An Introduction to the AP400 Array Processor

ANALOGIC
An Introduction to the
AP400 Array Processor

PROPRIETARY NOTICE

Analogic's AP400-based Signal Processing Systems utilize designs for which patents have been issued and/or are pending.

The information contained in this publication is derived in part from proprietary and patent data of the Analogic Corporation. This information has been prepared for the express purpose of assisting operating and maintenance personnel in the efficient use of the instrument described herein. Publication of this information does not convey any rights to use or reproduce it or to use for any purpose other than in connection with the installation, operation, and maintenance of the equipment described herein.

Analogic reserves the right to modify published specifications of equipment performance without prior published notice.

Second Edition
January, 1980
Copyright 1979 ANALOGIC CORPORATION
Printed in U.S.A. All rights reserved.
1 INTRODUCTION & GENERAL DESCRIPTION
  1.1 General
  1.2 Physical Description
  1.3 AP400 Card Set
  1.4 AP400 Design Features
     1.4.1 Operating Speeds
     1.4.2 Arithmetic Pipeline
     1.4.3 Memory
     1.4.4 Control Processor
     1.4.5 I/O Assembly
     1.4.6 Software
  1.5 Host-Array Processor Communications
     1.5.1 General
     1.5.2 General Sequence of Host-AP Operations
     1.5.3 A View from inside the AP400

2 PRINCIPLES OF AP400 OPERATION
  2.1 Introduction
  2.2 System Architecture
     2.2.1 Unit Functions
     2.2.2 AP400 Buses
  2.3 AP400 Pipeline Arithmetic Unit (PA)
     2.3.1 Pipeline Arithmetics
     2.3.2 The AP400 Pipeline Stages
        2.3.2.1 The Characterizer Stage
        2.3.2.2 The Multiplier Stage
        2.3.2.3 Accumulator/Logic Stage
  2.4 The Pipeline Arithmetic Command (PAC)
     2.4.1 General
     2.4.2 Elements of the PAC
     2.4.3 Pipeline Timing
     2.4.4 Pipeline Addressing
     2.4.5 Coding Considerations
  2.5 The Control Processor
     2.5.1 Functional Overview
     2.5.2 Program Memory
     2.5.3 Register and Arithmetic & Logic Unit (RALU)
     2.5.4 Stack Operation
     2.5.5 Interrupts
  2.6 Data Memory (DM)
  2.7 Input/Output (I/O)
     2.7.1 I/O Block Diagram (PDP-11 Interface)
     2.7.2 Host/AP Communications
     2.7.3 Programmed I/O
        2.7.3.1 Immediate Commands
        2.7.3.2 DATA (Non Immediate) Commands
     2.7.4 Direct Memory Access (DMA)
     2.7.5 Some Programming Considerations Implicit I/O Transfer Implementation
     2.7.6 AP Interrupt of Host
  2.8 Auxiliary Port
     2.8.1 Sequence of Operations
     2.8.2 Input Port
     2.8.3 Output Port
     2.8.4 Typical Use of the Auxiliary Port

3 SOFTWARE
  3.1 Introduction

4 HOST FUNCTION CALLS
  4.1 Introduction
  4.2 Function Control Blocks
     4.2.1 FCB Structure
     4.2.2 FCB Elements
  4.3 Function Parameter List Types
  4.4 Classification of Host Function Calls
     4.4.1 AP Resource Management
     4.4.2 AP Data Memory (Data Buffer) Management
     4.4.3 Input-Output Operations
     4.4.4 Logical Data Manipulation
     4.4.5 Straight Forward Computation
     4.4.6 Sophisticated Computation
  4.5 Host Function Library

5 AP400 ASSEMBLY LANGUAGE & MACHINE INSTRUCTIONS
  5.1 Introduction
  5.2 Instruction Execution Time
  5.3 Program Memory
  5.4 Assembly Language Instruction Listing
  5.5 AP Assembly Language Program Example
     5.5.1 Assembler Directives
     5.5.2 Instructions
     5.5.3 Data Storage Instructions

6 PAC LISTINGS
  6.1 Introduction
  6.2 Listing Format
  6.3 User Programming
ILLUSTRATIONS

Fig.# Title
1-1 AP400 Outline and Mounting Dimensions
1-2 AP400 Arithmetic Pipeline Assembly
1-3 AP400 Data Memory and Expansion Memory Assemblies
1-4 AP400 Control Processor Assembly
1-5 AP400 I/O Assembly (PDP-11)

2-1 AP400 System Architecture
2-2 Pipeline Timing Efficiencies
2-3 Pipeline Arithmetic (PA) Block Diagram
2-4 PA Characterizer Stage Block Diagram
2-5 PA Multiplier Stage, Block Diagram
2-6 PA Accumulator/Logic Stage, Block Diagram
2-7 PAC Decoding Sequence, Hardware & Software
2-8 Read/Write Timing Sequence
2-9 Mapping PAD Codes into Memory Addresses
2-10 Control Processor Simplified Block Diagram
2-11 Data Memory Simplified Block Diagram
2-12 Interface (I/O) Simplified Block Diagram
2-13 AP400 Front Panel Showing Status Register Indicators
2-14 Command & Memory Register Word Format
2-15 Host Read of Message & Status Register, Host Write of Immediate or Data Commands
2-16 Host Read/Write of Data Memory
2-17 DMA Formats Host TO/FROM AP
2-18 Word Count Register
2-19 AP to Host in DMA Operation
2-20 Host to AP DMA Timing Diagram
2-21 AP to Host 2-Word Transfer DMA Timing Diagram
2-22 AP Interrupt of Host Timing Diagram
2-23 Auxiliary Port Timing Diagram
2-24 Auxiliary Port Output Strobe Polarity Selection
2-25 AP400 Auxiliary Port Application Block Diagram
2-26 Auxiliary Port Application Detailed Block Diagram

3-1 Sample Host FORTRAN Program, Real FFT on 1024 Data Points
3-2 A Typical Linear Approximation Function
3-3 Implementing A Linear Approximation Function by Table Lookup

4-1 Function Control Block Contents

5-1 Sample AP Assembly Language Program Listing,
Negating Data Points in a Vector

TABLES

1-1 Typical Performance Characteristics
1-2 Typical Process (From Host Point of View)
2-1 Address Modifier Select
2-2 ALU Function Select
2-3 HOST TO AP Command Codes — In Hexadecimal
4-1 Function Control Block Elements—Description
1
INTRODUCTION
& GENERAL DESCRIPTION

1.1 GENERAL

The Analogic Array Processor AP400 is a high speed arithmetic computation unit designed to be operated in conjunction with a general purpose microcomputer, minicomputer, or a computer main frame. In combination with its host computer, the AP400 peripheral adds a powerful computing capability, providing economical signal and data processing at throughput rates 10 to 100 times faster than the stand alone computer.

The AP400 delivers cost-effective performance in both dedicated and general purpose applications. It is easily programmed, for example, for signal processing in tomography, sonar, seismic exploration, speech analysis, vibration analysis, image enhancement, and automatic test equipment applications.

1.2 PHYSICAL DESCRIPTION

As shown in Figure 1-1, the AP400 is configured in an EIA standard 19" (482.6mm) wide rack-mountable assembly, only 5.25" (133.35mm) high (also an EIA standard increment). The AP assembly includes the I/O board for the specified host, a Control Processor board, Arithmetic Pipeline board, and Memory board. It also includes its own power supply, real time clock assembly, and forced air cooling fans. In addition, as indicated in Figure 1-1, this assembly is designed for expansion up to the maximum data memory of 64K 24-bit words. Cabling to the host computer bus and to an auxiliary bus is ducted to the rear of the AP400 assembly. There is ample depth behind the case assembly depth of 19 inches (482.6mm) for rear cabinet interconnections.

Figure 1-1. AP400 Outline and Mounting Dimensions
The AP400 front panel includes 12 status indicator lights (including one to indicate the actuation and appearance of +5 volt power). They provide a visual indication of the relative operations of host computer and the AP400, and are useful in evaluating program efficiencies and in elementary trouble-shooting and diagnosis. Additional details are included in the functional descriptions in Chapter 2.

Figure 1-1 also illustrates the convenient access to the interior assembly. The assembly can slide forward on extensions built in to the wire card cage. During this operation a built-in "sleeve" also slides forward to maintain an efficient cooling configuration for the rear-mounted fans so that the equipment may be operated without damage in its extended position. Note, in Figure 1-1, that the extended position also permits access to the spare slots in which the Memory expansion boards are inserted.

1.3 AP490 CARD SET CONFIGURATION

The Array Processor may also be installed as an integral part of the host assembly. This is accomplished by installing the plug-in assembly boards and back plane within the computer main frame (or other peripheral). The card set installation does not include the front panel, power supplies, or cage assembly.

1.4 AP400 DESIGN FEATURES

1.4.1 Operating Speeds.

The Analogic AP400 small-size, low-power array processor features an arithmetic pipeline design that, along with high speed memory components, buffered command and data ports, and multilevel programming, results in efficient, real time, digital signal processing previously available only in machines with many more components and that require complex programming. Typical processing times are listed in the table below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logarithm</td>
<td>1.9</td>
</tr>
<tr>
<td>Exponential</td>
<td>2.4</td>
</tr>
<tr>
<td>Magnitude squared</td>
<td>1.0</td>
</tr>
<tr>
<td>Multiplication rate</td>
<td>up to 2.1</td>
</tr>
<tr>
<td>Addition, Subtraction rate</td>
<td>up to 6.3</td>
</tr>
<tr>
<td>512-Point Real FFT</td>
<td>1.5</td>
</tr>
<tr>
<td>1024-Point Real FFT</td>
<td>3.6</td>
</tr>
<tr>
<td>1024-Point Complex FFT</td>
<td>7.4</td>
</tr>
<tr>
<td>Real Convolution (512 Data, 1024 Kernel)</td>
<td>7.3</td>
</tr>
<tr>
<td>1024-Point Real Vector * Vector</td>
<td>0.6</td>
</tr>
<tr>
<td>32 x 32 Complex Matrix Transposition</td>
<td>1.9</td>
</tr>
</tbody>
</table>

1.4.2 Arithmetic Pipeline. (Figure 1-2)

The Arithmetic Pipeline is the basis for the high speed processing ability of the AP400. Its operation is described in detail in Chapter 2. In brief, the pipeline is internally programmed to receive eight 24-bit data words at the input of each pipeline pass and to produce four 24-bit data words at the output. The pipeline is structured into three stages of equal processing time. After it is once filled, data outputs occur at a rate equal to one-third of that required to fill the pipeline initially, as long as data is continuously input at the same rate. Each 24-bit data word in a group of words for a programmed pipeline operation represents the mantissa portion of a scaled data value. A common 16-bit exponent is stored for the group. The group of data words scaled for each such exponent is a "block", and the array processor operates primarily in a "block floating point mode" (The block floating point mode is further described in Chapter 3). Some of the arithmetic pipeline design features are:

- Normal Block Floating Point Data Format:
  - 24-bit BFP 2's Complement Mantissa
  - 16-bit BFP 2's Complement Exponent
- Eight 24-bit data words in; Four 24-bit data words out
- Up to 256 determinable Pipeline Arithmetic Commands
- Multiplication operation: 24 x 24-bit input; full 48-bit result, truncated or rounded to 24 bits.
- Access to data-dependent table entries
- Eight accumulators internal to pipeline, accessed as part of the pipeline operation without requiring external program cycle
- Guard bits for overflow protection
- Zero pipeline reconfiguration delay

1.4.3 Memory (Figure 1-3)

The standard memory includes 2K words of 22-bit program memory and 4K words of 24-bit data memory. The data memory may be expanded with additional 4K words on the standard board. Up to 64K data memory words may be configured in 4K increments, using Expansion Memory boards. It should be noted that the program memory in the Array Processor may be augmented by storage in the Host or Auxiliary peripherals, since it is software configured. Some of the key features incorporated in the program and data memory, are listed below:

**PROGRAM MEMORY**

- Standard Memory: 2048 x 22-bit, HMOS, 55 nsec RAM
- Address Register: 12 bits
- 8 locations for vectored interrupts
- Contents are downloaded from the Host

**DATA MEMORY**

- Standard Memory: 4096 words x 24-bit, HMOS, 55 nsec RAM
- Add-on Memory: 4096 additional words on board; Expansion Memory boards: up to 16K words (4K increments)
- Maximum Data Memory: 64K words
- Program Stack: 64 words
1.4.4 Control Processor (Figure 1-4)

The Control Processor is the Array Processor’s manager. It interprets Host-generated commands/instructions, and sets up the lists of addresses and commands for arithmetic unit processing, links programs, and passes addresses for data and parameters. In general, it functions to relieve the Host of the burden of managing the AP400. Some of the performance features designed into the AP400 Control Processor are:

- 19 classes of machine language instructions
- 16-bit computation word size
- 16 registers, 16-bits wide
- 8 levels of hardware vectored interrupts
- 8 special purpose hardware flags
- Single-word CP instruction cycle time: 160 nsec
- Maximum Host Memory and Auxiliary I/O DMA rate: 1.5 million words/sec.

1.4.5 I/O Assembly (Figure 1-5)

A single Input/Output (I/O) card provides all the communications between the AP400 and the Host computer, and between the AP400 and devices connected to the AP400 via the Auxiliary Ports. A dedicated I/O card is required for each Host computer with which the AP 400 is specified to interface. Each card contains the circuitry to carry out the following I/O tasks:

- Direct the AP400 status: Halt/Run, Single Step, etc.
- Transfer data to and from the Host under Programmed I/O
- Transfer data to and from Host via DMA
- Access various nodes of the Array Processor for diagnostic testing
- Transfer data in and out of Auxiliary Ports

1.4.6 Software

The AP400 Array Processor is fully supported with software packages of the following types:

- Applications: for problem solutions and real-time tasks
- Systems: for control of Host and AP400 activity
- Utilities: for software preparation and use
- Diagnostics: for hardware and software fault detection and isolation

Documentation for AP400 software packages and AP400 installation are provided by:

AP400 Processor Handbook
AP400 Function Reference Manual
AP400 Host System Software Reference Manual
AP400 Interactive Debugging Tool Reference Manual
AP400 Linker Reference Manual
AP400 Diagnostic Reference Manual
AP400 PAC Reference Manual
Quick-Reference Card -- AP Assembly Language
Quick-Reference Card -- AP400 Interactive Debugging Tool
AP400 Installation Manual
AP490 Installation Manual

1.5 HOST-ARRAY PROCESSOR COMMUNICATION

1.5.1 General

The following paragraphs describe a sequence of Host-Array Processor operations involved in executing an Array Processing function. This description illustrates the communications performed across the interfaces between the Host and the Array Processor. This section also introduces additional design features that contribute to the efficient operation of the AP400.

The scenario that follows is written from the viewpoint of an “observer” located in the Host computer who sees only the interface with the Array Processor and does not become aware or concerned with the operations internal to the Array Processor. Later paragraphs will consider the operation from the viewpoint of an “observer” in the Array Processor with similar constraints.

This illustration of a typical operation assumes that the AP400 is configured to interface with the designated Host computer, that the Host computer operating system FORTRAN compiler has been appended to include the AP FORTRAN calls. Also, that the Host operating system has been supplemented to include the AP Manager and AP Driver program modules. Normally, these are initializing actions and are completed at the time the AP400 is installed.

The sequence that is described includes many steps that are invisible to the system user. Almost all are invisible to the FORTRAN user of the Array Processor and only very few are apparent to the Host Assembly Language user.

1.5.2 The General Sequence of Host-Array Processor Operation

The Host-Array Processor interaction occurs by both Programmed I/O (PIO) and Direct Memory Access (DMA) types of interface operation, and each of the interactions below is identified as to the type involved. In general, the DMA interface is accomplished with a single instruction for the transfer of a block of data at a transfer rate limited only by the read/write speed of the memory and buses involved. The PIO interface typically requires a separate instruction for each of the handshake protocols.

1.5.3 A View from inside the AP400

The AP400 appears (from inside the interface boundary) as an independent, stored-program minicomputer. The Control Processor (CP) within the AP400 executes an assembled program of machine instructions according to the sequence called out by its program counter. In the AP400, the program counter is the Program Memory Address Register, PMAR. A program steps along at the
clock-controlled interval of 160 nanoseconds. Machine instructions may require a sequence of 2 or more such clock intervals. Most are executed in one, but a few may require up to 3 or 4, depending upon the arguments of the instruction.

When the programmed AP instruction calls for the use of the Pipeline Arithmetic unit (PA), the AP400 completes its control function by transferring four successive Command and Address words to the Command and Address Buffer (CAB). Its contents consist of the PA jobs to be done, and in the sequence to be accomplished. The CAB contents are continually changing as more pipeline commands are added, and as the existing ones are withdrawn to be processed.

The CAB has its own control pointer by which the 4-word instruction set is retrieved in the order stored. These instructions (PACs) are decoded in the pipeline in pre-programmed PROM's. The PAC's set up pipeline control signals and initial Data Memory addresses for PA processing of blocks of data beginning at that address. Addressed data is synchronously clocked through the pipeline at 1.92 microseconds per PAC and is repeated for as many PACs as required for the complete block. For each PAC, the address of the input data is indexed until the block of data has been processed. The PA operation proceeds independently of the CP operations (once the CP has transferred the command to the CAB), retrieving data values from designated memory locations or from a modified address location, and storing the results in programmed Data Memory locations.

The Command & Address Buffer (CAB) can store up to 64 24-bit words, and, since 4 such words comprise a pipeline instruction set, the CAB has the capacity to store up to 16 PA instruction sets. When the CAB is near full, it causes the CP clock to stop to prevent possible overflow of the CAB, and consequent loss of an instruction. (The PA clock is not stopped, and processing through the PA continues.) When the CAB has been emptied below the “full” level, the CP clock is restarted, and the program continues its execution. When the CAB is empty, the PA clock is stopped to prevent any errors from timing offsets in the Pipeline and Data Memory combination. Note that the operation is asynchronous with the Host timing, but is rigorously controlled within the AP400.
## Table 1.2

Typical Process (From Host Point of View)

<table>
<thead>
<tr>
<th>STEP</th>
<th>ACTION</th>
<th>TYPE OF I/O TRANSFER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Host loads the address of a Function Control Block (FCB), which resides in Host Memory, into a location in the AP Data Memory.</td>
<td>PIO</td>
</tr>
<tr>
<td>2.</td>
<td>Host announces its need for the AP400 to perform a function by loading the “Perform Function” message into the AP400 Message Register, and then interrupts the AP400.</td>
<td>PIO</td>
</tr>
<tr>
<td>3.</td>
<td>AP400 responds to the interrupt and fetches the message from the AP400 Interface.</td>
<td>(AP only)</td>
</tr>
<tr>
<td>4.</td>
<td>AP400 retrieves the FCB address from the AP400 Data Memory.</td>
<td>(AP only)</td>
</tr>
<tr>
<td>5.</td>
<td>AP400 accesses the FCB in Host Memory and transfers FCB to AP400 Data Memory.</td>
<td>DMA</td>
</tr>
<tr>
<td>6.</td>
<td>The AP Function specified in the FCB is initiated, and is executed based upon control information stored in the FCB.</td>
<td>(AP only)</td>
</tr>
<tr>
<td>7.</td>
<td>Any data required by the AP Function is retrieved by the AP directly from Host Memory. Likewise, any AP Function results to be directed to the Host Memory are placed there directly by the AP.</td>
<td>DMA</td>
</tr>
<tr>
<td>8.</td>
<td>When the AP Function is completed, the AP “marks” the FCB in Host Memory and checks for another AP Function FCB chained from this last one.</td>
<td>DMA</td>
</tr>
<tr>
<td>9.</td>
<td>If another FCB is chained to the last one, the AP retrieves it and the process described above repeats...without an interruption of the Host unless one is required for programmed synchronization of Host and AP operation.</td>
<td>DMA</td>
</tr>
<tr>
<td>10.</td>
<td>If no further FCB is chained, the AP places a &quot;Function Done&quot; message into the designated register, and interrupts the Host.</td>
<td>Interrupt</td>
</tr>
<tr>
<td>11.</td>
<td>When the interrupt is acknowledged, the Host may resume execution of a task that was suspended while awaiting the AP results, or may set a flag to indicate “AP Done”, which a subsequent Host task may utilize as necessary.</td>
<td>(HOST only)</td>
</tr>
<tr>
<td>12.</td>
<td>Meanwhile, the AP waits for another “Perform Function” message, and may continue to perform its ongoing operations (e.g. real-time input through the auxiliary I/O port).</td>
<td>(AP only)</td>
</tr>
</tbody>
</table>
Figure 1-2. AP400 Arithmetic Pipeline Assembly

Figure 1-3. AP400 Data Memory and Expansion Memory Assemblies
Figure 1-4. AP400 Control Processor Assembly

Figure 1-5. AP400 I/O Assembly (PDP-11)
2.1 INTRODUCTION

This chapter describes the system architecture of the AP400 and the implementation of the major functions. The word format and block floating point implementation in the AP400 are described in Chapter 3, Programming Considerations.

2.2 SYSTEM ARCHITECTURE

As shown in Figure 2-1, the AP400 is essentially four basic functional units interconnected by three buses (identified in the illustration) and other dedicated hardwired connections (not shown). The four functional units and their short form abbreviations are:
- Interface for Host and Auxiliary (I/O)
- Control Processor (CP)
- Pipeline Arithmetic (PA)
- Data Memory (DM)

In some configurations, there may be one (or more) Expansion Data Memory unit(s). Functionally, however, these are only extensions of the basic Data Memory, and their incorporation does not change the system block diagram as shown in Figure 2-1. Each AP400 is supplied with a Real Time Clock assembly, that is either incorporated in an Expansion Memory assembly (if installed), or is installed as a small pc-card plugged into the back plane assembly. The primary oscillator is located in the Control Processor unit.

The three AP400 internal buses and their short form abbreviations are:
- Command & Control Bus (CCB)
- Register and Arithmetic Logic Unit Bus (RALU)
- Data Bus (DB)

The AP400 has been designed so that related functions are, for the most part, located in the same physical assembly. Thus, the functional block subdivision shown in Figure 2-1 is used for the pc-board assemblies in the instrument, and appear on the board labels. The Interface assembly is Host dependent, and is designed for compatibility with a specific Host. By grouping the functions in separate physical units, and keeping all the interface functions on the I/O board, it is possible to adapt the AP400 to a new Host by replacing only the I/O board assembly.

Figure 2-1. AP400 System Architecture

2.2.1 Unit Functions

The Interface (I/O) provides for all the communications between the AP400 and its Host or Auxiliary peripherals. It provides for the transfer of data under programmed I/O or DMA transfer modes, and for accessing specified nodes in the Array Processor for diagnostic testing.

The Control Processor (CP) is the manager of the AP400. It is essentially a minicomputer, executing the programmed tasks passed to it by the Host, and using the AP400 Data Memory and Pipeline Arithmetic when programmed to do so. It contains its own microprocessor unit to perform various register-to-register, quantity-to-register, and register address modifications to support pipeline setup requirements. It also contains an Interrupt Vector structure for interrupt-driven processor coding, as well as read/write type of Program Memory.
The Pipeline Arithmetic (PA) is the “number crunching muscle” of the AP400. It processes 4 pairs of 24-bit input pairs (or 8 independent inputs) through three programmed stages: data characterization (allowing for input data-based modifications), multiplication, and arithmetic/logic operations and accumulations. The PA generates either 2 pairs of output values, or 4 independent outputs. This unit also contains the PACs in factory programmed PROMS that provide the pipeline control signals.

The Data Memory DM provides a contiguous space of 24-bit RAM locations for data storage and the three registers for addressing the memory: CP-DMAR, I/O-DMAR, and PA-DMAR. The Data Memory is expandable up to 65K 24-bit words. The basic DM board is configured with 4K RAM storage, and has the capability for on-board addition of another 4K. Thereafter, additional data memory is added in 4K increments are assembled on Expansion Memory boards, with up to 16K per Expansion Memory board. This unit also contains the Command & Address Buffer CAB that stores up to 64 24-bit words of instruction codes for the PA. A group of 4 words from the CAB completely defines a pass through the pipeline. The unit also contains the means for modifying the addresses of the input data for the pipeline.

2.2.2 AP400 Buses
The bus structure and operation within the AP400 are essentially invisible to the user. Their descriptions are included here to provide some reference for the unit descriptions that follow.

The Command & Control Bus (CCB) is a bi-directional bus, 8 bits wide. The commands put onto this bus determine the routing of data within the AP400 by way of the other buses. After a transfer has occurred, an address register may be incremented or a status bit set as part of the same command action. The use of the CCB minimizes the number of separate control signals needed to coordinate the actions of the four functional units of the AP400.

The CCB is also pipelined, so that the issue of one command occurs while the previous command is being executed. Every clock pulse (160 nanoseconds) a new command can be issued on the command bus. Transfers requiring more than 160 nanoseconds for execution because of propagation delays, are accomplished by the hardware issuing the identical command for two clock cycles.

When the CCB is not needed, a default command is issued that allows the CP to execute instructions not using the RALU bus, and connects the PA to the DM for pipeline operations.

The Data Bus DB is a bi-directional bus, 24 bits wide. It is used by the I/O when transferring data to/from the Host and from/to data memory. The data bus is actively used in the performance of the pipeline operations, transferring four pairs of data values from Data Memory locations specified by the pipe setup addresses to the pipeline. The pipeline outputs to Data Memory travel over a separate 8-bit wide connection and are formatted in 24-bit words for Data Memory and Data Bus. Access to Data Memory via the Data Bus is shared by the I/O, CP, and PA, and the priority is in that order (PA last). When either the I/O or CP require the use of the Data Bus, the PA operation is momentarily interrupted, by stopping its clock. This is called cycle stealing.

The Register & Arithmetic / Logic Unit Bus (RALU) is a bi-directional bus, 16 bits wide. It is used to transfer addresses from the Control Processor and I/O to and from the Data Memory. The RALU bus is also used to transfer most of the information to the CAB from the CP.

The control bus structure used in the AP400 greatly simplifies on-line program debugging and fault detection. The CCB allows examination of the contents of most storage elements inside the AP400 while any program is either running or temporarily halted. In this mode the CP can duplicate the actions of the HOST in issuing read commands to the AP400 Host Computer's request of the interface to do the same. Finally, the same paths and control logic are used for loading and reading back programs and data as for normal program execution.

2.3 AP400 PIPELINE ARITHMETIC UNIT (PA)

2.3.1 Pipeline Arithmetics

One way to increase the throughput when processing arrays of data is to parallel complete arithmetic units and to partition the data among them. This technique is costly in terms of hardware. It also causes programming complexity associated with maintaining correct synchronization among parallel units and in combining partial and final results.

Another way to increase throughput is to partition the arithmetic unit into stages and to introduce new data inputs to the first stage when the previous data moves to the second stage, etc. This is the technique used in the AP400 and three such stages are used. They are:

- Stage A: Data Characterization
- Stage B: Data Multiplication
- Stage C: Data Accumulation and Logical Manipulation

To increase the efficiency of such partitioning, the configuration of each stage and passing of data between stages are determined by program control. This flexibility meets the requirements for a wide range of processing functions. The speed advantage of a 3-stage pipeline is illustrated in Figure 2.2. As shown in the illustration, the processing cycle time is \( A + B + C \). When not pipelined, the processing of \( n \) data sets requires \( n \) processing cycle times, however fast or slow that may be.

In a 3-stage pipeline unit, the results of processing the first data set will not appear until after the full time of a processing cycle \( (A + B + C) \). But the second and succeeding data set results appear at intervals of one-third the processing cycle thereafter. Thus for \( n \) data
pairs, the processing time is $1 + \frac{1}{3(n-1)}$ cycles; and for large "n", the value approaches $1/3$ the time. Note also that the last data pair to enter the pipeline must be "pushed out" in some manner if no other data pair follows.

2.3.2 The AP400 Pipeline Stages

As shown in Figure 2-3, the 3-stage Pipeline Arithmetic unit receives eight 24-bit values at the input and delivers four 24-bit results at the output of the third stage.

Each stage of the PA is designed to perform several variations of that stage's function. Control signals decoded from the PIPE instruction of the AP program configure each stage so that the appropriate inputs are selected, and the desired arithmetic combinatorial or logical operations are performed. In brief, a programmed instruction in the AP Assembly Language program is translated into a set of control signals. These signals synchronize the configuring of each of the three stages with the stepping of the numerical data through the pipeline.

Typically, the numbers that are processed through the pipeline for a given function have been "normalized" for a block floating point value. Thus, the 24-bit numbers

![Diagram of Pipeline Timing Efficiencies](image)

Figure 2-2. Pipeline Timing Efficiencies
are all mantissas of the same block exponent. The PA includes a provision to examine the results and to keep track of the change required in the block exponent to normalize the block.

The three pipeline stages and their functions are:

- **Characterizer Stage**: a versatile type of data conditioner, that prepares multiplier and multiplicand inputs from source data or from tabular data indexed by the source data.
- **Multiplier Stage**: performs multiplication of selected multipliers and multiplicands with optional accumulation of partial products.
- **Accumulator/Logic Stage**: performs arithmetic and logical operations on selected multiplier stage outputs. These include accumulations, additions, subtractions, logical comparisons, and block exponent normalization functions.

Figure 2-3 also indicates an inter-stage storage and selection block function. This acts as a type of controlled cross-bar switching function, setting up appropriate selection of the four pairs of multipliers and multiplicands for the Multiplier stage from any set of inputs of the previous stage.

### 2.3.2.1 The Characterizer Stage

Figure 2-4 illustrates the logical operation of the PA Characterizer Stage on the four source number pairs. As noted earlier, these may be complex pairs or independent values in adjacent addresses. As shown in the illustration, control signals determine whether these source numbers are passed through unchanged, or whether some are used in a table lookup mode. When they are passed through unchanged, then all 4 pairs become possible multiple inputs. When the characterizer is used in a lookup mode, then the data values of S1...
are used to modify the initial address of S3 (and the data values of S2 modify those of S4). The algorithms that generate the address modifier use either the four, six or eight MSB's of S1R (or S2R); or a combination of the 2 or 4 MSB's of S1R and S1I (or S2R and S2I); or the leading zero count with or without the sign value of S1R (or S2R).

Table 2-1 indicates the codes (1 through 7) that are used to examine the leading bits (MSBs) of S1 and S2 in performing a modification of the address for S3 and/or S4. Code 0 results in no modification.

Since the table data is addressable at any valid memory location, this feature of the characterizer permits the user to substitute data tables in any generic-type algorithm. For example, a linear interpolation algorithm can be used with a data table to obtain generated functions (logarithms, trigonometric values, etc.), or to perform piecewise interpolation on incoming data variables.

2.3.2.2 The Multiplier Stage

The Multiplier Stage accepts eight (8) 24-bit input operands and delivers four (4) 24-bit output results. The result of the multiplication is a 48-bit word which is truncated or rounded to 24 bits, according to the decoded PAC instruction. When required, two 24-bit results may represent the two parts of a 48-bit double precision result. Figure 2-5 illustrates the logic flow for any one of the four adjacent multipliers in this stage. As shown in the illustration, decoded instructions develop control signals that configure the multipliers in five main groups:

1. To determine which input (S1R, S1I, S2R, S2I, etc.) will be a multiplier, multiplicand, or bypass operand. Recall that the S3 and/or S4 values may be table lookup data.
2. To determine whether the product will be rounded or truncated.
3. To determine whether the MSB's or the LSB's of the output, or the bypass operand will be passed to the Storage/Select for the next stage.
4. To determine whether an adjacent accumulator result will be introduced into the accumulator.
5. To determine whether the result of the multiplication will be scaled (downshifted) before passing to the next stage. The downshift may be 0, 1, 2, or 3 places.

2.3.2.3 Accumulator/Logic Stage

Figure 2-6 illustrates one of four processing units making up the third stage of the PA. Each unit includes two Arithmetic Logic Units (ALU's) and some data selection. The data being processed in this stage are selected from the four multiplier outputs (M1, M2, M3, and M4) and from eight accumulator registers of 24-bits each, labeled S1, S2, S3, and S4, and T1, T2, T3, and T4. These accumulators may be loaded either by a “loading” PAC prior to this PAC, or by the current PAC for use in the next pass of this PAC. Sign information from one of the multiplier outputs can also be used in the ALU operation, to provide conditional logic capabilities.
Figure 2-5. PA Multiplier Stage, Block Diagram
Both ALU's in each of the four processing units receive the same inputs labeled P and Q, but can form different functions of those inputs. Their arithmetic/logical combinations are determined in complementary pairs by the instruction-decoded control signals. Table 2-2 defines the 16 possible functions in each of the ALU's that are controlled by the instruction.

The outputs of each of the four units in this stage go to the PA output line and/or to replace an accumulator value in one of the eight accumulators.

A leading zero count function may be performed on data leaving the Accumulator/Logic stage. When this function is enabled by the decoded instruction, the leading-zero-count of the present computation is compared with the previous result of such a comparison, and the lower of the two is set up as output for later comparisons. At the end of a function processing operation, the output value represents the number of shifts to normalize, NSN, and may be used to modify the block exponent for later processing. The function is programmable so that it may be inhibited when the user knowledge of the data and the operation provides assurance that such a normalization would not be necessary.

2.4 THE PIPELINE ARITHMETIC COMMAND (PAC)

2.4.1 General

As shown in Figures 2-3 through 2-6, the Pipeline Arithmetic unit stages are configured for each pass of
Table 2-2

ALU FUNCTION SELECT

<table>
<thead>
<tr>
<th>HEX CODE</th>
<th>X ALU FUNCTION</th>
<th>Y ALU FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Q</td>
<td>Q MINUS P</td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>Q</td>
</tr>
<tr>
<td>2</td>
<td>P PLUS Q</td>
<td>Q MINUS P</td>
</tr>
<tr>
<td>3</td>
<td>P PLUS Q</td>
<td>Q PLUS P</td>
</tr>
<tr>
<td>4</td>
<td>P MINUS Q</td>
<td>Q</td>
</tr>
<tr>
<td>5</td>
<td>P PLUS Q PLUS CARRY</td>
<td>Q MINUS P PLUS CARRY - 1</td>
</tr>
<tr>
<td>6</td>
<td>P MINUS Q PLUS CARRY - 1</td>
<td>Q PLUS P PLUS CARRY</td>
</tr>
<tr>
<td>7</td>
<td>P OR Q</td>
<td>Q OR P</td>
</tr>
<tr>
<td>8</td>
<td>P AND Q</td>
<td>Q AND P</td>
</tr>
<tr>
<td>9</td>
<td>P EXOR Q</td>
<td>Q EXOP P (= Q EXN0P P)</td>
</tr>
<tr>
<td>A</td>
<td>IF MULT +</td>
<td>IF MULT +</td>
</tr>
<tr>
<td></td>
<td>THEN P</td>
<td>THEN Q</td>
</tr>
<tr>
<td></td>
<td>ELSE Q</td>
<td>ELSE P</td>
</tr>
<tr>
<td>B</td>
<td>IF MULT +</td>
<td>Q OR P</td>
</tr>
<tr>
<td></td>
<td>THEN P PLUS Q</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ELSE P MINUS Q</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>SPARE</td>
<td>SPARE</td>
</tr>
<tr>
<td>D</td>
<td>IF MULT +</td>
<td>Q</td>
</tr>
<tr>
<td></td>
<td>THEN P PLUS CARRY</td>
<td>IF MULT +</td>
</tr>
<tr>
<td></td>
<td>ELSE P PLUS CARRY - 1</td>
<td>THEN Q PLUS CARRY - 1</td>
</tr>
<tr>
<td>E</td>
<td>Q</td>
<td>ELSE Q PLUS CARRY</td>
</tr>
<tr>
<td>F</td>
<td>P MINUS Q</td>
<td>Q</td>
</tr>
</tbody>
</table>

Data through the "pipe" by a decoded command. The controls for each stage are derived from the command (shown in the sequence in Figure 2-7) by PROMs that are factory-programmed for the designated pipeline functions. If desired, a user may change these PROM's, and a documentation package is available to support this option. While the form of the command and its decoding are transparent to the FORTRAN and Host Assembly language programmer, brief descriptions of these items are included here to clarify the pipeline concepts described previously and to indicate the power of the AP400 Assembly Language instruction set. Although the AP400 Control Processor uses only 20 basic instructions, the Pipe instruction is expandable into 256 different PAC configurations. This macrocode expansion provides a highly flexible and efficient instruction set when writing AP400 Assembly Language code.

2.4.2 Elements of the PAC

Each PAC instruction in machine language consists of five instructions in the form of a PIPE, followed by four PAD's (setup instructions).

The PIPE may include one or more arguments identifying the PAC function, a scale factor operation, and a leading-zero count operation.

The PAD arguments include address codes for source and destination data. The actual memory addresses are computed from these codes, and are described later.

It should be noted that the PAC specifies one pass of a data set and associated commands through the three stages of the pipeline. The application-program structure determines the number of passes required as well as the amount of data to be processed. It is possible to Interleave PAC's; one data set of 8 (or 4 pairs) of values and commands pass through the pipeline as part of function "A", followed immediately by the data set and commands for function "B", followed by function "A", etc.
Figure 2-7. PAC Decoding Sequence, Hardware & Software
2.4.3 Pipeline Timing

A complete pass through the pipeline requires a number of discrete, well-defined operations: of fetching operands, operating on them, and storing the results. These are clocked through the pipeline at clock intervals of 160 nanoseconds, and a total of 36 such intervals are used for one pass. When the pipeline is kept busy, two number-pair results appear at the pipe output every 1.92 microseconds.

Figure 2-8 indicates the sequence of cycles in a pipeline pass. Twelve (12) cycles are used for read/write, and the remaining 24 cycles are used to accomplish the pipeline arithmetic operations. The 12 read/write time intervals use the Memory Data Bus, and if that bus is required for an I/O or CP operation the PA clock is temporarily stopped.

2.4.4 Pipeline Addressing

As shown in Figure 2-7, the source and destination addresses are encoded in the Control Processor (CP) by PAD set-up instructions. The four 22-bit words contain codes for the arithmetic operation, the register addresses for source and destination, and codes identified as D1 ADR and D2 ADR. The latter are used to control the mapping of the source and destination addresses into 16-bit address streams A1, A2, A3, and A4 as part of the 24-bit words that are stored in the Data Memory Command and Address Buffer. (The 16 bits address up to 64K of Memory locations).

Figure 2-9 includes a table that describes the mapping. It should be noted, as shown in Figure 2-9, that the pipeline processing accomplishes a "data replacement" action. That is, the pipeline source data in memory address S1 may be replaced by the D1 output at the end of the pipeline pass. (D1R replaces S1R and D2I replaces S2I) Note, also, that the addresses A3 and A4 may be modified by the actions of the characterizing stage in determining the location of S3 and S4.

2.4.5 Coding Considerations

The pipeline works at maximum efficiency by processing PIPE instructions in a continuous sequence. The Command & Address Buffer (CAB) queues up instructions for the pipeline, not only by storing up to 13 different PAC's, or pipeline instructions, but also by causing the same PAC to sequence through all the data points in a block of data. This latter action is usually more significant as far as elapsed time is concerned. Both actions allow the programmer ample time to group together any remaining coding instead of spreading it around within the program. This makes coding, or writing in AP Assembly Language, more straightforward and allows existing code to be understood more easily. At the same time, the pipeline can operate on a more continuous basis.
2.5 THE CONTROL PROCESSOR

2.5.1 Functional Overview

The Control Processor (CP) is the executive controller of the AP400. It is, essentially, a minicomputer that serves as central processing unit for the Array Processor. The functions of the Control Processor are to set up the lists of addresses and commands that the Pipeline Arithmetic unit then executes, to link programs (including parameter and initial conditions passing), and to handle programmed flag conditions and interrupts. As shown in Figure 2-10, the Control Processor includes: a 16-Register File Arithmetic & Logic Unit (RALU) microprocessor element, Program Counter, Program Memory Address Register, Command and Instruction Decoder blocks, Status Bits register, and the Interrupt Vector encoder. Communication with other units of the Array Processor is accomplished by the RALU and Command Code buses, as well as interconnecting wiring of the Pipeline Command, External Status lines, and the Interrupt Signals. An internal (CP) Instruction Bus is also used.

2.5.2 Program Memory

The Program Memory (PM) is 2048 words of 22 bits. It is loaded by the Host prior to run time with the AP400 Executive and the Function Library required for the applications being processed. The Program Memory is accessed by the Program Memory Address Register, which advances the 12-bit address pointer one word at a time. The PMAR receives inputs from the Vector Interrupt Encoder, the Decoded Command and Instruction bus, the RALU bus, and the Program Memory. Outputs from Program Memory are stored in a Program Memory Data Register which is used to implement overlapping normal fetch/execute instructions. (One CP instruction is being fetched from Program Memory while the previous instruction is being executed.) This overlapping is accomplished automatically within the Array Processor, and is invisible to the programmer. When a jump instruction is being executed, the fetched instruction is cycled through, but not executed.

NOTE: The CP cannot modify its own Program Memory. Thus, once a program is loaded by the Host computer, it

Figure 2-10. Control Processor Simplified Block Diagram

2-11
remains unchanged. However, during execution of a program, register values and Data Memory counters can change, and may have to be reinitialized before restarting program execution.

2.5.3 Register and Arithmetic & Logic Unit (RALU)

The RALU contains 16 registers, (RO through R15) that are each 16 bits long. These are used to develop addresses of sources and destinations for the pipeline arithmetic operations. Note that the RALU registers are addressable with 4-bit words (16 register addresses), but that their contents become 16-bit addresses for the Data Memory locations \(2^{16} = 64K\) maximum memory size. Calling out the memory locations for sources and destinations of arithmetic data is accomplished by register-to-register manipulation within the Control Processor RALU. Repetition control for pipeline operations is also executed by the Control Processor, and is accomplished by manipulation of index computations along with conditional jump and skip instructions. Refer to Chapter 5 for the machine instructions used to perform the register-to-register manipulations.

The CP can also access locations in Data Memory as part of its Instruction set on a cycle-stealing basis. This added capability enables the CP to manage its own Data Memory allocation, and relieves the Host (and the application programmer) of much of that burden. Another feature of this capability allows the CP to complete the “odds and ends” of a calculation that are scalar in nature, and thus inefficient for the vector (array) processing of the Arithmetic Pipeline (PA). For example, the CP can use the leading zero count to change the block exponent before transferring data in normalized block floating point format.

The CP accesses locations in the Host via the I/O board, with minimum Host burden. The CP tells the I/O where a block of data can be found in the AP Data Memory, where it is going in the Host memory, and how many words to transfer as a block. The execution of this data transfer to the Host is accomplished by the logic and control residing in the I/O board (refer to paragraph 2.7). Once the instructions are passed to the I/O the CP proceeds to perform its continuing tasks. The I/O returns a “transfer completed” signal when it has performed as directed.

2.5.4 Stack Operation.

Register R0 within the RALU is reserved as the STACK POINTER, and stack operations are accomplished by specific instructions. There are 64 words reserved for the stack. The CP automatically checks to see that a stack instruction is valid, and within the allowable range and location. The stack allows jumping to and from subroutines, interrupts, passing subroutine parameters, etc.

2.5.5 Interrupts

Host-to-AP400 interrupts are handled within the registers of the Vector Interrupt Encoder. Eight levels of interrupt priority are provided. An interrupt mask allows the inhibiting of individual or sets of interrupts. Interrupt enable/disable operations are indirectly controlled by software, using machine instructions.

Design precautions have been incorporated so that interrupt servicing cannot occur at times during the execution of a program such that recovery would not be possible. Instructions of more than one cycle must be completed. For example, four pipeline instructions (comprising the PIPE and four-PAD set) cannot be interrupted.

Interrupts from the AP400 to the Host can be held off as a consequence of the Host setting a bit in a Status Register in the interface board assembly. The Control Processor can cause an interrupt request to the Host only after this specific status bit is enabled.

Interrupts to the AP400 by inputs to the Auxiliary Interface input port are directed by the Status Register in the interface board assembly. Interrupts from the AP400 to the Auxiliary Port are implemented by setting status bits that can be examined by the Host.

2.6 DATA MEMORY (DM)

As shown in Figure 2-11, the Data Memory assembly interfaces with the other functional units of the AP400 via three buses (CCB, DB, and RALU), and via cable with Expansion Memory assemblies, if installed. The basic DM assembly provides 4096 contiguous words of memory on a single board, and space for an additional 4096 words on the same board. Expansion beyond the 8K available on one board is obtained by adding Expansion Memory boards, which are plugged into the main assembly backplane and cabled to the existing boards, as indicated in the illustration.

The DM board performs two primary functions:

1. It provides a buffer (CAB) between the pipeline commands (generated by the CP) and the pipeline execution control signals which control the pipeline proper. The former are developed in a quasi-random sequence, following the program instruction listing, while the latter are developed in a rigidly controlled timing sequence that synchronizes the pipeline setups with the read/write sequence from/to the data memory. (See Fig. 2-7.)

2. It accesses specified data memory addresses where the read/write operations are to be executed. The memory addresses are independently controlled for the Control Processor (CP-DMAR), the Interface (I/O-DMAR), and the Pipeline (PA-DMAR). These control signals (enabling the address registers as shown in Figure 2-11) are developed on a priority basis (PA lowest priority). The PA-DMAR defines the addresses for data sources and destinations in synchronism with PA operations. Thus, when the bus is usurped for other functions, the PA clock is stopped, stealing cycles from the PA operation for other data transfers to and from Data Memory.

As shown in the illustration, the Data Memory also includes the function of determining whether any of the addressed locations exceeds the maximum data memory. The maximum data memory size is determined, by the amount of Expansion Memory installed, and that number is translated into a wire-wrap jumper setting at the back plane (detailed in the Installation Manual). The Data Memory assembly compares the addressed location and transmits an error signal if the address exceeds the maximum memory.
The Command and Address data of 24 bits for the pipeline are received from the CP via the RALU bus (16 bits), as well as directly (8 bits). The 24 bits are latched into the CAB Input Register at the end of the clock cycle. They are then transferred into the CAB Buffer, which can hold up to 64 24-bit words. A complete pipeline instruction set consists of 4 such 24-bit words, so that the Buffer can hold up to 16 instructions for pipeline "passes".

The status of the CAB buffer is monitored during the program execution. When the buffer holds only 3 pipeline instructions, it is "empty", and the PA clock is stopped, preventing any further pipeline operation. The CP clock continues. When the CAB buffer contains 15½ pipeline instructions (62 24-bit words), it is considered "full", and prevents the transfer of any more instructions from the CP until it is emptied below the "full" threshold.

2.7 INPUT/OUTPUT (I/O)

The Input/Output (I/O) card provides all the communications between the AP400 and the Host computer and between the AP400 and devices connected to the Auxiliary Port. A different I/O card is required for each Host computer with which the AP400 is specified to interface. Each card contains the circuitry to carry out the following I/O tasks:

a. Direct the AP400 status: Halt/run, single step, etc.
b. Transfer data to and from the Host under programmed I/O;
c. Automatic DMA transfer of data to/from Host memory;
d. Automatic Auxiliary Port transfer;
e. Access various nodes of the Array Processor for diagnostic testing.
The sections that follow describe the circuitry for the I/O card used to interface the AP400 with a PDP11 computer via the Unibus, as well as the circuitry for using the Auxiliary Port. The Host dependent information will change for particular Host computers. Detailed descriptions of the I/O circuit operation, including detailed schematics, provide a complete description of the I/O capabilities of the AP400.

### 2.7.1 I/O Block Diagram (PDP11 Interface)

Figure 2-12 is a simplified block diagram of the PDP11-AP400 interface. The titles in the blocks are further defined in the complete schematic, reproduced at the end of this chapter as Figure 2-A, 2-B, and 2-C.

The AP400 requires two Host Memory addresses on the Unibus for data and commands, and one Host memory address for the Interrupt Vector.

The AP/Host Interface consists of bidirectional bus transceivers for the Address Bus, and the Data Bus. Incoming data is transferred to the internal memory bus via buffers. Outgoing data is latched from the internal bus in the data register.

Figure 2-12. Interface (I/O) Simplified Block Diagram
The Command Register stores the command the Host wishes the Interface to execute. The Message Register stores the message left for the Array Processor by the Host. All commands originating in the Host affect the Command and Message Registers.

A PROM-controlled Interface Control Sequencer generates all the timing controls required by the Unibus and the Array Processor interfaces.

The Auxiliary Input Port contains a holding register and interface control logic for handshaking. The Auxiliary Output Port also contains a holding register and handshaking logic. Each port has an interrupt line to the control processor.

The Host Memory Address Register (HMAR), and the Word/Control Register (WCR), are utilized in Direct Memory Access (DMA) operations. It should be noted that the Word/Control Register is also used in the operation of the Auxiliary Ports.

The internal interface between the I/O card and the Array Processor is carried out by the Memory Bus, the RALU Bus, and the Command Bus. The Interface Control uses the Command Bus to route data between the Host, the Data Memory, the Control Processor, and itself.

The status of the AP400 with respect to the communications across the interface is indicated on the AP400 front panel, as shown in Figure 2-13. The indicator lights on the panel indicate the SET/RESET conditions of eight of the 16 status bits in the STATUS/MESSAGE Register. The functions controlled by these bits are described below. (Bits 0 through 7 of the MESSAGE Register are not brought out to the front panel).

The Status Register is a "software handshake" register. Each bit is individually alterable by either the Host or the AP400, according to the appropriate protocol. For example, the Host may set the Host-to-AP Interrupt bit, but only the AP may clear it.

Both the Host and the AP may read the Status Register.

The functions controlled by the 8 status bits in the Status Register, as displayed on the AP400 front panel, are described below. The set condition is defined as a logic 1; the reset conditions as a logic 0.

**Status Bit 8: AP RUN**

When reset, this bit inhibits the clock in the Control Processor and in the Pipeline. Thus the Host can stop operation in the AP any time. The AP can use this bit to halt itself.

**Status Bit 9: AUXILIARY OUT**

This bit is used as a programmable output bit for the Auxiliary Output Port. It is not used internally in the Array Processor. It is also available to the Auxiliary Input Port.

**Status Bit 10: AUXILIARY IN**

This bit is used as a programmable output bit for the Auxiliary Input Port. It is not used internally in the Array Processor. It is also available to the Auxiliary Output Port.

**Status Bit 11: INTERRUPT ENABLE**

When set, this bit allows the AP to Interrupt the Host.

**Status Bit 12: AP-TO-HOST INTERRUPT PENDING**

When set, and when Status Bit 11 is set, the INTR REQ is set on the rise of Status Bit 12. Bit 12 is normally set by the Array Processor, and reset by the Host.

**Status Bit 13: HOST-TO-AP INTERRUPT PENDING**

This bit generates an interrupt in the Control Processor. It is normally set by the Host, and reset by the Control Processor.

**Status Bit 14: HI/LO**

When set, this bit points to the high 16 bits of Data Memory or the high 11 bits of Program Memory. When reset, it points to the low 8 eight bits of Data Memory or the low 11 bits of Program Memory.

**Status Bit 15: (Not Assigned)**

This bit may be assigned any function by the user.

### 2.7.2 Host/AP Communications.

There are three modes of communications between the Host and the AP:

a. **Programmed I/O**: Wherein each transfer requires the execution of an I/O instruction in the Host.

b. **Direct Memory Access**: Wherein data is transferred to/from the Host Memory (or any other addressed device) in bursts of up to 16 words.

c. **Interrupt**: Wherein the AP generates a Host processor interrupt and transfers an Interrupt Vector.

### 2.7.3 Programmed I/O.

In **PROGRAMMED I/O**, the PDP Unibus protocol requires that the Host be the "bus master" and the AP be the "bus slave". The Host issues an I/O transfer to one of two sequential device addresses that are decoded by the AP. The low address is the AP Command Address, while the high address is the AP Data Address. When using the Command Address, the Host "writes" to the Command Register and to the Message Register, and "reads" from the Message Register and the Status Register. When using the Data Address, the Host transfers data to/from the AP, as part of the standard Unibus protocol.
Figure 2.14 illustrates the word format and bit assignment of the Command & Message Register for the PIO transfers in the read and write operations.

There are two types of commands:

a. **Immediate**: Wherein the data word that is transferred is interpreted and the command involved is executed during the actual transfer time.

b. **Data or Non-Immediate**: Wherein the transferred command is stored and will be used to route the data that will be transferred with a Data Address.

Table 2.4 contains a listing of some of the commands that can be transferred.

c. Generate and place on the Command bus the necessary command code to direct the transfer of the data in the Interface, the Control Processor, or the Data Memory.

Figures 2.15 and 2.16 illustrate the timing relationships among these control signals for immediate transfer of data.

### Table 2-3

<table>
<thead>
<tr>
<th>HOST TO AP COMMAND CODES — IN HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IMMEDIATE CODES (18)</strong></td>
</tr>
<tr>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>RESET AP ...... F0</td>
</tr>
<tr>
<td>SINGLE STEP ... F2</td>
</tr>
<tr>
<td>CLEAR BIT 08 ...... 80 STOP CLOCK</td>
</tr>
<tr>
<td>SET BIT 08 ...... 82 START CLOCK</td>
</tr>
<tr>
<td>CLEAR BIT 09 ...... 88 AUX OUT</td>
</tr>
<tr>
<td>SET BIT 09 ...... 8A</td>
</tr>
<tr>
<td>CLEAR BIT 10 ...... 90 AUX IN</td>
</tr>
<tr>
<td>SET BIT 10 ...... 92</td>
</tr>
<tr>
<td>CLEAR BIT 11 ...... 98 INTERRUPT</td>
</tr>
<tr>
<td>SET BIT 11 ...... 9A ENABLE</td>
</tr>
<tr>
<td>CLEAR BIT 12 ...... 9C AP TO HOST</td>
</tr>
<tr>
<td>SET BIT 12 ...... 9D INTER. PEND</td>
</tr>
<tr>
<td>CLEAR BIT 13 ...... 9A H TO AP</td>
</tr>
<tr>
<td>SET BIT 13 ...... 9B INTR. PEND</td>
</tr>
<tr>
<td>CLEAR BIT 14 ...... A0 HI/LO</td>
</tr>
<tr>
<td>SET BIT 14 ...... A2 NOT IN USE</td>
</tr>
<tr>
<td>CLEAR BIT 15 ...... A8 PMAR (PC)</td>
</tr>
<tr>
<td>SET BIT 15 ...... AB FLAGS</td>
</tr>
<tr>
<td><strong>REGISTER WRITE CODES: (5)</strong></td>
</tr>
<tr>
<td>IO COUNT ...... 71</td>
</tr>
<tr>
<td>IO ADDRESS ...... 73</td>
</tr>
<tr>
<td>PMAR (PC) ...... B0 8D SCL/LZC/CNT ...... 24</td>
</tr>
<tr>
<td>IO-DMAR ...... 75</td>
</tr>
<tr>
<td>SCL/LZC ...... 25</td>
</tr>
<tr>
<td><strong>HI/LO (STATUS BIT 14) DEPENDENT CODES: (16)</strong></td>
</tr>
<tr>
<td>READ DM, FLIP HI/LO, &amp; INCREMENT IO × DMAR AFTER HI... DD</td>
</tr>
<tr>
<td>WRITE DM, FLIP HI/LO, &amp; INCREMENT IO × DMAR AFTER HI... D5</td>
</tr>
<tr>
<td>READ DM, FLIP HI/LO, &amp; INCREMENT IO × DMAR AFTER LO... D7</td>
</tr>
<tr>
<td>WRITE DM, FLIP HI/LO, &amp; INCREMENT IO × DMAR AFTER LO... C5</td>
</tr>
</tbody>
</table>
Figure 2-15. Host Read of Message & Status Register, Host Write of Immediate or Data Commands

Figure 2-16. Host Read/Write of Data Memory
2.7.3.1 IMMEDIATE Commands

Immediate Commands contain two parts: a lower and an upper byte. The lower byte contains the command code to be stored in the Command Register, while the upper byte contains the message (if one is required) which is stored in the Message Register. When the Host executes a READ of the Command Address, the contents of the Status Register are loaded into the upper byte, and the contents of the Message Register are loaded into the lower byte.

Immediate Commands WRITTEN by the Host affect only the Status Register bit written with the message register. These commands set and reset the eight bits of the Status Register on an individual basis.

Execution of an Immediate Command is accomplished in the following sequence:

When the Host asserts MSYN, the Interface decodes the command address and initiates the sequence to transfer and to store the data on the Host Data Bus in the Command and Message Registers.

If Status Bit 7 is set while Bit 2 and Bit 0 are reset, the sequencer gates the command code onto the Command Bus. PROMs decode the command and generate the signals to change the Status Bits, or cause a hardware reset of the AP, or single step the AP.

Reading or writing the Status Register steals one cycle from the AP Memory Bus.

The command RESET AP generates a hardware reset signal internal to the AP.

The SINGLE STEP command gates a single clock cycle to the Pipeline and causes the Control Processor to execute the next instruction.

2.7.3.2 DATA (Non-Immediate) Commands.

The Data Commands (Non-Immediate) are used to transfer 16-bit words between the Host and the AP. Therefore, they are “two-step” commands. In step 1, the Host loads the command into the AP Command Address (lower). The Interface stores the command in the Command Register. In step 2, the Host transfers a data word using the AP Data Address (upper). The Interface uses the command stored in the Command Register to place the data onto the bus and to transfer the data to/from the Host.

There are two classes of all the “Non-Immediate” Data Commands: Single word Read/Write, and Multiple word Read/Write. Multiple word commands are distinguished by having command bits 7, 2, and 0 all set (1).

2.7.4 Direct Memory Access (DMA).

In the DMA mode, the Unibus protocol is reversed. The AP becomes the master, and the Host becomes the slave. However, before the AP can become the master, it must first request and be granted the bus. The AP can then hold the bus only long enough to transfer up to 16 words.

Before the Interface can be instructed to execute a DMA transfer, the I/O Data Memory Address (I/O DMAR) and the Host Memory Address Register (HMAR) must be loaded with certain values. The I/O DMAR (which is physically located in the AP Data Memory board) be loaded with the starting address of the data to be transferred. The HMAR must be loaded with the starting address in the Host Memory of the data block to be transferred.

The Host Address Bus is an 18-bit bus. Bit A00 is a “byte” control bit, and it is not driven by the AP. Bit A17 is driven by the designated bit in the WCTR (Word Control Register). This bit will be incremented in the event the block transferred crosses the boundary at 65k words of Host Memory.

To initiate a DMA transfer, the Word/Control Register (WCTR) in the I/O board must be loaded with the complement of the number of words to be transferred as well as four control bits (See Figure 2-16). The complement of the number of words to be transferred is referred to as the “throttle count”, because it controls the rate at which large blocks of data can be moved. The throttle count in the WCTR is set by the AP400 in response to the program demands for the transfer of data.

![DMA Formats Host TO/FROM AP](image)
Figure 2-18. Word Count Register

CAUTION
When the throttle count goes to zero, A17, I/O ENABLE, HOST/AUX, and W/RC (Write/Read Control) are set to ZERO. The WCTR is reset by the Power-On and the Reset Command.

The sequence of AP to HOST DMA Operations is illustrated in the flow diagram of Figure 2-19. It is initiated by a Non-Processor Request, asserted on the Unibus. This can occur when the Word/Control Register (WCTR) is loaded, and when the I/O EN bit and the HOST/AUX bit are both set. The Host generates a Non-Processor Grant (NPG) that sets the NPYC. As soon as the bus is idle, the SGSYN is set and starts the Control Sequencer.

The interface asserts BBSY on the bus and will hold the bus for as long as BBSY is asserted.

On starting, the sequencer stores three control bits:
- Transfer is not initiated by the Host
- INTR CYC indicating it is not an interrupt cycle
- Control defining whether the DMA is to or from the Host (W/RC is 1 or 0, respectively).

In the case of DMA to Host, the Interface places the code on the Command Bus, which thereby accesses Data Memory at the location pointed to by the I/O DMAR, and stores the data in the Data Register. The I/O DMAR is incremented at the end of the Command bus cycle.

The sequencer has asserted MSYN on the Unibus and has placed the HMAR and the Data Register on the Unibus. Upon assertion of SSYN by the receiving slave device, the MSYN is negated, and the HMAR and WCTR are incremented. If the throttle count is not zero, the sequencer loops back and repeats the access of Data Memory and transfer to the Host sequence until the throttle count becomes zero.

When the throttle count does go to zero, NPYC is reset, BBSY, NPR, and SACK are negated. After MSYN is negated, the sequencer becomes idle.

DMA transfers from the Host are performed in a similar manner, except that the flows are in the opposite direction. The MSYN is not negated until after the data is stored in the Data Memory.

DMA transfers steal one cycle per word from the Data Memory bus.

Figures 2-20 and 2-21 illustrate the timing for DMA transfers.

Figure 2-19. AP to Host in DMA Operation

2.7.5 Some Programming Considerations Implicit in I/O Transfer Implementation.

Because the Host data word is only 16 bits, only the most significant 16 bits of the AP's data memory are transferred under DMA. There is no provision (as distinct from PIO transfer), for transferring the low order 8 bits. Longer words (24-bit data, for example) must be reformatted in the AP after loading, or prior to unloading to the Host.

Some registers cannot be read. The HMAR and the WCTR cannot be read by either the Host or the Control Processor (AP). Therefore, a certain amount of care should be exercised in their use.

The I/O DMAR and the HMAR registers are incremented with each word transferred and are altered by specific commands. The user is therefore permitted to transfer many word blocks sequentially after setting these two registers to their starting values.

The Control Processor can load the WCTR at any time without regard to the status of the Interface Sequencer. This feature can be used advantageously:

1. To cause an early termination to a DMA transfer;
2. To extend the block size to greater than 16 words.

By loading a throttle count of 15, the DMA in progress can be terminated reliably within one transfer time. By monitoring the I/O DMAR, the Control Processor can transfer as large a block of data as desired once the control of the Unibus has been given to the AP.
Figure 2-20. Host to AP DMA Timing Diagram

Figure 2-21. AP to Host 2-Word Transfer DMA Timing Diagram
This is done by continuously loading the WCTR with a throttle count of 0, and by checking the value of the I/O DMAR for the address of the last word to be transferred. This operation locks out all other Host devices on the Unibus.

Upon completion of a DMA transfer, the interface generates an interrupt to the Control Processor.

The I/O BUSY bit is available to the Control Processor to monitor the status of the interface during DMA operations.

The DMA must write to successive Host memory addresses when more than one word is being transferred in each burst (throttle count not set to 1).

2.7.6 AP Interrupt of the Host

The AP will interrupt the Host when Status Bits 11 and 12 (for the status register) are set. Bit 11 is an enabling bit that is set, or reset, only by the Host. Bit 12 is set by the AP400 as an Interrupt Request, and reset by the Host. The protocol on the Unibus is similar to that during DMA in that the interface must request the bus. When the bus is granted, only the Interrupt Vector is placed on the data bus. The Vector is used by the Host to locate the interrupt service routine.

The sequence of interrupt operation begins with the rising edge of Status Bit 12. At that event, the INTR REQ is set; and, if Status Bit 11 is also set, the Bus Request (BRn) is asserted. As in the DMA interrupt, when the Host asserts Bus Grant (BGn), the INTR CY is set which asserts SACK and sets SQ SYN. The sequencer selects and stores 3 bits which control the sequence. They are:

a. Transfer not initiated by Host;
b. INTR CY indicates an interrupt cycle;
c. Read/write control...not applicable

The sequencer controller then applies the Interrupt Vector, and asserts the INTR Host bus and asserts MSYN. Upon the assertion of SSYN by the Host, the Interface negates MSYN, SACK, and BBSY and returns to the idle state.

INTR REQ is reset upon the completion of the transfer of the Interrupt Vector or by the resetting of Status Bit 12.

The interrupt timing relationships are illustrated in Figure 2-22.

---

**Figure 2-22. AP Interrupt of Host Timing Diagram**
2.8 AUXILIARY PORT

The Auxiliary Port (AUX) provides a high speed digital input/output for data to/from another device. The port is composed of two 24-bit registers for input and output and the necessary handshake control signals. The ports may be used individually as unidirectional ports, or the two buses can be connected to form a single bidirectional port.

Use of the AUX ports is similar to the use of the interface in the Host/DMA mode in that both require certain registers to be set up prior to initiation of any transfer. It should be noted, however, that the Host/DMA and AUX operations are mutually exclusive. That is, only one or the other can be accomplished at a given time. However, programming can overlap Host/DMA with AUX/DMA for more efficient throughput.

2.8.1 Sequence of Operations.

In preparing for an AUX transfer, the I/O DMA must be loaded with the starting address of the table in the Data Memory. The HMAR need not be loaded, and the contents of the HMAR will NOT be affected by the AUX operations.

When the WCTR is loaded with the throttle count and the control bits, the Interface waits until the selected port READY signal is negated. The AUX INTF SYN is set for one cycle of the clock. During this time, the AUX command is placed on the Command Bus, and the 24-bit data word is transferred between the selected port and the Data Memory. The affected READY flip-flop is set, indicating to the port user that the next data word may be loaded into or out of the port. The sequence of operations takes 3 machine cycles to complete.

The throttle count is incremented automatically with Data Memory transfer. When the count reaches zero, the I/O ENABLE is reset, and an I/O DMA Complete interrupt is generated.

Although the WCTR in the interface may be occupied with the AUX operations, the Interface is still able to handle PIO operations with the Host.

The I/O logic handles any conflict between the Control Sequencer and AUX for use of the Command Bus (CCB) by generating control signal ALT CTL DIS (reference schematic B).

The AUX INTF SYN is reset one cycle after the Sequencer has released the Command Bus, and the AUX command is placed on the Command Bus only after the Sequencer removes its command from the CCB.

Contention for the Command Bus is not limited to the AUX and Control Sequencer in the Interface. The Control Processor is also a HEAVY user of that Command Bus.

In those cases where all three users attempt to access the Command Bus, the priority is established as follows:

- The Interface gets the bus (even though the Control Processor is on it); forcing the Control Processor off.
- The Control Processor gets the bus immediately after the Interface releases it.
- The CCBINST signal from the Control Processor holds the AUX port off the bus until the bus is released by the Control Processor.

2.8.2 Input Port.

The Input Port consists of a 24-bit register which is loaded by the external user when the TRS signal is asserted low. The Input Port is always ready (AIP READY is high). When TRS is asserted the AIP READY is reset, causing AUX PORT ENABLE to be asserted. After the data transfer is complete, the AIP READY is set. Refer to the timing diagram in Figure 2-23.

Status Bit 10 is available to the user on the input port connector, and the AUX IN Interrupt is available to the Control Processor. These may be used in an "interrupt" mode to synchronize the transfer of data in the I/O ports to an external event.

2.8.3 Output Port.

The Output Port contains a 24-bit register which is loaded by the Interface after the WCTR is loaded and the AOP RDY is set, and the OP RDY is asserted low. An on-board jumpering is used to establish the logic levels for asserting or negating the output port control/status signals. The jumper is field-installed, and may be changed at any time to interface the AP400 with new peripherals having the opposite logic level protocol. The timing diagram of Figure 2-24 shows the impact of strapping on timing and the operation of OPTRS.

The OPTRS signal strapping option can select ACTIVE HIGH or ACTIVE LOW operation. The output data is present only when OPTRS is in its active state. When restored to the inactive state, the AOPRDY is reset until the next word has been loaded from Data Memory.

Status Bit 9 and the AUX OUT Interrupt to the Control Processor are made available to the user at the output connector. They may be used, as for Status Bit 10, to implement a handshake protocol.

2.8.4 Typical Use of the Auxiliary Port

The Auxiliary Input Port configured as shown in Figure 2-25 can provide all the interface required between an array processor based digital signal processing system and a data acquisition system used to interface to sensors or transducers. The Auxiliary Output Port can be used to interface to a digital or analog subsystem which in turn interfaces to either a display or a control subsystem.

A more detailed example of the use of the input port is provided in Figure 2-26. This configuration uses the AP400 together with a PDP-11/04 Host computer to perform a spectrum analysis of two audio signals connected to the input of a data acquisition subsystem. The data acquisition subsystem consists of the anti-aliasing filters for each channel, a multiplexer to switch between channels, a sample and hold module, a 16-bit A/D converter, and the associated timing and logic circuits. Also shown is the controller and FIFO buffer used to provide inputs to the AP400 for a specific I/O Service Routine which requests 16 words at a time.

The overall sequence of operations is to digitize continuously in real time each of the two audio signals, transfer data to the AP400, perform an FFT on the signals, calculate the complex magnitude of the signals, compute the logarithm of the magnitudes, and continuously transfer the data to the Host computer. The
Figure 2-23. Auxiliary Port Timing Diagram

Figure 2-24. Auxiliary Port Output Strobe Polarity Selection
Host computer can then transfer the results to a display subsystem for presentation of the power spectral density of each of the two original signals.

A consideration for interfacing to the Auxiliary Input Port is the use of the control signals associated with the port.

The Auxiliary Input Port consists of a 24-bit data register and three control signals. For this example, the 8 least significant bits of the 24 bit data inputs are tied to ground reference and the 16 most significant inputs are used for the digitized signals. The use of the control bits is explained by the following sequence of operations:

1. An End of Conversion (EOC) signal from the A/D converter clocks 16-bit words into the FIFO buffer (Refer to Figure 2-26).  
2. When the FIFO has a word in it, a FIFO OUTPUT RDY signal clocks the Input Controller, and generates an IPTR/S.  
3. The trailing edge of the IPTR/S clocks this first word into the Auxiliary Input Port register and causes IPRDY to go into its busy state.  
4. Because the I/O Interrupt Service Routine has not yet acknowledged an interrupt, this first word remains in the Auxiliary Input Port register and the IPRDY signal does not toggle from its busy state.  
5. No further actions occur in the Auxiliary Input Port until an IPINTRPT occurs.  
6. When the FIFO buffer contains 16 words, a FIFO FULL signal clocks the Input Controller and causes an IPINTRPT.  
7. The IPINTRPT causes an interrupt of the AP400 Control Processor and the I/O Interrupt Service Routine transfers the first word (already in the input register) into Data Memory.  
8. After the input transfer from the Auxiliary Input Port Register to Data Memory is complete, the IPRDY signal is asserted (low).  
9. When IPRDY is asserted, it causes the Input Controller to clock IPTR/S and transfers the second word out of the FIFO buffer and into the Auxiliary Input Port Register.  
10. At the completion of this transfer, IPRDY goes to its busy state until the second word is transferred into Data Memory.  
11. After the transfer to Data Memory, IPRDY is again asserted and causes the next IPTR/S to be generated.  
12. Since the input word count in the I/O Interrupt Service Routine has been set at 16, the process of transferring words from the input register to Data Memory will continue until 16 words have been loaded.  
13. After the transfer of 16 words, the I/O Interrupt Service Routine disables the Auxiliary Input Port and the 17th word remains in the Auxiliary Input Port Register until the next IPINTRPT occurs.
Figure 2-26. Auxiliary Port Application Detailed Block Diagram
14. This in turn ceases transitions of iPRDY, and hence iPTRS until the cycle is repeated.

15. The next FIFO FULL causes the cycle to repeat.

It should be noted that provision for the time required for the I/O Interrupt Service Routine to set up for the transfer of data to Data Memory is accounted for by making the FIFO larger than 16 words. That is, the FIFO buffer must be able to handle additional input words from the A/D during the I/O setup time after the iPINTRT occurs. The high rate at which the Auxiliary Input Port can accept data after setup (approximately 1.5 Megahertz) assures that an overflow in the FIFO will not occur.

Another consideration when using the Auxiliary Input Port of the AP400 for real time continuous signal processing is the timing associated with inputting, processing, and outputting data. The composite processing time will determine the maximum signal bandwidth that can be used for this example. The AP400 Auxiliary I/O Port has been designed as a highly flexible interface either to input or to output digital data directly to the AP400 Data Memory. The use of this interface will generally be specific to the application and data format of the user. As an example, an application may require using the interface in a burst mode rather than a continuous processing mode as in this example.

Both HOST and Auxiliary input/output operations have been designed to overlap with processing operations going on in the pipeline. Overlapped operation utilizes the concept of cycle stealing where pipeline processing is delayed for one machine cycle every time an I/O operation requires access to Data Memory. To determine the processing time with overlapped I/O, it is first necessary to estimate the processing time required for a given number of data points without the I/O stealing cycles from the pipeline operations. This is done by determining the number of PAC's, or passes through the pipeline, for each processing function. Next, it is necessary to determine the number of delayed cycles caused by the overlapped process.

Each time a word is brought into Data Memory from the Auxiliary Input Port, pipeline processing is delayed for 1 cycle (160 nanoseconds).

Each time a word is transferred from Data Memory to the HOST, the pipeline is again delayed for 1 cycle.

For each Data Memory access in the I/O Interrupt Service Routine, two cycles are stolen from the processing time. This time is calculated by multiplying the number of interrupt (NI) by the number of Data Memory accesses. The number of interrupts is determined by:

\[ NI = \frac{\text{Number of Words to Transfer}}{\text{Throttle Count}} \]

The number of Data Memory accesses is determined from the actual I/O Interrupt Service Routine, and the throttle count for this example is 16 words.

Each time the I/O Interrupt Service Routine is activated, it interrupts the Control Processor servicing of the pipeline for a corresponding number of cycles and uses the CP to service the interrupt. The effect of the lack of availability of the CP to service the pipeline is determined by considering the possible states of the Command and Address Buffer (CAB) which is filled up by the CP, and in turn supplies inputs for the pipeline. These states are simply CAB full and CAB empty.

When the CAB is full, the pipeline is not dependent on the CP and processing continues (except for the Data Memory Accesses as previously discussed). When the CAB is empty, the pipeline is stopped and waiting for processing time is increased.

The number of lost cycles when the CAB is empty, is equal to the total number of cycles used by the CP I/O Interrupt Service Routine less those cycles that are used to access Data Memory. This number, multiplied by NI, and by the percent of time the CAB is empty, will be equal to another increment of time that must be added to overall processing time.

Below is a summary of processing time in microseconds for this application. The processing consists of first sorting the two channels and performing a 512 point real FFT computation, on the incoming data. Next, the data is reorganized, the complex magnitude is computed, the logarithm of the results are computed, and the results are output to the Host.

A. Uninterrupted Processing Time:

1) Sorting ........................................... 491
2) 512 point real FFT (two channels) ............ 2940
3) Reordering (two channels) ..................... 660
4) Magnitude Approximation (two channels) ...... 1420
5) Logarithms (two channels) ..................... 1000

B. Pipeline Time Lost to AUX I/O Input to Data Memory
(\% Pipe Activity x Number of Words x Cycle Time) . . 130

C. Pipeline Time Lost to API/HOST I/O
(\% Pipe Activity x Number of Words x Cycle Time) . . . 66

D. Data Memory Access by AUX I/O Interrupt Service Routine
(Number of Interrupts x Number of Accesses x Cycle Time x 2) ........................................ 224

E. Time Used by CP to Execute Interrupt Service Routine
(Minus DM Accesses) when CAB is Empty
(Number of Interrupts x % Empty x AUX I/O Cycles). 166

\[ \text{Total Processing Time} \approx 7097 \]

The effective increase in processing time caused by overlapped I/O is approximately 9% for this example, and it is apparent that the basic processing time (uninterrupted) is the determining factor when calculating overlapped I/O processing time.

The allowable input data rate for this example is determined by dividing the total number of words processed by the increment of time calculated for the processing, or:

\[ \frac{2 \times 512}{7.097 \times 10^3} = 144 \text{KHz} \]

The corresponding input data rate per channel is then 72KHz, providing a maximum input signal bandwidth of approximately 35KHz per channel.
Another example of using the Auxiliary Input Port is to operate in a burst mode rather than a continuous input mode.

For stochastic signal processing applications, an adequate sample of the data defines the information for all time intervals, and a burst mode of inputting data to the Auxiliary Port can be used without loss of information.

An AUX I/O Interrupt Service Routine that continually recycles for a predetermined number of cycles would be used for this type of processing. The AUX I/O interface requires three machine cycles or 480 nanoseconds to transfer a 24 bit word to Data Memory. Because the AUX I/O is third in priority for use of the Memory Bus, the Bus will not always be available when AUX I/O wants to transfer data. If the AUX I/O always had to wait, it would add 2 additional cycles to the transfer time. If the Bus is available 50% of the time it is requested by the AUX I/O, then one additional cycle on the average would be added and the allowable input data rate for this mode would be:

\[
\frac{1}{640 \text{ nanoseconds}} = 1.5 \text{ Megahertz}
\]

The detail provided here for using the AUX Input Port is to show how the Auxiliary I/O operations interact with the AP400 processing and control operations. Many alternatives are available to the user to implement efficient Auxiliary I/O routines the unburden the HOST computer and provide real time data acquisition and processing. Analogic is the leader in the field of high speed, high precision data acquisition systems and can provide complete real time data acquisition and signal processing systems to meet user requirements.
3 SOFTWARE

3.1 INTRODUCTION

This chapter describes the AP400 Array Processor software and includes discussions of system software, application software, utility software, and diagnostic software. Applications are discussed together with some generalized programming considerations. This chapter is an introduction to AP400 software. It is not intended to be used in place of a programming manual. Separate programming manuals provide detailed explanations and programming techniques for each level of AP400 software.

The AP400 Array Processor has been designed to allow the user to program in HOST FORTRAN, HOST Assembly Language, or AP Assembly Language. A well documented AP Assembler, AP Linker, and Interactive Debugging Tool (ITD) provide the necessary tools for rapid design, development, and debugging of user programs written in AP Assembly Language.

The complete AP400 Array Processor system includes all of the software at the following levels:

- **System Software**
  - For control of the Host computer and array processor activity.
- **Application Software**
  - For problem solution and real-time tasks.
- **Utility Software**
  - For software preparation and use.
- **Diagnostic Software**
  - For hardware and software fault detection and isolation.

3.2 SYSTEM SOFTWARE

The AP400 system software minimizes programming complexity and provides maximum user flexibility. AP400 system software programs are resident in both the Host computer and the Array Processor.

- **Host-Resident**
  - AP Manager/Driver
- **AP400-Resident**
  - AP Executive
  - AP Executive Service Subroutines

3.2.1 AP MANAGER

The AP Manager is one of the two programs resident in the Host that control access to the AP400 from the Host and provide services to help maintain orderly communication between the two systems.

The AP Manager interacts with Host Functions, providing certain error detection and handling services for them. The design of the AP Manager is as independent of a specific Host CPU Operating System as is possible. Host Operating System dependencies are restricted wherever possible, to the AP Driver.

Host Functions and Host Assembly Language programs utilize identical call formats in calling the AP Manager. The calling format is compatible to that used to access FORTRAN program callable subroutines. In many cases the user's FORTRAN program calls the AP Manager directly, and additional Host Functions are not required.

The AP Manager is a collection of modules that exist in a library, and as such only the routines the user actually requires need to be linked in by the Host Linker (e.g. PDP-11 Linker). The AP Manager has a number of subroutine-callable entry points, rather than a single entry point.

The FORTRAN calling sequence is as follows:

```
CALL subnam (arg1, arg2,...,argn)
```

On PDP-11 system, the HOST Assembly Language calling format for the AP Manager is as follows:

```
MOV  #ARGLST, R5
JSR  PC, subroutine name
```

```
ARGLST:
    BR  1$ ; a calling convention.

1$:
    Parameter
    address list

1$:
```

(Program continues)

Given the following FORTRAN call:

```
CALL KEXFCB (FCBADR, STATUS)
```

The Assembly Language equivalent would be as follows:

```
MOV  #1$R5
JSR  PC,KEXFCB
1$:
```

```
BR  2$  ; (program continues)
```

```
WORD  FCBADR
WORD  STATUS
2$:
```

The AP Manager determines the number of arguments in the parameter list by examination of the "BR 1$", and verifies that the number of arguments is correct and whether or not optional arguments are present.
The AP Manager will vary according to the complexity of the operating system, but is typically under 300 words.

3.2.2 AP Driver

The AP Driver is made up of several distinct components, including both run-time (interrupt servicing) and AP manipulation capabilities. The AP Driver performs direct communication between the Host and AP, including AP initialization, program loading, AP Function initiation, interrupt handling, etc. The AP Driver performs the actual load of Program and Data Memory contents.

Implementation of many of the AP Driver functions and characteristics vary among Drivers for different Hosts and Operating Systems. The following descriptions refer to DEC RT-11.

The AP Driver under RT-11 is divided into two parts to save Host Memory when possible. The Baseline Driver (about 0.5K words) and the Full Driver (routines totalling 1K words). The Baseline Driver is common to all programs that use the AP400. The Full Driver is required (in general) only by programs that need to load AP programs.

All calls to the Driver are followed with a check for an error. Errors are denoted by a carry bit being set; therefore, if the carry bit is not set, no error occurred.

Interrupts from the AP are handled by the Driver in the following manner. If an interrupt is received from the AP that is unsolicited, or unexpected, the Driver will just record the error. The next call to the Driver will return an error. The Driver will not output error messages, or kill the current program.

If the AP Driver receives a message from the AP that it does not understand, it will record the fact that it got an error. The Driver will then return an error on the next call it receives.

3.2.3 AP Executive

The AP Executive is the AP-resident supervisory program. It controls Host access to the AP, maintains orderly communication back and forth between the Host and AP, and provides function dispatching, interrupt, real-time and exception handling services. The AP Executive may be linked fully, partially or not at all with AP Functions before loading the AP400, depending upon the planned use of dynamic linking and loading.

A version of the AP Executive which contains minimal required services and no optional services is referred to as the “Core” Executive.

The AP Executive has optional Interrupt/Trap Handlers that provide various services for use when normal or abnormal interrupts or traps occur. In some cases (as with the I/O Done Interrupt), the Interrupt/Trap Handler code is located within an AP Service Subroutine.

3.2.4 AP Service Subroutines

The AP Service Subroutines provide centralized services for AP Functions such as Data Buffer finding and allocation, Function Control Block fetching, parameter list argument set-up, and Memory zeroing.

AP Service Subroutines are linked and loaded into the AP with the AP Executive if they are needed by the AP Functions being used.

3.3 APPLICATION SOFTWARE

3.3.1 General

The major segments of the Application Software are functions or subroutines which reside in AP Program Memory and AP Data Memory during run time. A library of these functions is supplied with the AP400 from which specific application programs may be assembled. For example, the selection of the Hamming Window Function, FFT Function, Magnitude Approximation Function, Log10 Function, and appropriate management functions would provide the user with all the subroutines or stored programs necessary to construct a spectrum analysis application program.

Because the AP400 also has an Assembler which operates in the Host, a user can develop other routines for applications where additional or unique algorithms must be implemented.

These routines require complementary routines to be located in both the Host and the AP400. These are referred to as Host Functions and AP Functions. A symmetry exists between Host-based and AP-based application software. For nearly every Host Function there exists one or more AP Functions.

Host Functions are routines which call up AP Functions in the AP400 or AP management functions in the AP Manager. Host Functions may be called from Host FORTRAN and/or Assembly Language programs. A convenient way of using the AP400 is via FORTRAN, calling up selected Host Functions from a Host Function Library.

When called, most Host Functions set up Function Control Blocks (FCBs) to invoke specific AP Functions in the AP400. The information placed in FCB's constructed by Host Functions comes from the Host Function call, from defaults written into the Host Function, from default parameters placed in the FCB originally, and from control conditions established through prior Host Function calls.

Once syntactical and certain logical checks have been performed on the Host Function call, and the FCB has been constructed, the Host Function calls up the AP Manager to communicate the address of the FCB and the “Execute FCB” command to the AP Executive. Before, during, and after AP Function execution, the AP Manager performs a variety of interactive operations between the calling task, the Host Operating System, and the AP Driver.

3.3.2 Requirements

To utilize AP Host Functions, the Host Operating System should support a FORTRAN compiler and a Linker (or similar) program capable of accessing Host Functions called by a user's FORTRAN program from the library of Host Functions supplied by ANALOGIC.

AP400 Host Functions may be called from Host Assembly Language as readily as from FORTRAN. If the user's system does not support a Linker (or similar) program capable of library access, Analogic can supply Host Functions as individual object modules rather than as a single library module.
While certain AP400 Host Function-related features require peripherals in order to operate (e.g. run-time function loading requires a random access storage device), most do not, and may be used on any system with enough main memory to support a resident Host Operating System, the user's program, the AP Host Functions required, and the AP Manager/Driver.

The "average" AP Host Function requires (for most systems) under 10 16-bit Host Memory words for a call to the function and approximately 40 words for the function itself.

Host Functions may be called either from FORTRAN or from Host Assembly Language. Both calling methods take full advantage of the ability of the Host Function to set up a Function Control Block for access by the AP and to screen for certain syntactical and logical errors in the call.

The user may, of course, choose to bypass Host Functions and set up single or linked FCB's and call the AP Manager directly.

3.3.3 Function Naming Conventions

Function names serve the purpose of identifying functions, of organizing functions into related groups, of distinguishing among function types or versions, and of relating various levels of Host and AP function-type software.

A 5 or 6-character Host Function name is made up of:

K [Name] [Version or Type]
3-4 chars. 1 digit

Where:
- The prefix K serves to make Host Function names unique from standard or user-written FORTRAN functions, and establishes that a returned status value will be a 2's complement integer value.
- The Name is a descriptive 3- or 4-character name which represents in mnemonic form the objective of the Host Function. A small number of Host Functions may have a Name of 5 characters.
- The Version or Type identifies the function uniquely from other similar (yet different) functions. Other Host and AP Functions may support different data types or algorithms which perform the same task.

A 5 or 6-character AP Function name is made up of:

Q [Name] [Version or Type]
3 or 4 chars. 1 digit

Host and AP Function names relate to each other (for example KMUL1 and QMUL1)

3.3.3.1 Calling the Host Function

- In FORTRAN the Host Function is called by CALL KMUL1 (parameter list).
- In Host Assembly Language the Host Function is called by JSR PC,KMUL1, with the parameter list pointed-to by register R5 (PDP-11).
- The parameter list is identical in both cases.

3.3.3.2 Host Function Implementation (in Host Assembly Language)

- Host Function module (file) name is KMUL1.
- Host Function entry-point label is KMUL1.
- Function is retrieved from a Host-specific-format function library by the name KMUL1.

- In the Host Function, the symbol IMUL1 is equated to the value of the Function ID for this function.

- The symbol IMUL1 is then used in the Function Control Block to supply the Function ID number itself.

3.3.3.3 AP Function Implementation (in AP Assembly Language)

- The AP Function module (file) name is QMUL1.
- The AP Function entry-point label is QMUL1.
- The symbol IMUL1 is equated to the value of the Function ID for the Function.

The symbol IMUL1 and the entry point label QMUL1 are then used in the FUNC directive of the AP Function to allow recognition of that AP Function by the AP Executive, from the Function ID retrieved from the FCB which in turn is retrieved from Host memory.

Usually, the Function Subroutine(s) called by an AP Function will be named similarly to the Host and AP Function. In most cases, the Function Subroutine name will be simply the Host or AP Function name, without the K or Q. As with the Host and AP Functions, Function Subroutine names frequently terminate with a type or version number from 1 to 9. This trailing digit indicates feature variations among functions, such as speed, accuracy, size, flexibility, etc.

In general, the AP Executive has the only access to the AP Function's Qxxxx entry point, through the AP Assembly Language FUNC Directive in the AP Function.

3.4 UTILITY SOFTWARE

Since each Host CPU and Operating System is relatively unique, the actual implementation of AP400 Utility Software will differ somewhat among systems. Precautions have been taken in the design and implementation of all Utilities to minimize these system-to-system differences. These include the use of a modular software structure, that isolates system-dependent features (such as file access and I/O) from system-independent features. The actual implementation of Utility code is in an industry-compatible subset of ANSI-66 standard FORTRAN.

3.4.1 AP Assembler

The AP Assembler allows the user to translate AP Assembly Language programs to produce a linkable/loadable object module and a program listing with flagged errors and instruction-by-instruction machine language code. The AP400 Machine Language code produced in the AP Object/Load Module is eventually stored in either AP Program Memory and/or AP Data Memories.

The AP Assembler allows the user to specify one or more AP source files to be assembled together to optionally produce an AP Object/Load Module and optionally a program listing with the output expressed in Hexadecimal, Octal, or Decimal radix. The Object/Load Module produced by the Assembler may be immediately loaded into the AP400, or it may be linked with other OIL Modules to produce another, single, OIL Module.

The user's control of the AP Assembler is via simple, one-line commands, which may be entered from the keyboard or, on many systems, placed in indirect command files. An example of the assembly of an AP Assembly Language source module MUL1.APA and the
production of an Object/Load Module and Assembly Listing (PDP-11 RSX-11M) is:

>ASM MUL1, MUL1 = SYMDEF, MUL1

The leftmost-named file will be produced by the Assembler and will be called MUL1.APO (AP Object/Load Module); the next file will contain the Assembly Listing and will be called MUL1.LST. The two input files to be assembled together are SYMDEF.APA and MUL1.APA.

3.4.2 AP Linker

The AP Linker combines two or more AP Object/Load Modules produced either by the Assembler or, previously, by the AP Linker itself, and produces another, single new O/L Module. The output of the AP Linker may be loaded into the AP400 or it may be linked with other AP O/L Modules.

For one version, the user supplies the AP Linker with the names of the O/L Modules to be read, the name of the single result O/L Module to be produced, and the name of the file which is to contain the Object/Load Map that is produced as a result of the linking operation. Another version of the AP Linker also allows the user to form and manipulate entire libraries of AP O/L Modules, and automatically selects modules implicitly called-for by those modules explicitly specified in the AP Linker command string.

3.4.3 Interactive Debugging Tool (IDT)

The AP400 IDT is useful for both software and hardware debugging and fault detection and isolation. It provides the user with interactive access to internal elements of AP400 architecture. Programs may be single-stepped, run, or run under (very powerful) breakpoint control. Memories, general registers, I/O registers, and flags may be inspected and modified at will. During IDT execution, the user may specify or select Binary, Octal, Decimal, and/or Hexadecimal radix for input and output. IDT is controlled through the use of simple 2-character commands entered from the keyboard or stored in macro commands. An example of a typical sequence of operations, where the user is about to debug a newly-assembled (and/or linked) Object/Load Module, follows. In the example, the user has selected the Hexadecimal radix for input and output. The user's entries are underlined in this example, and always follow the IDT > prompt.
**TYPICAL SEQUENCE OF OPERATIONS WITH IDT**

<table>
<thead>
<tr>
<th>IDT &gt; RL ROUTN1</th>
<th>User reads program ROUTN1.APO from the default system device and loads it into AP Program and Data Memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDT &gt; D 125.300</td>
<td>User places the value 300 (Hex) into AP Data Memory location 125 (Hex).</td>
</tr>
<tr>
<td>IDT &gt; D 124</td>
<td>User requests the contents of AP Data Memory location 124. IDT responds.</td>
</tr>
<tr>
<td>IDT &gt; D S</td>
<td>The user requests that a series of Data Memory locations be displayed.</td>
</tr>
</tbody>
</table>

```
0124 07F4A0
```

0124 07F4A0

0125 000300

The “S” argument, when used with Data Memory, Program Memory, or Register Manipulation commands, indicates that either the next 10 memory locations or all 16 general-purpose CP registers should be displayed. In the case of memories, the “S” argument also causes IDT’s address pointer for that memory to be stepped ahead by 10 (Decimal).

0126 C000FF

0127 000000

0128 000000

0129 000001

012A FFFF

012B 002000

012C 340000

012D 355001

IDT > R 7.  

```
734F0 2D00
```

IDT > R 7

The user displays the contents of CP General Register 7, with the option to modify its contents if necessary, or leave it intact (option exercised).

IDT > EX

The user requests performance of the Program Memory Address Register, during execution.

IDT > PC

0054

IDT > R 8

```
8 01FE
```

As well, the user checks the value of CP register #8, since in this example, this program should not be executing beyond Program Memory location 42 until the contents of CP register #8 have gone higher than 200.

<table>
<thead>
<tr>
<th>IDT &gt; HX</th>
<th>Since this condition is unexpected by the user, he directs IDT to halt AP Execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDT &gt; BK 2</td>
<td>An IDT Breakpoint (#2) will be set to rapidly determine the cause of the routine executing beyond Program Memory address 42, with CP register #8 containing the proper value.</td>
</tr>
<tr>
<td>PMAR GT' 42</td>
<td>A defined Breakpoint may contain any reasonable number of conditions of many types; their truefalse states will be continually AND’ed together logically during subsequent AP program execution. If more than one Breakpoint is defined, then the T/F outcome of each will be OR’ed together during execution.</td>
</tr>
<tr>
<td>R8 LE 200</td>
<td>The user directs the IDT to set the PMAR to 20, preparatory to breakpoint execution.</td>
</tr>
<tr>
<td>END</td>
<td>The user directs IDT to begin breakpoint execution. IDT will rapidly single-step AP program execution, each time checking for a logical AND of “true” for the set of breakpoint conditions specified.</td>
</tr>
</tbody>
</table>

**BREAKPOINT 2 CONDITIONS MET!**

IDT announces that the program has attempted to execute beyond Program Memory address 42 before the value in CP register #8 exceeded 200.

A quick check of CP Register 8 shows that its contents are not appropriate for this program’s execution beyond Program Memory location 42.

A check of the PC shows exactly where, during program execution, the set of conditions occurred. IDT replies that the PMAR is now 43.

The user may now inspect registers, Data Memory locations, flags, or other AP structures to determine the cause of this error. As well, the user might choose to continue execution by single-stepping the program “by hand” and inspecting the full machine state after each instruction execution.
3.5 DIAGNOSTICS

3.5.1 General
A set of diagnostic software programs is included as part of the standard software package delivered with each AP400. These diagnostics provide a user capability to check various parts of the AP400 and to isolate faults if a malfunction within the AP400 is suspected. The diagnostics allow the user to localize a suspected malfunction to a board level and to determine the nature of the malfunction.

3.5.2 Typical Diagnostic Program
A typical diagnostic summary is shown below:

<table>
<thead>
<tr>
<th>TEST NAME:</th>
<th>ADT007</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST TYPE:</td>
<td>Data Memory Logic Test.</td>
</tr>
<tr>
<td>DESCRIPTION:</td>
<td></td>
</tr>
</tbody>
</table>

TWO ACCUMULATORS IN PIPELINE ARE INITIALIZED. THE PIPELINE THEN GENERATES A SUCCESSION OF NUMBERS. THE CAB ADDRESSING PUTS THEM INTO SUCCESSIVE MEMORY LOCATIONS. THE CONTROL PROCESSOR CHECKS THAT EACH MEMORY LOCATION HAS THE RIGHT CONTENTS.

THE TESTS REPEAT TO INSURE THAT EACH CAB LOCATION IS TRIED FOR NEARLY ALL VALID DATA MEMORY LOCATIONS.

HOST CPU REQUIREMENTS:
NONE (AP MEMORY SIZING IS SELF-CONTAINED)

OPERATION:
NO OPERATOR INTERVENTION REQUIRED

INTERPRETATION OF RESULTS:
FAILURE TO EXECUTE TO COMPLETION IS LIKELY TO MEAN THAT THERE IS A HARDWARE FAILURE ON THE DATA MEMORY CARD IN THE VICINITY OF THE SCRATCHPAD CHIPS.

WHEN PROCESS TERMINATES ON ERROR, THE FOLLOWING ARE REGISTER CONTENTS:

R2-LOCATION OF MEMORY WORD UNDER TEST
R3-HIGHEST DATA MEMORY LOCATION TO BE TESTED
R4-FLAG STEPPING FROM 8 DOWN TO 1 INDICATING TEST VARIATION
R5-NUMBER OF NO-OP PACS BEFORE TEST PACS
R6-VALUE READ FROM LO PART OF DM WORD
R7-VALUE READ FROM HI PART OF DM WORD
R8-EXPECTED VALUE OF TEST

EXECUTION TIME:
ABOUT 2 SECONDS PER 4K OF DATA MEMORY
3.6 PROGRAMMING CONSIDERATIONS

3.6.1 Introduction

When implementing an application with an array processor, there are certain choices available to the user. Choices relating to overall application requirements are typically: throughput speed, accuracy, memory size, host burden, interfacing hardware, and development time/cost. Complementing these are programming choices relating to processing algorithms and data integrity, and programming considerations including selection of rounding or truncating, required number of processing iterations, scaling techniques, and table-based function argument resolution.

After the key questions of AP400 hardware configuration and computational specifications have been answered, there are other software development choices relating to programming level, selection of Host/AP I/O routines, and the use of Auxiliary I/O. Some considerations are included in this section to provide a better understanding of the features of the AP400.

3.6.2 Programming Level Choice

The user has access to the full computational and logical power of the AP400 on any or all of several programming levels:

**FORTRAN**

Powerful FORTRAN higher-level language function calls provide full access to the AP400 Function Library with a minimum of user-programming and interaction with the internal Array Processor Operation.

**HOST ASSEMBLY LANGUAGE ...Two Key Methods**

When system throughput speed, and flexibility must be maximized, the user can make significant gains by programming in Host Assembly Language. The user can make calls to Host Functions from Host Assembly Language (exactly as from Host FORTRAN); or, for further improvement, the user can make calls to AP Functions via chained Function Control Blocks, rather than use individual calls to the AP by individual Host Functions.

**AP ASSEMBLY LANGUAGE ... Several Methods**

When system performance must be optimized, or unique capabilities not available in existing AP400 functions are required, the user may readily achieve his objectives through program development in AP Assembly Language. This may be done, simply, by combining two or more existing AP Functions, with little or no actual programming taking place or, by using existing AP Functions and Service Subroutines in different program configurations. The user can also create his own functions, directly accessing the AP400 Pipeline and even I/O when necessary. Full flexibility in the use of AP400 computational and logical resources is available to the user via the Assembly Language of the Array Processor. A vertical architecture is used. Registers, flags, the arithmetic pipeline, and all other internal structures are available to the User via individual one to four-word instructions ranging from simple two-register operations to more complex multi-operation macros. The result is a familiar minicomputer type Machine and Assembly Language with the benefit of powerful arithmetic capability.

3.6.3 Number Formats

An arithmetic computation such as an algebraic addition can produce a result with one more bit in it than in either operand. Hence, some attention must be paid to how data gets scaled before, during, and after a series of additions and subtractions. Recall, that all floating point numbers use two parts, a fraction (sometimes called value or mantissa) and an exponent (sometimes called scale factor or characteristic). To represent numbers in **full floating point**, each number in an array is represented explicitly with both a fraction and an exponent. In **block floating point**, a common exponent is extracted that applies equally to all numbers in the array or vector, and the individual numbers in the array are represented relative to that common exponent.

For example:

- **Real Number Array**
  
  (0.01592, 0.00375, 0.00048)

- **Full Floating Point**
  
  (10⁻¹x 0.1592, 10⁻² x 0.375, 10⁻³x 0.48)

- **Block Floating Point**
  
  10⁻¹x (0.1592, 0.0375, 0.0048)

When full floating point numbers are added, before the fractions can be combined, the exponents must be compared to see which is the smaller number (assuming both are normalized). Then the fraction of the smaller number is downsized by as many places as the difference in exponents to align the decimal points. Additional operations are sometimes needed afterward to normalize the result. That is, to provide a full fraction for the word size available and assign the corresponding exponent.

When two vectors are being added using full floating point numbers, the comparison, calculation of the amount of shift, and the actual shifting must be done for each number pair for the two vectors. This is not necessary with block floating point numbers as used in the AP400. In block floating point, only a single exponent comparison between the two vectors is needed and then all decimal points within each vector will be aligned. Accumulating numbers within a vector, such as occurs when implementing a moving average or approximate integration, is simpler in block floating point, since with only a single exponent, no comparison, calculation of shift, or shifting need occur.

For the AP400, the precision of each word within a vector is one part in 2²⁰ or 0.00001%. The dynamic range within a block representing a vector, as measured for variables such as level, amplitude, magnitude or linear terms is 138 decibels. Additional dynamic range can be obtained through use of double precision word formats, for variables such as energy, power, magnitude squared, or quadratic terms.

3.6.4 Block Floating Point Implementation

Block floating point rather than full floating point was selected for implementation in the AP400 to produce an array processor capable of executing high speed vector computation with much simplified, and less iterative hardware. This, in turn, provides a significant
reduction in price for this new generation of array processors with an inherent increased reliability resulting from fewer components. As well as a reduction in arithmetic hardware, the elimination of the continual up and down shifting realizes a significant reduction in memory from what is required in full floating point.

Implementation of block floating point processing in the AP400 causes each vector to be "tagged" with two numbers: an exponent common to all elements of the vector, and a count of how many upshifts are needed to block-normalize the vector.

Block-normalizing provides for the maximum precision available within the vector. Although the "tag" keeps track of the exponent required for block normalizing, the actual exponent for the block is selected after downshifting within the block to prevent dropping higher order bits in a pipeline computation. This exponent selection is referred to as scaling and is based on the minimum leading zero count (LZC) within the vector and how much number growth can occur at each pipeline pass of the algorithm being implemented.

After a block of data is processed in the pipeline, the resultant vector has associated with it a new block exponent as well as a new LZC. The AP400 is configured so that a maximum number growth of three can occur in a single pipeline pass. If there are to be multiple passes and the scaling cannot be handled within the word format, a programmed routine (using one of the PAC's) can be used to automatically handle the number growth. This is done by monitoring the LZC of the data as it leaves the pipeline and using this value to control a programmable shifter within the pipeline to operate on the next pass through. The number of shift positions required in this programmable shifter needs only to be large enough to handle the largest number growth (three) that can occur in one pass of numbers through the pipeline. The Control Processor, using a specified subroutine, does the actual block exponent manipulation to respond to the shifting required in the pipeline and keeps track of how much shifting has been done via a "Pipeline Scaling Register".

This unburdening of the user is an example of the successful implementation of a cost effective feature, the user of block floating point.

Another example is the automatic conversion to block floating point format. Function subroutines which are callable either from Host FORTRAN of Host Assembly Language, automatically convert the data being transferred to or from the Array Processor. For example, Host FORTRAN function: CALL KHIAB (NSIG, 1, 1024), causes 1024 words of Host Integer data (NSIG) to be transferred to the AP400 Data Memory and placed in Data Buffer #1 in block floating point format. Also, CALL KABHF (ANS, 20, 513) causes 513 words of block floating point data to be transferred from Data Buffer #20 (in AP400 Data Memory) to 513 locations in Host memory defined as (ANS). This transfer includes the automatic conversion from block floating point in the AP400 to full floating point in the Host.

A final consideration in the implementation of block floating point is the reduction in the number of machine operations required relative to the use of full floating point. As an example, the number of shift operations done in performing the "butterfly" in a RADIX 2 FFT is reduced by more than two thirds when implemented in block floating point rather than full floating point.

3.6.5 Sample HOST FORTRAN Program

Figure 3-1 is an AP400 application written in HOST FORTRAN. The program sequence performs a real FFT on 1024 data points read from a file on disc, followed by a 3-point digital filter, a magnitude approximation, and an averaging over 50 spectrum.

The key Function calls where the heaviest AP utilization and HOST-AP interaction occur are:

- CALL KHIAB
- CALL KFVSC
- CALL KTHPFC
- CALL KCMPAR
- CALL KADD

The program shown is in a non-linked format and overhead time is required for every HOST Function call to the AP. By linking or chaining together those calls where the heaviest HOST and AP interaction occurs, a single HOST Function call can be used to replace several calls. The can be done by linking the HOST Functions by a HOST Linker (e.g., DEC Linker) and the corresponding AP Functions by the AP Linker. This type of linking requires the user to write AP Assembly Language Code to perform the links and also requires the writing of a new Function Control Block in HOST Assembly Language. The overall result of this linking or chaining will be an increase in throughput speed for the application program.

The function KWAIT is appended to the program to allow the AP to complete processing before the Host starts printing out the answer. This is used to allow the AP to run asynchronously at its fastest speed, while the Host interrupts only after each AP processing stage is completed.

3.6.6 Table-Based Functions

The Characterizer Stage of the Pipelined Arithmetic Unit can be used for high speed computation of table-defined functions. For example, the same general purpose linear interpolation formulae can be used on different tabular data to form functions such as logarithms, square roots, sines, and reciprocals. In addition, table-based functions can be used to calibrate or linearize data that has been input to the AP400 from transducers or sensors before using the data in a specific signal processing function. In a real time signal processing operation, this provides the capability to calibrate "on-the-fly" and the calibration tables can be updated as often as required.

An example of the use of a linear interpolation algorithm to modify data that has been input to Data Memory is presented to illustrate how table-based functions are performed in the AP400. Figure 3-2 shows an arbitrary function \( y = f(x) \), that is used to modify the data \( x \). The function \( y \) could be a calibration compensation curve to linearize a known non-linear response of a transducer.

To begin, consider the function \( y = f(x) \) to have been approximated by 64 linear segments, defined by
DIMENSION ANS(513), NSIG(1024), FLTR(2)

CALL KRESET
CALL KLOAD (1, 'APPROG')

N = 50
FLTR(1) = .25
FLTR(2) = .25
CALL KSETIW(0)
CALL KHFB (FLTR,4,2)

CALL KALDB (513,20)
CALL KZRB (20)
XX = 1/FLOAT (N)
CALL KHFB (XX,6,1)

CALL ASSIGN (7, 'INDAT.DAT')
xxx
xxx
DEFINE FILE 7 (N,1024,U,IRECN)
D0 1000 I = 1,N
READ (7) NSIG

CALL KHIAB (NSIG,1,1024)
CALL KFFTR1 (1024,2,1)

CALL KFVSC (10,2)
CALL KTHPFC (2,10,4)

CALL KCMBAR (3,2)
CALL KADD (20,3,20)

1000 CONTINUE
CALL KMULS (20,20,6)

CALL KABHF (ANS,20,513)

CALL KWAIT
PRINT 10, ANS
10 FORMAT (5F16.7)

CALL KEXIT
END

Reset the AP400.
Load AP Program/Data Memories from an AP400 Ob-
ject/Load Module stored on disc.
Establish the number of iterations for this process.
The filter to be used will be
.25, 1., .25.
Transfer the two end-points of the filter into the AP; call
it Data Buffer #4 (DBF 4).
Allocate DBF 20 required later for summation of results;
DBF 20 has space for 513 values.
Zero-out DBF 20 before starting summation.
Compute reciprocal of number of points.
Transfer the reciprocal of the number of points into the
AP; call it Data Buffer #6.

Prepare to read data from a file on disc.
Perform the read-and-process operation "N" times.
Read 1024 points from an unformatted file into array
NSIG.
Transfer 1024 points into the AP.
Perform Forward FFT on contents of DBF 1, and place
results in DBF 2.
Re-order the data in DBF 2, placing it in DBF 10.
Perform a three-point filtering operation on the data in
DBF 10; results go to DBF 2.
Perform a complex magnitude operation on the contents
of DBF 2; results go to DBF 3.
Sum the results of the most recent operation (DBF 3) in-
to previous results (DBF 20).
End of the iterative procedure.
Multiply each point in data set (DBF 20) by the inverse of
N, thus averaging each of the 513 respective result
points.
Transfer the 513 result points from DBF 20 to the array
ANS in the Host.
Wait for the last AP operation to complete.
Print the result.
Exit from this program through the AP Manager.
break-points \( X_{T1}, X_{T2}, \ldots, X_{T64} \). The Characterizer Stage of the PA is programmed to operate on the first 6 bits (\( 2^6 = 64 \)) of the truncated input data word and use these leading bits to access Data Memory. Prior to accessing Data Memory, an offset address \( To \) is added to the truncated version of the data word. Since tabular data may be loaded into any contiguous space in Data Memory, this offset is the starting address of the tabular data. (Refer to Figure 3-3 for a pictorial representation of the operations being discussed). The combined address is then used to access the slope \( m \) and the intercept \( b \) of the straight line between \( X_{T1} \) and \( X_{T2} \).

The truncated data \( X_T \) is then the argument for the two values \( m(X_T) \) and \( b(X_T) \).

The pictorial of Figure 3-3 shows how the values of \( m(X_T) \) and \( b(X_T) \) are used to compute the function \( y = X \cdot m(X_T) + b(X_T) \) where \( X \) is now the original 24-bit data word and \( (X_T) \) is the first 6-bits. 24-bit data is stored in both \( S1R \) and \( S2R \), since two table-based functions can be performed in the PA at the same time. The table starting address in Data Memory \( To \) is used as the source address \( S3R \).

It is also possible to perform a two-dimensional table-based function with the Characterizer Stage configured to use the leading bits of two input data values \( S1R \) and \( S1I \). The table data is loaded into Data Memory so that the combined two-dimensional argument is the address used, together with the offset \( To \), to access table parameters.

![Figure 3-2. A Typical Linear Approximation Function](image-url)
Figure 3-3. Implementing A Linear Approximation Function by Table Lookup
4.1 INTRODUCTION

This chapter lists some of the functions included in the Standard Library of Host Functions that are supplied by Analogic for use with the AP400 Array Processor. This chapter also describes how the function call is implemented by the use of Function Control Blocks. Refer to the AP400 Function Reference Manual and AP400 Processor Handbook for a complete list of these Functions and instructions for their use with the AP400.

Although the functions described in this chapter are presented in the standard one-line FORTRAN call format, the same function may be called in the Host Assembly Language. In the latter case, the arguments which make up the parameter list are called in separate lines of instructions rather than on the one line. The format in this chapter, however, uses only the one-line, FORTRAN format.

To execute array processor functions in response to a Host function program, the AP400 Program Memory must first be loaded with the AP-resident function. These may be downloaded as a complete library before any processing; or they may be selectively loaded before their use in a particular program; or they may be loaded "on the fly" as called in the program.

4.2 FUNCTION CONTROL BLOCKS

The elements of the standard function format are represented in a Function Control Block (FCB), used in the communication between the Host and the AP400 in implementing that function. When the Host program instruction calls one of these functions, the Host Function's response is to set up a Function Control Block and to move the arguments of the Host Function Call into appropriate registers in this block. Then, when the Host-resident AP Manager and Driver pass the call to the AP400, the AP400 is able to retrieve the parameters defined in the FCB and execute the function in the AP400. (For more details on the sequence of this interface, refer to the AP400 Function Reference Manual.)

4.2.1 FCB Structure

The structure of the FCB consists of two parts: a main part in which the structure is fixed, and a secondary part in which a variable structure provides the flexibility to allow a variety of parameter list formats to be communicated. Figure 4-1 illustrates the two parts of the FCB Format; the second part is typical of and may vary from one function to another. The example of Figure 4-1 is representative of a single FCB, where the two parts are attached. An alternate FCB format "links" the two parts.

4.2.2 FCB Elements

Each element of the FCB identified in Figure 4-1 represents one word in Host memory. On most systems, each Host Memory word consists of 16 bits. Where the element is a physical Host Memory address, it is possible that such an address can exceed the capacity of a 16-bit word. Therefore, an allowance is made for two 16-bit words to define the Host Memory Address. When only one word is required for the address, the lower-addressed word will be set to 0. Table 4-1 provides a description of the Function Control Block word elements.

<table>
<thead>
<tr>
<th>Word</th>
<th>Element Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Function ID Number</td>
</tr>
<tr>
<td>1</td>
<td>Control Information</td>
</tr>
<tr>
<td>2</td>
<td>Done Flag</td>
</tr>
<tr>
<td>3-4</td>
<td>Link to Next FCB</td>
</tr>
<tr>
<td>5</td>
<td>Parameter List Type</td>
</tr>
<tr>
<td>6</td>
<td>Number of Arguments</td>
</tr>
<tr>
<td>7</td>
<td>Parameter List Length</td>
</tr>
<tr>
<td>8-9</td>
<td>Host Memory Address</td>
</tr>
<tr>
<td>10-11</td>
<td>Data Buffer ID Number</td>
</tr>
<tr>
<td>12</td>
<td>Other Argument</td>
</tr>
<tr>
<td>13</td>
<td>Other Argument</td>
</tr>
</tbody>
</table>

Figure 4-1. Function Control Block Contents
### Table 4-1
FUNCTION CONTROL BLOCK ELEMENTS — DESCRIPTION

<table>
<thead>
<tr>
<th>FCB ELEMENT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ID Number</td>
<td>The Function ID Number is associated with, and recognized by, a particular AP Function. It is a 16-bit positive numeric code, where the values 0 - 32767 are reserved for use by ANALOGIC, and 32768 - 65535 are available for the user.</td>
</tr>
<tr>
<td>Control Information</td>
<td>Individual bits in the Control entry specify AP action in a variety of situations. A typical control instruction may require the AP to interrupt the Host when the AP has finished with a function.</td>
</tr>
<tr>
<td>Done Flag</td>
<td>The Done Flag word is set to 0 by the AP while the AP is processing a FCB. The Done Flag word is set to a positive, non-zero value if and when the AP completes the individual FCB successfully. The Done Flag is set to a negative value to reflect an error condition should an error occur during processing.</td>
</tr>
<tr>
<td>Link to Next FCB*</td>
<td>Host Memory address of the first word of the next FCB in a linked list (chain) of FCB's. If this is a single FCB, or if it is the last FCB in a linked list, this entry is set to 0.</td>
</tr>
<tr>
<td>Parameter List Type</td>
<td>Identification of the contents and format of the list of arguments that make up the FCB Parameter List. (See “Parameter List Types”.)</td>
</tr>
<tr>
<td>Number of Arguments</td>
<td>Specifies the number of arguments in the FCB Parameter List. The Control and Done Flag arguments (above) are NOT counted, since they are in the fixed structure of the FCB and are always present.</td>
</tr>
<tr>
<td>Parameter List Length</td>
<td>Specifies the length of the following Parameter List, in Host Memory Words. This information is useful to the AP Executive when it fetches an FCB from Host Memory.</td>
</tr>
<tr>
<td>Host Memory Address*</td>
<td>Host Memory Address of the first word of data, which may be a scalar, vector, matrix, complex pairs, etc. The AP Function will utilize this address in accessing data. The first (lower-addressed) word contains the high-order address bits, and the second (higher addressed) word contains the low-order address bits. In application software, where it is known that a Host Memory Address is restricted to 16 bits or less, and that the AP Function does not use standard AP Executive Service Subroutines to handle the address, only one word need be used.</td>
</tr>
<tr>
<td>Data Buffer ID*</td>
<td>The 8-bit ID of a Data Buffer which already resides in AP Data Memory, or which is to be established by the AP Function called. The word following the Data Buffer ID is ignored, but must be allocated if the standard Parameter List Setup Service Subroutines of the AP Executive are used by the AP Function called. In application software where it is known that the AP Function does not use standard AP Executive Parameter List Setup Service Subroutines to handle the ID, only one word need be used.</td>
</tr>
<tr>
<td>Other Arguments</td>
<td>Miscellaneous arguments which are defined by the specific AP Function. These may include actual values, pointers to AP Real-time Data Acquisition Ports, etc.</td>
</tr>
</tbody>
</table>

*Requires two memory words.
4.3 FUNCTION PARAMETER LIST TYPES

A number of standard parameter list types have been defined for AP400 functions. These serve to limit the variability that might otherwise appear among parameter list types for various functions, and allow Analogic to provide a reasonable number of parameter list handling routines for use by AP Functions in the interpretation of Function Control Block parameter list contents.

User-written AP Functions may utilize any of these standard, supported, parameter list types; as well, they may define and use unique types specially suited for a particular application.

The following list describes each of the currently-supported parameter list types. The VAL argument always implies a single 16-bit value, for parameter list types 1 through 8. The HMA argument always describes a doubleword Host Memory Address, and the DBI argument always describes an 8-bit Data Buffer ID stored in the first (lower-addressed) of two words.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No arguments, or one or more arguments defined by and for a specific function.</td>
<td>VAL, VAL</td>
</tr>
<tr>
<td>1</td>
<td>One or more Data Buffers in AP Data Memory</td>
<td>DB1a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DB1a, DB1b, ...</td>
</tr>
<tr>
<td>2</td>
<td>A single-word integer value, followed by one or more Data Buffer Identifiers</td>
<td>VAL, DB1a</td>
</tr>
<tr>
<td>3</td>
<td>A single-word integer value, followed by one or more Host Memory addresses</td>
<td>VAL, HMAa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL, HMAa, HMAb, ...</td>
</tr>
<tr>
<td>4</td>
<td>A single-word integer value, followed by one Host Memory address and one or more Data Buffer Identifiers</td>
<td>VAL, HMAa, DB1a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL, HMAa, DB1a, DB1b, ...</td>
</tr>
<tr>
<td>5</td>
<td>A single-word integer value, followed by two Host Memory addresses and one or more Data Buffer Identifiers</td>
<td>VAL, HMAa, HMAb, DB1a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL, HMAa, HMAb, DB1a, DB1b, ...</td>
</tr>
<tr>
<td>6</td>
<td>A single-word value, followed by one Data Buffer Identifier, and one or more Host Memory addresses</td>
<td>VAL, DB1a, HMAa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL, DB1a, HMAa, HMAb, ...</td>
</tr>
<tr>
<td>7</td>
<td>One Host Memory Address and a single Data Buffer Identifier, followed by any number of single-word integer values</td>
<td>HMAa, DB1a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAa, DB1a, VALa</td>
</tr>
<tr>
<td>8</td>
<td>One or More Host Memory addresses</td>
<td>HMAa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAa, HMAb, ...</td>
</tr>
</tbody>
</table>
4.4 CLASSIFICATION OF HOST FUNCTION CALLS

Host (and AP) Functions may be classified by the degree of demand on the computational/logical resources of the AP. In the latter mode, the functions can be grouped as indicated below:

AP Resource Management:
Functions that are generally used to control AP operation and which determine certain status information of and for the AP. These functions are actually part of the AP Manager and Driver. Examples of functions in this category are KSEITIW and KWAIT.

AP Data Memory (Data Buffer) Management
These functions control the use of data buffers in AP Data Memory or allow the retrieval of status information regarding the Data Buffer area. Examples of functions in this category are KALDB and KDSBP.

Input-Output Operations
These functions are used in the transfer of data to and from the AP400 and the Host and Auxiliary Ports. They include the operations to transform the data into compatible formats for the devices/computers involved. Examples of functions in this category are KHFAB, KABHI, and KABAX.

Logical Data Manipulation
These Functions are intensive in data movement and logical operations, but perform little or no calculations. Examples of functions in this category are KDBDB, and KBRVR.

Straightforward Computation
These are functions which are generally non-iterative, and which perform limited calculations without table lookup. Examples of functions in this category are KMUL, KMLCS, and KCONJ.

Most functions that operate upon two or more AP Data Buffers, or that use at least one source and one destination Data Buffer, may be performed upon the same AP Data Buffer. For example, the contents of AP Data Buffer 71 may be squared "in place" from FORTRAN, via

CALL KMUL(71,71,71).

Sophisticated Computation
These are functions which make extensive use of the Arithmetic Pipeline and logical capabilities of the AP400. They frequently use table lookup operations in their implementation. Examples of functions in this category are KFFTR2, KTHPFC, and KSN.

Host Functions may also be classified as simple, where one Host Function call initiates one AP Function Call; or as compound, where one Host Function Call initiates two or more AP Functions, called individually, or through linked FCB's.

These Host Functions may call upon a series of AP Functions via linked FCB's to accomplish frequently required multi-step operations. For example, in performing the Convolution Function, the AP implements the operation by multiplying the kernel and the FFT of the data and then performing the inverse FFT of the product. Accomplishing these operations in response to the call for a "Convolution" links the separate FCB's that perform the FFT of the data, the vector product, and the inverse FFT, .... all independently of Host Intervention.

AP RESOURCE MANAGEMENT

KSEITIW

SET IMMEDIATE/WAIT RETURN MODE
CALL KSEITIW (WAITCD)
WHERE:

\[
\text{WAITCD} = 0 \quad \text{to initiate "immediate-return" mode.}
\]

\[
\text{WAITCD} \neq 0 \quad \text{to initiate "wait-untill-done" mode.}
\]

COMMENT:
This function sets a flag in the AP Manager, which determines whether control is returned to the user program after a Host Function has set up a Function Control Block (FCB) or only after waiting for the task initiated by the FCB to complete.

KWAIT

WAIT FOR ALL FCB's TO COMPLETE
CALL KWAIT
COMMENT:
This function waits for all FCB's in a chain to complete before returning to the caller.
### AP DATA MEMORY (DATA BUFFER) MANAGEMENT

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KALDB</strong></td>
<td></td>
</tr>
</tbody>
</table>
| **ALLOCATE AP DATA BUFFER**  
**KALDB (VAL, DB1a)** |
| **WHERE:**  
**VAL =** | Size of AP Data Buffer required, in AP Data Memory words. "VAL" must be a single-word integer variable or constant. The DBF's block/exponent/NSN word is not included in this count. |
| **DB1a =** | ID to assign to AP data buffer to be allocated. "DB1a" must be a single-word integer variable or constant. If the DBF was previously allocated, it must be of size equal to specified size (VAL). |
| **COMMENT:** |
| This Host function calls up a corresponding AP function in the AP400, which in turn calls up the selected "Allocate AP Data Buffer" routine. |

| **KDBSP** |
| **DETERMINE AVAILABLE DATA BUFFER SPACE**  
**KDBSP (HMA)** |
| **WHERE:**  
**HMA =** | Is the location in Host Memory to place the result. "HMA" must be a single word integer variable. The value returned is a magnitude number, so if there is more than 32K words available, this number will appear negative. |
| **COMMENT:** |
| This Host function calls up a corresponding AP function in the AP400, which determines the amount of available space in the AP for data buffers. |
## INPUT-OUTPUT OPERATIONS

### KHFAB

**TRANSFER DATA: HOST (FLTG.PT) TO AP (BFP)**

**KHFAB (HMA, DBI, VAL)**

**WHERE:**

- **HMA** = Host Memory address of first word of data set to be transferred to AP Data Memory.
- **DBI** = ID to assign to AP Data Buffer to be allocated. “DBI” must be a single-word integer variable or constant. If the DBF was previously allocated, it must be of size equal to specified size (VAL).
- **VAL** = Size of AP Data Buffer required, in AP Data Memory words. Equal to the number of floating point values to be transferred. “VAL” must be a single-word integer variable or constant. The DBF’s block exponent/NSN word is not included in this count.

**COMMENT:**

This Host Function calls up a corresponding AP function in the AP400 which in turn calls up the selected Data Transfer Routine. Data is transferred from HOST Memory in true floating point format to AP Data Memory in block floating point format.

### KABHI

**TRANSFER DATA: AP (BFP) TO HOST (2-COMP.INTGR.)**

**CALL KABHI (HMA, DBI, ISIZE, SCL)**

**WHERE:**

- **HMA** = The Host Memory address of the first word of the data set to receive data.
- **DBI** = The ID of the Data Buffer which contains the data to be transferred.
- **ISIZE** = The number of values to be transferred.
- **SCL** = A scaling factor; a power of 2 by which the data should be scaled before being transferred to the Host.

**COMMENT:**

The Host Function calls up the corresponding AP Function, which in turn calls up the selected data transfer routine. Data is scaled and converted into Host 2's-complement integer format, and is transferred into the Host.

### KABAX

**TRANSFER DATA: AP (BFP) TO AUX. I/O PORT**

**CALL KABAX (DBI, ISIZE, SCL)**

**WHERE:**

- **DBI** = The ID of the Data Buffer which contains the data to be transferred.
- **ISIZE** = The number of values to be transferred.
- **SCL** = A scaling factor; a power of 2 by which the data should be scaled before being transferred out of the AP’s Auxiliary Output Port.

**COMMENT:**

The Host Function calls up the corresponding AP Function, which in turn calls up the selected data transfer routine. Data is scaled as necessary, and is transferred through the Auxiliary Output Port.
# HOST FUNCTION CALLS

## LOGICAL DATA MANIPULATION

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KDBDB</strong> (DBla, DBlb)</td>
<td>ID of destination AP Data Buffer to “DBla” must be a single-word integer variable or constant. ID of source Data Buffer “DBlb” must be a single-word integer variable or constant. <strong>COMMENT:</strong> This Host function calls up a corresponding AP function in the AP400, which move the contents of one data buffer into another data buffer. The data buffer being moved to does not need to be allocated, but if it is it must be at least as large as the source data buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KBRVR</strong> (DBla, DBlb)</td>
<td>ID of AP destination data buffer “DBla” must be a single-word integer variable or constant. ID of source data buffer “DBlb” must be a single-word integer variable or constant. <strong>COMMENT:</strong> This Host Function calls up a corresponding AP function in the AP400, which will place the result of the real bit-reverse ordering of each element of one data buffer into another data buffer. The data buffer being moved to does not need to be allocated, but if it is it must be at least as large as the source data buffer. The destination data buffer may be the same as the source.</td>
</tr>
</tbody>
</table>
### STRAIGHTFORWARD COMPUTATION

**KMUL**

**MULTIPLY TWO REAL VECTORS:**

CALL KMUL (DBla, DBlb, DBlc)

**WHERE:**

- **DBla** = The ID of the destination Data Buffer.
- **DBlb** = The ID of source Data Buffer #1.
- **DBlc** = The ID of source Data Buffer #2.

**COMMENT:**

The Host Function calls the corresponding AP Function which calculates the point-by-point product of two real vectors and places the resulting vector in the destination Data Buffer.

---

**KMLSC**

**MULTIPLY A COMPLEX VECTOR BY A COMPLEX SCALER**

CALL KMLSC (DBla, DBlb, DBlc)

**WHERE:**

- **DBla** = The ID of the destination Data Buffer.
- **DBlb** = The ID of the source Data Buffer containing the complex scalar.
- **DBlc** = The ID of the source Data Buffer containing the complex scalar.

**COMMENT:**

The Host Function calls the corresponding AP Function, which calculates the product of a complex scalar and each point of a complex vector and places the resulting vector in the destination Data Buffer.

---

**KCONJ**

**COMPLEX CONJUGATE**

KCONJ (DBla, DBlb)

**WHERE:**

- **DBla** = ID of AP destination Data Buffer. “DBla” must be a single-word integer variable or constant.
- **DBlb** = ID of source Data Buffer. “DBlb” must be a single-word integer variable or constant.

**COMMENT:**

This Host Function calls up a corresponding AP function in the AP400, which will place the result of the complex conjugate of each element of one Data Buffer into another Data Buffer. The Data Buffer being moved to does not need to be allocated, but if it is it must be at least as large as the source Data Buffer. The destination Data Buffer may be the same as the source.
SOPHISTICATED COMPUTATION

KFFTR2

FORWARD FFT (INTERLACED REAL TO VARIANT-ORDER COMPLEXED)
CALL KFFTR2 (ISIZE, DBId, DBIs)

WHERE:

ISIZE = The size of the vector to be FFTed
DBId = The ID of the Data Buffer to receive the result
DBIs = The ID of the Data Buffer containing the source vector.

COMMENT:
The Host Function calls up a corresponding AP Function, which calls up the selected Fast Fourier Transform. This function transforms the data in one Data Buffer and places the result in another Data Buffer.

KTHPFC

THREE POINT CONVOLUTION (REAL-BY-COMPLEX)
KTHPFC (DBla, DBlb, DBlc)

WHERE:

DBla = ID of AP Data Buffer to hold result data, "A". "DBla" must be a single-word integer variable or constant. DBF need not have been previously allocated. If not already allocated, DBF will be allocated; size will equal that of source Data Buffer "B". If result DBF was previously allocated, it must be of size equal to source Data Buffer "B".

DBlb = ID of AP Data Buffer holding one source data set, "X". "DBlb" must be a single-word integer variable or constant. DBF must have been previously allocated in AP Data Memory.

DBlc = ID of AP Data Buffer holding scalar source data set "B". It should contain two values. "DBlc" must be a single-word integer variable or constant. DBF must have been previously allocated in AP Data Memory.

COMMENT:
This Host Function calls up a corresponding AP function in the AP400, which in turn calls up the selected "Three Point Convolution (Real-by-Complex)" function subroutine to perform the following operation:

\[ \text{Ar}(l) = B(1) \times \text{Xr}(l-1) + \text{Xr}(l) + B(2) \times \text{Xr}(l + 1), \]
\[ \text{Al}(l) = B(1) \times \text{Xl}(l-1) + \text{Xl}(l) + B(2) \times \text{Xl}(l + 1), \]

Where the subscripts refer to the real and imaginary parts of a complex number.

This Host Function version assumes that source data already resides in two AP Data Memory Data Buffers, and that the result data will be placed in another AP Data Memory Data buffer.
TRIGONOMETRIC SINE
KSIN (DBla, DBlb)

WHERE:
DBla = ID of AP destination Data Buffer. "DBla" must be a
single-word integer variable or constant.
DBlb = ID of source Data Buffer. "DBlb" must be a single-word
integer variable or constant.

COMMENT:

This Host Function calls up a corresponding AP Func-
tion in the AP400, which will place the trigonometric
sine of a Data Buffer into another Data Buffer.
The destination Data Buffer does not need to be
allocated, but if it is it must be at least as large as the
source Data Buffer.
The destination Data Buffer may be the same as the
source.
## 4.5 HOST FUNCTION LIBRARY

The following list identifies Host Functions currently being used in AP400 Array Processor applications.

<table>
<thead>
<tr>
<th>HOST FUNCTION LIBRARY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KABORT</strong> — ABORT THE CURRENTLY EXECUTING AP OPERATION</td>
</tr>
<tr>
<td><strong>KCTL</strong> — SET UP AP CONTROL WORD</td>
</tr>
<tr>
<td><strong>KDETCH</strong> — DETACH AP INTERRUPT VECTOR (UNDER RT-11 V3B)</td>
</tr>
<tr>
<td><strong>KEREX</strong> — SPECIFY FATAL ERROR SERVICE ROUTINE</td>
</tr>
<tr>
<td><strong>KEXFCB</strong> — EXECUTE FUNCTION CONTROL BLOCK</td>
</tr>
<tr>
<td><strong>KEXIT</strong> — EXIT PROGRAM THROUGH AP DRIVER</td>
</tr>
<tr>
<td><strong>KLOAD</strong> — LOAD A NAMED AP OBJECT MODULE</td>
</tr>
<tr>
<td><strong>KRESET</strong> — AP RESET (COMPLETE HARDWARE AND SOFTWARE)</td>
</tr>
<tr>
<td><strong>KRINIT</strong> — REINITIALIZE AP (SOFTWARE RESTART)</td>
</tr>
<tr>
<td><strong>KSETIW</strong> — RUN IN IMMEDIATE-RETURN VS. WHEN-AP-DONE MODE</td>
</tr>
<tr>
<td><strong>KSTAT</strong> — CHECK AP STATUS (FIP#, last status returned)</td>
</tr>
<tr>
<td><strong>KSYNC</strong> — SYNCHRONIZE FROM HOST TO AP (IMMEDIATE)</td>
</tr>
<tr>
<td><strong>KWAIT</strong> — WAIT FOR LAST FUNCTION CALL TO COMPLETE</td>
</tr>
<tr>
<td><strong>KWTFCB</strong> — WAIT FOR COMPLETION OF A SPECIFIC FCB</td>
</tr>
<tr>
<td><strong>KTHRTL</strong> — ADJUST AP INTERFACE BUS THROTTLE SETTING</td>
</tr>
<tr>
<td><strong>KALDB</strong> — ALLOCATE A DATA BUFFER</td>
</tr>
<tr>
<td><strong>KRLDB</strong> — RELEASE A DATA BUFFER</td>
</tr>
<tr>
<td><strong>KRBDS</strong> — RELEASE ALL DATA BUFFERS</td>
</tr>
<tr>
<td><strong>KDBSP</strong> — DETERMINE AVAILABLE DATA BUFFER SPACE</td>
</tr>
<tr>
<td><strong>KDBTS</strong> — SET DATA BUFFER ALLOCATION TABLE SIZE</td>
</tr>
<tr>
<td><strong>KHIAB</strong> — XFR DATA: HOST (2’S CMP INTEGER) TO AP (BFP)</td>
</tr>
<tr>
<td><strong>KHMAB</strong> — XFR DATA: HOST (16-BIT MAGNITUDE) TO AP (BFP)</td>
</tr>
<tr>
<td><strong>KABHI</strong> — XFR DATA: AP BFP TO HOST (2’S CMP INT) SCALED</td>
</tr>
<tr>
<td><strong>KABHB</strong> — XFR DATA: AP BFP TO HOST (1-WORD BFP)</td>
</tr>
<tr>
<td><strong>KHFAB</strong> — XFR DATA: HOST (PDP-11 FLTG PT) TO AP (BFP)</td>
</tr>
<tr>
<td><strong>KABHF</strong> — XFR DATA: AP (BFP) TO HOST (PDP-11 FLTG PT)</td>
</tr>
<tr>
<td><strong>KZRDB</strong> — CLEAR A DATA BUFFER</td>
</tr>
<tr>
<td><strong>KDBBD</strong> — MOVE ONE DATA BUFFER TO ANOTHER DATA BUFFER</td>
</tr>
<tr>
<td><strong>KNRDB</strong> — NORMALIZE A DATA BUFFER</td>
</tr>
<tr>
<td><strong>KBRVR</strong> — REORDER DATA IN BIT-REVERSED SEQUENCE (REAL)</td>
</tr>
<tr>
<td><strong>KBRVC</strong> — REORDER DATA IN BIT-REVERSED SEQUENCE (CPLX)</td>
</tr>
<tr>
<td><strong>KFVSC</strong> — REORDER DATA FROM FFT VARIANT TO SEQUENT (CPLX)</td>
</tr>
<tr>
<td><strong>KSFVC</strong> — REORDER DATA FROM SEQUENT TO FFT VARIANT (CPLX)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Host Function Library (Continued)</td>
</tr>
<tr>
<td>----------------------------------</td>
</tr>
<tr>
<td>KLPE2</td>
</tr>
<tr>
<td>KEXPE</td>
</tr>
<tr>
<td>KEXP D</td>
</tr>
<tr>
<td>KPWLA1</td>
</tr>
<tr>
<td>KPWLA2</td>
</tr>
<tr>
<td>KPWLA3</td>
</tr>
<tr>
<td>KCONV</td>
</tr>
</tbody>
</table>

The Library of Host Functions is continually being expanded to include additional functions for real-time signal processing applications, such as: image processing, seismic data processing, and vibration analysis, where array processors are vital to achieve the increased processing speed. Examples of these functions currently under development include:

Data Multiplex/De-Multiplex...
Separate one data set out of another. Useful for selectively processing real/imaginary points from a complex data set, or for retrieving acquired data from a set of multiplexed readings.

Merge two data sets. Useful for combining, for example, real and imaginary data points into a complex data set.

Matrix Operation
Transposition of a square or rectangular matrix.
Invert a matrix.

Data Set Minimum/Maximum Operation...
Find minimum-valued point in a data set.
Find maximum-valued point in a data set.
Threshold all points of a data set.

Data Companding...
Compaction of a data set. Especially useful in image storage and retrieval, where point-to-point value differences are small, and the volume of data is large.
Expansion of a compacted data set.

Statistical Measures...
Mean and Root-Mean-Squared-Deviation (RMSD) for a data set.
Histogram of a data set (frequency of occurrence of values, presented spectrally).

Trigonometric Operations...
Arctangent.
Coordinate conversion.

FFT Variations...
Forward, Inverse FFT's requiring fewer coefficient table entries.
Fully AP-resident 2-dimensional FFTs.

I/O Operations...
General Host control of auxiliary input/output.
Additional special-purpose and general-purpose I/O routines.
5.1 INTRODUCTION

The AP400 has been designed with a straightforward, operation/operands type of Assembly Language. Its style, usage, and even many instructions are familiar to anyone experienced in Assembly Language programming for most common minicomputer systems, such as the Digital Equipment Corporation PDP-11 series or the Data General NOVA or ECLIPSE series.

The AP400 Assembly Language evolves directly from the AP400 Machine Language, a vertically-architected, powerful mechanism for control of AP400 operation via programmed, sequential-instruction execution within the AP400 Control Processor.

AP400 Assembly and Machine Language instructions are invisible to the user who is programming in Host FORTRAN or Host Assembly Language. For the user who chooses to program in AP400 Assembly Language, though, this section presents an insight into processor operation and the versatility and power of the AP400 instruction set.

5.2 INSTRUCTION EXECUTION TIME

The Control Processor executes instructions sequentially, one-at-a-time, synchronized with the AP400’s 160-nanosecond Master Clock cycle. Execution time is a multiple of 160 nanoseconds, with the majority of the instructions executing in a single 160-nanosecond cycle. In general, instructions that require more than one cycle in order to complete execution are those involving references to AP Data Memory or registers that are not a part of the 16-register CP General Register set. Branch- and Skip-type instructions may take more than 1 cycle in certain cases, but frequently this time may be restricted to 1 cycle by the use of the “deferred execution” form of the branch-type instruction.

5.3 PROGRAM MEMORY

The AP400 Program Memory consists of 2048 locations of 22 bits each. Although Program Memory addresses are always expressed as absolute quantities in AP400 hardware, AP400 Assembly Language fully supports relative addressing, such that AP400 programs are normally written in fully relocatable, linkable code.

The AP400 Program Memory Address Register (also called the Program Counter or PC) is normally incremented by +1 upon execution of each instruction during AP operation. However, a Branch- or Skip-type instruction will directly load the PMAR with a new value when a branch is required.

AP400 Program Memory may not be modified by a running program. (AP programs utilize whatever space is required for constants, work areas, and the like, in AP Data Memory). This simple design characteristic is a highly effective mechanism to minimize debug time and to enhance program reliability.

5.4 ASSEMBLY LANGUAGE INSTRUCTION LISTING

In the pages that follow, each AP Assembly Language instruction is briefly described. Its Assembly Language mnemonic, an example of its typical use, and its hexadecimal Machine Language Instruction are also presented. AP Assembly Language Assembler Directives, that control parameters of program assembly, are also presented.

5.5 AP ASSEMBLY LANGUAGE PROGRAM EXAMPLE

Figure 5-1 provides a listing of the AP Assembly Language routine to change the sign of all data points in a vector. This routine demonstrates the use of the pipeline by the PIPE instruction. Reference is also made, in this example, to the use of service subroutines that provide a high degree of flexibility when using AP Assembly Language to encode new functions.
ASSEMBLER DIRECTIVES

RELATIVE PROGRAM MEMORY ORIGIN PMORG
The Relative Program Memory Origin Directive sets the relative address at which the assembler will assemble the following code in Program Memory.
EXAMPLE: PMORG 0 No code generated.

RELATIVE DATA MEMORY ORIGIN DMORG
The Relative Data Memory Origin Directive sets the relative address at which the assembler will assemble the following code in Data Memory.
EXAMPLE: DMORG 0 No code generated.

ABSOLUTE PROGRAM MEMORY ORIGIN PMORGA
The Absolute Program Memory Origin Directive sets the absolute address at which the assembler will assemble the following code in Program Memory.
EXAMPLE: PMORGA 10 No code generated.

ABSOLUTE DATA MEMORY ORIGIN DMORGA
The Absolute Data Memory Origin Directive sets the absolute address at which the assembler will assemble the following code in data Memory.
EXAMPLE: DMORGA 40 No code generated.

ASSEMBLY LISTING CONTROL PRINT
The Assembly Listing Control Directive allows the user to list or not list portions of his source code.
EXAMPLE: PRINT OFF No code generated.

NEW PAGE PAGE
The New Page Directive instructs the assembler to start a new printed page on the listing.
EXAMPLE: PAGE No code generated.

PAGE TITLE TITLE
The Page Title Directive gives the assembler a title to print at the top of each page of the listing.
EXAMPLE: TITLE SUBTRACT SUBROUTINE No code generated.

CONDITIONAL ASSEMBLY ASSEM
the Conditional Assembly Directive instructs the assembler to assemble or not assemble portions of the user source code.
EXAMPLE: ASSEM OFF No code generated.

INTERNAL GLOBAL DEFINITION IGLOBL
The Internal Global Definition Directive informs the assembler that the given list of symbols are defined in this module, and may be referenced by other modules.
EXAMPLE: IGLOBL ENTRY TABLE 1 No code generated.

EXTERNAL GLOBAL REFERENCE EGLOBL
The External Global Reference Directive informs the assembler that the given list of symbols are not defined in this module, but are defined in another module, and will be defined at link time.
EXAMPLE: EGLOBL ADD DMFREE No code generated.
DEFAULT RADIX
The Default Radix Directive informs the assembler what radix is to be assumed when no explicit radix specification is given in a numeric quantity.

EXAMPLE: RADIX H No code generated.

MODULE NAME
The Module Name Directive tells the assembler what name and version to give to the object module generated by the assembler.

EXAMPLE: NAME FFT, 001 No code generated.

FUNCTION ENTRY POINT
The Function Entry Point Directive informs the assembler of a function number and its corresponding entry point in this module.

EXAMPLE: FUNC 100, QADD No code generated.

SYMBOL DEFINITION
The Symbol Definition Directive assigns a value to a given symbol.

EXAMPLE: TABEND: EQU TABLE + 10 No code generated.

REPEAT CODE
The Repeat Code Directive directs the assembler to assemble the following code as many times as specified.

EXAMPLE: REPEAT 4 No code generated.

ASSEMBLY END
The Assembly End Directive informs the assembler that the end of the source code has been reached.

EXAMPLE: END No code generated.
INSTRUCTIONS

NO OPERATION
The No Operation Instruction performs no function in the control processor, acting only as a placeholder for one word in program memory.
EXAMPLE: NOP MACHINE INSTRUCTION: 000000

PIPELINE SETUP
The Pipeline Setup Directive informs the assembler of the PAC and other parameters to be used in the pipeline instruction sequence to be generated. Used in conjunction with PAD.
EXAMPLE: PIPE PREGMV, SCL0, LZCOFF No code generated

PIPEDLINE ADDRESS
The Pipeline Address instruction sets up a pipeline instruction sequence, specifying the addresses of the source and result data.
EXAMPLE: PAD R2 = R2 + R5, D2R1 MACHINE INSTRUCTION: 0A5002

SET REGISTER TO VALUE
The Set Register to a Value Instruction stores the given value in the specified register.
EXAMPLE: SETR R3 = 200 MACHINE INSTRUCTION: 102003

SET REGISTER TO REGISTER EXPRESSION
The Set Register to Register Expression Instruction computes the value of the specified expression and stores the computed value in the specified register.
FORMS:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>'COMPRs</td>
</tr>
<tr>
<td>−Rs</td>
<td>Rs'AND'Rd or Rd'AND'Rs</td>
</tr>
<tr>
<td>Rs + 1</td>
<td>Rs'OR'Rd or Rd'OR'Rs</td>
</tr>
<tr>
<td>Rs − 1</td>
<td>Rs'XOR'Rd or Rd'XOR'Rs</td>
</tr>
<tr>
<td>Rd + Rs or Rs + Rd</td>
<td>Rs'BIC'Rd</td>
</tr>
<tr>
<td>Rd − Rs</td>
<td>Rs'XNOR'RD</td>
</tr>
<tr>
<td>Rs − Rd</td>
<td></td>
</tr>
<tr>
<td>Rs + Rd + 1 or Rs + 1 + Rd</td>
<td>For the SET in-</td>
</tr>
<tr>
<td>Rd + RS + 1 or Rd + 1 + Rs</td>
<td>struction</td>
</tr>
<tr>
<td>Rd − Rs − 1 or Rd − 1 − Rs</td>
<td>only, these</td>
</tr>
<tr>
<td>Rs − Rd − 1 or Rs − 1 − Rd</td>
<td>forms may</td>
</tr>
<tr>
<td>Rs + quan</td>
<td>Rs'AND' quan</td>
</tr>
<tr>
<td>Rs − quan</td>
<td>Rs'OR' quan</td>
</tr>
<tr>
<td>Rs + 1 + quan</td>
<td>Rs'XOR'quand</td>
</tr>
<tr>
<td>Rs − 1 − quan</td>
<td></td>
</tr>
<tr>
<td>−Rs + quan</td>
<td></td>
</tr>
</tbody>
</table>

EXAMPLES: SET R2 = Rs − R5 MACHINE INSTRUCTION: 055102
SET R4 = R3'AND' % B11 MACHINE INSTRUCTION: 223034
**SKIP ON GREATER THAN OR EQUAL TO ZERO**

**SKIPGE**

The Skip on Greater Than or Equal to Zero Instruction computes the value of the expression, optionally stores the result in a register, and skips the next instruction if the computed value was greater than or equal to zero.

**FORMS:** All the Arithmetic Expression Forms specified for the SET instruction are supported.

**EXAMPLES:**  
- `SKIPGE R1 = R1 - 60`  
- `SKIPGE R1 = R1 - R2`

**MACHINE INSTRUCTION:**  
- `2DC601`
- `052701`

---

**SKIP ON LESS THAN ZERO**

**SKIPLT**

The Skip on Less Than Zero Instruction computes the value of the expression, optionally stores the result in a register, and skips the next instruction if the result was less than zero.

**FORMS:** All the Arithmetic Expression Forms specified for the SET instruction are supported.

**EXAMPLE:** `SKIPLT R3 = R3`

**MACHINE INSTRUCTION:**  
- `073E03`

---

**SKIP IF EQUAL TO ZERO**

**SKIPEQ**

The Skip if Equal to Zero Instruction computes the value of the expression, optionally stores the result in a register, and skips the next instruction if the computed value was equal to zero.

**FORMS:** All the logical expression forms specified for the SET Instruction are supported.

**EXAMPLE:** `SKIPEQ R3 = R3' OR' R5`

**MACHINE INSTRUCTION:**  
- `055F03`

---

**SKIP IF NOT EQUAL TO ZERO**

**SKIPN**

The Skip if Not Equal to Zero Instruction computes the value of the expression, optionally stores the result in a register, and skips the next instruction if the computed value was not equal to zero.

**FORMS:** All the logical forms specified for the SET Instruction are supported.

**EXAMPLE:**  
- `SKIPEQ R2' OR' R3`

**MACHINE INSTRUCTION:**  
- `053C02`

---

**ACCESS TO INTERNAL AP REGISTERS**

**MOVE**

The MOVE Instruction gives the user access to many of the AP's internal registers, other than the 16 CP general registers.

**EXAMPLE:** `MOVE REGHMA, R3`

**MACHINE INSTRUCTION:**  
- `300003`

---

**JUMP TO LOCATION**

**JMP**

The Jump to Location Instruction branches to the specified location.

**EXAMPLE:**  
- `JMP DONE`

**MACHINE INSTRUCTION:**  
- `350000`

---

**JUMP TO LOCATION DEFERRED**

**JMPD**

The Jump to Location Deferred Instruction branches to the specified location after executing the instruction following the JMP instruction. The JMPD instruction may also jump to an address specified in a register.

**EXAMPLE:**  
- `JMPD R1`
- `JMPD LOOP`

**MACHINE INSTRUCTION:**  
- `3F0001`
- `370000`

---

**TEST FLAG AND JUMP**

**TFJ**

The Test Flag and Jump Instruction compares the state of the given flag with the desired state of the flag, and if the states are the same, then the jump is taken.

**EXAMPLE:**  
- `TFJ F1 = 0, F1S0`

**MACHINE INSTRUCTION:**  
- `350001`
TEST FLAG AND JUMP DEFERRED  TFJD
The Test Flag and Jump Deferred Instruction is the same as the TFJ instruction except that the instruction following the TFJD instruction is always executed whether or not the branch is taken.
EXAMPLE: TFJD F6 = 1, RDY  MACHINE INSTRUCTION:  35000C

DECREMENT REGISTER AND BRANCH IF NON-ZERO  DBNZ
The Decrement Register and Branch if Non-Zero Instruction decrements the specified register and stores the result back in the register. If the result was not zero then a branch to the specified location occurs.
EXAMPLE: DBNZ R1, LOOP  MACHINE INSTRUCTION:  310001

DECREMENT REGISTER AND BRANCH IF NON-ZERO, DEFERRED  DBNZD
The Decrement Register and Branch if Non-zero Deferred Instruction is the same as the DBNZ instruction, except that the instruction following the DBNZD is executed whether or not the branch is taken.
EXAMPLE: DBNZD R3, LOOP1  MACHINE INSTRUCTION:  330003

JUMP TO SUBROUTINE  JSR
The Jump to Subroutine Instruction pushes the address of the next instruction onto the stack, and branches to the specified address.
EXAMPLE:  JSR FLHWT  MACHINE INSTRUCTION:  3C0000

RETURN FROM SUBROUTINE  RTN
The Return from Subroutine Instruction pops the return address off the stack and branches to that address.
EXAMPLE:  RTN  MACHINE INSTRUCTION:  3B0000

RETURN FROM SUBROUTINE AND SKIP  RTNS
The Return from Subroutine and Skip Instruction pops the return address off the stack, adds 1 to it, and then branches to the computed address.
EXAMPLE:  RTNS  MACHINE INSTRUCTION:  3B0800

RETURN FROM INTERRUPT  RTNI
The Return From Interrupt Instruction returns control to the code that was executing prior to the last interrupt.
EXAMPLE:  RTNI  MACHINE INSTRUCTION:  390000

INTERRUPT MASK LOAD  LDMSK
The Interrupt Mask Load Instruction places the given value, or register contents, in the interrupt mask. This enables or disables selected interrupts.
EXAMPLE:  LDMSK R1  MACHINE INSTRUCTION:  3E0801

INTERRUPT MASK STORE  STMSK
The Interrupt Mask Store Instruction places the current value of the Interrupt Mask into the specified register.
EXAMPLE:  STMSK R1  MACHINE INSTRUCTION:  3E0001
 INTERRUPT ENABLE AND DISABLE
INTR
The Interrupt Enable/Disable Instruction will turn all interrupts off or on, or clear any specified pending interrupts.
EXAMPLE: INTR ON MACHINE INSTRUCTIONS: 380800
INTR CLR 390800

LOAD REGISTER FROM DATA MEMORY
LDREG
The Load Register From Data Memory Instruction allows the user to load the contents of the high or low portion of a Data Memory word into a specified register.
EXAMPLE: LDREG R8,R2,LO MACHINE INSTRUCTION: 302988

LOAD REGISTER FROM DATA MEMORY AND INCREMENT ADDRESS REGISTER
LDREGI
The Load Register From Data Memory and Increment Address Register Instruction is the same as the LDREG instruction except that the address register is incremented after use.
EXAMPLE: LDREGI R6,R5,LO MACHINE INSTRUCTION: 345986

DECREMENT ADDRESS REGISTER AND LOAD REGISTER FROM DATA MEMORY
LDREGD
The Decrement Address Register and Load Register from Data Memory Instruction is the same as the LDREG instruction, except that the address register is decremented before use.
EXAMPLE: LDREGD R6,R5 MACHINE INSTRUCTION: 325BF6

STORE REGISTER IN DATA MEMORY
STREG
The Store Register in Data Memory Instruction allows the user to store a value from a given register in Data memory at the address specified in another register.
EXAMPLE: STREG R6,R5 MACHINE INSTRUCTION: 305386

STORE REGISTER IN DATA MEMORY AND INCREMENT ADDRESS REGISTER
STREGI
The Store Register in Data Memory and Increment Address Register Instruction is the same as the STREG instruction, except that the address register specified is incremented after being used as an address.
EXAMPLE: STREGI R6,R5 MACHINE INSTRUCTION: 345386

DECREMENT ADDRESS REGISTER AND STORE REGISTER IN DATA MEMORY
STREGD
The Decrement Address Register and Store Register in Data Memory Instruction is the same as the STREG instruction, except that the address register specified is decremented before use.
EXAMPLE: STREGD R6,R5 MACHINE INSTRUCTION: 325386
PUSH TO STACK
The Push to Stack Instruction pushes the value in the specified register onto the
stack.
EXAMPLE: PUSH R6 MACHINE INSTRUCTION: 340386

POP FROM STACK
The Pop from Stack Instruction removes the current value on the stack and places it
in the specified register.
EXAMPLE: POP R6 MACHINE INSTRUCTION: 320BF6

SKIP IF CONDITION IS TRUE
The Skip if Condition is True Instruction skips the next instruction if the given ex-
pression is true.
FORMS: Rd'GT'exp
Rd'GE'exp
Rd'EQ'exp
Rd'NE'exp
Rd'LT'exp
Rd'LE'exp

EXAMPLES: SKIP R2'NE'4E MACHINE INSTRUCTION: 2B04E2
SKIP R3'GE'R4 MACHINE INSTRUCTION: 054503
DATA STORAGE INSTRUCTIONS

DEFINE STORAGE  
The Define Storage Instruction allows the user to place an arbitrary value in Program or Data Memory.  
EXAMPLE:  DS  123456  DATA MEMORY CONTENTS  123456

DEFINE BYTE  
The Define Byte Instruction allows the user to place 3 bytes in one word of Data Memory.  
EXAMPLE:  DB  1,2,3  DATA MEMORY CONTENTS  010203

DEFINE PARTIAL WORD  
The Define Partial Word Instruction allows the user to specify the high 16 bits, and low 8 bits of a word in Data Memory.  
EXAMPLE:  DP  456,1  DATA MEMORY CONTENTS  045601

DEFINE WORD  
The Define Word Instruction allows the user to store a value in Data Memory. The word may be expressed as a decimal fraction.  
EXAMPLES:  DW  %F.5  DATA MEMORY CONTENTS  400000  DW  -%F.125  F00000
DATE 22-AUG-79 09:23:30   ASM V02.1P

001 TITL E  FNC SUB:  NEGATE
002 NAME   NEC1, 001           ;NAME AND VERSION FOR THE OBJECT MODULE.
003 RADI X  H             ;SPECIFY HEXADECIMAL RADIX FOR ASSEMBLY LISTING.
004  ;INTERNALLY DEFINED GLOBALIZED SYMBOLS:     (IGLOBL)
005  IGLOBL  NEG          ;SUBROUTINE ENTRY POINT.
006  ;EXTERNALLY DEFINED GLOBALIZED SYMBOLS:     (EGLOBL)
007  EGLOBL  EXIT1, SETSCL   ;SUBROUTINE ENTRY POINTS.
008  
009 ;PIPELINE PAC SYMBOL DEFINITIONS:
010  
011 00000054 PCHSMV:  EQU  %H54    ;CHANGE SIGN AND MOVE PAC.
012 00000000 PMORC  0            ;START OF RELOCATABLE PROGRAM MEMORY CODE.
013  
014 NEG:    ;ENTRY POINT FOR CHANGE SIGN ROUTINE.
015 
016 SET R4= R4+1/2          ;CALCULATE THE SMALLEST MULTIPLE OF FOUR GREATER
017 P000000 00059204        ; THAN OR EQUAL TO THE GIVEN LENGTH IN ORDER TO
018 P000001 00064204        ; PERFORM FOUR RECTIONS PER PIPELINE COMMAND:
019 P000002 00307981        ;CALCULATE (L+1)/2.  (REGISTER R4 WILL BE USED
020 P000003 0030683C        ;CALCULATE (L+3)/4.  AS THE PIPELINE ITERATION
021 P000004 003079BF        ;IS COUNTER.)
022 P000005 00308381        ;STORE IT IN THE RESULT VECTOR.
023 P000006 00100011        ;STORE IT IN THE RESULT VECTOR.
024 P000007 003C0000        ;SET THE NUMBER OF GUARD BITS REQUIRED FOR THIS
025 P000008 00100022        ;OPERATION BEFORE CALLING "SETSCL".
026 P000009 00057101        ;CALL SERVICE SUBROUTINE IN ORDER TO ADJUST THE
027 P000010 00038103        ;SOURCE VECTORS OF THE RESULT DATA BUFFER AND PLACE
028 P00000B 00A2541        ;THE REQUIRED SETTING IN THE PIPELINE SCALING
029 P000011 00A2B91 0002F83 000A2403
030 SET R2= 2             ;SET THE PIPELINE ADDRESSING INCREMENT.
031 P00000C 00037101        ;MODIFY THE SOURCE AND RESULT DATA ADDRESSES SO
032 P00000D 00038103        ;THE FIRST ADDRESSING INCREMENTS WILL CAUSE
033 P00000E 00057101        ;THE ADDRESSES TO POINT TO THE FIRST POINTS IN
034 P00000F 00057101        ;THE SOURCE AND RESULT AREAS.  (-1 IS USED
035 P00000G 00057101        ;SINCE THE ADDRESSES POINT TO THE BX/NSN
036 P00000H 0007101         ;WORDS OF THE AREAS, INITIALLY.)
037 NEGL:  PIPE  PCHSMV, SCLREG, LZC12 ;USE THE CHANGE SIGN-AND-MOVE PAC
038 PAD R1= R1+R2, S1   ;SOURCE VECTORS. INCREMENT BY TWO.
039 PAD R1= R1+R2, S2   ;SOURCE VECTORS. INCREMENT BY TWO.
040 PAD R3= R3+R2, D1R1 ;RESULT VECTORS. INCREMENT BY TWO.
041 PAD R3= R3+R2, D2R1 ;RESULT VECTORS. INCREMENT BY TWO.
042 P00000B 00A2541        ;DECREMENT REGISTER AND BRANCH BACK IF NOT DONE.
043 P000012 003100B4        ;JUMP TO SERVICE SUBROUTINE WHICH WAITS FOR THE
044 P000013 00330000        ;PIPELINE TO FINISH, READS THE PIPELINE
045 P000014 003100B4        ;LEADING-ZERO COUNT REGISTER, AND ADJUSTS THE
046 P000015 003100B4        ;RESULT DATA BUFFER'S NSN. ITS 'RTM' WILL
047 P000016 003100B4        ;CAUSE A RETURN TO THE CALLING ROUTINE.
048 P000017 003100B4
049 P000018 003100B4
050 P000019 003100B4
051 P00001A 003100B4
052 P00001B 003100B4
053 P00001C 003100B4
054 P00001D 003100B4
055 P00001E 003100B4
056 P00001F 003100B4
057 P000020 003100B4
058 P000021 003100B4
059 P000022 003100B4
060 P000023 003100B4
061 P000024 003100B4
062 P000025 003100B4
063 P000026 003100B4
064 END

Figure 5.1. Sample AP Assembly Language Program Listing, Negating Data Points in a Vector

COPYRIGHT 1979 - PRINTED IN U.S.A. 5-10
6.1 INTRODUCTION

This chapter lists the PAC (Pipeline Arithmetic Commands) operations that are pre-programmed and reside in the PROMs. The PAC operations are implemented as the PIPE instruction (up to 256 different variations), used when programming in AP Assembly Language. Each PAC is identified by an ID NUMBER (hexadecimal format) that is unique for that PAC and a mnemonic name that is used when calling that PAC as a PIPE instruction. Refer to the AP400 Assembler Reference Manual for details of using a PIPE instruction.

6.2 LISTING FORMAT

The list of PACs is prepared in PAC ID Number (hexadecimal format) sequence, and for each PAC the table includes a description of the function performed, Assembly Language Mnemonic, and the explicit formula relationship between output number pairs (D1 and D2) and input number pairs (S1, S2, S3, and S4).

Note the coding used to express the input and output values; inputs are Xi, outputs are Oi.

Some PACs are “dual”, in that they provide for the processing of data in a parallel mode. That is, the basic arithmetic operation performed by the PAC requires only two number pair inputs for a number pair output, so that the AP400 pipeline with 4 number pair inputs and 2 number pair outputs is configured as a parallel processing operation for this function.

Some arithmetic operations require the interleaving of two or more PACs. In this case, the processing accomplishes one pass with the first, or A PAC, and then the results of this pass become the inputs to the pipeline in a second, or B PAC, etc., before a computed output appears in the designated address locations (e.g., PACs #0F, &10).

The formula description may include an explicit reference to the use of a table. In some cases this table is to be supplied by the programmer (user) for each application. In other PACs, the table is defined at the factory.

6.3 USER PROGRAMMING

User programming of the AP400 generally consists of programming the Control Processor but can also include programming the pipeline. For the most part, these two tasks are separate. Control Processor, which is programmed in Assembly Language, processes the program, provides the pipeline with instructions, and manages data flow and storage throughout the Array Processor. The Pipeline takes in eight pieces of data and an instruction and puts out four pieces of data each 1.92 microseconds. Depending upon the instruction, the pipeline can perform any one of 256 operations. The programming of the instructions for the pipeline results in the programming of a PROM set that is used to decode the instruction. It is the decoded output from the PROM that configures the Pipeline’s internal switching and logic matrix.

Although the AP400 is delivered to a user with a PROM instruction set as described in this Section, the user can, if required, develop a new or additional PROM instruction set. This may be desired if the user has specific algorithms that may be more efficiently implemented with a PROM instruction set for a specific application. An optional PAC Developmental Package is available for users who may wish to perform Pipeline programming.
<table>
<thead>
<tr>
<th>PAC No. (HEX)</th>
<th>NAME</th>
<th>MNEMONIC</th>
<th>ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Regular Move</td>
<td>PREGMV</td>
<td>O1 = X1 + X2, O2 = X3, O4 = X4</td>
</tr>
<tr>
<td>02</td>
<td>Pairs-Swap Move</td>
<td>PRSWAP</td>
<td>O1 = X1, O2 = X2, O3 = X3, O4 = X4</td>
</tr>
<tr>
<td>03</td>
<td>Four Subtractions</td>
<td>PSUBT</td>
<td>O1 = X1-X2, O2 = X3, O4 = X5-X6, O4 = X7-X8</td>
</tr>
<tr>
<td>04</td>
<td>Four Additions</td>
<td>PSUM</td>
<td>O1 = X1 + X2</td>
</tr>
<tr>
<td>05</td>
<td>Four Multiplications</td>
<td>PMULT</td>
<td>O1 = X1*X2 (i + 4)</td>
</tr>
<tr>
<td>06</td>
<td>Complex Multiplication</td>
<td>PCPXML</td>
<td>O1 = X1<em>X2</em>X3, O1 = X2*X4, O4 = X5-X6, O4 = X7-X8</td>
</tr>
<tr>
<td>07</td>
<td>Radix 2 FFT Butterfly</td>
<td>PR2FLY</td>
<td>O1 = S1, O2 = S2, O3 = X3 + (X5<em>X7</em>X9*X8)</td>
</tr>
<tr>
<td>08</td>
<td>Sum of 4 Multiplications</td>
<td>PMLAD2</td>
<td>O3 = O4 = (Xi*Xi+i) for i = 1,2,3,4</td>
</tr>
<tr>
<td>09</td>
<td>Multiply-Add</td>
<td>PMLAD1</td>
<td>O3 = X1<em>X5 + X3</em>X8, O4 = X2<em>X7 + X4</em>X8</td>
</tr>
<tr>
<td>0A</td>
<td>Normalize Floating Pt.</td>
<td>PNORM1</td>
<td>D1R(D2R) = F1, D1I(D2I) = E1, where F1 and E1 are normalized mantissa and exponent of input S1R and S1I</td>
</tr>
<tr>
<td>0B</td>
<td>Absolute Value of Real No.</td>
<td>PABSBR</td>
<td>O1 = [Xi], for i = 1,2,3,4</td>
</tr>
<tr>
<td>0C</td>
<td>Index Set Generator, Initial</td>
<td>PINGNA</td>
<td>Not Used</td>
</tr>
<tr>
<td>0D</td>
<td>Index Set Generator, Iterative</td>
<td>PINGIB</td>
<td>Generate an index for an Input Data Set</td>
</tr>
<tr>
<td>0E</td>
<td>64-Segment Function</td>
<td>FNC64P</td>
<td>A 64-segment piecewise linear interpolator</td>
</tr>
<tr>
<td>0F</td>
<td>Division-Initial Step</td>
<td>PIVIDA</td>
<td>Not Used</td>
</tr>
<tr>
<td>10</td>
<td>Division-Iterative</td>
<td>PIVIDB</td>
<td>See PAC #47</td>
</tr>
<tr>
<td>11</td>
<td>Logarithm-Initial Step</td>
<td>LOGARA</td>
<td>See PAC #47</td>
</tr>
<tr>
<td>12</td>
<td>Logarithm-Iterative Oi</td>
<td>LOGARB</td>
<td>O1 = [sign Xi] (Xi)^i, O1 = 1 = [sign Xi] (Xi)^i, i = 3</td>
</tr>
<tr>
<td>13</td>
<td>Vector Signed-Squared</td>
<td>PVSGSQ</td>
<td>O1 = S1 + T3, O2 = S2 + T1, O3 = S3-S1, O4 = T1-S2</td>
</tr>
<tr>
<td>14</td>
<td>Radix-4 FFT, A</td>
<td>P4FFTA</td>
<td>S1 = X6<em>X7 + X8</em>X9, S2 = X5<em>X7 + X8</em>X9</td>
</tr>
<tr>
<td>15</td>
<td>Radix-4 FFT, B</td>
<td>P4FFTB</td>
<td>O1 = S3, O2 = S4, S1 = (X7<em>X6 + X8</em>X5)-S1, S2 = (X8<em>X6 + X7</em>X5) + S2, S3 = X3, S4 = X4</td>
</tr>
<tr>
<td>16</td>
<td>Radix-4 FFT, C</td>
<td>P4FFTC</td>
<td>T1 = S2 + (X7<em>X8-X7</em>X5), S1 = S3 + (X7<em>X6 + X8</em>X5) + S2 + T1 = X7<em>X6 + X8</em>X5 + S3, T3 = S3-(X7<em>X5-X8</em>X6)</td>
</tr>
<tr>
<td>17</td>
<td>Vector Dot Product</td>
<td>PVDPDR</td>
<td>LSB → D2R, MSB → D2I</td>
</tr>
<tr>
<td>18</td>
<td>Magnitude Squared of Complex No.</td>
<td>PMAGSQ</td>
<td>O3 = X5<em>X5 + X6</em>X6, O4 = X3<em>X7 + X4</em>X8</td>
</tr>
<tr>
<td>19</td>
<td>Double Length Sum</td>
<td>DLSMNX</td>
<td>O4 = Xi</td>
</tr>
<tr>
<td>1A</td>
<td>Sum of 8 Inputs</td>
<td>SUMOCT</td>
<td>O1 = Xi+1 for i = 1,2,3,5,7</td>
</tr>
<tr>
<td>1B</td>
<td>Four Adjacent Multiples</td>
<td>PADJML</td>
<td>Not Used</td>
</tr>
<tr>
<td>1C</td>
<td>M.T.I. Type Filter</td>
<td>PMTFL</td>
<td>O1 = max[X1, X5], O3 = min[X3, X7]</td>
</tr>
<tr>
<td>1D</td>
<td>Larger/Smaller Ordering, 1</td>
<td>PLGSM1</td>
<td>O1 = S1, O2 = max[X1, X5], O3 = min[X3, X7], O4 = min[X3, X7]</td>
</tr>
<tr>
<td>1E</td>
<td>Larger/Smaller Ordering, 2</td>
<td>PLGSM2</td>
<td>S1 = max[X3, X7]</td>
</tr>
<tr>
<td>1F</td>
<td>3rd Order Polynomial</td>
<td>POLY3R</td>
<td>Si = 0, Ti = 0, for i = 1 to 4</td>
</tr>
<tr>
<td>20</td>
<td>Clear All Accumulators</td>
<td>PCLRAC</td>
<td>If X5 &lt; 0, O1 = T1, O2 = T2</td>
</tr>
<tr>
<td>21</td>
<td>Load Accumulators</td>
<td>PLAC12</td>
<td>If X5 &lt; 0, X1 = T1, X2 = T2, O1 = X1, O2 = X2</td>
</tr>
<tr>
<td>22</td>
<td>Load Accumulators</td>
<td>PLAC34</td>
<td>If X5 &lt; 0, X1 = T1, X2 = T2, O1 = X1, O2 = X2</td>
</tr>
<tr>
<td>23</td>
<td>Read Accumulators</td>
<td>PRAC12</td>
<td>O1 = T1, O2 = T2, O3 = S1, O4 = S2</td>
</tr>
<tr>
<td>24</td>
<td>Read Accumulators</td>
<td>PRAC34</td>
<td>O1 = T3, O2 = T4, O3 = S3, O4 = S4</td>
</tr>
<tr>
<td>25</td>
<td>Left-Right Interpolation</td>
<td>PLFIN</td>
<td>If X1 &gt; 0, then O3 = O4 = (1-X1)<em>X4 + X1</em>X3</td>
</tr>
<tr>
<td>26</td>
<td>Upshift Multiplication</td>
<td>QDUPML</td>
<td>If X1 &lt; 0, then O3 = O4 = (1-X1)<em>X4 + X1</em>X5</td>
</tr>
<tr>
<td>27</td>
<td>Number of Shifts to Normalize</td>
<td>PNLZPN</td>
<td>See PAC No. 65</td>
</tr>
<tr>
<td>28</td>
<td>Block Floating Point to Floating Point Conversion</td>
<td>PNSCM2</td>
<td>DBSCV</td>
</tr>
</tbody>
</table>

**LEGEND**

Xi: Pipe input for i = 1, 2, ..., 8 [S1R, S1I, etc]  
Oi: Pipe output for i = 1,2,3,4 [D1R, D1I, etc]  
S1:Ti: Accumulators S and T for i = 1,2,3,4  
Ti(Xk): Table value with argument Xk
<table>
<thead>
<tr>
<th>PAC No. (HEX)</th>
<th>NAME</th>
<th>MNEMONIC</th>
<th>ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td>Double Length to Floating Pt. Conv.</td>
<td>DBFPCV</td>
<td>$\Sigma_{i=1}^{n} O_i = MSB, O_3 = LSB$</td>
</tr>
<tr>
<td>2B</td>
<td>Double Length Sum, 6 Single No.</td>
<td>PDLSCG</td>
<td>$S_1 = X_3 \cdot X_4, S_2 = (X_5 \cdot X_1 \cdot X_6 \cdot X_1) \cdot (X_7 \cdot X_1 \cdot X_8 \cdot X_1)$</td>
</tr>
<tr>
<td>2C</td>
<td>Real FFT, A</td>
<td>RLFFTA</td>
<td>$T_1 = X_4 + X_3, T_2 = (X_7 \cdot X_1 \cdot X_8 \cdot X_1) + (X_5 \cdot X_1 \cdot X_6 \cdot X_1)$</td>
</tr>
<tr>
<td>2D</td>
<td>Real FFT, B</td>
<td>RLFFTB</td>
<td>$O_1 = T_2 + S_2, O_2 = S_1 + (X_1 + X_2), O_3 = T_2 - S_2, O_4 = (X_1 \cdot X_2 + S_1, X_1 + X_2, S_2 = X_3 + X_4$</td>
</tr>
<tr>
<td>2E</td>
<td>Real FFT, C</td>
<td>RLFFTC</td>
<td>$O_1 = S_1 + T_1, O_3 = T_1 - S_1$</td>
</tr>
<tr>
<td>2F</td>
<td>Radix-4, Real Wts and Inverse FFT</td>
<td>RRAWIFA</td>
<td>$\rightarrow$ PAC No. 80</td>
</tr>
<tr>
<td>30</td>
<td>Radix-4, Real Wt, and FFT, B</td>
<td>R4WIFB</td>
<td>$\rightarrow$ PAC No. 4F</td>
</tr>
<tr>
<td>31</td>
<td>Radix-4, Real Wt, and FFT, C</td>
<td>RPR4FL</td>
<td>$O_1 = S_3 + T_1, O_2 = S_4 + T_2, O_3 = S_1 - T_4, O_4 = S_2 - S_3; S_1 = T_4 + S_1, S_2 = T_3 + S_2, S_3 = T_1 - S_3, S_4 = T_2 - S_4$</td>
</tr>
<tr>
<td>32</td>
<td>Inverse Tranform, Pass 1</td>
<td>PRLLFT</td>
<td>$O_3 = X_3 \cdot X_7 + X_2 \cdot X_8 + X_1 \cdot X_5, O_4 = X_4 \cdot X_7 + (X_1 \cdot X_5 \cdot X_2 \cdot X_6)$</td>
</tr>
<tr>
<td>33</td>
<td>Radix 2, Real FFT</td>
<td>PR2RML</td>
<td>$O_1 = X_3 \cdot X_7 \cdot X_4 \cdot X_6 + X_1 \cdot O_2 = X_1 \cdot (X_3 \cdot X_5 \cdot X_4 \cdot X_6)$</td>
</tr>
<tr>
<td>34</td>
<td>Radix-2, Complex FFT Butterfly</td>
<td>R2FTB</td>
<td>$O_3 = X_2 \cdot X_4 \cdot X_5 + X_3 \cdot X_6, O_4 = X_2 \cdot (X_4 \cdot X_5 + X_3 \cdot X_6)$</td>
</tr>
<tr>
<td>35</td>
<td>Load T Accumulators</td>
<td>PLDTAC</td>
<td>$O_1 = S_1 + X_1, O_2 = S_2 + X_2, S_3 = X_3, S_4 = X_4$</td>
</tr>
<tr>
<td>36</td>
<td>Load S Accumulators</td>
<td>PLDSAC</td>
<td>See PAC #5A</td>
</tr>
<tr>
<td>37</td>
<td>DEC to AP Floating Pt. Conv.</td>
<td>PDECSP</td>
<td>$O_i = X_i \cdot T(X_i)$(Table Lookup)</td>
</tr>
<tr>
<td>38</td>
<td>Real Modifies Imaginary</td>
<td>PRM18</td>
<td>$If X_2 \cdot X_4 \cdot X_6 \cdot X_8 = 0, O_2 = X_2 \cdot X_4 \cdot X_6$</td>
</tr>
<tr>
<td>39</td>
<td>Double Subtraction, Positive</td>
<td>PRDSSP</td>
<td>$If X_2 \cdot X_4 \cdot X_6 \cdot X_8 = 0, O_2 = X_4 \cdot X_5 \cdot X_3 \cdot X_5$</td>
</tr>
<tr>
<td>3A</td>
<td>Offset Vector Multiply</td>
<td>POFMUL</td>
<td>$O_1 = X_1 \cdot X_5 \cdot X_1, O_2 = X_3 \cdot T(X_3), O_3 = T(X_1), O_4 = T(X_3)$</td>
</tr>
<tr>
<td>3B</td>
<td>X Minus Table Value</td>
<td>PSMINT</td>
<td>$O_1 = X_1 \cdot X_5 \cdot X_1$ and $S_1 = O_2 = LSB \cdot (X_1 \cdot X_4)$ and $S_2 = LSB \cdot (X_2 \cdot X_5)$ and $S_3 = LSB \cdot (X_2 \cdot X_5)$ and $S_1 = O_4 = LSB \cdot (X_2 \cdot X_5)$ and $S_2,$</td>
</tr>
<tr>
<td>3C</td>
<td>Piecewise Linear Approx. 8 Bit</td>
<td>PWL8P8</td>
<td>$\rightarrow$ PAC No. 47</td>
</tr>
<tr>
<td>3D</td>
<td>Sum of Products</td>
<td>PWL8PD</td>
<td>$O_3 = X_1 \cdot X_5 + X_2 \cdot X_6, O_4 = X_3 \cdot X_7 + X_4 \cdot X_8$</td>
</tr>
<tr>
<td>3E</td>
<td>3-Point Digital Filter</td>
<td>PTHRPF</td>
<td>$O_1 = S_1 + X_1 \cdot X_4, O_2 = S_2 \cdot (X_1 \cdot X_2 \cdot X_4)$; $S_1 = X_2 \cdot X_1 \cdot X_3, S_2 = X_2 \cdot X_3$</td>
</tr>
<tr>
<td>3F</td>
<td>Or All Members of Vector</td>
<td>PORALL</td>
<td>$O_i = LSB \cdot [X_i \cdot X_{i+4}]$</td>
</tr>
<tr>
<td>40</td>
<td>Upshift Multiplication</td>
<td>PUPMCL</td>
<td>$O_1 = LSB \cdot (X_1 \cdot X_3)$ and $S_1 = O_2 = LSB \cdot (X_1 \cdot X_4)$ and $S_2,$</td>
</tr>
<tr>
<td>41</td>
<td>Upshift and Multiply</td>
<td>PUNPAK</td>
<td>$O_3 = LSB \cdot (X_2 \cdot X_3)$ and $S_1 = O_4 = LSB \cdot (X_2 \cdot X_4)$ and $S_2,$</td>
</tr>
<tr>
<td>42</td>
<td>Not Identified</td>
<td></td>
<td>Reserved for Release #2</td>
</tr>
<tr>
<td>43</td>
<td>Piecewise Linear Approx. 8 Bit</td>
<td>PWL8P6</td>
<td>$O_1 = X_1 \cdot T(X_1) + T(X_1), O_2 = X_3 \cdot T(X_3) + T(X_3)$</td>
</tr>
<tr>
<td>47</td>
<td>Piecewise Linear Approx. 8 Bit</td>
<td>PWL8P6</td>
<td>$O_3 = O_1 + X_2, O_4 = O_2 + X_4$</td>
</tr>
<tr>
<td>48</td>
<td>Piecewise Linear Approx. 6 Bit</td>
<td>PWL8P6</td>
<td>$O_1 = X_1 \cdot T(X_1) + T(X_1), O_2 = X_3 \cdot T(X_3) + T(X_3),$</td>
</tr>
<tr>
<td>49</td>
<td>Piecewise Linear Approx. 4 Bit</td>
<td>PWL8P6</td>
<td>$O_3 = T_1 \cdot T(X_1) + T(X_1), O_4 = T_2 \cdot T(X_3) + T(X_3),$</td>
</tr>
<tr>
<td>4A</td>
<td>APBFP to AP Floating Point</td>
<td>PNRMFP</td>
<td>$O_1 = LSB \cdot [X_1 \cdot T(X_1)], O_2 = LSB \cdot [X_3 \cdot T(X_3)], O_3 = T(X_1), O_4 = T(X_3)$</td>
</tr>
<tr>
<td>4B</td>
<td>Radix-2 Real FFT</td>
<td>PRFRFT</td>
<td>$O_i = 0$ if $X_i &lt; 0, O_i = 2 \cdot (X_i \cdot X_5)$ if $X_i \cdot X_5 &gt; 0$ for $i = 2, 4$</td>
</tr>
<tr>
<td>4C</td>
<td>Bound Below Zero</td>
<td>PTHRSX</td>
<td>$O_1 = S_1, O_2 = S_3, O_3 = X_1 \cdot X_5 + LSB \cdot (X_3 \cdot X_6),$</td>
</tr>
<tr>
<td>4D</td>
<td>Scale and Add</td>
<td>PADDSX</td>
<td>$O_4 = X_2 \cdot X_5 + LSB \cdot (X_4 \cdot X_6); S_1 = LSB \cdot (X_3 \cdot X_6); X_1 \cdot X_5,$</td>
</tr>
<tr>
<td>4E</td>
<td>Pair Swap and Scaler Mult</td>
<td>PSSMMLT</td>
<td>$S_2 = LSB \cdot (X_4 \cdot X_6 \cdot X_2 \cdot X_5),$</td>
</tr>
<tr>
<td>4F</td>
<td>Radix 4, Real Wt, &amp; FFT-1</td>
<td>PR4FB2</td>
<td>$O_1 = X_1 \cdot X_5 + X_2 \cdot X_5, O_3 = X_2 \cdot X_5 + X_2 \cdot X_5$</td>
</tr>
</tbody>
</table>

LEGEND
Xi: Pipe Input for $i = 1, 2, \ldots, 8$ [S1R, S1I, etc.]
Oj: Pipe Output for $i = 1, 2, 3, 4$ [D1R, D1I, etc.]
Si,Ti: Accumulators S and T for $i = 1, 2, 3, 4$
T(Xk): Table Value with argument Xk
### PIPELINE ARITHMETIC COMMANDS PACs
(Continued)

<table>
<thead>
<tr>
<th>PAC No. (HEX)</th>
<th>NAME</th>
<th>MNEMONIC</th>
<th>ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Radix 4, Real Wt &amp; FFT</td>
<td>PR4IFA</td>
<td>( O_1 = S_1, O_2 = S_2 ); ( T = X^4*X_4 + X^6*X_7, S_2 = X^6*X_7*X^8*X_4 )</td>
</tr>
<tr>
<td>51</td>
<td>Scaler Multiplication</td>
<td>PSR5RW</td>
<td>( O_i = \sum X_i \cdot X_5 ) for ( i = 1, 2, 3, 4 ) ( S_1 = S_3 )</td>
</tr>
<tr>
<td>52</td>
<td>Convolution, Initial</td>
<td>PCONVS</td>
<td>( O_3 = \sum X_i \cdot (X_i + 4), S_1 = O_3 ) ( S_1 = S_1 ) ( O_3 = S_1 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>53</td>
<td>Convolution, Iterative</td>
<td>PCONVT</td>
<td>( O_i = X_i ) for ( i = 1, 2, 3, 4 ) ( S_1 = S_1 ) ( O_3 = S_2 + S_1 )</td>
</tr>
<tr>
<td>54</td>
<td>Change Sign and Move</td>
<td>PCHSMV</td>
<td>( O_i = X_i ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>55</td>
<td>BFP to DEC Floating Pt</td>
<td>PBFPDC</td>
<td>Converts mantissa from AP to DEC format ( O_i = X_i \cdot X_5 ) for ( i = 1, 2, 3, 4 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>56</td>
<td>Magnitude Scalar Mult.</td>
<td>PMAGSC</td>
<td>( O_i = X_i \cdot X_5 ) for ( i = 1, 2, 3, 4 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>57</td>
<td>Piecewise Linear Approx with Mag.</td>
<td>PWLPMB</td>
<td>( O_i = X_i \cdot T(X_i) + T_6(X_i) ) ( O_3 = X^3*X_7(X_3) + T_8(X_3) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>58</td>
<td>Newton's Method of 1/X</td>
<td>PNTREC</td>
<td>( O_i = X_1 \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>59</td>
<td>Three Point Filter (Complex)</td>
<td>PTHPF1</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5A</td>
<td>DEC FP to AP Format</td>
<td>PDCFP1</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5B</td>
<td>Add &amp; Subtract</td>
<td>PADSBI</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5C</td>
<td>Separates Integer &amp; Fraction Parts</td>
<td>PINTFR</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5D</td>
<td>Leading-Zero-Dependent Shifts</td>
<td>PLZSHF</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5E</td>
<td>Address 1 Modifies Address 2</td>
<td>PA1M2</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>5F</td>
<td>Add &amp; Subtract Adjacent</td>
<td>PAD2AD</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>60</td>
<td>Radix-2 Real, FFT First Stage</td>
<td>PR2R1</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>61</td>
<td>Radix-2, Real FFT, Second Stage</td>
<td>PR2R2</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>62</td>
<td>Multiply &amp; Add a Constant</td>
<td>PMLAD3</td>
<td>( O_i = X_i \cdot X_j \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>63</td>
<td>Complex Conjugate</td>
<td>PSCPXC</td>
<td>( O_i = X_i \cdot X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>64</td>
<td>Piecewise Linear Approx, 8-Bit</td>
<td>PWL8MG</td>
<td>( O_i = X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
<tr>
<td>65</td>
<td>Double Length to Single Length</td>
<td>PDLSLC</td>
<td>( O_i = X_5 ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) ) ( S_1 = S_1 ) ( O_3 = S_2 + \sum X_i \cdot (X_i + 4) )</td>
</tr>
</tbody>
</table>

**Legend**
- Xi: Pipe Input for \( i = 1, 2, \ldots, 8 \) (S1R, S1L, etc)
- Oi: Pipe Output for \( i = 1, 2, 3, 4 \) (D1R, D1L, etc)
- Si, Ti: Accumulators S and T for \( i = 1, 2, 3, 4 \)
- T(Xk): Table Value with argument Xk

**Notes**
- Reserved for 2nd Release