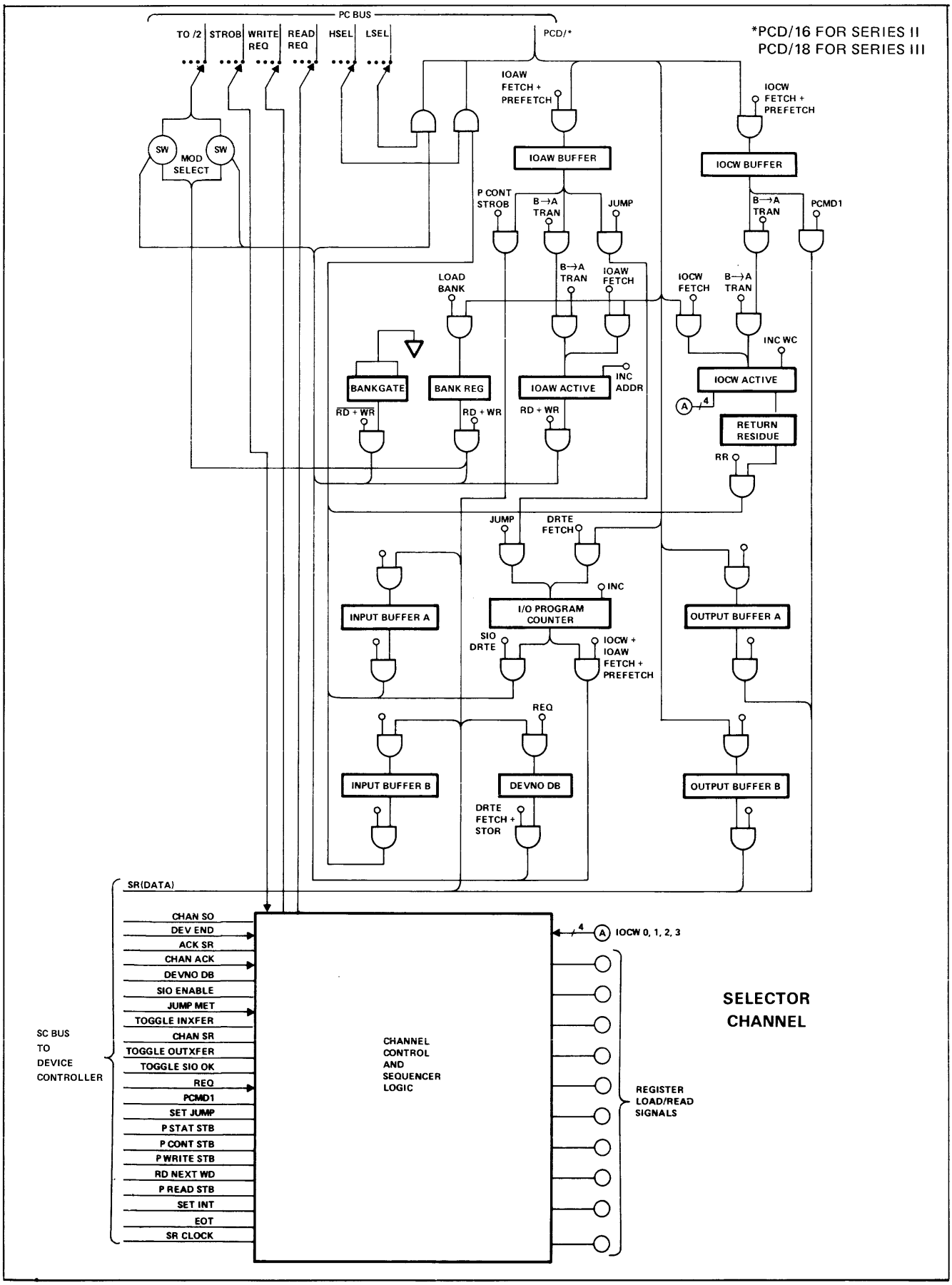
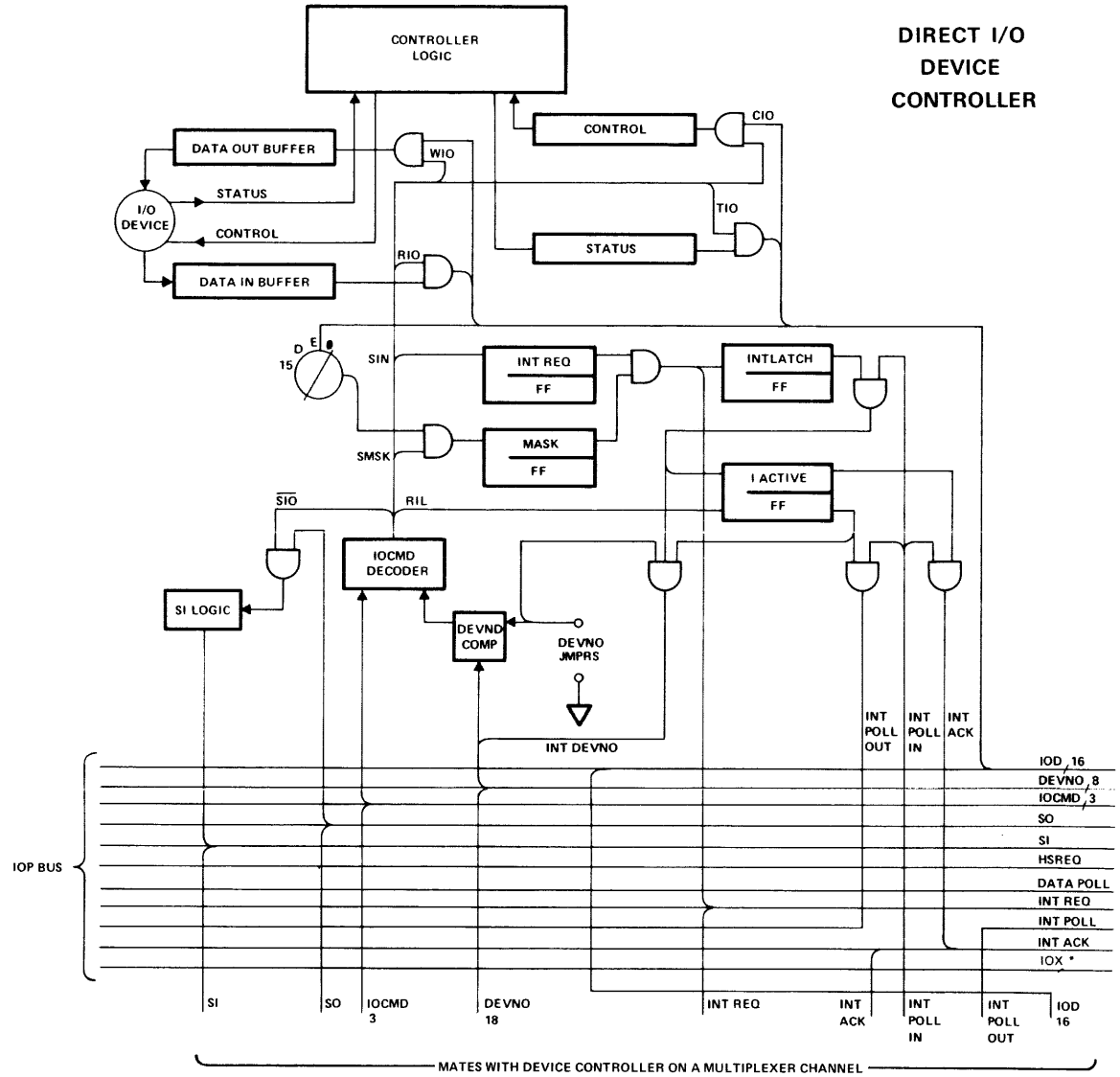


Figure 7-13. Selector Channel Simplified Logic Diagram



*IOX/2 FOR SERIES II
IOX/4 FOR SERIES III

Figure 7-14. Direct I/O Device Controller Simplified Logic Diagram

INDEX

A

Absence trap, 4-3
Absent code segment interrupt, 6-17
Absent data segment interrupt, 6-17
ABS MCU option field code, 3-57
Absolute addresses, 3-18
Access to DB area, 3-28
ACOR register, 3-66
Actual parameters, 4-24
ADD function field code, 3-44
ADDO function field code, 3-44
Address computer output register, 3-66
Addressing conventions, 3-18
Address parity error interrupt, 6-14
Address transfer, 7-17
ALU, 3-65
ALU function generator, 3-65
AND function field code, 3-44
Arithmetic logic unit, 3-65
Assumed binary point, 3-6
Asynchronous terminal controller, 2-10
ATTACHIO intrinsic, 5-8

B

Basic arithmetic operations, 4-20
Basic executable entity, 4-1
Binary point, 3-6
BIT6 skip field code, 3-50
BIT8 skip field code, 3-50
Bit test instructions, 3-11
Block, 5-1
Blocked I/O, 5-13
Block-level description, 3-29
BMUX, 3-31
BNDT function field code, 3-44
Bounds checking, 3-25
Bounds violation, 3-25
Bounds violation interrupt, 6-14
Branch instructions, 3-9
BSP0 store field code, 3-40
BSP1 store field code, 3-40
BUS store field code, 3-40
Bus system, 3-29
Byte addressing, 3-24
Byte format, 3-6
Byte index, 3-25

C

CAD function field code, 3-44
CADO function field code, 3-44
CalComp plotter, 2-9
CAND function field code, 3-44
Carry bit, 3-12
Carry flip-flop, 3-65
Carry indicator,3-12
CCA, 3-14
CCA special field code, 3-54
CCB, 3-14
CCB special field code, 3-54
CCC, 3-14
CCD, 3-14
CCE special field code, 3-54
CCG special field code, 3-54
CCL special field code, 3-54
CCPX special field code, 3-54
CCRY special field code, 3-55
CC S-bus field code, 3-37
CCZ special field code, 3-55
Central data bus, 3-31
Central data bus transmissions, 7-5
Central processor unit (CPU)
  description, 3-32
  features, 2-2
  functional operation, 7-1
  registers, 3-15
CF1 special field code, 3-55
CF2 special field code, 3-55
CF3 special field code, 3-55
CIO instruction, 5-5
CIR, 3-33
CIR S-bus field code, 3-37
CLIB special field code, 3-55
Clock cycle, 3-3
CLO special field code, 3-55
CLSR special field code, 3-55
CMD MCU option field code, 3-57
CMUX, 3-33
CMUX control, 3-33
CNTR register, 3-64
Code, 2-2
Code indexing, 3-21
Code indirect indexing, 3-20
Code segment, 4-4
Code segmentation, 4-4
Code segment evolution, 4-5
Code segment linkage, 4-10
Code segment registers, 3-15
Code segment table, 4-7
Code segment table extension, 4-7
Code sharing, 4-2
Cold load interrupt, 6-17
Command a module, 7-8
Conditional jump, 3-32
Condition code logic, 3-65
Condition codes, 3-14
Control I/O instruction, 5-5
Control I/O order, 5-17
Counter register, 3-64
CPU adress transmit, 7-5
CPU registers, 3-15
CPX1 register, 3-66
CPX1 S-bus field code, 3-37
CPX2 register, 3-66

CPX2 S-bus field code, 3-37
CRL MCU option field code, 3-57
CRRY skip field code, 3-50
CRS function field code, 3-44
CST, 4-7
CST violation interrupt, 6-15
CSTX, 4-7
CTF special field code, 3-55
CTRH S-bus field code, 3-37
CTRH store field code, 3-40
CTRL S-bus field code, 3-37
CTRL store field code, 3-40
CTRM skip field code, 3-50
CTSD function field code, 3-44
CTSS function field code, 3-45
Current conditions, 3-12
Current data segment, 3-15
Current instruction register, 3-33
Current stack marker, 3-15

D

Data, 2-2
Data base register, 3-63
Data computer output register, 3-66
Data formats, 3-4
Data indexing, 3-21
Data indirect indexing, 3-21
Data input registers, 7-13
Data limit of data segment, 3-15
Data limit register, 3-62
DATA MCU option field code, 3-57
Data output registers, 7-13
Data parity error interrupt, 6-14
Data poll, 7-10
Data privacy, 4-2
Data recorder, 2-9
Data references, 3-26
Data segmentation, 4-5
Data segment registers, 3-15
Data segment table, 4-7
Data service priority, 2-8
Data stacks, 2-2
Data transfers, 7-17
DB-Bank register, 3-15
DB MCU option field code, 3-57
DB register, 3-15, 3-62
DB S-bus field code, 3-37
DB store field code, 3-40
DCAD function field code, 3-45
DCOR register, 3-66
DCSR special field code, 3-55
Decimal corrector, 3-65
Decimal divide by zero interrupt, 6-17
Decimal overflow interrupt, 6-16
Deleting a word from the stack, 2-3
Description of systems, 1-1
Device controller, 5-16
Device information table, 5-4
Device numbers, 2-8, 5-5
Device reference table, 2-8
Direct byte addressing, 3-25
Direct, indexed byte addressing, 3-25
Direct I/O, 2-8
Direct I/O operation, 5-9
Direct read, 5-9
Direct write, 5-11
Disc memory, 4-3
Dispatcher, 6-3
Dispatcher interrupt, 6-11
Dispatcher marker, 6-3
DISP instruction, 6-22
Displacement, 3-20
DIT, 5-4
DL register, 3-15, 3-62
DL S-bus field code, 3-37
DL store field code, 3-40
DLT, 5-4
Double-word, fixed-point format, 3-6
Double-word indexing, 3-25
DPOP MCU option field code, 3-57
Driver linkage table, 5-4
DRT, 2-8
DRT fetch, 7-14
DST, 4-7
DST violation interrupt, 6-15
DVSB function field code, 3-45

E

Effective address, 3-20
END, I, 6-6
End I/O order, 5-17
End of transfer by word count, 7-18
End with interrupt command, 6-6
EVEN skip field code, 3-50
Examples of byte addressing, 3-24
Examples of indexing, 3-23
Examples of indirect addressing, 3-22
Execute sequences, 7-22
EXIT instruction, 4-12
Extended precision floating-point divide by zero interrupt, 6-16
Extended precision floating-point overflow interrupt, 6-16
Extended precision floating-point underflow interrupt, 6-16
External interrupts, 6-4

F

F1 skip field code, 3-50
F2 skip field code, 3-50
F3 skip field code, 3-50
Fetch an operand, 7-6
Fetch next instruction, 7-5
Fetch sequence, 7-21
FHB special field code, 3-55
Field instructions, 3-11
Files, 5-1
File system operation, 5-2
First level interrupt, 6-10
First Q location, 4-16
Floating-point data representation, 3-8

Floating-point format, 3-6
Floating-point overflow interrupt, 6-16
Floating-point underflow interrupt, 6-16
Formal parameters, 4-24
Function field, 3-3
Function field decoder, 3-36

G

General format instructions, 3-9
Global data area, 4-18

H

HBF special field code, 3-55
HP 2640, 2-9
HP 3000 Series II and III hardware organization, 2-1

I

ICS, 6-2
ICS internal interrupts, 6-6
Illegal memory address interrupt, 6-14
ILT, 5-4
Immediate instructions, 3-11
Implicit addressing, 4-21
INC function field code, 3-46
INCN special field code, 3-55
INCO function field code, 3-46
INCT special field code, 3-55
Indexing, 3-21
Index register, 3-17, 3-62
Indirect addressing, 3-20
Indirect byte addressing, 3-25
Indirect, indexed byte addressing, 3-25
INDR skip field code, 3-50
Initiator sequence, 7-20
Input/output, 2-7
Input/output system, 5-1
Input transfer, 7-18
INSR special field code, 3-55
Instructions, 2-6
Instruction formats, 3-8
Instruction groups, 3-10
INT DEVNO, 7-12
Integer divide by zero interrupt, 6-16
Integer overflow interrupt, 6-15
Internal interrupts, 6-6
Interrupt control logic, 7-12
Interrupt control stack, 6-2
Interrupt handler, 6-12
Interrupt I/O order, 5-17
Interrupt linkage table, 5-4
Interrupt poll, 6-5
Interrupt priority, 2-8
Interrupt program pointer, 6-8
Interrupt receiver code, 6-4
Interrupts
  absent code segment, 6-17
  absent data segment, 6-17
  address parity error, 6-14
  bounds violation, 6-14
  cold load, 6-17
  CST violation, 6-15
  data parity error, 6-14
  decimal divide by zero, 6-17
  decimal overflow, 6-16
  DST violation, 6-15
  extended precision floating-point divide by zero, 6-16
  extended precision floating-point overflow, 6-16
  extended precision floating-point underflow, 6-16
  floating-point overflow, 6-16
  floating-point underflow, 6-16
  illegal memory address, 6-14
  integer divide by zero, 6-16
  integer overflow, 6-15
  invalid ASCII digit, 6-16
  invalid decimal digit, 6-17
  invalid word count, 6-17
  module, 6-15
  non-responding module, 6-14
  power fail, 6-15
  power on, 6-17
  privileged mode violation, 6-15
  result word count overflow, 6-17
  stack overflow, 6-15
  stack underflow, 6-15
  STT entry uncallable, 6-17
  STT violation, 6-15
  system parity error, 6-14
  trace, 6-17
  unimplemented instruction, 6-15
Interrupt system, 6-1
Interrupt types, 6-1
Invalid ASCII digit interrupt, 6-16
Invalid decimal digit interrupt, 6-17
Invalid word count interrupt, 6-17
I/O address word, 5-16
I/O and interrupt instructions, 3-11
IOA S-bus field code, 3-37
IOA store field code, 3-40
IOAW, 5-16
IOAW fetch, 7-16
IOAW store, 7-16
I/O command word, 5-16
IOCW, 5-16
IOCW fetch,.7-15
IOD data in register, 7-13
IOD data out register, 7-13
I/O data routes, 7-10
I/O driver, 5-5
IOD S-bus field code, 3-37
IOD store field code, 3-40
I/O instructions, 5-5
I/O orders, 5-16
IOP, 5-14
IOP bus, 3-31
IOP data in register, 7-13
IOP data out register, 7-13
I/O priorities, 7-9
I/O processor, 5-14
I/O program area, 5-4
I/O programming, 5-16
I/O program operation, 5-18

I/O program word, 5-16
I/O program word transfers, 7-15
IOQ, 5-4
I/O queue, 5-4
IOR function field code, 3-46
I/O system, 5-1
IXIT instruction, 6-24

J

JLUI skip field code, 3-50
JMP function field code, 3-46
JSB function field code, 3-46
Jump I/O order, 5-16

L

LBF special field code, 3-55
Line printers, 2-9
LLZ shift field code, 3-53
LRZ shift field code, 3-53
Local variables, 4-19
Logical format, 3-6
Logical record, 5-1
Long floating-point format, 3-7
Look up table, 3-33
Look up table ROM, 3-33
Loop control instructions, 3-11
LUT ROM, 3-33

M

Machine instructions, 2-6
Magnetic tape unit, 2-9
Main memory, 2-7
Mapper, 3-33
Mapper control, 3-33
Main program call, 4-27
MCU, 4-6, 7-1
MCU option field decoder, 3-61
Memory addressing, 3-18
Memory address instructions, 3-11
Memory addressing modes, 3-19
Memory address register, 7-16
Memory banks, 4-6
Memory management, 4-3
Memory segmentation, 4-1
Microcode, 2-6
Microcode instructions, 3-3
Microcode jumps, 3-35
MiniDataStation, 2-9
Mode bit, 6-5
MOD S-bus field code, 3-37
Module control unit, 4-6, 7-1
Module interrupt, 6-15
Move instructions, 3-11
MPAD function field code, 3-46
MPY instruction, 4-21
MREG R-bus field code, 3-59
MREG store field code, 3-40
Multiplexer channel, 5-15
Multiplexer channel bus, 3-31

N

NCRY skip field code, 3-50
NEG skip field code, 3-50
Next instruction register, 3-33
NEXT skip field code, 3-51
NF1 skip field code, 3-51
NF2 skip field code, 3-51
NIR, 3-33
NIR MCU option field code, 3-57
NOFL skip field code, 3-51
Non-ICS internal interrupts, 6-6
Non-responding module interrupt, 6-14
NPRV skip field code, 3-52
NSME skip field code, 3-52
NZRO skip field code, 3-52

O

ODD skip field code, 3-52
Operand register, 3-63
OPINP flip-flop, 3-63
OPND MCU option field code, 3-57
OPND register, 3-63
OPND S-bus field code, 3-37
Output transfer, 7-17
Overflow bit, 3-12
Overflow flip-flop, 3-65
Overflow indicator, 3-12
Overflow, stack, 3-26
Overview of system operation, 3-31

P

PADD R-bus field code, 3-59
PADD S-bus field code, 3-38
Parameters, 4-24
PB-Bank register, 3-15
PB MCU option field code, 3-57
PB register, 3-15, 3-62
PB S-bus field code, 3-38
PB store field code, 3-40
PCAL instruction, 4-12
PCB, 4-1
PCLK S-bus field code, 3-38
PCLK store field code, 3-41
PCLOCK register, 3-63
Peripherals, 2-9
Pipeline, 3-2
PL R-bus field code, 3-59
PL register, 3-15, 3-62
PL store field code, 3-41
PNLR function field code, 3-47
PNLS function field code, 3-47
POPA special field code, 3-56
Popping the stack, 3-4
POP special field code, 3-56
Port controller, 7-19
POS skip field code, 3-52
Post indexing, 3-21
Power bus, 3-31
Power fail interrupt, 6-15

Power on interrupt, 6-17
Pre-adder, 3-65
P register, 3-15, 3-63
Previous Q, 4-19
Privileged mode, 3-18
Privileged mode bit, 3-12
Privileged mode violation interrupt, 6-15
Procedure call, 4-12
Procedures, 4-3
Process clock register, 3-63
Process control block, 4-1
Processes, 4-1
Processor registers, 3-61
Program, 4-1
Program base of code segment, 3-15
Program base register, 3-62
Program control, 3-25
Program control instructions, 3-11
Program counter register, 3-63
Program file, 4-4
Program limit of code segment, 3-15
Program limit register, 3-62
Programmed I/O, 2-8
Program references, 3-26
Program transfer, 3-25
P S-bus field code, 3-38
Pseudo enabling/disabling the dispatcher, 6-24
P store field code, 3-40
Punched-tape unit, 2-9
Pusing a word on the stack, 2-3
PUSH store field code, 3-41

Q

QASL function field code, 3-47
QASR function field code, 3-48
QDWN S-bus field code, 3-38
Q mode, 3-21
Q pointer, 4-14
Q register, 3-15, 3-63
Q S-bus field code, 3-38
Q store field code, 3-41
QUP store field code, 3-41

R

RA R-bus field code, 3-59
RAR register, 3-34
RAR store field code, 3-42
RA S-bus field code, 3-38
RA store field code, 3-42
RB R-bus field code, 3-59
RB S-bus field code, 3-38
RB store field code, 3-42
RBR S-bus field code, 3-38
R-bus field decoder, 3-61
RBUS R-bus field code, 3-59
R-bus register, 3-65
RC R-bus field code, 3-59
RC S-bus field code, 3-39
RC store field code, 3-42
RD R-bus field code, 3-60
RD S-bus field code, 3-39
RD store field code, 3-42
Read I/O instruction, 5-5
Read I/O order, 5-17
Read-only lmemory, 3-35
Recursion, 4-26
Re-entrancy, 4-4
Register control instructions, 3-11
Registers
  ACOR, 3-66
  address computer output, 3-66
  CIR, 3-33
  CNTR, 3-64
  code segment, 3-15
  counter, 3-64
  CPU, 3-15
  CPX1, 3-66
  CPX2, 3-66
  current instruction, 3-33
  data base, 3-63
  data computer output, 3-66
  data input, 7-13
  data limit, 3-62
  data output, 7-13
  data segment, 3-15
  DB, 3-15, 3-63
  DB-Bank, 3-15
  DCOR, 3-66
  DL, 3-15, 3-62
  index, 3-17, 3-62
  IOD data in, 7-13
  IOD data out, 7-13
  IOP data in, 7-13
  IOP data out, 7-13
  memory address, 7-16
  next instruction, 3-33
  NIR, 3-33
  operand, 3-63
  OPND, 3-63
  PB, 3-15, 3-62
  PB-Bank, 3-15
  PCLOCK, 3-63
  PL, 3-15, 3-62
  P, 3-15, 3-63
  process clock, 3-63
  processor, 3-61
  program base, 3-62
  program counter, 3-63
  program limit, 3-62
  Q, 3-15, 3-63
  RAR, 3-34
  R-bus, 3-65
  ROM address, 3-34
  save, 3-35
  S-bus, 3-65
  scratch pad 0, 3-62
  scratch pad 1, 3-62
  scratch pad 2, 3-63
  scratch pad 3, 3-63
  SM, 3-17, 3-63
  SP0, 3-62
  SP1, 3-62

SP2, 3-63
SP3, 3-63
SR counter, 3-62
SR, 3-17
stack bank, 3-17
stack limit, 3-62
stack memory, 3-63
stack, 3-62
status, 3-17, 3-64
top-of-stack, 3-62
TR0R, 3-62
TR1R, 3-62
TR2R, 3-62
TR3R, 3-62
TR0S, 3-62
TR1S, 3-62
TR2S, 3-62
TR3S, 3-62
X, 3-62
Z, 3-17, 3-62
Relative addressing, 3-18
Renamer, 3-61
REPC function field code, 3-48
REPN function field code, 3-48
Result word count overflow interrupt, 6-17
Return residue I/O order, 5-16
RIO instruction, 5-5
RLZ shift field code, 3-53
RND MCU option field code, 3-57
RNP MCU option field code, 3-57
RNS MCU option field code, 3-57
ROA MCU option field code, 3-57
ROD MCU option field code, 3-58
ROM, 3-35
ROM address register, 3-34
ROM function field code, 3-48
ROMI function field code, 3-48
ROMN function field code, 3-49
ROMX function field code, 3-49
ROND MCU option field code, 3-58
RONP MCU option field code, 3-58
RONS MCU option field code, 3-58
ROP MCU option field code, 3-58
ROS MCU option field code, 3-58
ROSA MCU option field code, 3-58
ROSD MCU option field code, 3-58
RRZ shift field code, 3-53
RSB skip field code, 3-52

S

Save register, 3-35
SBR store field code, 3-42
S-bus field decoder, 3-36
S-bus register, 3-65
SBUS S-bus field code, 3-39
Scratch pad 0 register, 3-62
Scratch pad 1 register, 3-62
Scratch pad 2 register, 3-63
Scratch pad 3 register, 3-63
SCRY special field code, 3-56
SDFG special field code, 3-56
Secondary global data area, 4-18
Second level interrupt, 6-11
Segment, 4-3
Segment transfer table, 4-12
Segmented libraries, 4-4
Selector channel, 5-16, 7-19
Selector channel bus, 3-31
Sense I/O order, 5-17
Separate code and data, 2-2
Service request, 7-14
Set bank I/O order, 5-17
SET INT, 6-4
Set interrupt command, 6-4
Set interrupt I/O instruction, 5-5
Set interrupt instruction, 6-4
Set mask I/O instruction, 5-5
SF1 special field code, 3-56
SF2 special field code, 3-56
SF3 special field code, 3-56
Shifter, 3-65
Shift field, 3-3
Shift field decoder, 3-36
Shift instruction group, 3-9
SIFG special field code, 3-56
SIO, 2-8
SIN, 6-4
Single-word, fixed-point format, 3-6
Skip field, 3-4
Skip field decoder, 3-36
SL1 shift field code, 3-53
S MCU option field code, 3-58
S-minus relative addressing, 2-4
S mode, 3-21
SM register, 3-17, 3-63
SM S-bus field code, 3-39
SMSK I/O instruction, 5-5
SM store field code, 3-43
SOV special field code, 3-56
SP0 R-bus field code, 3-60
SP0 register, 3-62
SP0 store field code, 3-43
SP1 R-bus field code, 3-60
SP1 register, 3-62
SP1 S-bus field code, 3-39
SP1 store field code, 3-43
SP2 register, 3-63
SP2 S-bus field code, 3-39
SP2 store field code, 3-43
SP3 register, 3-63
SP3 S-bus field code, 3-39
SP3 store field code, 3-43
Special field, 3-4
Special field decoder, 3-36
Special instructions, 3-11
Split stack, 4-5
S pointer, 3-17
SR, 7-14
SR1 shift field code, 3-53
SR4 skip field code, 3-52
SR counter, 3-62
SRL2 skip field code, 3-52
SRL 3 skip field code, 3-52

SRN4 skip field code, 3-52
SRNZ skip field code, 3-52
SR R-bus field code, 3-60
SR register, 3-17
SRZ skip field code, 3-52
SSL, 4-5
SST, 4-12
Stack bank register, 3-17
Stack limit register, 3-62
Stack limits, 3-25
Stack marker, 3-15, 4-18
Stack marker chain, 4-17
Stack memory register, 3-63
Stack operation, 4-14
Stack Op instructions, 3-9
Stack overflow, 3-26
Stack overflow interrupt, 6-15
Stack register, 3-62
Stack-structured data, 2-2
Stack, top of, 2-3
Stack underflow, 3-26
Stck underflow interrupt, 6-15
Start I/O instruction, 5-5
STA S-bus field code, 3-39
STA store field code, 3-43
Status register, 3-17, 3-64
Status word format, 3-12
Store an operand, 7-7
Store field, 3-3
Store field decoder, 3-36
STT entry uncallable interrupt, 6-17
STT violation interrupt, 6-15
SUB function field code, 3-49
SUB0 function field code, 3-49
Subprogram's view of the stack, 2-4
SWAB shift field code, 3-53
Swapping, 4-1
SWCH S-bus field code, 3-39
System configuration, 1-1
System features, 2-1
System-level description, 3-1
System library, 4-3
System parity error interrupt, 6-14
System segmented library, 4-5

T

Table structures, 4-7
T-bus, 3-3
Temporary storage, 4-19
Terminal input processor, 5-11
Terminal monitor, 5-11
Terminals, 2-9
TERMINATE process, 6-22
Test I/O instruction, 5-5
TEST skip field code, 3-52
Then current conditions, 3-12
TIO I/O instruction, 5-5
TIP, 5-11
TNAME code, 3-61
Top of stack, 2-3
Top-of-stack registers, 3-62
TR0R register, 3-62
TR1R register, 3-62
TR2R register, 3-62
TR3R register, 3-62
TR0S register, 3-62
TR1S register, 3-62
TR2S register, 3-62
TR3S register, 3-62
Trace interrupt, 6-17
Transfer modes, 7-10

U

UBNT function field code, 3-49
UBUS R-bus field code, 3-60
UBUS S-bus field code, 3-39
Unblocked I/O, 5-13
Unconditional jump, 3-32
UNC skip field code, 3-52
Underflow, stack, 3-26
Unimplemented instruction interrupt, 6-15
User subprogram library, 4-4
USL, 4-4

V

Variable-length segmentation, 2-4
V-bus control, 3-34
V-bus MUX, 3-34
Virtual memory, 4-1

W

W-bit, 3-33
WIO I/O instruction, 5-5
Word Addressing, 3-28
WRA MCU option field code, 3-58
WRD MCU option field code, 3-58
Write I/O instruction, 5-5
Write I/O order, 5-17
WRS MCU option field code, 3-58

X

X-bit, 3-21
XC R-bus field code, 3-60
XOR function field code, 3-49
X R-bus field code, 3-60
X register, 3-62
X store field code, 3-43

Z

ZERO skip field code, 3-52
Z R-bus field code, 3-60
Z register, 3-17, 3-62
Z store field code, 3-43

Part No. 30000-90020
Printed in U.S.A. 7/78

HEWLETT hp PACKARD

5303 Stevens Creek Blvd., Santa Clara, California 95050