



FLOATING POINT  
SYSTEMS, INC.

**Programmable  
I/O Processor  
(PIOP) Manual**

860-7350-002

by FPS Technical Publications Staff

**Programmable  
I/O Processor  
(PIOP) Manual**

**860-7350-002**

3rd Edition, June 1978  
Publication No. FPS 7350-02

### NOTICE

The material in this manual is for information purposes only and is subject to change without notice.

Floating Point Systems, Inc. assumes no responsibility for any errors which may appear in this publication.

Copyright © 1978 by Floating Point Systems, Inc.  
Beaverton, Oregon 97005

All rights reserved. No part of this publication may be reproduced in any form or by any means without permission in writing from the publisher.

Printed in USA

## CONTENTS

|   | Page |
|---|------|
| CHAPTER 1 INTRODUCTION                            |      |
| 1.1 PURPOSE                                       | 1-1  |
| 1.2 SCOPE   | 1-2  |
| 1.3 GENERAL DESCRIPTION                           | 1-3  |
| 1.4 FUNCTIONAL DESCRIPTION                        | 1-6  |
| 1.4.1 Buffer Memory (FIFO)                        | 1-9  |
| 1.4.2 ALU/Scratchpad Memory                       | 1-11 |
| 1.4.3 CPU/Program Source Memory                   | 1-13 |
| 1.5 MAJOR COMPONENTS                              | 1-16 |
| 1.6 PIOP FEATURES                                 | 1-17 |
| <br>  |      |
| CHAPTER 2 INSTRUCTION SET                         |      |
| 2.1 INTRODUCTION                                  | 2-1  |
| 2.2 INSTRUCTION WORD                              | 2-2  |
| 2.3 INSTRUCTION SET                               | 2-4  |
| <br>  |      |
| CHAPTER 3 FUNCTIONAL ELEMENTS                     |      |
| 3.1 INTRODUCTION                                  | 3-1  |
| 3.2 TRANSCEIVER                                   | 3-2  |
| 3.3 REGISTERS                                     | 3-4  |
| 3.4 OTHER ELEMENTS                                | 3-9  |
| <br>  |      |
| CHAPTER 4 PROGRAMMING                             |      |
| 4.1 INTRODUCTION                                  | 4-1  |
| 4.1.1 Programming Hints                           | 4-2  |
| 4.1.2 Reference Material                          | 4-3  |
| 4.2 USING THE TRANSCEIVER                         | 4-4  |
| 4.2.1 Transceiver Operation                       | 4-4  |
| 4.2.2 Transceiver Formats                         | 4-7  |
| 4.2.3 Transceiver Instructions                    | 4-9  |
| 4.2.4 Transceiver Timing Considerations           | 4-13 |
| 4.2.5 Interaction of Transceiver Instructions     | 4-21 |
| 4.2.6 DMA Transfers                               | 4-22 |
| 4.3 USING THE ALU                                 | 4-25 |
| 4.3.1 ALU Source (Inputs)                         | 4-30 |
| 4.3.2 ALU Destination (Output)                    | 4-30 |
| 4.3.3 ALU Function                                | 4-31 |
| 4.3.4 Using ALU Instructions                      | 4-32 |
| 4.4 USING THE PROGRAM SOURCE MEMORY               | 4-34 |
| 4.4.1 Instructions That Use Program Source Memory | 4-37 |
| 4.4.2 Branch and Jump Instructions                | 4-40 |
| 4.5 INTERRUPT HANDLING                            | 4-44 |
| 4.6 COMMUNICATING WITH THE AP                     | 4-46 |

## CHAPTER 5 ASSEMBLER

|     |                     |      |
|-----|---------------------|------|
| 5.1 | INTRODUCTION        | 5-1  |
| 5.2 | THE BASICS          | 5-2  |
| 5.3 | WRITING PROGRAMS    | 5-11 |
| 5.4 | USING THE ASSEMBLER | 5-13 |
| 5.5 | SAMPLE LISTINGS     | 5-14 |

## CHAPTER 6 PROGRAMMABLE I/O CHANNEL (PIOC)

|     |                                  |      |
|-----|----------------------------------|------|
| 6.1 | INTRODUCTION                     | 6-1  |
| 6.2 | CHANNEL INSTRUCTIONS             | 6-2  |
| 6.3 | WRITING CHANNEL COMMAND PROGRAMS | 6-6  |
| 6.4 | ACCESSING DISK DATA USING DKPIOC | 6-10 |
| 6.5 | AP/PIOP PROCESS SYNCHRONIZATION  | 6-12 |
| 6.6 | PIOC ERROR CONDITIONS            | 6-16 |

## CHAPTER 7 FORTRAN OPERATIONS

|        |   |      |
|--------|---|------|
| 7.1    | INTRODUCTION                                | 7-1  |
| 7.2    | FORTRAN CALLS                               | 7-3  |
| 7.2.1  | Load PIOP From AP MD (PPLOAD)               | 7-4  |
| 7.2.2  | Start PIOP (PPGO)                           | 7-5  |
| 7.2.3  | Reset PIOP (PPRS)                           | 7-6  |
| 7.2.4  | Get PIOP Status (PPSTAT)                    | 7-7  |
| 7.2.5  | Wait for PIOP (PPWAIT)                      | 7-8  |
| 7.2.6  | Read PIOP Flag From AP (PPFRD)              | 7-9  |
| 7.2.7  | Set PIOP Flag From AP (PPFSET)              | 7-10 |
| 7.2.8  | Clear PIOP Flag From AP (PPFCLR)            | 7-11 |
| 7.2.9  | Initialize PIOP Disk Parameters (INPPDK)    | 7-12 |
| 7.2.10 | Read Data From PIOP Disk to AP MD (RDPPDK)  | 7-15 |
| 7.2.11 | Write Data From AP MD to PIOP Disk (WRPPDK) | 7-17 |
| 7.2.12 | Write To and Read From PIOP Disk (WRDPPD)   | 7-19 |
| 7.2.13 | Start PIOP Channel (PCGO)                   | 7-21 |
| 7.2.14 | Get PIOP Channel Status (PCSTAT)            | 7-23 |
| 7.2.15 | PIOP Execute Loader (PEXEC)                 | 7-25 |
| 7.3    | SAMPLE PROGRAMS                             | 7-27 |
| 7.3.1  | Fortran Subroutine Example                  | 7-27 |
| 7.3.2  | Fortran Program Example                     | 7-30 |

CHAPTER 8 PIOP DEBUGGER - PPDEBUG

|       |  |      |
|-------|--|------|
| 8.1   | INTRODUCTION   | 8-1  |
| 8.2   | OPERATING PROCEDURE                                    | 8-2  |
| 8.3   | MONITORING REGISTERS AND MEMORY LOCATIONS              | 8-2  |
| 8.3.1 | "E", Open and Examine                                  | 8-3  |
| 8.3.2 | "+", "-", and "." Examine Next, Last and<br>Re-examine | 8-4  |
| 8.3.3 | "C", Change  | 8-5  |
| 8.4   | CHANGING INPUT/OUTPUT FORMATS                          | 8-7  |
| 8.4.1 | "N", Set Radix   | 8-8  |
| 8.4.2 | "F", Set/Reset Floating-Point I/O                      | 8-9  |
| 8.5   | MEMORY LOADING AND DUMPING                             | 8-12 |
| 8.5.1 | "Y", Yank from a File                                  | 8-12 |
| 8.5.2 | "W", Write to a File                                   | 8-13 |
| 8.5.3 | "Z", Zero the AP                                       | 8-15 |
| 8.5.4 | Preparing Data Files for Yanking                       | 8-15 |
| 8.6   | EXECUTING PROGRAMS                                     | 8-17 |
| 8.6.1 | "I", Initialize the PIOP                               | 8-17 |
| 8.6.2 | "R", Run a PIOP Program                                | 8-17 |
| 8.6.3 | "X", Exit to PPDEBUG                                   | 8-17 |

APPENDIX A INSTRUCTION SET

APPENDIX B PIOP INTERCONNECTIONS

APPENDIX C SPECIAL STORAGE ELEMENTS

|     |                     |     |
|-----|---------------------|-----|
| C.1 | INTRODUCTION        | C-1 |
| C.2 | FIFO MEMORY ELEMENT | C-2 |
| C.3 | STACK               | C-4 |

APPENDIX D SUMMARY OF PPDEBUG COMMANDS

|     |  |     |
|-----|--|-----|
| D.1 | INTRODUCTION                               | D-1 |
| D.2 | PROGRAM EXECUTION COMMANDS                 | D-2 |
| D.3 | REGISTER EXAMINATION/MODIFICATION COMMANDS | D-3 |
| D.4 | MEMORY LOAD/DUMP COMMANDS                  | D-4 |
| D.5 | ACCESSIBLE FUNCTIONAL UNITS                | D-5 |

## ILLUSTRATIONS

| Figure No. | Title   | Page |
|------------|---|------|
| 1-1        | PIOP Used as Basic I/O Interface                    | 1-5  |
| 1-2        | PIOP Used as Main Data Management Processor         | 1-5  |
| 1-3        | PIOP Used an an I/O Bus                             | 1-5  |
| 1-4        | PIOP - - Overall Block Diagram                      | 1-7  |
| 1-5        | Buffer Memory (Transceiver) - Simplified<br>Diagram | 1-10 |
| 1-6        | ALU/Scratchpad Memory - Simplified Diagram          | 1-12 |
| 1-7        | CPU/Program Source Memory - Simplified Diagram      | 1-14 |
| 2-1        | PIOP Instruction Word                               | 2-3  |
| 3-1        | Control Register (CR)                               | 3-6  |
| 3-2        | Device Command Register (DC)                        | 3-7  |
| 4-1        | Transceiver - Simplified Diagram                    | 4-5  |
| 4-2        | Transceiver Formats                                 | 4-8  |
| 4-3        | IN to Empty FIFO                                    | 4-13 |
| 4-4        | Multiple IN's to Empty FIFO                         | 4-14 |
| 4-5        | SETMAW Instruction                                  | 4-15 |
| 4-6        | SETMAW Loop   | 4-16 |
| 4-7        | SETMAW IN Loop                                      | 4-18 |
| 4-8        | SETMAR OUT Loop                                     | 4-20 |
| 4-9        | ALU Instruction Formats                             | 4-25 |
| 4-10       | ALU Logic - Block Diagram                           | 4-28 |
| 4-11       | Program Source Address Logic                        | 4-34 |
| 4-12       | Next Address Control Logic                          | 4-36 |
| 4-13       | TR PS, IOR Instruction Cycles                       | 4-37 |
| 4-14       | Program Cycles                                      | 4-38 |
| 4-15       | Interrupt Timing                                    | 4-45 |
| 6-1        | Channel Instruction Format                          | 6-2  |
| 6-2        | Channel Program Example 1                           | 6-8  |
| 6-3        | Channel Program Example 2                           | 6-13 |
| 7-1        | Timing for Block FFT                                | 7-32 |
| B-1        | PIOP Interconnection Diagram                        | B-2  |
| C-1        | Operation of a Typical FIFO Memory                  | C-3  |
| C-2        | Stack Operation                                     | C-5  |

## TABLES

| Table No. | Title   | Page |
|-----------|---|------|
| 1-1       | Applicable Publications                       | 1-2  |
| 2-1       | Shorthand Notation Conventions                | 2-4  |
| 2-2       | PIOP Instructions                             | 2-5  |
| 2-3       | Expanded PIOP Instructions                    | 2-12 |
| 2-4       | Symbol Definitions                            | 2-13 |
| 2-5       | Cross-Reference List                          | 2-14 |
| 3-1       | ALU Status Conditions                         | 3-7  |
| 3-2       | Device Status (Assignment for Disk Interface) | 3-8  |
| 3-3       | Flags   | 3-8  |
| 4-1       | Programming Subjects                          | 4-1  |
| 4-2       | I/O Word Formats                              | 4-7  |
| 4-3       | Transceiver Instructions                      | 4-9  |
| 4-4       | ALU Instructions                              | 4-26 |
| 4-5       | ALU Designations                              | 4-29 |
| 4-6       | Summary of ALU Instructions                   | 4-32 |
| 4-7       | Stack-Related Instructions                    | 4-40 |
| 4-8       | BIT # FIELD                                   | 4-41 |
| 4-9       | Branch Instructions                           | 4-42 |
| 4-10      | Extended Branch Instructions                  | 4-43 |
| 4-11      | AP Device Instructions                        | 4-46 |
| 6-1       | Addressing Modes                              | 6-2  |
| 6-2       | Channel Instruction Operation Codes           | 6-4  |
| 6-3       | Disk Data Format Types                        | 6-10 |
| 7-1       | Fortran Calls                                 | 7-1  |
| 7-2       | Error Information                             | 7-2  |
| A-1       | PIOP Instructions                             | A-2  |
| A-2       | PIOP Expanded Instructions                    | A-9  |
| A-3       | Symbol Definitions                            | A-10 |
| A-4       | Cross-Reference List                          | A-11 |

## CHAPTER 1

### INTRODUCTION

#### 1.1 PURPOSE

The purpose of this manual is to provide the information necessary to understand, program, and use the Programmable Input/Output Processor (PIOP). The PIOP is a general-purpose programmable controller that interfaces peripheral devices to the Floating Point Systems' AP-120 Array Processor. Throughout the remainder of this manual, the Array Processor is referred to as either the "AP-120" or "AP," and the peripheral device is referred to as the "external device."

## 1.2 SCOPE

This manual is a user's document and, therefore, stresses software information related to the PIOP. Hardware information, from a programming standpoint, is also included. A general description of the PIOP is covered in this chapter. Subsequent chapters describe programming, the programmable I/O channel (PIOC) which is a software interpreter residing in the PIOP, the assembler, Fortran operations, debugging programs, and diagnostic programs.

Although the PIOP communicates with the AP and handles transfers of data between the AP and the external device, the AP itself is not described in this manual. However, those AP functions that are necessary for a complete understanding of the PIOP are included. If more information is desired on the AP, the reader should refer to the manuals listed in Table 1-1 below. Any of these manuals can be ordered from Floating Point Systems, P.O. Box 23489, Portland, Oregon 97223.

Table 1-1 Applicable Publications

| TITLE                        | NUMBER  |
|------------------------------|---------|
| PROCESSOR HANDBOOK           | 7259-02 |
| MAINTENANCE MANUAL           | 7270    |
| DIAGNOSTIC SOFTWARE MANUAL   | 7284-03 |
| AP MATH LIBRARY              | 7288-03 |
| PAGE SELECT MANUAL           | 7365    |
| PROGRAMMERS REFERENCE MANUAL | 7319    |

0033

### 1.3 GENERAL DESCRIPTION

The programmable I/O processor (PIOP) is an independent parallel processor. This processor, which can be tightly coupled to an AP, can serve both as an auxiliary main data management processor to the AP and as an I/O processor for handling one or more external devices.

The PIOP can be used in a number of ways, depending on the particular user's requirements. Three of these uses are to serve as a main data management processor, as an I/O interface, and as an I/O bus. Each of these uses is briefly described below:

main data  
management processor

Because the PIOP contains its own parallel processor, instruction set, and program source memory, it can control data flow to or from the AP in data "bursts" of 6 MHz or in sustained data transfers of 3 MHz.

The PIOP can extract data from the AP's main data memory, perform arithmetic operations on the data, and then return the results to the AP main data memory. The data handled by the PIOP is processed in parallel with the AP's processing operations.

Although the PIOP does not have a main memory, it has access to the AP main data memory. This access, coupled with the high transfer rate between the PIOP and AP, permits the PIOP to use the AP main data memory for storage whenever required.

#### I/O interface

When functioning as an interface, the PIOP may be connected to any external device that is compatible with TTL logic. Unlike other interfaces that are hardwired to service a specific unit (such as a disk), the PIOP contains a 20-bit device register that permits the PIOP to serve as a programmable interface.

In other words, this device register can be programmed in such a manner that the PIOP generates and recognizes the handshaking and data signals required by the external device. Thus, microcode within the PIOP can respond to any type of external device. There is no need to change the hardware when servicing a different type of device.

#### I/O bus

The PIOP may be microprogrammed in such a way that it generates the typical signals of an I/O bus. When used in this manner, the PIOP serves as an interface between the AP and multiple external devices.

The flexibility of the PIOP allows it to perform the above tasks as needed, all under program control. For example, a typical PIOP operation might begin with data transfers to and from the AP for processing of data, then might serve as an I/O interface to handle transfers from the external device to the AP for processing (or from the device to the PIOP for integer calculations), then might once again serve as a parallel processor to the AP, then might serve as an I/O bus for handling multiple external devices, etc.

Although it is beyond the scope of this manual to describe hardware configurations, four possible configurations are shown in Figures 1-1 through 1-3 in order to illustrate the flexibility of the PIOP and to emphasize the importance of the programmer's task.

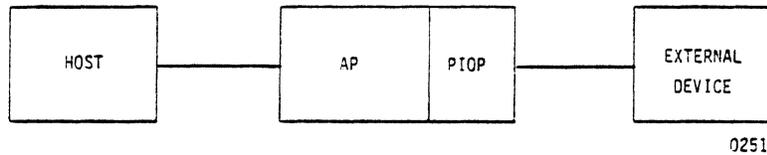


Figure 1-1 PIOP Used as Basic I/O Interface

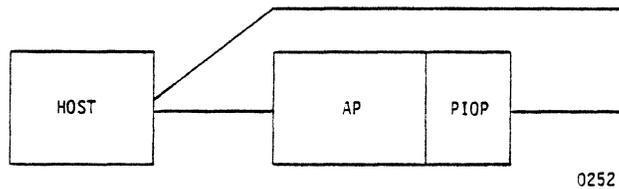


Figure 1-2 PIOP Used as Main Data Management Processor

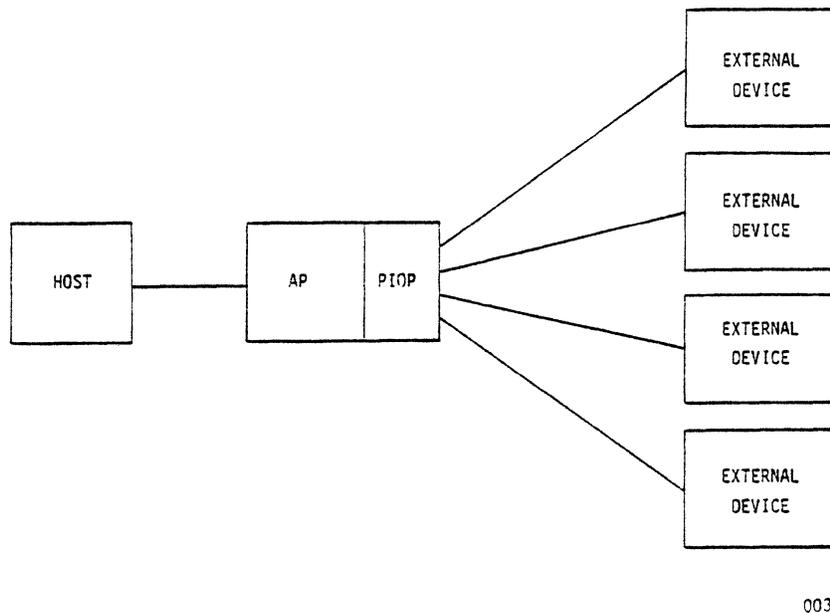


Figure 1-3 PIOP Used as an I/O Bus

#### 1.4 FUNCTIONAL DESCRIPTION

Although it is beyond the scope of this manual to provide detailed hardware information, it is necessary for the programmer to have a basic understanding of the PIOP hardware, particularly the data paths. These data paths permit information flow between the PIOP and other devices and also interconnect major PIOP components.

Figure 1-4 is an overall block diagram of the PIOP. The major functional units shown on the drawing are: ALU/scratchpad memory, CPU/program source memory, and buffer memory. This buffer is a first-in, first-out (FIFO) memory element. Other components shown in the drawing include the formatter and APMA, device command, IOR, and MDI registers. The following discussion of the block diagram shown in Figure 1-4 is intended primarily to indicate the flow of data through the PIOP. Subsequent block diagrams and related descriptions explain why these paths are used.

The ALU/scratchpad memory consists of an arithmetic and logic unit (ALU) and sixteen 20-bit registers that can be accessed by the programmer. The ALU performs arithmetic and logic operations required by the PIOP. The 16 registers serve as a scratchpad memory. Either the output of the ALU or the contents of any one of the registers (as selected by the programmer) can be loaded into the APMA and device command registers.

The APMA (AP memory address) register is used to supply the address in the AP's main data memory that is to be used when performing DMA transfers between the PIOP and the AP. The device command (DC) register performs two basic functions: it can be used to supply the DMA address when transferring data between the PIOP and the external device, or it can be used to supply necessary command and handshaking signals to the device. These signals might include such commands as start, stop, read, write, and acknowledge. Because the contents of the device command register can be determined by the programmer, the device command register can function as a programmable interface.

An output bus is used to send ALU status information to the CPU/program source memory. Such status information includes sign, zero, overflow, etc.

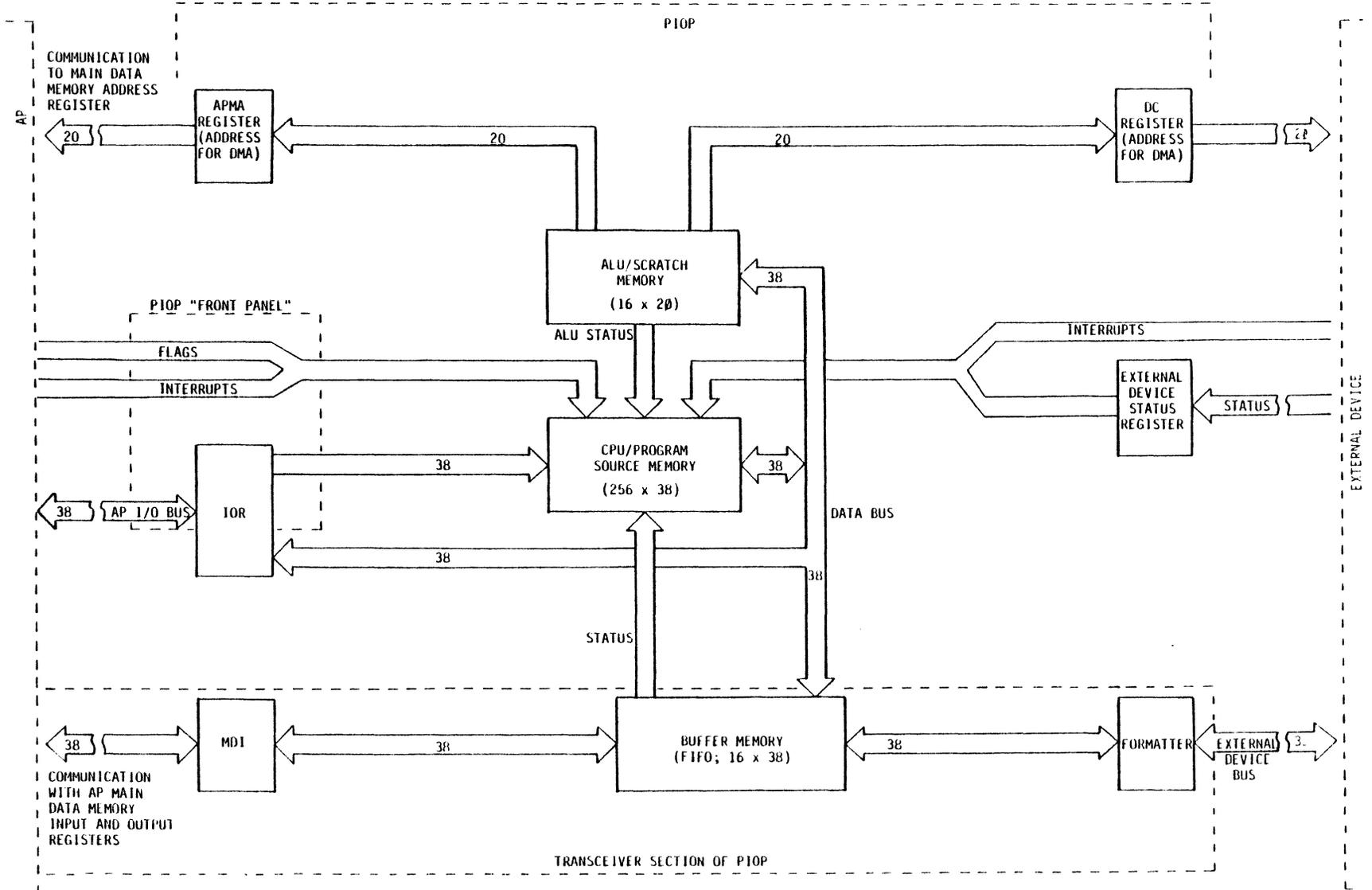


Figure 1-4 PIOP - Overall Block Diagram

The output of the ALU/scratchpad memory can also be applied to the data bus. This means that the output of the ALU or the contents of the selected scratchpad register can be applied to any one of three other components: the CPU/program source memory, the buffer memory, or the input/output register (IOR).

The CPU/program source memory consists of a CPU and a 256-word random access memory. This memory is used to store the instructions needed by the PIOP while the CPU decodes the instructions and directs all data flow by enabling the various PIOP elements as needed.

One of the functions of the CPU is to determine the address of the next instruction. Because of this, various status and interrupt signals are applied to the CPU. These signals include: flags and interrupts from the AP, status from the ALU, status and interrupts from the external device, and status from the buffer memory. The CPU uses this information to compute the required next address. In other words, one of the jobs of the CPU is to generate the branches and jumps (computed GO TO's) that are required.

The CPU/program source memory also receives information from the input/output register (IOR). This register permits communication between the PIOP and the AP and is used as a temporary storage device when transferring information into the CPU/program source memory. This information might be an instruction that is loaded directly into the program source memory, or it might be an instruction that is just decoded by the CPU for execution by the PIOP, or it might be data that the CPU routes to the data bus at the proper time.

The buffer memory contains a 16-word by 38-bit first-in, first-out (FIFO) element that compensates for different data rates between the PIOP, the AP, and the external device. For example, the external device may load the FIFO at a slow rate but the FIFO might be read by the PIOP in a high-speed burst.

The buffer memory output buffer can be loaded from a number of sources. If it is loaded from the data bus, it can receive information from either the ALU/scratchpad memory or from the CPU/program source memory. This means that the buffer could be loaded with data from the ALU, with data from any one of the 16 scratchpad registers, with instructions from the program source memory, or with the contents of the IOR (through the CPU).

The buffer memory can be loaded from the AP MD bus when it is communicating with the AP or it can be loaded from the external device through the formatter.

The output of the buffer memory can be sent to the AP (through the MDI register), to the external device (through the formatter and external data bus), or to the data bus where, under CPU control, it can be loaded into any component connected to the data bus.

The following paragraphs (1.4.1 through 1.4.3) describe the three major components of the PIOP: buffer memory, ALU/scratchpad memory, and CPU/program source memory.

#### 1.4.1 BUFFER MEMORY (FIFO)

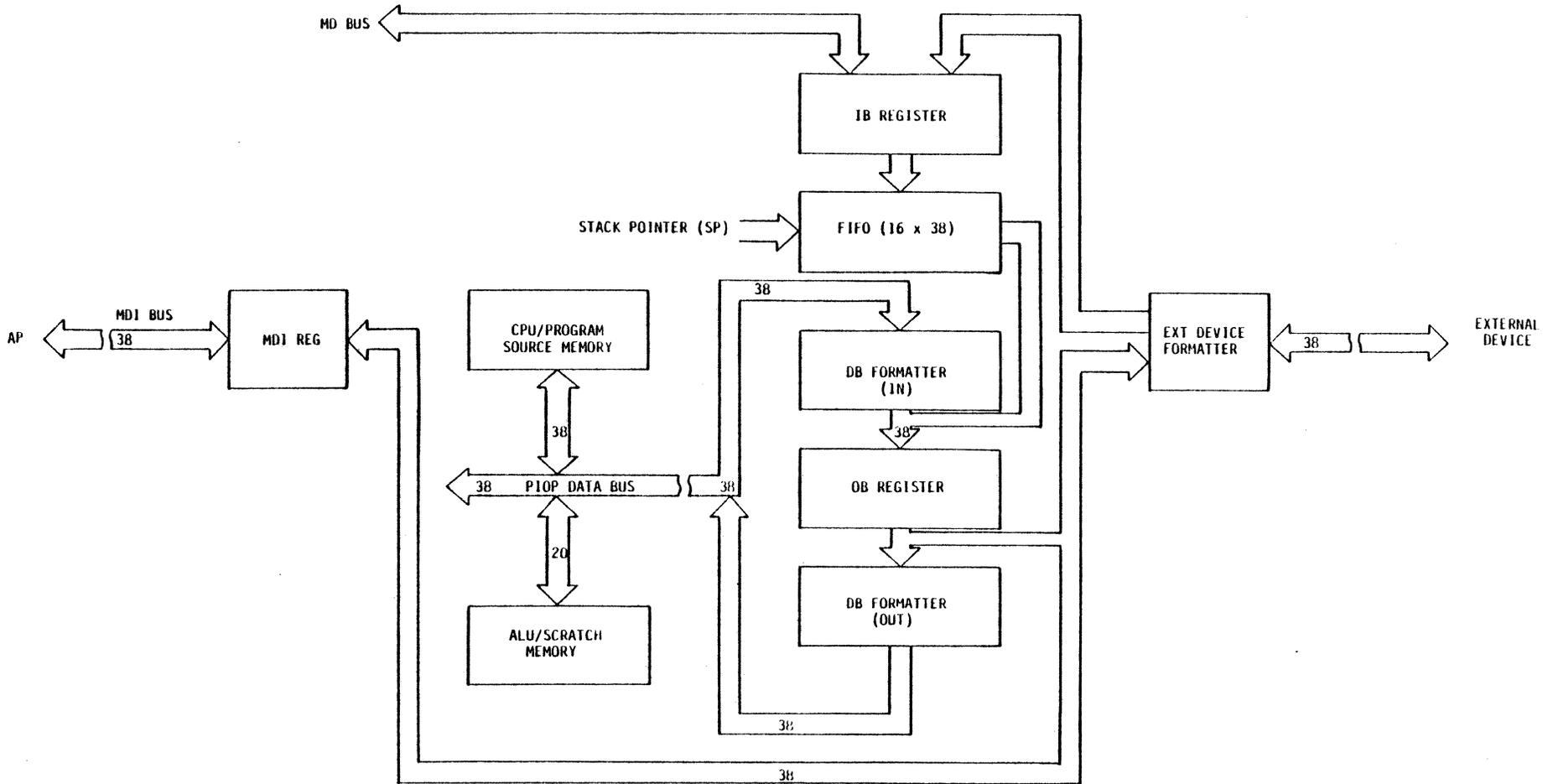
Figure 1-5 is a simplified diagram of the buffer memory which is also referred to as the "transceiver." The main purpose of the buffer memory is to compensate for different data rates between the AP and the external device. The AP, for instance, can load information into the FIFO at high speed and then the data can be retrieved by the external device at a slower rate. The buffer memory can also be used to store information from either the AP or the external device until the PIOP is ready to use the information.

Data from the AP is applied directly to the input buffer register while data from the external device is applied through the external device formatter to the same register. The data is then loaded into the FIFO (first-in, first-out) memory element. The FIFO location where the word is to be stored is controlled by a pointer.

When a word is to be retrieved from the FIFO, the pointer selects the proper word which is applied to the output buffer register. The output buffer can also be loaded from the data bus through its formatter.

The contents of the output buffer register can be sent to one of three places: to the MDI register (for transfer to the AP), to the external device formatter (for transfer to the external device), or to the PIOP data bus formatter (for transfer to other PIOP elements).

Notice that information on the data bus can be applied to the output buffer register. This permits data from other PIOP components to be loaded into the output buffer for transfer to either the AP or to the external device.



0145

Figure 1-5 Buffer Memory (Transceiver) - Simplified Diagram

#### 1.4.2 ALU/SCRATCHPAD MEMORY

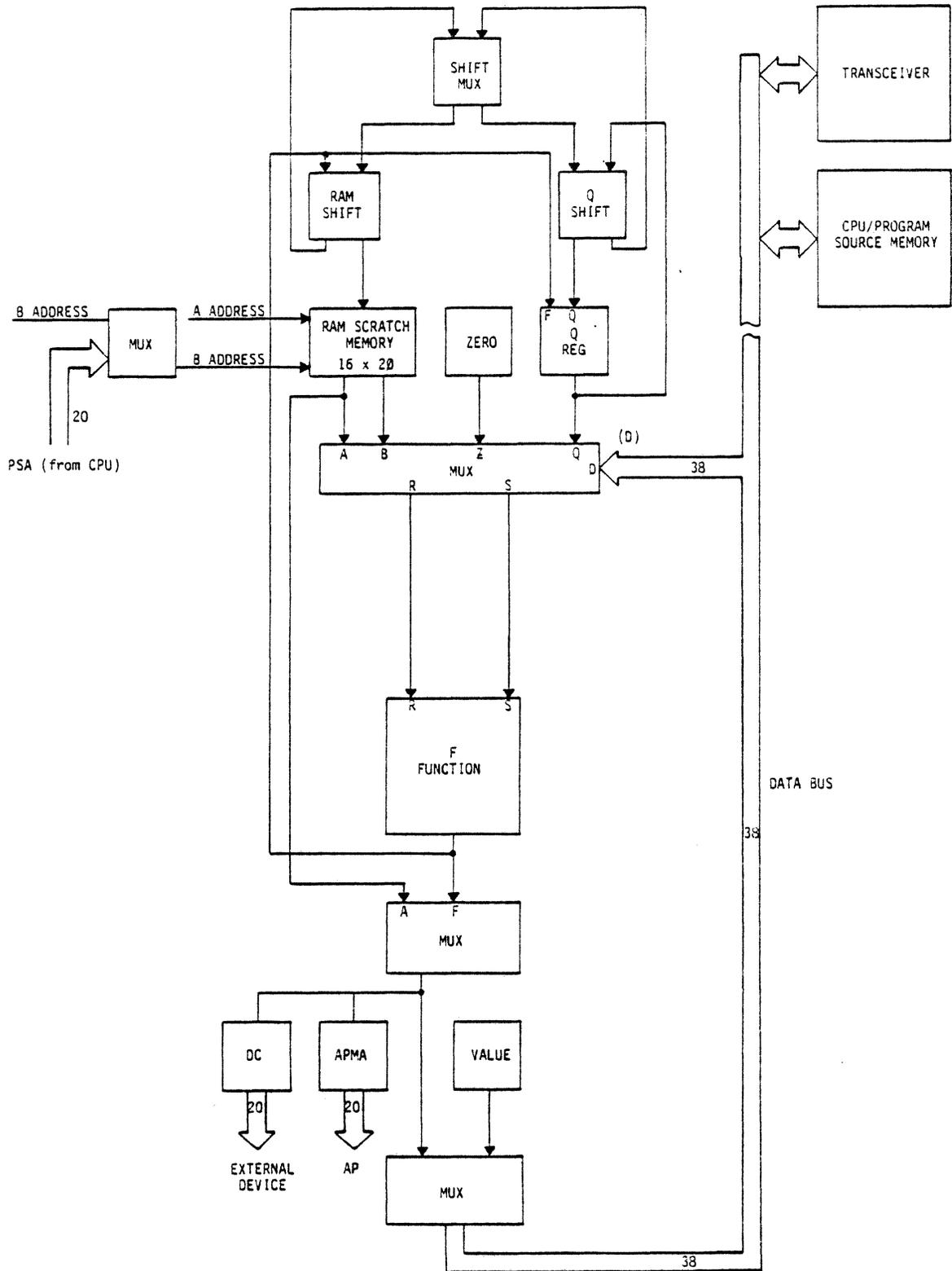
Figure 1-6 is a simplified diagram of the ALU/scratchpad memory within the PIOP. This component contains the ALU that performs the arithmetic and logic functions required by the PIOP and also contains a number of scratchpad registers that can be accessed by the programmer.

The scratchpad memory consists of sixteen 20-bit registers that can be accessed by the programmer. Any one or two of these registers can be selected at a time by the A and B address inputs to the memory. Notice that the B address is fed through a multiplexer. The other input to this multiplexer is the PSA (program source address) from the CPU. Thus, the program source address can be used to select one of the 16 registers, if desired (used for ALU dump/load from/to the AP).

The contents of the selected register, or registers, are applied to a multiplexer. Other inputs to this multiplexer are: all zeros, the contents of the Q register (an internal work register), the data on the data bus. This multiplexer, which is controlled by decoding a double-operand instruction, selects two of the five multiplexer inputs designated as R and S.

R and S are fed into the function ALU (F) which performs the necessary arithmetic or logic operation on the two inputs as specified by the CPU. The output of the ALU is then applied to another multiplexer. The other input to this multiplexer is the contents of the scratchpad register selected by address A.

The multiplexer output (Y) is applied to the device control (DC) register and to the AP memory address (APMA) register. Thus, depending on which of these two registers is enabled, the output of the ALU (or the contents of the register selected by address A) can be used as the address needed by the AP for DMA transfers, or it can be used as a command word for the external device.



0146

Figure 1-6 ALU/Scratchpad Memory - Simplified Diagram

Inputs to the final multiplexer on the drawing are: value (a specific value selected by the programmer) or the output of the previous multiplexer which is either the ALU output or the contents of register A. The selected output is then applied to the data bus where it can be fed to one of the other main PIOP components connected to the data bus.

Notice that the output of the function ALU (F) is fed back to both the RAM shift and Q shift logic. The output of the RAM and Q shift logic is fed to a shift multiplexer which performs any required shifts on the scratchpad or Q registers, under control of the CPU.

#### 1.4.3 CPU/PROGRAM SOURCE MEMORY

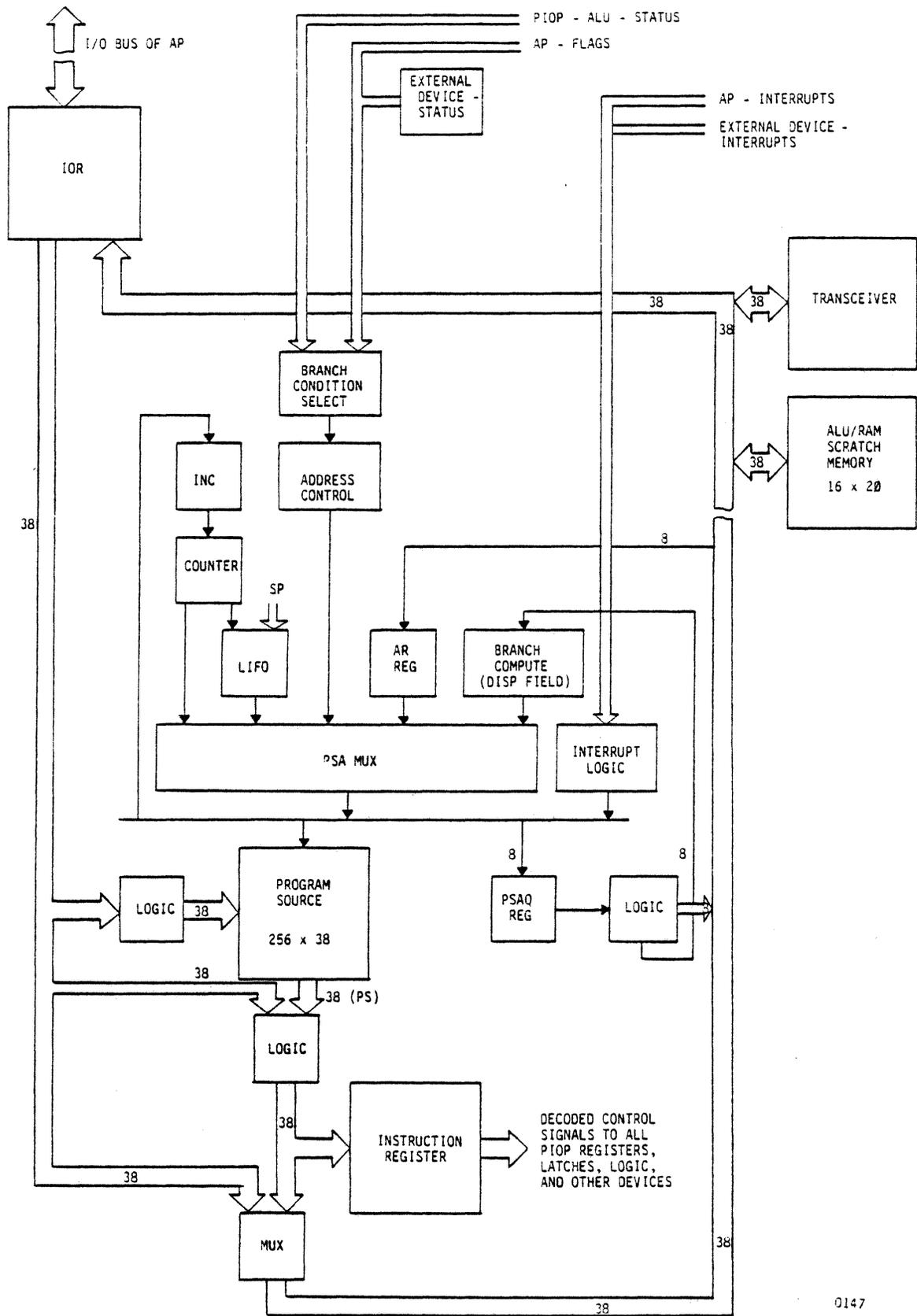
Figure 1-7 is a simplified diagram of the CPU/program source memory. This component consists of the program source memory which stores the instructions to be executed by the PIOP, the program source address logic which determines the address of the next instruction to be executed, and the control buffer which decodes the instruction and generates the control signals needed by the PIOP to perform the specified task.

A program counter, referred to as the "program source address" or "PSA," determines which instruction is to be executed next. This PSA can be changed by the branch, jump, and subroutine instructions.

The program source memory is loaded from the input/output register (IOR) by means of a PIOP instruction. This path is used because the program is typically stored in the AP's main data memory and then loaded into the PIOP's program source memory as needed.

The key to the program source memory is the PSA (program source address) which is shown on Figure 1-7 directly below the PSA multiplexer. It is this address that determines which instruction is to be retrieved from the program source memory. There are five inputs to this PSA multiplexer: the counter, the LIFO stack, the next address control logic, the address register (AR), and the branch condition select logic. Each of these inputs is described separately in subsequent paragraphs.

During a typical operation of the CPU/program source memory, the address of the instruction to be executed is placed on the data bus and loaded into the address register (AR). This address might come from the ALU or one of the 16 scratchpad registers, from the FIFO in the transceiver, or from the input/output register (IOR). If the multiplexer selects this input as its output, the contents of the address register are used to select the appropriate instruction in memory.



0147

Figure 1-7 CPU/Program Source Memory - Simplified Diagram

When sequential memory locations are to be addressed, the PSA is fed to an incrementer (INC) and applied through a register to the multiplexer. When this input is used as the multiplexer output, the PSA is the next sequential address.

Another input to the PSA multiplexer is the output of the LIFO (last-in, first-out) subroutine stack. Whenever the program jumps to a subroutine, the address of the next instruction in the main program is pushed on to this stack. When the program returns from the subroutine, the stack is popped and the former PSA is applied through the multiplexer to the memory. Loading (pushing) and retrieving (popping) addresses from the stack are under control of a stack pointer (SP).

Another input to the PSA multiplexer is the next address control logic which generates a memory address based on the branch condition select logic. In this case, inputs from the PIOP (ALU status), from the AP (flags), and from the external device (device) are applied to the branch condition select logic. Based on the condition tested and the results of the test, the branch condition select logic and the next address control logic compute the branch address needed. For example, the PIOP may test an error status bit in the external device, and if an error exists, branch to an error-handling routine. The address of the error-handling routine would be computed by the next address control logic and fed through the PSA multiplexer to the memory.

The output of the branch compute logic is another input to the PSA multiplexer. This logic generates a branch address based on a displacement field. This displacement (DISP) field is part of the instruction word and, therefore, under the programmer's control.

Another source of PSA's is the interrupt logic. The interrupt logic receives interrupt information from either the AP or the external device. Based on the interrupt information received, the interrupt logic generates the appropriate interrupt trap location (the first four locations in program source memory are reserved for traps).

Once the PSA multiplexer selects the appropriate input to be used as the PSA, the PSA is applied to the program source memory and is also stored in the PSAQ register. When this register is enabled, the PSA is controlled by a logic element that either sends the PSA back to the branch compute logic as an input, or places the PSA on the data bus. When placed on the data bus, the PSA can be sent to any one of four places: back to the address register, to the ALU/scratchpad memory, to the transceiver, or to the IOR.

The instruction selected by the PSA is retrieved from the program source memory and applied through logic to a control buffer (CB) and to a multiplexer. The control buffer decodes the instruction and provides all of the control signals needed by the PIOP logic elements. Either the instruction or the contents of the IOR can be selected by the multiplexer for application to the data bus.

## 1.5 MAJOR COMPONENTS

The major components of the PIOP are listed below and described in detail in Chapter 3 of this manual.

|                            |  |
|----------------------------|--|
| program source memory      | 38-bit by 256-word writable control store used for program instructions  |
| ALU registers              | sixteen 20-bit RAM registers that are part of the ALU. Typically used to hold arguments for PIOP instructions.                                       |
| Q register                 | 20-bit RAM register that is also part of the ALU. Used as a work register.   |
| address register           | used for programmed jump and branch addresses  |
| control register           | 20-bit register that selects transceiver format, indicates PIOP status, and arms interrupts  |
| I/O register               | used for communication between the AP and the PIOP; can be accessed by either system   |
| device command register    | 20-bit register that contains address and/or command information for the external device   |
| device status inputs       | 8 sense lines that indicate external device status   |
| ALU status                 | 8 bits that indicate status of the PIOP's ALU. These status bits appear as conditions which can be tested, and as bits in the PIOP control register. |
| AP memory address register | used by the DMA logic to save the AP main data address   |
| FIFO hardware element      | first-in, first-out hardware structure used for burst data handling. Can hold sixteen 38-bit words.  |
| LIFO stack                 | 4-word stack used for subroutine linkage   |
| transceiver                | formats and buffers data being transferred between the AP and the external device  |
| ALU                        | performs the arithmetic and logic functions required by the PIOP.  |

## 1.6 PIOP FEATURES

Some of the features of the PIOP that may be of interest to the programmer are briefly described below:

- LIFO stack                    A 4-level subroutine stack that permits nesting of subroutines. Stack pointer operation is automatic. This stack is used with both jump to subroutine and return from subroutine instructions.
- FIFO element                    A 16-word, first-in, first-out element that permits synchronous transfers of data between the AP and the PIOP at a 6 MHz rate. The write pointer is advanced automatically when writing data while a program instruction is used to advance the read pointer. Both pointers can be reset under program control.
- format handling                The transceiver can transfer data between the device and the AP in any one of four different formats. The format to be used can be selected by the programmer. Subfield addressing within each format is possible.
- instructions                    The instruction set is used in a micro-programmed format (38-bit instruction word) to allow parallel processing of multiple microinstructions.
- expanded formats                In the macro format, 4 bits of the instruction word can select 1 of 16 arithmetic or logic operations. In the expanded format, bit positions are redefined so that arithmetic operations can be selected by 12 bits (five fields) which increase the number of operations that can be performed.
- Other fields in the instruction word may also be redefined. This permits the programmer to use only those instructions needed for a particular job and makes programming simpler and more effective.

data transfer  
instructions

Separate instructions are provided for:

- a. transfers between the PIOP and the external device
- b. transfers between the PIOP and the AP's main data memory
- c. transfers between PIOP elements

This permits multiple transfers to take place in one instruction cycle.

branching

The PIOP instruction set includes four unconditional jump instructions and seven conditional branch instructions. Jumps may be either relative or absolute while branches are always relative.

interrupt arming

In addition to the capability of enabling or disabling interrupts, the PIOP also permits interrupts to be "armed" or "disarmed." If an interrupt occurs when an interrupt is disabled but "armed," the interrupt is not serviced but is, however, stored for future use. This interrupt can then be acted upon once it is enabled.

interrupts

Four interrupts trap to specific locations in program source memory. These traps occur only if the particular interrupt is armed and enabled.

In addition to the PIOP features, the software supplied with the system also permits easier and more efficient programming. Three such examples are:

- |                             |   |
|-----------------------------|---|
| programmable I/O<br>channel | A software interpreter (residing in the PIOP) that interprets channel programs stored in main data memory. Channel program instructions are structured in 4-word blocks. The first word is the op code and addressing modes, and the next three words contain arguments. One of three addressing modes (immediate, normal, and indirect) can be used with each argument. (See Chapter 6.) |
| assembler                   | The assembler provided with the PIOP is a 2-pass assembler written in Fortran IV. PIOP assembly language (PPAL) instructions are assembled for subsequent use by the PIOP debugger or from Fortran.   |
| Fortran calls               | A number of Fortran calls are available for communication with the PIOP, the AP disk, and the PIOP disk channel. In addition, there is a Fortran call for an executive loader.  |

## CHAPTER 2

### INSTRUCTION SET

#### 2.1 INTRODUCTION

This chapter introduces the instruction set so that the reader is exposed to the instruction word format and the various instructions and arguments that are used when programming the programmable I/O processor (PIOP).

It is not the purpose of this chapter to provide detailed descriptions of individual instructions but simply to present an overview so that the reader can become familiar with the structure of the instruction word and the mnemonics used for individual fields and instructions. Detailed information on individual instructions is presented in subsequent chapters of this manual.

The complete PIOP instruction set is presented in tabular form in Appendix A of this manual in order to provide a quick reference when using the PIOP instruction set.

## 2.2 INSTRUCTION WORD

The PIOP uses a 38-bit instruction word (bits 2 through 39) which is shown in Figure 2-1. When looking at the instruction word, it must be remembered that the PIOP is a parallel processor. Therefore, all operations selected by the entire instruction word are performed simultaneously.

Note that the basic fields in the instruction word may be redefined for other uses. Thus, for example, if the EXPAN field contains an octal 14, then bits 20 through 39 are redefined and become the VALUE field rather than the fields shown in the basic word. Because octal 14 happens to be the TCVR instruction (transmit VALUE to control register), this means that the programmer can load any desired value into the control register by placing the appropriate number in the VALUE field.

When using the assembler, the number to be loaded in the VALUE field is indicated by a literal following the instruction. For instance, if the programmer wants to load the number 40000 into the VALUE field, he or she would use the instruction, TCVR 40000. The literal must be a 20-bit positive number. The only other instruction that is used to load the VALUE field when using the assembler is the TVDB instruction (transmit VALUE to data bus).

The redefined fields shown in Figure 2-1 permit great versatility in programming. For example, although both unconditional jump instructions and conditional branch instructions are defined by the PSA CONTROL field, jump and branch instructions use different displacements. If the PSA CONTROL field contains a jump instruction, then the instruction uses the DISP8 field (8-bit displacement). This jump can be either an absolute or relative jump. In either case, the appropriate number is taken from the DISP8 field. On the other hand, the PSA CONTROL field uses the BIT # and DISP5 (5-bit displacement) fields if a branch instruction is used. In this case, the branch instruction tests a flag or status bit that has been specified by the BIT # field and then, if the proper condition is met, performs a relative branch based on the value in the DISP5 field.

Instruction word fields are used for commands and arguments. For example, the ALU field may use either one or two arguments. If only one argument is to be used, it is specified by either the A or B field. On the other hand, if two arguments are to be used, they are specified by the A and the B field.

Although a zero in certain fields results in no operation for that field, there is only one NOP (no operation) instruction. This instruction is all 0's in the entire PIOP instruction word. The NOP instruction is not listed in the instruction set but is available when using the assembler.

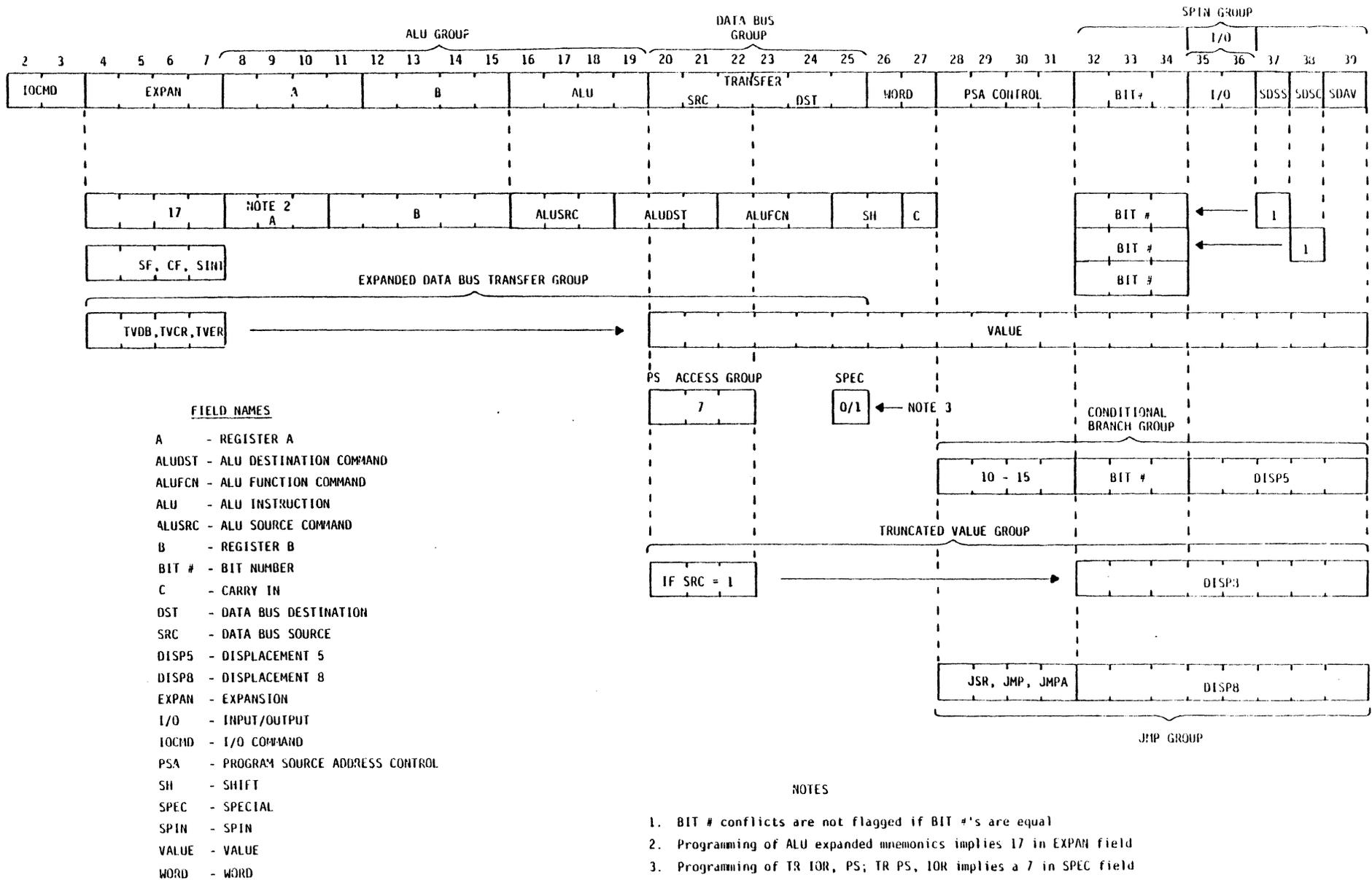


Figure 2-1 PIOP Instruction Word

### 2.3 INSTRUCTION SET

The entire PIOP instruction set is presented in tabular form. The tables are broken down according to instruction word fields.

The particular conventions that apply to the "shorthand notation" column only are listed in Table 2-1 below.

Table 2-1 Shorthand Notation Conventions

| SYMBOL | MEANING          | EXAMPLE                          | REMARKS  |
|--------|------------------|----------------------------------|--|
| ( )    | contents of      | source = (8)<br><br>source = PSA | Source equals the <u>contents of</u> register 8.<br><br>Source equals the program source address.  |
| > or < | moved into       | A > CR<br><br>PSA < PSA +1       | The contents of the register specified by the A field is moved into the CR register.<br><br>The program source address is incremented and moved into the program source address. |
| @      | deferred address | write @APMA                      | Write data into the location which has the address specified by the contents of the APMA register.   |

0038

The complete PIOP instruction set is presented in the following tables:

|           |                                  |   |
|-----------|----------------------------------|---|
| Table 2-2 | Basic PIOP Instruction Set       | Lists all of the instructions in the basic format.                        |
| Table 2-3 | Expanded PIOP Instruction Set    | Lists the instructions available in the expanded format.                  |
| Table 2-4 | Symbols Used For Expanded Format | Defines the symbols used in Table 2-3.                                    |
| Table 2-5 | Cross-Reference List             | Lists all of the instructions (basic and expanded) in alphabetical order. |

Table 2-2 PIOP Instructions

| FIELD | OCTAL CODE | MNEMONIC                  | MEANING   | DESCRIPTION   | SHORTHAND NOTATION | OTHER FIELDS (ARGUMENT) |
|-------|------------|---------------------------|---|---|--------------------|-------------------------|
| IOCMD | 0          | -                         | -   | No operation.   | -                  | -                       |
|       | 1          | SETMAR                    | <u>set memory address, read</u>   | Initiates a DMA read cycle to fetch data from the AP's main data memory at the address specified by the ALU output. Data is not available until 6 cycles later. The sequence is:<br><br><ol style="list-style-type: none"> <li>1. SETMAR instruction</li> <li>2. MDCR2* true (request to AP DMA channel)</li> <li>3. MDCA2 true (acknowledge from AP DMA channel)</li> <li>4. WAIT</li> <li>5. DCH02 (loads data into FIFO input buffer)</li> <li>6. FIFO</li> <li>7. DATA AVAILABLE (If FIFO was empty)</li> </ol> | Read @ APMA        | -                       |
|       | 2          | SETMAW                    | <u>set memory address, write</u>  | Initiates a DMA write cycle at the location specified by the ALU output. Data is written into the AP's main data memory. Data is available in memory after the third cycle. The sequence is:<br><br><ol style="list-style-type: none"> <li>1. SETMAW instruction</li> <li>2. CYCLE REQUEST (data in FIFO output buffer taken)</li> <li>3. CYCLE ACKNOWLEDGE (data now in memory)</li> </ol>   | Write @ APMA       | -                       |
| 3     | SETDA      | <u>set device address</u> | Loads the device control register with data present on the ALU bus at the end of the instruction cycle. The device control register is a write-only register. | ALU > DVCMD   | -                  |                         |

0039

Table 2-2 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC | MEANING                                   | DESCRIPTION   | SHORTHAND NOTATION  | OTHER FIELDS (ARGUMENT) |
|-------|------------|----------|---|---|---------------------|-------------------------|
| EXPAN | 0          | -        | -   | No operation.   | -                   | -                       |
|       | 1          | CF A     | <u>clear flag</u>                         | Clears the flag specified by A (A is specified in the BIT # field).   | Clear flag BIT #    | BIT #                   |
|       | 2          | RFF      | <u>reset FIFO</u>                         | Resets the FIFO pointers. Causes DATA VALID and FIFO FULL to go false (clear).<br><br>New data entering FIFO (through IN or SETMAR instructions or external handshake) falls through to the output buffer and causes DATA VALID to be true (set). | -                   | -                       |
|       | 3          | AFF      | <u>advance FIFO</u>                       | Advances FIFO read pointer. New data is written into FIFO output buffer at the end of the instruction cycle. If no valid words are in the FIFO, DATA VALID goes false (clear).  | -                   | -                       |
|       | 4          | SF x     | <u>set flag</u>                           | Sets the flag specified by x (x is specified in the BIT # field).   | Set flag BIT #      | BIT #                   |
|       | 5          | SINT x   | <u>set interrupt</u>                      | Sets interrupt x. The interrupt that is set executes in the second cycle after the SINT instruction.  | Set interrupt BIT # | BIT #                   |
|       | 6          | ENINT    | <u>enable interrupts</u>                  | Enables interrupt logic. Pending interrupts start executing on the next cycle.  | -                   | -                       |
|       | 7          | DISINT   | <u>disable interrupts</u>                 | Disables interrupt logic.   | -                   | -                       |
|       | 10         | NOP      | -   | No operation.   | -                   | -                       |
|       | 11         | START    | <u>start</u>                              | Begin program execution at current PSA location.  | Start               | -                       |
|       | 12         | HALT     | <u>halt</u>                               | Stop immediately. Nothing else in the instruction executes.   | PSA < PSA + 1       | -                       |
|       | 13         | PSAB     | <u>program source address, register B</u> | Causes the four least significant bits of the PSA to be used as ALU register address B. Can be used for sequential loading of ALU registers while PIOP is halted.   | PSA > B             | -                       |
|       | 14         | TVCR x   | <u>transmit value to control register</u> | Transfers the value x (from VALUE field) on to the data bus and loads the control register (CR) with that value at the end of this cycle.   | VALUE > CR          | VALUE                   |
|       | 15         | TVDB x   | <u>transmit value to data bus</u>         | Transfers the value x (from VALUE field) on to the data bus.  | VALUE > DB          | VALUE                   |
|       | 16         | -        | -   | Not used.   | -                   | -                       |
|       | 17         | -        | -   | Indicates expanded ALU instruction format.  | -                   | ALU EXPAN               |

0040

Table 2-2 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC  | MEANING  | DESCRIPTION   | SHORTHAND NOTATION  | OTHER FIELDS USED |
|-------|------------|-----------|--|---|---------------------|-------------------|
| A     | 1-17       | -         | -  | Contains address of one of 16 internal ALU registers.   | -                   | -                 |
| B     | 1-17       | -         | -  | Contains address of one of 16 internal ALU registers.   | -                   | -                 |
| ALU   | 0          | -         | -  | No operation.   | -                   | -                 |
|       | 1          | MOVB B    | <u>move data</u>                               | Move data bus contents to ALU register B. ALU output is that data.  | DB > B              | B                 |
|       | 2          | ADD A,B   | <u>add data</u>                                | Add the data bus contents to the contents of register A and store results in register B. ALU output is (DATA)+(A).                            | DB + A > B          | A,B               |
|       | 3          | ANDD A,B  | logical " <u>and</u> " of <u>data</u>          | Logically "and" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"and"(A).          | DB <u>and</u> A > B | A,B               |
|       | 4          | ORD A,B   | logical " <u>or</u> " of <u>data</u>           | Logically "or" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"or"(A).            | DB <u>or</u> A > B  | A,B               |
|       | 5          | XORD A,B  | logical " <u>exclusive or</u> " of <u>data</u> | Logically "exclusive or" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"xor"(A). | DB <u>xor</u> A > B | A,B               |
|       | 6          | PASSD     | <u>pass data</u>                               | Data on data bus passes through the ALU unchanged and unsaved. The data appears on ALU outputs.   | DB > Y              | -                 |
|       | 7          | PASSA A,B | <u>pass register A</u>                         | Data in register A is gated to ALU outputs. Data in register B is written in to itself. PASSA is a fast ALU path.                             | A > Y<br>B > B      | A,B               |
|       | 10         | INCB B    | <u>increment register B</u>                    | Increment register B contents. ALU output is (B) + 1.   | B + 1 > Y           | B                 |
|       | 11         | DECB B    | <u>decrement register B</u>                    | Decrement the ALU register B contents. ALU output is (B) - 1.   | B - 1 > Y           | B                 |
|       | 12         | INCD      | <u>increment data bus</u>                      | Increment data on the data bus (D) and pass through the ALU (not saved).  | DB + 1 > Y          | -                 |
|       | 13         | DECD      | <u>decrement data bus</u>                      | Decrement data on the data bus (D) and pass through the ALU (not saved).  | DB - 1 > Y          | -                 |
|       | 14         | ADD A,B   | <u>add register A to register B</u>            | Add register A to register B, store the results in register B. ALU outputs = (A) + (B).   | A + B > Y           | A,B               |
|       | 15         | SUB A,B   | <u>subtract register A from register B</u>     | Subtract register A from register B and store results in register B. ALU outputs = (B) - (A).   | B - A > Y           | A,B               |
|       | 16         | PASSB     | <u>pass register B</u>                         | Pass register B contents unchanged on to the Y bus.   | B > Y               | B                 |
|       | 17         | PASSQ     | <u>pass register Q</u>                         | Pass Q register contents to ALU BUS (Y).  | Q > Y               | -                 |

Table 2-2 PIOP Instructions (cont.)

| FIELD   | OCTAL CODE | MNEMONIC     | MEANING                               | DESCRIPTION  | SHORTHAND NOTATION                           | OTHER FIELDS USED |   |
|---|------------|--------------|---------------------------------------|--|--|-------------------|---|
| TRANSFER (SRC)  | 0          | TR ALU, _    | arithmetic and logic unit             | Source of the data bus is the ALU output (Y).  | DB < Y                                       | -                 |   |
|   | 1          | TR (DISP8) _ | displacement 8                        | Source of the data bus is the contents of the DISP8 field.                                       | DB < DISP8                                   | DISP8             |   |
|   | NOTE 1     | 2            | TR FF, _                              | FIFO   | Source of the data bus is FIFO input buffer. | DB < IB           | - |
|   | 3          | TR IOR, _    | input/output register                 | Source of the data bus is the contents of the I/O register.                                      | DB < IOR                                     | -                 |   |
|   | 4          | TR PSA, _    | program source address                | Source of the data bus is the program source address register.                                   | DB < PSA                                     | -                 |   |
|   | 5          | -            | -                                     | -  | -  | -                 |   |
|   | 6          | TR CR, _     | control register                      | Source of the data bus is the contents of the control register.                                  | DB < CR                                      | -                 |   |
|   | 7          | -            | -                                     | Indicates that the SPEC (special) field is to be used as the next field in the instruction word. | GO TO SPEC                                   | -                 |   |
| TRANSFER (DST)  | 0          | -            | -                                     | No operation.  | -  | -                 |   |
|   | 1          | -            | -                                     | -  | -  | -                 |   |
|   | NOTE 2     | 2            | TR, FF                                | FIFO   | Destination is the FIFO output buffer.       | DB > OB           | - |
|   | 3          | TR, IOR      | input/output register                 | Destination is the I/O register.   | DB > IOR                                     | -                 |   |
|   | 4          | TR, AR       | address register                      | Destination is the address register of the CPU.  | DB > AR                                      | -                 |   |
|   | 5          | -            | -                                     | -  | -  | -                 |   |
|   | 6          | TR, CR       | control register                      | Destination is the control register.   | DB > CR                                      | -                 |   |
|   | 7          | -            | -                                     | -  | -  | -                 |   |
| SPEC  | 0          | TR PS, IOR   | program source, input/output register | Transfers program source word into the I/O register (2-cycle instruction).                       | IOR < PS                                     | PSA CONTROL       |   |
|   | 1          | TR IOR, PS   | see above                             | Transfers contents of I/O register to program source (2-cycle instruction).                      | PS < IOR                                     | PSA CONTROL       |   |
| NOTES   |            |              |                                       |  |  |                   |   |
| 1. Source loaded on data bus at <u>beginning</u> of cycle.<br>2. Destination loaded on data bus at <u>end</u> of cycle. |            |              |                                       |  |  |                   |   |

Table 2-2 PIOP Instructions (cont.)

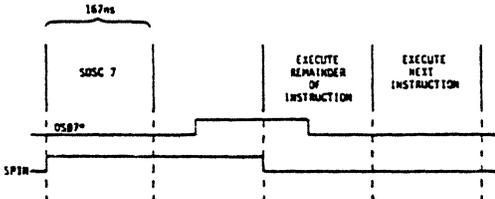
| FIELD       | OCTAL CODE | MNEMONIC  | MEANING                                 | DESCRIPTION  | SHORTHAND NOTATION                                   | OTHER FIELDS USED |
|-------------|------------|-----------|---|--|--|-------------------|
| PSA CONTROL | 0          | -         | -                                       | No Operation.  | -  | -                 |
|             | 1          | JMPAR     | <u>jump to address register</u>         | Absolute jump to address contained in the PIOP address register (AR). Address register can be loaded as a data bus destination. The contents of the register is the 8 LSB's of the data bus.<br><br>This instruction uses no other fields and is, therefore, useful for tight loops and computed GO TO's.  | PSA < AR   | -                 |
|             | 2          | JMPST     | <u>jump to stack</u>                    | Jump to address at top of stack. Does <u>not</u> change stack contents so is <u>not</u> a subroutine return instruction. This instruction uses no other fields.  | PSA < ST   | -                 |
|             | 3          | JMPA V    | <u>jump absolute</u>                    | Jump to absolute address V which is contained in the DISP8 field.  | PSA < DISP8  | DISP8             |
|             | 4          | POP       | <u>pop the stack</u>                    | Advance subroutine return stack to the next address. This instruction does <u>not</u> change PSA.  | -  | -                 |
|             | 5          | PUSH      | <u>push the stack</u>                   | Enter the current address plus one in to the subroutine return stack. This instruction does <u>not</u> change the PSA.   | PSA + 1 > ST   | -                 |
|             | 6          | RTN       | <u>return</u>                           | Jump to address at the top of the stack and advance the stack to the next address (POP the stack).   | POP AND JMPST  | -                 |
|             | 7          | JSR V     | <u>jump to subroutine, relative</u>     | Jump by the relative location V as specified by the DISP8 field. Enter the current location plus one into the stack.   | PSA < PSA + DISP8 ,<br>PUSH                          | DISP8             |
|             | 10         | BOSC x, y | <u>branch if device status is clear</u> | If device status BIT # x is clear, branch relative as specified by y. The maximum displacement is +17 to -20 octal locations. If x=7, then a high level on DS07* was sampled at the beginning of this instruction. y may be specified as a relative argument.<br><br><u>NOTE</u><br><br>DS07* is one of eight sense lines (DS00* - DS07*) that allow the PIOP to be controlled externally.   |  | BIT #, DISP5      |
|             | 11         | BOSS x, y | <u>branch if device status is set</u>   | Same as above except BIT # x must be set for the branch to occur (DS07* line low if x=7).  |  | BIT #, DISP5      |
|             | 12         | BFC x, y  | <u>branch if flag clear</u>             | If flag BIT # x is clear, branch relative as specified by y (DISP5). The maximum displacement is +17 to -20 octal locations.   | If condition is true, then:<br><br>PSA < PSA + DISP5 | BIT #, DISP5      |
|             | 13         | BFS x, y  | <u>branch if flag set</u>               | Same as above, except branch occurs if flag is set.  | If condition is <u>not</u> true, then:               | BIT #, DISP5      |
|             | 14         | BISC x, v | <u>branch if ALU status is clear</u>    | If internal status BIT # x is clear (zero), branch as specified by y (DISP5). Maximum displacement is +17 to -20 octal locations.<br><br>Internal status BIT # is defined as follows:<br><br>If set: 0 = FIFO data valid<br>1 = FIFO full<br>2 = R shift out<br>3 = Q shift out<br>4 = ALU carry<br>5 = ALU zero<br>6 = ALU sign<br>7 = ALU overflow<br><br>Note that bits 2 through 7 above also appear in the control register (CR). | PSA < PSA + 1  | BIT #, DISP5      |

Table 2-2 PIOP Instructions (cont.)

| FIELD       | OCTAL CODE | MNEMONIC  | MEANING                              | DESCRIPTION  | SHORTHAND NOTATION   | OTHER FIELDS USED |
|-------------|------------|-----------|--------------------------------------|--|--|-------------------|
| PSA CONTROL | 15         | BISS x, y | branch if ALU status is set          | <p>Same as BISC except that status must be set (1) for the branch to occur.</p> <p>These instructions are alternate mnemonics for the eight BISS and eight BISC mnemonics.</p> <p>BFV DISP Branch if FIFO data valid<br/>                     BFF DISP Branch if FIFO full<br/>                     BFOT DISP Branch if R-shift output = 1<br/>                     BQOT DISP Branch if Q-shift output = 1<br/>                     BC DISP Branch if carry set<br/>                     BZ DISP Branch if ALU=0<br/>                     BM DISP Branch if ALU is minus<br/>                     BOVF DISP Branch if overflow = 1</p> <p>BNFV DISP Branch if FIFO data <u>not</u> valid<br/>                     BNFF DISP Branch if FIFO <u>not</u> full<br/>                     BNFOT DISP Branch if R-shift output = 0<br/>                     BNQOT DISP Branch if Q-shift output = 0<br/>                     BNC DISP Branch if ALU carry out is 0<br/>                     BNZ DISP Branch if ALU is not 0<br/>                     BP DISP Branch if ALU is positive<br/>                     BNOVF DISP Branch if ALU overflow = 0</p> | <p>If condition is true, then:<br/>                     PSA &lt; PSA + DISP5</p> <p>If condition is <u>not</u> true, then:<br/>                     PSA &lt; PSA + 1</p> | BIT #, DISP5      |
|             | 16         | BNZST     | branch if ALU <u>not zero, stack</u> | <p>If ALU output is non-zero, branch to the location at the top of the stack. For example:</p> <p>TVDB 10; MOVD CNT<br/>                     PUSH<br/>                     DEC CNT<br/>                     BNZST<br/>                     HALT</p> <p>The above loops 10 times before halting.</p>  | PSA < ST   | -                 |
|             | 17         | JMP X     | <u>jump</u>                          | Jump unconditionally to the relative address specified by X.   | PSA < PSA + DISP8  | DISP8             |

0044

Table 2-2 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC | MEANING                                  | DESCRIPTION   | SHORTHAND NOTATION | OTHER FIELDS USED |
|-------|------------|----------|--|---|--------------------|-------------------|
| IO    | 0          | -        | -  | No operation.   | -                  | -                 |
|       | 1          | OUT      | <u>output</u>                            | Places FIFO output buffer contents on external device bus (DEV02* through DEV39*) and advances format logic. The format is specified by the FORMAT field in the control register.   | -                  | -                 |
|       | 2          | IN       | <u>input</u>                             | Loads the FIFO input buffer with data on the external device bus (DEV02* through DEV39*) at the end of the present cycle. This instruction also advances the format logic. The format is specified by the FORMAT field in the control register.   | -                  | -                 |
|       | 3          | IORST    | <u>input/output reset</u>                | Causes PIORST* (PIOP reset) to go true (low) which, by convention, resets all devices connected to the PIOP bus.  | -                  | -                 |
| SPIN  | BIT 37     | SDSC x   | <u>spin until device status is clear</u> | Spin until device status (BIT #) is clear. PIOP spins (waits) until device status line referenced by x (in BIT # field) is clear (high level) and then executes the remainder of that instruction. Device status state is sampled at the beginning of the instruction cycle.<br><br>   | -                  | BIT #             |
|       | BIT 38     | SDSS x   | <u>spin until device status is set</u>   | Same as above, except the DS07* level is inverted.  | -                  | BIT #             |
|       | BIT 39     | SDAV     | <u>spin until data available</u>         | Spin until FIFO data is available. The PIOP spins (waits) until the FIFO contains valid data.<br><br>SETMAR; PASSB BUF; RFF<br>SDAV; TRFF, DB; WORD 3; MOVD 0; AFF<br><br>The above instruction sequence puts valid data from the AP's main data location (buffer) into ALU register 0 and then the AFF resets the data valid flag. The spin is a minimum of five cycles. | -                  | -                 |

| INPUT/OUTPUT DATA FORMAT |      |        |                    |       |
|--------------------------|------|--------|--------------------|-------|
| FIELD                    | CODE | WORD   | DB TRANSFERS       | BITS  |
| WORD                     | 0    | WORD 0 | low mantissa (ML)  | 24-39 |
|                          | 1    | WORD 1 | high mantissa (MH) | 12-23 |
|                          | 2    | WORD 2 | exponent           | 2-11  |
|                          | 3    | WORD 3 | full word          | 2-39  |

0045

Table 2-3 Expanded PIOP Instructions

| FIELD  | OCTAL CODE | MNEMONIC | MEANING                  | DESCRIPTION   | SHORTHAND NOTATION                | OTHER FIELDS USED |
|--------|------------|----------|--------------------------|---|-----------------------------------|-------------------|
| ALUSRC | 0          | AQ       | -                        | <p>All of these codes are used to select the data source for the R and S input fields of the ALU.</p> <p>Note that A and B fields are deferred. That is, the A (or B) field selects one of 16 registers. The contents of the selected register is then moved into either the S or R input field of the ALU.</p> | A > R, Q > S                      | A                 |
|        | 1          | AB       | -                        |   | A > R, B > S                      | B                 |
|        | 2          | ZQ       | -                        |   | $\emptyset$ > R, Q > S            | -                 |
|        | 3          | ZB       | -                        |   | $\emptyset$ > R, B > S            | B                 |
|        | 4          | ZA       | -                        |   | $\emptyset$ > R, A > S            | A                 |
|        | 5          | DA       | -                        |   | DB > R, A > S                     | A                 |
|        | 6          | DQ       | -                        |   | DB > R, Q > S                     | -                 |
|        | 7          | DZ       | -                        |   | DB > R, $\emptyset$ > S           | -                 |
| ALUDST | 0          | Q        | Internal work register   | <p>All of these codes are used to select the destination that is to receive the ALU output function.</p> <p>Mnemonics are:</p> <p>Q = internal work register</p> <p>F = ALU function</p> <p>Y = ALU output bus</p> <p>Note that a right shift is a divide by 2 while a left shift is a multiply by 2.</p>       | F > Q, F > Y                      | ALUFCN            |
|        | 1          | NP       | -                        |   | F > Y                             | ALUFCN            |
|        | 2          | A        | A field                  |   | F > B, A > Y                      | A, ALUFCN         |
|        | 3          | F        | ALU function             |   | F > B, F > Y                      | B, ALUFCN         |
|        | 4          | RQ       | Right shift Q            |   | F/2 > B, Q/2 > Q, F > Y           | B, ALUFCN         |
|        | 5          | RF       | Right shift ALU function |   | F/2 > B, F > Y                    | B, ALUFCN         |
|        | 6          | LQ       | Left shift Q             |   | 2F > B, 2Q > Q, F > Y             | B, ALUFCN         |
|        | 7          | LF       | Left shift ALU function  |   | 2F > B, F > Y                     | B, ALUFCN         |
| ALUFCN | 0          | AD       | add                      | <p>These codes are the function performed by the ALU.</p> <p>R and S are ALU input operands.</p> <p>The ALUSRC field selects the source for R and S; the ALUDST selects the destination for the ALU output after the selected function has been performed.</p>  | F = R + S + C                     | ALUSRC, ALUDST    |
|        | 1          | SB       | subtract                 |   | F = S - R                         | ALUSRC, ALUDST    |
|        | 2          | SR       | subtract, reverse        |   | F = R - S                         | ALUSRC, ALUDST    |
|        | 3          | OR       | logical "or"             |   | F = R <u>or</u> S                 | ALUSRC, ALUDST    |
|        | 4          | AN       | logical "and"            |   | F = R <u>and</u> S                | ALUSRC, ALUDST    |
|        | 5          | NA       | logical "nand"           |   | F = " <u>not</u> " R and S        | ALUSRC, ALUDST    |
|        | 6          | XO       | exclusive "or"           |   | F = R <u>xor</u> S                | ALUSRC, ALUDST    |
|        | 7          | XN       | "exclusive "nor"         |   | F = " <u>not</u> " R <u>xor</u> S | ALUSRC, ALUDST    |
| SH     | 0          | -        | default                  | <p>Shift in zeros</p> <p>Shift in ones.</p> <p>Rotate (shift out becomes shift in)</p> <p>Sign extend on right shift; fill with zeros on left shift.</p>  | -                                 | -                 |
|        | 1          | N        | -                        |   | -                                 | -                 |
|        | 2          | R        | rotate                   |   | -                                 | -                 |
|        | 3          | A        | arithmetic shift         |   | -                                 | -                 |
| C      | 0          | -        | default                  | -   | F = F                             | ALUFCN            |
|        | 1          | I        | -                        | -   | F = F + 1                         | ALUFCN            |

Table 2-4 Symbol Definitions

| SYMBOL | DESCRIPTION   |
|--------|---|
| A      | Register A - one of 16 internal registers (scratchpad memory of ALU). The specific register to be used is specified by a 4-bit binary number in the A field.  |
| B      | <p>Register B address - one of 16 internal registers (scratchpad memory of ALU). The specific register to be used is specified by a 4-bit binary number in the B field.</p> <p style="text-align: center;"><u>NOTE</u></p> <p>The same 16 registers are used by both A and B fields. For example, the A field may specify register #2 while the B field may specify register #14.</p> |
| DB     | Data Bus - the bi-directional bus connecting the transceiver to the other PIOP circuits. The mnemonic DB is also used for data bus.   |
| Q      | Register Q - an internal work register.   |
| R      | ALU Input Register R - one of two inputs to the ALU. Designates the left-hand input in a double-operand statement.  |
| S      | ALU Input Register S - One of two inputs to the ALU. Designates the right-hand input in a double-operand statement.   |
| Y      | ALU Output Bus Y - indicates the output bus of the ALU. More specifically, the output of the ALU Bus Select Logic.  |
| Z      | Represents binary 0's. For example, the expression Z > R indicates that all zeros are loaded into the ALU R input register.   |
| F      | Results of the ALU function which are applied to the ALU destination.   |

0047

Table 2-5 Cross-Reference List

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION                                 |
|----------|--------|------------|--|
| A        | ALUDST | 2          | $F > B, A > Y$                                     |
| A        | SH     | 3          | -  |
| AB       | ALUSRC | 1          | $A > R, B > S$                                     |
| AD       | ALUFCN | 0          | $R + S + C$  |
| ADD      | ALU    | 14         | $A + B > B$  |
| ADD      | ALU    | 2          | $DB + A > B$                                       |
| AFF      | EXPAN  | 3          | -  |
| AN       | ALUFCN | 4          | $R$ <u>and</u> $S$                                 |
| ANDD     | ALU    | 3          | $DB$ <u>and</u> $A > B$                            |
| AQ       | ALUSRC | 0          | $A > R, Q > S$                                     |
| BDSC     | PSA    | 10         |  |
| BOSS     | PSA    | 11         | If condition is true, then:<br>$PSA < PSA + DISP5$ |
| BFC      | PSA    | 12         |  |
| BFS      | PSA    | 13         | If condition is not true, then:<br>$PSA < PSA + 1$ |
| BISC     | PSA    | 14         |  |
| BISS     | PSA    | 15         |  |
| BNZST    | PSA    | 16         | $PSA < ST$   |
| CF       | EXPAN  | 1          | clear flag BIT #                                   |
| DA       | ALUSRC | 5          | $DB > R, A > S$                                    |
| DB       | DST    | 0          | -  |
| DECB     | ALU    | 11         | $B - 1 > B$  |
| DECD     | ALU    | 13         | $DB - 1 > Y$                                       |
| OISINT   | EXPAN  | 7          | -  |
| DQ       | ALUSRC | 6          | $DB > R, Q > S$                                    |
| DZ       | ASUSRC | 7          | $DB > R, \emptyset > S$                            |

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION        |
|----------|--------|------------|---------------------------|
| ENINT    | EXPAN  | 6          | -                         |
| F        | ALUDST | 3          | $F > B, F > Y$            |
| HALT     | EXPAN  | 12         | Halt, $PSA < PSA + 1$     |
| IN       | IO     | 1          | -                         |
| INCB     | ALU    | 10         | $B + 1 > B$               |
| INCD     | ALU    | 12         | $DB + 1 > Y$              |
| IORST    | IO     | 7          | -                         |
| JMP      | PSA    | 17         | $PSA < PSA + DISP8$       |
| JMPA     | PSA    | 3          | $PSA < DISP8$             |
| JMPAR    | PSA    | 1          | $PSA < AR$                |
| JMPST    | PSA    | 2          | $PSA < ST$                |
| JSR      | PSA    | 7          | $PSA < PSA + DISP8, PUSH$ |
| LF       | ALUDST | 7          | $2F > B, F > Y$           |
| LQ       | ALUDST | 6          | $2F > B, 2Q > Q, F > Y$   |
| MOVD     | ALU    | 1          | $DB > B$                  |
| N        | SH     | 1          | -                         |
| NA       | ALUFCN | 5          | "not" $R$ and $S$         |
| NP       | ALUDST | 1          | $F > Y$                   |

0048

Table 2-5 Cross-Reference List (cont.)

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION      |
|----------|--------|------------|-------------------------|
| OR       | ALUFCN | 3          | R <u>or</u> S           |
| ORD      | ALU    | 4          | DB <u>or</u> A > B      |
| OUT      | IO     | 2          | -                       |
| PASSA    | ALU    | 7          | A > Y, B > B            |
| PASSB    | ALU    | 16         | B > Y                   |
| PASSD    | ALU    | 6          | DB > Y                  |
| PASSQ    | ALU    | 17         | Q > Y                   |
| POP      | PSA    | 4          | -                       |
| PSAB     | EXPAN  | 13         | PSA > B                 |
| PUSH     | PSA    | 5          | PSA + 1 > ST            |
| Q        | ALUDST | 0          | F > Q, F > Y            |
| R        | SH     | 2          | -                       |
| RF       | ALUDST | 5          | F/2 > B, F > Y          |
| RFF      | EXPAN  | 2          | -                       |
| RQ       | ALUDST | 4          | F/2 > B, Q/2 > Q, F > Y |
| RTN      | PSA    | 6          | POP and JMPST           |
| SB       | ALUFCN | 1          | S - R                   |
| SDAV     | SPIN   | -          | -                       |
| SDSC     | SPIN   | -          | -                       |
| SDSS     | SPIN   | -          | -                       |
| SETDA    | IOCMD  | 3          | ALU > DVCMD             |
| SETMAR   | IOCMD  | 1          | Read APMA               |
| SETMAW   | IOCMD  | 2          | Write APMA              |
| SF       | EXPAN  | 4          | Set flag BIT #          |
| SINDC    | IO     | 4          | -                       |
| SINDS    | IO     | 3          | -                       |
| SINT     | EXPAN  | 5          | Set interrupt BIT #     |
| SOTDC    | IO     | 6          | -                       |
| SOTOS    | IO     | 5          | -                       |
| SR       | ALUFCN | 2          | R - S                   |
| START    | EXPAN  | 11         | Start                   |
| SUB      | ALU    | 15         | B - A > B               |

| MNEMONIC     | FIELD   | OCTAL CODE | SHORTHAND NOTATION   |
|--------------|---------|------------|----------------------|
| TR ALU, --   | TR(SRC) | 0          | DB < Y               |
| TR,CR, --    | TR(SRC) | 6          | DB < (CR)            |
| TR(DISP8),-- | TR(SRC) | 1          | DB < (DISP8)         |
| TR FF, --    | TR(SRC) | 2          | DB < (IB)            |
| TR IOR, --   | TR(SRC) | 3          | DB < (IOR)           |
| TR IOR, PS   | SPEC    | 1          | PS < (IOR)           |
| TR PS, IOR   | SPEC    | 0          | (IOR) < PS           |
| TR PSA, --   | TR(SRC) | 4          | DB < (PSA)           |
| TR --, AR    | TR(DST) | 4          | DB > (AR)            |
| TR --, CR    | TR(DST) | 6          | DB > (CR)            |
| TR --, FF    | TR(DST) | 2          | DB > (OB)            |
| TR --, IOR   | TR(DST) | 3          | DB > (IOR)           |
| TVCR         | EXPAN   | 14         | VALUE > CR           |
| TVOB         | EXPAN   | 15         | VALUE > DB           |
| TVEX         | EXPAN   | 16         | VALUE > EXP          |
| WORD 0       | WORD    | 0          | -                    |
| WORD 1       | WORD    | 1          | -                    |
| WORD 2       | WORD    | 2          | -                    |
| WORD 3       | WORD    | 3          | -                    |
| XN           | ALUFCN  | 7          | "not" R <u>xor</u> S |
| XO           | ALUFCN  | 6          | R <u>xor</u> S       |
| XORD         | ALU     | 5          | DB <u>xor</u> A > B  |
| ZA           | ALUSRC  | 4          | Ø > R, A > S         |
| ZB           | ALUSRC  | 3          | Ø > R, B > S         |
| ZQ           | ALUSRC  | 2          | Ø > R, Q > S         |

## CHAPTER 3

### FUNCTIONAL ELEMENTS

#### 3.1 INTRODUCTION

As an aid in understanding the programmable I/O processor (PIOP), brief descriptions of the major functional elements of the system are presented in the following paragraphs. These paragraphs cover: the transceiver, the registers, and other PIOP elements. Note, however, that it is not the intent of this chapter to describe all PIOP functional elements but only those elements that are of interest to the programmer. All of the elements described are shown in the block diagrams in Chapter 1 of this manual.

### 3.2 TRANSCIEVER

The transceiver portion of the PIOP formats and buffers the data transferred between the external device and the AP. Thus, the transceiver can compensate for different device speeds.

A brief description of the transceiver and the associated FIFO (first-in, first-out) memory element is given below:

transceiver

Formats and buffers data transferred between the AP main data memory and the external device. The transceiver is under PIOP program control.

The transparent transceiver can transfer 38-bit words between the PIOP and the external device in one of four formats: full 38-bit word, three 16-bit words, two 16-bit words with truncated mantissa, or two 16-bit words with truncated exponent.

When transferring data between the transceiver and other PIOP elements, the full 38-bit word or the EXP, MH, ML fields.

When transferring data, the FIFO memory provides automatic word sequencing. However, the length of time it takes a word to sequence through the FIFO must be considered during programming to ensure that data is available at the proper time.

The transceiver contains input and output format logic, input and output buffer registers for the FIFO memory, and the FIFO (first-in, first-out) memory element. This memory element is described below.

transceiver FIFO  
memory element

A first-in, first-out (FIFO) memory used for data transfers. This memory can hold up to sixteen 38-bit words.

Data may be loaded into the FIFO input buffer from either the external device bus (DEV) or from the AP's main data memory bus (MD). Data is entered one word at a time. The FIFO output can be advanced under program control in order to "skip" words.

One application of the FIFO can be to compensate for different data rates between the PIOP, the AP, and the external device. For example, the external device may load the FIFO at a slow rate but the FIFO can be read by the PIOP in a high-speed burst.

### 3.3 REGISTERS

The PIOP contains various registers that can be used for such functions as loading addresses, reading status bits, issuing commands to external devices, loading operands for arithmetic and logical operations, etc.

A brief description of each of these registers is given below:

ALU registers  
(scratchpad memory)

16 individual 20-bit RAM registers in the ALU. These registers store operands for the ALU operations.

Specific registers are accessed by addresses in the A and B fields as specified by the ALU instruction currently being executed.

Q register

A 20-bit internal work register in the ALU. Loaded by the ALU instructions in the expanded format (ALUSRC and ALUDST fields).

address register (AR)

An 8-bit register used for computed GO TO's. The register contents are used by the JMPAR instruction as an absolute address.

I/O register (IOR)

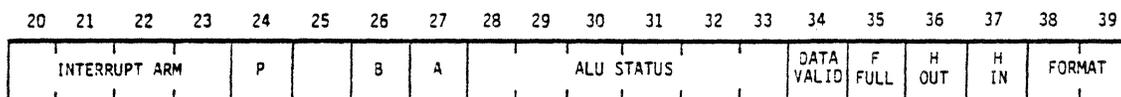
A 38-bit register used for communication between the AP and the PIOP. This register can be accessed by either the AP or the PIOP.

This register is also used as an intermediate storage device during program source fetch/store instructions.

This register appears to the AP as device address 100.

All of the previously-mentioned registers are used for addresses or data. The remaining registers are used primarily for command and status functions as described below. In addition, sense lines and individual register bits that provide status information are also described.

|                              |   |
|------------------------------|---|
| control register (CR)        | <p>A 20-bit read/write register used for internal PIOP control functions and status indications. The control register can be loaded by the TVCR VALUE instruction (transfer value to control register). The bit pattern for the control register is shown in Figure 3-1.</p>  |
| device command register (DC) | <p>A 20-bit write-only register that contains address and/or command information for controlling an external device. This register can be specified as a destination by the DST portion of the TRANSFER field.</p> <p>The meaning of each bit in this register is both program and device dependent. A typical bit assignment is shown in Figure 3-2.</p> |
| ALU status                   | <p>5 bits in the control register are used to indicate the status of the ALU. These bits can be tested by the BISS and BISC (branch if status set and branch if status clear) instruction. The individual ALU status bits in the control register along with other ALU status bits are listed in Table 3-1.</p>   |
| device status (DS)           | <p>8 sense lines that indicate the status of the external device used with the PIOP. The specific meaning of each line is dependent on the particular application.</p> <p>These sense lines can be tested by the BDSS, BDSC, SDSC, and SPIN instructions. A typical example of these lines when used as a disk interface is shown in Table 3-2.</p>       |
| flags                        | <p>8 flags that can be accessed by both the AP and the PIOP. These flags are defined by the program. A typical example is shown in Table 3-3.</p> <p>The flags can be set by the SF instruction, cleared by the CF instruction, and tested by the BFS and BFC instructions.</p>   |



| FIELD                      | BIT   | VALUE | NAME     | FUNCTION   |  |
|----------------------------|-------|-------|----------|--|--|
| INTERRUPT ARM              | 20    | 1     | INT0     | When set, arms interrupt zero which is the highest priority interrupt. |  |
|                            | 21    | 1     | INT1     | When set, arms interrupt one.  |  |
|                            | 22    | 1     | INT2     | When set, arms interrupt two.  |  |
|                            | 23    | 1     | INT3     | When set, arms interrupt three.  |  |
| ALU STATUS                 | -     | 24    | 1        | -  | Enable FIFO pass feature.  |
|                            | -     | 26    | 1        | -  | Indicates state of format logic.   |
|                            | -     | 27    | 1        | -  | Indicates state of format logic.   |
|                            | 7     | 28    | 1        | OVERFLOW   | When set, indicates that the output of the ALU is an overflow condition.   |
|                            | 6     | 29    | 1/0      | SIGN   | Indicates the sign of the ALU output. (0 = positive; 1 = negative.)  |
|                            | 5     | 30    | 1        | F=0  | When set, indicates that the output of the ALU function is zero.   |
|                            | 4     | 31    | 1        | CARRY  | When set, indicates that the ALU function resulted in a carry bit.   |
|                            | 3     | 32    | 1/0      | Q-SHIFT OUT  | Contains the bit that was shifted out during a Q-shift operation. Note that a Q-shift can be either a left or right shift. |
|                            | 2     | 33    | 1/0      | R-SHIFT OUT  | Contains the bit that was shifted out during an R-shift (RAM-shift) operation.   |
|                            | -     | 34    | 1        | DATA VALID   | Indicates that data is valid.  |
| -                          | 35    | 1     | FIFOFULL | Indicates that the FIFO is full.                                       |  |
| FIFO I/O HANDSHAKE ENABLES | 36    | 1     | -        | When set, enables the <u>output</u> handshake feature.                 |  |
|                            | 37    | 1     | -        | When set, enables the <u>input</u> handshake feature.                  |  |
| FORMAT SELECT              | 38,39 | 0 0   | -        | Transceiver format 0 (full words)                                      |  |
|                            |       | 0 1   | -        | Transceiver format 1 (EXP, MH, ML)                                     |  |
|                            |       | 1 0   | -        | Transceiver format 2 (high word, low word)                             |  |
|                            |       | 1 1   | -        | Transceiver format 3 (high word, low word)                             |  |

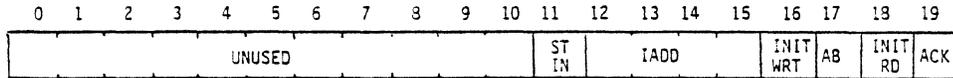
can be used for conditional branch instructions

NOTE

A more thorough description of transceiver formats is given in chapter 4.2.3.

0050

Figure 3-1 Control Register (CR)



| BIT     | VALUE | NAME               | FUNCTION   |
|---------|-------|--------------------|--|
| 0 - 10  | -     | -                  | Unused.  |
| 11      | 1     | /STROBE IN         | When set, transfers data between the PIOP and the disk input buffer.                         |
| 12 - 15 | 0/1   | /IADD0 thru /IADD3 | When a specific bit is set, selects the address of one of four disk registers.               |
| 16      | 1     | /INIT WRITE        | When set, initiates a write operation (data from PIOP is to be written into disk buffer).    |
| 17      | 1     | /ABORT             | When set, aborts the current disk operation.   |
| 18      | 1     | /INIT READ         | When set, initiates a read operation (data from disk is moved into disk buffer).             |
| 19      | 1     | /FIFO ACK          | When set, provides an acknowledge signal that is part of the disk/PIOP handshaking sequence. |

NOTES

1. The / before each command is a convention used with a disk.
2. The above is only a representative example of how the DVCMD register is used with an external device. In this example, a disk storage device is used.

0051

**Figure 3-2 Device Command Register (DC)  
(Assignment for Disk Interface)**

**Table 3-1 ALU Status Conditions**

| CR<br>BIT | BIT# | VALUE | NAME                      | FUNCTION  |
|-----------|------|-------|---------------------------|---|
| -         | 0    | = 1   | DATA VALID                | When set, indicates that the FIFO output buffer has been loaded. Reset by RFF or by advancing FIFO more times than it was loaded. |
| -         | 1    | = 1   | FIFO FULL                 | When set, indicates that the FIFO is full.  |
| 33        | 2    | = 1   | SHIFT OUT FROM F REGISTER | When set, indicates that there was a shift out of the function register of the ALU.   |
| 32        | 3    | = 1   | SHIFT OUT FROM Q REGISTER | When set, indicates that there was a shift out of the ALU's Q register (internal work register).                                  |
| 31        | 4    | = 1   | CARRY                     | When set, indicates that the ALU function resulted in a carry.  |
| 30        | 5    | = 1   | ZERO                      | When set, indicates that the ALU function resulted in a zero.   |
| 29        | 6    | = 0/1 | SIGN                      | Indicates the sign of the result of the ALU operation.<br>0 = positive<br>1 = negative  |
| 28        | 7    | = 1   | OVERFLOW                  | When set, indicates that the ALU operation resulted in an overflow condition.   |

Table 3-2 Device Status (Assignment for Disk Interface)

| <u>BIT</u> | <u>VALUE</u> | <u>NAME</u> | <u>FUNCTION</u>   |
|------------|--------------|-------------|---|
| 0 - 4      | -            | -           | Unused  |
| 5          | 1            | OPEN CABLE  | When set, indicates that an open cable condition exists.            |
| 6          | 1            | VERIFY      | When set, indicates that the disk is performing a verify operation. |
| 7          | 1            | FIFO REQ    | When set, provides the FIFO request handshaking signal.             |

0053

Table 3-3 Flags

| <u>BIT</u> | <u>SIGNAL</u> | <u>AP DEVICE ADDRESS</u> | <u>REMARKS</u>                              |
|------------|---------------|--------------------------|---|
| 0          | FLAG 0        | 110                      | AP - set by OUT<br>cleared by IN            |
| 1          | FLAG 1        | 111                      | When DA is set,<br>flag is gated to IORDY.  |
| 2          | FLAG 2        | 112                      |   |
| 3          | FLAG 3        | 113                      | PIOP - Flag set by SF<br>Flag cleared by CF |
| 4          | FLAG 4        | 114                      |   |
| 5          | FLAG 5        | 115                      |   |
| 6          | FLAG 6        | 116                      |   |
| 7          | FLAG 7        | 117                      |   |

0054

### 3.4 OTHER ELEMENTS

Other PIOP functional elements are listed below along with a brief description of each element. A more thorough discussion is contained in Chapter 4 of this manual.

program source  
memory

A 256-word by 38-bit writable control store which can be used to store data or program instruction words.

This memory is addressed by PSA (program source address) logic. Data is entered by means of the TR IOR,PS instruction and retrieved by the TR PS,IOR instruction. The selected word is decoded by the control buffer during execution time.

instruction register

During run time, decodes data from either the program source memory or from the I/O register (IOR) if the deposit is made from an AP program (SNSA, DA=100). The latter is explained more fully in paragraph 4.6.

PSA multiplexer

The program source address (PSA) is determined by the output of a multiplexer. This multiplexer has the following six input sources:

- a. program counter
- b. (DISP8) - the contents of the DISP8 field in the instruction word
- c.  $PSAQ + (DISP8) \text{ modulo } 256$  - the PSA register plus contents of DISP8
- d.  $PSAQ + (DISP8) \text{ biased}$  - the same as above but biased rather than modulo 256
- e. subroutine stack
- f. interrupts

## ALU

Performs the arithmetic and logic operations required by the PIOP. In addition, contains 16 registers that can be accessed by the programmer.

In addition to the normal instructions used to select arithmetic or logic operations (such as ADD and OR), the ALU field can be expanded.

The expanded field provides eight double-operand instructions for selecting the two ALU inputs, eight instructions for selecting the function to be performed, and eight instructions for selecting the destination that is to receive the output of the ALU.

## CHAPTER 4

### PROGRAMMING

#### 4.1 INTRODUCTION

This chapter provides the basic information needed to program the programmable I/O processor (PIOP) and is divided into five basic parts as shown in Table 4-1. Although this chapter is devoted to software, explanations of hardware are included where necessary for a complete understanding of the programming techniques. Although the reader was briefly exposed to the instruction set in Chapter 2, this chapter provides more detailed information on individual instructions.

The purpose of this chapter is to provide only the information needed for PIOP interface programming. However, because the PIOP is used with the array processor, information related to AP programming is included whenever necessary. If more detailed information on AP programming is desired, the reader should refer to the applicable publications listed in Chapter 1 of this manual.

Before referring to the subjects listed in Table 4-1, it is recommended that the reader consult the programming hints given in paragraph 4.1.1.

Table 4-1 Programming Subjects

| SUBJECT                   | PARAGRAPH | DESCRIPTION   |
|---------------------------|-----------|---|
| TRANSCEIVER               | 4.2       | Describes basic operation of the transceiver and FIFO memory element, word formats, the 8 instructions used in programming the transceiver, timing considerations, and DMA transfers. |
| ARITHMETIC & LOGIC UNIT   | 4.3       | Defines the ALU registers that can be accessed by the programmer, describes ALU operation, and explains how to use the ALU instructions in either the normal or expanded format.      |
| PROGRAM SOURCE MEMORY     | 4.4       | Describes the program source memory, the address control logic used with the memory, instructions related to program source memory, and the branch and jump instructions.             |
| INTERRUPT HANDLING        | 4.5       | Covers the instructions and timing used for handling interrupts.  |
| COMMUNICATING WITH THE AP | 4.6       | Describes how the PIOP communicates with the AP.  |

0055

#### 4.1.1 PROGRAMMING HINTS

The following hints are provided to aid the programmer in writing efficient programs for the PIOP. Each of the items described below is discussed more fully in the appropriate part of this chapter.

##### timing

Certain timing constraints must be considered when programming the PIOP. Because of the parallel nature of the PIOP, it is possible to write a microinstruction that appears to be valid but is actually illegal.

For example, a single instruction might request data and also request processing of that data. This is illegal because the entire instruction is executed in one cycle which means that the requested process does not have valid data to act upon. Therefore, certain PIOP instructions take two or more cycles to execute properly.

Instructions of this type fall into two categories: instructions that transfer data in and out of the PIOP, and instructions that transfer data in and out of the program source memory.

##### a. Transferring Data In/Out of the PIOP

The instructions affected by timing are listed below and described more fully in paragraphs 4.2.4 through 4.2.6.

IN  
OUT  
SETMAR  
SETMAW  
SETDA

##### b. Transferring Data In/Out of the PS Memory

The instructions affected by timing are listed below and described more fully in paragraph 4.4.

TR PS,IOR  
TR IOR,PS

##### traps

Always start programs at program source memory location 4 or higher because locations 0 through 3 are reserved for interrupt traps.

armed Always disarm interrupts in order to initialize  
interrupts the interrupts. Disarming clears unwanted  
interrupts that may be queued.

subroutines Make certain to end all subroutines with an RTN  
(return) instruction in order to return to the  
proper place in the main program.

#### 4.1.2 REFERENCE MATERIAL

Reference material that may aid the programmer is included in the appendices of this manual. The following material is included:

Appendix A Instruction word diagram and instruction set tables. Same as those included in Chapter 2 but reproduced in the appendix for quick reference.

Appendix B Interconnection diagram. Illustrates the various lines connecting the PIOP to the AP and to the external devices.

Appendix C Describes operation of the FIFO element and the subroutine return stack for those readers who might not be familiar with these concepts.

## 4.2 USING THE TRANSCEIVER

The transceiver basically consists of a FIFO (first-in, first-out) memory element and related logic such as formatting logic and input and output buffers. This 16-word memory element is used to provide buffering in order to synchronize transfers of data. For example, the external device might load up to 16 words in the FIFO in a fast data transfer and then these words might be retrieved from the FIFO, one word at a time, for transfer into the AP's main data memory. On the other hand, the external device might load the words at a slow rate and the PIOP might then retrieve the words in a high-speed transfer. The FIFO memory element may be used by either the external device or the AP.

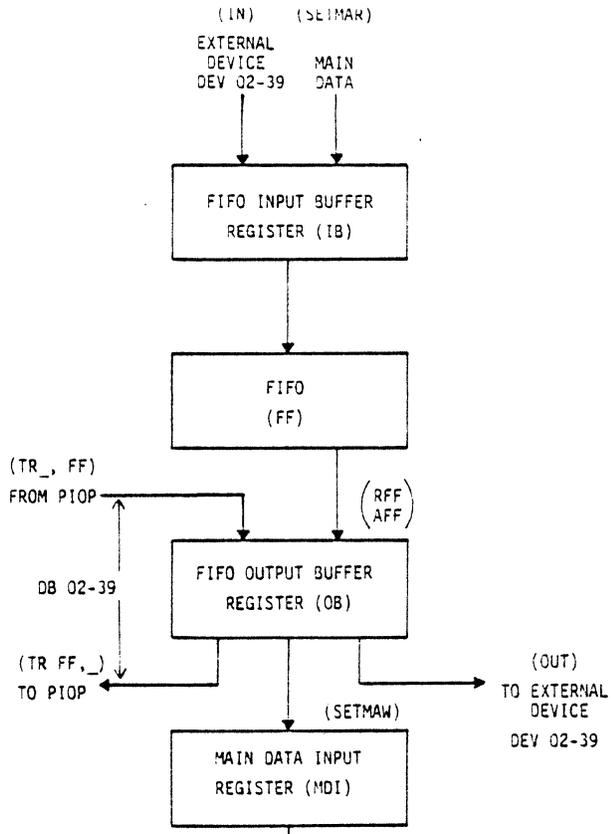
The following paragraphs provide a general description of transceiver operation, a discussion of the various word formats used with the transceiver, and an explanation of the instructions used when programming the transceiver.

A brief description of FIFO memory elements is contained in Appendix B of this manual.

### 4.2.1 TRANSCEIVER OPERATION

Figure 4-1 is a simplified diagram of the transceiver. Data from either the external device or the AP's main data memory is applied through a FIFO input buffer (IB) to the FIFO memory element. This memory element is capable of storing sixteen 38-bit words. Because of the memory's operation (first-in, first-out), the first word loaded into the memory is always the first word retrieved from the memory.

The word that is retrieved from the FIFO memory element enters the FIFO output buffer (OB). The output of this buffer can then be applied to one of three places: to the PIOP, to the external device, or to the main data input register (MDI) which supplies the data to the AP's main data memory.



NOTE  
 THE INSTRUCTIONS WHICH  
 IMPLEMENT THE PATHS  
 ON THE FIGURE ARE  
 SHOWN IN PARENTHESES.

0056

Figure 4-1 Transceiver - Simplified Diagram

As shown in the figure, data from the PIOP can also be applied directly to the FIFO output buffer (OB) for transfer to either the external device or through the MDI register to the main data memory.

The instruction mnemonics shown in parenthesis on the figure indicate the specific instruction that implements a particular data path. For example, the input buffer can receive data from the external device by means of the IN instruction or can receive data from the AP's main data memory by means of a SETMAR instruction.

Loading and retrieving data from the FIFO memory is controlled by two pointers: a write pointer and a read pointer. The write pointer is the address of the location to receive the next word when the FIFO is loaded. The read pointer is the address of the next word that is to be read. If both pointers are equal (read and write addresses identical), then the FIFO memory is either empty or full.

In order to understand the first-in, first-out operation of the FIFO memory, assume that both pointers are initially pointing to the first location in an empty FIFO. As each new word is loaded, the write pointer advances to the next sequential location but the read pointer does not move. The write pointer continues advancing as long as words are being loaded into the FIFO. When data is retrieved, the read pointer advances to the next location after the word has been read. Thus, when retrieving data, the first word out of the memory is the first word that had been loaded into memory.

It should be noted that there are two PIOP instructions directly related to the pointers used with the FIFO memory in the transceiver: RFF and AFF. A brief explanation of each of these instructions is given below:

|     |              |   |
|-----|--------------|---|
| RFF | Reset FIFO   | Resets <u>both</u> the read and write pointers by returning them to the first address in the memory.  |
| AFF | Advance FIFO | Advances the FIFO <u>read pointer</u> to the next sequential location in the memory. (The write pointer is advanced automatically whenever the low mantissa portion of the FIFO input buffer is written.) |

#### 4.2.2 TRANSCEIVER FORMATS

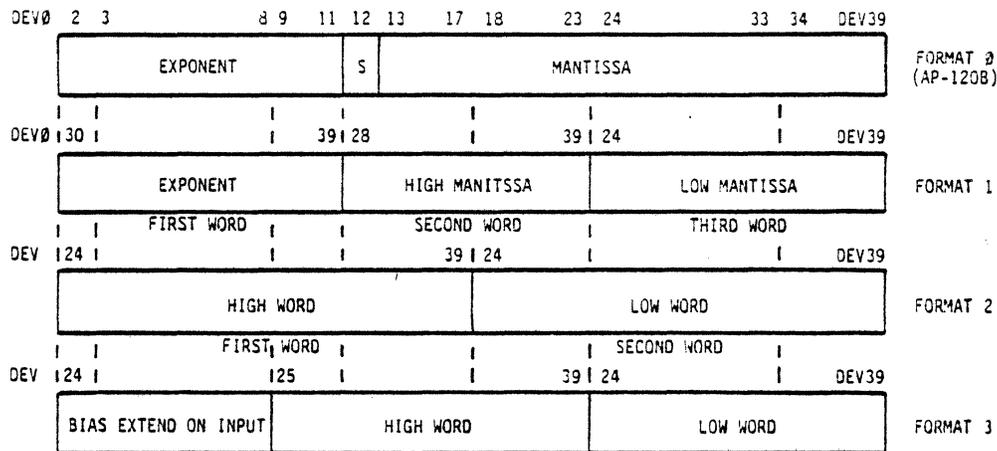
Although the transceiver always deals with a 38-bit AP floating-point word, the word may be transferred between the transceiver and the external device in any one of four different formats. Thus, the 38-bit word may be transferred as one full word, as three separate 16-bit words, as two separate 16-bit words with a truncated mantissa, or as two separate 16-bit words with a truncated exponent. Transfers can occur in either direction (from the external device to the transceiver or vice versa). It should be noted that these formats do not apply when transferring words between the transceiver and the internal data bus (DB).

The four possible transceiver formats are listed in Table 4-2 below and illustrated in Figure 4-2.

Table 4-2 I/O Word Formats

| DATA TYPE   | FORMAT | DESCRIPTION  |
|---|--------|--|
| 38-bit AP floating-point word (10-bit exponent; 28-bit signed mantissa) | 0      | Full word is transferred.  |
|   | 1      | Full word transferred as three separate 16-bit words. These words are: low mantissa, high mantissa, and exponent.      |
|   | 2      | Mantissa truncated and remainder of word transferred as two 16-bit words. These two words are: low word and high word. |
|   | 3      | Exponent truncated and remainder of word transferred as two 16-bit words. These two words are: low word and high word. |

0057



IN/OUT WORD DEFINITIONS

DB TRANSFERS

(REGARDLESS OF SELECTED FORMAT)

FORMAT 0 - N/A

FORMAT 1 - 1st = EXPONENT  
 2nd = HIGH MANTISSA  
 3rd = LOW MANTISSA  
 (advance word pointer)

FORMAT 2 - 1st = HIGH WORD  
 2nd = LOW WORD  
 (advance word pointer)

FORMAT 3 - 1st = HIGH WORD  
 2nd = LOW WORD

WORD 0 - LOW MANTISSA  
 WORD 1 - HIGH MANTISSA  
 WORD 2 - EXPONENT  
 WORD 3 - FULL WORD

AUTOMATIC  
 SEQUENCING

NOTES

1. AFF INSTRUCTION MAY BE USED TO ADVANCE FIFO READ POINTER.
2. DB TRANSFERS CAN SET "DATA VALID" BUT WILL NEVER ADVANCE READ OR WRITE POINTERS.
3. POINTERS WILL ADVANCE WITH ANY FULL WORD OR LOW BYTE TRANSFER (INCLUDING MD).
4. WORD FIELD ONLY AFFECTS DATA BUS TRANSFERS.

0058

Figure 4-2 Transceiver Formats

### 4.2.3 TRANSCEIVER INSTRUCTIONS

There are eight instructions that are directly related to transceiver operation. These instructions are listed in Table 4-3 below and described in the following paragraphs.

Table 4-3 Transceiver Instructions

| INSTRUCTION | FIELD | DESCRIPTION   |
|-------------|-------|---|
| IN          | IO    | Input. Data transfer into the PIOP.   |
| OUT         | IO    | Output. Data transfer out of the PIOP.  |
| SETMAR      | IOCMD | Set memory address, read. Performs a DMA read from main data memory.  |
| SETMAW      | IOCMD | Set memory address, write. Performs a DMA write to the main data memory. Advances the read pointer.                       |
| AFF         | EXPAN | Advance FIFO. Advances the FIFO read pointer.   |
| RFF         | EXPAN | Reset FIFO. Resets FIFO read and write pointers.  |
| TR FF,OB    | S&C   | Assembler mnemonic. Transfer contents of the output buffer (OB) to the data bus.  |
| TR DB,FF    | DST   | Assembler mnemonic. Transfer data on the data bus to the input buffer (IB). Sets the data valid bit (FIFOMT* goes false). |

0059

**IN** - This instruction is used for data transfers into the transceiver. The IN instruction transfers a data word from the external device into the FIFO input buffer (IB). The format is determined by the state of bits 38 and 39 (FORMAT field) in the control register while the particular word to be transferred is determined by the format logic.

In the cycle following the IN instruction, the contents of the input buffer (IB) are written into the FIFO memory. If the FIFO is empty at this time (that is, at the beginning of the instruction cycle after the IN instruction), then the valid data word is available in the FIFO output buffer (OB) at the beginning of the next cycle.

For example:

```
RFF          "reset FIFO pointers
IN           "load word into IB from device bus
NOP
TR FF,DB    "data word from the IB is available
            "in the OB. DATA VALID is true.
```

OUT - This instruction is used for data transfers out of the transceiver. During the cycle in which the OUT instruction occurs, the contents of the OB are gated to the external device data lines according to the data format type selected by the control register FORMAT field (bits 38 and 39) and the data word as selected by the transceiver format logic.

For example:

```
OUT          "transfer word on to device bus from
            "FIFO output buffer. At the end of the
            "cycle, advance to the next word as
            "determined by the format
```

SETMAR - This instruction initiates a DMA cycle to fetch data from the location specified by the ALU output. Five cycles later, data enters the input buffer (IB). This 38-bit data overwrites the previous contents of the IB. If the FIFO memory is empty, then data is available in the output buffer (OB) after the next cycle.

For example:

```
RFF          "reset FIFO pointers
SETMAR;ADD 0,1 "load APMA with (REG 0) (REG 1)
NOP          "cycle request
NOP          "cycle acknowledge
NOP          "wait
NOP          "data is written into input buffer
NOP          "data passes through FIFO
TR FF,DB    "data is available here
```

SETMAW - This instruction determines the memory address for a write operation. The SETMAW instruction initiates a DMA write cycle at the location specified by the ALU output. Data is taken from the FIFO output buffer (OB) at the end of the next cycle. Three cycles later, the data arrives in main data memory at the specified address. The FIFO is advanced on the cycle after the SETMAW instruction.

For example:

|                           |                                     |
|---------------------------|-------------------------------------|
| TR IOR,FF;INCB CNT;SETMAW | "load APMA with (CNT)+1; load       |
|                           | "FIFO output buffer (OB) from IOR   |
| NOP                       | "cycle request; during this cycle,  |
|                           | "data is read from the OB. The FIFO |
|                           | "pointer is advanced.               |
| NOP                       | "cycle acknowledge                  |

AFF - This instruction advances the FIFO pointer (the FIFO write pointer is advanced whenever the low mantissa portion of the input buffer is written). When this instruction is used, the output buffer contains a new word at the beginning of the next cycle. If no word was written, the word is not valid.

For example:

|          |  |
|----------|--|
| RFF      | "reset FIFO pointers                       |
| IN       | "load a word into the input buffer (IB)    |
| IN       | "load a second word into the IB            |
| NOP      | "DATA VALID true. Word transferred by      |
|          | first IN instruction is available here.    |
| AFF      | "advance FIFO read pointer; the word       |
|          | transferred by the first IN instruction    |
|          | is still available here.                   |
| TR FF,DB | "word transferred by second IN instruction |
|          | is available here.                         |

RFF - This instruction resets both FIFO pointers so that the FIFO memory appears to be empty. It also initiates the format control logic to input or output the first word of the selected format.

For example (assume full word format selected):

|              |  |
|--------------|--|
| IN           | "transfer first word into input buffer (IB)  |
| IN           | "transfer second word into IB  |
| IN           | "transfer third word into IB   |
| RFF          | "reset FIFO pointers   |
| IN           | "transfer fourth word into IB  |
| IN           | "transfer fifth word into IB   |
| NOP          | "fourth word available here (first through third words no longer available because of RFF) |
| TR FF,DB;AFF | "fourth word still available here  |
| TR FF,DB     | "fifth word available here   |

TR FF,DB - This instruction transfers the contents of the output buffer (OB) to the data bus according to the word specified by the WORD field in the PIOP instruction word.

TR DB,FF - This instruction loads the FIFO output buffer (OB) from the data bus according to the format specified by the FORMAT field in the control register and the word specified by the WORD field in the PIOP instruction word.

A RFF instruction should always follow a TR DB, CR instruction before data is transferred to/from the external device.

Interaction of these eight transceiver-related instructions is described in subsequent paragraphs.

#### 4.2.4 TRANSCEIVER TIMING CONSIDERATIONS

When using certain PIOP instructions that are related to the transceiver, the sequence of actions begun by one instruction may overlap the sequence of actions begun by another instruction. In addition, data from one instruction may not be available until a few cycles later, depending on the condition of the FIFO memory. Therefore, it is important to know the instruction timing considerations in order to program the PIOP properly. The instructions that require consideration of timing are the instructions which interact with external devices. These instructions are:

|        |  |
|--------|--|
| IN     | Strobes data at the end of the instruction cycle |
| OUT    | Places data on to the output bus                 |
| SETMAR | Initiates a DMA read cycle                       |
| SETMAW | Initiates a DMA write cycle                      |

Instruction timing is covered in this section by providing a number of typical examples. If it is necessary to review the transceiver instructions, refer to paragraph 4.2.4.

##### Example 1      Using an IN instruction with an empty FIFO memory

The timing for this example is shown in Figure 4-3. This example shows the timing of an IN instruction when the FIFO memory is empty. Note that the data requested by the IN instruction is not available for the PIOP until two cycles later.

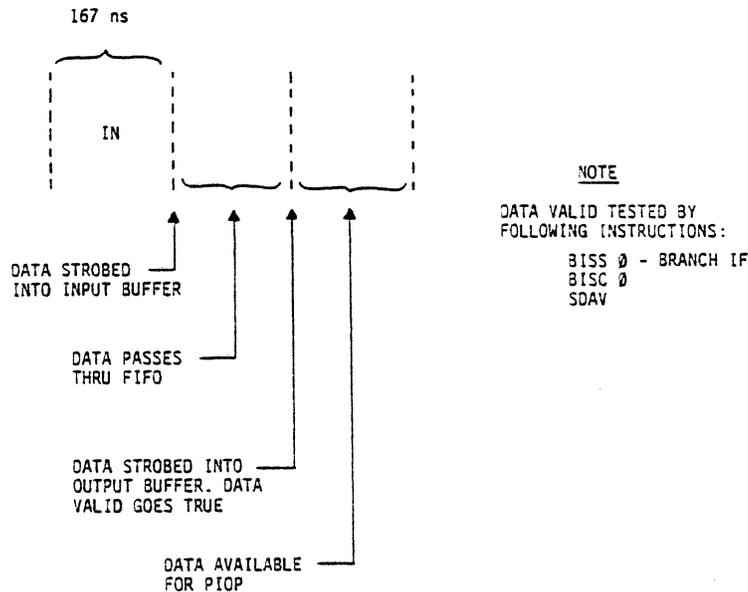


Figure 4-3 IN to Empty FIFO

0060

Example 2

Multiple IN instructions with an empty FIFO memory

The timing for this example is shown in Figure 4-4. This example shows the timing when the FIFO memory is empty and a number of IN instructions are used. Note that the data read by the first IN instruction is available at the beginning of the third cycle. This word is still available at the beginning of the sixth cycle. By the time the sixth cycle begins, two subsequent words have been stored in the FIFO memory.

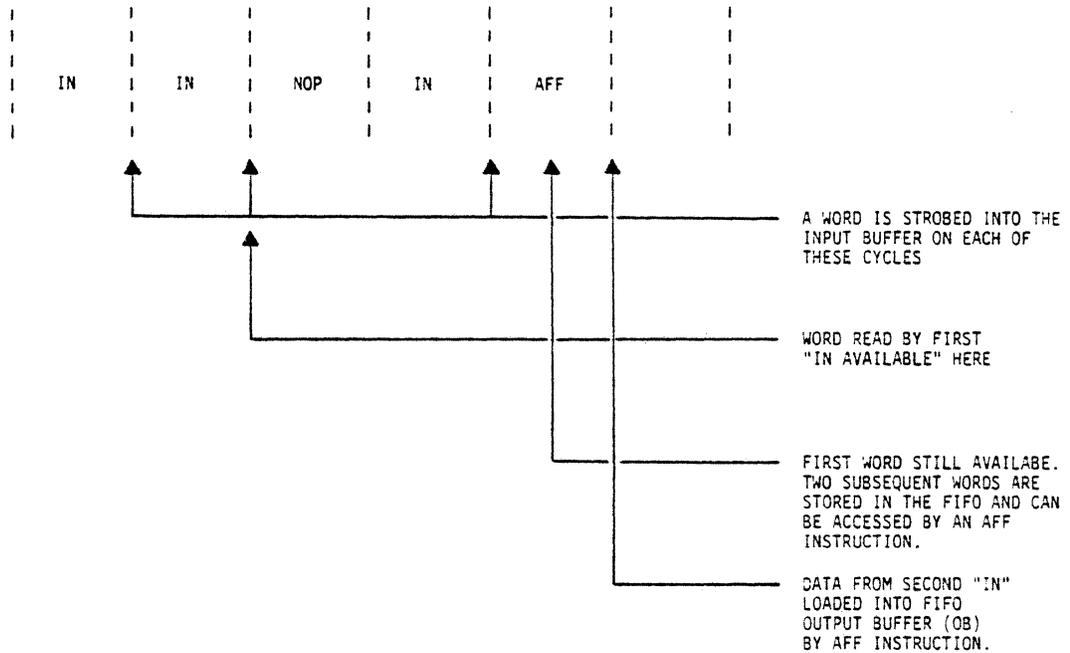


Figure 4-4 Multiple IN's to Empty FIFO

0061

Example 3

SETMAW Instruction

The timing for this example is shown in Figure 4-5. This example assumes that there is a valid word in the FIFO memory. This word could be there already, or it could be placed there by one of the following methods:

- a. an AFF instruction in the same cycle
- b. a TR DB,FF instruction in the same cycle
- c. an IN instruction in the previous cycle

Note that the data to be written by the SETMAW instruction is not written into the MDI register until the end of the second instruction cycle.

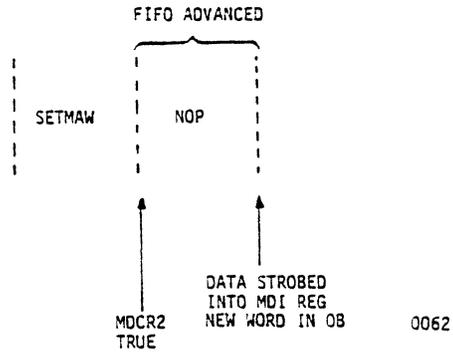
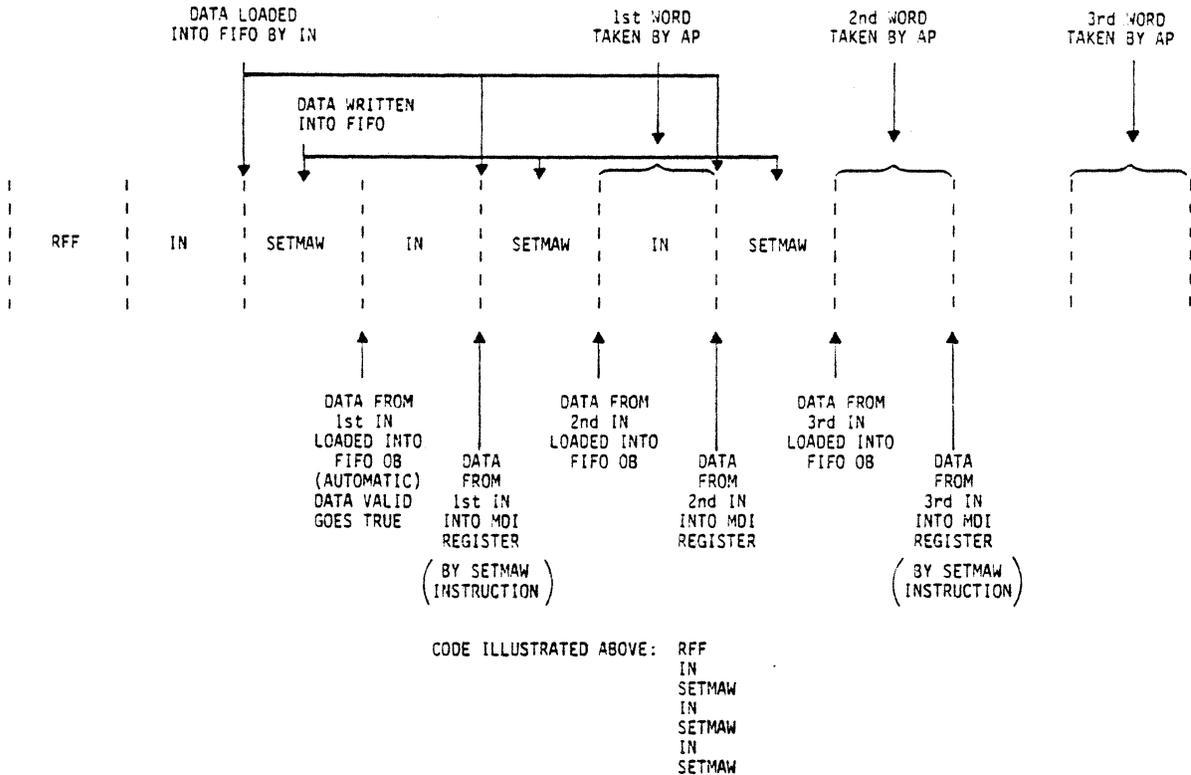


Figure 4-5 SETMAW Instruction

Example 4

3 MHz IN Instruction, SETMAW Loop

The timing for this example is shown in Figure 4-6.



CODE ILLUSTRATED ABOVE: RFF  
IN  
SETMAW  
IN  
SETMAW  
IN  
SETMAW

NOTE  
NO ADDRESS GENERATION IS  
SHOWN FOR SETMAW INSTRUCTION

0063

Figure 4-6 SETMAW Loop

Example 5

SETMAW IN Loop

The timing for this example is shown in Figure 4-7. The figure illustrates the following program. Note that it is assumed that DS00\* goes low to indicate external device data ready. Also, the program assumes that APMA = ALU register 0.

```
IN;SDSS 0                                "IN to initialize
                                           FIFO memory

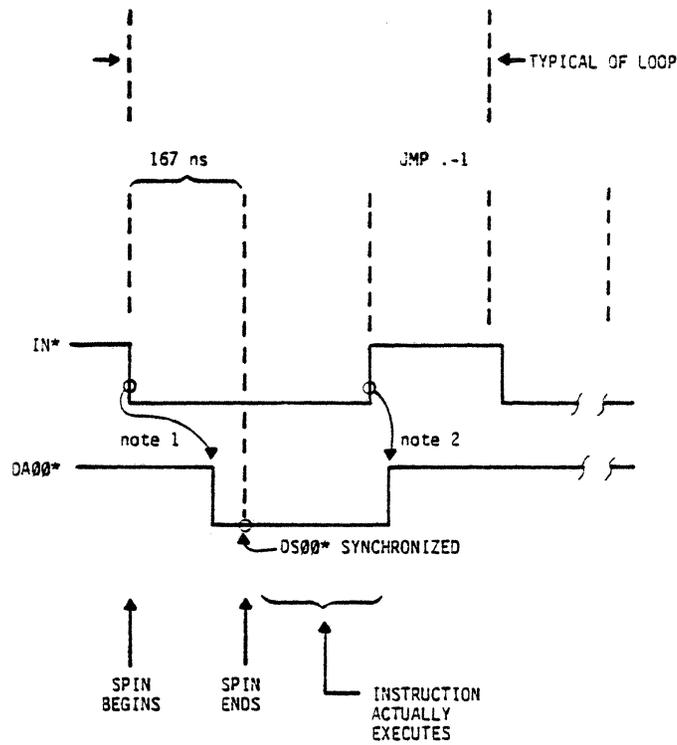
NOP

SETMAW;IN;SDSS 0;INCB 0                  "Perform DMA write to
                                           "AP main data memory;
                                           "wait for device
                                           "ready; increment ALU
                                           "register 0

NOP

SETMAW;IN;SDSS 0;INCB 0                  "Perform DMA write to
                                           "AP main data memory;
                                           "wait for device
                                           "ready; increment ALU
                                           "register 0

JMP .-1                                   "Loop to previous
                                           "instruction
```



NOTES

- 1 DS00 MUST BE TRUE WITHIN 125 ns OF IN\* GOING LOW TO GET THE NEXT CYCLE.
- 2 DS00 MUST BE FALSE WITHIN 125 ns OF IN\* GOING HIGH OR THE NEXT CYCLE PROCEEDS WITHOUT HANDSHAKING.
- 3 DATA MUST REMAIN VALID UNTIL IN GOES FALSE.

0064

Figure 4-7 SETMAW IN Loop

Example 6

SETMAR OUT Loop

The timing for this example is shown in Figure 4-8. The figure illustrates the following program. Note that it is assumed that DS00\* goes low to indicate valid data. Also, the program assumes that APMA = ALU register 0.

```
SETMAR;INCB 0          "Set up a buffer of
                        "data to cover up the
                        "access time

NOP

SETMAR;INCB 0

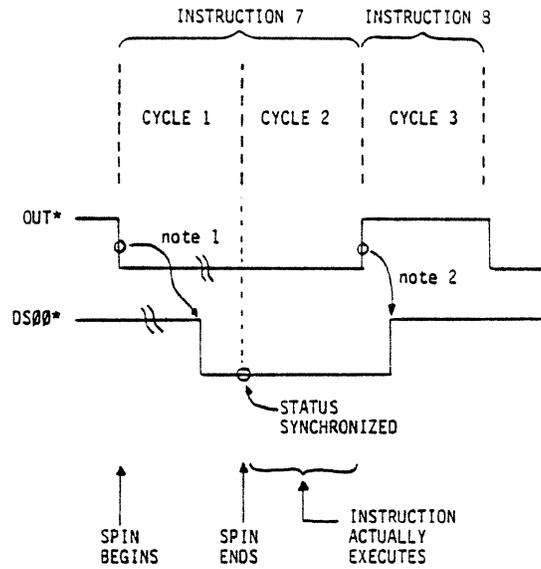
NOP

SETMAR;INCB 0

NOP

SDSS 0;SETMAR;INCB 0; OUT "Spin out, initiate
                           "read for later. OUT
                           "advances the FIFO
                           "to the next word

JMP .-1
```



**NOTES**

- 1 DS00 MUST BE TRUE WITHIN 125ns OF OUT\* GOING LOW TO GET NEXT CYCLE.
- 2 DS00 MUST BE FALSE WITHIN 125ns OF OUT\* GOING HIGH OR THE NEXT CYCLE PROCEEDS WITHOUT HANDSHAKING
- 3 DATA IS VALID DURING TIME OUT\* IS LOW

0065

**Figure 4-8 SETMAR OUT Loop**

#### 4.2.5 INTERACTION OF TRANSCEIVER INSTRUCTIONS

Interaction between some of the transceiver instructions occurs when the sequence of actions begun by one instruction overlaps the sequence of actions begun by another instruction.

Instruction interaction is best described by using a few examples. It is assumed that the reader is already familiar with the transceiver instructions and timing considerations as described previously.

- Example 1        SETMAR  
                  NOP  
                  NOP  
                  RFF  
                  NOP                "data is written here  
                  NOP  
                  TR FF,DB
- Example 2        TR 1,FF;SETMAW        "1 is written into main data  
                  TR 2,FF        "because 2 is not gated into the  
                                  "output buffer until the end of  
                                  "the cycle after the output buffer  
                                  "is gated into the MDI register
- Example 3        RFF  
                  IN  
                  IN  
                  NOP  
                  SETMAW;AFF        "this writes data from the second IN  
                                  "because data is in the output buffer  
                                  "after SETMAW is loaded into the MDI  
                                  "register
- Example 4        RFF  
                  IN  
                  IN;AFF  
                  SETMAW            "data from second IN is written into  
                                  "main data. This is similar to  
                                  "example 3 except the AFF was placed  
                                  "before SETMAW

|                  |              |   |
|------------------|--------------|---|
| <u>Example 5</u> | IN;RFF<br>IN | "at this point the FIFO is empty;<br>"therefore, the output buffer is<br>"overwritten in the next cycle |
|                  | NOP          | "first IN data available in the   |
|                  | TR FF,DB     | "output buffer second IN data<br>"available in the output buffer  |

#### 4.2.6 DMA TRANSFERS

All data transfers between the AP and the PIOP are direct memory access (DMA) transfers. The direction of transfer is determined by either a read (SETMAR) or write (SETMAW) instruction. Regardless of the direction of data transfer, the output of the ALU is moved into the PIOP's AP memory address register (APMA) which indicates the address used for the data transfer. When transferring data, data is moved between the AP and the FIFO memory in the transceiver portion of the PIOP.

The SETMAR and SETMAW instructions each perform two basic operations. The first operation sets up the main data memory address by loading the output of the ALU into the APMA register. The second operation is the DMA transfer of data.

The APMA register is loaded from the ALU each time a SETMAR or SETMAW instruction is executed. Five cycles later, the 38-bit data word appears in the FIFO input buffer (IB). At this point, the data is available for gating on to the data bus or for transmission to the external device. Therefore, it takes six cycles before the results of a DMA read operation can be used. However, because a DMA request can be made every other cycle, the effective rate can be as high as one word every two cycles.

The six cycles that occur after a DMA read (SETMAR) instruction are:

- 0 = SETMAR
- 1 = cycle request
- 2 = cycle acknowledge
- 3 = wait
- 4 = DCH01
- 5 = FIFO
- 6 = data valid at the beginning of this cycle

Because the APMA is loaded from the ALU, it is necessary for the programmer to know the output of the ALU. This is normally handled by setting up the ALU with the proper value when using either a SETMAR or SETMAW instruction.

As an example of how to set up the ALU with the proper value, assume that it is desired to read data from location 12 in the AP's main data memory. The following instruction could be used:

```
TVDB 12.; PASSD; SETMAR
```

The TVDB 12. instruction transmits the value in the VALUE field (which is decimal 12 in this example) to the data bus. The PASSD instruction moves the data on the data bus (D) to the ALU output bus. Therefore, the value 12. becomes the ALU output when these two instructions are executed. The SETMAR instruction then takes the ALU output (which is now 12.) and loads it into the APMA. Because SETMAR reads data from the address specified by the contents of the APMA, data is read from main data memory location 12 when the SETMAR is executed.

Another method of setting up addresses for DMA transfers is to load the starting address in one of the ALU's internal registers. Assume, for example, that register 6 has been loaded with a starting address of 200. Sequential DMA transfers could be made by using the following instructions:

```
PASSB 6; SETMAR
NOP           "cannot execute two SETMAR's in a row
INCB 6; SETMAR
JMP .-1
```

The PASSB 6 instruction moves the contents of register 6 to the ALU output bus. Thus, the APMA points to location 200 for the first DMA. The INCB 6 instruction increments the contents of register 6 and the JMP .-1 causes the program to go back to the first instruction. Because register 6 has been incremented, the SETMAR instruction now reads the contents of main data memory location 201. This process is repeated and data is read from sequential memory locations. Unfortunately, this results in a never-ending loop. Therefore, a more realistic example is given in the next paragraph.

One method of performing DMA read or write transfers from or to sequential memory locations is to set up three internal ALU registers as follows:

```
register 1 = counter (number of words to be transferred)
register 2 = starting address (first memory location to be read or
                    loaded)
register 3 = increment value
```

For the sake of this example, assume that register 1 is loaded with 7, register 2 with 200, and register 3 with 1.

Once these registers have been set up, the following instructions can be used to perform sequential DMA transfers:

|                 |  |
|-----------------|--|
| ADD 3,2; SETMAR | "Adds the increment value to the starting<br>"address and moves the resultant value into<br>"register 2. Because the result of an ADD is<br>"on the ALU output bus, the value now in<br>"register 2 is loaded into the APMA register.<br><br>"Because this instruction is set up as part<br>"of a loop, register 2 should actually be<br>"loaded with the starting address minus the<br>"increment value. Therefore, the first time<br>"the instruction is executed, it performs the<br>"DMA read at the correct starting address. |
| DEC 1           | "Decrements the counter to indicate one word<br>"had been transferred.   |
| BNZ .-2         | "If the counter is not zero, branches back<br>"to the first instruction and repeats the<br>"sequence.  |
| HALT            | "If the counter is zero, indicating all 7 words<br>"were transferred, the program stops. The AP can<br>"test to see if the program is running or not.  |

Typically, the first instruction in the above example would have an OUT following the SETMAR as shown below:

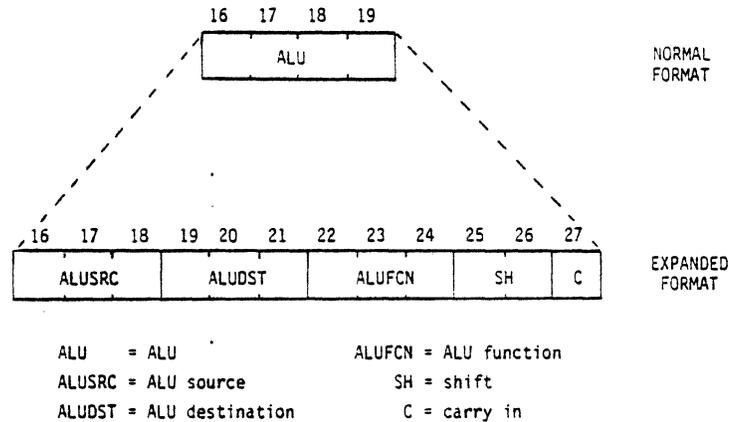
ADD 3,2; SETMAR; OUT

With the OUT instruction added, DMA transfer is first made by the SETMAR and then data is transferred to the external device by the OUT instruction. However, there are timing constraints that must be followed because the data must be in the FIFO output buffer at the time of the OUT instruction (refer to paragraph 4.2.5, Timing Considerations).

### 4.3 USING THE ALU

The arithmetic and logic unit (ALU) in the PIOP not only performs the arithmetic and logical operations required by the PIOP, but also contains 17 registers that can be accessed by the program.

Operations performed by the ALU may be specified in either the normal format or in the expanded format as shown in Figure 4-9.



0066

Figure 4-9 ALU Instruction Formats

In the normal format, the four bits making up the ALU field can be used to select 1 of 16 arithmetic or logical operations. In the expanded format, the ALU field is expanded into 12 bits which make up the ALUSRC, ALUDST, ALUFCN, SH, and C fields.

When using the normal format, fewer operations can be used than in the expanded format but less instruction word bits are used. The bits that are thus saved are used to make up the data bus source (SRC) and data bus destination (DST) fields so that data bus transfer fields can be implemented in the same cycle as the ALU operation.

When using the normal format, the 4-bit ALU field is used to specify the operation to be performed. The 16 instructions that can be selected by this ALU field are listed in Table 4-4.

Table 4-4 ALU Instructions

| Octal Code | Mnemonic | Meaning       | Octal Code | Mnemonic | Meaning                    |
|------------|----------|---------------|------------|----------|----------------------------|
| 0          | NOP      | no operation  | 10         | INCB     | increment register B       |
| 1          | MOVD     | move data     | 11         | DECB     | decrement register B       |
| 2          | ADD      | add data      | 12         | INCD     | increment data bus         |
| 3          | ANDD     | logical "and" | 13         | DECD     | decrement data bus         |
| 4          | ORD      | logical "or"  | 14         | ADD      | add reg. A to reg. B       |
| 5          | XORD     | logical "xor" | 15         | SUB      | subtract reg.A from reg. B |
| 6          | PASSD    | pass data     | 16         | PASSB    | pass register B            |
| 7          | PASSA    | pass reg. A   | 17         | PASSQ    | pass register Q            |

0067

A complete description of each of the above 16 instructions is given in Chapter 2 of this manual which describes the PIOP instruction set.

When using the expanded ALU format, it is necessary for the programmer to have a basic understanding of the ALU logic. A simplified block diagram of the ALU is illustrated in Figure 4-10 and described in the following paragraphs.

Each of the 16 addressable RAM registers shown on the drawing can be selected by either the A or B address input which corresponds to the A and B fields of the PIOP basic instruction word. The contents of the selected register are gated to the ALU data source selector. This selector, which functions as a multiplexer, receives inputs from: the ALU registers, the Q register, the direct data in line (which comes from the PIOP's data bus), and logic that inputs all 0's to the selector.

The information that is to be gated from the ALU data source selector is determined by instructions in the ALUSRC (ALU source) field of the expanded ALU format. Two outputs (R and S) are selected and applied to the R and S inputs of the ALU itself. Both of these ALU inputs are selected by a single instruction. For example, the ALUSRC field instruction "AQ" indicates that the register A contents are applied to the ALU R input and that the Q register contents are applied to the ALU S input.

Once information is loaded into the ALU, the particular function performed is determined by instructions in the ALUFCN (ALU function) field. These instructions include arithmetic operations such as ADD and SUB and logical operations such as AND and OR.

The ALU output (which is labelled "F" for function) may be applied to the output data selector, or back to the Q register, or back to the RAM shift logic. The output data selector logic selects either the function (F) from the ALU or the output of the A register as an output (labelled "Y"). The ALUDST field selects the appropriate function that is to be applied to the Y output of the output data selector.

This Y output of the ALU logic can be selected as the source of the data bus by using the ALU instruction in the DBSRC field of the basic instruction word.

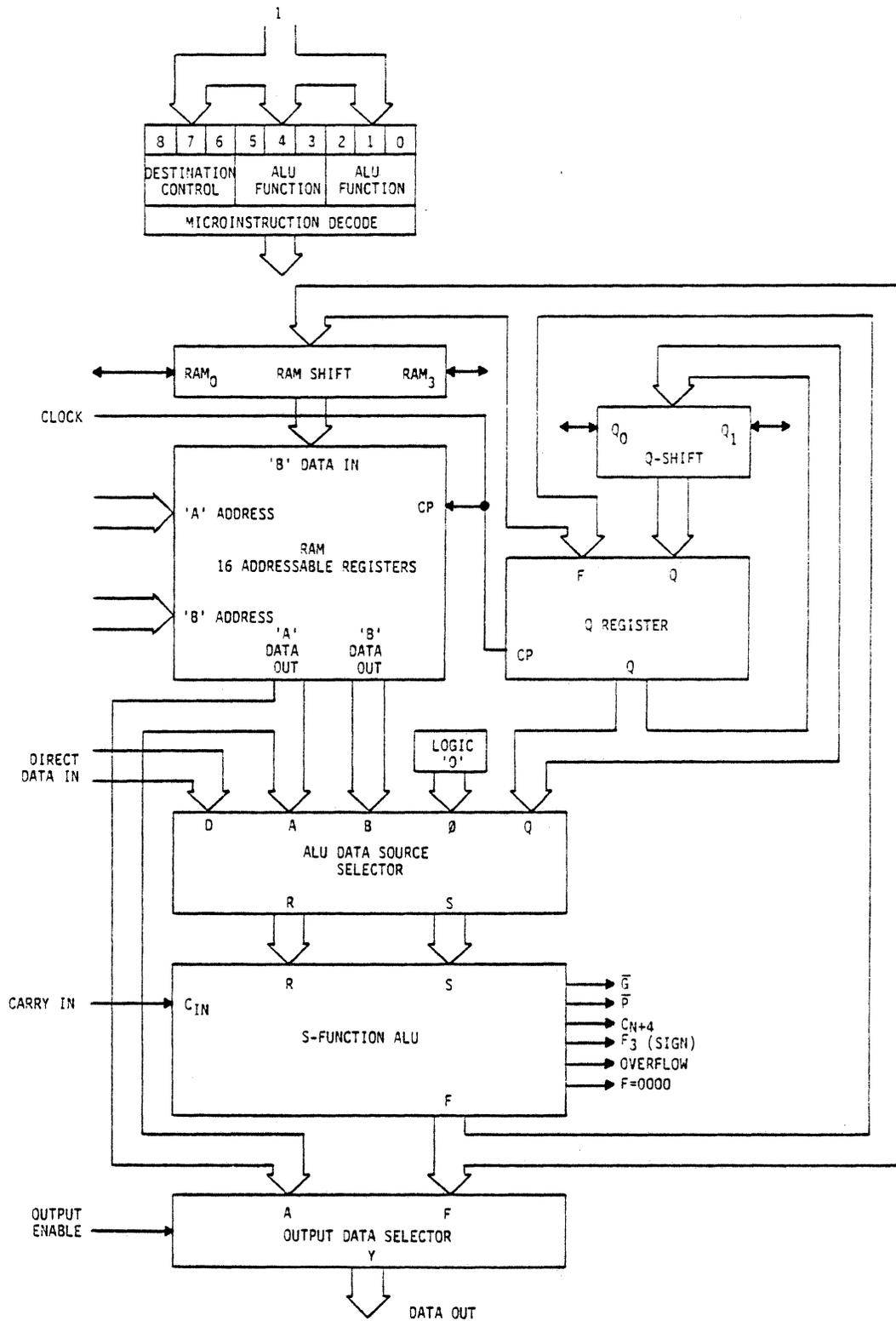


Figure 4-10 ALU Logic - Block Diagram

0068

Subsequent descriptions of ALU programming use letter designations to represent the ALU registers, inputs, outputs, and the data bus. These letters, along with a brief description, are listed in Table 4-5 below.

Table 4-5 ALU Designations

| SYMBOL  | DESCRIPTION   | REMARKS   |
|---------|---|---|
| A       | Register A - one of 16 internal registers. The specific register to be used is specified by a 4-bit binary number in the A field.   | Source or destination address for ALU operands or results |
| B       | Register B address - one of 16 internal registers. The specific register to be used is specified by a 4-bit binary number in the B field.<br><br><u>NOTE</u><br><br>The same 16 registers are used by both A and B fields. For example, the A field may specify register #2 while the B field may specify register #14. |   |
| D or DB | Data Bus - the bi-directional bus connecting the transceiver to the other PIOP circuits. The mnemonic DB is also used for data bus.   | -   |
| Q       | Register Q - an internal work register.   | -   |
| R       | ALU Input Register R - one of two inputs to the ALU. Designates the left-hand input in a double-operand statement.  | These are outputs of the ALU operand multiplexer          |
| S       | ALU Input Register S - One of two inputs to the ALU. Designates the right-hand input in a double-operand statement.   |   |
| Y       | ALU Output Bus Y - indicates the output bus of the ALU. More specifically, the output of the ALU Bus Select Logic.  | -   |
| Z       | Represents binary 0's. For example, the expression $Z > R$ indicates that all zeros are loaded into the ALU R input register.   | -   |
| F       | Results of the ALU function which are applied to the ALU destination.   | -   |

0069

A description of the instructions that are used when the expanded ALU format has been selected is given in the following paragraphs. These paragraphs cover the ALU source (inputs to the ALU), destination (information used as ALU output), and function (selected operation of the ALU).

#### 4.3.1 ALU SOURCE (INPUTS)

The ALUSRC (ALU source) field contains a double-operand instruction that specifies which two inputs are to be applied to the ALU. The five possible inputs are:

A = register A  
B = register B  
D = data bus  
Z = binary 0's  
Q = register Q

The first letter in the 2-letter statement (double-operand statement) represents the R input to the ALU while the second letter represents the S input to the ALU. For example:

ZQ = binary 0's applied to R input;  
      contents of Q register applied to S input

It is important to remember that not all possible 2-letter combinations are valid. For instance, the combination "AZ" is not a valid instruction although the combination "ZA" is a valid instruction.

The permissible operand combinations are:

AQ  
AB  
ZQ  
ZB  
ZA  
DA  
DQ  
DZ

A complete description of each of the above combinations is given in the instruction set explanation (refer to Chapter 2).

#### 4.3.2 ALU DESTINATION (OUTPUT)

The ALUDST (ALU destination) field contains one of eight instructions that specifies where the output of the ALU is to be set. The ALU output is labelled "F" (for ALU function).

The eight instructions that can be selected in the ALUDST field are as follows:

$Q = F \rightarrow Q, F \rightarrow Y$   
 $NP = F \rightarrow Y$   
 $A = F \rightarrow B, A \rightarrow Y$   
 $F = F \rightarrow B, F \rightarrow Y$   
 $RQ =$  Right shift Q and F, depending on SH (shift) field with  $F \rightarrow Y$ . For example:  
 $F/2 \rightarrow B, Q/2 \rightarrow Q, F \rightarrow Y$   
 $RF =$  Right shift F, depending on SH (shift) field with  $B \rightarrow Y$ . For example:  
 $F/2 \rightarrow B, F \rightarrow Y$   
 $LQ =$  Left shift F and Q, depending on SH (shift) field with  $B \rightarrow Y$ . For example:  
 $2F \rightarrow B, 2Q \rightarrow Q, F \rightarrow Y$   
 $LF =$  Left shift F, depending on SH (shift) field with  $F \rightarrow Y$ . For example:  
 $2F \rightarrow B, F \rightarrow Y$

Note that the selected destination for the above instructions is one of the following: the Q register (internal work register), Y (the ALU output bus), or the B register (one of 16 internal ALU registers selected by the REG B field in the instruction word).

#### 4.3.3 ALU FUNCTION

The ALUFCN (ALU function) field contains one of eight instructions that specifies which arithmetic or logical operation the ALU is to perform. The result of the selected function is labelled "F." Thus, when a particular operation is selected, the result of that operation (function) is sent to the destination specified by the ALUDST field.

The eight functions that can be selected by the ALUFCN field are listed below. Note that "R" and "S" indicate the two inputs to the ALU and "C" indicates the carry bit as specified by the C (carry) field.

|                   |                                      |
|-------------------|--------------------------------------|
| add               | $F = R + S + C$                      |
| subtract          | $F = R - S - \text{NOT } C$          |
| subtract, reverse | $F = S - R - \text{NOT } C$          |
| logical "or"      | $F = R \text{ "or" } S$              |
| logical "and"     | $F = R \text{ "and" } S$             |
| logical "nand"    | $F = \text{NOT } R \text{ "and" } S$ |
| exclusive "or"    | $F = R \text{ "xor" } S$             |
| exclusive "nor"   | $F = \text{NOT } R \text{ "xor" } S$ |

A complete description of each of the above instructions, including appropriate octal codes, is given in the instruction set explanation (refer to Chapter 2).

#### 4.3.4 USING ALU INSTRUCTIONS

One of the advantages of the expanded format is that a number of different ALU operations can be executed in a single cycle. For example, the ALU source, destination, and function can all be selected by the expanded format and then executed in a single instruction word cycle.

The available ALU instructions are summarized in Table 4-6.

Table 4-6 Summary of ALU Instructions

| OCTAL CODE | FIELDS IN EXPANDED FORMAT |        |        |       |       |
|------------|---------------------------|--------|--------|-------|-------|
|            | ALUSRC                    | ALUDST | ALUFCN | SHIFT | CARRY |
| 0          | AQ                        | Q      | AD     | -     | -     |
| 1          | AB                        | NP     | SB     | N     | I     |
| 2          | ZQ                        | A      | SR     | R     | -     |
| 3          | ZB                        | F      | OR     | A     | -     |
| 4          | ZA                        | RQ     | AN     | -     | -     |
| 5          | DA                        | RF     | NA     | -     | -     |
| 6          | DQ                        | LQ     | XO     | -     | -     |
| 7          | DZ                        | LF     | XN     | -     | -     |

0070

When using this expanded format, the function field (ALUFCN) represents the basic op code and the shift and increment fields (SHIFT and C) are considered extensions of this code.

Either two, three, or four arguments may be used with the basic op code. If two arguments are used, they are always source (ALUSRC) and destination (ALUDST). If three arguments are used, then the third argument is either register A or register B (REG A or REG B). If four arguments are used, then the third argument is always register A (REG A) and the fourth argument is always register B (REG B).

The function, extensions, and arguments must always be expressed in the following order:

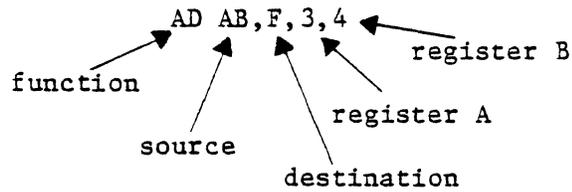
function-increment-shift      source,destination,register A,register B

In other words, the instruction fields must be defined in this order:

ALUFCN-SHIFT-C      ALUSRC, ALUDST, REG A, REG B

When defining REG A and REG B, they should always be reduced to a number from 0 to 15 in order to specify the particular internal register that is used.

The following is an example of a combined ALU instruction with four arguments:



The above instruction indicates that:

AD = an addition is to be performed

AB = the two sources for the ALU (the values to be added) are the contents of register A and the contents of register B

F = the destination of the result is the ALU output bus and register B (in other words:  $A + B \rightarrow B$ )

3 = register A is ALU internal register number 3

4 = register B is ALU internal register number 4

Note that no extensions (shift or increment) were used in the above example.

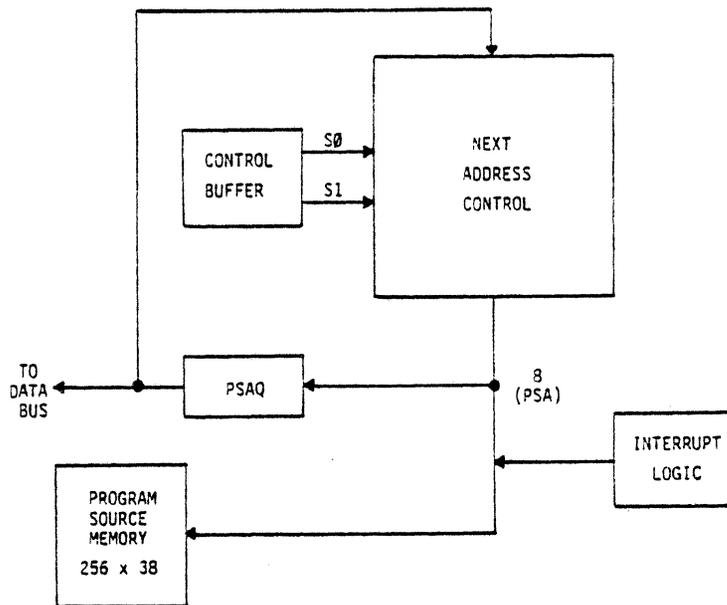
Some other examples of ALU combined instructions are:

- |              |  |
|--------------|--|
| AD ZQ,NP     | basic op code plus two arguments (source and destination); no extensions   |
| AD ZB,LQ 5   | three arguments (source, destination, register); the register number (5) can be specified in either the REG A or REG B field |
| ADN AB,RF,7  | basic op code with one extension (N); extension indicates to shift in all 1's  |
| ADI AB,F,8,9 | basic op code with one extension (increment) and four arguments (source, destination, register A, register B)                |
| ADIR DQ,Q    | basic op code with two extensions (I and R); I indicates an increment and R indicates a rotate                               |
| OR ZA,F,2,3  | move register 2 to register 3  |

#### 4.4 USING PROGRAM SOURCE MEMORY

The program source memory in the PIOP is a writable control store that holds the program instructions that are to be executed. A program counter, referred to as the "program source address," determines which instruction is to be executed next. The contents of this program counter can be changed by the branch, jump, and subroutine instructions in the PSA CONTROL field of the PIOP instruction word.

Figure 4-11 is a simplified block diagram of the logic that controls the program source memory. When an instruction word is decoded, the instruction register (which performs the decoding) sends the appropriate signals to the next address control logic. Based on these signals, this logic then performs one of two operations: increments the current address to point to the next sequential instruction, or computes a completely new address for the next instruction if required because some type of jump or branch instruction has been decoded.



0071

Figure 4-11 Program Source Address Logic

After the appropriate address has been generated by the next address control logic, the address is applied to both the program source memory and the PSAQ register. Once the program source memory has been addressed, then the selected instruction word is applied to the control buffer for decoding.

Notice that the output of the PSAQ register is fed back to the next address control logic. This line is used only when stepping through sequential memory locations. The current address is fed back to the address control logic, incremented, and then used for the next address.

If a computed jump or branch address is required, or if an interrupt occurs, then the address control logic uses different input data to compute the next address.

It is necessary to remember the difference between the terms "PSA" and "PSAQ." The term PSAQ is used for the address of the currently executing instruction while PSA is used for the address of the next instruction.

#### NOTE

The term "PSA" refers to a condition while the term "PSAQ" refers to a register that can be loaded and read. Unless otherwise specified, both this manual and the assembler use both terms to indicate the register (PSAQ).

Figure 4-12 is a block diagram of the next address control logic. This logic contains the address register (AR) which is normally loaded from the PSAQ. However, this register can also be loaded from the data bus by using the AR instruction in the DST field.

The figure also shows the 4 x 4 subroutine stack and related stack pointer which are used with the JSR, RTN, PUSH, and POP instructions in the PSA CONTROL field of the instruction word.



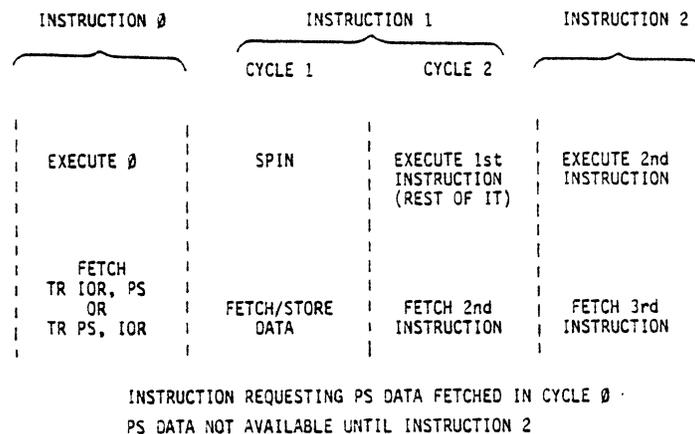
The multiplexer shown in Figure 4-12 selects one of four inputs as the output to be applied to the program source memory. The D input (direct inputs) is the displacement value required for jump and branch instructions. The displacement is selected by either the DISP5 or DISP8 fields. The AR input represents the contents of the address register (which contains the current address). The F input is the word from the top of the stack. The uPC (microprogram) input represents the incremented address necessary when addressing sequential memory locations.

The output of the multiplexer is applied to output control elements for gating to the program source memory and to the PSAQ register. If sequential addressing is being used, then the output is also fed to the incrementer.

Subsequent paragraphs cover the special instructions that use program source memory and the branch and jump instructions.

#### 4.4.1 INSTRUCTIONS THAT USE PROGRAM SOURCE MEMORY

There are two instructions in the SPEC (special) field that can use the program source (PS) memory. The input/output register (IOR) is the only element that can be used as a source or destination for these PS operations. The two instructions are: TR PS, IOR and TR IOR, PS. Both of these are 2-cycle instructions. These cycles are shown in Figure 4-13.



0073

Figure 4-13 TR PS, IOR Instruction Cycles

As shown in the figure, the TR PS, IOR (or TR IOR, PS) portion of the instruction is executed during the first cycle. The rest of the instruction is executed in the second cycle. There is no data on the data bus during the second cycle. The IOR can be gated to the data bus in the next instruction for use in that instruction.

The address of the operation is forced by the PSA CONTROL field. For example:

```
TR IOR,PS; JMPA 15.
```

The above instruction uses program source (PS) location 15. to store the contents of the IOR. Note that the program counter is always incremented during a PS fetch/store instruction.

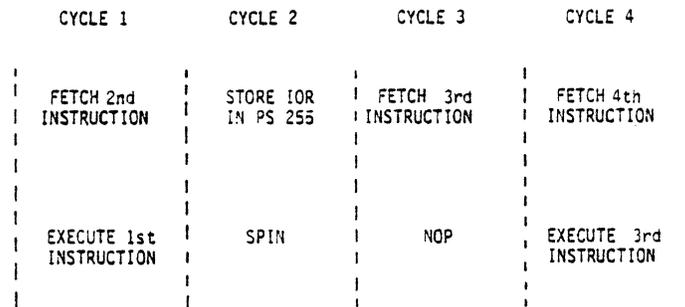
As an example of how to use the TR IOR,PS instruction correctly, assume that it is desired to store the sum of ALU registers 0 and 1 in program source memory location 255 and then return. This could be done by using the following program:

```
ADD 0,1; TR ALU, IOR      "Add the contents of registers 0
                          "and 1; move the result to the
                          "I/O register

TR IOR,PS; JMPA 255      "Transfer the contents of the I/O
                          "register to program source memory
                          "location 255

RTN                       "Return from subroutine
```

The above program takes four PIOP cycles for execution. The four required cycles are shown in Figure 4-14.



0074

Figure 4-14 Program Cycles

As shown in Figure 4-14, the first instruction (ADD 0,1; TR ALU,IOR) is executed during cycle 1. The second instruction (TR IOR,PS; JMPA 255) is fetched during this cycle. During the second cycle, the contents of the IOR are stored in PS location 255. The third instruction (RTN) is not fetched until the third cycle and is not executed until the fourth cycle.

As an example of how to use the TR PS,IOR instruction correctly, assume that it is desired to read data from the program source memory at the absolute address contained in the address register (AR) and then store this data in ALU register 0. This could be done by using the following program:

```
TR PS,IOR; JMPAR          "Transfer the contents of the program
                          "source memory location specified by
                          "the address register to the I/O
                          "register

TR IOR,DB; MOVD 0; RTN    "Transfer the contents of the I/O
                          "register to the data bus; move data
                          "on data bus to ALU register 0;
                          "return from subroutine
```

The above program is executed in three cycles. During the first cycle, the first instruction is fetched. During the second cycle, the first instruction is executed and the second instruction is fetched. During the third cycle, the second instruction is executed.

The following is an example of incorrect usage of the program source instruction:

```
TR PS,IOR; MOVD 0
RTN
```

In the above example, the first instruction indicates that data from program source memory is to be transferred to the I/O register and that data on the data bus is to be moved into ALU register 0.

There are two problems with the above program. First of all, the desired data ends up in the IOR in the first cycle and the MOVD 0 instruction executes in the second cycle. Therefore, moving data from the data bus into register 0 causes invalid data to be moved into the register because the IOR is not gated on to the data bus automatically.

The second problem is that no address was forced by the PSA CONTROL field; therefore, the data accessed is the RTN instruction.

#### 4.4.2 BRANCH AND JUMP INSTRUCTIONS

The PSA CONTROL field in the PIOP instruction word can be used to select 1 of 15 instructions. There are four unconditional jump instructions, seven conditional branch instructions (plus 16 extended versions), and four instructions that manipulate the subroutine return stack. Jumps may be made to relative or absolute addresses while all conditional branches are made to relative addresses.

The subroutine return stack provides return address linkage when executing subroutines. These two instructions are JSR (jump to subroutine) and RTN (return from subroutine). Because the stack is a 4-word stack, up to four subroutines can be nested. The programmer should always end every subroutine with a RTN instruction.

Two other instructions, PUSH and POP, are also used with the subroutine return stack. These instructions, as well as the JSR and RTN instructions, are briefly described in Table 4-7 below.

Table 4-7 Stack-Related Instructions

| INSTRUCTION | DESCRIPTION  |
|-------------|--|
| JSR         | Branches to the location specified by the contents of the DISPB field (relative to the current location). The current PSA + 1 is pushed on to the stack. |
| RTN         | Branches to the address at the top of of the stack and advances the pointer.   |
| PUSH        | Forces the address of the next sequential instruction on to the stack.   |
| POP         | Advances the stack pointer, thereby discarding the value at the top of the stack.  |

0075

When using branch instructions, the instruction first tests some condition and then branches according to the results of the test. The BIT # field selects the number of condition to be tested while the instruction itself determines the type of condition. For example, the BFS instruction (branch if flag set) tests a flag. The flag number is specified by the value in the BIT # field. The possible items that can be specified by the BIT # field are listed in Table 4-8.

Table 4-8 BIT # FIELD

| BIT # FIELD | DESCRIPTION  |
|-------------|--|
| FLAG #      | Specifies the number of the flag to be tested. Any one of eight flags (0 - 7) can be selected.   |
| DS #        | Specifies the external device status line to be tested. Any one of eight lines can be selected.  |
| STATUS      | Specifies the internal status bit to be tested. Any one of the following eight bits can be selected. <ul style="list-style-type: none"> <li>0 = FIFO data valid</li> <li>1 = FIFO full</li> <li>2 = R-shift out</li> <li>3 = Q-shift out</li> <li>4 = ALU carry</li> <li>5 = ALU zero</li> <li>6 = ALU sign</li> <li>7 = ALU overflow</li> </ul> |

0076

A list of all standard branch instructions, along with a brief description of each instruction, is given in Table 4-9. The 16 extended instructions are presented in Table 4-10.

Table 4-9 Branch Instructions

| TYPE        | INSTRUCTION | MEANING                          | DESCRIPTION   |
|-------------|-------------|----------------------------------|---|
| JUMP        | JMP         | relative jump                    | Causes an unconditional jump to a relative location. Jumps can be made in either direction and can be up to 256 locations from the current location. The effective address is the sum of the 8-bit DISP8 field and the current address.   |
|             | JMPA        | absolute jump                    | Causes an unconditional jump to an absolute address. This address is the contents of the DISP8 field.   |
|             | JMPAR       | absolute jump                    | Causes an unconditional jump to an absolute address. This address is the contents of the address register (AR).   |
|             | JMPST       | jump to stack                    | Causes an unconditional jump to the top of the stack.   |
| BRANCH      | BDSC        | branch if device status is clear | <p>All branch instructions cause conditional jumps to a relative location. This location is the sum of the contents of the 5-bit biased DISP5 field and the current address.</p> <p>The maximum number of locations that can be used are 16 locations forward (+16) or 17 locations backward (-17) from the current address.</p> <p>When testing a register bit for a branch condition, the bit is specified in the BIT # field of the PIOP instruction word.</p> |
|             | BDSS        | branch if device status is set   |   |
|             | BFC         | branch if flag is clear          |   |
|             | BFS         | branch if flag is set            |   |
|             | BISC        | branch if ALU status is clear    |   |
|             | BISS        | branch if ALU status is set      |   |
|             | BNZST       | branch if ALU is not zero        | Conditional branch to the top of the stack. In effect, this is a conditional JMPST instruction.   |
| SUB-ROUTINE | JSR         | jump to subroutine               | The current program counter (PSA) is pushed on to the stack and the jump is then made to a relative location which is the sum of the DISP8 field and the current address.   |
|             | RTN         | return from subroutine           | The stack is popped (the former program counter is retrieved) and the program jumps to the address specified by the contents of the word just popped. In effect, this is a POP and JMPST instruction.   |
| STACK       | PUSH        | push the stack                   | No jump. The program counter (PSA) +1 is pushed onto the top of the stack.  |
|             | POP         | pop the stack                    | No jump. Retrieves the contents from the top of the stack.  |

0077

Table 4-10 Extended Branch Instructions

| BASIC BRANCH    | VARIATION  | DESCRIPTION                          |
|-----------------|------------|--------------------------------------|
| BISS variations | BFV DISP   | Branch if FIFO data valid            |
|                 | BFF DISP   | Branch if FIFO full                  |
|                 | BFOT DISP  | Branch if R-shift output = 1         |
|                 | BQOT DISP  | Branch if Q-shift output = 1         |
|                 | BC DISP    | Branch if carry set                  |
|                 | BZ DISP    | Branch if ALU=0                      |
|                 | BM DISP    | Branch if ALU is minus               |
|                 | BOVF DISP  | Branch if overflow = 1               |
| BISC variations | BNFV DISP  | Branch if FIFO data <u>not</u> valid |
|                 | BNFF DISP  | Branch if FIFO <u>not</u> full       |
|                 | BNFOT DISP | Branch if R-shift output = 0         |
|                 | BNQOT DISP | Branch if Q-shift output = 0         |
|                 | BNC DISP   | Branch if ALU carry out is 0         |
|                 | BNZ DISP   | Branch if ALU is not 0               |
|                 | BP DISP    | Branch if ALU is positive            |
|                 |            | BNOVF DISP                           |

0078

#### 4.5 INTERRUPT HANDLING

The PIOP contains four interrupt lines that are connected to the external device (INT0\* through INT3\*). Interrupt receivers are "armed" by the control register (CR) to activate that particular input line.

In general, interrupt lines are enabled or disabled depending on whether or not the programmer wants to be interrupted at that particular time. Armed interrupts are queued if the interrupt is disabled. Interrupts are activated by a high-to-low transition on the associated interrupt line.

The four interrupt lines (INT0\* through INT3\*) trap to program source memory locations 0 through 3, respectively. These traps occur only if the interrupts are armed and enabled. Interrupt 0 is the highest priority interrupt and interrupt 3 is the lowest priority interrupt. Only interrupt 0 can interrupt spins.

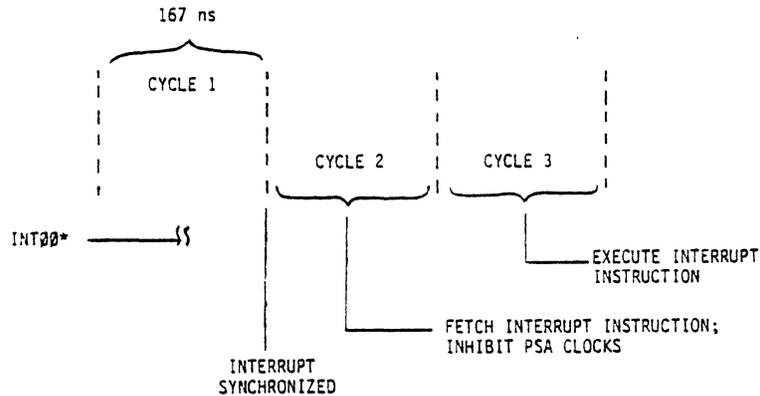
Interrupts are armed/disabled by setting/clearing bits 20 through 23 in the control register. Interrupts are enabled/disabled under program control. There are three instructions related to interrupt handling. These instructions are:

|        |   |
|--------|---|
| SINT   | set interrupt as specified by BIT # field (test purposes) |
| ENINT  | enable the interrupts                                     |
| DISINT | disable the interrupts                                    |

The AP instruction SNSA with device address (DA) 102 causes interrupt 3.

When an interrupt is received, the instruction at the appropriate interrupt address is fetched and executed and then normal program execution continues from the point where it was interrupted. If a multiple instruction service routine (interrupt handling subroutine) is required, a JSR instruction in the interrupt location saves the correct return address in the subroutine return stack. The return address that was saved is the location after the last instruction executed before the interrupt. Note that the saved address is not the location after the interrupt instruction.

The interrupt timing shown in Figure 4-15 below assumes that interrupts have been armed and disabled.



0079

Figure 4-15 Interrupt Timing

In summary:

- a. Individual interrupts are armed by bits 20 through 23 the PIOP control register (CR). Interrupts occurring while disarmed are lost.
- b. Interrupts are enabled by the ENINT instruction. The interrupts remain enabled until disabled (DISINT instruction). Interrupts occurring while disabled are executed normally when enabled, provided they have been armed previously.
- c. The interrupt return address is saved by executing a JSR instruction in the interrupt location. This JSR saves the location which would have been executed next had there been no interrupt.

#### 4.6 COMMUNICATING WITH THE AP

When the AP communicates with the PIOP, the AP first loads a value into its device register (DA). This value represents the address of the external device that is to communicate with the AP. Table 4-11 below lists the device addresses used to communicate with the PIOP, the instructions that can be used with these addresses, and the function of the instructions.

Table 4-11 AP Device Instructions

| ADDRESS IN AP<br>DEVICE ADDRESS (DA)<br>REGISTER | AP<br>INSTRUCTION         | FUNCTION  |
|--|---------------------------|---|
| 100 (IOR)  | IN<br>OUT<br>SNSA<br>SNSB | gates IOR to IOBUS<br>gates IOR from IOBUS<br>deposits IOR into PIOP control buffer<br>executes instruction in control<br>buffer. Instruction remains in<br>control buffer. PSA increments      |
| 101  | SNSA<br>SNSB<br>OUT       | 0 - PIOP running<br>1 - PIOP not running<br>resets PIOP<br>sets interrupt 3 (AP interrupt of<br>PIOP - lowest priority interrupt)<br><u>NOTE</u><br>BUSY gated to IODROY<br>in both cases above |
| 110 (FLAG 7)                                     | -                         | -   |
| 111 (FLAG 6)                                     | SNSA                      | reads specified flag  |
| 112 (FLAG 5)                                     | SNSB                      | clears specified flag   |
| 113 (FLAG 4)                                     | OUT                       | sets specified flag   |
| 114 (FLAG 3)                                     | -                         | -   |
| 115 (FLAG 2)                                     | -                         | -   |
| 116 (FLAG 1)                                     | -                         | -   |
| 117 (FLAG 0)                                     | -                         | -   |

0080

Two examples of programs that communicate between the AP and the PIOP are presented in the following paragraphs.

Example 1

In this example, it is assumed that main data memory location 10 contains the following instruction: PASSB 0; TR ALU, IOR.

|                     |   |
|---------------------|---|
| LDDA; DB=100        | "Load DA to access IOR                  |
| LDMA; DB=10         | "Get instruction                        |
| NOP                 | "Wait                                   |
| NOP                 | "Wait                                   |
| OUT; DB=MD          | "Load PIOP IOR                          |
| SNSA                | "Deposit instruction                    |
| SNSB                | "Execute instruction                    |
| IN; DPX<DB; DB=INBS | "Read IOR which contains ALU register 0 |

If a START instruction is executed, execution begins at PSA. The following instruction sequence is illegal:

SETMA or INCMA or DECMA or LDMA  
SNSB (in the next instruction)

Example 2

```
"EXAMPLE:      CALL PPLOAD(100,0,50)
"              STORES INTO PIOP PROGRAM SOURCE MEMORY LOCATIONS
"              0,1,.....,48,49 THE CONTENTS OF AP MAIN DATA
"              LOCATIONS 100,101,....,148,149.
"
"ENTER WITH FOLLOWING S-PAD PARAMETERS:
"  NAME          NUMBER
"      APA      $EQU 0          "BASE ADDRESS IN AP120B MAIN DATA MEMORY
"      PPA      $EQU 1          "BASE ADDRESS IN PIOP PROGRAM SOURCE MEMORY
"      N        $EQU 2          "WORD COUNT
"LOCAL VARIABLE:
"      TMP1     $EQU 0          "TEMPORARY
"
PPLOAD: JSR PPWAIT              "BE SURE PIOP NOT RUNNING
        MOV APA,APA;SETMA      "GET FIRST MD WORD
        LDSPI TMP1;DB=377      "8 BIT MASK
        DEC PPA;                "BACK UP ADDR BECAUSE OF PIOP H/W
        DPX<0                  "CLEAR DPX(0)
        AND TMP1,PPA           "MASK ADDRESS TO 8 BITS
        LDSPI TMP1;DB=1400     "CODE FOR 'JMPA' COMMAND
        OR TMP1,PPA;DPX<DB;DB=SPFN;WRITMAN "FORM 'JMPA PPA' COMMAND
        LDSPNL TMP1;RPSA;BR .+2 "GET PSA OF THIS INSTRUCTION
        "BRANCH AROUND NEXT INSTRUCTION
        $VAL 0,0,16,40000     "THIS IS A PIOP 'TR IOR,PS' INSTRUCTION
        INC TMP1;SETTMA       "SETUP TMA TO OUTPUT INSTRUCTION LATER
        RDA;LDSPNL TMP1      "SAVE DA
        LDDA;DB=100          "DA TO PIOP I/O REGISTER
        OUT;DB=DPX           "SET IOR TO 'JMPA PPA' COMMAND
        SNSA                 "DEPOSIT COMMAND INTO PIOP CTL BUF
        SNSB                 "EXECUTE 'JMPA PPA' COMMAND
        RPSFT;OUT            "SET IOR TO 'TR IOR,PS' COMMAND
        SNSA                 "DEPOSIT COMMAND INTO PIOP CTL BUF
        INCMA                "GET SECOND MD WORD
        OUT;DB=MD            "FIRST PIOP WORD TO IOR
LOOP:   INCMA;                "1. GET NEXT MD WORD
        SNSB;                " EXECUTE 'TR IOR,PS'
        " PIOP ADDR INCREMENTS ALSO
        DEC N                " DECREMENT COUNT
        OUT; DB=MD;          "2. SET IOR TO MD WORD
        BGT LOOP            " LOOP UNTIL DONE
        RETURN;LDDA;DB=SPFN;MOV TMP1,TMP1 "RESTORE DA AND EXIT
        $END
```

## CHAPTER 5

### ASSEMBLER

#### 5.1 INTRODUCTION

This chapter describes how to use the PIOP assembler for writing programs and is divided into three major parts:

|                     |   |
|---------------------|---|
| basics              | describes the instruction format, constants, symbols, expressions, pseudo ops, and op codes |
| writing programs    | describes comments, concatenation, labels, and errors                                       |
| using the assembler | describes the assembler loading procedure and also includes sample files                    |

## 5.2 THE BASICS

The PIOP assembler is referred to as PPAL (PIOP Program Assembly Language). This assembler is written in Fortran IV and provides a powerful tool for developing programs for the PIOP. By using this assembler, instructions having many components can be easily encoded.

As an example, the following instruction has seven component instructions:

```
SETMAR;DISINT;XORD 0,1;TR FF,DB;WORD 1;JMPST;SINDS 6
```

The above instruction expresses the following:

- a. read from main data memory
- b. disable all interrupts
- c. "exclusive-or" register 0 with data on the data bus and store result in register 1
- d. transfer the contents of the FIFO internal buffer (IFFB) to the data bus
- e. format the FIFO transfer as a WORD 1 type
- f. jump to the location given by the top of the LIFO stack if the ALU output is not zero
- g. spin until device status bit is 6, then do an IN transfer

The remainder of this paragraph is devoted to a discussion of PPAL assembler basics and covers such items as: instruction format, constants, symbols, expressions, pseudo ops, and op codes.

## Instruction Format

An instruction consists of an op code and 0-4 operands. The op code is separated from the operands by one or more spaces; the operands are separated from each other by commas. There are no column restrictions.

Example:            ADD 1,2

## Constants

A constant is a decimal, octal, or hex integer. A number is octal by default (as in AP code), unless immediately followed by a point (.) to indicate decimal, or immediately preceded by an X and a zero (X0) to indicate hex. Note that this places a restriction on symbols and labels, i.e. they may not begin with "X0".

Example:            400     (=256)  
                    400.   (=400)  
                    X0400  (=1024)

## Symbols

Symbols may contain up to 6 alpha-numeric characters, beginning with a letter. Symbols represent 16-bit integers.

Example:            ADD REG1,REG2

## Expressions

Expressions consist of constants and/or symbols separated by the operators plus (+) and/or minus (-). They are evaluated left to right, and may begin with a minus. No parentheses are allowed. A period in the place of a constant or symbol indicates the current location counter. Expressions may be used wherever a constant may be used.

Example:            -400.+CHECK+IT-X020  
                    .+2

## PSEUDO OPS

### \$END

This must be at the end of each program.

### \$EQU

This equates a symbol to a 16-bit integer. The symbol precedes \$EQU by one or more spaces; an expression follows \$EQU by one or more spaces. Any symbol used in THIS expression must have been previously defined.

Example:               REG2 \$EQU 2  
                      HERE \$EQU .  
                      BIAS \$EQU 20.  
                      EXP \$EQU 12+BIAS

### \$LOC

This changes the current location counter to the value of the expression following \$LOC. The value should not exceed 255. A label may precede \$LOC (WITHOUT a colon).

Example:               HERE \$LOC 15  
                      THERE \$LOC HERE+10  
                      \$LOC .+1

### \$VAL

This uses one word of the PIOP program source memory, filling it directly with values provided by the programmer. A label may precede \$VAL (again, no colon with pseudo ops), and it must be followed by 3 consecutive expressions, separated by commas. The 3 expressions represent the 3 pieces of the 38-bit PS word: the top 10 bits, the middle 12 bits, and the bottom 16 bits.

Example:               \$VAL 0,LABEL-1,12.+X010  
                      LABEL \$VAL .+1,7777,-1

### \$SUB

This establishes the name of the Fortran subroutine output (see the section on output). It is not required when the Fortran output is not desired, and will be ignored if present. This pseudo-op can be put anywhere in the program prior to the \$END statement.

Example:               \$\$SUB TEST

## OP CODES

### Branch Field

|    |       |                    |                                 |
|----|-------|--------------------|---------------------------------|
| 0  | (nop) | -                  |                                 |
| 1  | JMPAR | -                  |                                 |
| 2  | JMPST | -                  |                                 |
| 3  | JMPA  | abs. loc.          |                                 |
| 4  | POP   | -                  |                                 |
| 5  | PUSH  | -                  |                                 |
| 6  | RTN   | -                  |                                 |
| 7  | JSR   | 8-bit disp.        |                                 |
| 8  | BDSC  | bit #, 5-bit disp. |                                 |
| 9  | BDSS  | bit #, 5-bit disp. |                                 |
| 10 | BFC   | bit #, 5-bit disp. |                                 |
| 11 | BFS   | bit #, 5-bit disp. |                                 |
| 12 | BISC  | bit #, 5-bit disp. | (see extended branch mnemonics) |
| 13 | BISS  | bit #, 5-bit disp. | ( " " " " )                     |
| 14 | BNZST | -                  |                                 |
| 15 | JMP   | 8-bit disp.        |                                 |

### Extended Branch Mnemonics

| Instruction | Represents  |                     |                                      |
|-------------|-------------|---------------------|--------------------------------------|
| BFV         | 5-bit disp. | BISS 0, 5-bit disp. | Branch if FIFO data valid            |
| BFF         | 5-bit disp. | BISS 1, 5-bit disp. | Branch if FIFO full                  |
| BFOT        | 5-bit disp. | BISS 2, 5-bit disp. | Branch if R-shift output = 1         |
| BQOT        | 5-bit disp. | BISS 3, 5-bit disp. | Branch if Q-shift output = 1         |
| BC          | 5-bit disp. | BISS 4, 5-bit disp. | Branch if carry set                  |
| BZ          | 5-bit disp. | BISS 5, 5-bit disp. | Branch if ALU = 0                    |
| BM          | 5-bit disp. | BISS 6, 5-bit disp. | Branch if ALU is minus               |
| BOVF        | 5-bit disp. | BISS 7, 5-bit disp. | Branch if overflow = 1               |
|             |             |                     |                                      |
| BNFV        | 5-bit disp. | BISC 0, 5-bit disp. | Branch if FIFO data <u>not</u> valid |
| BNFF        | 5-bit disp. | BISC 1, 5-bit disp. | Branch if FIFO <u>not</u> full       |
| BNFOT       | 5-bit disp. | BISC 2, 5-bit disp. | Branch if R-shift output = 0         |
| BNQOT       | 5-bit disp. | BISC 3, 5-bit disp. | Branch if Q-shift output = 0         |
| BNC         | 5-bit disp. | BISC 4, 5-bit disp. | Branch if ALU carry out is 0         |
| BNZ         | 5-bit disp. | BISC 5, 5-bit disp. | Branch if ALU is <u>not</u> zero     |
| BP          | 5-bit disp. | BISC 6, 5-bit disp. | Branch if ALU is positive            |
| BNOVF       | 5-bit disp. | BISC 7, 5-bit disp. | Branch if ALU overflow = 0           |

Branch op codes have 0-2 arguments. For branches with one argument, the argument is either an 8-bit absolute location or an 8-bit relative displacement, depending on the op code. For branches with 2 arguments, the first is an expression of value 0-7, representing a bit # or flag # (same field), and the second is a 5-bit relative displacement biased by 20 (octal).

```
Example:      RTN
              POP
              JSR SUBR
              JMPA 10
              BFS 3,CHOICE
              BISS 5,LABEL
              BZ LABEL      (same as preceding instruction)
```

#### Control Field

```
0 (nop) -
1 CF expression
2 RFF -
3 AFF -
4 SF expression
5 SINT expression
6 ENINT expression
7 DISINT expression
8
9 START -
10 HALT -
11 PSAB -
12 TVCR expression
13 TVDB expression
14 TVER expression
15 (set by use of expanded ALU ops)
```

For the op codes with one argument, the argument should be an expression reducing to either 0-7 (using the bit # field) or to a 20-bit positive number (using the value field).

```
Example:      START
              SF 3
              TVDB 40000.
              FLAG1 $EQU 1
              CF FLAG1
```

## Data Bus Transfer

|   | source | dest |
|---|--------|------|
| 0 | ALU    | DB   |
| 1 | (disp) | ER   |
| 2 | FF     | FF   |
| 3 | IOR    | IOR  |
| 4 | PSA    | AR   |
| 5 | TM     | TM   |
| 6 | CR     | CR   |
| 7 | *      | TMA  |

\* indicates use of the SPEC field

|   | spec       |
|---|------------|
| 0 | PS, IOR    |
| 1 | IOR, PS    |
| 2 | APMA, DB   |
| 3 | DVCMD, DB  |
| 4 | APMA, IOR  |
| 5 | DVCMD, IOR |
| 6 | ER, IOR    |
| 7 | TMA, IOR   |

These instructions all have the format   TR arg1,arg2   .  
Any single source can be combined with any single destination:  
TR src,dest   . The operand pairs must be in the given order.  
The 8-bit displacement field is used for   TR expression,dest   .

Example:           TR ALU,ER  
                  TR TM, TM  
                  TR 12, TMA  
                  TR APMA, DB  
                  TR ER, IOR  
                  SYM \$EQU X014  
                  TR SYM+5., DB

### I/O Fields

| I/O       | SDSC         | SDSS         | SDAV     |
|-----------|--------------|--------------|----------|
| 0 (nop)   | 0 (nop)      | 0 (nop)      | 0 (nop)  |
| 1 OUT -   | 1 SDSC bit # | 1 SDSS bit # | 1 SDAV - |
| 2 IN -    |              |              |          |
| 3 IORST - |              |              |          |

For the op codes requiring an argument, it should be an expression reducing to 0-7 (using the bit # field).

Example:           IN  
                  IORST  
                  SDSS 5  
                  BIT \$EQU 3  
                  SDSC BIT+1  
                  SDAV

### Word Field

0 WORD expression  
1 WORD expression  
2 WORD expression  
3 WORD expression

These take the form: WORD arg  
where "arg" is an expression reducing to 0-3.

Example:           WORD 2  
                  ONE \$EQU 1  
                  HERE \$LOC 2  
                  WORD ONE-.+3

### Addr Bus Field

0 (nop)  
1 SETMAR -  
2 SETMAW -  
3 SETDA -

There are no arguments.

Example:           SETMAR

Alu Macro Field

|    |       |     |
|----|-------|-----|
| 0  | (nop) | -   |
| 1  | MOVD  | B   |
| 2  | ADDD  | A,B |
| 3  | ANDD  | A,B |
| 4  | ORD   | A,B |
| 5  | XORD  | A,B |
| 6  | PASSD | -   |
| 7  | PASSA | A,B |
| 8  | INCB  | B   |
| 9  | DECB  | B   |
| 10 | INCD  | -   |
| 11 | DECD  | -   |
| 12 | ADD   | A,B |
| 13 | SUB   | A,B |
| 14 | PASSB | B   |
| 15 | PASSQ | -   |

For those with one argument, the argument should be an expression reducing to 0-15. For those with 2 arguments, both should be expressions reducing to 0-15, and A always precedes B.

Example:        INCD  
                 REG1 \$EQU 7  
                 REG2 \$EQU 5  
                 INCB REG2  
                 ADD REG1,REG2

## Extended ALU Field

|   | func | inc | shift | src | dest |
|---|------|-----|-------|-----|------|
| 0 | AD   | **  | *     | AQ  | Q    |
| 1 | SB   | I   | N     | AB  | NP   |
| 2 | SR   |     | R     | ZQ  | A    |
| 3 | OR   |     | A     | ZB  | F    |
| 4 | AN   |     |       | ZA  | RQ   |
| 5 | NA   |     |       | DA  | RF   |
| 6 | XO   |     |       | DQ  | LQ   |
| 7 | XN   |     |       | DZ  | LF   |

\* default shift zeroes

\*\* default no increment

The basic op codes may have one or 2 extensions (increment and shift, in that order if both are desired), and have 2-4 arguments. The first argument is src, the second is dest. These determine the necessity for 0-2 more arguments. The third argument, if there are only 3, represents A or B. If there are 4 arguments, they are expressed in the following order:

func-inc-shift src,dest,A,B

Both A and B should be expressions reducing to 0-15.

Example:

|      |               |
|------|---------------|
| AD   | ZQ, NP        |
| AD   | ZB, LQ, 5     |
| AD   | AB, F, 3, 4   |
| ADN  | ZB, RF, 7     |
| ADI  | AB, F, 8., 9. |
| ADIR | DQ, Q         |

### 5.3 WRITING PROGRAMS

#### Comments

A comment on a line is indicated by a double quote. Everything following the quote on the same line is considered a comment.

Example:           ADD 1,2 "this is a comment

#### Concatenation

As with AP code, semi-colons concatenate instructions within the same Program Source (PS) word. A semi-colon at the end of a line (preceding any comments) indicates that the following line is part of the same PS word. If the following line has only a comment, the line after that is still part of the same word.

Example:           PASSQ; POP; "comment 1  
                  "comment 2  
                  TR ALU,DB  
(This is all in the same PS word.)

#### Labels

A label is followed by a colon (except for pseudo ops). Another label and colon may follow, ad infinitum. Any labels must precede any code on the same LINE. A PS word which uses more than one line may have labels on each line.

Example:           LAB1: LAB2: PASSQ; "COMMENT  
                  LAB3:       POP  
(This is all in the same PS word.)

## Errors

Errors detected by Pass 1 of the assembler are printed preceding the regular listing in the following format:

```
LOCATION n *** error message ***
```

An error detected during Pass 2 is printed following the line in which it was detected.

Although an attempt by the programmer to use the same field more than once within a single PS word is normally an error, it is permitted in cases where the contents of that field is the same for all usages within the word.

Example:           CF 5; SDSS 5  
(Both instructions use the bit # field, but its contents would be the same for both anyway, so this is legal.)

Example:

```
PASS 1
LOCATION 0000 *** INCORRECT LABEL FORMAT ***
PASS 2
0000 000000 LA#: PASSQ
      000360
      000000

0001 000000      NOOP
      000000 *** ILLEGAL OP-CODE ***
      000000

0002 000002 $VAL 2,3
      000003 *** MISSING EXPRESSION ***
      000000

      $END

**** 3 ERRORS ****

SYMBOL  VALUE
```

## 5.4 USING THE ASSEMBLER

When started running, the assembler asks for 3 filenames (source file, load module file, and listing file, respectively), for a radix, for the type of listing, and for the type of load module desired (see below). The radix refers to the object code and location counter printed on the listing, and is available in octal, decimal, or hex. The listing may be a full listing or an error-only listing (for making small changes in big programs).

### OUTPUT: THE LOAD MODULE

There is no linker for the PIOP. All necessary subroutines must be assembled together. The regular load module outputted by the PIOP assembler is similar in format to that outputted by APLINK using the "E" command (i.e. output suitable for APSIM). The first number in the load module file is the number of locations to be loaded (in F4.0 format), up to 256. Each following line represents one 38-bit PS word, in 3 pieces (10- , 12- , and 16-bit) in 3F7.0 format.

The alternate format of the PIOP load module is a Fortran subroutine consisting mainly of data statements (similar to the "A" command output from APLINK). The PIOP code becomes a Fortran-callable, self-loading, optionally self-starting program when used in conjunction with the AP program PEXEC, which this Fortran subroutine calls. See Appendix D for an example.

The parameters of this Fortran subroutine are as follows:

MDADR - AP main data address at which the PIOP code will be loaded.  
PPSA - PIOP program source address at which the PIOP code will be loaded (if desired).  
FLAG - 0 through 3:  
    0 load into AP only  
    1 load into AP, thence into PIOP  
    2 load into AP and PIOP, and start PIOP running at beginning of loaded routine  
    3 load into AP and start channel program  
SIZE - number of PIOP words

The call would be:

```
CALL name(MDADR,PPSA,FLAG,SIZE)
```

SIZE is an output parameter; the rest are input parameters. PPSA is not used if FLAG= 0 or 3.

## 5.5 SAMPLE LISTINGS

The following are samples of: a source file, a listing file, a load module file, and a Fortran subroutine output.

### Sample Source File

```
$$SUB SAMPLE
"
LAB0: PASSQ;          "COMMENT
LAB1: WORD 2         "COMMENT
LAB2: SETMAR;        "COMMENT
LAB3: WORD 2;        "COMMENT
LAB4: PASSA 12.,1    "COMMENT
LAB5: TR FF,FF;      "COMMENT
LAB6: PASSA 12,10.;  "COMMENT
LAB7: IORST;         "COMMENT
LAB8: SETMAW         "COMMENT
      JMPA XOAB;      "COMMENT
      PASSQ           "COMMENT
SYM $EQU 10          "COMMENT
$LOC SYM             "COMMENT
MOVE $VAL .,2,SYM   "COMMENT
$END
```

Sample Listing File

PASS 1  
PASS 2

\$SUB SAMPLE

"

```
0000 000000 LAB0: PASSQ;          "COMMENT
      000360 LAB1: WORD 2         "COMMENT
      020000

0001 000414 LAB2: SETMAR;        "COMMENT
      000560 LAB3: WORD 2;        "COMMENT
      020000 LAB4: PASSA 12.,1    "COMMENT

0002 001012 LAB5: TR FF,FF;      "COMMENT
      005164 LAB6: PASSA 12,10.;  "COMMENT
      100030 LAB7: IORST;         "COMMENT
      LAB8: SETMAW                "COMMENT

0003 000000          JMPA XOAB;   "COMMENT
      000360          PASSQ        "COMMENT
      001653

      000010 SYM $EQU 10          "COMMENT

0010          $LOC SYM            "COMMENT

0010 000010 MOVE $VAL .,2,SYM    "COMMENT
      000002
      000010
```

\$END

\*\*\*\* 0 ERRORS \*\*\*\*

| SYMBOL | VALUE  |
|--------|--------|
| LAB0   | 000000 |
| LAB1   | 000000 |
| LAB2   | 000001 |
| LAB3   | 000001 |
| LAB4   | 000001 |
| LAB5   | 000002 |
| LAB6   | 000002 |
| LAB7   | 000002 |
| LAB8   | 000002 |
| SYM    | 000010 |
| MOVE   | 000010 |

Sample Load Module File

9.

|      |       |        |
|------|-------|--------|
| 0.   | 240.  | 8192.  |
| 268. | 368.  | 8192.  |
| 522. | 2676. | 32792. |
| 0.   | 240.  | 939.   |
| 0.   | 0.    | 0.     |
| 0.   | 0.    | 0.     |
| 0.   | 0.    | 0.     |
| 0.   | 0.    | 0.     |
| 8.   | 2.    | 8.     |

Sample Fortran Subroutine Output

```
      SUBROUTINE SAMPLE (MDADR,PPSA,FLAG,SIZE)
C
      INTEGER PPSA,FLAG,SIZE,PIECE(3)
      REAL CODE(9,3)
C
      DATA CODE(1,1),CODE(1,2),CODE(1,3)/0.,240.,8192./
      DATA CODE(2,1),CODE(2,2),CODE(2,3)/268.,368.,8192./
      DATA CODE(3,1),CODE(3,2),CODE(3,3)/522.,2676.,32792./
      DATA CODE(4,1),CODE(4,2),CODE(4,3)/0.,240.,939./
      DATA CODE(5,1),CODE(5,2),CODE(5,3)/0.,0.,0./
      DATA CODE(6,1),CODE(6,2),CODE(6,3)/0.,0.,0./
      DATA CODE(7,1),CODE(7,2),CODE(7,3)/0.,0.,0./
      DATA CODE(8,1),CODE(8,2),CODE(8,3)/0.,0.,0./
      DATA CODE(9,1),CODE(9,2),CODE(9,3)/8.,2.,8./
      M=MDADR-1
      SIZE=9
      DO 20 I=1,9
      DO 10 J=1,3
10      PIECE(J)=IPFIX(CODE(I,J))
      CALL APDEP(PIECE,14,M+I)
20      CONTINUE
      IF (FLAG.LE.0 .OR. FLAG.GT.3) RETURN
      CALL PEXEC (MDADR,PPSA,FLAG,SIZE)
      RETURN
      END
```

## CHAPTER 6

### PROGRAMMABLE I/O CHANNEL (PIOC)

#### 6.1 INTRODUCTION

The PIOP programmable I/O channel (PIOC) is a software construct that permits many I/O operations to be carried out by the PIOP without having to program the PIOP itself. The PIOC allows the AP to process in parallel with the PIOP and provides a means of communication between the PIOP and the AP in order to synchronize the processing where desired. The PIOC also provides for I/O operations between the AP main data memory and a device such as a disk controller interfaced to the PIOP.

The PIOC operates by interpreting a channel command program which resides in AP main data memory. A channel command program is written as a series of channel instructions, each of which contains information about the PIOP operation to be performed and appropriate parameters needed to carry out the operation. Channel instructions are described in detail in paragraph 6.2

Two versions of the PIOC interpreter are available from Floating-Point Systems. The first, referred to as DKPIOC, is used when the PIOP is interfaced to the Systems Industries SI9500 Disk Controller. The second, called GPIOC, is a general channel construct, with specific handshaking conventions that allow the PIOP to communicate with a variety of external devices, such as A/D converters and bulk memories.

The form of the channel command programs and many of the channel instructions are the same for both versions of the PIOC. Certain channel instructions may apply to only one or the other of the versions, or their interpretations may be somewhat different, depending on the version. Differences, where they exist, will be noted in the chapter and indicated by "(DKPIOC)" or "(GPIOC)."

The following paragraphs describe the channel command language, give some programming examples, and describe channel error conditions.

## 6.2 CHANNEL INSTRUCTIONS

A channel instruction consists of four 38-bit words in the format shown below in Figure 6-1.

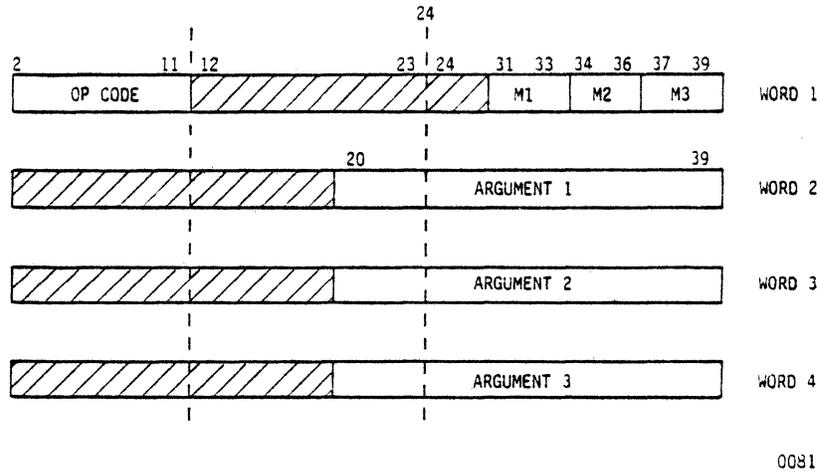


Figure 6-1 Channel Instruction Format

The first word of the instruction contains the operation code of the operation to be performed and three fields, M1, M2, and M3, which select the addressing mode for argument 1, argument 2, and argument 3, respectively. The three arguments occupy the right-most 20 bits of instruction words 2, 3, and 4.

One of three addressing modes - immediate, normal, or indirect - can be selected for each of the three arguments. A description of the three modes is given in Table 6-1.

Table 6-1 Addressing Modes

| M1, M2, or M3<br>CODE | ADDRESS<br>MODE | DESCRIPTION                                      |
|-----------------------|-----------------|--|
| 0                     | Immediate       | Argument contains operand itself.                |
| 1                     | Normal          | Argument contains address of operand.            |
| 2                     | Indirect        | Argument contains pointer to address of operand. |

0082

Channel instructions must reside in AP main data memory to be executed by the PIOP channel interpreter. Consequently, all addresses referred to by channel instructions are AP main data memory addresses.

To illustrate the difference in addressing modes, consider the following example. Suppose we are given the following main data locations and contents:

| <u>MAIN DATA ADDRESS</u> | <u>CONTENTS</u> |
|--------------------------|-----------------|
| 100                      | 2000            |
| 2000                     | 4751            |

Suppose argument 1 = 100. Then the operand used in connected with argument 1 depends on the addressing mode selected. In particular:

| <u>M1</u> | <u>MODE</u> | <u>OPERAND 1</u> | <u>COMMENTS</u>  |
|-----------|-------------|------------------|--|
| 0         | Immediate   | 100              | The argument itself is the operand.                                    |
| 1         | Normal      | 2000             | The argument itself contains the address (100) of the operand.         |
| 2         | Indirect    | 4751             | The argument contains a pointer to this address (2000) of the operand. |

The operand codes determine the PIOP operation to be performed by the channel instruction. A list of operation codes, together with the functions they perform, is given in Table 6-2 below. Mnemonics are included with each code for convenience. Where the operation is applicable only to either the DKPIOC or GPIOC, an appropriate indication is made next to the op code number. Operand 1 is abbreviated: OP1. OP2 = OP1 + OP2 means that operand 2 is replaced by the sum of operand 1 and the current value of operand 2. Remember that all operations such as ADD, SUB, RSH, etc., produce a 20-bit integer result in the AP main data memory. Zeros are written into the high 18 bits (bits 2-19) by these operations.

All instructions, even those requiring fewer than three arguments (e.g., ADD, MOV, etc.), must be written in the 4-word format. Arguments not needed by an instruction can contain arbitrary data. Since the channel does not use these words, they are available for other programming uses such as temporary storage.

Table 6-2 Channel Instruction Operation Codes

| OP CODE        | MNEMONIC | TITLE                         | FUNCTION  |
|----------------|----------|-------------------------------|---|
| 0              | ADD      | Main Data Add                 | $OP2 = OP1 + OP2$   |
| 1              | SUB      | Main Data Subtract            | $OP2 = OP2 - OP1$   |
| 2              | MOV      | Main Data Move                | $OP2 = OP1$   |
| 3              | AND      | Main Data And                 | $OP2 = OP1 \text{ and } OP2$  |
| 4              | IOR      | Main Data Or                  | $OP2 = OP1 \text{ or } OP2$   |
| 5              | XOR      | Main Data Exclusive Or        | $OP2 = OP1 \text{ xor } OP2$  |
| 6              | RSH      | Main Data Logical Right Shift | $OP2 = OP1$ shifted right 1 bit with a 0 in the most significant bit.   |
| 7              | JMPC     | Conditional Jump              | If $OP1$ and $OP2 \neq 0$ , then jump to $OP3$ (i.e., execute the command instruction beginning at main data address $OP3$ ), else execute the next sequential instruction.   |
| 10             | JMP      | Unconditional Jump            | Jump to $OP3$ (i.e., execute the command instruction beginning at main data address $OP3$ ).  |
| 11             | SF       | Set Flag                      | Set PIOP flag $OP1$ to 1 (flags 0 thru 6 can be used).  |
| 12             | CF       | Clear Flag                    | Clear PIOP flag $OP1$ to 0 (flags 0 thru 6 can be used).  |
| 13             | WFS      | Wait On Flag Set              | PIOP waits until flag $OP1$ is set to 1 by the AP (flags 0 thru 6 can be used).   |
| 14             | HALT     | Halt Channel                  | Stops PIOP execution.   |
| 15<br>(DKPIOC) | READ     | Disk Read                     | Transfer a block of $OP1$ 38-bit words from the disk, beginning at logical record number $OP2$ through the PIOP to AP main data memory, beginning at address $OP3$ .  |
| 16<br>(DKPIOC) | WRITE    | Disk Write                    | Transfer a block of $OP1$ 38-bit words from AP main data memory, beginning at address $OP3$ , through the PIOP to the disk, beginning at logical record number $OP2$ .  |
| 17<br>(DKPIOC) | INIT 300 | Disk Initialization (300 MB)  | Define the physical parameters of the disk corresponding to logical record number 0. The disk formatter control register is set to $OP1$ . The port and cylinder numbers are set by $OP2$ , and the head and sector numbers are set by $OP3$ . (Refer to paragraph 6.4 for details of the bit meanings.) INIT 300 must be executed prior to any READ or WRITE command on a 300 megabyte disk. |
| 20<br>(DKPIOC) | INIT 80  | Disk Initialization (80 MB)   | Same as above except for an 80 megabyte disk.   |
| 21<br>(DKPIOC) | SEEK     | Disk Seek                     | Causes the disk to be positioned to the port/cylinder address set in $OP1$ .  |
| 22<br>(DKPIOC) | FORMAT   | Disk Data Format              | Defines the data format for transfer of data to/from the disk where the format type and number is specified in $OP1$ . The four possible format types are listed in Table 6-3.  |
| 23<br>(DKPIOC) | RREG     | Read Register                 | Read disk register specified by $OP1$ into address $OP2$<br>$OP1 = 0$ = disk control register<br>$20$ = data buffer<br>$100$ = port/cylinder register<br>$140$ = seek status register<br>$200$ = word count register<br>$220$ = communication reg. (dual CPU)<br>$240$ = error register<br>$300$ = head/sector register<br>$340$ = seek address register                                      |
| 24<br>(DKPIOC) | WRTREG   | Write Register                | Write $OP2$ to the disk register specified by $OP1$ .   |

A convention used with the AP to describe a 38-bit word is to break the word into three pieces as follows:

| <u>BITS</u> | <u>NAME</u>   |
|-------------|---------------|
| 2-11        | exponent      |
| 12-23       | high mantissa |
| 24-39       | low mantissa  |

Thus, if in the first word of a channel instruction the op code = 7, M1 = 2, M2 = 0, and M3 = 1, then this word could be expressed as

7, 0, 201

If, in the second word of a channel instruction, argument 1 = 1234, then this would be written as

0, 0, 1234

This convention is used in writing channel programs in the following chapters.

### 6.3 WRITING CHANNEL COMMAND PROGRAMS

A channel command program consists of a series of channel instructions which must be loaded into AP main data memory in order to be executed by the PIOP. Normally, the command program will occupy a block of consecutive words in main data memory. Channel instructions are executed sequentially, except where jump instructions (JMP, JMPC) are involved. A channel program can be started by a call to PCGO either from the Fortran level, i.e., call PCGO, or from APAL, i.e., JSR PCGO. A command program is normally stopped by executing a channel HALT instruction (op code 14), although certain error conditions such as an improper op code, or a disk hardware error, can cause the channel program to halt prematurely. Such conditions are described in Chapter 6.6

Consider a simple example to illustrate the way that channel command programs are structured.

#### Example 1

Suppose that we want to use the PIOP to add the integer values in the right-most 20 bits of AP main data locations 100g through 105g and store the results in location 200g. (Obviously, this can be done more efficiently with AP instructions, but we're illustrating the PIOP.)

The major steps to be performed are:

1. initialize the sum to zero
2. add a 20-bit value to the sum
3. update pointer to next value
4. decrement count
5. if count not zero, go back to step 2
6. store result
7. halt

Let us now write a channel program for the PIOP to add these numbers. Let the program reside in main data, beginning at location 0 (although any location would do).

| <u>STEP</u> | <u>AD</u> | <u>INSTRUCTION</u> | <u>COMMENT</u>   |
|-------------|-----------|--------------------|--|
| 1.          | 0         | 1,0,000            | Clears location 2 by subtracting location 2 from itself. Contents of 1, 2 and 3 are arbitrary. Location 2 will be used as the sum.                                       |
|             | 1         | 0,0,0              |  |
|             | 2         | 0,0,0              |  |
|             | 3         | 0,0,0              |  |
| 2.          | 4         | 0,0,110            | Adds the contents of the word pointed to by MD5 (initially 100) to the sum in MD2. Contents of MD7 are arbitrary. MD5 will be incremented on each pass through the loop. |
|             | 5         | 0,0,100            |  |
|             | 6         | 0,0,2              |  |
|             | 7         | 0,0,0              |  |
| 3.          | 10        | 0,0,010            | Adds 1 to the address pointer, MD5. Contents of MD13 are arbitrary.  |
|             | 11        | 0,0,1              |  |
|             | 12        | 0,0,5              |  |
|             | 13        | 0,0,0              |  |
| 4.          | 14        | 1,0,000            | Decrement the loop counter, MD16 which was initially 6.  |
|             | 15        | 0,0,1              |  |
|             | 16        | 0,0,6              |  |
|             | 17        | 0,0,0              |  |
| 5.          | 20        | 7,0,010            | Test (by means of a logical AND) if counter MD16 is zero. If not, go back to step 2, MD4. If zero, then proceed to step 6, MD24.   |
|             | 21        | 0,0,7              |  |
|             | 22        | 0,0,16             |  |
|             | 23        | 0,0,4              |  |
| 6.          | 24        | 2,0,110            | Store the sum in location 200.   |
|             | 25        | 0,0,2              |  |
|             | 26        | 0,0,200            |  |
|             | 27        | 0,0,0              |  |
| 7.          | 30        | 14,0,0             | Halt the PIOP. Contents of MD31, 32, 33 are arbitrary.   |
|             | 31        | 0,0,0              |  |
|             | 32        | 0,0,0              |  |
|             | 33        | 0,0,0              |  |

As an aid in writing channel programs, the PPAL assembler can be used. Mnemonics can be defined to identify op codes, addresses, and constants, making the program more readable. In addition, the PPAL assembler provides a Fortran program output for the channel program that can be called at runtime to load the channel program into AP main data, beginning at a specified address. (Refer to the PIOP assembler, Chapter 5.) A PPAL program for the above example is shown in Figure 6-2.

```

    SSUB EXAMP1
"
"   --- ABSTRACT ---
"THIS PIOC CHANNEL PROGRAM ADDS THE INTEGER VALUES IN THE RIGHTMOST
"20-BITS OF AP MAIN DATA LOCATIONS 100 THROUGH 105 (OCTAL) AND STORES
"THE SUM IN LOCATION 200 (OCTAL).
"
"DEFINE THE OP CODES BY MNEMONICS:
    ADD  $EQU 0
    SUB  $EQU 1
    MOV  $EQU 2
    JMPC $EQU 7
    HALT $EQU 14
"
"THE CHANNEL PROGRAM EXPECTS TO BE LOADED BEGINNING AT MD ADDRESS 0,
"BUT THAT CAN BE CHANGED BY EDITING THE VALUE FOR OFFSET BELOW AND
"REASSEMBLING.
    OFFSET $EQU 0           "MD ADDRESS FOR FIRST WORD OF CHANNEL
                           "PROGRAM
"DEFINE DATA ADDRESSES:
    MDA   $EQU 100         "ADDRESS OF FIRST DATA WORD
    RESULT $EQU 200       "ADDRESS FOR RESULT
"
"DEFINE CONSTANTS:
    ONE   $EQU 1           "USED FOR INCREMENT/DECREMENT
    N     $EQU 6           "WORD COUNT
    ZMASK $EQU 7           "MASK FOR COUNTER
"ADDRESSING MODES:
    M000  $EQU 0           "MODE M1=0, M2=0, M3=0
    M010  $EQU 10         "MODE M1=0, M2=1, M3=0
    M110  $EQU 110        "MODE M1=1, M2=1, M3=0
"
"THE PROGRAM CAN BE LOADED INTO MD FROM FORTRAN AND STARTED BY THE
"FOLLOWING:
"    CALL GPIOC(1000,0,1,I) / LOADS PIOC INTERPRETER (COULD BE DKPIOC)
"
"
"
"    CALL EXAMP1(0,0,3,I) / LOADS CHANNEL PROGRAM BEGINNING AT
"                          / MD0 AND BEGINS EXECUTION
"
"
"
"

```

Figure 6-2 Channel Program Example 1

```

"HERE BEGINS THE CHANNEL PROGRAM:
"
EXAMP1  $VAL SUB,0,M000          "1. CLEARS LOCATION USED FOR SUM
        $VAL 0,0,0
SUM      $EQU .+OFFSET          "  NEXT LOCATION IS USED FOR SUM
        $VAL 0,0,0
        $VAL 0,0,0
"
"THIS IS THE BEGINNING OF THE CHANNEL PROGRAM LOOP
"
LOOP     $EQU .+OFFSET
        $VAL ADD,0,M110          "2. ADDS VALUE TO SUM
MDPTR    $EQU .+OFFSET          "  NEXT LOCATION POINTS TO DATA
        $VAL 0,0,MDA
        $VAL 0,0,SUM
        $VAL 0,0,0
"
        $VAL ADD,0,M010          "3. UPDATES POINTER TO DATA
        $VAL 0,0,ONE
        $VAL 0,0,MDPTR
        $VAL 0,0,0
"
        $VAL SUB,0,M000          "4. DECREMENT COUNTER
        $VAL 0,0,ONE
CTR      $EQU .+OFFSET          "  NEXT LOCATION IS USED AS COUNTER
        $VAL 0,0,N
        $VAL 0,0,0
"
        $VAL JMPC,0,M010         "5. BRANCH BACK TO LOOP
        $VAL 0,0,ZMASK          "  IF NOT DONE
        $VAL 0,0,CTR
        $VAL 0,0,LOOP
"
        $VAL MOV,0,M110         "6. STORE THE SUM
        $VAL 0,0,SUM
        $VAL 0,0,RESULT
        $VAL 0,0,0
"
        $VAL HALT,0,0           "7. HALT THE PIOP
        $VAL 0,0,0
        $VAL 0,0,0
        $VAL 0,0,0
"
        $END                     "END OF PROGRAM

```

Figure 6-2 Channel Program Example 1 (cont.)

#### 6.4 ACCESSING DISK DATA USING DKPIOC

Physically, the data on a disk is located through the PIOP disk interface by specifying a port, cylinder, head, and sector address. One sector contains 256 16-bit words.

The DKPIOC interpreter allows the user to access the disk by means of "logical records", where one logical record is defined to be 256 AP words. Prior to read/write calls to the disk, the user specifies the port, cylinder, head, and sector which will be called logical record 0 for subsequent reads and writes. In a channel program this is done by means of the disk initialization instruction (INIT 80 or INIT 300). The instruction used depends on whether an 80 megabyte (CDC 9762) or 300 megabyte disk (CDC 9766) is being used. INIT 80 can be used for a 40 megabyte disk (CDC 9760) and INIT 300 for a 150 megabyte disk (CDC 9764).

Prior to read/write calls, the user specifies 1 of 4 formats by which data is to be transferred between the disk and the AP main data memory. In a channel program, this is done by means of a formatting instruction (FORMAT). The four possible formats are summarized in Table 6-3. Note that one logical record occupies one disk sector in format 0, three disk sectors in format 1, and two disk sectors in formats 2 and 3.

Logical records are defined relative to the most recent record 0 initialization. For example, if format type 1 is specified and if logical record 0 is defined as disk port 1, cylinder 4, head 2, sector 3, then logical record 1 refers to port 1, cylinder 4, head 2, sector 6.

Table 6-3 Disk Data Format Types

| TYPE | NAME                      | DISK WORDS PER AP WORD | DESCRIPTION   |
|------|---------------------------|------------------------|---|
| 0    | 16-bit                    | 1                      | Only the low mantissa 16 bits of MD (bits 24-39) is transferred to the disk.  |
| 1    | 38-bit                    | 3                      | The 38-bit AP word is transferred to the disk in three parts:<br>a. low mantissa (bits 24-39)<br>b. high mantissa (bits 12-23)<br>c. exponent (bits 2-11) |
| 2    | 32-bit truncated mantissa | 2                      | The mantissa of the AP word is truncated and the remaining 32 bits transferred in two parts:<br>a. low word (bits 18-33)<br>b. high word (bits 2-17)      |
| 3    | 32-bit truncated exponent | 2                      | The exponent of the AP word is truncated and the remaining 32 bits transferred in two parts:<br>a. low word (bits 24-39)<br>b. high word (bits 8-23)      |

3253

The INIT 80 or INIT 300 instruction requires that contents be specified for three 16-bit disk formatter registers as follows: (Bits here are numbered 15 to 0, left-to-right. Unspecified bits are not used.)

Disk formatter control register:

Bit 3 - verify  
9 - format enable  
11 - strobe late  
12 - strobe early  
13 - offset -  
14 - offset +

Port/cylinder address register:

9-0 - cylinder number  
(0-410 if CDC 9760 or 9764)  
(0-822 if CDC 9762 or 9766)

Head/sector address register:

Bits 9-5 - head number  
(0-4 if CDC 9760 or 9762)  
(0-18 if CDC 9764 or 9766)  
4-0 - sector number (0-31)

Generally, all control register bits can be set to zero. Thus, to initialize logical record 0 to port 1, cylinder 4, head 2, sector 3, the channel instruction for an 80 megabyte disk is:

14, 0, 0  
0, 0, 0  
0, 0, 2004  
0, 0, 103

## 6.5 AP/PIOP PROCESS SYNCHRONIZATION

It is important to remember that the AP and the PIOP can run in parallel. A common requirement using this capability is evident when the PIOP is bringing new data from a device such as a disk into AP main data memory while the AP is processing data previously transferred from the device. For processes such as this, the eight PIOP communication flags provide a convenient means of synchronization of AP and PIOP processing.

To illustrate a synchronized process between the AP and the PIOP, consider the following example:

### Example 2

Suppose that 100 blocks of floating point data, each 1024 38-bit words long, reside on an 80 megabyte disk, beginning at port 1, cylinder 7, head 3, sector 2. We want to do a real to complex FFT on each block and accumulate the auto-spectrum of all blocks. In the interest of speed, the data transfer and the computation should be overlapped, thus, we should double buffer the data in main data memory.

Calling the two data buffers in main data memory A and B, and calling two PIOP flags AFLAG and BFLAG, we can conceive of the two processes as follows:

| <u>PIOP Process</u>          | <u>AP Process</u>              |
|------------------------------|--------------------------------|
| Sets AFLAG, BFLAG            | Start DKPIOC                   |
| Initialize pointers, etc.    |                                |
| PPLOOP: Wait until AFLAG set | Initialize block counter       |
| Read into buffer A           |                                |
| Clear AFLAG                  | APLOOP: Wait until AFLAG clear |
| Wait until BFLAG set         | Process buffer A               |
| Read into buffer B           | Set AFLAG                      |
| Clear BFLAG                  | Wait until BFLAG clear         |
| Go to PPLOOP                 | Process buffer B               |
|                              | Set BFLAG                      |
|                              | Decrement count                |
|                              | Go to APLOOP if not zero       |
|                              | If done, turn off PIOP         |

A channel program to perform the PIOP double buffering process is shown below in Figure 6-3.

```

    $SUB EXAMP2
"
"   --- ABSTRACT ---
"THIS PIOP CHANNEL PROGRAM READS BLOCKS OF N 38-BIT WORDS FROM AN 80-MEGABYTE
"DISK AND ALTERNATELY PUTS THEM IN TWO DIFFERENT BUFFERS IN AP MAIN DATA
"MEMORY. THE READS BEGIN FROM LOGICAL RECORD 0.
"
"FOR PURPOSES OF ILLUSTRATION:
"      N = 1024
" FWA BUFF A = 2048
"      B = 4096
"
"   AND LOGICAL RECORD 0 IS DEFINED AS PORT 1, CYLINDER 7, HEAD 3, SECTOR 2.
"
"   FORMAT 1 IS USED (38-BIT DATA TRANSFER AS 3 WORDS: LOW MANTISSA, HIGH
"   MANTISSA, AND EXPONENT).
"
"9 MAIN DATA LOCATIONS IMMEDIATELY FOLLOWING THE BODY OF THE CHANNEL PROGRAM
"ARE INITIALIZED TO REFLECT THESE VALUES, BUT THEY COULD BE EASILY MODIFIED
"AS A GROUP FROM ANOTHER AP PROGRAM (OR APPUT) PRIOR TO CALLING THE CHANNEL
"PROGRAM.
"
"THE CHANNEL PROGRAM EXPECTS TO BE LOADED BEGINNING AT MD ADDRESS 64., BUT
"THAT CAN BE CHANGED BY EDITING THE VALUE FOR OFFSET BELOW AND REASSEMBLING
"THE PROGRAM WITH PPAL.
"
"   OFFSET $EQU 64.           "MD ADDRESS FOR FIRST WORD OF CHANNEL PROGRAM
"
"FOR COMMUNICATION WITH THE AP, PIOP FLAGS 0 AND 1 ARE USED.
"   AFLAG $EQU 0
"   BFLAG $EQU 1
"
"AFLAG IS CLEARED BY THE CHANNEL PROGRAM TO INDICATE DATA HAS BEEN READ
"INTO BUFFER A. THE CHANNEL PROGRAM EXPECTS AFLAG TO BE SET BY THE AP
"WHEN PROCESSING ON BUFFER A IS COMPLETE AND READY FOR NEW DATA TO BE
"LOADED INTO BUFFER A. BFLAG IS USED SIMILARLY WITH RESPECT TO BUFFER B.
"
"THE CHANNEL PROGRAM WILL CONTINUE UNTIL STOPPED BY THE AP, E.G. BY JSR PPRS.
"
"MNEMONICS FOR CHANNEL OP CODES:
"   ADD $EQU 0           "ADD
"   SUB $EQU 1           "SUBTRACT
"   JMP $EQU 10          "JUMP
"   SF $EQU 11           "SET FLAG
"   CF $EQU 12           "CLEAR FLAG
"   WFS $EQU 13          "WAIT UNTIL FLAG SET
"   HALT $EQU 14         "PIOP HALT
"   READ $EQU 15         "DISK READ
"   INIT80 $EQU 20       "DISK INITIALIZE LOGICAL RECORD 0
"   FORMAT $EQU 22       "FORMAT DATA
"
"CONSTANTS FOR ADDRESSING MODES
"   M000 $EQU 0          "MODE M1=0, M2=0, M3=0
"   M100 $EQU 100       "MODE M1=1, M2=0, M3=0
"   M110 $EQU 110       "MODE M1=1, M2=1, M3=0
"   M111 $EQU 111       "MODE M1=1, M2=1, M3=1
"

```

Figure 6-3 Channel Program Example 2

"HERE BEGINS THE CHANNEL PROGRAM

"

EXAMP2 SVAL SF,0,M000 "INITIALLY SET AFLAG  
SVAL 0,0,AFLAG  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL SF,0,M000 "INITIALLY SET BFLAG  
SVAL 0,0,BFLAG  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL FORMAT,0,M100 "SPECIFY DATA FORMAT TYPE  
SVAL FMT,0,0  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL INIT80,0,M111 "INITIALIZE DISK PARAMETERS  
SVAL 0,0,CR "FOR LOGICAL RECORD 0  
SVAL 0,0,PC  
SVAL 0,0,HS

"

SVAL SUB,0,M110 "START READING AT RECORD 0  
SVAL 0,0,LR  
SVAL 0,0,LR  
SVAL 0,0,0

"

"HERE BEGINS THE LOOP TO READ INTO BUFFERS A AND B

"

PPLOOP \$EQU .+OFFSET  
SVAL WFS,0,M000 "WAIT FOR AFLAG TO BE SET BY AP  
SVAL 0,0,AFLAG  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL READ,0,M111 "READ N WORDS FROM DISK TO BUFFER A  
SVAL 0,0,N  
SVAL 0,0,LR  
SVAL 0,0,A

"

SVAL CF,0,M000 "CLEAR AFLAG  
SVAL 0,0,AFLAG  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL ADD,0,M110 "UPDATE LOGICAL RECORD NUMBER  
SVAL 0,0,LRINC  
SVAL 0,0,LR  
SVAL 0,0,0

"

SVAL WFS,0,M000 "WAIT FOR BFLAG TO BE SET BY AP  
SVAL 0,0,BFLAG  
SVAL 0,0,0  
SVAL 0,0,0

"

SVAL READ,0,M111 "READ N WORDS INTO BUFFER B  
SVAL 0,0,N  
SVAL 0,0,LR  
SVAL 0,0,B

"

Figure 6-3 Channel Program Example 2 (cont.)

```

        $VAL CF,0,M000          "CLEAR BFLAG
        $VAL 0,0,BFLAG
        $VAL 0,0,0
"
        $VAL ADD,0,M110        "UPDATE LOGICAL RECORD NUMBER
        $VAL 0,0,LRINC
        $VAL 0,0,LR
        $VAL 0,0,0
"
        $VAL JMP,0,M100        "GO BACK TO BEGINNING OF LOOP
        $VAL 0,0,PPLOOP
        $VAL 0,0,0
        $VAL 0,0,0
"
"PARAMETER STORAGE:
"
A      $EQU .+OFFSET
        $VAL 0,0,2048.        "ADDRESS OF BUFFER A
"
B      $EQU .+OFFSET
        $VAL 0,0,4096.        "ADDRESS OF BUFFER B
"
N      $EQU .+OFFSET
        $VAL 0,0,1024.        "WORD COUNT
"
FMT    $EQU .+OFFSET
        $VAL 0,0,1            "FORMAT TYPE 1 (38-BITS, 3 PARTS)
"
CR     $EQU .+OFFSET
        $VAL 0,0,0            "DISK CONTROL REGISTER WORD
"
PC     $EQU .+OFFSET
        $VAL 0,0,2007        "PORT 1, CYLINDER 7
"
HS     $EQU .+OFFSET
        $VAL 0,0,142         "HEAD 3, SECTOR 2
"
LRINC  $EQU .+OFFSET
        $VAL 0,0,4           "RECORDS PER DATA BLOCK
"
LR     $EQU .+OFFSET
        $VAL 0,0,0           "CURRENT LOGICAL RECORD NUMBER
"
"
SEND   "END OF CHANNEL PROGRAM

```

Figure 6-3 Channel Program Example 2 (cont.)

## 6.6 PIOC ERROR CONDITIONS

The PIOC uses eight consecutive main data locations to hold status information about the channel program performance. Following termination of the channel program, the user can examine the contents of these words to determine whether the program was successfully completed or whether the channel program was abnormally terminated.

The channel program is normally started by using the AP subroutine PCGO, which can be called either from Fortran or from the APAL program. Calling parameters for PCGO are, 1) the starting address of the channel program, and, 2) the first word address of the 8-word status buffer.

The first word of the status buffer contains a 0 if the channel program was successfully completed, otherwise the first word contains the main data address of the channel instruction whether the error occurred.

If a channel error occurs, status buffer words two through eight contain information relating to the particular error. For example, if a disk hardware error occurs using DKPIOC, then the following information is stored in the status buffer:

- word 1 Address of channel instruction where error occurred
- word 2 Disk controller error register
- word 3 Disk controller seek status register
- word 4 Disk controller port/cylinder register
- word 5 Disk controller head/sector register
- word 6 Disk controller control register
- word 7 Disk word count
- word 8 Reserved for future use

Conditions which can cause a channel program to terminate abnormally are:

- a. Illegal op code
- b. Disk hardware error (DKPIOC) or drive not connected (open cable)

## CHAPTER 7

### FORTRAN OPERATIONS

#### 7.1 INTRODUCTION

There are a number of Fortran level calls that are available for use with the PIOP. These calls can be used for communication with: the PIOP, the AP disk, or the PIOP disk channel.

The available Fortran calls are listed in Table 7-1. In addition, a description of each call is presented later in this chapter.

Table 7-1 Fortran Calls

| GENERAL PURPOSE                      | FORTRAN CALL  | DESCRIPTION   |
|--------------------------------------|---|---|
| COMMUNICATION WITH PIOP              | Ppload<br>PPGO<br>PPRS<br>PPSTAT<br>PPWAIT<br>PPFrd<br>PPFSET<br>PPFCLR | Load PIOP from AP-120 main data memory.<br>Start PIOP.<br>Reset PIOP.<br>Get PIOP run/halt status.<br>Wait for PIOP to halt.<br>Read a PIOP flag from the AP-120.<br>Set a PIOP flag from the AP-120.<br>Clear a PIOP flag from the AP-120. |
| COMMUNICATION WITH AP DISK           | INPPDK<br>RDPPDK<br>WRPPDK<br>WRDPPD                                    | Initialize PIOP disk parameters.<br>Read data from PIOP disk to AP-120 main data memory.<br>Write data from AP-120 main data memory to PIOP disk.<br>Write data to, and then read data from, the PIOP disk.                                 |
| COMMUNICATION WITH PIOP DISK CHANNEL | PCGO<br>PCSTAT  | Start PIOP channel program.<br>Get PIOP channel error status.   |
| OTHER                                | PEXEC   | PIOP executive loader.  |

0089

The PIOP assembly code programs are identified by subroutine name. The assembler produces a Fortran subroutine of that name which contains the 38-bit PIOP instruction words in DATA statements. The subroutine is called at run time with a parameter that causes one of the following actions to occur:

- a. The 38-bit instruction words are loaded into the AP main data memory.
- b. The 38-bit instruction words are loaded into the PIOP program memory through the AP main data memory.
- c. The 38-bit instruction words are loaded into PIOP program memory and the PIOP begins execution of the loaded instructions (that is, load and go).
- d. If the 38-bit instruction words constitute a channel program, then the words are loaded into the AP main data memory and start the PIOC interpreter (previously loaded into the PIOP) to execute the loaded channel program (that is, load channel and go).

When using Fortran level calls that communicate with the AP disk, the calls execute through DKPIOC (the PIOP disk channel interpreter). This is loaded into the PIOP from the Fortran level by calling DKPIOC. (Refer to item c. above.)

Error information for the PIOP channel is returned in an 8-word channel status buffer in the AP main data memory. This error information is listed in Table 7-2:

Table 7-2 Error Information

| WORD | ERROR INFORMATION  |
|------|--|
| 0    | Word = 0 if channel operation was successfully completed.<br>Word = address in channel program of operation being attempted when error occurred. |
| 1    | Disk controller error register.  |
| 2    | Disk controller seek status register.  |
| 3    | Disk controller port/cylinder address register.  |
| 4    | Disk controller head/sector address register.  |
| 5    | Disk controller control register.  |
| 6    | Word count register.   |
| 7    | Reserved for future use.   |

## 7.2 FORTRAN CALLS

A description of each of the 15 available PIOP Fortran calls is presented in subsequent paragraphs. The description of each call follows the same format which contains the following information:

PURPOSE:

FORTRAN CALL:

PARAMETERS:

DESCRIPTION:

EXAMPLE:

EXECUTION TIME:

PROGRAM SIZE:

APAL CALL:

SCRATCH:

EXTERNALS:

In addition to the Fortran calls, examples of Fortran subroutines created by the PIOP assembler are presented in paragraph 7.3.

A description of the conventions used in the calls is presented in Chapter 3 of the Math Library Manual (FPS 7288-03).

7.2.1 LOAD PIOP FROM AP MD (PPLOAD)

PURPOSE: To load the PIOP program source memory from the AP-120B main data memory.

FORTRAN CALL: CALL PPLOAD(A,C,N)

PARAMETERS: A = Source vector base address (AP MD)  
 C = Destination vector base address (PIOP PS)  
 N = Element count

FORMULA:  $C(m) = A(m)$  for  $m=0$  to  $N-1$

DESCRIPTION: Moves N 38-bit words beginning at AP main data address A to PIOP program source memory beginning at address C.

EXAMPLE: CALL PPLOAD(100,0,50)  
 Stores into PIOP program source memory locations 0,1,.....,48,49 the PIOP program stored in AP main data locations 100,101,.....,148,149.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us) |                 |
|---------------|------|---------|-------|-----------|-----------------|
| TIME/LOOP:    | 0.3  | 0.3     | 0.3   | 4.7       | (167 ns memory) |
| (us)          | 0.3  | 0.3     | 0.3   | 4.7       | (333 ns memory) |
| PROGRAM SIZE: | 32   |         |       |           | (167 ns memory) |
| (AP words)    | 32   |         |       |           | (333 ns memory) |

---

APAL CALL: JSR PPLOAD  
 SCRATCH: SP(0-2,14,15),DPX(0),MD  
 EXTERNALS: PPWAIT

### 7.2.2 START PIOP (PPGO)

PURPOSE: To start the PIOP running.

FORTRAN CALL: CALL PPGO(A)

PARAMETERS: A = PIOP program starting address  
(PIOP PS)

FORMULA: N/A

DESCRIPTION: Starts the PIOP running beginning at PIOP program source address A. This routine assumes that the program has been loaded previously (see PPLDAC).

EXAMPLE: CALL PPLDAC(100,10,50)  
CALL PPGO(10)  
A 50-word PIOP program is loaded from AP main data memory locations 100,101,....,148,149 into PIOP program memory locations 10,11,....,58,59 by PPLDAC. PPGO then causes the PIOP to start executing the program, beginning at PIOP program location 10.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 3.7  | 3.7     | 3.7   | N/A (167 ns memory) |
| (us)      | 3.7  | 3.7     | 3.7   | N/A (333 ns memory) |

|               |    |  |  |                 |
|---------------|----|--|--|-----------------|
| PROGRAM SIZE: | 22 |  |  | (167 ns memory) |
| (AP words)    | 22 |  |  | (333 ns memory) |

---

APAL CALL: JSR PPGO  
SCRATCH: SP(0,1,14,15),DPX(0)  
EXTERNALS: PPWAIT

### 7.2.3 RESET PIOP (PPRS)

PURPOSE: To reset the PIOP.

FORTRAN CALL: CALL PPRS

PARAMETERS: N/A

FORMULA: N/A

DESCRIPTION: Stops the PIOP, clears the PIOP interrupts, and issues a reset command to the interface between the PIOP and the external device.

EXAMPLE: CALL PPRS  
Stops the PIOP, clears the PIOP interrupts, and issues a reset command to the interface between the PIOP and the external device.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 0.7  | 0.7     | 0.7   | N/A (167 ns memory) |
| (us)          | 0.7  | 0.7     | 0.7   | N/A (333 ns memory) |
| PROGRAM SIZE: | 4    |         |       | (167 ns memory)     |
| (AP words)    | 4    |         |       | (333 ns memory)     |

---

APAL CALL: JSR PPRS  
SCRATCH: SP(0)  
EXTERNALS: None

#### 7.2.4 GET PIOP STATUS (PPSTAT)

PURPOSE: To test the run/halt status of the PIOP.

FORTRAN CALL: CALL PPSTAT

PARAMETERS: N/A

FORMULA: SP(15) = 1 if PIOP running  
0 if PIOP halted

DESCRIPTION: Tests the run/halt status of the PIOP. If the PIOP is running, SP(15) is set to 1. If the PIOP is not running, SP(15) is set to 0.

EXAMPLE: CALL PPSTAT  
CALL APCHK(I)  
The run/halt status of the PIOP is tested by PPSTAT. The status is returned in integer variable I by APCHK. If the PIOP was running, I=1, otherwise I=0.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 1.0  | 1.0     | 1.0   | N/A (167 ns memory) |
| (us)      | 1.0  | 1.0     | 1.0   | N/A (333 ns memory) |

|               |   |  |  |                 |
|---------------|---|--|--|-----------------|
| PROGRAM SIZE: | 6 |  |  | (167 ns memory) |
| (AP words)    | 6 |  |  | (333 ns memory) |

---

APAL CALL: JSR PPSTAT  
SCRATCH: SP(14)  
EXTERNALS: None

### 7.2.5 WAIT FOR PIOP (PPWAIT)

PURPOSE: To wait for the PIOP to halt.

FORTRAN CALL: CALL PPWAIT

PARAMETERS: N/A

FORMULA: N/A

DESCRIPTION: Waits until the PIOP has halted. This routine should be followed by a call to APWR.

EXAMPLE: CALL PPWAIT  
CALL APWR  
PPWAIT waits until the PIOP has halted before continuing.  
APWR waits until the AP1208 has halted before continuing.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | ##   | ##      | ##    | N/A (167 ns memory) |
| (us)      | ##   | ##      | ##    | N/A (333 ns memory) |

(Execution time depends on when the PIOP halts. The routine is completed within 2.5 us after the PIOP halts.)

|               |    |                 |
|---------------|----|-----------------|
| PROGRAM SIZE: | 10 | (167 ns memory) |
| (AP words)    | 10 | (333 ns memory) |

---

APAL CALL: JSR PPWAIT  
SCRATCH: SP(14,15)  
EXTERNALS: PPSTAT

### 7.2.6 READ PIOP FLAG FROM AP (PPFRD)

PURPOSE: To read one of the eight PIOP communication flags from the AP.

FORTRAN CALL: CALL PPFRD(N)

PARAMETERS: N = PIOP flag number (0 to 7)

FORMULA: SP(15) = PIOP FLAG (N) where N is from 0 to 7  
and the flag value is either 0 or 1

DESCRIPTION: Causes the AP to read the value of PIOP communication flag N, and put the value into SP(15). The value read will be either 0 or 1.

Any of the 8 PIOP communication flags can be set, cleared, or read from either the AP or the PIOP.

EXAMPLE: CALL PPFRD(4)  
CALL APCHK(I)  
The value of PIOP flag 4 is put into SP(15) by PPFRD. APCHK then puts the value into integer variable I, which will be 1 if the flag was set, 0 if the flag was cleared.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 1.2  | 1.2     | 1.2   | N/A (167 ns memory) |
| (us)          | 1.2  | 1.2     | 1.2   | N/A (333 ns memory) |
| PROGRAM SIZE: | 7    |         |       | (167 ns memory)     |
| (AP words)    | 7    |         |       | (333 ns memory)     |

---

APAL CALL: JSR PPFRD  
SCRATCH: SP(14,15)  
EXTERNALS: None

### 7.2.7 SET PIOP FLAG FROM AP (PPFSET)

PURPOSE: To set one of the eight PIOP communication flags from the AP.

FORTRAN CALL: CALL PPFSET(N)

PARAMETERS: N = PIOP flag number (0 to 7)

FORMULA: PIOP FLAG (N) = 1 where N is from 0 to 7

DESCRIPTION: Causes the AP to set the value of PIOP communication flag number N to 1, where N must be from 0 to 7.

Any of the 8 PIOP communication flags can be set, cleared, or read from either the AP or the PIOP.

EXAMPLE: CALL PPFSET(4)  
The value of PIOP flag 4 is set to 1 by the AP.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 0.8  | 0.8     | 0.8   | N/A (167 ns memory) |
| (us)          | 0.8  | 0.8     | 0.8   | N/A (333 ns memory) |
| PROGRAM SIZE: | 5    |         |       | (167 ns memory)     |
| (AP words)    | 5    |         |       | (333 ns memory)     |

---

APAL CALL: JSR PPFSET  
SCRATCH: SP(14,15)  
EXTERNALS: None

### 7.2.8 CLEAR PIOP FLAG FROM AP (PPFCLR)

PURPOSE: To clear one of the eight PIOP communication flags.

FORTRAN CALL: CALL PPFCLR(N)

PARAMETERS: N = PIOP flag number (0 to 7)

FORMULA: PIOP FLAG (N) = 0 where N is from 0 to 7

DESCRIPTION: Causes the AP to clear the value of PIOP communication flag number N to 0, where N must be from 0 to 7.

Any of the 8 PIOP communication flags can be set, cleared, or read from either the AP or the PIOP.

EXAMPLE: CALL PPFCLR(4)  
The value of PIOP flag 4 is cleared to 0 by the AP.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 0.8  | 0.8     | 0.8   | N/A (167 ns memory) |
| (us)          | 0.8  | 0.8     | 0.8   | N/A (333 ns memory) |
| PROGRAM SIZE: | 5    |         |       | (167 ns memory)     |
| (AP words)    | 5    |         |       | (333 ns memory)     |

---

APAL CALL: JSR PPFCLR  
SCRATCH: SP(14,15)  
EXTERNALS: None

## 7.2.9 INITIALIZE PIOP DISK PARAMETERS (INPPDK)

**PURPOSE:** To specify the disk type, the data format, and the physical parameters for PIOP disk logical record 0. These parameters are used by subsequent disk read/write calls which are made through the PIOP disk channel interpreter program.

**FORTRAN CALL:** CALL INPPDK(DKTYPE,FMT,CR,PC,HS,WKA)

**PARAMETERS:** DKTYPE = Disk type flag  
= 0 - CDC 9760 or 9762 disk  
= 1 - CDC 9764 or 9766 disk

FMT = Data format type  
= 0 - Low mantissa portion of MD  
(bits 24-39) only  
(1 disk word = 1 AP word)  
= 1 - Complete 38-bit AP word in  
3 parts:  
-- Low mantissa (bits 24-39)  
-- High mantissa (bits 12-23)  
-- Exponent (bits 2-11)  
(3 disk words = 1 AP word)  
= 2 - Mantissa truncated with remaining  
32 bits transferred in 2 parts:  
-- Low word (bits 18-33)  
-- High word (bits 2-17)  
(2 disk words = 1 AP word)  
= 3 - Exponent truncated with remaining  
32 bits transferred in 2 parts:  
-- Low word (bits 24-39)  
-- High word (bits 8-23)

(Note: Bits numbered 2-39, left to right.)

CR = Contents for disk formatter  
control register  
Bit 3 - verify  
9 - Format Enable  
11 - Strobe Late  
12 - Strobe Early  
13 - Offset-  
14 - Offset+

PC = Contents for port/cylinder  
address register  
Bits 11-10 - Port number (0-3)  
9-0 - Cylinder number  
(0-410 if CDC 9760, 9764)  
(0-822 if CDC 9762, 9766)  
= 1024\*Port + Cylinder

## 7.2.9 INITIALIZE PIOP DISK PARAMETERS (INPPDK) (cont.)

HS = Contents for head/sector  
address register  
bits 9-5 - Head number  
          (0-4 if CDC 9760, 9762)  
          (0-18 if CDC 9764, 9766)  
bits 4-0 - Sector number (0-31)  
          = 32\*Head + Sector  
(Note: For CR, PC, and HS, bits numbered 15-0  
left to right. Unspecified bits not used.)

WKA = Base address in AP MD of 20-word  
work buffer  
Words 0-7 - Channel status buffer  
      " 8-19 - Channel program

FORMULA: N/A

DESCRIPTION: Selects the type of disk to be used. Specifies the format by which data will be transferred between the disk and AP main data memory. Specifies the contents of the disk formatter control register (which handles such special control functions as format enable, read verify after write, data strobe delays, and head position offsets). Initializes the physical location of logical record 0 on the disk in terms of the port/cylinder number, and the head/sector number. A logical record is defined as 256 AP main data words. Depending on the format selected, a logical record will occupy 1, 2, or 3 256-word sectors on the disk.

A 20-word work buffer in AP main data memory, beginning at address WKA, is used by the routine. The first 8 words are for the channel error status buffer, while the last 12 are used for a channel program which the routine generates.

After generating the channel program, the routine calls the PIOP disk channel interpreter to execute it. The PIOP disk channel interpreter program DKPIOC must have been loaded into the PIOP program memory prior to the call to INPPDK.

All parameters defined by INPPDK are valid for subsequent disk read/write calls (e.g. RPPDK, WRPPDK) as long as DKPIOC remains resident in the PIOP program memory.

7.2.9 INITIALIZE PIOP DISK PARAMETERS (INPPDK) (cont.)

disk is specified. Subsequent data transfers will be made in format 1, whereby 38-bit AP words are transferred to/from the disk as 3 16-bit words (low mantissa, high mantissa, and exponent). For this format a logical record occupies 3 256-word disk sectors. Thus logical record 1 would refer to port 1, cylinder 4, head 2, sector 8. AP1208 main data memory locations 100-119 are used as a work area.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 9.8  | 9.8     | 9.8   | N/A (167 ns memory) |
| (us)          | 10.8 | 10.8    | 10.8  | N/A (333 ns memory) |
| PROGRAM SIZE: | 45   |         |       | (167 ns memory)     |
| (AP words)    | 45   |         |       | (333 ns memory)     |

-----

APAL CALL: JSR INPPDK  
 SCRATCH: SP(0,1,14,15),DPX(0),DPY(0)  
 EXTERNALS: PCGO,PCSTAT  
 EXAMPLE: CALL DKPIOC(0,0,2,ISIZE)  
 CALL INPPDK(0,1,0,1028,69,100)  
 The call to DKPIOC brings the PIOP disk channel interpreter program through AP1208 main data locations 0-(ISIZE-1) and into PIOP program memory locations 0-(ISIZE-1). INPPDK specifies that disk port 1, cylinder 4, head 2, sector 5 will define logical record 0 for subsequent read/write calls to the disk (e.g., RDPPDK, WRPPDK). A CDC 9762 (80 megabyte)

## 7.2.10 READ DATA FROM PIOP DISK TO AP MD (RDPPDK)

**PURPOSE:** To cause the PIOP to read a block of data from the PIOP disk into AP120B main data memory.

**FORTRAN CALL:** CALL RDPPDK(MDA,LR,N,WKA,MODE)

**PARAMETERS:** MDA = Base address (MD) for data from disk  
LR = Logical record on disk where data begins  
N = word count (MD words)  
WKA = Base address (MD) of 16-word work buffer  
Words 0-7 - Channel status buffer (see PCGO)  
" 8-19 - Channel program  
MODE= Exit mode flag  
MODE = 0 - Exit after PIOP halts  
= 1 - Exit after PIOP starts

**FORMULA:** N/A

**DESCRIPTION:** Causes the PIOP to read a block of N words into AP main data memory, beginning at address MDA, from the PIOP disk, beginning at logical record LR. The format by which data is transferred from the disk must have been specified by a call to INPPDK prior to calling RDPPDK. A logical record is defined to be 256 AP main data words long. Depending on the format, either 1, 2, or 3 16-bit disk words will be stored into each AP main data word. The physical disk location of logical record 0 is defined by INPPDK.

The routine sets up a 8-word channel program in the 16-word work area, and starts the PIOP disk channel interpreter program DKPIOC which initiates the data transfer. If MODE=0, the RDPPDK will wait until the PIOP stops (transfer complete or an error condition) before returning to the calling program. If MODE=1, RDPPDK will return to the calling program immediately after starting the PIOP.

**EXAMPLE:** CALL DKPIOC(0,0,2,ISIZE)  
CALL INPPDK(0,1,0,1028,69,100)  
CALL RDPPDK(1000,0,500,100,0)  
The call to DKPIOC brings the PIOP disk channel interpreter program through AP120B main data locations 0-(ISIZE-1) and into PIOP program memory locations 0-(ISIZE-1). INPPDK specifies that disk port 1, cylinder 4, head 2, sector 5 will define logical record 0 for subsequent read/write calls to the disk (e.g., RDPPDK, WRPPDK). A CDC 9762 (80 megabyte)

7.2.10 READ DATA FROM PIOP DISK TO AP MD (RDPPDK) (cont.)

disk is specified. Subsequent data transfers will be made in format 1, whereby 38-bit AP words are transferred to/from the disk as 3 16-bit words (low mantissa, high mantissa, and exponent). For this format a logical record occupies 3 256-word disk sectors. Thus logical record 1 would refer to port 1, cylinder 4, head 2, sector 8. RDPPDK then causes the PIOP to read 500 38-bit words (1500 16-bit disk words) from the disk beginning at logical record 0, and stores them into AP main data locations 1000-1499. Main data locations 100-115 are used as a work area by RDPPDK. RDPPDK waits until the PIOP stops and returns the error status in SP(15) and in AP locations 100-107. If the data is to be immediately transferred to the host without an intervening AP processing call (e.g. VADD, RFFT, etc.), then CALL APWR should follow RDPPDK.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 6.0  | 6.0     | 6.0   | N/A (167 ns memory) |
| (us)      | 6.5  | 6.5     | 6.5   | N/A (333 ns memory) |

(Execution time is for MODE=1)

PROGRAM SIZE: 42 (167 ns memory)  
 (AP words) 42 (333 ns memory)

---

APAL CALL: JSR RDPPDK  
 SCRATCH: SP(0,1,14,15),DPX(0)  
 EXTERNALS: PCGO,PCSTAT

## 7.2.11 WRITE DATA FROM AP MD TO PIOP DISK (WRPPDK)

PURPOSE: To cause the PIOP to write a block of data from AP120B main data memory to the PIOP disk.

FORTRAN CALL: CALL WRPPDK(MDA,LR,N,WKA,MODE)

PARAMETERS: MDA = Base address (MD) for data for disk  
LR = Logical record on disk where data begins  
N = Word count (MD words)  
WKA = Base address (MD) of 16-word work buffer  
    words 0-7 - Channel status buffer (see PCGO)  
    words 8-19 - Channel program  
MODE = Exit mode flag  
    MODE = 0 - Exit after PIOP halts  
    MODE = 1 - Exit after PIOP starts

FORMULA: N/A

DESCRIPTION: Causes the PIOP to write a block of N words from AP main data memory, beginning at address MDA, into the PIOP disk, beginning at logical record LR. The format by which data is transferred to the disk must have been specified by a call to INPPDK prior to calling WRPPDK. A logical record is defined to be 256 AP main data words long. Depending on the format, either 1, 2, or 3 16-bit disk words will be stored for each AP main data word. The physical disk location of logical record 0 is defined by INPPDK.

The routine sets up a 8-word channel program in the 16-word work area, and starts the PIOP disk channel interpreter program DKPIOC which initiates the data transfer. If MODE=0, the WRPPDK will wait until the PIOP stops (transfer complete or an error condition) before returning to the calling program. If MODE=1, WRPPDK will return to the calling program immediately after starting the PIOP.

EXAMPLE: CALL DKPIOC(0,0,2,ISIZE)  
CALL INPPDK(0,1,0,1028,69,100)  
CALL WRPPDK(1000,0,500,100,0)  
The call to DKPIOC brings the PIOP disk channel interpreter program through AP120B main data locations 0-(ISIZE-1) and into PIOP program memory locations 0-(ISIZE-1). INPPDK specifies that disk port 1, cylinder 4, head 2, sector 5 will define logical record 0 for subsequent read/write calls to the disk (e.g., RDPPDK, WRPPDK). A CDC 9762 (80 megabyte)

7.2.11 WRITE DATA FROM AP MD TO PIOP DISK (WRPPDK) (cont.)

disk is specified. Subsequent data transfers will be made in format 1, whereby 38-bit AP words are transferred to/from the disk as 3 16-bit words (low mantissa, high mantissa, and exponent). For this format a logical record occupies 3 256-word disk sectors. Thus logical record 1 would refer to port 1, cylinder 4, head 2, sector 8. WRPPDK then causes the PIOP to write 500 38-bit words (1500 16-bit disk words) to the disk beginning at logical record 0, from AP main data locations 1000-1499. Main data locations 100-115 are used as a work area by WRPPDK. WRPPDK waits until the PIOP stops and returns the error status in SP(15) and in AP locations 100-107.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 6.0  | 6.0     | 6.0   | N/A (167 ns memory) |
| (us)      | 6.5  | 6.5     | 6.5   | N/A (333 ns memory) |

(Execution time is for MODE=1)

|               |    |                 |
|---------------|----|-----------------|
| PROGRAM SIZE: | 42 | (167 ns memory) |
| (AP words)    | 42 | (333 ns memory) |

-----

APAL CALL: JSR WRPPDK  
 SCRATCH: SP(0,1,14,15),DPX(0)  
 EXTERNALS: PCGO,PCSTAT

## 7.2.12 WRITE TO AND READ FROM PIOP DISK (WRDPPD)

**PURPOSE:** To cause the PIOP to write a block of data to the PIOP disk from AP120B main data memory, and then to read a (possibly different) block of data from the disk into AP120B main data memory.

**FORTRAN CALL:** CALL WRDPPD(MDAW,LRW,NW,MDAR,LRR,NR,WKA,MODE)

**PARAMETERS:**

- MDAW = Base address (MD) for data for disk write
- LRW = Logical record on disk for disk write
- NW = Word count (MD words) for disk write
- MDAR = Base address (MD) for data from disk read
- LRR = Logical record on disk for disk read
- NR = Word count (MD words) for disk read
- WKA = Base address (MD) of 20-word work buffer
  - Words 0-7 - Channel status buffer (see PCGO)
  - " 8-19 - Channel program

MODE= Exit mode flag

- MODE = 0 - Exit after PIOP halts
- = 1 - Exit after PIOP starts

**FORMULA:** N/A

**DESCRIPTION:** Causes the PIOP to write a block of NW words from AP main data memory, beginning at address MDAW, into the PIOP disk, beginning at logical record LRW, and then read a block of NR words into AP main data memory, beginning at address MDAR, from the disk, beginning at logical record LRR. The format by which data is transferred to and from the disk must have been specified by a call to INPPDK prior to calling WRDPPD. A logical record is defined to be 256 AP main data words long. Depending on the format, either 1, 2, or 3 16-bit disk words will be stored for each AP main data word. The physical disk location of logical record 0 is defined by INPPDK.

The routine sets up a 12-word channel program in the 20-word work area, and starts the PIOP disk channel interpreter program DKPIOC which initiates the data transfer. If MODE=0, the WRDPPD will wait until the PIOP stops (transfer complete or an error condition) before returning to the calling program.

7.2.12 WRITE TO AND READ FROM PIOP DISK (WRDPPD) (cont.)

If MODE=1, WRDPPD will return to the calling program immediately after starting the PIOP.

EXAMPLE:

```
CALL DKPIOC(0,0,2,ISIZE)
CALL INPPDK(0,1,0,1028,69,100)
CALL WRDPPD(1000,0,500,2000,6,800,100,1)
```

The call to DKPIOC brings the PIOP disk channel interpreter program through AP120B main data locations 0-(ISIZE-1) and into PIOP program memory locations 0-(ISIZE-1). INPPDK specifies that disk port 1, cylinder 4, head 2, sector 5 will define logical record 0 for subsequent read/write calls to the disk (e.g., RDPPDK, WRDPPD). A CDC 9762 (80 megabyte) disk is specified. Subsequent data transfers will be made in format 1, whereby 38-bit AP words are transferred to/from the disk as 3 16-bit words (low mantissa, high mantissa, and exponent). For this format a logical record occupies 3 256-word disk sectors. Thus logical record 1 would refer to port 1, cylinder 4, head 2, sector 8. WRDPPD then causes the PIOP to write 500 38-bit words (1500 16-bit disk words) to the disk beginning at logical record 0, from AP main data locations 1000-1499, and to read 800 38-bit words (2400 16-bit disk words) from the disk beginning at logical record 6 (port 1, cylinder 4, head 2, sector 23) into main data locations 2000-2799. After starting the PIOP, WRDPPD returns immediately to the calling program. Main data locations 100-119 are used as a work area by WRDPPD. The channel status can be examined later by calling PCSTAT, or by reading main data locations 100-107.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 6.7  | 6.7     | 6.7   | N/A (167 ns memory) |
| (us)      | 7.7  | 7.7     | 7.7   | N/A (333 ns memory) |

(Execution time is for MODE=1)

PROGRAM SIZE: 46 (167 ns memory)  
 (AP words) 46 (333 ns memory)

---

APAL CALL: JSR WRDPPD  
 SCRATCH: SP(0,1,14,15),DPX(0)  
 EXTERNALS: PCGO,PCSTAT

### 7.2.13 START PIOP CHANNEL (PCGO)

**PURPOSE:** To start running a PIOP Channel (PIOC) program.

**FORTRAN CALL:** CALL PCGO(A,S)

**PARAMETERS:** A = Base address of channel program  
in AP MD  
S = Base address of 8-word channel status  
buffer in AP MD

**FORMULA:** N/A

**DESCRIPTION:** Starts running the PIOP Programmable I/O Channel (PIOC) interpreter (DKPIOC or GPIOC), which interprets and executes a channel program located in AP1208 main data memory, beginning at address A. A channel program is written as a series of channel instructions, each of which contains information about the PIOP operation to be performed and appropriate parameters needed to carry out the operation.

Status information regarding the channel is returned by the interpreter in a 8-word buffer beginning at address S in main data. Following termination of the channel program, the routine PCSTAT can be used to determine if the program was completed successfully. If an error occurred, the status information can be obtained by looking in the 8-word buffer. Status information is PIOP configuration dependent.

For the PIOP disk interface configuration:

S(0) = 0 if channel program was completed, or  
= Address (MD) of channel instruction  
where error occurred  
S(1) = Disk controller error register  
S(2) = " " seek status register  
S(3) = " " port/cylinder address register  
S(4) = " " head/sector address register  
S(5) = " " control register  
S(6) = " " word count  
S(7) = (unused)

Prior to execution of PCGO, the PIOC interpreter program (DKPIOC or GPIOC) must be loaded into the PIOP program memory, and the channel program must be in AP1208 main data memory, beginning at address A.

Operation of the PIOC is described in detail in the PIOP Manual (FPS-7350), chapter 6.

7.2.13 START PIOP CHANNEL (PCGO) (cont.)

EXAMPLE:      CALL DKPIOC(0,0,2,LINT)  
                   CALL CHANPG(1000,0,0,LCH)  
                   CALL PCGO(1000,992)  
                   CALL PCSTAT(992)  
                   CALL APCHK(I)

The call to DKPIOC brings the PIOP disk channel interpreter program through AP120B main data locations 0-(LINT-1) and into PIOP program memory locations 0-(LINT-1). Suppose the channel program to be executed is called CHANPG. (Channel programs are normally written in using PPAL -- see PIOP Manual, section 6.3.) Then the call to CHANPG loads the channel program into AP main data locations 1000-(1000+LCH-1). Then PCGO causes the PIOP to execute the channel program CHANPG. AP main data locations 992-999 are used to return channel status information. PCSTAT waits for the PIOP to stop, and sets SP(15) to indicate if channel errors occurred. APCHK reads SP(15) and sets integer variable I to 0 if no channel errors were detected, or to the address of the channel instruction where errors occurred.

| EXECUTION     | BEST | TYPICAL | WORST | SETUP(us)           |
|---------------|------|---------|-------|---------------------|
| TIME:         | 3.7  | 3.7     | 3.7   | N/A (167 ns memory) |
| (us)          | 3.7  | 3.7     | 3.7   | N/A (333 ns memory) |
| PROGRAM SIZE: | 22   |         |       | (167 ns memory)     |
| (AP words)    | 22   |         |       | (333 ns memory)     |

-----  
 APAL CALL:      JSR PCGO  
 SCRATCH:        SP(0,14,15),DPX(3)  
 EXTERNALS:      PPWAIT

## 7.2.14 GET PIOP CHANNEL STATUS (PCSTAT)

**PURPOSE:** To determine whether a PIOP channel program has been successfully run.

**FORTRAN CALL:** CALL PCSTAT(S)

**PARAMETERS:** S = Base address of 8-word channel status buffer in AP MD

**FORMULA:** SP(15) = S(0) = 0 if channel program completed, or  
= address (MD) of channel instruction where error occurred

**DESCRIPTION:** Waits for PIOP to stop running and then sets SP(15) to indicate channel status. If the channel program has been completed successfully, then SP(15)=0, if not, then SP(15) equals the address of the channel instruction being executed by the PIOP when the error occurred. If an error occurred, the status information can be obtained by looking in the 8-word buffer. Status information is PIOP configuration dependent.

For the PIOP disk interface configuration:

S(0) = 0 if channel program was completed, or  
= Address (MD) of channel instruction where error occurred

S(1) = Disk controller error register  
S(2) = " " seek status register  
S(3) = " " port/cylinder address register  
S(4) = " " head/sector address register  
S(5) = " " control register  
S(6) = " " word count  
S(7) = (unused)

**EXAMPLE:** CALL DKPIOC(0,0,2,LINT)  
CALL CHANPG(1000,0,0,LCH)  
CALL PCGO(1000,992)  
CALL PCSTAT(992)  
CALL APCHK(I)

The call to DKPIOC brings the PIOP disk channel interpreter program through AP120B main data locations 0-(LINT-1) and into PIOP program memory locations 0-(LINT-1). Suppose the channel program to be executed is called CHANPG. (Channel programs are normally written in using PPAL -- see PIOP Manual, section 6.3.) Then the call to CHANPG loads the channel program into AP main data locations 1000-(1000+LCH-1). Then PCGO causes the PIOP to execute the channel program CHANPG. AP main data locations 992-999 are used to return channel status information. PCSTAT waits for the PIOP to stop,

7.2.14 GET PIOP CHANNEL STATUS (PCSTAT) (cont.)

and sets SP(15) to indicate if channel errors occurred. APCHK reads SP(15) and sets integer variable I to 0 if no channel errors were detected, or to the address of the channel instruction where errors occurred.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | ##   | ##      | ##    | N/A (167 ns memory) |
| (us)      | ##   | ##      | ##    | N/A (333 ns memory) |

(Execution time depends on when the PIOP halts. The routine is completed within 3.2 us after the PIOP halts.)

|               |    |  |  |                 |
|---------------|----|--|--|-----------------|
| PROGRAM SIZE: | 15 |  |  | (167 ns memory) |
| (AP words)    | 15 |  |  | (333 ns memory) |

---

APAL CALL: JSR PCSTAT  
 SCRATCH: SP(14)  
 EXTERNALS: PPWAIT

## 7.2.15 PIOP EXECUTE LOADER (PEXEC)

**PURPOSE:** To execute one of the following options with respect to PIOP programs in main data memory: 1) Load the program into PIOP program memory, 2) Load the program into PIOP program memory and start the PIOP program, 3) start the PIOP channel interpreter.

**FORTRAN CALL:** CALL PEXEC(MDA,PPSA,FLAG,N)

**PARAMETERS:** MDA = Base address (MD) of PIOP code  
PPSA = Base address (PIOP) for PIOP code  
FLAG = Loading sequence flag  
      =1 - Load 38-bit words from AP MD into PIOP PS  
      =2 - Load 38-bit words from AP MD into PIOP PS and start PIOP  
      =3 - Start PIOP channel interpreter using 38-bit words in AP MD as channel program (IOCL)  
N = Word count (38-bit words)

**FORMULA:** N/A

**DESCRIPTION:** This routine performs one of three loading options on 38-bit PIOP instruction words stored in AP1208 main data memory, beginning at address MDA. If FLAG=1, the instruction words are loaded into PIOP program memory, beginning at address PPSA. If FLAG=2, the instruction words are loaded into PIOP program memory, beginning at address PPSA, and the PIOP is started at program address PPSA. If FLAG=3, the instruction words are treated as a PIOP channel program I/O command list (IOCL), and the PIOP channel interpreter is started. Then the interpreter, which must be loaded prior to the call to PEXEC, begins executing the channel program at AP1208 main data address MDA.

Every PIOP program which is assembled by the PIOP assembler (PPAL) into a Fortran subroutine contains a CALL PEXEC statement. The calling sequence for the Fortran subroutine is:

CALL name (MDA,PPSA,FLAG,N)

The subroutine, when executed, will load the 38-bit program words from the host into AP1208 main data memory, beginning at address MDA. The number of words loaded, N, is passed back to the calling program, and to PEXEC, which is called with parameters MDA, PPSA, FLAG, and N, if FLAG equals 1, 2, or 3.

7.2.15 PIOP EXECUTE LOADER (PEXEC) (cont.)

EXAMPLE:       CALL PEXEC(100,50,2,40)  
              Loads the PIOP program stored in AP1208  
              main data locations 100-139 into PIOP  
              program memory locations 50-89, and starts  
              the PIOP running at program memory location  
              50. Control returns to the calling program  
              immediately after the PIOP starts.

| EXECUTION | BEST | TYPICAL | WORST | SETUP(us)           |
|-----------|------|---------|-------|---------------------|
| TIME:     | 1.7  | 1.7     | 1.7   | N/A (167 ns memory) |
| (us)      | 1.7  | 1.7     | 1.7   | N/A (333 ns memory) |

(Add execution time for PPLOAD if FLAG=1, PPLOAD and  
PPGO if FLAG=2, or PCGO if FLAG=3.)

|               |    |                 |
|---------------|----|-----------------|
| PROGRAM SIZE: | 70 | (167 ns memory) |
| (AP words)    | 70 | (333 ns memory) |

---

APAL CALL:       JSR PEXEC  
SCRATCH:        SP(0-2,4,5,14,15),DPX(0),MD  
EXTERNALS:       PPLOAD,PPGO,PCGO

### 7.3 SAMPLE PROGRAMS

As an aid to the programmer, two sample Fortran programs are presented in the following paragraphs. These programs are:

|   |   |
|---|---|
| Fortran subroutine  | Provides the following sample listings: <ul style="list-style-type: none"><li>a. PIOP assembly code source</li><li>b. Assembled listing</li><li>c. Fortran subroutine created by the PIOP assembler</li></ul> |
| Fortran program to run both the AP and the PIOP with a disk | Provides the following: <ul style="list-style-type: none"><li>a. Program listing</li><li>b. Timing explanation for the program (Figure 7-1)</li></ul>   |

#### 7.3.1 FORTRAN SUBROUTINE EXAMPLE

The following three examples of a Fortran subroutine are: PIOP assembly code source listing, assembled listing, and the subroutine listing as created by the PIOP assembler.

##### Example 1 - PIOP Assembly Code Source

```
$SUB TEST

$SUB TEST
LAB0: PASSG; "COMMENT
LAB1: WORD 2 "NUMBER 1
LAB2: SETMAR; "NUM 2
LAB3: WORD 2; "NUM 3
LAB4: PASSA 12..1 "NUM 4
LAB5: TR FF,FF; "NUM 5
LAB6: PASSA 12,10.; "NUM 6
LAB7: IORST; "NUM 7
LAB8: SETMAW "NUM 8
LAB9: JMPA X0A6; "NUM 9
LAB10: PASSG "NUM 10
SYM $EQU 10
MOVE $LOC SYM
$VAL ..2,$SYM
SEND
```

Example 2 - Assembled Listing

PASS 1

PASS 1

PASS 2

\$SUB TEST

```
0000 000000 LAB3: PASSG; "COMMENT
      0003E0 LAB1: WORD 2 "NUMBER 1
      020000

0001 000414 LAB2: SETMAR; "NUM 2
      000560 LAB3: WORD 2; "NUM 3
      020000 LAB4: PASSA 12.,1 "NUM 4

0002 001012 LAB5: TR FF,FF; "NUM 5
      0051E4 LAB6: PASSA 12,10.; "NUM 6
      100030 LAB7: IORST; "NUM 7
      LAB8: SETMAW "NUM 8

0003 000000 LAB9:JMFA XCAB; "NUM 9
      000360 LAB10: PASSG "NUM 10
      0016B3

      000010 SYM $EQU 10

0010          MOVE $LOC SYM

0010 000010 $VAL .,2,SYM
      000002
      000010
```

\$END

\*\*\*\* 0 ERRORS \*\*\*\*

| SYMBOL | VALUE  |
|--------|--------|
| LAB0   | 000000 |
| LAB1   | 000000 |
| LAB2   | 000001 |
| LAB3   | 000001 |
| LAB4   | 000001 |
| LAB5   | 000002 |
| LAB6   | 000002 |
| LAB7   | 000002 |
| LAB8   | 000002 |
| LAB9   | 000003 |
| LAB10  | 000003 |
| SYM    | 000010 |
| MOVE   | 000010 |

Example 3 - Fortran Subroutine Created by PIOP Assembler

```
SUBROUTINE TEST(MDADR,PPSA,FLAG,SIZE)

SUBROUTINE TEST(MDADR,PPSA,FLAG,SIZE)
C
INTEGER PPSA,FLAG,SIZE,PIECE(3)
REAL CODE(9,3)
C
DATA CODE(1,1),CODE(1,2),CODE(1,3)/0.,240.,8192./
DATA CODE(2,1),CODE(2,2),CODE(2,3)/268.,368.,8192./
DATA CODE(3,1),CODE(3,2),CODE(3,3)/522.,2676.,32792./
DATA CODE(4,1),CODE(4,2),CODE(4,3)/0.,240.,939./
DATA CODE(5,1),CODE(5,2),CODE(5,3)/0.,0.,0./
DATA CODE(6,1),CODE(6,2),CODE(6,3)/0.,0.,0./
DATA CODE(7,1),CODE(7,2),CODE(7,3)/0.,0.,0./
DATA CODE(8,1),CODE(8,2),CODE(8,3)/0.,0.,0./
DATA CODE(9,1),CODE(9,2),CODE(9,3)/8.,2.,8./
M=MDADR-1
SIZE=9
DO 20 I=1,9
DO 10 J=1,3
10  PIECE(J)=IPFIX(CODE(I,J))
CALL APDEP(PIECE,14,M+I)
20  CONTINUE
IF (FLAG.LE.0 .OR. FLAG.GT.3) RETURN
CALL PEXEC(MDADR,PPSA,FLAG,SIZE)
RETURN
END
```

### 7.3.2 FORTRAN PROGRAM EXAMPLE

The following is a sample Fortran program that runs both the AP and the PIOP with a disk. This program performs a block FFT (fast Fourier transform). An explanation of program timing is given in Figure 7-1.

```
      SUBROUTINE BLKFFT(N,M)
C
C  ROUTINE TO DEMONSTRATE USE OF AP120B WITH PIOP DISK INTERFACE AND
C  FORTRAN CALLABLE PIOP DISK CHANNEL SOFTWARE.
C
C  THIS ROUTINE READS AN N BY N ARRAY OF COMPLEX FLOATING POINT NUMBERS, M
C  COLUMNS AT A TIME, COMPUTES THE COMPLEX FFT OF EACH COLUMN, AND WRITES THE
C  FFT RESULTS BACK ONTO THE DISK.  THUS AT THE COMPLETION OF THE ROUTINE
C  THE N BY N COMPLEX ARRAY HAS BEEN REPLACED BY THE N BY N COMPLEX ARRAY
C  RESULTING FROM TAKING THE FORWARD COMPLEX FFT OF EACH OF THE N COLUMNS
C  OF THE ORIGINAL ARRAY.  THE ROUTINE COULD BE USED TO PROCESS DATA
C  IN ONE-DIMENSION WHEN PERFORMING A 2-D FFT OPERATION.
C
C  CONDITIONS UPON ENTRY:
C
C      ORIGINAL N BY N COMPLEX ARRAY IS STORED ON DISK BEGINNING AT LOGICAL
C      RECORD 0
C
C      N = DIMENSION OF THE ARRAY
C      M = NUMBER OF COLUMNS TO BE PROCESSED IN AP IN ONE PASS
C
C  CONDITIONS UPON EXIT:
C
C      ORIGINAL ARRAY REPLACED WITH N BY N COMPLEX ARRAY RESULTING FROM
C      TAKING COMPLEX FFT OF EACH OF THE N COLUMNS
C
C  EXAMPLE:  SUPPOSE N=1024, SO THAT 1024 X 1024 X 2 = 2097152 FLOATING-POINT
C            WORDS RESIDE ON THE DISK.  IF WE CHOOSE M=8, I.E. DO 8 COMPLEX
C            1024-POINT FFTS ON EACH PASS, THEN
C            NWD =16384
C            NPASS= 128
C            NREC = 64
C
C  INITIALIZE AP120B
C      CALL APCLR
C  INITIALIZE PIOP
C      CALL PPRS
C  LOAD PIOP DISK CHANNEL INTERPRETER THRU AP MAIN DATA INTO PIOP
C      CALL DKPIOC(0,0,2,ISIZE)
C  DEFINE DATA FORMAT 1 (38-BITS IN 3 PARTS) ON 80 MEGABYTE (CDC 9762) DISK
C  DEFINE DISK LOGICAL RECORD 0 (PORT/CYLINDER 0, HEAD/SECTOR 0)
C  ONE LOGICAL RECORD EQUALS 256 AP WORDS.
C      CALL INPPDK(0,1,0,0,0,0)
C  INITIALIZE LOGICAL RECORD POINTERS FOR READING AND WRITING
C      LRR=0
C      LRW=0
C  NUMBER OF FLOATING-POINT NUMBERS TO BE READ EACH PASS
C      NPN=N+N
C      NWD=NPN*M
C  NUMBER OF LOGICAL RECORDS EACH PASS
C      NREC=NWD/256
```

Fortran Program Example (cont.)

```

C NUMBER OF PASSES
      NPASS=NPN/M
C ALLOCATE AP MAIN DATA FOR DOUBLE BUFFERING AND A WORK AREA
      IA=0
      IB=NWD
      IW=IB+NWD
C
C BEGIN PROCESSING BY READING FIRST BLOCK FROM DISK INTO BUFFER A
      CALL RDPPDK(IA,LRR,NWD,IW,1)
      LRR=LRR+NREC
C READ NEXT BLOCK INTO BUFFER B
      CALL RDPPDK(IB,LRR,NWD,IW,1)
      LRR=LRR+NREC
C
C NOW WE CAN GET INTO MAIN LOOP, WHICH WILL READ, PROCESS, AND WRITE
C TWO BLOCKS OF DATA--ONE IN BUFFER A, THE OTHER IN BUFFER B. PIOP
C AND AP OPERATIONS ARE OVERLAPPED TO A GREAT EXTENT.
C
      DO 100 I=1,NPASS,2
C
C PROCESS BUFFER A IN MD WITH AP
      CALL PROCES(IA,N,M)
C WRITE FFT RESULTS FROM BUFFER A TO DISK, READ NEW BLOCK INTO BUFFER A
      CALL WRDPPD(IA,LRW,NWD,IA,LRR,NWD,IW,1)
      LRW=LRW+NREC
      LRR=LRR+NREC
C PROCESS BUFFER B IN MD WITH AP
      CALL PROCES(IB,N,M)
C WRITE FFT RESULTS FROM BUFFER B TO DISK, READ NEW BLOCK INTO BUFFER B
      CALL WRDPPD(IB,LRW,NWD,IB,LRR,NWD,IW,1)
      LRW=LRW+NREC
      LRR=LRR+NREC
100  CONTINUE
C
C END OF PROCESSING LOOP
C
C WAIT FOR PIOP TO COMPLETE FINAL DATA TRANSFER
      CALL PPWAIT
      CALL APWR
C EXIT
      RETURN
      END

      SUBROUTINE PROCESS(IBUF,N,M)
C
C ROUTINE PERFORMS M N-POINT COMPLEX FFTS OF DATA IN AP MAIN DATA
C BEGINNING AT ADDRESS IBUF.
C
      LADR=IBUF
      DO 1 I=1,M
      CALL CFFT(LADR,N,1)
1    LADR=LADR+N+N
      RETURN
      END

```

Figure 7-1 illustrates the timing for the program given on the previous pages.

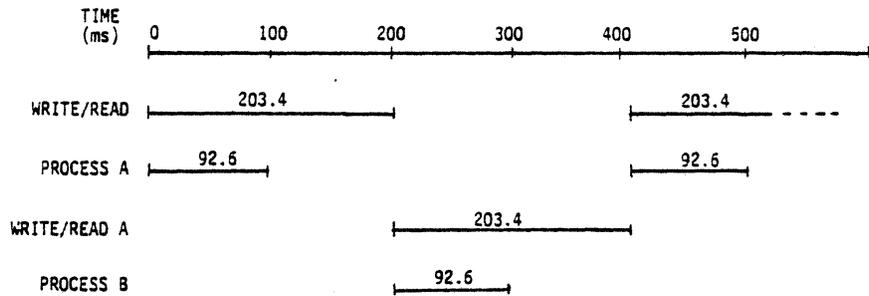
- ASSUMPTIONS:
1. Slow (333ns) main data memory.
  2. Disk transfer rate of 200,000 38-bit words/sec (600,000 16-bit words/sec).
  3. Disk cylinder-to-cylinder time of 7ms.
  4. Disk latency of 8.3ms
  5. Host system APEX overhead of 2ms per call.
  6. DMA cycle-stealing interference of 10%.

LET: N = 1024      1024-point CFFT in 8.7ms  
M = 8

|               |                      |               |
|---------------|----------------------|---------------|
| AP PROCESSING | 8 x CFFT             | = 69.6ms      |
| TIME:         | 10% DMA interference | = 7.0         |
|               | 8 x host overhead    | = 16.0        |
|               |                      | <u>92.6ms</u> |

|                    |                        |                |
|--------------------|------------------------|----------------|
| DMA TRANSFER RATE: | 2 x 1024 x 2 x 8 words | = 163.8ms      |
| (WRITE & READ)     | 1 x DMA interference   | = 7.0          |
|                    | 1 x host overhead      | = 2.0          |
|                    | 2 x disk access        | = 14.0         |
|                    | 2 x disk latency       | = 16.6         |
|                    |                        | <u>203.4ms</u> |

PROCESSING LOOP TIMING:



0091

Figure 7-1 Timing for Block FFT

## CHAPTER 8

### PIOP DEBUGGER - PPDEBUG

#### 8.1 INTRODUCTION

PPDEBUG provides an interactive facility for checking PIOP programs. PPDEBUG is initiated from within the AP debug program APDEBUG by the "J" command, which causes APDEBUG to call PPDEBUG. When the debugging session is complete, the "X" command returns control to APDEBUG. In this way, debugging of both AP and PIOP code can be accomplished from APDEBUG.

PPDEBUG has commands to:

1. Examine and/or change memory locations and registers inside the PIOP.
2. Examine and/or change AP main data memory locations.
3. Examine contents as program words, integers, or floating-point numbers.
4. Run PIOP programs.

## 8.2 OPERATING PROCEDURE

Debugging is the process of detecting, locating, and removing mistakes from a program. When the programmer wishes to debug a PIOP program, he loads the program into PPDEBUG. The user may then control program execution, causing the program to breakpoint at selected program locations so that he can examine the contents of registers or memory locations. Contents may be examined as program words, integers, or floating-point numbers.

PPDEBUG types a "\*" when ready for user input. An error message is typed when an error is detected.

## 8.3 MONITORING REGISTERS AND MEMORY LOCATIONS

Registers and memory locations in the PIOP may be opened, examined, and modified using one of the following commands:

- E open and examine locations in the PIOP (or AP main data memory)
- + examine the next higher location in a PIOP memory (or AP main data memory)
- examine the next lower location in a PIOP memory (or AP main data memory)
- C change the open location
- . re-examine the currently open location
- Z zeros out all PIOP memories

A register in the PIOP is opened with an "E" (exam), "+" (next), or "-" (last) command. PPDEBUG gets the value of the desired location in the PIOP and types out the value on the user console. If desired, the contents of that location may be changed with a "C" (change) command. A "." (re-examine) types the contents of the open register.

### 8.3.1 "E", Open and Examine

To open and examine a register in the PIOP:

```
E      (cr)
name (cr)
```

where NAME is the name of the desired register.

To open and examine a memory location in the PIOP:

```
E          (cr)
name       (cr)
location (cr)
```

where NAME is the memory name and LOCATION is the desired memory location.

A list of the examinable registers and memories is given in Appendix D, page D - 5. For the purposes of PPDEBUG, all functional units of the PIOP which have addresses are considered "memories." This includes PIOP program source memory, the PIOP ALU registers, as well as the AP main data memory.

Some examples:

1. Examine main data memory location 23.

```
*
E (cr)
MD (cr)
23 (cr)
-234.0000000
*
```

MD location 23 contains -234.0.

2. Examine the address register.

```
*
E (cr)
AR (cr)
40
*
```

MA contains 40.

### 8.3.2 "+", "-", and "." Examine Next, Last and Re-examine

To open and examine the next higher sequential memory location above a currently open memory location:

+ (cr)

To open and examine the next lower sequential memory location below a currently open memory location:

- (cr)

To re-examine the currently open memory location:

. (cr)

#### Examples:

1. Examine AP main data memory locations 23 and 24.

```
*
E (cr)
MD (cr)
23 (cr)
-234.000000
*
```

MD location 23 contains -234.0, now examine MD location 24.

```
*
+ (cr)
MD 000024
789.000000
*
```

MD location contains 789.0

2. Examine ALU registers 7 and 6.

```
*
E (cr)
SP (cr)
7 (cr)
000027
*
```

ALU register 7 contains 27. Now examine register 6.

```
*
- (cr)
SP 000006
-136
*
-
```

ALU register 6 contains -136.

### 8.3.3 "C", Change

To change the contents of a currently "open" register or memory location to a specified value:

```
C (cr)
value (cr)
```

where VALUE is an integer(s) or floating-point number (depending upon what register or memory is "open"). (See paragraph 9.4)

To change a register or MEMORY location, the user must first "open" it by doing an "E", "+", or "-" command.

Examples:

1. Examine AP main data memory location 20 and then change its value to -97.5.

```
*
E (cr)
MD (cr)
20 (cr)
76.0000000
*
C (cr)
-97.5 (cr)
*
-
```

Main data memory location 20 contained 76.0. The user changed it to contain -97.5.

2. Now change main data memory location 21 to -63.4.

```
*
+      (cr)
MD  000020
-3.000000000
*
C      (cr)
-63.4 (cr)
*
-
```

MD location 21 contained -3.0 and was changed to contain -63.4.

3. Examine ALU register 3 and change its value to 17.

```
*
E      (cr)
ALU (cr)
3      (cr)
56
*
C      (cr)
17     (cr)
*
-
```

ALU register 3 contained 56 and was changed to contain 17.

To examine locations 156 and 157 of PIOP program source memory,  
type:

```
*
E      (cr)
PS (cr)
156 (cr)
000400 000216 001507
*
+      (cr)
PS  000157
001700 000140 000000
*
-
```

#### 8.4 CHANGING INPUT/OUTPUT FORMATS

The input and output format used when examining and changing registers and memory locations may be selected using the following commands:

N sets the radix for integers

F sets the format for input/output of 38-bit wide registers and memory words

PPDEBUG selects the proper format for input/output depending upon the word size of the particular register or memory location that is open and the setting of the above two flags:

1. 16-bit words: (includes the 8-bit registers AR, FLAG, and PSA) These locations are examined or changed as integers in the radix as selected by "N".
2. 38-bit words: AP, main data memory, PIOP program source memory, etc. These locations are examined or changed as either floating-point numbers, or as three integers, depending upon the "F" flag. 20-bit registers, such as ALU and CR, are displayed in 38-bit format.

The listing of accessible PIOP registers and memories on page D - 5 specifies the width of each as:

16-bit (integer word)  
or 38-bit (floating-point word or program word)

#### NOTE

Integer output is always unsigned on the range 0-177777 (octal), or 0-65536 (decimal), or 0-FFFF (hexadecimal). Thus, negative two's complement numbers will be typed out as their 16-bit unsigned equivalent. For example (in octal), -1 would be output as 177777, and -2 as 177776, and so forth.

#### 8.4.1 "N" Set Radix

To set the radix for all integers input/output to PPDEBUG:

```
N      (cr)
radix (cr)
```

where the radix is either 8, 10, or 16 for octal, decimal, or hexadecimal radices, respectively. (Note that the radix number is always in decimal.)

The contents of 16-bit wide registers (AR, FLAG, and PSA) are examined and changed using the integer radix as set by the "N" command. In addition, memory addresses are also entered using the current radix.

On type-outs, octal numbers may be recognized as having six digits, decimal numbers as having five digits, and hex numbers as having four digits.

The default radix is either octal or hex depending upon the conventions of the host computer.

#### Examples:

1. Examine S-pad register 10 (decimal) in all three radices (starting in decimal).

```
*
E (cr)
AR (cr)
128
*
N (cr)
8 (cr)
*
. (cr)
200
*
N (cr)
16 (cr)
*
. (cr)
80
*
```

The value of the AR register is 256 (decimal), or 200 (octal), or 80 (hexadecimal).

#### 8.4.2 "F" Set/Reset Floating-Point I/O

To select floating-point input/output of 38-bit registers and memory words:

F (cr)  
1 (cr)

To select integer (in the current integer radix) input/output of 38-bit wide registers and memory locations:

F (cr)  
0 (cr)

38-bit wide registers are split into three pieces: 10-bit exponent, 12-bit high mantissa (bits 0-11), and 16-bit low exponent (bits 12-27) for integer I/O.

Main data memory, PIOP program source memory, and ALU registers are among the registers and memories whose I/O is governed by the "F" flag.

Both examining and changing of 38-bit registers are effected by the "F". The default setting of the "F" switch is 1 for floating-point I/O.

##### 1. Examine command output formats:

F=1: (floating-point number)

F=0: (exponent) (high mantissa) (low mantissa)

##### 2. Change command input formats:

F=1: C (cr)  
(floating-point number) (cr)

F=0: C (cr)  
(exponent) (cr)  
(high mantissa) (cr)  
(low mantissa) (cr)

Legal floating-point numbers are of the following form:

+or-XX.YYE+or-ZZ

where XX is the integer part  
YY is the fraction part  
ZZ is the exponent

Any of the three parts may be omitted, except in the case when an exponent is used. In this case, either an integer part or a fraction part must also be included. The signs may be omitted if "+" is used. The decimal point may be omitted if not needed. No spaces are allowed inside floating-point numbers.

The following are all legal floating-point inputs.

```
-2.3E6
 .7E-3
-2
 3.65
 .7
```

Examples:

1. Examine main data address six in both floating-point and integer. (Assume the integer radix is 16.)

```
*
E (cr)
MD (cr)
6 (cr)
-1.000000000
*
F (cr)
0 (cr)
*
. (cr)
MD 0006
0200 0400 0000
*
```

MD register six contains -1.0. Its exponent is 200 (hexadecimal) which has an exponent value of zero (0). The fraction part is 4000000 (hexadecimal) which is a fraction of -1.0

2. Now change the exponent to 204 and the fraction to 2000000 and set "F" to 1:

```
*  
C (cr)  
204 (cr)  
200 (cr)  
0 (cr)  
*  
F (cr)  
1 (cr)  
*  
.  
MD 0006  
8.000000000  
*-
```

## 8.5 MEMORY LOADING AND DUMPING

Blocks of memory locations may be loaded and dumped to and from files with the following commands:

```
Y yank (load) into a memory from a file
W write out the contents of a memory to a file
Z zero all the PIOP memories
```

The user should be aware that the procedure for typing in filenames varies greatly according to the respective system. In some systems, the notion of user files is nonexistent. In these cases, a logical unit number referring to an I/O device, which was opened previously by JCL control statements must be entered in place of a filename. Other systems allow access to disk files, line printers and terminals by symbolic names. Thus, what must be entered for a filename depends on the convention of each respective system. The examples given below are only meant to be representative and may not be legal on a given system.

### 8.5.1 "Y", Yank from a File

To load a memory from a file:

```
Y                (cr)
memory name      (cr)
starting location (cr)
filename         (cr)
```

where MEMORY NAME is an AP or PIOP memory, the beginning memory address is loaded at the STARTING LOCATION. The name of the file from which the data is to be read is called FILENAME. The filename must, of course, be in the proper form as determined by the particular host operating system.

Yank is typically used to load programs into PIOP program memory and data into AP main data memory. Some examples:

1. Load a program into PS location 0. The program is assumed to be in a file named MYPROG which was made using the output from PPAL.

```
*
Y                (cr)
PS                (cr)
0                 (cr)
MYPROG           (cr)
*
```

2. Load data into MD starting at location 20 from a file called DATA.

```
*  
Y      (cr)  
MD     (cr)  
20     (cr)  
DATA  (cr)  
*  
—
```

#### 8.5.2 "W", Write to a File

To write the contents of a memory into a file:

```
W              (cr)  
memory name   (cr)  
starting location (cr)  
ending location (cr)  
filename      (cr)
```

where MEMORY NAME is an AP or PIOP memory, STARTING LOCATION is the initial address to be written, ENDING ADDRESS is the last address to be written and FILENAME is the name of the file into which the data is to be written.

Some examples:

1. Write main data memory locations 20 through 40 into a file called DUMP.

```
*  
W      (cr)  
MD     (cr)  
20     (cr)  
40     (cr)  
DUMP  (cr)  
*  
—
```

2. Write main data locations 3 through 6 to the line printer (first, in floating-point format, and second, in integer format). (Strictly as an example, the line printer is called LP:.) Note that data pad may be dumped only from HWDEBUG.

```
*  
F (cr)  
1 (cr)  
*  
W (cr)  
MD (cr)  
3 (cr)  
6 (cr)  
LP: (cr)  
*  
F (cr)  
0 (cr)  
*  
W (cr)  
MD (cr)  
3 (cr)  
6 (cr)  
LP: (cr)  
*  
_
```

If the user mistypes a "W" command, he has several options to abort the command. If the wrong memory name or starting address was typed, then the command may be cancelled by entering an ending address (which is lower than the starting address). In PPDEBUG, an unwanted dump already underway (for example, when a location 1000 was typed, whereas location 100 was wanted) can be aborted by a USER BREAK. How this is accomplished varies from system to system. Typically, on single-user mini-computer systems, it is accomplished by raising the most significant bit of the host switch register.

### 8.5.3 "Z", Zero the AP

The "Z" command zeros out all the ALU registers and program source memory locations in the PIOP. This is accomplished by:

```
Z (cr)
```

### 8.5.4 Preparing Data Files for Yanking

Data files may be prepared by the user for loading into MD and PS by using PPDEBUG. The format of the data files is as follows:

```
data count  
data item #1  
data item #2  
... ..  
data item #N
```

All entries must be left justified, one entry per line.

The data count is the number of memory locations to be filled and written as a real number (with a decimal point). Thus, if there were three data items, the count would be "3".

The format is determined by the "F" switch setting for 38-bit memories. For integer formats, the radix is determined by the N (radix) setting. When floating point numbers are used they appear one per line. Also, integers must appear one per line in the file. Thus, for 38-bit memories written in integer format (F=0), three integers (exponent, high mantissa, low mantissa) written on three separate lines must be included for each memory location.

Some examples:

1. Four element floating point data file:

```
4.  
1.2  
.3  
-6E7  
2.3E-5
```

2. Three element integer data for a 38-bit wide memory which will load three integers into the low mantissa.

3  
0  
0  
1  
0  
0  
2  
0  
0  
3

## 8.6 EXECUTING PROGRAMS

PIOP program execution may be controlled with the following commands:

```
I initialize the PIOP
R run a PIOP program
X exit to the host operating system
```

### 8.6.1 "I", Initialize the PIOP

To initialize (reset) the AP:

```
I (cr)
```

In PPDEBUG, an interface reset is done to the AP. This is necessary to stop a program that has "run away."

### 8.6.2 "R", Run a PIOP Program

To run a PIOP program:

```
R      (cr)
location (cr)
R      (cr)
10     (cr)
*
_
```

PPDEBUG signals program return merely by a "\*".

### 8.6.3 "X", Exit to PPDEBUG

To complete a PIOP debugging session and exit to APDEBUG:

```
X (cr)
```

PPDEBUG types "EXIT PPDEBUG".

APPENDIX A

INSTRUCTION SET

This appendix presents the PIOP instruction set in tabular form as follows:

|           |           |                       |
|-----------|-----------|-----------------------|
| Table A-1 | (7 pages) | Basic Instruction Set |
| Table A-2 | (1 page)  | Expanded Instructions |
| Table A-3 | (1 page)  | Symbols for Table A-2 |
| Table A-4 | (2 pages) | Cross-Reference Set   |

Table A-1 PIOP Instructions

| FIELD | OCTAL CODE | MNEMONIC                  | MEANING   | DESCRIPTION   | SHORTHAND NOTATION | OTHER FIELDS (ARGUMENT) |
|-------|------------|---------------------------|---|---|--------------------|-------------------------|
| IOCMD | 0          | -                         | -   | No operation.   | -                  | -                       |
|       | 1          | SETMAR                    | <u>set memory address, read</u>   | Initiates a DMA read cycle to fetch data from the AP's main data memory at the address specified by the ALU output. Data is not available until 6 cycles later. The sequence is:<br><br>1. SETMAR instruction<br>2. MDCR2* true (request to AP DMA channel)<br>3. MDCA2 true (acknowledge from AP DMA channel)<br>4. WAIT<br>5. DCHO2 (loads data into FIFO input buffer)<br>6. FIFO<br>7. DATA AVAILABLE (If FIFO was empty) | Read @ APMA        | -                       |
|       | 2          | SETMAW                    | <u>set memory address, write</u>  | Initiates a DMA write cycle at the location specified by the ALU output. Data is written into the AP's main data memory. Data is available in memory after the third cycle. The sequence is:<br><br>1. SETMAW instruction<br>2. CYCLE REQUEST (data in FIFO output buffer taken)<br>3. CYCLE ACKNOWLEDGE (data now in memory)   | Write @ APMA       | -                       |
| 3     | SETDA      | <u>set device address</u> | Loads the device control register with data present on the ALU bus at the end of the instruction cycle. The device control register is a write-only register. | ALU > DVCMO   | -                  |                         |

0039

Table A-1 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC | MEANING                                   | DESCRIPTION   | SHORTHAND NOTATION  | OTHER FIELDS (ARGUMENT) |
|-------|------------|----------|---|---|---------------------|-------------------------|
| EXPAN | 0          | -        | -   | No operation.   | -                   | -                       |
|       | 1          | CF A     | <u>clear flag</u>                         | Clears the flag specified by A (A is specified in the BIT # field).   | Clear flag BIT #    | BIT #                   |
|       | 2          | RFF      | <u>reset FIFO</u>                         | Resets the FIFO pointers. Causes DATA VALID and FIFO FULL to go false (clear).<br><br>New data entering FIFO (through IN or SETMAR instructions or external handshake) falls through to the output buffer and causes DATA VALID to be true (set). | -                   | -                       |
|       | 3          | AFF      | <u>advance FIFO</u>                       | Advances FIFO read pointer. New data is written into FIFO output buffer at the end of the instruction cycle. If no valid words are in the FIFO, DATA VALID goes false (clear).  | -                   | -                       |
|       | 4          | SF x     | <u>set flag</u>                           | Sets the flag specified by x (x is specified in the BIT # field).   | Set flag BIT #      | BIT #                   |
|       | 5          | SINT x   | <u>set interrupt</u>                      | Sets interrupt x. The interrupt that is set executes in the second cycle after the SINT instruction.  | Set interrupt BIT # | BIT #                   |
|       | 6          | ENINT    | <u>enable interrupts</u>                  | Enables interrupt logic. Pending interrupts start executing on the next cycle.  | -                   | -                       |
|       | 7          | DISINT   | <u>disable interrupts</u>                 | Disables interrupt logic.   | -                   | -                       |
|       | 10         | NOP      | -   | No operation.   | -                   | -                       |
|       | 11         | START    | <u>start</u>                              | Begin program execution at current PSA location.  | Start               | -                       |
|       | 12         | HALT     | <u>halt</u>                               | Stop immediately. Nothing else in the instruction executes.   | PSA < PSA + 1       | -                       |
|       | 13         | PSAB     | <u>program source address, register B</u> | Causes the four least significant bits of the PSA to be used as ALU register address B. Can be used for sequential loading of ALU registers while PIOP is halted.   | PSA > 8             | -                       |
|       | 14         | TVCR x   | <u>transmit value to control register</u> | Transfers the value x (from VALUE field) on to the data bus and loads the control register (CR) with that value at the end of this cycle.   | VALUE > CR          | VALUE                   |
|       | 15         | TVOB x   | <u>transmit value to data bus</u>         | Transfers the value x (from VALUE field) on to the data bus.  | VALUE > DB          | VALUE                   |
|       | 16         | -        | -   | Not used.   | -                   | -                       |
|       | 17         | -        | -   | Indicates expanded ALU instruction format.  | -                   | ALU EXPAN               |

0040

Table A-1 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC  | MEANING  | DESCRIPTION   | SHORTHAND NOTATION  | OTHER FIELDS USED |
|-------|------------|-----------|--|---|---------------------|-------------------|
| A     | 1-17       | -         | -  | Contains address of one of 16 internal ALU registers.   | -                   | -                 |
| B     | 1-17       | -         | -  | Contains address of one of 16 internal ALU registers.   | -                   | -                 |
| ALU   | 0          | -         | -  | No operation.   | -                   | -                 |
|       | 1          | MOVD B    | <u>move data</u>                               | Move data bus contents to ALU register B. ALU output is that data.  | DB > B              | B                 |
|       | 2          | ADD A,B   | <u>add data</u>                                | Add the data bus contents to the contents of register A and store results in register B. ALU output is (DATA)+(A).                            | DB + A > B          | A,B               |
|       | 3          | ANDD A,B  | logical " <u>and</u> " of <u>data</u>          | Logically "and" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"and"(A).          | DB <u>and</u> A > B | A,B               |
|       | 4          | ORD A,B   | logical " <u>or</u> " of <u>data</u>           | Logically "or" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"or"(A).            | DB <u>or</u> A > B  | A,B               |
|       | 5          | XORD A,B  | logical " <u>exclusive or</u> " of <u>data</u> | Logically "exclusive or" the data bus contents with the contents of register A and store results in register B. ALU output is (DATA)"xor"(A). | DB <u>xor</u> A > B | A,B               |
|       | 6          | PASSD     | <u>pass data</u>                               | Data on data bus passes through the ALU unchanged and unsaved. The data appears on ALU outputs.   | DB > Y              | -                 |
|       | 7          | PASSA A,B | <u>pass register A</u>                         | Data in register A is gated to ALU outputs. Data in register B is written in to itself. PASSA is a fast ALU path.                             | A > Y<br>B > B      | A,B               |
|       | 10         | INCB B    | <u>increment register B</u>                    | Increment register B contents. ALU output is (B) + 1.   | B + 1 > Y           | B                 |
|       | 11         | DECB B    | <u>decrement register B</u>                    | Decrement the ALU register B contents. ALU output is (B) - 1.   | B - 1 > Y           | B                 |
|       | 12         | INCD      | <u>increment data bus</u>                      | Increment data on the data bus (D) and pass through the ALU (not saved).  | DB + 1 > Y          | -                 |
|       | 13         | DECD      | <u>decrement data bus</u>                      | Decrement data on the data bus (D) and pass through the ALU (not saved).  | DB - 1 > Y          | -                 |
|       | 14         | ADD A,B   | <u>add register A to register B</u>            | Add register A to register B, store the results in register B. ALU outputs = (A) + (B).   | A + B > Y           | A,B               |
|       | 15         | SUB A,B   | <u>subtract register A from register B</u>     | Subtract register A from register B and store results in register B. ALU outputs = (B) - (A).   | B - A > Y           | A,B               |
|       | 16         | PASSB     | <u>pass register B</u>                         | Pass register B contents unchanged on to the Y bus.   | B > Y               | B                 |
|       | 17         | PASSQ     | <u>pass register Q</u>                         | Pass Q register contents to ALU BUS (Y).  | Q > Y               | -                 |

Table A-1 PIOP Instructions (cont.)

| FIELD  | OCTAL CODE | MNEMONIC       | MEANING                               | DESCRIPTION  | SHORTHAND NOTATION | OTHER FIELDS USED |
|--|------------|----------------|---------------------------------------|--|--------------------|-------------------|
| TRANSFER (SRC)   | 0          | TR ALU, _      | arithmetic and logic unit             | Source of the data bus is the ALU output (Y).  | DB < Y             | -                 |
|  | 1          | TR (DISP8) , _ | displacement 8                        | Source of the data bus is the contents of the DISP8 field.                                       | DB < DISP8         | DISP8             |
| NOTE 1   | 2          | TR FF, _       | FIFO                                  | Source of the data bus is FIFO input buffer.   | DB < IR            | -                 |
|  | 3          | TR IOR, _      | input/output register                 | Source of the data bus is the contents of the I/O register.                                      | DB < IOR           | -                 |
|  | 4          | TR PSA, _      | program source address                | Source of the data bus is the program source address register.                                   | DB < PSA           | -                 |
|  | 5          | -              | -                                     | -  | -                  | -                 |
|  | 6          | TR CR, _       | control register                      | Source of the data bus is the contents of the control register.                                  | DB < CR            | -                 |
|  | 7          | -              | -                                     | Indicates that the SPEC (special) field is to be used as the next field in the instruction word. | GO TO SPEC         | -                 |
| TRANSFER (DST)   | 0          | -              | -                                     | No operation.  | -                  | -                 |
|  | 1          | -              | -                                     | -  | -                  | -                 |
| NOTE 2   | 2          | TR, FF         | FIFO                                  | Destination is the FIFO output buffer.   | DB > OB            | -                 |
|  | 3          | TR, IOR        | input/output register                 | Destination is the I/O register.   | DB > IOR           | -                 |
|  | 4          | TR, AR         | address register                      | Destination is the address register of the CPU.  | DB > AR            | -                 |
|  | 5          | -              | -                                     | -  | -                  | -                 |
|  | 6          | TR, CR         | control register                      | Destination is the control register.   | DB > CR            | -                 |
|  | 7          | -              | -                                     | -  | -                  | -                 |
| SPEC   | 0          | TR PS, IOR     | program source, input/output register | Transfers program source word into the I/O register (2-cycle instruction).                       | IOR < PS           | PSA CONTROL       |
|  | 1          | TR IOR, PS     | see above                             | Transfers contents of I/O register to program source (2-cycle instruction).                      | PS < IOR           | PSA CONTROL       |
| NOTES  |            |                |                                       |  |                    |                   |
| <p>1. Source loaded on data bus at <u>beginning</u> of cycle.</p> <p>2. Destination loaded on data bus at <u>end</u> of cycle.</p> |            |                |                                       |  |                    |                   |

0042

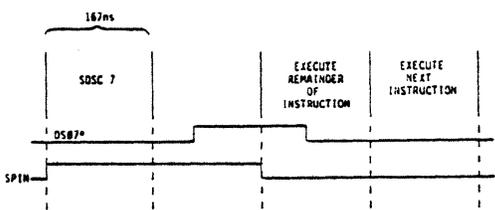
Table A-1 PIOP Instructions (cont.)

| FIELD       | OCTAL CODE | MNEMONIC  | MEANING                                 | DESCRIPTION  | SHORTHAND NOTATION                                   | OTHER FIELDS USED |
|-------------|------------|-----------|---|--|--|-------------------|
| PSA CONTROL | 0          | -         | -                                       | No Operation.  | -  | -                 |
|             | 1          | JMPAR     | <u>jump to address register</u>         | Absolute jump to address contained in the PIOP address register (AR). Address register can be loaded as a data bus destination. The contents of the register is the 8 LSB's of the data bus.<br><br>This instruction uses no other fields and is, therefore, useful for tight loops and computed GO TO's.  | PSA < AR   | -                 |
|             | 2          | JMPST     | <u>jump to stack</u>                    | Jump to address at top of stack. Does <u>not</u> change stack contents so is <u>not</u> a subroutine return instruction. This instruction uses no other fields.  | PSA < ST   | -                 |
|             | 3          | JMPA V    | <u>jump absolute</u>                    | Jump to absolute address V which is contained in the DISPB field.  | PSA + DISPb  | DISPB             |
|             | 4          | POP       | <u>pop the stack</u>                    | Advance subroutine return stack to the next address. This instruction does <u>not</u> change PSA.  | -  | -                 |
|             | 5          | PUSH      | <u>push the stack</u>                   | Enter the current address plus one in to the subroutine return stack. This instruction does <u>not</u> change the PSA.   | PSA + 1 > ST   | -                 |
|             | 6          | RTN       | <u>return</u>                           | Jump to address at the top of the stack and advance the stack to the next address (POP the stack).   | POP AND JMPST  | -                 |
|             | 7          | JSR V     | <u>jump to subroutine, relative</u>     | Jump by the relative location V as specified by the DISPB field. Enter the current location plus one into the stack.   | PSA < PSA + DISPB ,<br>PUSH                          | DISPB             |
|             | 10         | BDSC x, y | <u>branch if device status is clear</u> | If device status BIT # x is clear, branch relative as specified by y. The maximum displacement is +17 to -20 octal locations. If x=7, then a high level on DS07* was sampled at the beginning of this instruction. y may be specified as a relative argument.<br><br><u>NOTE</u><br><br>DS07* is one of eight sense lines (DS00* - DS07*) that allow the PIOP to be controlled externally.   |  | BIT #, DISPB      |
|             | 11         | BDSS x, y | <u>branch if device status is set</u>   | Same as above except BIT # x must be set for the branch to occur (DS07* line low if x=7).  |  | BIT #, DISPB      |
|             | 12         | BFC x, y  | <u>branch if flag clear</u>             | If flag BIT # x is clear, branch relative as specified by y (DISP5). The maximum displacement is +17 to -20 octal locations.   | If condition is true, then:<br><br>PSA < PSA + DISP5 | BIT #, DISP5      |
|             | 13         | BFS x, y  | <u>branch if flag set</u>               | Same as above, except branch occurs if flag is set.  | If condition is <u>not</u> true, then:               | BIT #, DISP5      |
|             | 14         | BISC x, v | <u>branch if ALU status is clear</u>    | If internal status BIT # x is clear (zero), branch as specified by y (DISP5). Maximum displacement is +17 to -20 octal locations.<br><br>Internal status BIT # is defined as follows:<br><br>If set: 0 = FIFO data valid<br>1 = FIFO full<br>2 = R shift out<br>3 = Q shift out<br>4 = ALU carry<br>5 = ALU zero<br>6 = ALU sign<br>7 = ALU overflow<br><br>Note that bits 2 through 7 above also appear in the control register (CR). | PSA < PSA + 1  | BIT #, DISP5      |

Table A-1 PIOP Instructions (cont.)

| FIELD       | OCTAL CODE | MNEMONIC  | MEANING                              | DESCRIPTION  | SHORTHAND NOTATION   | OTHER FIELDS USED |
|-------------|------------|-----------|--------------------------------------|--|--|-------------------|
| PSA CONTROL | 15         | BISS x, y | branch if ALU status is set          | <p>Same as BISC except that status must be set (1) for the branch to occur.</p> <p>These instructions are alternate mnemonics for the eight BISS and eight BISC mnemonics.</p> <p>BFV DISP Branch if FIFO data valid<br/>                     BFF DISP Branch if FIFO full<br/>                     BFOT DISP Branch if R-shift output = 1<br/>                     BQOT DISP Branch if Q-shift output = 1<br/>                     BC DISP Branch if carry set<br/>                     BZ DISP Branch if ALU=0<br/>                     BM DISP Branch if ALU is minus<br/>                     BOVF DISP Branch if overflow = 1</p> <p>BNFV DISP Branch if FIFO data <u>not</u> valid<br/>                     BNFF DISP Branch if FIFO <u>not</u> full<br/>                     BNFOT DISP Branch if R-shift output = 0<br/>                     BNQOT DISP Branch if Q-shift output = 0<br/>                     BNC DISP Branch if ALU carry out is 0<br/>                     BNZ DISP Branch if ALU is not 0<br/>                     BP DISP Branch if ALU is positive<br/>                     BNOVF DISP Branch if ALU overflow = 0</p> | <p>If condition is true, then:<br/>                     PSA &lt; PSA + DISP5</p> <p>If condition is <u>not</u> true, then:<br/>                     PSA &lt; PSA + 1</p> | BIT #, DISP5      |
|             | 16         | BNZST     | branch if ALU <u>not zero, stack</u> | <p>If ALU output is non-zero, branch to the location at the top of the stack. For example:</p> <p>TVOB 10; MOVD CNT<br/>                     PUSH<br/>                     DEC CNT<br/>                     BNZST<br/>                     HALT</p> <p>The above loops 10 times before halting.</p>  | PSA < ST   |                   |
|             | 17         | JMP X     | <u>jump</u>                          | <p>Jump unconditionally to the relative address specified by X.</p>  | PSA < PSA + DISP8  |                   |

Table A-1 PIOP Instructions (cont.)

| FIELD | OCTAL CODE | MNEMONIC | MEANING                                  | DESCRIPTION  | SHORTHAND NOTATION | OTHER FIELDS USED |
|-------|------------|----------|--|--|--------------------|-------------------|
| IO    | 0          | -        | -  | No operation.  | -                  | -                 |
|       | 1          | OUT      | <u>output</u>                            | Places FIFO output buffer contents on external device bus (DEV02* through DEV39*) and advances format logic. The format is specified by the FORMAT field in the control register.  | -                  | -                 |
|       | 2          | IN       | <u>input</u>                             | Loads the FIFO input buffer with data on the external device bus (DEV02* through DEV39*) at the end of the present cycle. This instruction also advances the format logic. The format is specified by the FORMAT field in the control register.  | -                  | -                 |
|       | 3          | IORST    | <u>input/output reset</u>                | Causes PIORST* (PIOP reset) to go true (low) which, by convention, resets all devices connected to the PIOP bus.   | -                  | -                 |
| SPIN  | BIT 37     | SDSC x   | <u>spin until device status is clear</u> | Spin until device status (BIT #) is clear. PIOP spins (waits) until device status line referenced by x (in BIT # field) is clear (high level) and then executes the remainder of that instruction. Device status state is sampled at the beginning of the instruction cycle.<br><br> <p>Only INT0 (interrupt 0) interrupts spins. If interrupted, the remainder of the instruction is not executed. Upon return from the interrupt, the next instruction is executed. The SPIN is <u>not</u> reentered.</p> | -                  | BIT #             |
|       | BIT 38     | SDSS x   | <u>spin until device status is set</u>   | Same as above, except the DS07* level is inverted.   | -                  | BIT #             |
|       | BIT 39     | SDAV     | <u>spin until data available</u>         | Spin until FIFO data is available. The PIOP spins (waits) until the FIFO contains valid data.<br><br>SETMAR; PASSB BUF; RFF<br>SDAV; TRFF, DB; WORD 3; MOVD 0; AFF<br><br>The above instruction sequence puts valid data from the AP's main data location (buffer) into ALU register 0 and then the AFF resets the data valid flag. The spin is a minimum of five cycles.  | -                  | -                 |

| INPUT/OUTPUT DATA FORMAT |      |        |                    |       |
|--------------------------|------|--------|--------------------|-------|
| FIELD                    | CODE | WORD   | DB TRANSFERS       | BITS  |
| WORD                     | 0    | WORD 0 | low mantissa (ML)  | 24-39 |
|                          | 1    | WORD 1 | high mantissa (MH) | 12-23 |
|                          | 2    | WORD 2 | exponent           | 2-11  |
|                          | 3    | WORD 3 | full word          | 2-39  |

Table A-2 PIOP Expanded Instructions

| FIELD  | OCTAL CODE | MNEMONIC | MEANING                  | DESCRIPTION  | SHORTHAND NOTATION                  | OTHER FIELDS USED |
|--------|------------|----------|--------------------------|--|-------------------------------------|-------------------|
| ALUSRC | 0          | AQ       | -                        | All of these codes are used to select the data source for the R and S input fields of the ALU.<br><br>Note that A and B fields are deferred. That is, the A (or B) field selects one of 16 registers. The contents of the selected register is then moved into either the S or R input field of the ALU. | $A > R, Q > S$                      | A                 |
|        | 1          | AB       | -                        |  | $A > R, B > S$                      | B                 |
|        | 2          | ZQ       | -                        |  | $\emptyset > R, Q > S$              | -                 |
|        | 3          | ZB       | -                        |  | $\emptyset > R, B > S$              | B                 |
|        | 4          | ZA       | -                        |  | $\emptyset > R, A > S$              | A                 |
|        | 5          | DA       | -                        |  | $DB > R, A > S$                     | A                 |
|        | 6          | DQ       | -                        |  | $DB > R, Q > S$                     | -                 |
|        | 7          | DZ       | -                        |  | $DB > R, \emptyset > S$             | -                 |
| ALUDST | 0          | Q        | Internal work register   | All of these codes are used to select the destination that is to receive the ALU output function.<br><br>Mnemonics are:<br>Q = internal work register<br>F = ALU function<br>Y = ALU output bus<br><br>Note that a right shift is a divide by 2 while a left shift is a multiply by 2.                   | $F > Q, F > Y$                      | ALUFCN            |
|        | 1          | NP       | -                        |  | $F > Y$                             | ALUFCN            |
|        | 2          | A        | A field                  |  | $F > B, A > Y$                      | A, ALUFCN         |
|        | 3          | F        | ALU function             |  | $F > B, F > Y$                      | B, ALUFCN         |
|        | 4          | RQ       | Right shift Q            |  | $F/2 > B, Q/2 > Q, F > Y$           | B, ALUFCN         |
|        | 5          | RF       | Right shift ALU function |  | $F/2 > B, F > Y$                    | B, ALUFCN         |
|        | 6          | LQ       | Left shift Q             |  | $2F > B, 2Q > Q, F > Y$             | B, ALUFCN         |
|        | 7          | LF       | Left shift ALU function  |  | $2F > B, F > Y$                     | B, ALUFCN         |
| ALUFCN | 0          | AD       | add                      | These codes are the function performed by the ALU.<br><br>R and S are ALU input operands.<br><br>The ALUSRC field selects the source for R and S; the ALUDST selects the destination for the ALU output after the selected function has been performed.  | $F = R + S + C$                     | ALUSRC, ALUDST    |
|        | 1          | SB       | subtract                 |  | $F = S - R$                         | ALUSRC, ALUDST    |
|        | 2          | SR       | subtract, reverse        |  | $F = R - S$                         | ALUSRC, ALUDST    |
|        | 3          | OR       | logical "or"             |  | $F = R \text{ or } S$               | ALUSRC, ALUDST    |
|        | 4          | AN       | logical "and"            |  | $F = R \text{ and } S$              | ALUSRC, ALUDST    |
|        | 5          | NA       | logical "nand"           |  | $F = \text{"not"} R \text{ and } S$ | ALUSRC, ALUDST    |
|        | 6          | XO       | exclusive "or"           |  | $F = R \text{ xor } S$              | ALUSRC, ALUDST    |
|        | 7          | XN       | "exclusive "nor"         |  | $F = \text{"not"} R \text{ xor } S$ | ALUSRC, ALUDST    |
| SH     | 0          | -        | default                  | Shift in zeros<br>Shift in ones.<br>Rotate (shift out becomes shift in)<br>Sign extend on right shift; fill with zeros on left shift.  | -                                   | -                 |
|        | 1          | N        | -                        |  | -                                   | -                 |
|        | 2          | R        | rotate                   |  | -                                   | -                 |
|        | 3          | A        | arithmetic shift         |  | -                                   | -                 |
| C      | 0          | -        | default                  | -  | $F = F$                             | ALUFCN            |
|        | 1          | I        | -                        | -  | $F = F + 1$                         | ALUFCN            |

Table A-3 Symbol Definitions

| SYMBOL | DESCRIPTION   |
|--------|---|
| A      | Register A - one of 16 internal registers (scratchpad memory of ALU). The specific register to be used is specified by a 4-bit binary number in the A field.  |
| B      | <p>Register B address - one of 16 internal registers (scratchpad memory of ALU). The specific register to be used is specified by a 4-bit binary number in the B field.</p> <p style="text-align: center;"><u>NOTE</u></p> <p>The same 16 registers are used by both A and B fields. For example, the A field may specify register #2 while the B field may specify register #14.</p> |
| DB     | Data Bus - the bi-directional bus connecting the transceiver to the other PIOP circuits. The mnemonic DB is also used for data bus.   |
| Q      | Register Q - an internal work register.   |
| R      | ALU Input Register R - one of two inputs to the ALU. Designates the left-hand input in a double-operand statement.  |
| S      | ALU Input Register S - One of two inputs to the ALU. Designates the right-hand input in a double-operand statement.   |
| Y      | ALU Output Bus Y - indicates the output bus of the ALU. More specifically, the output of the ALU Bus Select Logic.  |
| Z      | Represents binary 0's. For example, the expression Z > R indicates that all zeros are loaded into the ALU R input register.   |
| F      | Results of the ALU function which are applied to the ALU destination.   |

0047

Table A-4 Cross-Reference List

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION              |
|----------|--------|------------|---------------------------------|
| A        | ALUDST | 2          | $F > B, A > Y$                  |
| A        | SH     | 3          | -                               |
| AB       | ALUSRC | 1          | $A > R, B > S$                  |
| AD       | ALUFCN | 0          | $R + S + C$                     |
| ADD      | ALU    | 14         | $A + B > B$                     |
| ADDD     | ALU    | 2          | $DB + A > B$                    |
| AFF      | EXPAN  | 3          | -                               |
| AN       | ALUFCN | 4          | <u>R and S</u>                  |
| ANOD     | ALU    | 3          | <u>DB and</u> $A > B$           |
| AQ       | ALUSRC | 0          | $A > R, Q > S$                  |
| BDSC     | PSA    | 10         |                                 |
| BDSS     | PSA    | 11         | If condition is true, then:     |
| BFC      | PSA    | 12         | $PSA < PSA + DISP8$             |
| BFS      | PSA    | 13         | If condition is not true, then: |
| BISC     | PSA    | 14         | $PSA < PSA + 1$                 |
| BISS     | PSA    | 15         |                                 |
| BNZST    | PSA    | 16         | $PSA < ST$                      |
| CF       | EXPAN  | 1          | clear flag BIT #                |
| DA       | ALUSRC | 5          | $DB > R, A > S$                 |
| DB       | DST    | 0          | -                               |
| DECB     | ALU    | 11         | $B - 1 > B$                     |
| DECD     | ALU    | 13         | $DB - 1 > Y$                    |
| DISINT   | EXPAN  | 7          | -                               |
| DQ       | ALUSRC | 6          | $DB > R, Q > S$                 |
| DZ       | ASUSRC | 7          | $DB > R, \emptyset > S$         |

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION        |
|----------|--------|------------|---------------------------|
| ENINT    | EXPAN  | 6          | -                         |
| F        | ALUDST | 3          | $F > B, F > Y$            |
| HALT     | EXPAN  | 12         | Halt, $PSA < PSA + 1$     |
| IN       | IO     | 1          | -                         |
| INCB     | ALU    | 10         | $B + 1 > B$               |
| INCD     | ALU    | 12         | $DB + 1 > Y$              |
| IORST    | IO     | 7          | -                         |
| JMP      | PSA    | 17         | $PSA < PSA + DISP8$       |
| JMPA     | PSA    | 3          | $PSA < DISP8$             |
| JMPAR    | PSA    | 1          | $PSA < AR$                |
| JMPST    | PSA    | 2          | $PSA < ST$                |
| JSR      | PSA    | 7          | $PSA < PSA + DISP8, PUSH$ |
| LF       | ALUDST | 7          | $2F > B, F > Y$           |
| LQ       | ALUDST | 6          | $2F > B, 2Q > Q, F > Y$   |
| MOVD     | ALU    | 1          | $DB > B$                  |
| N        | SH     | 1          | -                         |
| NA       | ALUFCN | 5          | "not" R and S             |
| NP       | ALUDST | 1          | $F > Y$                   |

0048

Table A-4 Cross-Reference List (cont.)

| MNEMONIC | FIELD  | OCTAL CODE | SHORTHAND NOTATION      |
|----------|--------|------------|-------------------------|
| OR       | ALUFCN | 3          | R <u>or</u> S           |
| ORD      | ALU    | 4          | DB <u>or</u> A > B      |
| OUT      | IO     | 2          | -                       |
| PASSA    | ALU    | 7          | A > Y, B > B            |
| PASSB    | ALU    | 16         | B > Y                   |
| PASSD    | ALU    | 6          | DB > Y                  |
| PASSQ    | ALU    | 17         | Q > Y                   |
| POP      | PSA    | 4          | -                       |
| PSAB     | EXPAN  | 13         | PSA > B                 |
| PUSH     | PSA    | 5          | PSA + 1 > ST            |
| Q        | ALUDST | 0          | F > Q, F > Y            |
| R        | SH     | 2          | -                       |
| RF       | ALUDST | 5          | F/2 > B, F > Y          |
| RFF      | EXPAN  | 2          | -                       |
| RQ       | ALUDST | 4          | F/2 > B, Q/2 > Q, F > Y |
| RTN      | PSA    | 6          | POP and JMPST           |
| SB       | ALUFCN | 1          | S - R                   |
| SDAV     | SPIN   | -          | -                       |
| SDSC     | SPIN   | -          | -                       |
| SDSS     | SPIN   | -          | -                       |
| SETDA    | IOCMD  | 3          | ALU > DVCMD             |
| SETMAR   | IOCMD  | 1          | Read APMA               |
| SETMAW   | IOCMD  | 2          | Write APMA              |
| SF       | EXPAN  | 4          | Set flag BIT #          |
| SINDC    | IO     | 4          | -                       |
| SINDS    | IO     | 3          | -                       |
| SINT     | EXPAN  | 5          | Set interrupt BIT #     |
| SOTDC    | IO     | 6          | -                       |
| SOTDS    | IO     | 5          | -                       |
| SR       | ALUFCN | 2          | R - S                   |
| START    | EXPAN  | 11         | Start                   |
| SUB      | ALU    | 15         | B - A > B               |

| MNEMONIC     | FIELD   | OCTAL CODE | SHORTHAND NOTATION   |
|--------------|---------|------------|----------------------|
| TR ALU, --   | TR(SRC) | 0          | DB < Y               |
| TR,CR, --    | TR(SRC) | 6          | DB < (CR)            |
| TR(DISP8),-- | TR(SRC) | 1          | DB < (DISP8)         |
| TR FF, --    | TR(SRC) | 2          | DB < (IB)            |
| TR IOR, --   | TR(SRC) | 3          | DB < (IOR)           |
| TR IOR, PS   | SPEC    | 1          | PS < (IOR)           |
| TR PS, IOR   | SPEC    | 0          | (IOR) < PS           |
| TR PSA, --   | TR(SRC) | 4          | DB < (PSA)           |
| TR --, AR    | TR(DST) | 4          | DB > (AR)            |
| TR --, CR    | TR(DST) | 6          | DB > (CR)            |
| TR --, FF    | TR(DST) | 2          | DB > (OB)            |
| TR --, IOR   | TR(DST) | 3          | DB > (IOR)           |
| TVCR         | EXPAN   | 14         | VALUE > CR           |
| TVDB         | EXPAN   | 15         | VALUE > DB           |
| TVEX         | EXPAN   | 16         | VALUE > EXP          |
| WORD 0       | WORD    | 0          | -                    |
| WORD 1       | WORD    | 1          | -                    |
| WORD 2       | WORD    | 2          | -                    |
| WORD 3       | WORD    | 3          | -                    |
| XN           | ALUFCN  | 7          | "not" R <u>xor</u> S |
| XO           | ALUFCN  | 6          | R <u>xor</u> S       |
| XORD         | ALU     | 5          | DB <u>xor</u> A > B  |
| ZA           | ALUSRC  | 4          | Ø > R, A > S         |
| ZB           | ALUSRC  | 3          | Ø > R, B > S         |
| ZQ           | ALUSRC  | 2          | Ø > R, Q > S         |

## APPENDIX B

### PIOP INTERCONNECTIONS

Figure B-1 illustrates the lines and buses that connect the PIOP to the AP and that connect the PIOP to the external device. This diagram is not intended to be a complete schematic but, rather, is presented to aid the programmer in understanding how the PIOP communicates with the outside world.

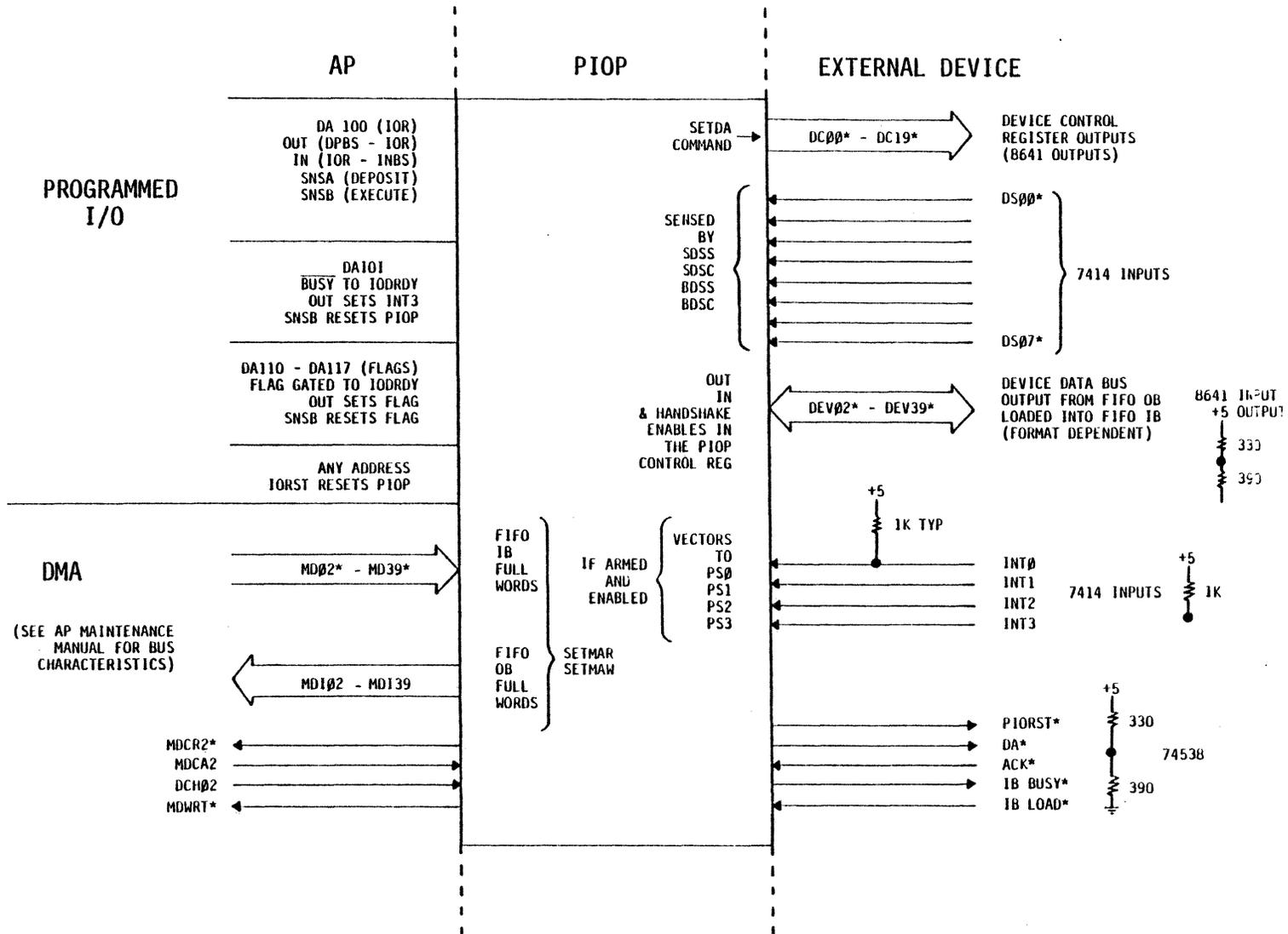


Figure B-1 PIOP Interconnection Diagram

## APPENDIX C

### SPECIAL STORAGE ELEMENTS

#### C.1 INTRODUCTION

Certain hardware elements are used to store data in a specific manner. One such element stores data on a first-in, first-out basis and is, therefore, referred to as a FIFO memory element. Another such element stores data on a last-in, first-out basis and is referred to as a "stack."

Both of these elements are used in the PIOP. One element is the FIFO memory which is part of the transceiver. The other element is the subroutine return stack. Each of these elements is discussed in a subsequent paragraph.

## C.2 FIFO MEMORY ELEMENT

A first-in, first-out (FIFO) memory element allows information to be retrieved in the same sequence as it was stored. Thus, a FIFO memory might be thought of as a buffer between elements that operate at different speeds. One element might load the memory slowly while another element might retrieve data from the memory in a high-speed transfer. A FIFO memory is often referred to as a "fall through" memory because data is entered at the top of the memory and is allowed to "fall through" to the bottom of the memory where it can then be retrieved.

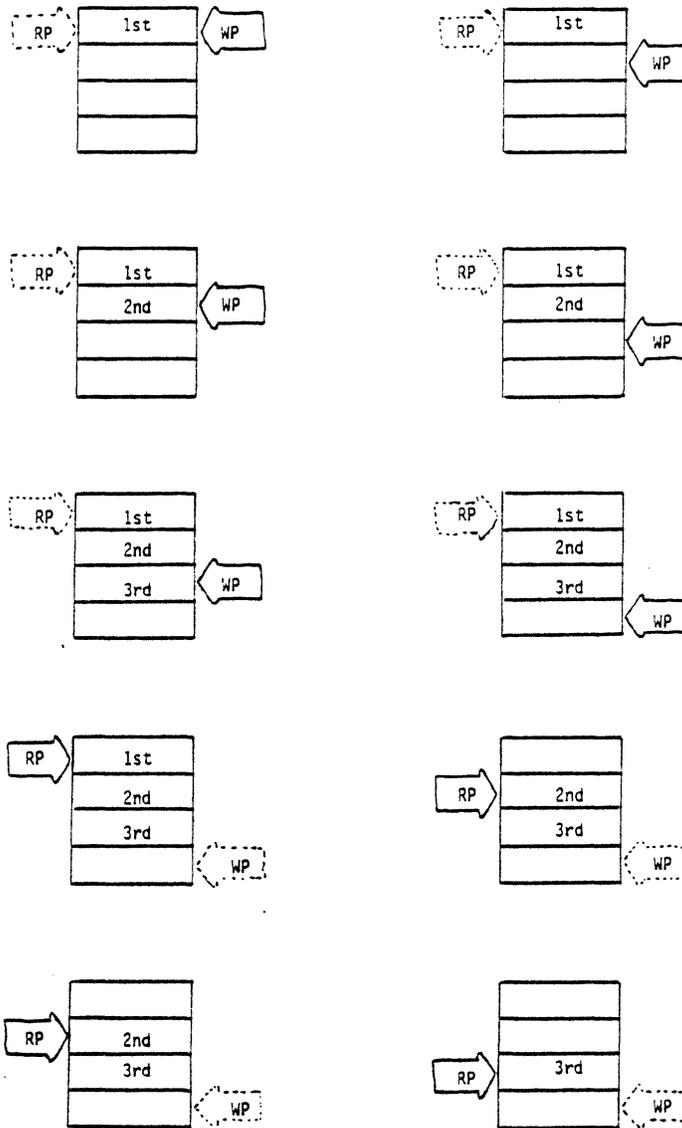
Operation of a FIFO memory is dependent on two pointers: a write pointer and a read pointer. Whenever data is to be loaded into the memory, the write pointer indicates the first available memory location. After the data is loaded, the pointer moves to the next sequential location. This process is repeated as often as necessary to load the required data, or until the memory becomes full. During read operations, the read pointer is initially positioned at the memory location where the first data item has been stored. After the data item is read, the read pointer moves to the next sequential memory location. This process is continued until all of the required data items have been read.

Operation of a typical FIFO memory is shown in Figure C-1. As shown on the figure, both the read pointer (RP) and the write pointer (WP) are initially equal.

As shown in Step 1 in the figure, the first data item (1st) is loaded into the memory location indicated by the write pointer (WP) and then the pointer is incremented to move it to the next sequential location. Note that the read pointer (RP) never moves during a write operation.

During Step 2, the second data item (2nd) is loaded and then the write pointer is incremented to move it to the next sequential location. During Step 3, the third data item is loaded and the pointer advanced again. This operation can continue as long as data items are to be entered or until the memory is full.

When unloading the memory (reading), the first data item is read (retrieved from memory) and the read pointer (RP) is then incremented to advance it to the next location. This process is continued until all items have been read. Note that the data items are read in the same order as they were written. That is, the data loaded first (1st) is the data that is unloaded first.



0093

Figure C-1 Operation of a Typical FIFO Memory

### C.3 STACK

A last-in, first-out stack is used to provide return address linkage when executing subroutines. This type of stack allows items to be added in sequential order and then be retrieved or deleted from the stack in the reverse order. It is not necessary for the programmer to keep track of the actual locations that data is loaded into; this is handled automatically by a "stack pointer."

A last-in, first-out stack is also referred to as a "pushdown" stack. As each item is added to the stack, the previous item is "pushed" down into the stack, and the last item added takes the top position on the stack. The words "push" (moved down into the stack) and "pop" (retrieve the most recently stored item from the top of the stack) are used to describe stack operations.

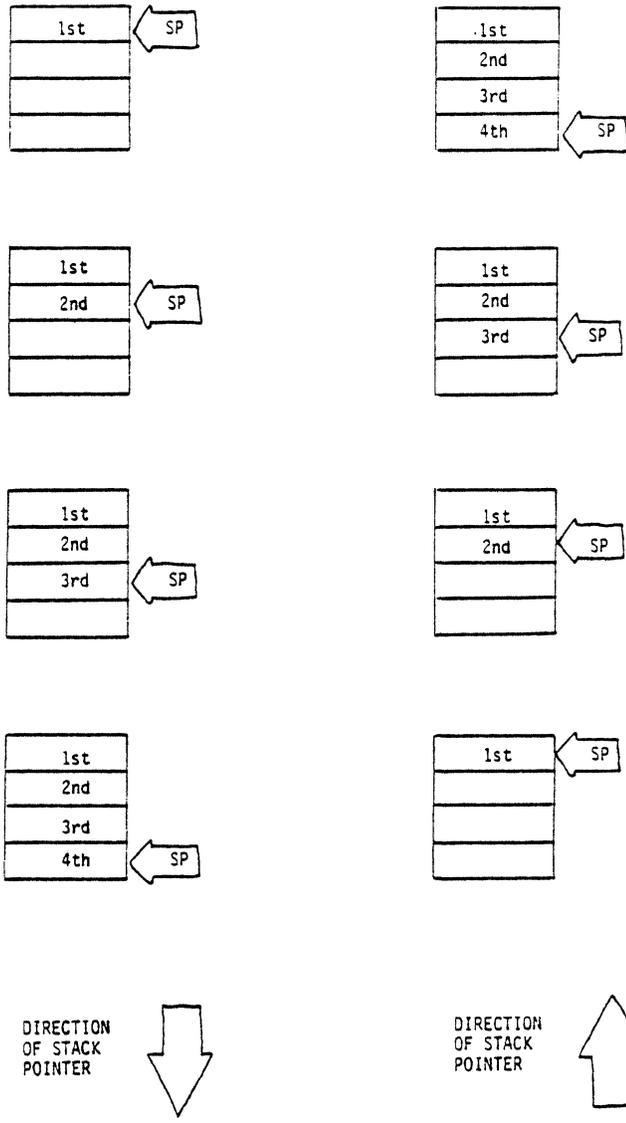
Whenever a subroutine call is made (a JSR) instruction, the JSR saves the current program counter by pushing it on to the stack. Once the program counter is saved, the JSR causes the program to jump to the specified location.

Whenever a return from subroutine call is made (an RTN instruction), the RTN instruction pops the stored program counter from the top of the stack and then causes the program to jump to the location specified by the popped word. In other words, the program jumps back to the same location it was prior to the JSR execution.

When a subroutine call is made within a subroutine, it is referred to as "nesting." In this case, the program counter from the first subroutine call is pushed on the stack first, then the program counter from the second subroutine call is pushed on the stack, etc. When returning to the main program, the last program counter stored is popped first to return to the last subroutine called. Then the next program counter is popped, etc. The last word to be popped is the first program stored. Because a 4-word stack is used in the PIOP, up to four subroutines can be nested.

Figure C-2 illustrates how the stack functions. Note that unlike the FIFO memory, the stack has only one pointer which is referred to as the "stack pointer" or "SP." This pointer initially points to some location. When a word is loaded into this location, the pointer moves down to the next sequential location. This process may continue until the stack is full.

When a word is to be popped, it is retrieved from the current location of the stack pointer and the pointer then moves to the preceding location. This process may continue until the stack is empty. Note that the last word pushed on to the stack (4th word) is the first word popped from the stack.



0094

Figure C-2 Stack Operation

## APPENDIX D

### SUMMARY OF PPDEBUG COMMANDS

#### D.1 INTRODUCTION

Abbreviations used in the following appendix:

| Symbol | Meaning   |
|--------|---|
| (cr)   | carriage return   |
| loc    | an integer location number                                  |
| count  | an integer count  |
| val    | an integer value  |
| fpn    | a floating-point number in form acceptable to FORTRAN       |
| mem    | the name of a PIOP internal memory (or AP main data memory) |
| reg    | the name of a PIOP internal register                        |

Debug types an "\*" when ready for further action. An "ERROR MESSAGE" is typed when a command is not understood.

## D.2 PROGRAM EXECUTION COMMANDS

|     |      |  |
|-----|------|--|
| I   | (cr) | Initialize. Reset the PIOP before program execution is resumed next. |
| R   | (cr) | Run. Begin program execution at PIOP program source                  |
| loc | (cr) | location LOC.  |
| X   | (cr) | Exit to APDBUG.  |

### D.3 REGISTER EXAMINATION/MODIFICATION COMMANDS

E (cr) Examine register. Print out the contents of PIOP  
reg (cr) register REG.

E (cr) Examine memory. Print out the contents of PIOP  
mem (cr) memory MEM (or APMD), location LOC.  
loc (cr)

. (cr) Re-examine the currently open register or memory  
location (the last location examined).

+ (cr) Examine the next higher sequential memory location  
of the memory that is currently open.

- (cr) Examine the next lower sequential memory location  
of the memory that is currently open.

F (cr) Floating Point Flag, affects the input/output of  
val (cr) 38-bit wide registers and memory locations.

VAL=0 3 integers (exponent, high mantissa, low  
mantissa)

VAL=1 floating-point.

C (cr) Change. Change the contents of the currently open  
val (cr) register of memory location to VAL. The format of  
VAL depends on the width of the current open locations  
as follows:

16-bit wide registers: an integer of the current radix

38-bit wide registers:

F=0 VAL (cr) three integers in the current radix  
VAL (cr) which represents the exponent, high  
VAL (cr) mantissa, and low mantissa

F=1: FPN (cr) a floating point number legal to FORTRAN

N (cr) Number radix. Set the radix for integer user  
VAL (cr) I/O to VAL, which must be 8 (for octal), 10 (for  
decimal) or 16 (for hexadecimal).

Z (cr) Zero. Zero out all ALU registers and PIOP program  
source memory.

#### D.4 MEMORY LOAD/DUMP COMMANDS

Y            (cr)    Yank. Load memory MEM starting at location  
MEM          (cr)    LOC from an external data FILENAME.  
LOC          (cr)  
filename     (cr)

W            (cr)    Write. Dump memory MEM starting at location  
MEM          (cr)    (START) and ending at location (STOP) to  
START        (cr)    external data FILENAME.  
STOP         (cr)    MEM can be PS or MD.  
filename     (cr)

## D.5 ACCESSIBLE FUNCTIONAL UNITS

AP Functional Units that may be examined or changed with PPDEBUG:

### MEMORIES

| Mnemonic | Name                       | Width |
|----------|----------------------------|-------|
| PS       | PIOP program source memory | 38    |
| MD       | AP main data memory        | 38    |
| ALU      | ALU registers              | 20    |

### REGISTERS

| Mnemonic | Name                                       | Width |
|----------|--|-------|
| AR       | address register                           | 8     |
| FF       | FIFO register                              | 38    |
| FLAG     | 8 PIOP flags<br>(flags 0-7, left-to-right) | 8     |
| IOR      | I/O register                               | 38    |
| PSA      | program source address                     | 8     |
| CR       | control register                           | 20    |
| Q        | Q register                                 | 20    |

## Notice to the Reader

- Help us improve the quality and usefulness of this manual.

Your comments and answers to the following READERS COMMENT form would be appreciated.

- To mail: fold the form in three parts so that Floating-Point Systems' mailing address is visible for the post office carrier; seal.

Thank You

## READERS COMMENT FORM

Document Title \_\_\_\_\_

*Your comments and answers will help us improve the quality and usefulness of our publications. If your answers require qualification or additional explanation, please comment in the space provided below.*

How did you use this manual?

- ( ) AS AN INTRODUCTION TO THE SUBJECT
- ( ) AS AN AID FOR ADVANCED TRAINING
- ( ) TO LEARN OF OPERATING PROCEDURES
- ( ) TO INSTRUCT A CLASS
- ( ) AS A STUDENT IN A CLASS
- ( ) AS A REFERENCE MANUAL
- ( ) OTHER \_\_\_\_\_

Did you find this material . . .

- |                       | YES | NO  |
|-----------------------|-----|-----|
| ● USEFUL?             | ( ) | ( ) |
| ● COMPLETE?           | ( ) | ( ) |
| ● ACCURATE?           | ( ) | ( ) |
| ● WELL ORGANIZED?     | ( ) | ( ) |
| ● WELL ILLUSTRATED?   | ( ) | ( ) |
| ● WELL INDEXED?       | ( ) | ( ) |
| ● EASY TO READ?       | ( ) | ( ) |
| ● EASY TO UNDERSTAND? | ( ) | ( ) |

*Please indicate below whether your comment pertains to an addition, deletion, change or error; and, where applicable, please refer to specific page numbers.*

| Page | Description of error or deficiency |
|------|------------------------------------|
|      |                                    |
|      |                                    |
|      |                                    |
|      |                                    |

From:

Name \_\_\_\_\_  
 Firm \_\_\_\_\_  
 Address \_\_\_\_\_  
 Telephone \_\_\_\_\_

Title \_\_\_\_\_  
 Department \_\_\_\_\_  
 City, State \_\_\_\_\_  
 Date \_\_\_\_\_

First Class  
Permit No. A-737  
Portland,  
Oregon

**BUSINESS REPLY**

No postage stamp necessary if mailed in the United States

Postage will be paid by:

**FLOATING POINT SYSTEMS, INC.**

P. O. Box 23489

Portland, Oregon 97223

Attention: Technical Publications



FLOATING POINT  
SYSTEMS, INC.

CALL TOLL FREE 800-547-1445  
P.O. Box 23489, Portland, OR 97223  
(503) 641-3151, TLX: 360470 FLOATPOINT PTL