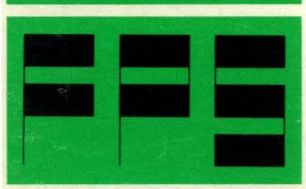


FLOATING POINT
SYSTEMS, INC.

Programmers Reference Manual Part One

by FPS Technical Publications Staff

Programmers Reference Manual Part One

1st Edition, January 1978

Publication No. FPS-7319

NOTICE

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

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

Printed in U.S.A.

CONTENTS

CHAPTER 4 DETAILED DESCRIPTIONS OF THE FUNCTIONAL UNITS

FIGURES

TABLES

APPENDIX

CHAPTER 1

INTRODUCTION

1.1 PURPOSE

This manual provides the information necessary for the programmer to write programs for the AP-120B Array Processor to be assembled by the AP Assembler (APAL). It is designed for use both as an introduction to programming the AP-120B and as a reference manual.

1.2 SCOPE

This manual contains material on how to program the AP-120B including detailed descriptions of the AP's functional units and its instruction set.

Chapter 1 includes a general description of the AP-120B and an example of its application. Chapter 2 introduces the AP-120B's functional units.

A review of floating point arithmetic theory and the format used by the AP-120B are in Chapter 3.

Chapter 4 includes detailed descriptions of the functional units, including programming examples and considerations. How to take advantage of the AP-120B's pipeline processing is discussed in Chapter 5, "How to Program the AP-120B."

The Appendix includes both a brief summary of the instruction set, Appendix D, and a discussion of each instruction, Appendix E. A diagram of the instruction field layout in Appendix D shows at a glance the relationship of the functional units to the instruction word.

For information about the Software Packages supplied with the AP-120B, the AP Math Library or the Program Development Package refer to Table 1-1, Related Publications, in Section 1.6.

1.3 GENERAL DESCRIPTION

The AP-120B is a high-speed (167-ns cycle time) peripheral floating-point arithmetic Array Processor, which is intended to work in parallel with a host computer.

Its internal organization is particularly well suited to performing the large numbers of reiterative multiplications and additions required in digital signal processing, matrix arithmetic, statistical analysis, and numerical simulation.

The highly parallel structure of the AP-120B allows the "overhead" of array indexing, loop counting, and data fetching from memory to be performed simultaneously with arithmetic operations on the data. This allows much faster execution than on a typical general-purpose computer, where each of the above operations must occur sequentially.

The AP-120B achieves its high speed through the use of fast commercial integrated circuit elements and an architecture that permits each logical unit of the machine to operate independently and at maximum speed.

Specifically:

1. Programs, constants, and data each reside in separate, independent memories, to eliminate memory accessing conflicts.
2. Independent floating-point multiply and adder units allow both arithmetic operations to be initiated every 167 ns.
3. Two large (32 locations each) blocks of floating-point accumulators are available for temporary storage of intermediate results from the multiplier, adder, or from memory.
4. Address indexing and counting functions are performed by an independent integer arithmetic unit that includes 16 integer accumulators.

In a typical application, such as a Fast Fourier Transform, the above features allow nearly the entire computation to be overlapped with data memory access time.

Effective processing precision is enhanced by 38-bits of internal data width, an internal floating-point format with optimum numerical properties, and a convergent rounding algorithm.

1.4 SYSTEM OVERVIEW

A general block diagram of AP-120B arithmetic paths appears in Figure 1.1.

Connection is made to the host in a manner that permits data transfers to occur under control of either the Host Computer or the AP-120B. For most host computers, this will mean that the AP-120B is interfaced to both the programmed I/O and DMA channels.

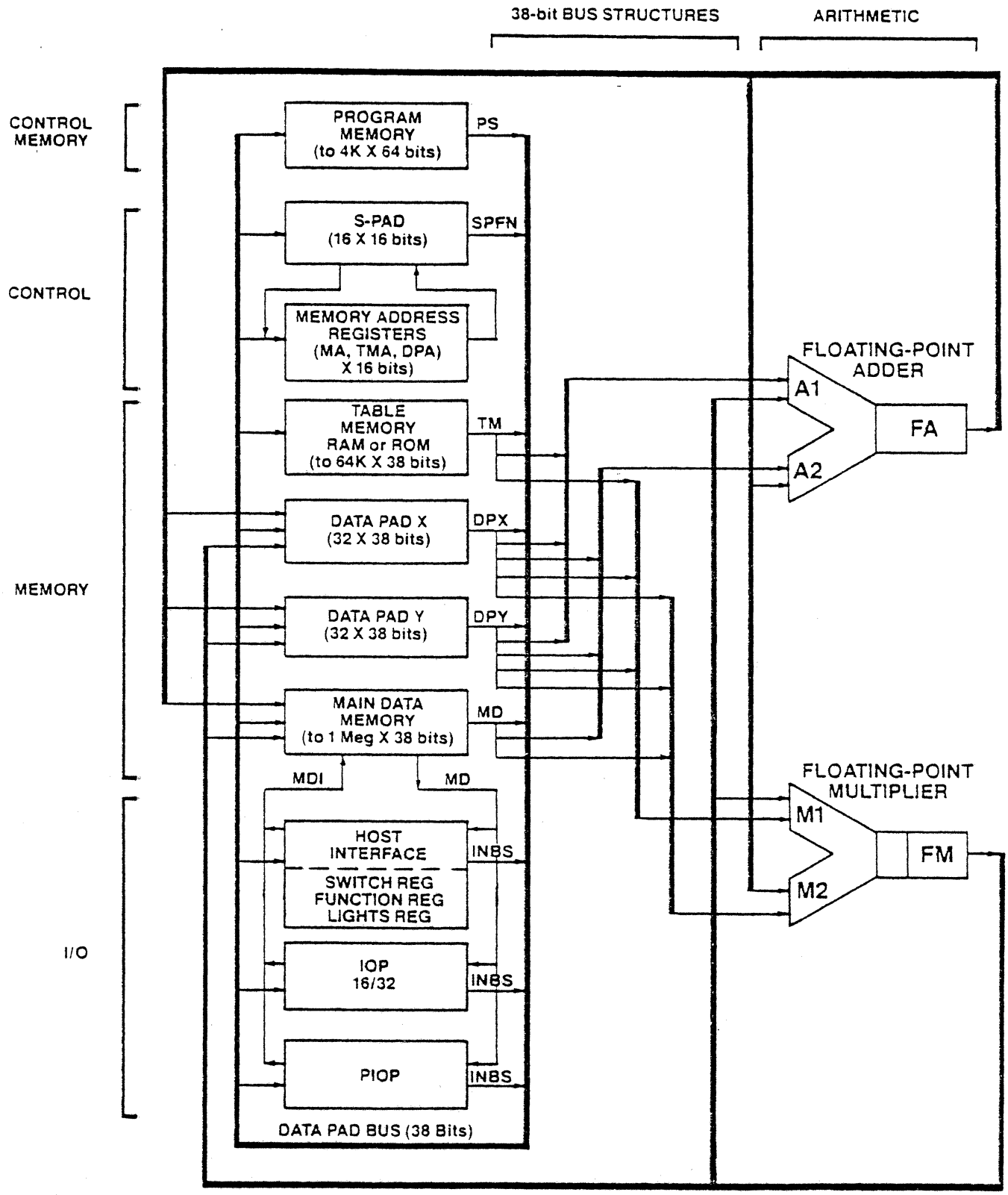


Figure 1-1 AP-120B Arithmetic Paths

The system elements are interconnected with multiple parallel paths so that transfers can occur in parallel. All internal floating-point data paths are 38-bits in width (10-bit biased binary exponent and 28-bit 2's complement mantissa).

Data Memory (MD) is organized in 8K-word modules of 38-bit words each, expandable up to 64K words in the main chassis. The effective memory cycle time (interleaved) is 333 ns.

Table Memory (TM) is used for storage of constants (FFT constants), and is tied to a separate data path so as not to interfere with Data Memory. It is bipolar, 167 ns read-only memory, and is organized in 512-word, 38-bit increments.

Data Pad X (DPX) and Data Pad Y (DPY) are two blocks of 32 floating accumulators each. Each is a two-part register block, wherein one register may be read and another written from each block in one instruction cycle.

The Floating Adder (FA) consists of two input registers (A1 and A2) and a two-stage pipe-line which performs the operations, and convergently rounds the normalized result.

The Floating Multiplier (FM) consists of input registers (M1 and M2) and a three-stage pipe-line which performs the multiply operation. Products are normalized and convergently rounded 38-bit numbers.

The S-PAD consists of 16 integer registers and an integer arithmetic unit which is used to form operand addresses and to perform integer arithmetic.

1.5 EXAMPLE AP-120B APPLICATION

A simple FFT processing sequence would go as follows:

Initial conditions are that the FFT program is resident in Program Source Memory internal to the AP-120B, the array to be transformed is resident in host memory, and the host CPU has initiated the AP-120B processor with an I/O instruction.

1. The AP-120B requests host DMA cycles to transfer the array from host memory to internal data memory. Data is converted from host floating-point format to internal AP-120B floating-point format "on the fly."
2. The FFT algorithm is performed, with data remaining in internal AP-120B format. This yields the benefit of 38-bit precision and convergent rounding during the critical phases of processing.
3. The frequency domain array is transferred back to host memory by requesting host DMA cycles. Data is converted from internal format to host format "on the fly."
4. The AP-120B proceeds to another process or stops executing, depending on previously established conditions. An interrupt to the host can be issued.

The AP-120B is most efficiently used when a sequence of operations is performed on one or more sets of data which reside in internal data memory. This reduces data-transfer overhead, and retains maximum numerical precision. For example, a reasonable sequence would be to transfer a trace and a filter, FFT both, array multiply, inverse FFT, and transfer the result back to host memory.

The AP-120B Data Memory has DMA capability. That is to say that MD cycles can be stolen from the AP-120B microprocessor by the interface. This capability allows Host Computer DMA to AP-120B DMA data transfers to occur, thereby minimizing both host CPU and AP-120B overhead.

The AP-120B has been designed with enough flexibility built-in so that its power can be harnessed in a variety of ways. Subsequent sections describe its use in detail.

1.6 SOFTWARE

Four packages of software are supplied with the AP-120B which assist the user toward the solution of his particular processing task.

1.6.1 APEX (A.P. Executive)

APEX is a mechanism for communicating with the AP-120B via a series of FORTRAN or machine language subroutine "calls." The executive driver routine interprets the particular user call and directs the AP-120B to perform the specified action. For example, in Fortran, to load an array A containing N real data points into the AP-120B, and perform a real Fast Fourier Transform upon that data:

```
IA=0
CALL APPUT (A,IA,N,2)
CALL RFFT (IA,N,1)
```

Both the Standard Applications Subroutines described below and user developed AP-120B programs may be called from the host computer using APEX.

1.6.2 APMATH (A.P. Math Library)

These are subroutines written in AP-120B assembly language which are callable from host computer Fortran or machine language programs using APEX. They are listed in the AP-120B Math Library.

1.6.3 Program Development Package

Four Fortran IV programs which are compiled on the host computer during installation aid user program development.

These are:

1.	APAL	A.P. Assembly Language. A cross-assembler which provides a two pass assembly of symbolic coding into an object module. APAL generates detailed error diagnostics.
2.	APLINK	A.P. Linker. Links and relocates separate APAL object modules together into a single execution module.
3.	APDBUG	A.P. Debugger. An interactive debugging program. The user may selectively set breakpoints, examine and change memory and register contents, and run program segments.
4.	APSIM	A.P. Simulator. Called by APDBUG, APSIM provides a programmed simulation of the various hardware elements of the AP-120B. All timing characteristics of the AP-120B are emulated, and the floating point arithmetic is simulated (including rounding) to the least significant bit. APSIM is a convenient tool in bringing up new AP-120B programs off line without interfering with production runs.

1.6.4 APTEST (A.P. Test Programs)

APTEST is a collection of interactive diagnostic test and verify programs which aid in isolation of hardware faults. These are:

1.	APTEST	A.P. Tester. Exercises the Panel, DMA interface, and various internal registers and memories. Tests Main Data Memory with simple patterns and then with random numbers.

		Board level diagnostic indicators are provided.
2.	APPATH	A.P. Path Tester. Tests the various internal data paths and gives board level diagnostics.
3.	APARTH	A.P. Arithmetic Test. Tests the floating point adder, multiplier, and S-Pad arithmetic unit with pseudo-random number and operation sequences.
4.	FIFFT	Forward/Inverse FFT Test. Verifies the correct operation of the AP-120B as a complete unit by doing forward/inverse FFT transforms on both spikes and random number sequences.

Table 1-1 lists Floating Point Systems publications related to AP-120B software.

Table 1-1 Related Publications

Manual	Number
Processor Handbook	7259
Software Development Package Manuals (includes APAL Array Processor Assembly Language Manual, APLINK-Array Processor Linking Loader Manual)	7292
AP-120B DEBUG-Array Processor DeBugger Manual	7364
AP-120B Diagnostic Software Manual (includes APTEST, APPATH, APARTH, FIFFT)	7284
AP-120B Math Library Paarts I & II	7288-02, 03
IOP 16/38 Users Manual	7310R
Programmable I/O Processor (PIOP) (scheduled winter quarter 1978)	7350

CHAPTER 2

FUNCTIONAL OVERVIEW

The hardware of the AP-120B is composed of three types of functional elements.

1. Logical and control elements
    a. Control unit
    b. S-Pad unit

2. Floating-Point arithmetic elements
    a. Floating-point adder
    b. Floating-point multiplier

3. Memory elements
    a. Data Pad unit
    b. Main data memory unit
    c. Table memory unit

Each of these funtional units is INDEPENDENT and thus can independently perform the programmed operations for which it was designed in parallel with the other functional units.

2.1 CONTROL UNIT

The Control Unit, as illustrated by Figure 2-1, consists of:

a. Program Source Memory (PS)
b. Program Source Address (PSA) Register
c. Control Buffer (CB) with decoding logic
d. Subroutine Return Stack (SRS)
   with subroutine return stack pointer (SRA)

The operation of the AP-120B is controlled by the execution of 64-bit instruction words which reside in Program Source (PS) Memory. The program word for the next instruction to be performed is selected by the address in the Program Source Address (PSA) register. At the initiation of the next machine cycle, this program word is transferred to the Control Buffer (CB) where it is decoded and executed. The PSA is incremented by one unless a branch in the current instruction causes the PSA to move to another location in Program Source (PS) memory. Access to Program Source memory and instruction decoding are overlapped so that the AP-120B can operate at a 6 MHz rate (167 ns).

Branching is accomplished in two manners. A short-range branch is provided by adding the 5-bit branch displacement field to the current PSA. This gives a branch range of from -20(octal) to +17(octal). A long-range jump to any location in PS is accomplished by loading the desired target address into PSA.

Subroutine jumps are made by a "JSR" instruction which saves the current PSA in the Subroutine Return Stack (SRS) and sets PSA to the subroutine address. Return is via a "RETURN," which loads the PSA with the last entered return address on the SRS.

SRA (Subroutine Return Address) is the Subroutine Return Stack pointer, which is automatically incremented or decremented as subroutines are called and returns are made from the subroutine.

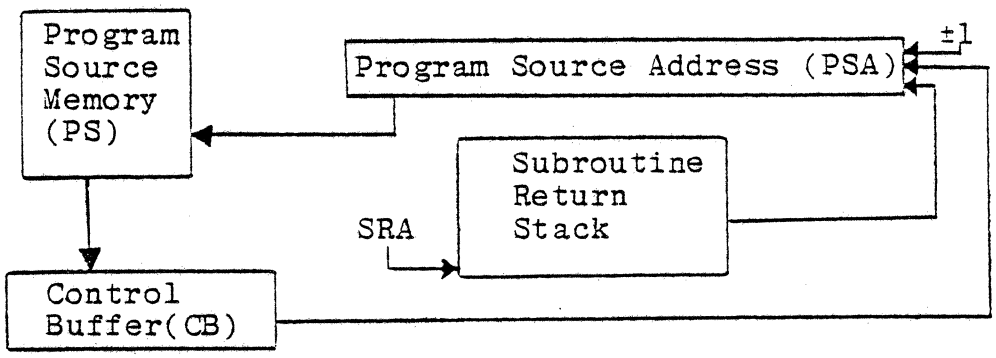


Figure 2-1 Control Unit

2.2 S-PAD UNIT

This unit, illustrated by Figure 2-2, performs the integer address indexing, loop counting and control functions necessary to direct completion of a given algorithm. In form, it is similar to familiar mini-computers such as the PDP-11 or Nova.

The S-Pad contains sixteen 16-bit directly-addressable registers. The contents of these registers pass through a special integer ALU associated with this unit.

The output of the ALU may be directed back to the specified S-Pad destination register, and also to any of the following address memory registers: Memory Address (MA), Table Memory Address (TMA), or Data Pad Address (DPA).

The S-PAD integer ALU functions include:

FUNCTION	EFFECT	
a. Move	S→D	S-Source register
b. Logical Complement	$\bar{S}$→D	
c. Clear	0→D	D-Destination register
d. Increment	S+1→D	
e. Decrement	S-1→D	
f. Add	D+S→D	
g. Subtract	D-S→D	
h. Logical AND	D AND S→D	
i. Logical OR	D OR S→D	
j. Logical Equivalence	D EQV S→D	

The output of the S-PAD ALU (called S-PAD FUNCTION or SPFN), may be used unmodified, shifted left once, shifted right once, or shifted right twice.

A hardware bit-reverse function included in the S-Pad accomplishes the bit swapping necessary to access data in scrambled order after an FFT.

The S-PAD ALU also sets three condition bits in the AP-120B Status Register depending upon the output of the ALU/shifter:

N: set if result <0; cleared otherwise

Z: set if result =0; cleared otherwise

C: set if a carry occurred; cleared otherwise

These bits may be tested by the next AP instruction, and a branch made depending upon whether the specified condition was true.

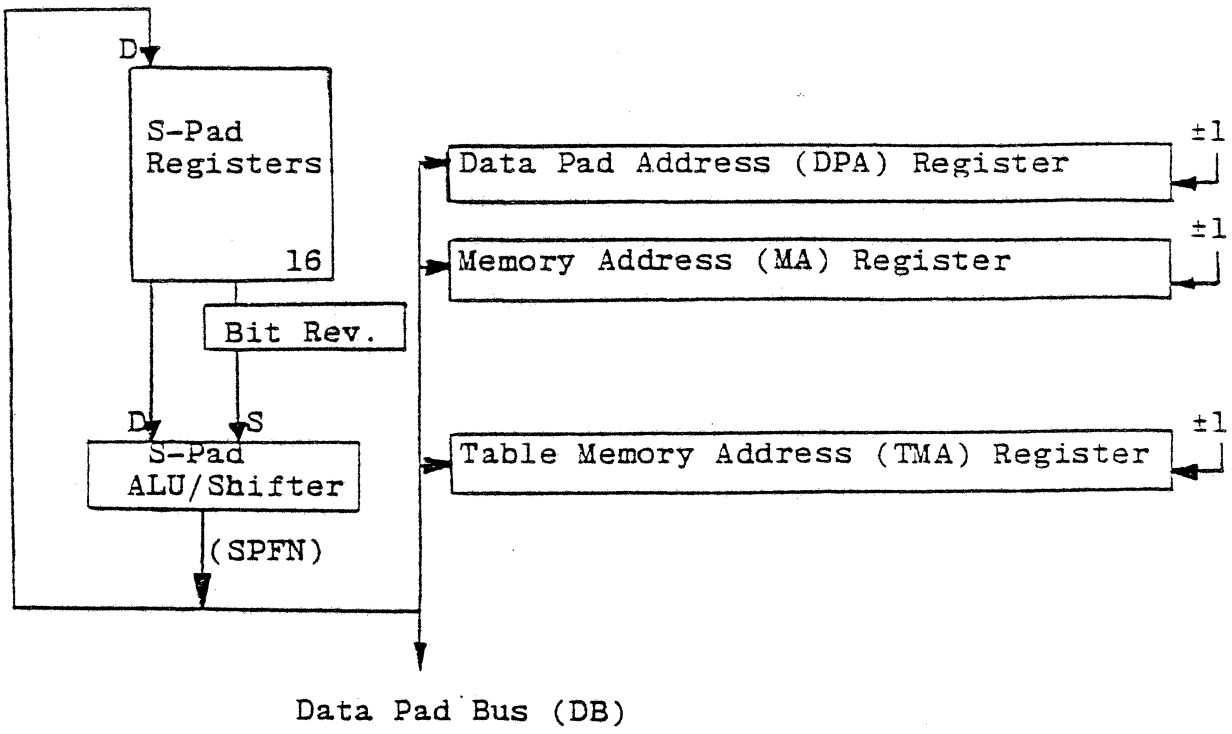


Figure 2-2 S-Pad Unit

2.3 FLOATING POINT ADDER UNIT

The Floating Point Adder, shown in Figure 2-3, does addition (or subtraction) operations on the contents of the Adder input registers (A1 and A2). The operation is performed in two stages, each of which takes one machine cycle.

In the first stage, the exponents of the two numbers are compared and the fractions are aligned by shifting the fraction of the smaller number right. The fractions are then added (or subtracted). In the second stage the resulting fraction is normalized and convergently rounded.

Since the two stages are independent of each other, a new pair of numbers may be entered into A1 and A2 every AP cycle (167 ns). The result is available for use two cycles later (333 ns).

In effect, the Floating Adder (FA) is a pipeline, where new inputs may be entered into the pipeline stream every cycle. Initiation of an add operation loads the two numbers to be added into the A1 and A2 input registers. The previous Adder input is pushed down the pipeline to the Adder Buffer register. One cycle later the completed result (called FA) from the Buffer is available for storage or use by another unit. Thus a new add may be started every 167 ns, and the result is ready 333 ns later.

A1 may be loaded from Data Pad (DP), from the output of the Floating Multiplier (FM), or from Table Memory (TM). A2 may be loaded from Data Pad (DP), from the output of the Floating Adder (FA), or from Data Memory (MD).

The output of the Floating Adder (FA) may be directed to the Multiplier (M2), to the Adder (A2), to Data Pad (DP), or to Memory Input (MI).

The operations performed by the Floating Adder are:

a. A1+01A2
b. A1-A2
c. A2-A1
d. A1 EQV A2
e. A1 AND A2
f. A1 OR A2
g. Convert A2 from signed magnitude to 2's complement format
h. Convert A2 from 2's complement to signed magnitude format
i. Scale A2
j. Absolute value of A2
k. Fix A2

Four condition bits in the AP Status Register are set or cleared by the Floating Adder depending upon the current result:

FZ - set to one if result is zero, else cleared to zero.
FN - set to one if result is negative, else cleared to zero.
FO - set to one if exponent overflow occurred. The result was forced to the signed maximum value.
FU - set to one if exponent underflow occurred. The result was forced to zero.

The overflow and underflow bits remain set until cleared by the program.

These bits may be tested by the instruction after the Floating Adder result is completed; i.e., three cycles after the Floating Adder operation was initiated.

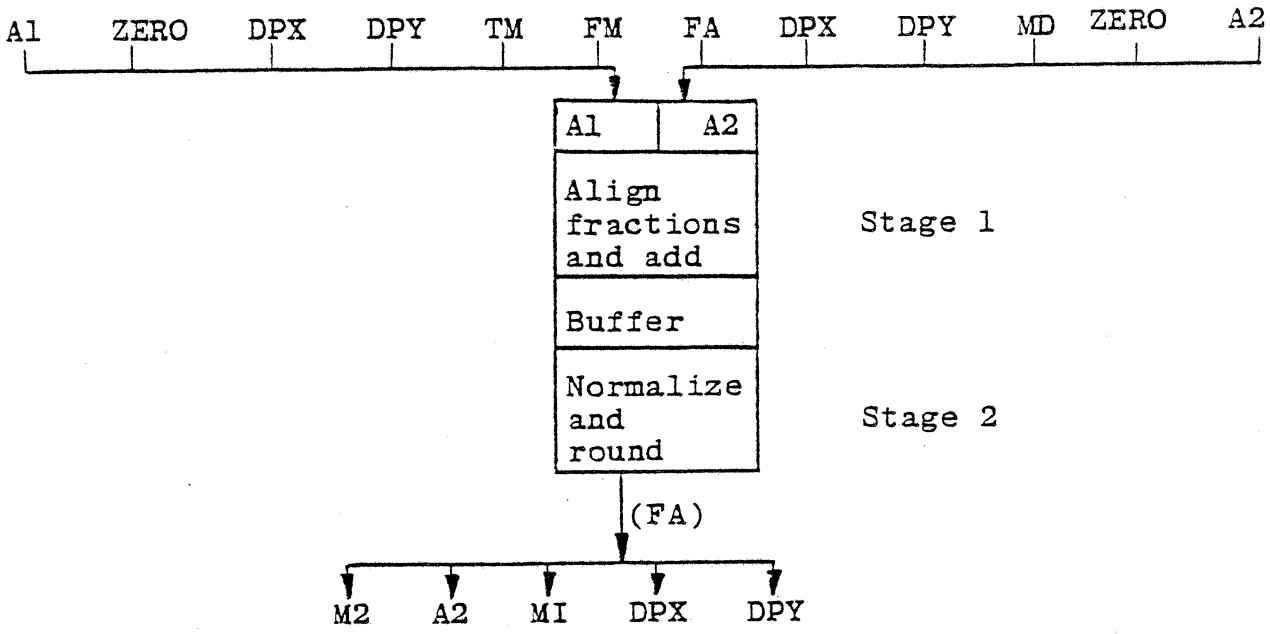


Figure 2-3 Floating-Point Adder Unit

2.4 FLOATING POINT MULTIPLIER UNIT

The Floating Multiplier, Figure 2-4, forms the product of the two multiplier input registers (M1 and M2). The product is formed in three stages, each of which takes one machine cycle.

In the first stage, the 56-bit product of the two 28-bit fractions are partially completed. The second stage completes the product of the fractions. In the third and final stage the exponents are added, and the mantissa product is normalized and convergently rounded.

The Floating Multiplier, like the Floating Adder, is organized as a pipeline. Initiation of a multiply loads the two numbers to be multiplied into the M1 and M2 input registers. The two previous multiplier inputs are pushed down the pipeline to Buffer 2 and Buffer 3 respectively. One cycle later, the result from Buffer 3 is available for storage or use by another unit.

Thus a new product may be started every 167 ns, and the result is ready 500 ns later.

M1 may be loaded from Data Pad (DP), the output of the Floating Multiplier (FM) or from Table Memory (TM). M2 is loaded from Data Pad (DP), the Adder (FA), or from Main Data Memory (MD).

Two error bits in the AP Status Register are affected by the Floating Multiplier:

FO - set if exponent overflow occurred. The result was forced to the signed maximum value.
FU - set is exponent underflow occurred. The result was forced to zero.

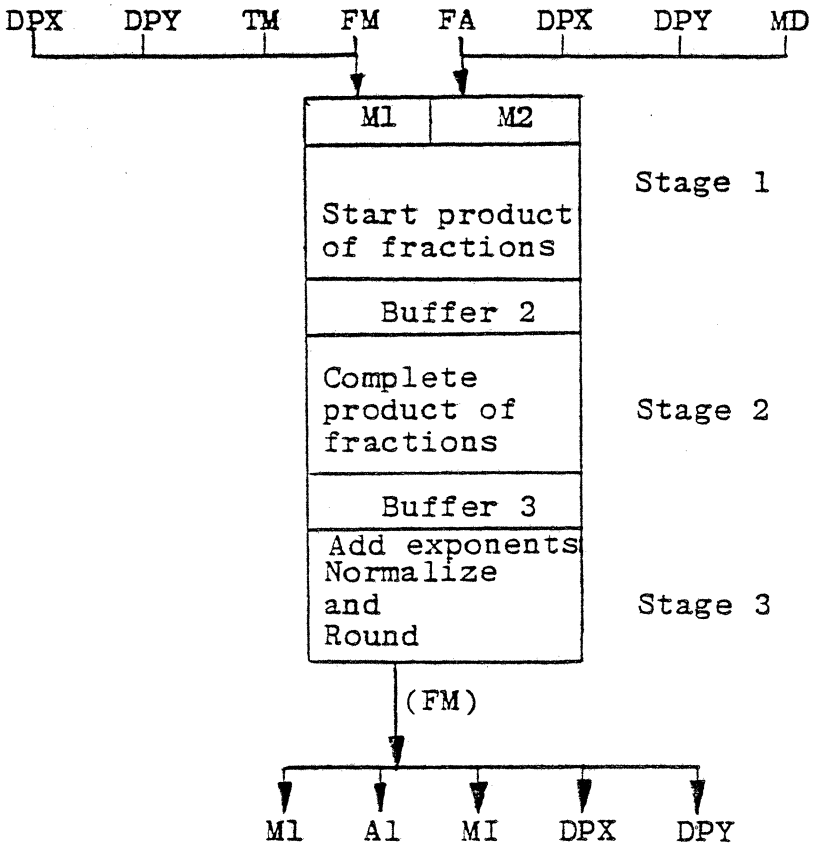


Figure 2-4 Floating Multiplier

2.5 DATA PAD UNIT

Data Pad, illustrated by Figure 2-5, consists of two fast accumulator blocks, each with 32 floating-point locations, called Data Pad X (DPX) and Data Pad Y (DPY). In a single machine cycle the contents of one location from each Data Pad may be read out and used. In addition, data may also be stored into one location in each Data Pad in the same cycle. That is, for example, in a single instruction (167 ns) a multiply may be initiated specifying one argument from DPX and another from DPY; an Adder result (FA) may be stored into a DPX location, and a data element in Main Data stored into a DPY location. On the very next instruction similar multiple Data Pad accessing could be accomplished again.

The two memories are addressed via a combination of the Data Pad Address (DPA) register and four index field values contained in a given instruction word. DPA may be thought of as a base address register or stack pointer. It may be loaded from the S-Pad (SPFN) or its contents may be incremented or decremented by one.

For a given read or write operation, say reading from Data Pad X, an index value contained in the instruction is added to the current contents of DPA to give the effective address for that particular operation. The four index fields (one each for read DPX, read DPY, write DPX, and write DPY) are each 3 bits wide, and have a range from -4 to +3 relative to DPA.

Data from either Data Pad may be used by the Multiplier (M1, M2), Adder (A1, A2), or Memory Input (MI). Data may be stored into Data Pad from the Adder (FA), Multiplier (FM), S-Pad Function output (SPFN), the Command Buffer Value (VALUE), or from Data Pad (DP).

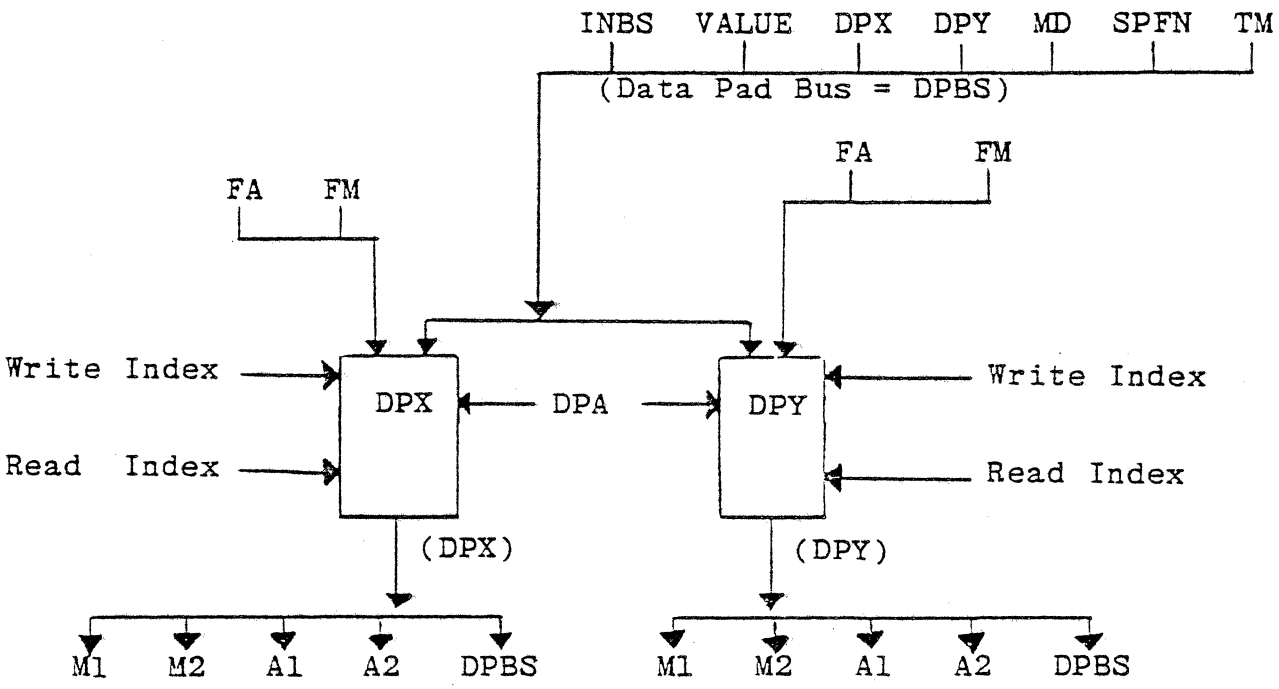


Figure 2-5 Data Pad

2.6 DATA MEMORY UNIT

The Data Memory unit, illustrated in Figure 2-6, is the primary data store for the AP-120B. It is available in 38-bit wide 8K modules which have an interleaved cycle time of 333 or 167 ns.

The memory unit contains a Memory Data (MD) buffer and a Memory Input (MI) buffer. Data read from memory is placed by the controller into MD, while data is written into memory from the MI. The Memory Address (MA) register points to the desired memory location.

In referencing memory for read or write operations, the selected operation is initiated by making a change to the Memory Address (MA) register. The MA register may be loaded from the S-Pad (SPFN) or its contents incremented or decremented by one.

A write operation is specified by loading MI with the data to be written during the same instruction in which MA is changed. This data is then written into memory from MI during the next two AP cycles. Data may be loaded into MI from the Floating Adder (FA), Floating Multiplier (FM), Data Pad (DP), Memory (MD), Table Memory (TM), the Input Bus (INBS), S-Pad Function (SPFN), or the Command Buffer Value (VALUE). A memory operation may be initiated every other cycle. The intervening cycle may be used for any other AP-120B function except another memory initiate.

When a memory READ is initiated, the requested memory data is placed by the memory controller into the Memory Data (MD) register 3 cycles after the request was made. Two instructions after the read request, another memory operation may be initiated. Again, the intervening cycle may be used for any non-memory functions. Data in MD may be used by the Floating Adder (A2), Floating Multiplier (M2), or Data Pad (DP).

To optimize the operation of the AP-120B it is necessary for the programmer to "look ahead" and initiate memory reads prior to the actual time that arguments from data memory are to be used in a calculation.

The system provides a "memory lock-out" which serves to insure that erroneous reads and writes of memory do not occur. If a memory initiate occurs while memory is "busy," further program execution is halted until the previous memory cycle is completed.

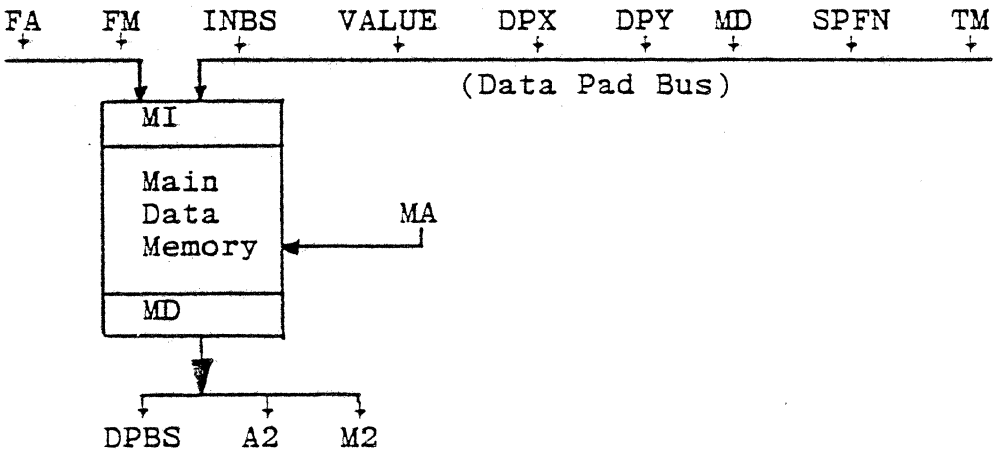


Figure 2-6 Data Memory Unit

2.7 TABLE MEMORY UNIT

The repeated use of standard constants (such as complex roots of unity and transcendental values) in signal processing routines dictate their ready availability to the programmer. A separate Table Memory (TM), shown in Figure 2-7, eliminates memory accessing conflicts by allowing data values and table values (constants) to be placed in separate memory banks.

Values read from Table Memory are placed by the controller into the Table Memory (TM) buffer register. The Table Memory Address (TMA) register serves as a pointer to the desired location. The standard TM is ROM. RAM is available as an option.

A Table Memory read is initiated by changing the contents of TMA, either by loading a value from the S-PAD (SPFN), or by incrementing or decrementing the contents of TMA.

A new table value may be requested every machine cycle. This value is available for use two cycles later. The value may be used by the Floating Adder (A1), Floating Multiplier (M1), or Data Pad (DP).

In FFT mode (i.e., when a FFT is being computed), the address in TMA is interpreted by the hardware to be an angle which points to the appropriate root of unity for a particular step in the algorithm. This allows the full table of roots of unity to be compressed into a single quadrant of cosines.

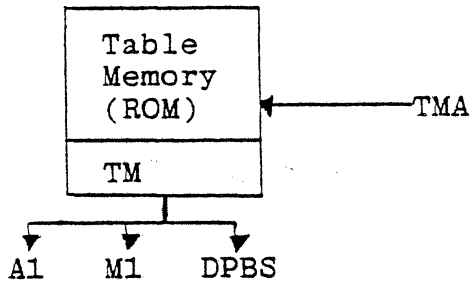
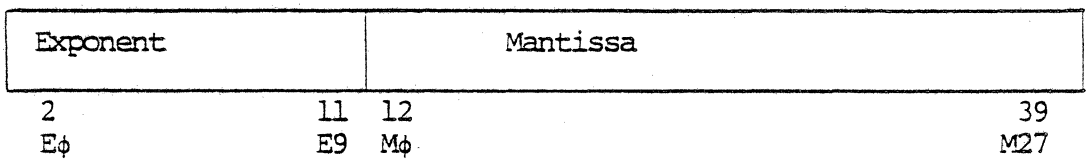


Figure 2-7 Table Memory

2.8 INTERNAL FLOATING POINT FORMAT

Floating-point data internal to the AP-120B is represented as follows:



Where:

Mantissa	28-bit two's complement fraction
Exponent	10-bit binary exponent, biased by 512

The value of a floating-point number in this format is defined as:

Mantissa * 2 (Exponent -512)

The dynamic range of this format is from 0.5 * 2(-512) to (1-2 [-28]) * 2(511); or, from 3.7 * 10(-155) to 6.7 * 10(153).

The 28-bit fraction, combined with the convergent rounding algorithm used in the Floating Adder and Multiplier, gives a maximum relative error of 7.5 * 10(-9) per arithmetic operation. This is a precision of 8.1 decimal digits. As a comparison, unrounded IBM 360 format gives only 6.0 decimal digits of arithmetic accuracy.

The convergent rounding hardware rounds up when the magnitude of the remainder is GREATER than 1/2 of the least significant bit of the mantissa. This serves to minimize truncation errors in long series of arithmetic calculations.

Format conversion between Host format and AP-120B format occurs in the Interface and in the Floating Adder unit. The dynamic range of the internal format is large enough to accomodate IBM 360 format and other Host formats. The extended precision of the AP-120B internal format insures that accuracy is maintained during critical stages of data analysis.

CHAPTER 3

FLOATING POINT ARITHMETIC THEORY

The subject matter within this summary requires a certain amount of preparatory discussion regarding types of numbering systems and notation as well as explanations of AP-120B FLOATING POINT number format and FLOATING-POINT ARITHMETIC OPERATIONS.

Those familiar with computer numbering systems may wish to bypass the preliminary sections of this summary and begin with the section dealing with the AP-120B FPN (FLOATING-POINT NUMBER) format. If the user understands the AP Floating-Point format in Section 3.4 and the fundamentals of floating-point addition and multiplication he can skip to Chapter 4.

Accordingly, this summary is presented in sections - general to specific - in the following manner:

1) INTRODUCTION

2) GENERAL NUMBERING SYSTEMS

   * BASE

   * RADIX

   * TYPES OF NOTATION METHODS

3) NUMBER FORMATS

   * FIXED-POINT

   * FLOATING-POINT

   * NORMALIZATION

4) AP-120B FLOATING-POINT NUMBER FORMAT

5) AP-120B FLOATING-POINT ARITHMETIC OPERATIONS (overview)

   * FLOATING-POINT ADDITION, SUBTRACTION AND MULTIPLICATION

   * ROUNDING/TRUNCATION

   * OVERFLOW AND UNDERFLOW

3.1 INTRODUCTION

The AP-120B FLOATING-POINT ARITHMETIC section consists of two units: the FLOATING-POINT ADDER (FADDR) and the FLOATING-POINT MULTIPLIER (FMULR). Both FADDR and FMULR operate on numbers represented in the AP-120B FLOATING-POINT NUMBER format (FPN).

The purpose of this summary is to acquaint the reader with a general overview on the most common types of digital computer numbering systems, and then to focus in on the format and characteristics of the particular numbering system used by the AP-120B FLOATING-POINT ARITHMETIC hardware.

3.2 GENERAL NUMBERING SYSTEMS

3.2.1 Base

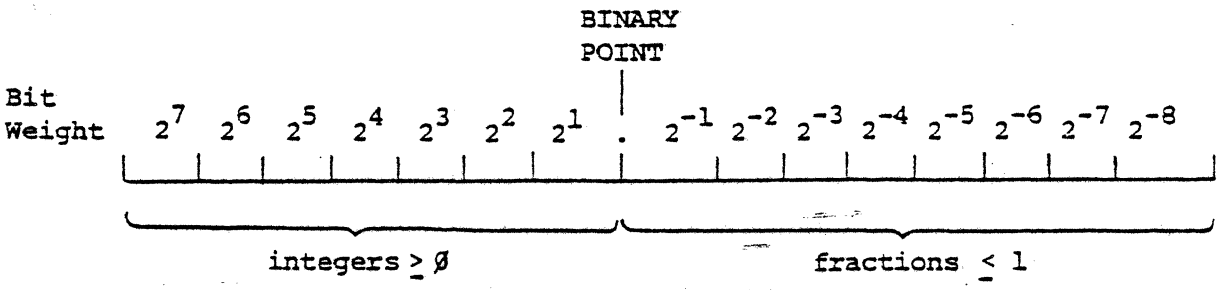
The BASE of a numbering system denotes how many different elements are used to represent progressive digits. For example, the decimal numbering system in general use today uses ten different elements to convey value (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The decimal system, then, is defined as a BASE(10) numbering system.

Digital computers, however, are not designed to efficiently handle a BASE(10) numbering system. Since the basic hardware element of most computers is a flip-flop or latch capable of only two significant states (1 and 0), the most convenient numbering system for digital computers is the two-element BINARY number system (BASE(2)), or numbering systems that are based on some power of two, such as OCTAL, (BASE(8)), or HEXI-DECIMAL (BASE(16)).

3.2.2 Radix Point

A RADIX POINT is that reflexive point where all numbers to the left represent integer quantities greater than or equal to 0 and where all numbers to the right represent fractional quantities less than 1.

In BASE(10) systems, the RADIX POINT is specifically termed the DECIMAL POINT. In BASE (2) systems, it is termed the BINARY POINT. Since the AP-120B uses a BASE(2) number system, the term BINARY POINT will be used in this summary.



3.2.3 Types of Binary Notation Systems

Once the base of a numbering system has been established, the question remains - what type of notation system will be used to represent a BINARY NUMBER in such a manner as to convey not only magnitude, but also sign.

There are four popular systems of notation used to accomplish this purpose.

1) SIGNED-MAGNITUDE notation

2) ONES-COMPLEMENT notation

3) TWOS-COMPLEMENT notation

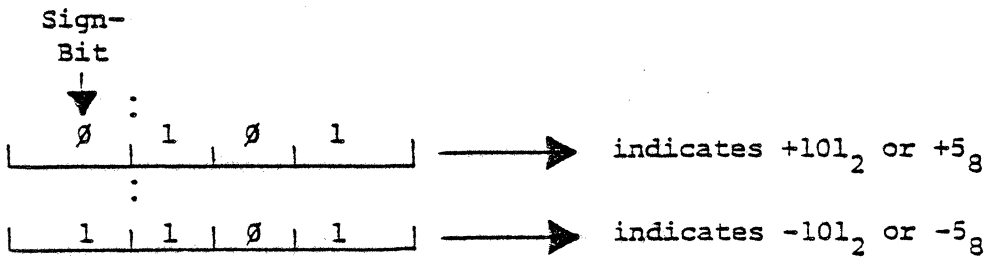
4) EXCESS or BIAS notation

* Signed-Magnitude

SIGNED-MAGNITUDE notation uses a Sign-Bit to indicate the sign of the quantity being represented. The integer value is in the same format for both negative and positive values - only the Sign-Bit changes to indicate the sign of the quantity represented. Usually the Sign-Bit is positioned to the left of the number and typically a "0" in the Sign-Bit position indicates a positive number, and a "1" indicates a negative number.

Example:

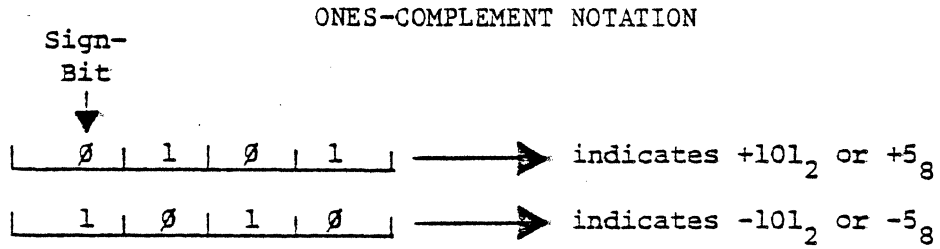
SIGNED-MAGNITUDE NOTATION



* Ones-Complement

ONES-COMPLEMENT notation also uses a Sign-Bit on the left of the number to denote sign. A positive number represented in SIGN-MAGNITUDE or ONES-COMPLEMENT is exactly the same in appearance. Negative numbers, however, are represented differently in that the Sign-Bit not only changes, but the quantity is complemented, as well.

Example:



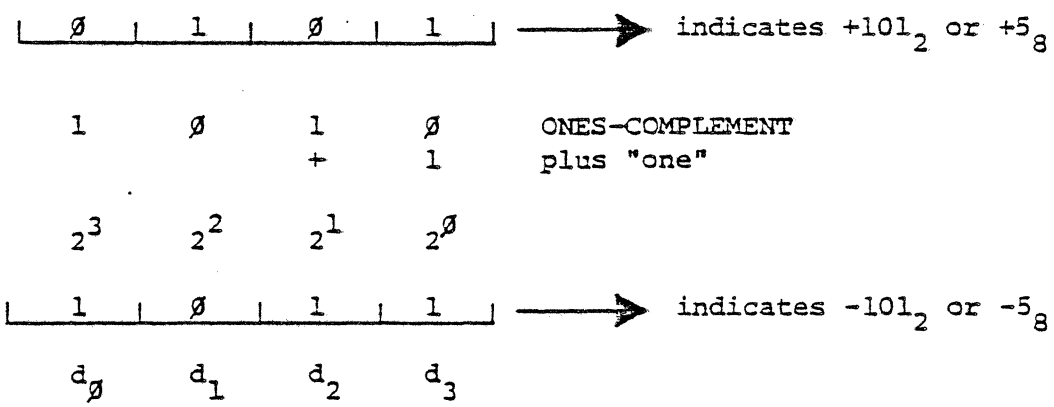
The major drawback of both SIGN-MAGNITUDE and ONES-COMPLEMENT notation systems is that there are two possible forms in which to represent the value ZERO (0).

* Twos-Complement

Positive values represented in TWOS-COMPLEMENT format appear the same as in the SIGN MAGNITUDE and ONES-COMPLEMENT formats. The SIGN-BIT is also to the left. However, for negative values, the number is complemented as in ONES-COMPLEMENT, and then a BINARY ONE is added.

Example:

TWOS-COMPLEMENT NOTATION



Integers represented in TWOS-COMPLEMENT notation may be expressed by the following equation:

$$TV = \left[(-d_Ø * 2^{N-1}) + \left(\sum_{i=1}^{N-1} d_i * 2^{N-(i+1)}\right)\right]$$

where: TV = TRUE VALUE
d(0) = digit in 0th position (SIGN BIT)
d(i) = digit in ith position
N = number of bits in the data word

Given below is a range of values for a 4-bit data word expressed in TWOS-COMPLEMENT notation.

	Sign			
Digit Weight	-2^3	2^2	2^1	2^0
Bit Position	0	1	2	3

						True Value (octal)
Range of all possible numbers	0	1	1	1		+7
	0	1	1	0		+6
	0	1	0	1		+5
	0	1	0	0	Positive Numbers	+4
	0	0	1	1		+3
	0	0	1	0		+2
	0	0	0	1		+1
	0	0	0	0	Zero	0
	1	1	1	1		-1
	1	1	1	0		-2
	1	1	0	1		-3
	1	1	0	0	Negative numbers	-4
	1	0	1	1		-5
	1	0	1	0		-6
	1	0	0	1		-7
	1	0	0	0		-10

N = number of bits in the data word

TWOS-COMPLEMENT notation has several advantages over SIGNED-MAGNITUDE and ONES-COMPLEMENT notation, in that:

* The problem of two possible forms of zero disappears; only one form of zero is possible, and

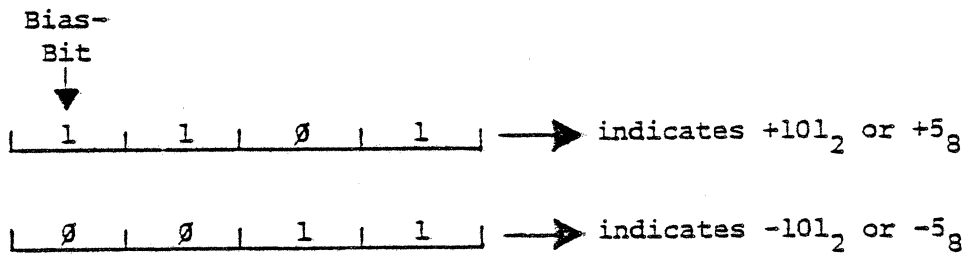
* TWOS-COMPLEMENT numbers may be added and subtracted without concern for the sign of each number if sign-extended by one bit. The result obtained from either operation will be correctly represented in TWOS-COMPLEMENT form.

Note also that the maximum-negative number possible in TWOS-COMPLEMENT is "1" greater (in magnitude) than the maximum-positive number.

* Excess (Bias) Notation

Another notation system capable of differentiating negative from positive quantities is the EXCESS or BIAS system. This method simply establishes a mid-point in the range of all possible numbers that can be represented in a given length data word. The mid-point is given a null or zero value and all numbers exceeding the null point are increasingly positive, and all numbers below the null point are increasingly negative.

BIAS NOTATION

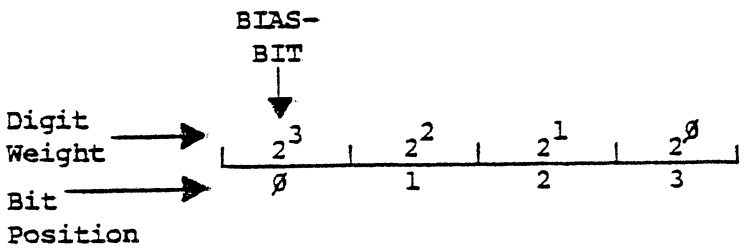


Integers represented in BIAS notation may be expressed by the following equation:

$$TV = \left[\left(\sum_{i+\emptyset}^{N-1} d_i * 2^{N-(i+1)}\right) - (2^{N-1})\right]$$

where: TV = TRUE VALUE
d(i) = Digit in ith position
N = Number of bits in the data word

Given below is the range of values for a 4-bit data word expressed in BIAS notation:



	Bias Bit					True Value (Octal)
	1	1	1	1		+7
	1	1	1	Ø		+6
	1	1	Ø	1		+5
	1	1	Ø	Ø	Positive Numbers	+4
	1	Ø	1	1		+3
	1	Ø	1	Ø		+2
	1	Ø	Ø	1		+1
Range of all possible numbers	1	Ø	Ø	Ø	← Mid Point (zero)	0
	Ø	1	1	1		-1
	Ø	1	1	Ø		-2
	Ø	1	Ø	1		-3
	Ø	1	Ø	Ø	Negative Numbers	-4
	Ø	Ø	1	1		-5
	Ø	Ø	1	Ø		-6
	Ø	Ø	Ø	1		-7
	Ø	Ø	Ø	Ø		-10

Three things become apparent:

1) The BIAS-BIT occupies bit position 0. When BIAS = "1", it indicates that a positive quantity resides in the remaining bits of the data word. When the BIAS = "0", it indicates that a negative quantity resides in the remaining bits.

2) The APPARENT VALUE exceeds the TRUE VALUE by the weight of the BIAS BIT. In other words:

   TRUE VALUE = APPARENT VALUE -2^{N-1}
   Where: N = the number of bits in the Data Word

3) The maximum-negative number possible is "1" greater in magnitude than the maximum-positive number possible.

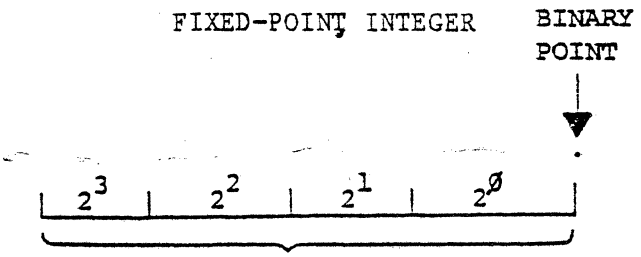
The AP-120B uses a mixed system of TWOS-COMPLEMENT notation and BIAS notation for the AP-120B FLOATING POINT NUMBER FORMAT.

3.3 NUMBER FORMATS

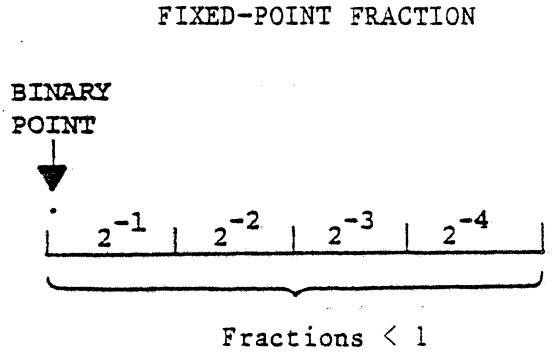
3.3.1 Fixed-Point Numbers

FIXED-POINT formatted numbers are numbers which have a stationary or fixed BINARY-POINT. FIXED-POINT formatted numbers are termed according to the location at which the BINARY-POINT is fixed. In other words, if a given number has a BINARY-POINT fixed to the extreme right of the data word, the number is said to be in the fixed-point INTEGER format, since all non-zero values that the format can represent are integers with magnitude $\geq$ 1. Conversely, if the BINARY-POINT is fixed to the extreme left of the data-word, the number is said to be in the fixed-point FRACTION format since all values that can be represented are fractions with magnitude $\leq$ 1. If the BINARY-POINT is fixed at some point between the two extremes, the number is said to be in fixed-point MIXED-NUMBER format.

Examples, assume a 4-bit data-word

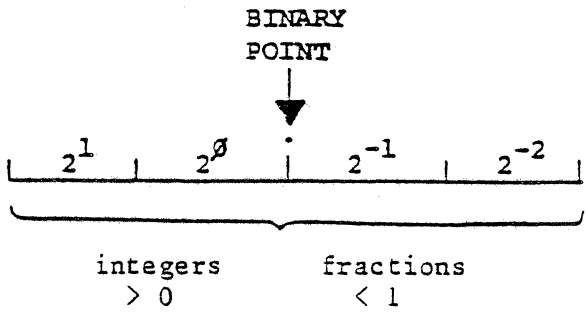


Magnitude of integers $\geq$ 0



Fractions < 1

FIXED-POINT MIXED-NUMBER



Since FIXED-POINT INTEGER formats and FIXED-POINT FRACTION formats are elements of the AP-120B FLOATING-POINT format, both are discussed in more detail, below:

FIXED-POINT INTEGERS

Unsigned FIXED-POINT INTEGERS may be expressed in the following equation:

$$TV = \left(\sum_{i=0}^{N-1} d_i * 2^{N-(i+1)}\right)$$

However, in order to represent a signed FIXED-POINT INTEGER, a notation system must be used. Given below is the equation for a FIXED-POINT INTEGER expressed in BIAS notation.

BIASED FIXED-POINT INTEGER

$$TV = \left[\left(\sum_{i=0}^{N-1} d_i * 2^{N-(i+1)}\right) - (2^{N-1})\right]$$

The utility of expressing a BIASED FIXED-POINT INTEGER will become apparent as AP-120B FLOATING-POINT NUMBERS are discussed, later in this summary.

FIXED-POINT FRACTIONS

Unsigned FIXED-POINT FRACTIONS may be expressed by the following equation:

$$TV = \left(\sum_{i=0}^{N-1} d_i * 2^{-(i+1)}\right)$$

However, in order to represent a signed FIXED-POINT FRACTION, we must express the fraction using a notation system. Given below, is the equation for a FIXED-POINT FRACTION expressed in TWOS-COMPLEMENT notation:

TWOS-COMPLEMENT FIXED-POINT FRACTION

$$TV = [(-d_{\emptyset} * 2^{\emptyset}) + (\sum_{i=1}^{N-1} d_i * 2^{-i})]$$

Again, the utility of expressing a TWOS-COMPLEMENT FIXED-POINT FRACTION will be seen when the AP-120B FLOATING-POINT format is introduced later in this summary.

3.3.2 Floating-Point Numbers

FLOATING-POINT NUMBERS are a product of two FIXED-POINT NUMBERS - the BASE raised to a SIGNED-EXPONENT (expressed is a FIXED-POINT INTEGER) times a SIGNED-FRACTION (MANTISSA) expressed as a FIXED-POINT FRACTION. A general expression for FLOATING-POINT NUMBERS is given below:

FLOATING-POINT NUMBER = BASE $^{\text{exponent}}$ * MANTISSA

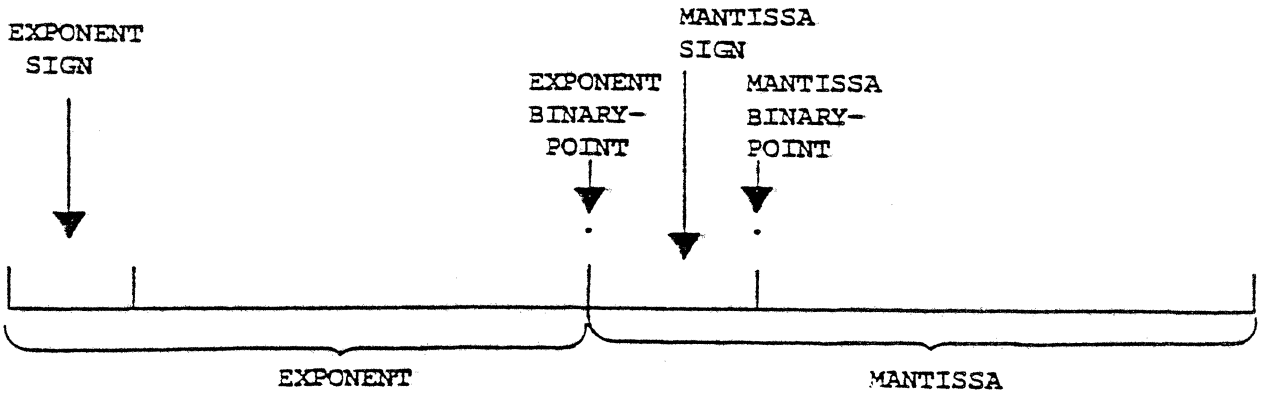
Where BASE = BASE of a given number system (in many digital computers, BASE = 2)

EXPONENT = Signed FIXED-POINT INTEGER

MANTISSA = Signed FIXED-POINT FRACTION

A FLOATING-POINT NUMBER is so-called because the BINARY-POINT is not fixed but is allowed to float, thereby increasing the range of numbers that may be represented in a fixed-length data word. Due to its logarithmic nature, a FLOATING-POINT NUMBER is able to represent numbers ranging from very small positive or negative fractions to very large positive or negative integers. A typical format for a FLOATING-POINT NUMBER is given on the following page.

TYPICAL FLOATING POINT NUMBER FORMAT



FLOATING-POINT NUMBERS possess unique manipulative characteristics in that a given number can be represented by various combinations of EXPONENT and MANTISSA values. The value of any given FLOATING-POINT NUMBER is preserved while changing the individual values of its component parts - as long as the EXPONENT is:

* Correspondingly incremented by the same number that the MANTISSA is right-shifted or,

* Correspondingly decremented by the same number that the MANTISSA is left-shifted

within the constraints imposed by a fixed-length data word.

Example:

The value (+.5) may be represented in FLOATING-POINT format as:

$$[(2^{0}) * (+.5)]$$

But, (+.5) can just as well be represented as:

$$[(2^{1}) * (+.25)]$$

or

$$[(2^{2}) * (+.125)]$$

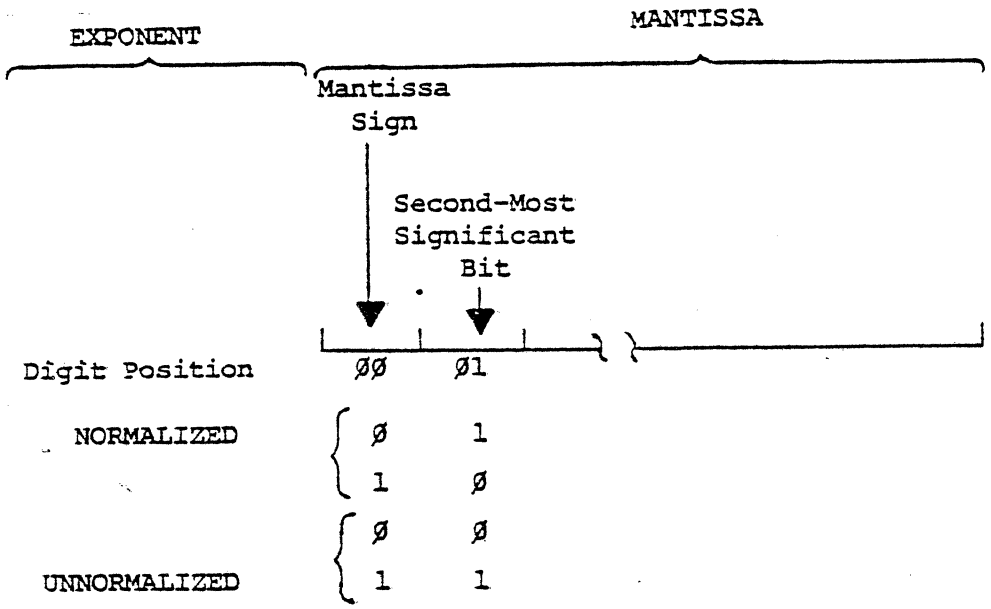
The difference is that only the first representation is NORMALIZED.

3.3.3 Normalization

NORMALIZATION of a TWOS-COMPLEMENT MANTISSA FPN is the process whereby the bits of the MANTISSA are left-shifted or right-shifted and the EXPONENT correspondingly decremented or incremented until the second-most significant bit of the MANTISSA is not equal to the

MANTISSA-SIGN (the most significant bit of the MANTISSA). A FLOATING-POINT NUMBER is said to be NORMALIZED only when the SIGN of the MANTISSA and the second-most significant bit of the MANTISSA are unequal. The major exception to this rule is a mantissa of zero which cannot be normalized. The above normalization rule guarantees that positive non-zero mantissas lie in the range 0.5 to 0.9999 (0.10000000 to 0.1111111111 in binary) and negative mantissas in the range -1.0 to -0.4999 (1.00000000 to 1.011111111111). Typically the unnormalized mantissa -0.5 (1.10000000) is allowed since it can be handled by the hardware.

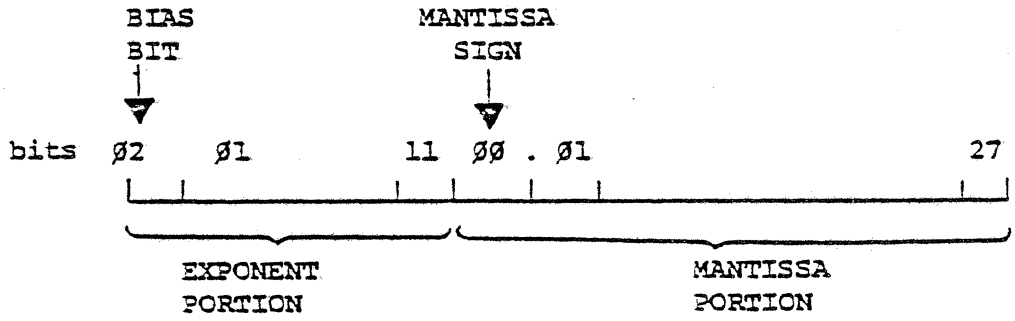
Example:



The major importance is that NORMALIZATION preserves the maximum number of significant bits in the MANTISSA while protecting against problems arising from OVERFLOW out of the bits of significance (the NORMALIZATION process CAN involve a right shift).

3.4 AP-120B FLOATING-POINT FORMAT (FPN)

The AP-120B uses a 38-bit data word to express FLOATING-POINT NUMBERS internally (See Figure 3-1). The most significant 10 bits of the word are dedicated to expressing the EXPONENT in biased notation and the remaining 28 bits are dedicated to expressing the MANTISSA in TWOS-COMPLEMENT notation.



THE AP-120B FLOATING-POINT NUMBER (FPN) may be expressed in the same manner used earlier to express FLOATING-POINT NUMBERS generally, viz.:

$$FPN = BASE^{exponent} * MANTISSA$$

In the AP-120B FLOATING-POINT format, however, the EXPONENT SIGN is indicated by use of a BIAS-BIT rather than a SIGN-BIT. Specifically, the EXPONENT of the AP-120B FPN is expressed as a BIASED FIXED-POINT INTEGER and the MANTISSA is expressed as a TWOS-COMPLEMENT FIXED-POINT FRACTION. The general equation may be re-expressed as follows:

$FPN = 2^{[BIASED\ FIXED\text{-}POINT\ INTEGER]} * [TWOS\text{-}COMPLEMENT\ FIXED\text{-}POINT\ FRACTION]$

Using the equations developed earlier in this summary for BIASED FIXED-POINT INTEGERS and TWOS-COMPLEMENTED FIXED-POINT FRACTIONS, we now have:

$$FPN = 2^{\left[\left(\sum_{i=\emptyset}^{N_e-1} d_i * 2^{N_e-(i+1)}\right)-\left(2^{N_e-1}\right)\right]} * \left[\left(-d_\emptyset * 2^{\emptyset}\right) + \left(\sum_{i=1}^{N_M-1} d_i * 2^{-i}\right)\right]$$

Where: N(e) = number of digits in the EXPONENT
N(m) = number of digits in the MANTISSA

Since, in the AP-120B, N(e) = 10 and N(M) = 28, the FPN may finally be the AP-120B, the AP expressed as follows (the exponent sum has been corrected for the fact that the digits start at number 2)

$$FPN = 2^{[(\sum_{i=2}^{11} d_i * 2^{11-i}) - (512)]} * [-d_0 + (\sum_{i=1}^{27} d_i * 2^{-i})]$$

Expressed in this manner, it can be seen that the effect of the BIAS scheme (with respect to the EXPONENT) is to shift the TRUE VALUE of the EXPONENT by a factor of (-512), the weighted value of the BIAS-BIT (2^9). Accordingly, the TRUE VALUE of the AP-120B FPN EXPONENT can be expressed in the following manner:

$$\begin{matrix}\text{TRUE} \\ \text{VALUE} \\ \text{(EXPONENT)}\end{matrix} = \begin{matrix}\text{APPARENT} \\ \text{VALUE} \\ \text{(EXPONENT)}\end{matrix} - 512$$

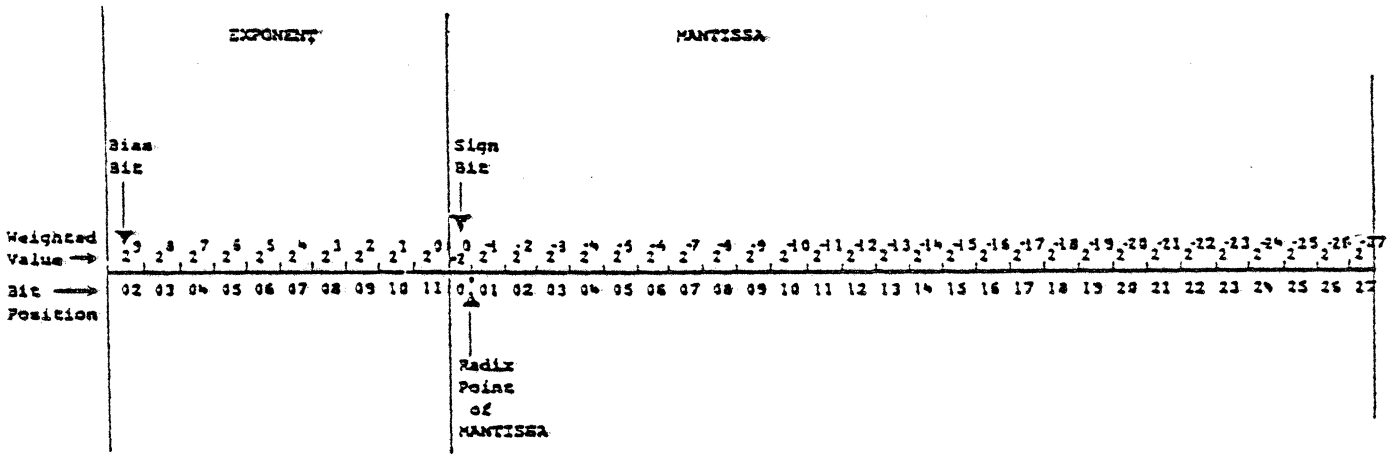
Note that the range of the FPN EXPONENT is:
[+511 to -512].

The range of the MANTISSA (.M) is:
$[+1 > .M \geq -1]$

The dynamic range for the AP-120B FPN is:

$[(0.5 * 2^{-512}) \text{ to } (1.0-2^{-27}) * 2^{511})]$ or,
from $3.7 * 10^{-155}$ to $6.7 * 10^{153}$

Figure 3-1 AP-120B FLOATING POINT NUMBER FORMAT



EXAMPLES

DECIMAL VALUE	EXPONENT (BIAS BIT ØØ, Ø1 … Ø9)	MANTISSA (SIGN ØØ, Ø1 … 27)	COMMENTS
Ø.Ø	Ø ØØØ ØØØ ØØØ	Ø ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	ZERO (See Note)
1.Ø	1 ØØØ ØØØ ØØ1	Ø 1ØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	$(2^{1})*(.5)$
-1.0	1 ØØØ ØØØ ØØØ	1 ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	$(2^{0})*(-1.0)$
.5	1 ØØØ ØØØ ØØØ	Ø 1ØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	$(2^{0})*(.5)$
-.5	1 ØØØ ØØØ ØØØ	1 1ØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	$(2^{0})*(-.5)$
$6.7 * 10^{153}$	1 111 111 111	Ø 111 111 111 111 111 111 111 111 111	APMAX
$-6.7 * 10^{153}$	1 111 111 111	1 ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	APMAX
$3.7 * 10^{-155}$	Ø ØØØ ØØØ ØØØ	Ø 1ØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ	APMIN

Although (0.0) and (-.5) examples are not in NORMALIZED form, both FADDR and FMULR can handle these two specific cases correctly.

NOTE

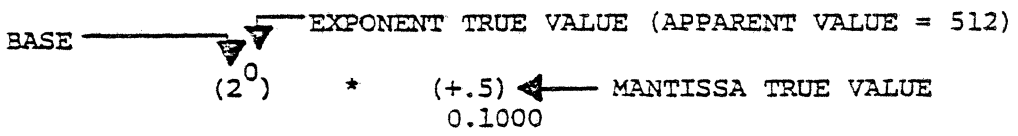
Although the APPARENT VALUE of ZERO EXPONENT = 0, the TRUE VALUE of ZERO EXPONENT = -512. (Which multiplied with a MANTISSA of .0, still equals 0.0).

3.5 AP-120B FLOATING-POINT ARITHMETIC OPERATIONS (OVERVIEW)

The AP-120B is capable of simultaneous FLOATING-POINT ADDER (FADDR) and FLOATING-POINT MULTIPLIER (FMULR) operation. This section will present a general overview of the arithmetic operations of both units and discuss operations common to both - such as requirements for NORMALIZATION and CONVERGENT-ROUNDING/TRUNCATION.

First, a brief review of the manipulative characteristics of FPN's is in order. As stated in the preceding section, the same-value FPN may be expressed in various combinations of EXPONENT and MANTISSA values. The key requirement for maintaining the value of any given FPN is that the EXPONENT must be correspondingly incremented or decremented by the same number that the MANTISSA is right-shifted or left-shifted, respectively. Example:

The number (+.5) may be represented, in NORMALIZED form, by the following combination of EXPONENT and MANTIS$A values - - -



However, the same number may be expressed, in UNNORMALIZED form, by the following combinations of EXPONENT and MANTISSA - - -

(2^1) * (.25)
0.01000

or

(2^2) * (+.125)
0.001000

Note that as the EXPONENT value, in each case, was incremented by 1, the corresponding MANTISSA value was decremented by means of halving - right-shifting the MANTISSA bits 1 position. Had the example decremented the sample EXPONENTS by some number "N", then the corresponding MANTISSA's, in each case, would have been incremented by means of left-shifting the MANTISSA bits "N" positions.

3.5.1 Floating-Point Addition, Subtraction and Multiplication

* Addition

In order to ADD two FPN's, the MANTISSA's of both operands must be expressed in terms of the same EXPONENT. This requirement is met, in the AP-120B, by means of a comparison operation which compares the EXPONENTS of the two operands - retaining the larger EXPONENT as the EXPONENT of the result and right-shifting the MANTISSA of the smaller operand the number of positions that reflect the difference between the two EXPONENTS. For example, assume the following ADD operation:

$$4 + 8 = 12$$

In FLOATING POINT format, the operands may be expressed as follows:

$$[(2^{3}) * (.5)] + [(2^{4}) * (.5)] = [\quad]$$
$$0.10000. \qquad 0.10000$$

First, the EXPONENTS are compared. The larger EXPONENT becomes the EXPONENT of the result while the MANTISSA of the smaller operand is right-shifted the number of positions that reflect the difference in the two EXPONENTS:

$$[(2^{4}) * (.25)] + [(2^{4}) * (.5)] - [(2^{4}) * (\ )]$$
$$0.010000 \qquad 0.100$$

Then, the MANTISSA's are algebraically added:

$$[(2^{4}) * (.25)] + [(2^{4}) * (.5)] = [(2^{4}) * (.75)]$$
$$0.110000$$

Note that the MANTISSA of the result is already NORMALIZED. Had it been UNNORMALIZED, then the FADDR logic would have left-shifted the MANTISSA and correspondingly decremented or incremented the EXPONENT until the second-most significant bit of the MANTISSA was unequal to the most- significant bit (MANTISSA sign).

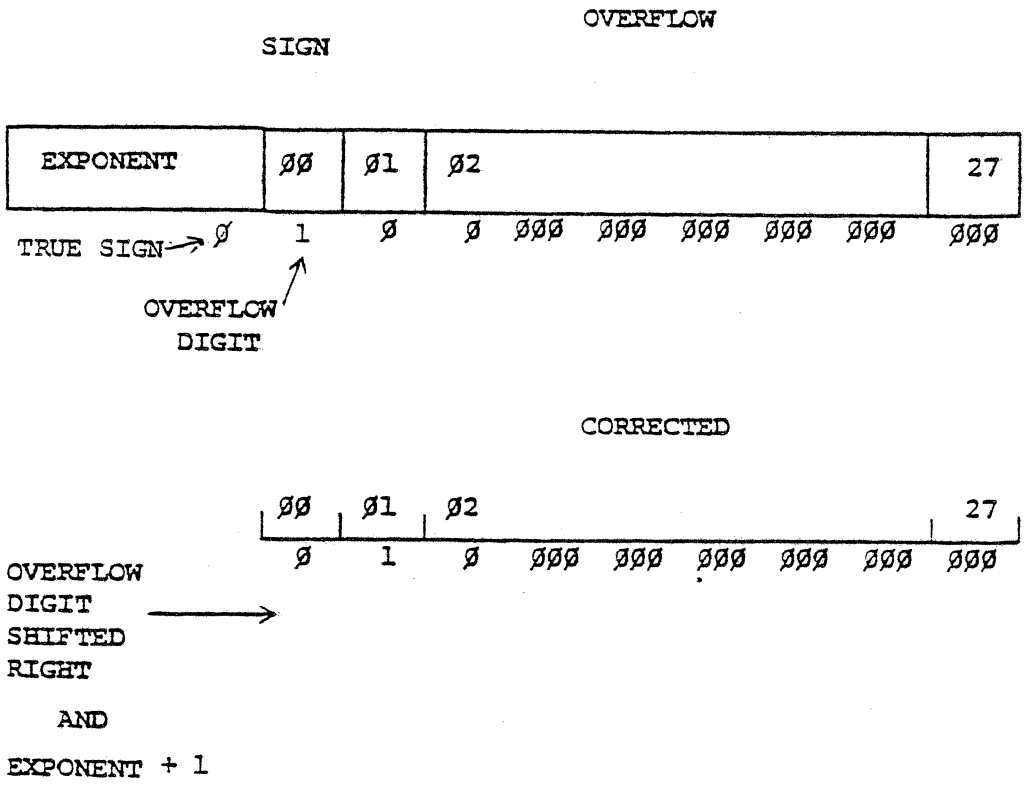
* Subtraction

SUBTRACTION of two FPN's is achieved by negating the subtrahend and adding the negated subtrahend to the minuend. Negation is achieved by two's complementing the subtrahend. See example below:

MINUEND	0100	+4	1100	-4
SUBTRAHEND	0011	+3	1101	-3
NEGATE SUBTRAHEND	1101	-3	0011	+3
RESULT	0001	+1	1111	-1

Mantissa Overflow

In MANTISSA OVERFLOW situations (when a digit of numerical significance has carried to the left of the radix point), the proper result is achieved by shifting the MANTISSA to the right one digit and incrementing the EXPONENT.



* Multiplication

MULTIPLYING two FPN's with different EXPONENTS requires that (1) the EXPONENTS be algebraically added, and (2) the MANTISSA's be multiplied. Except for two specific cases, the operands involved in FLOATING-POINT MULTIPLICATION must be NORMALIZED in order to obtain a correct result. As with the addition example above, detailed discussion of result NORMALIZATION and ROUNDING operations is reserved for later sections of this summary.

01Assume the following multiplication operation:

$$(12) * (.25) = 3$$

In AP-120B FLOATING=POINT format, the operands would be expressed as:

$$[(2^{4}) * (.75) * [(2^{-1}) * (.5) = [\quad]$$

First, the EXPONENTS are algebraically added. The sum of the EXPONENTS becomes the EXPONENT of the preliminary result.

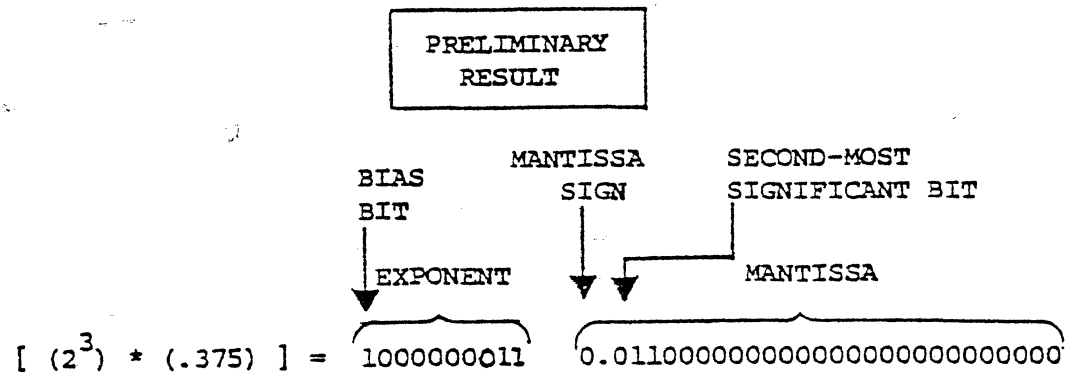
$$[(2^{4}) * (.75)] * [(2^{-1}) * (.5)] = [(2^{3}) * (\quad)]$$

Then, the MANTISSA's are multiplied. The product of the MANTISSA's becomes the MANTISSA of the preliminary result.

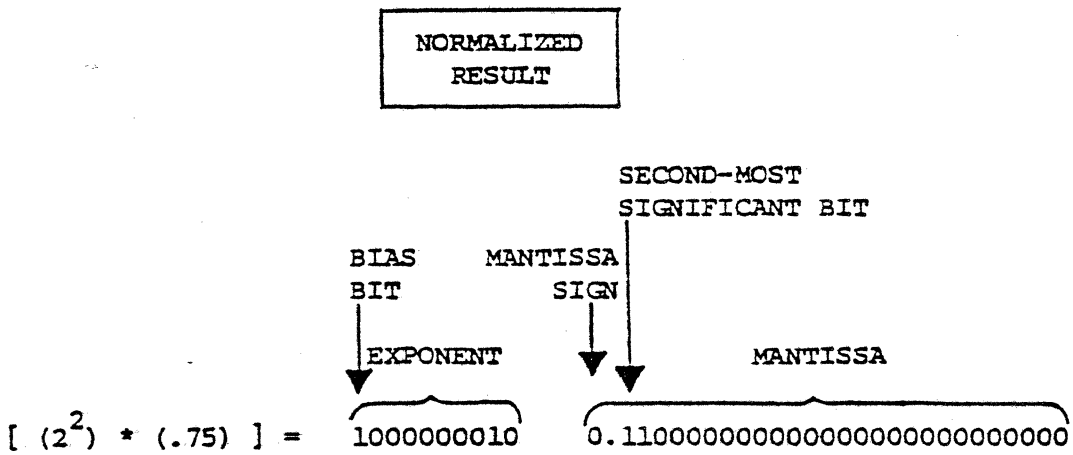
$$[(2^{4}) * (.75)] * [(2^{-1}) * (.5)] = [(2^{3}) * (.375)]$$

0.11000 0.1000 .01100000 Preliminary Result

Note, however, that the PRELIMINARY-RESULT is not NORMALIZED. Accordingly, the FMULR logic will left-shift the MANTISSA bits and correspondingly decrement the EXPONENT until the second-most significant bit of the MANTISSA is not equal to the MANTISSA-SIGN. To illustrate this operation, let us portray the PRELIMINARY RESULT, of the example, in its bit configuration:



The FMULR logic will left-shift the MANTISSA bits and correspondingly decrement the EXPONENT until the second-most significant bit of the MANTISSA is not equal to the MANTISSA-SIGN. (In this example, the MANTISSA is left-shifted one position and the EXPONENT is decremented by one).



Unlike the Floating Add Logic, the FMULR logic can only shift the MANTISSA one position left or right since it can be shown for normalized twos- complement mantissas that the worst case normalization required is only one position.

The AP-120B FMULR is capable of operating on the following UNNORMALIZED operands:

		EXPONENT	MANTISSA
(1)	ZERO =	0000000000	00000000000000000000000000

and,

(2) Any number with a MANTISSA of (-.5), such as:

EXPONENT	
any number	11000000000000000000000000

Note that ZERO has an exponent with a TRUE-VALUE of -512.

3.5.2 Rounding

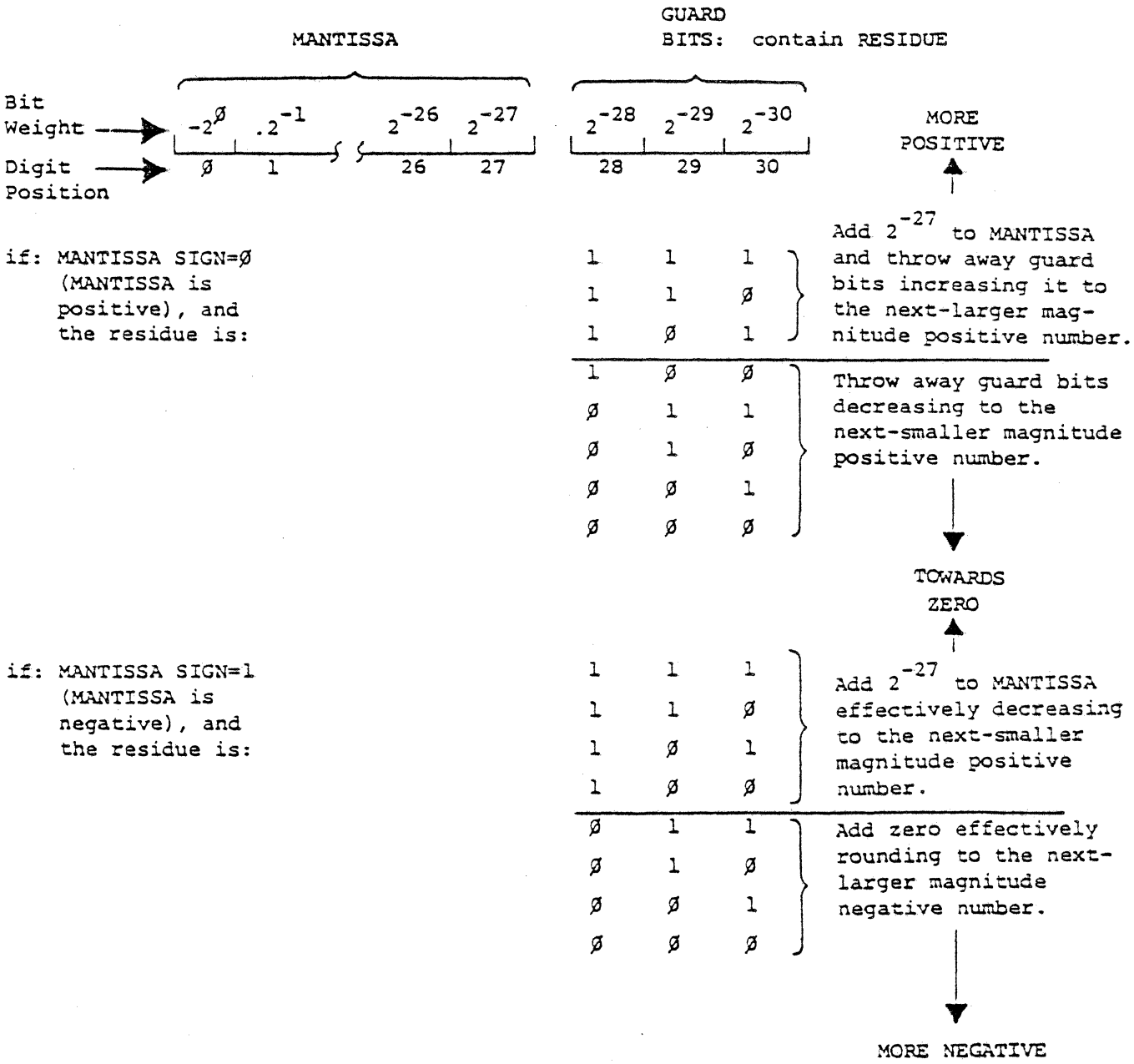
During a FLOATING-POINT MULTIPLY (FMUL) or most of the FADD group operations (See FADDR Summary, Chapter 4), the preliminary-result obtained is NORMALIZED and then CONVERGENTLY-ROUNDED. NORMALIZATION operations have been discussed earlier in this summary. The purpose of this section is to present the particular ROUNDING method employed by the AP-120B - - - CONVERGENT-ROUNDING.

CONVERGENT-ROUNDING is so-called because the frequency distribution curve for the rounding-decision matrix is slightly skewed toward zero (See Table 3-1).

Both FADDR and FMULR units employ a 3-bit extension of the PRELIMINARY-RESULT MANTISSA in order to store bits of significance generated off the least significant bit of the MANTISSA during a given operation. These bits - - called GUARD-BITS - - reside to the immediate right of MANTISSA bit 27 and have bit weights of 2^{-28}, 2^{-29}, and 2^{-30}, respectively.

Prior to NORMALIZATION bits 31 to 58 of the FADDR Result (bits 31 to 50 of the FMULR Result) are inclusively OR'ed into bit 30 of the residue. This has the effect of guaranteeing that the rounding occurs only when the magnitude of the residue is strictly greater than 2^{-28} (= one half of the LSB).

Table 3-1
The AP-120B ROUNDING-DECISION TABLE



The value contained in the GUARD BITS is called the RESIDUE. The magnitude of the RESIDUE and the sign of the PRELIMINARY-RESULT MANTISSA jointly determine the direction in which the PRELIMINARY-RESULT will be rounded (either toward the next-larger-magnitude number or toward zero).

If the rounding operation causes mantissa overflow, the AP-120B hardware takes care of it by shifting the mantissa to the right one digit and incrementing the exponent.

Note that the TRUE VALUE of the RESIDUE may be expressed as:

$$TV_{RESIDUE} = [(-d_{\emptyset} * 2^{-27}) + (\sum_{i=28}^{30} d_i * 2^{-i})]$$

Where: d(0) = Digit in 0(th) position (MANTISA-SIGN)

D(i) = GUARD-BIT in i(th) position

The ROUNDING-DECISION is based on the following relation:

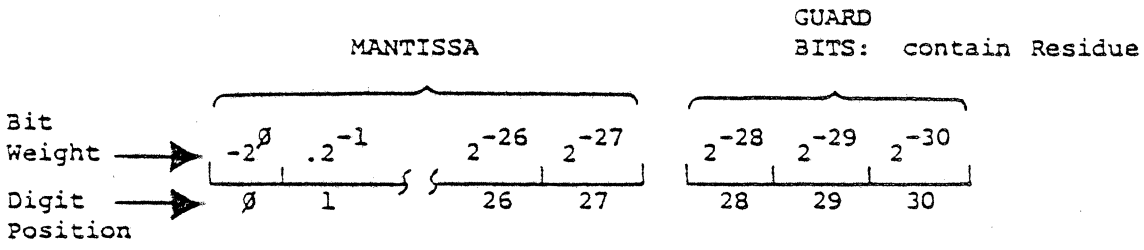
$$[(-d_{\emptyset} * 2^{-27}) + (\sum_{i=28}^{30} d_i * 2^{-i})] > [(.5) * (2^{-27})]$$

Where d(27) = Least significant bit of the preliminary-result MANTISSA

Note that if the relation is true and 11 = magnitude, the PRELIMINARY RESULT MANTISSA will be rounded to the next-larger magnitude number.

The floating adder has two operations which truncate--FSCLT and FIXT. When a preliminary result is positive, the FA truncates by ignoring the residue--throwing it away. If the Preliminary result is negative and there is any residue, 2(-27) is added to the mantissa; this makes the result closer to the next smaller magnitude number. Truncation always goes towards zero (See Table 3-2).

Table 3-2
The AP-120B TRUNCATION-DECISION TABLE



if: MANTISSA SIGN=Ø (MANTISSA is positive), and the residue is:	1	1	1	Throw away guard bits decreasing to the next-smaller magnitude positive number.
	1	1	Ø	
	1	Ø	1	
	1	Ø	Ø	
	Ø	1	1	
	Ø	1	Ø	
	Ø	Ø	1	
	Ø	Ø	Ø	

TOWARDS ZERO

if: MANTISSA SIGN=1 (MANTISSA is negative), and the residue is:	1	1	1	Add 2^{-27} to MANTISSA effectively decreasing to the next-smaller magnitude negative number.
	1	1	Ø	
	1	Ø	1	
	1	Ø	Ø	
	Ø	1	1	
	Ø	1	Ø	
	Ø	Ø	1	
	Ø	Ø	Ø	

3.5.3 Overflow And Underflow

The current result of a FADDR or FMULR operation (FA, FM) is tested as to OVERFLOW (See Note 1) and UNDERFLOW (See Note 2) conditions. If an OVERFLOW condition occurs, the AP-120B hardware will force the maximum signed Floating- Point number possible (APMAX or APNMAX) onto FA or FM and the OVF bit of the APSTATUS register will be set to "1". Similarly, if an UNDERFLOW condition occurs, the AP-120B hardware will force a Floating Point 0.0 (ZERO) onto FA or FM and set the UNF bit of APSTATUS REGISTER to "1".

	EXPONENT			SIGN	MANTISSA								
	ØØ		09	ØØ	Ø1								27
AP MAX	1	111	111 111	Ø	111	111	111	111	111	111	111	111	111
AP NMAX	1	111	111 111	1	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ
ZERO	Ø	ØØØ	ØØØ ØØØ	Ø	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ	ØØØ

NOTES

1. Overflow. The EXPONENT of the current FADDR or FMULR Result exceeds an APPARENT VALUE of 1023. (TRUE VALUE = 511). If the sign of the MANTISSA of the offending result is positive, the AP-120B hardware forces the maximum-positive FLOATING-POINT NUMBER (APMAX) into FA or FM, depending on which operation caused the error condition. If the MANTISSA of the offending result is negative, the maximum-negative FLOATING-POINT NUMBER (APNMAX) is forced onto FA or FM.

2. Underflow. When a current FADDR or FMULR operation produces an EXPONENT Result less than an APPARENT VALUE of 0 (TRUE VALUE = -512). The AP-120B hardware will force a FLOATING-POINT 0.0 (ZERO) into FA or FM, depending on which operation caused the error condition.

Note that FLOATING-POINT ZERO has an EXPO NENT with an apparent value of 0. (TRUE VALUE = -512) and a MANTISSA with a TRUE VALUE of 0.

CHAPTER 4

DETAILED DESCRIPTION OF THE FUNCTIONAL UNITS

4.1 S-PAD SUMMARY

4.1.1 General Description

The SCRATCH PAD (S-PAD) illustrated in Figure 4-1 is generally analogous to the arithmetic/logical and control units of most mini-computers. Operations with the S-PAD group of the AP-120 instruction set are primarily used to perform integer addressing and loop counting operations.

S-PAD is a 16-bit wide arithmetic/logical unit which contains sixteen 16-bit directly addressable registers (SP(0-15)). Operands for S-PAD operations are termed S-PAD SOURCE REGISTER (SP(SPS)) and S-PAD DESTINATION REGISTER (SP(SPD)). The particular S-PAD registers selected as operands for a given operation are determined by the values in the SPS and SPD fields of the current instruction word.

The result of a given S-PAD operation, termed the S-PAD FUNCTION (SPFN), becomes available for use during the current instruction cycle and may be applied to the following elements:

* MAIN DATA MEMORY ADDRESS REGISTER (MA)
* TABLE MEMORY ADDRESS REGISTER (TMA)
* DATA PAD ADDRESS REGISTER (DPA)
* FLOATING ADDER (FADDR) A1 Input Register
* DEVICE ADDRESS REGISTER (DA), and/or
* onto the DATA PAD BUS (DB)

Three condition bits of the APSTATUS Register are set or cleared depending on the state of the particular SPFN result. These bits (C, N, Z) may be tested (one cycle after the appropriate SPFN becomes enabled) by appropriate operations within the BRANCH group and/or SPEC group (STEST field), of the AP-120B Instruction Set. These bits remain latched if no S-PAD arithmetic operation is specified.

The S-PAD integer ALU functions include:

Function	Effect
a. Move	S→ D S-Source register
b. Logical complement	S→ D D-Destination register
c. Clear	0→D
d. Increment	S+1→ D
e. Decrement	S−1→ D
f. Add	D+S→ D
g. Subtract	D−S→D
h. Logical AND	D AND S→ D
i. Logical OR	D OR S→ D
j. Logical Equivalence	D EQV S→ D

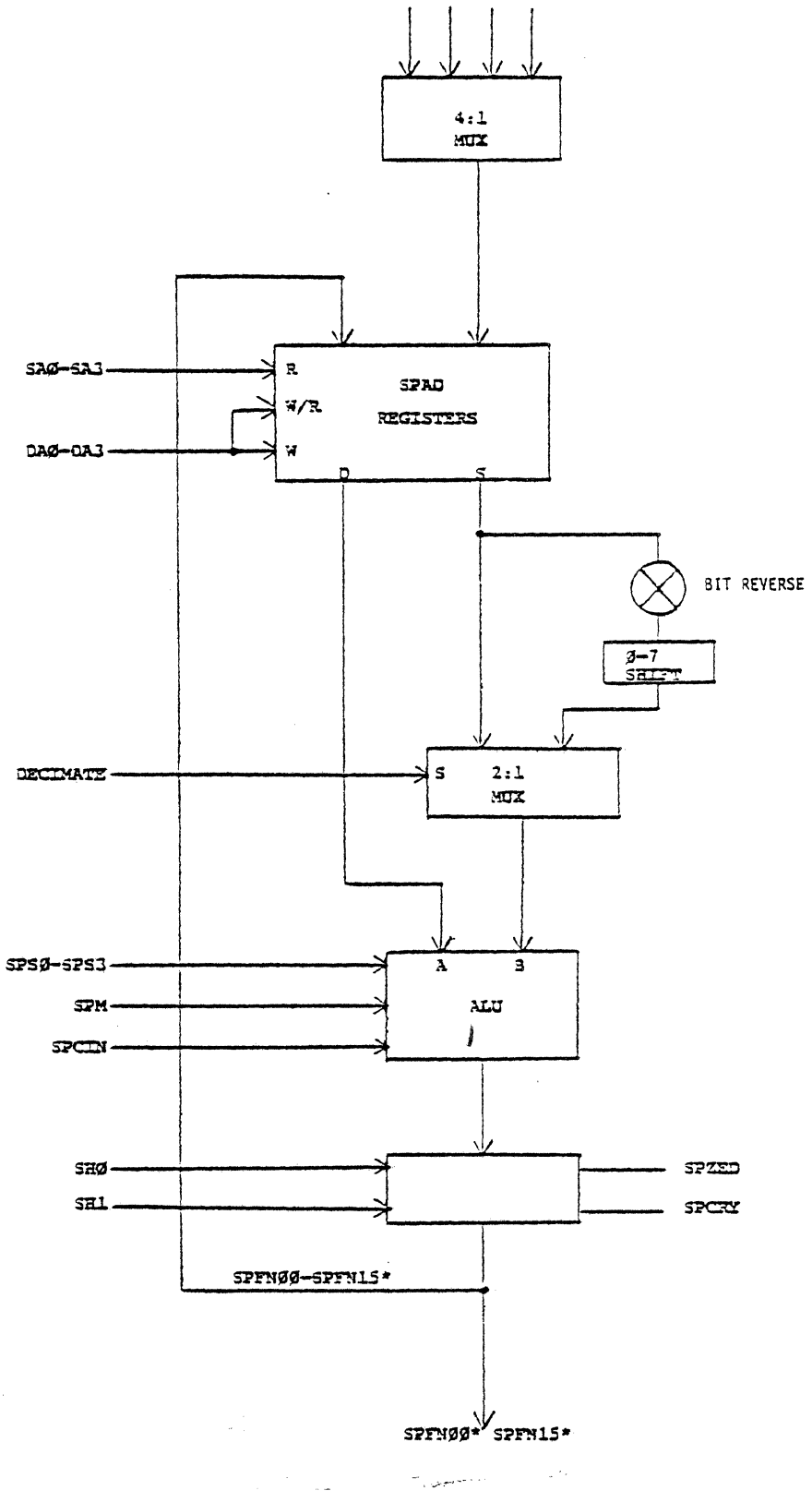


Figure 4-1 S-PAD Block Diagram

4.1.2 S-Pad Operations

The S-PAD may perform either single-operand or double-operand arithmetic/logical operations. The operations available within the SOP field are double-operand while the operations available within the SOP1 field are single-operand.

4.1.2.1 Single-Operand Operations

S-PAD single-operand operations use the currently specified SP(SPD) as the sole operand. (See SOP1)

SINGLE-OPERAND INSTRUCTIONS AND ALLOWABLE S-PAD MODIFIERS

OPERATION	L,R RR SHIFT	BIT REVERSE	S-PAD NO-LOAD (#)	REGISTER(S) USED
INC	YES	NO	YES	SP(SPD)
DEC	YES	NO	YES	SP(SPD)
COM	YES	NO	YES	SP(SPD)
CLR	YES	NO	YES	SP(SPD)

4.1.2.2 Double-Operand Operations

S-PAD double-operand operations use both currently specified SP(SPS) AND SP(SPD) as operands for a given operation. (See SOP)

DOUBLE-OPERAND INSTRUCTIONS AND ALLOWABLE S-PAD MODIFIERS

OPERATION	SHIFT	BIT (&) REVERSE	S-PAD NO LOAD (#)	REGISTER(S) USED
MOV	YES	YES	YES	SP(SPS),SP(SPD)
ADD	YES	YES	YES	SP(SPS),SP(SPD)
SUB	YES	YES	YES	SP(SPS),SP(SPD)
AND	YES	YES	YES	SP(SPS),SP(SPD)
OR	YES	YES	YES	SP(SPS),SP(SPD)
EQV	YES	YES	YES	SP(SPS),SP(SPD)

The result of either single-operand or double-operand operations - SPFN-is normally stored back into the SP(SPD) unless an S-PAD NO-LOAD is specified.

4.1.3 S-Pad Source and Destination Registers (SP(SPS)) (SP(SPD))

As stated before, up to two of the 16 S-PAD Registers may be accessed per instruction. The particular SP(SPS) and/or SP(SPD) to be used in a given operation is specified by the respective SPS and SPD fields in the instruction word.

SP registers may be labeled by name, using the following assembler pseudo-operation:

```
N $EQU 0        Meaning: Assign S-PAD Register "0" the name "N"*
```

All SP names must be declared in this manner before including them in an instruction. Additionally, it is permissible to specify more than one name for a given SP and it may be useful to do so when using the same SP to perform separate tasks in a program.

* NOTE

Don't confuse the designated number of the SP with its contents. For example: N $EQU 5 specifies that S-P 5 will be labeled as "N". But, the contents of "N" would not necessarily be equal to the value of five.

4.1.4 S-Pad Function (SPFN)

The result of a given S-PAD operation is termed the S-PAD FUNCTION (SPFN). The SPFN is normally loaded "back" into the currently designated SP(SPD) . SPFN is available during the current instruction cycle as an input to any of the three MEMORY ADDRESS REGISTERS: MEMORY ADDRESS REGISTER (MA), TABLE MEMORY ADDRESS REGISTER (TMA), DATA PAD ADDRESS (DPA), and also may be enabled onto the DATA PAD BUS (DB), which may input to the I/O Device Address register (DA).

```
Example: ADD 5,6; DB=SPFN
         (Result of ADD 5,6 is placed upon DB)>
```

4.1.5 S-Pad Modifiers "#", "Sh", "&"

Three optional modifiers may be used to alter the effect of an S-PAD operation:

S-PAD MODIFIERS

FIELD	OPERATOR	EFFECT
"B"	"&"	BIT REVERSE SP(SPS) before using as an S-PAD operand for double-operand operations. (See 4.1.7, BIT REVERSE)
"SH"	"L","R", or "RR"	SHIFT result of S-PAD operation. The shifted result becomes available as SPFN.
		OPERATOR MEANING EXAMPLE L Left shift once ADDL R Right shift once ADDR RR Right shift twice ADDRR
COND	"#"	S-PAD NO-LOAD inhibit the normal "back" storing of SPFN into the currently specified SP(SPD). S-PAD NO-LOAD disables the remaining operations in the BRANCH GROUP for the current instruction cycle. (See BRANCH)
ADDL# &5,and 6		Add a bit-reversed SP(SPS) to SP(SPD), shift the result left once, then inhibit "back loading" SP(SPD).

NOTE

Loading of Condition Bits (C, N, Z) is not inhibited by the "#".

4.1.6 S-Pad Associated Test and Branch Operations

Three Condition Bits (C, N, Z) are set or cleared in APSTATUS, depending on the condition of the current SPFN. These bits become valid and may be tested (and branches made accordingly), as of the next instruction cycle.

S-PAD-RELATED BITS IN APSTATUS REGISTER

Z (bit 5) Set to "1" one cycle after the current SPFN=0.

N (bit 6) Set to "1" one cycle after the current SPFN<0.

C (bit 7) S-PAD Carry Bit:

- *If an S-PAD shift was specified in forming SPFN, then C reflects the last bit shifted off the SP as a result of ANY S-PAD shift operation. (L, R, or RR).
- *If an S-PAD shift was not specified, then C reflects the state of the S-PAD Carry Bit as set by the last S-PAD operation.

Note that S-PAD-Related APSTATUS Bits set by the current SPFN become valid and may be tested by a conditional branch as of the NEXT instruction. S-PAD, NOP and SPEC group operations DO NOT affect the state of condition bits in APSTATUS.

S-PAD RELATED BRANCH OPERATIONS

INSTRUCTION-WORD GROUP	SUB-GROUP	INSTRUCTION MNEMONIC	BRANCH CONDITION	RELATED BIT(S) IN APSTATUS
BRANCH	COND	BEQ	SPFN=0	Z
		BNE	SPFN=0	Z
		BGE	SPFN≥0	N
		BGT	SPFN>0	Z,N
SPEC	STEST	BLT	SPFN<0	N
		BNC	S-PAD CARRY=1	C
		BZC	S-PAD CARRY=0	C

For details on BRANCH TARGET ADDRESS formulation, see SPEC summary, Section 4.2. Some S-PAD timing examples are listed in Table 4-1.

Table 4-1 S-PAD TIMING EXAMPLES

INSTRUCTION	SP(2)	SP(3)	DB (12-27)	SPFN	APSTATUS BITS N	Z	C	BRANCH CONDITIONS BLT	BEQ	BNC
1. -	3	-	-	-	-	-	-	-	-	-
2. DEC 2	3	-	-	2	-	-	-	-	-	-
3. DEC 2	2	-	-	1	0	0	1	F	F	T
4. DEC 2	1	-	-	0	0	0	1	F	F	T
5. NOP	0	-	-	-1	0	1	1	F	T	T
6. NOP	0	-	-	-1	0	1	1	F	T	T
.										
.										
.										
1. -	5	7	-	-	-	-	-	-	-	-
2. SUB 2,3	5	7	-	2	-	-	-	-	-	-
3. NOP	5	2	-	-3	0	0	1	F	F	T
4. NOP	5	2	-	-3	0	0	1	F	F	T
.										
.										
.										
1. -	5	7	-	-	-	-	-	-	-	-
2. SUB# 2,3	5	7	-	2	-	-	-	-	-	-
3. NOP	5	7	-	2	0	0	1	F	F	T
4. NOP	5	7	-	2	0	0	1	F	F	T
.										
.										
.										
1. -	3	-	-	-	-	-	-	-	-	-
2. LDSPI 2; DB=-1	3	-	-1	3	-	-	-	-	-	-
3. NOP	-1	-	-	-1	0	0	1	F	F	T
4. NOP	-1	-	-	-1	0	0	1	F	F	T

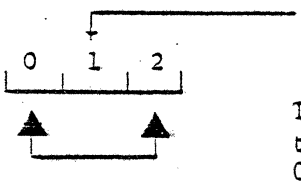
- means don't know or don't care
F = Branch Condition False
T = Branch Condition True

4.1.7 Bit Reverse

4.1.7.1 General Description

BIT-REVERSE is the process in which the contents of an address-subscript for an array base-address are reflexively transformed. In other words, the contents of the bits of an address-subscript undergo a bit-for-bit exchange about the center or reflex point of the word. Example:

Assume a 3 bit word:

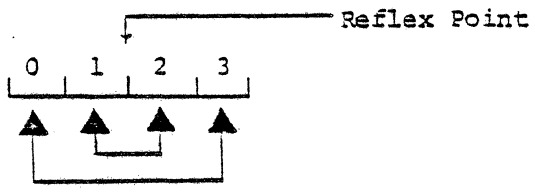


The contents of bit 02 are exchanged with the contents of bit 0. The center bit, bit 01, is unchanged.

The word would be transformed as follows:

CONTENTS BEFORE BIT-REVERSING	CONTENTS AFTER BIT-REVERSING
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

If the address-subscript is an even numbered word, the contents would be transformed in the following manner:



4.1.7.2 Bit-Reverse General Application

In the course of FAST FOURIER TRANSFORM operations it is sometimes desirable to access a data array in bit-reversed order. For an eight-point data array A:

NORMAL ORDER	BIT-REVERSED ORDER	SUBSCRIPT IN NORMAL ORDER	SUBSCRIPT IN BIT-REVERSED ORDER
A(0)	A(0)	0	0
A(1)	A(4)	1	4
A(2)	A(2)	2	2
A(3)	A(6)	3	6
A(4)	A(1)	4	1
A(5)	A(5)	5	5
A(6)	A(3)	6	3
A(7)	A(7)	7	7

The array is accessed in normal order by successively incrementing the subscript. Similarly, the array is accessed in bit-reversed order by incrementing the subscript. The bit-reversed value of the subscript, however, is then used each time to access the array in bit-reversed order.

4.1.7.3 AP-120B Bit-Reverse Application

When the S-PAD BIT-REVERSE (&) is specified for an S-PAD operation, the following process occurs:

1) A 15-bit wide BIT-REVERSE (&) is performed on bits 0-14 of the SP(SPS). Bit 15 is set to 0.

2) The bit-reversed result is right shifted using the APSTATUS BIT-REVERSE FIELD (APSTATUS (Bits 13-15)) as the shift count. Bit 15 is again set to 0.

3) The bit-reversed, shifted word is then used as the source-operand for the particular S-PAD operation specified. Note that the S-PAD operation is performed AFTER the bit-reverse operation.

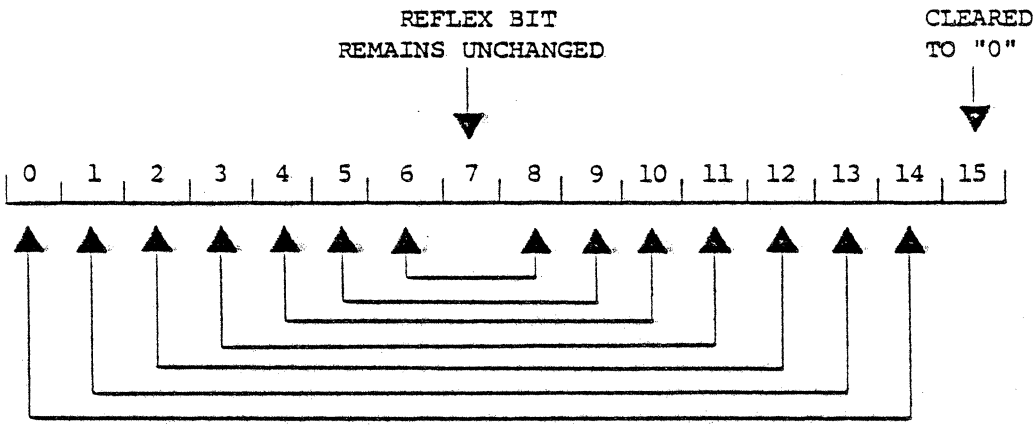
TO ELABORATE:

* First-

The contents of SP(SPS) are bit-reversed in the following manner: The contents of bit "0" are "swapped" with the contents of bit "14", the contents of bit "01" are "swapped" with the contents of bit"13", and so on. Bit "07" remains unchanged. Since the complex numbers of a data-array consist of a real part and an imaginary part, each address of an

array actually consists of two successive memory locations, the first containing the imaginary part. Therefore, the index of the real part of the number is always "even"; bit "15" of the index (SP(SPS)) is always cleared to "0".

Example:



* Second-

The result from the 15-bit wide BIT-REVERSE operation is shifted right so as to fit into the actual width of the word being processed. The shift involves Bits 0-14, with zeros filled in on the left and into Bit 15.

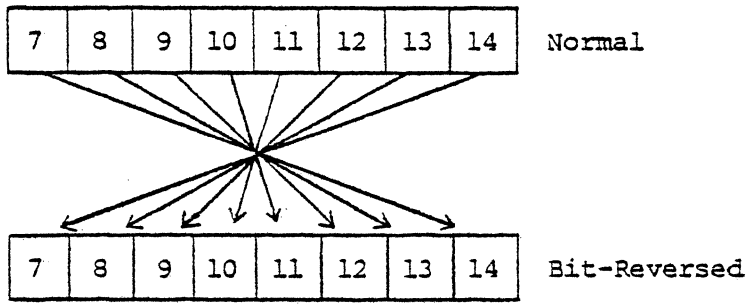
The number of shifts that will be done during the alignment shift depends upon the value contained in the Bit-Reverse field (Bits 13-15). The programmer must place the correct value into the BIT-REVERSE field before using a BIT-REVERSE operation.

The value that must be pre-programmed into the APSTATUS REGISTER BIT-REVERSE FIELD depends on the size of the complex data array, and is listed below accordingly. The Shift Count = 15-n, where n = the power of two.

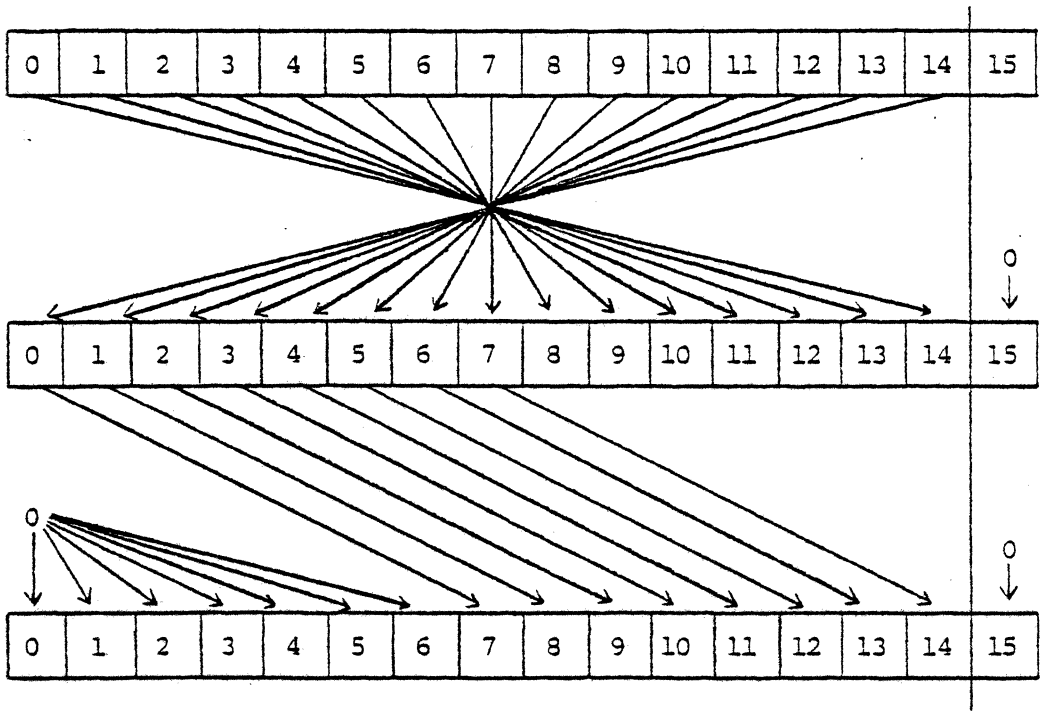
Array Size	Number of Bits in the Subscript word	Shift Count to be placed in Bit Reverse Field of APSTATUS REGISTER
32,768	15	0
16,384	14	1
8,192	13	2
4,096	12	3
2,048	11	4
1,024	10	5
512	9	6
256	8	7

Example: To bit-reverse an 8-bit address word

1) Desired bit-reverse:



2) As accomplished by the bit-reverse operator (&) with the Bit-Reversed Shift count set to 7:



* Third-

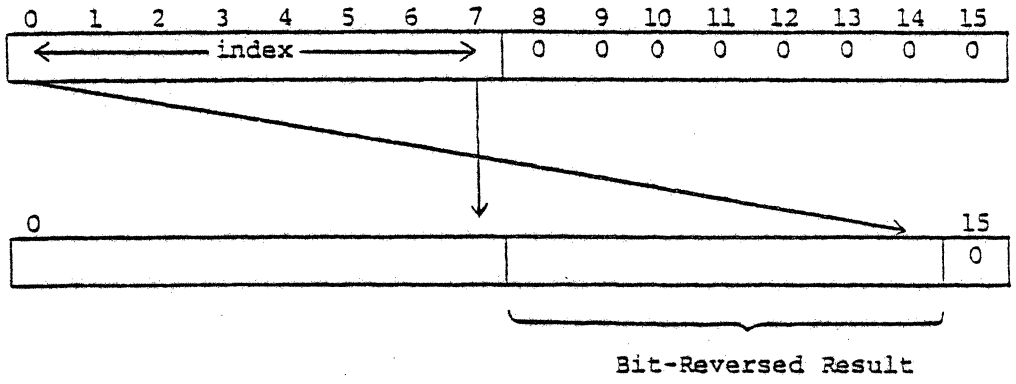
The bit-reversed, shifted SP(SPS) is now ready to be used as the source operand in an S-PAD operation. Most commonly, BIT-REVERSE is used in the following manner:

ADD# &subs, base;SETMA	This operation bit-reverses an array subscript and adds it to the base-address of an array. This sum is used to initiate a memory fetch. The NO-LOAD(#) is stipulated so as to prevent the base-address (contained in SP(SPD)) from being overwritten by the SPFN produced from the ADD operation.

BIT-REVERSE (&) clears bit 15 of a subscript-word, and hence is intended to be used to access complex arrays. A subscript to be bit-reversed must always be even, and point to the real part of a complex pair. The bit-reverse shift count must agree with the word width of the subscript, which in turn must agree with the size of the complex array being accessed. That is, an eight-bit subscript, which will address an array of 2(8)=256 complex pairs, needs a shift count of 15-8=7.

Since the shift count is limited to a maximum of seven places, 256 points is the smallest complex array that can be directly accessed using the BIT-REVERSE operator.

There is a technique, however, that permits use of the BIT- REVERSE operator for arrays smaller than 256 complex points. This technique is based on the fact that a right shift after bit-reversal is equivalent to a left shift before bit-reversal. Thus, if the index for a small array is placed in the left byte of the 16-bits of an S-PAD Register (a left shift by seven places) and if the increment for that index is correspondingly left shifted so that necessary index alteration operations will take place consistently, then the BIT-REVERSE shifter can be used with a shift count of 8-n where n=the power of two.



BIT-REVERSE operator for small arrays

($N \leq 256$ $n \leq 8$)

4.1.8 General Programming Rules (S-Pad Group)

* Only one S-PAD operation may be specified per instruction cycle.

* If a single-operand instruction is specified, then SP(SPD) is the only operand used. The BIT-REVERSE field is disabled.

* If a double-operand instruction is specified, then both SP(SPS) and SP(SPD) are used. SP(SPS) may be "bit-reversed" before use.

* SUBtract will subtract the first argument (SP(SPS)) specified from the last argument specified (SP(SPD)). Example:

  SUB 6,5 will subtract the contents of SP 6 from the contents of SP 5.

* The preliminary result of an S-PAD operation may be shifted before becoming SPFN. The contents of SPFN are stored back into the currently designated SP(SPD) unless an S-PAD NO-LOAD (#) is specified. SPFN reflects the final result of the current S-PAD operation including any S-PAD modification operations stipulated.

* The condition of SPFN sets or clears the C, N, and Z bits of the APSTATUS Register - effective as of the NEXT instruction cycle, accordingly:

  * Branches which test the condition of SPFN may be made one instruction following the appropriate S-PAD operation.

* IMPORTANT NOTE: Although the contents of SP(SPD) will not be changed during a "NOP" instruction following an S-PAD operation, SPFN will change - reflecting the new contents of SP(SPD).

  Although the S-Pad related bits in APSTATUS will not be altered by an S-PAD NOP, the programmer should exercise care when executing an RSPFN op-code, (See RDREG, I/O), since the contents of SPFN may be altered from their state during the last S-PAD operation.

4.1.9 S-Pad Carry Bit Conditions

During any of the operations listed below (except OR) the S-PAD CARRY BIT will be set to "1" if the corresponding equation for the current operation specified is true.

Note that when using S-PAD test and branch operations, Bit C of the APSTATUS REGISTER will reflect the state of the S-PAD CARRY BIT for the last preceding S-PAD operation unless a shift modifier was specified for that operation. If an S-PAD shift was specified, "C" will reflect the state of the last bit shifted off the end.

S-PAD CARRY-BIT RELATIONS

If true; then S-PAD CARRY BIT is set to "1"

Operation equations

SUB	$(SP(SPD)) + (\overline{SP(SPS)}) + 1 \geq 2(16)$
INC	$[(SP(SPD)) + 1] \geq 2(16)$
DEC	$(SP(SPD)) + 177777 \geq 2(16)$
COM	$(SP(SPD)) + 177777 \geq 2(16)$
CLR	$(SP(SPD) + .(SP(SPD)) \geq 2(16)$
ADD	$(SP(SPD)) + (SP(SPS)) \geq 2(16)$
OR	S-PAD CARRY BIT is set to "0"
EQV	$(SP(SPD)) + (\overline{SP(SPS)}) > 2(16)$
AND	$[(SP(SPD)) \text{ AND } (\overline{SP(SPS)})] + (SP(SPD)) > 2(16)$
MOV	$[(SP(SPD)) \text{ AND } (\overline{SP(SPS)})] + [(SP(SPD)) \text{ OR } (SP(SPS))] > 2(1[)$

4.1.10 Programming Example

Load Data Pad X with an array "A," with N elements starting at Main Data Memory location 3721x. "CTR" is in S-Pad register which will be used as a counter.

```
1.            CLR# CTR; SETDPA      "Set DPA to 0
2.            LDMA; DB=3721         "Fetch the first
                                    "element
3.            LDSPI CTR; DB=N       "Initialize "CTR"
                                    "to N
4. LOOP:      INCMA; DEC CTR        "Fetch next element,
                                    "Ax+1
5.            DPX<MD;               Store Ax into DPXx,
              INCDPA; BNE LOOP      "advance DPA and test
                                    "counter.
```

Below is a chart of the above loop, for N=3 elements.

Inst. #	Memory MA	MI	Data DPA	Pad 0	1	2	S-Pad "CTR"	Test
1.	---	---	0	--	--	--	---	
2.	3721	---	0	--	--	--	---	
3.	---	---	0	--	--	--	3	
4.	3722	---	0	--	--	--	3	
5.	---	A_0	0	A_0	--	--	2	true
4.	3723	---	1	A_0	--	--	2	
5.	---	A_1	1	A_0	A_1	--	1	true
4.	---	---	2	A_0	A_1	--	1	
5.	---	A_2	2	A_0	A_1	A_2	0	false

A generalization on the above example is to fetch array "A" from every Kth memory location.

The increment is stored in S-Pad register "STEP," and the array pointer is stored in "PTR:"

```
1.        LDSPI STEP; DB=K          "Initialize
                                    ""STEP" to K
2.        CLR# CTR; SET DPA         "Set DPA to 0
3.        LDMA; DB=BASE             "Fetch the first
                                    "element, Ax
4.        LDSPI CTR; DB=N           "Initialize "CTR"
                                    "to N
5. LOOP:  ADD STEP, PTR; SETMA      "Advance memory
          BEQ DONE                  "pointer.  Fetch
                                    "next element,
                                    "Ax+x.  Test
                                    "counter and jump
                                    "out if done.
6.        DPX<MD; INCDPA            "Store Ax into
          DEC CTR; BR LOOP          "DPXx, advance
                                    "DPA Decrement
                                    ""CTR" and jump
                                    "back to LOOP.
```

4.2 SPECIAL OPERATIONS GROUP SUMMARY (SPEC)

The op-codes available within the SPEC group fall within the following functional groups: Branch and Test Instructions, Jumps, Data Transfer Instructions and Program Control instructions related to PROGRAM SOURCE ADDRESS REGISTER (PSA) and SUBROUTINE RETURN ADDRESS STACK (SRS) modification.

Discussion of the components involved in the SPECIAL OPERATIONS GROUP (SPEC) is presented in the following manner:

1) Test, Branch and Jump logic related to op-codes within this group,

including:

* Appropriate Control bits of the APSTATUS REGISTER

* AP-120B GENERAL FLAGS

* Types of branches and Branch Timing Implications

2) Data Transfer Operations, including:

* The PROGRAM SOURCE MEMORY (PS) -- theory of operation, addressing, and types of formats available.

* The VIRTUAL FRONT PANEL (PANEL) and PANEL registers related to SPEC GROUP transfer operations.

3) PROGRAM ADDRESS and SUBROUTINE ADDRESS modification (JUMPS)

* Program Jumps

* Jumps to Subroutines (JSRS)

4.2.1 Branch Operations

Branches available within the SPECIAL OPERATIONS GROUP (SPEC) may be used in tandem with other branch instructions available within the CONDITIONAL BRANCH GROUP (see BRANCH). Both groups of branches use the DISP field (instruction word Bits 27-31) to calculate the BRANCH TARGET ADDRESS. If branch op-codes are used from both groups in the same instruction word, the branch operations are OR'ed. Example:

```
BFLT LOOP; BEQ LOOP
```

Meaning: (If either FA is < 0, or SPFN is = 0, then the program will branch to the location indicated as LOOP).

4.2.1.1 Test Conditions

One of the following functions is tested to determine the status of a given branch operation decision.

- *Selected bits of the AP-120B INTERNAL STATUS REGISTER (APSTATUS)
- *AP-120B GENERAL FLAGS
- *DATA-PAD BUS state

* SPEC-Related Bits in APSTATUS Register

Bit Name	Condition	Related SPEC Op-Code(s)
FN (bit 4)	Set to "1" when Floating Adder Result (FA) is negative; reset to "0" when FA is zero or positive.	BFLT
N (bit 6)	Set to "1" when S=PAD FUNCTION (SPFN) is negative; reset to "0" when SPFN is zero or positive.	BLT
C (bit 7)	Set to "1" when an S-PAD operation produces a carry either by arithmetic generation or as a result of an S-PAD SHIFT operation; reset to "0" when no S-PAD carry is produced.	BNC,BZC
IFFT (bit 11)	Set to "1" when AP-120B is processing in the IFFT mode. (Set or cleared via programmed instructions).	BIFN,BIFZ

* AP-120B General Flags

There are four flags (0,1,2,3) available for general use within the AP-120B. These flags may be set to "1" or cleared to "0" by software instructions (See FLAG, I/O). A list of the general FLAGS and their respective branch operations is given below:

AP-120B GENERAL FLAGS

AP FLAG	Branch Contingency	Related SPEC Op-code
FL0	Set to "1"	BFL0
FL1	Set to "1"	BFL1
FL2	Set to "1"	BFL2
FL3	Set to "1"	BFL3

Note that a minimum of one cycle must intervene between a flag modification instruction (e.g., SFL0) and the related test instruction.

* Data-Pad Bus (DB) Test

Additionally, the data enabled onto the DATA-PAD BUS (DB) may be tested as to state and branches made accordingly:

STATE (DPBS)	Related SPEC Branch Op-Code
Negative	BDBN
Zero or Positive and Unnormalized	BDBZ

WARNING

These branches will not work correctly following an instruction from the PS field (e.g., RPSF). They will branch as if DB=0.

4.2.1.2 Branch Target Address Formulation

A BRANCH TARGET ADDRESS is formed by summing the current PSA with the biased DISP field in the following manner:

(PSA) + (DISP - 20(octal)) PSA	Meaning: The current contents of the PROGRAM SOURCE ADDRESS REGISTER (PSA) are added to the contents of the biased 5-bit DISplacement field (instruction word bits 27-32).

The BRANCH TARGET ADDRESS thus formed will be the next program location to be executed if the current branch condition is satisfied.

Biased Displacement Field

The DISP field contains a 5-bit BIASED integer. BIAS is the leftmost bit of the field. When Bias = 1, DISP contains a positive value in the remaining four bits of the field. When BIAS = 0, DISP contains a negative value in the remaining bits, in unsigned twos-complement form. The TRUE-VALUE of DISP is always 20(octal) less than its APPARENT-VALUE.

The range of BRANCH TARGET Addresses using DISP is from -20(octal) to +17 (octal) locations relative to the current PROGRAM SOURCE ADDRESS (PSA).

4.2.1.3 Branch Timing

All of the conditions tested within this group of branch instructions relate to the state of the appropriate device (e.g., DB, SPFN, FA, etc.) as of the previous instruction cycle.

For example, a BFLT op-code tests the condition of the FLOATING ADDER OUTPUT (FA) enabled onto the FA Bus during the preceding instruction. Similarly, a BLT op-code tests the state of the S-PAD FUNCTION (SPFN) enabled during the preceding instruction cycle.

All the Status Bits related to branch operations are latched into the APSTATUS register one cycle after the condition that generated them becomes enabled. A Status Bit will remain set or cleared from that time until one cycle after the next operation produces a condition that will change the Status Bit.

If the condition to be tested was set via an LDAPS instruction, then a one cycle delay must be observed before testing it. Note that the LDAPS wins out over the dynamic FA, FM or S-PAD condition in determining the state of APSTATUS.

The following examples provide explicit illustrations of the timing relationships for the different types of branch conditions.

Example: FLOATING ADDER RESULT (FA) TESTED

			Result Available	Appropriate Status Bit	
t1	FADD	TM,MD	-	-	Note that the result of the
t2	FADD	DPX(-1),DPY(2)	-	-	first FADD (FADD(t1)) be-
t3	NOP		FA(t1)		comes available as FA at
t4	BFLT	LOOP	FA(t1)	FN(t1)	t3. The earliest that one
t5	NOP		FA(t1)	FN(t1)	could test it then is at
t6	NOP		FA(t1)	FN(t1)	t4. Note also that FN in
t7	FADD		FA(t1)	FN(t1)	APSTATUS will continue to
t8	NOP		FA(t2)	FN(t1)	reflect the state of this
t9	BFLT	LOOP1	FA(t2)	FN(t2)	particular FADD result until two cycles after the next FADD operation. The result (FA(t2)) of FADD(t2) becomes available at t8 and the earliest one could test it is at t9, as shown.

The timing of the preceding example is applicable to all FA and FM related branches (APSTATUS bits FN, FZ, OVF and UNF).

Example: S-PAD FUNCTION (SPFN) TESTED

		Result Available	Appropriate Status Bit	
t1	ADD3,4	SPFN(t1)	——	Note that the result of the first S-PAD OPERATION (ADD 3,4) becomes available during t1. The earliest one could test this result, then, is at t2. But to illustrate the point that the BRANCH CONDITION BITS latch, the example, does not specify the appropriate test-branch op-code until t5. Note also that more than one branch may be made on the same SPFN, even at a different time, as long as the SPFN state for the operation inquiry is still latched in APSTATUS (i.e., BGE (t6). The appropriate APSTATUS bit condition produced from the ADD OPERAtion at t1 is not altered until one cycle after the next S-PAD OPERATION (e.g., t8). Note also that WRTEXP is effectively an S-PAD NOOP with respect to the condition bits of the APSTATUS REGISTER. The BLT at t8 tests the result of the ADD 4, 5 instruction at t7.
t2	NOP	*SPFN(t1)	N(t1)	
t3	WRTEXP	*SPFN(t1)	N(t1)	
t4	NOP	*SPFN(t1)	N(t1)	
t5	BLT LOOP1	*SPFN(t1)	N(t1)	
t6	BGE LOOP	*SPFN(t1)	N(t1)	
t7	ADD 4,5	SPFN(t7)	N(t1)	
t8	BLT LOOP 3	*SPFN(t7)	N(t7)	

NOTE

*The value in SPFN will reflect the updated contents of SP(SPD) during times t2 to t6 and t8 to t9.

Example: DATA PAD BUS (DB) TESTED

		Result Available on DB	
t1	DB=MD	DB(t1)	Note that DB is not latched as in the above example and is available for testing for only one cycle following the appropriate enabling operation. In order to perform two sequential branch operations on the same DB, one must "hold" it by a duplicate operation (e.g., the branch at t5 will test the DB (t4). But it is necessary to re-enable DB at t5 in order to perform the branch at t6).
t2	BDBN LOOP1	ZERO	
t3	NOP	ZERO	
t4	DB=DPX(2)	DB(t4)	
t5	BDBN LOOP1; DB=DPX(2)	DB(t4)	
t6	BDBZ LOOP2	ZERO	

Example: APSTATUS TEST following LDAPS

t1	ADD1, 2; LDAPS; DB=value	(See Note)
t2	BLT LOOP1	"Test result of ADD 1, 2 at t1
t2	BEQ LOOP2	"Test result of LDAPS at t1
t3	FADD	
t4	FADD	
t5	LDAPS; DB=value	
t6	BFLT LOOP3	"Test result of FADD at t4
t7	BGE LOOP4	"Test result of LDAPS at t5

NOTE

The cycle following the LDAPS is dominated by the most recent dynamic conditions, e.g., ADD 1,2 at t1 determines BRANCH status during t2 and FA result during t5 controls branch during t6. The cycle after that, the LDAPS dominates---thus the branches at t3 and t7 test the result of the preceding LDAPS instruction.

Example: AP-120B FLAG TESTED

```
t1  CFL0
t2  NOP
t3  SFL0
t4  BFL0 LOOP1
t5  BFL0 LOOP1; CFL0
```

Note that a set/clear flag operation is effective, with respect to testing, one cycle after being performed. Accordingly, the BFL0 at t4 will find the flag clear and will not result in a program branch. The SFL0 at t3 will be effective as of t4 and the BFL0 at t5 will produce a program branch.

4.2.2 Data Transfer Operations

The op-codes within the PSEVEN, PSODD, and PS fields of SPEC group deal with program-word transfer operations between PROGRAM SOURCE MEMORY (PS) and the LITES or SWITCH registers, (LITES, SWR) in the AP VIRTUAL FRONT PANEL (PANEL). The transfers are via the PANEL-BUS (PNLBS) and the DB. The PROGRAM SOURCE WORD may be transferred in QUARTER-WORD, HALF-WORD, or FLOATING-POINT LITERAL format. All PS read and load operations are two-cycle instructions. They behave like one-cycle instructions in that they execute on the first cycle, but they require a second cycle to fetch the next instruction. Therefore, they behave like two cycles, as far as MD timing is concerned.

The available PS formats are presented in Figure 4-2.

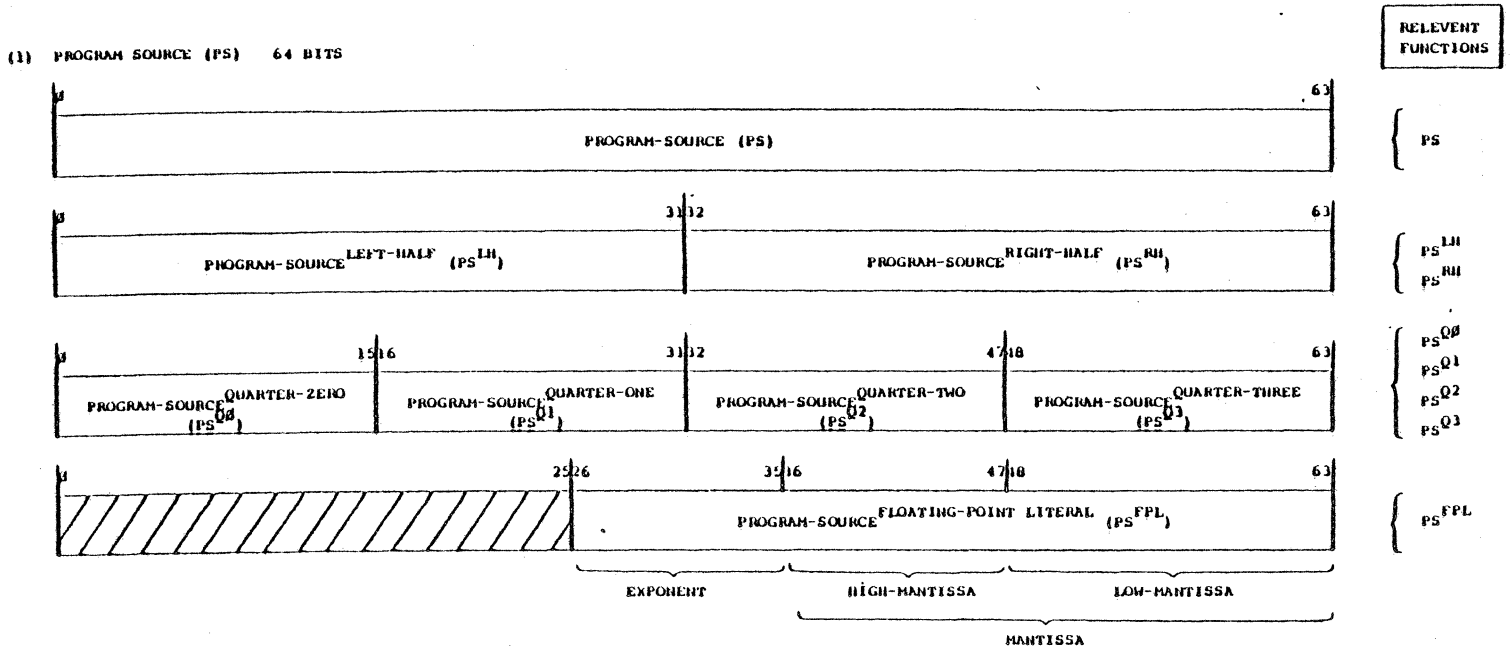


Figure 4-2 PS Formats

4.2.2.1 Program Source Memory (PS)

The Program Source Memory file (PS) shown in Figure 4-3 contains the 64 bit instruction words by which programs in the AP-120B are executed. The location of the Program Source Word to be executed is pointed to by the PROGRAM SOURCE ADDRESS REGISTER (PSA) and the instruction contained in that location is decoded, and the appropriate control-functions are generated, by the CONTROL BUFFER (CB).

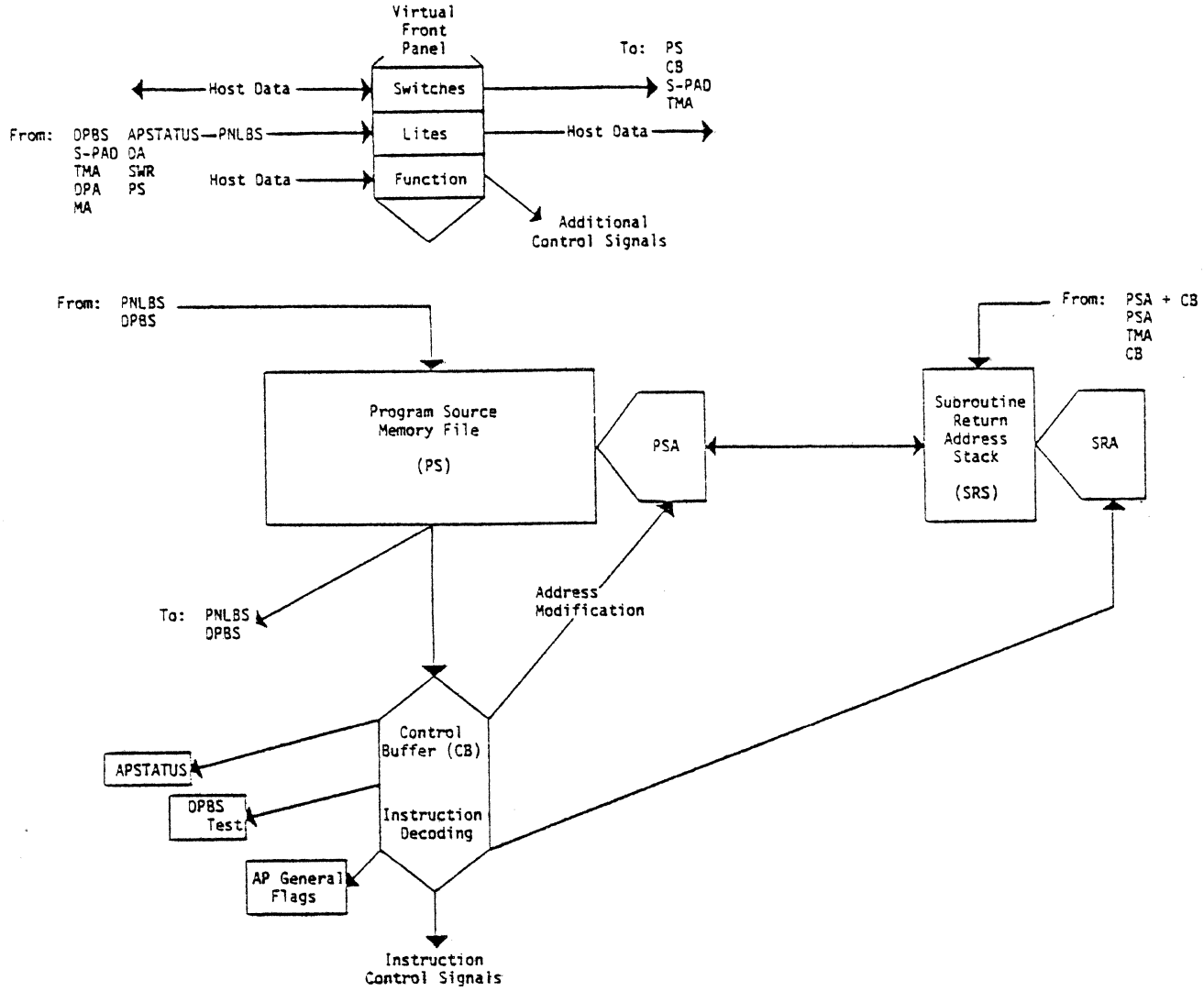


Figure 4-3 Program Source Memory File

Addressing of the PROGRAM SOURCE MEMORY is accomplished by reference to the 12 bit PROGRAM SOURCE ADDRESS REGISTER (PSA). PSA may be sequentially incremented, altered by either branch or jump instruction, or modified by the several SUB-ROUTINE ADDRESS RETURN STACK (SRS) -related instructions.

4.2.2.2 Program Source Transfer Operation Addressing

When transferring data to or from PROGRAM SOURCE MEMORY (PS), addressing of the PS word is accomplished by either of the following methods:

1. ABSOLUTE - The address of the PROGRAM SOURCE WORD to be read/written is ABSOLUTE from either:
   * The 12 bit address contained in the VALUE field of the current instruction word,
   * the least significant 12 bits of the TABLE MEMORY ADDRESS (TMA), or
   * the 12-bit address currently enabled onto the PANEL BUS (PNLBS).

2. RELATIVE - The address of the PROGRAM SOURCE WORD to be transferred is RELATIVE to the current PSA. The address is formed by adding the current contents of the PROGRAM SOURCE ADDRESS REGISTER (PSA) with the 12-bit address contained in the VALUE field of the current instruction word.

4.2.2.3 SPEC-related Virtual Front Panel (PANEL) registers

The LITES REGISTER (LITES) and SWITCH REGISTER (SWR) of the VIRTUAL FRONT PANEL (PANEL) are involved in many SPEC group transfer op-codes. Both are 16-bit wide registers.

The LITES register is used as a destination register of the PNLBS in the appropriate SPEC group transfer operations. The host can read LITES but cannot load it. The AP-120B can load LITES, but cannot read it. The SWITCH REGISTER is used as a source register for the PNLBS. The host can load and read SWR. The AP-120B can only read SWR.

Data transfers between PS and the appropriate PANEL registers are via the PANEL BUS (PNLBS) and require two machine cycles to execute (See Note). (For a complete description of the VIRTUAL FRONT PANEL, see I/O SUMMARY).

NOTE

When executing these instructions, the programmer must not attempt to execute any other operation which uses the PNLBS for addressing or transferring or a PNLBS conflict will occur.

4.2.3 Program Source Address Modification

The op-codes contained in the SETPSA fields are used to modify the program location by means of forcing a specified value into the PROGRAM SOURCE ADDRESS REGISTER (PSA). The op-codes within the SETPSA field fall into two categories:

* Program Jumps
* Jumps to subroutines

4.2.3.1 Program Jumps

ABSOLUTE or RELATIVE PROGRAM JUMPS available in the SETPSA field replace the contents of the PROGRAM SOURCE ADDRESS REGISTER (PSA) with a specified value. The Jump may be ABSOLUTE, whereby the contents of PSA are replaced by the current contents of a specified register, or the address currently enabled upon a specified Data Bus; or the Jump may be RELATIVE - whereby the data from a specified source is algebraically added to the current contents of PSA.

4.2.3.2 Jumps to Subroutines

The AP-120B uses an address stack, termed the SUB-ROUTINE RETURN ADDRESS STACK (SRS), which is normally used to save the current contents of PSA plus "1" in the execution of a SUB-ROUTINE JUMP OPERATION (JSP). The address thus saved may be retrieved via a RETURN operation (see RETURN, BRANCH) at the completion of a given sub-routine operation thus enabling the program to return to the next program-source location following the JSR.

4.2.3.3 SRS Operation

The SRS is a LAST-IN - FIRST-OUT (LIFO) memory stack with a capacity for sixteen 12-bit addresses. Addressing control for the SRS is performed by an up/down counter termed the SUB-ROUTINE RETURN ADDRESS REGISTER (SRA). The address last stored into SRS during a sub-routine jump will be the address enabled into PSA upon a return operation. Note, however, that the address stored in this "last-in" position of SRS can be changed via use of selected operations in the SETEXIT field.

The SUB-ROUTINE RETURN STACK (SRS) can store up to a maximum of 16 address words at one time, allowing the programmer to specify a maximum of 16 nested sub-routines. Attempting to exceed the maximum number of nested sub-routines will set the SUB-ROUTINE RETURN ADDRESS OVERFLOW (SRAO) in the APSTATUS register, as will a RETURN operation that has not been preceded by a SUB-ROUTINE JUMP OPERATION.

An example of SUB-ROUTINE JUMP and RETURN operations is given below:

MAIN PROGRAM	SUB-ROUTINE 1	SUB-ROUTINE 2
N(0) ZZ	N(15) SUB1:ZZ	N(30) SUB2:ZZ
N(1) ZZ	N(16) ZZ	N(31) ZZ
N(2) ZZ	N(17) JSR SUB2	N(32) ZZ
N(3) JSR SUB1	N(18) ZZ	N(33) ZZ
N(4) HALT	N(19) RETURN	N(34) RETURN

ZZ= non-related operation

Note that the MAIN PROGRAM will jump to SUB-ROUTINE 1 (N(15)) upon executing the JSR instruction at N(3). During this jump, the SRA is incremented and N(4) (current PSA + 1) will be stored into the "last-in" position of the SUB-ROUTINE RETURN STACK (SRS).

The program will now execute in SUB-ROUTINE 1 until location N(17), at which time a jump will occur to SUB-ROUTINE 2 at location N(30). The SRA is again incremented, and the address N(18) (the now current PSA + 1) becomes the "last-in" address stored into SRS.

This program will now execute in SUB-ROUTINE 2 until location N(34), where a RETURN will be executed. At this time, the "last-in" address N(18) will be written into PSA and the program will jump back to that location. SRA is decremented so that now N(4) again becomes the "last-in" address contained in SRS.

The Program Proceeds: At N(19) a RETURN will again be executed, and the SRS will write the address N(4) into PSA. The program will halt at that location. SRA is decremented and the "last-in" address becomes the address written into SRS one time before address N(4) was intially written into SRS.

WARNING

The JSR instruction has a number of hardware timing problems not detected by APAL or APSIM.

NOTE

A JSR must not be followed immediately by a RETURN.

Example:

ILLEGAL

```
N(14)   JSR SUB2
N(15)   RETURN
```

Also, any two-cycle instruction immediately before a JSR is illegal.

Example:

ILLEGAL

```
N(13)   RPSF
N(14)   JSR SUB2
```

If the breakpoint of the front panel is used to stop the machine so that PSA is pointing to an instruction before a JSR, it is not possible to continue.

Example:

ILLEGAL

Breakpoint at N(4) followed by

```
N(5)  ZZ
N(6)  JSR SUB1
```

A halt followed by a JSR is also illegal.

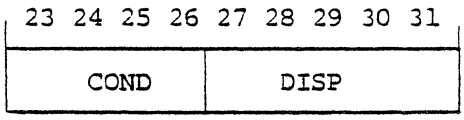
Example:

ILLEGAL

```
N(13)   HALT
N(14)   JSR SUB2
```

4.2.3.4 Conditional Branch Op-Code Group

Purpose: UNCONDITIONAL and CONDITIONAL BRANCHES to program locations within a range of +17(octal) to -20(octal) locations relative to the current PROGRAM SOURCE ADDRESS (PSA), the SUBROUTINE RETURN OPERATION, and the S-PAD NO LOAD function.



COND — CONDITIONAL BRANCH: Four-bit field whose value determines the specific operation to be performed within the BRANCH group.

DISP — DISPLACEMENT: Five-bit field whose value with the BIAS subtracted is added to the current PSA to form the BRANCH TARGET ADDRESS for the specified branch operations. (BIAS = 20(octal))

4.2.4 Branch Group Summary

The CONDITIONAL BRANCH op-code group consists of two fields: COND (instruction word bits 23-26) and DISP (instruction word bits 27-31). The value contained in the COND field selects the one operation of the BRANCH GROUP to be performed during the current instruction cycle.

The operations available within the COND field are:

1) S-PAD NO-LOAD (#),
2) UNCONDITIONAL BRANCH
3) CONDITIONAL BRANCHES and,
4) SUB-ROUTINE RETURN

When an UNCONDITIONAL BRANCH, is specified, or the branch condition of a CONDITIONAL BRANCH is satisfied, the program will branch to the PROGRAM SOURCE location obtained by adding the current contents of PROGRAM SOURCE ADDRESS REGISTER (PSA) with the value (BIAS removed) contained in the DISP field. The BRANCH TARGET ADDRESS thus obtained is limited to a range from -20(octal) to +17(octal) locations relative to the current PSA.

For a detailed discussion of the addressing, timing and programming examples for Branch type operations, refer to SPEC SUMMARY, Part 1 - Test, Branch and Jump Operations. All sections of Part 1 apply equally to Branch operations within this group EXCEPT Part 1a - Test Conditions. Discussion of test conditions appropriate to BRANCH GROUP op-coded is presented below.

4.2.4.1 Test Conditions

The following conditions are tested by op-codes within the BRANCH GROUP to determine whether or not the branch condition for the appropriate operation is satisfied.

1. Selected Bits of the AP INTERNAL STATUS REGISTER (APSTATUS)

2. Selected Input/Output Flags

* BRANCH Related Bits in APSTATUS REGISTER

Bit Name	Condition	Related BRANCH Op-code(s)
OVF (bit 0)	Set to "1" when the FA or FM available during the preceding cycle has OVER-FLOWED. Once set to a 1, OVF remains latched until cleared by the program or host computer.	BFPE (See note)
UNF (bit 1)	Set to "1" when the FA or FM available during the preceding cycle has UNDER-FLOWED. Once set to a "1", UNF remains latched until cleared by the program or host computer.	BFPE (See Note)
DIVZ (bit 2)	Set to "1" when a divide by zero has been detected by the divide software. Remains latched until set or cleared by the program or host computer.	BFPE (See Note)
FZ (bit 3)	Set to "1" when the FA available during the preceding cycle was zero. Cleared to "0" otherwise.	BFEQ, BFNE, BFGT (See Note)
FN (bit 4)	Set to "1" when the FA available during the preceding cycle was negative. Cleared to "0" otherwise.	BFGE, BFGT (See Note)
Z (bit 5)	Set to "1" when SPFN during the most recent preceding S-PAD operation equaled 0. Cleared to 0 otherwise.	BEQ, BNE, BGT (See Note)
N (bit 6)	Set to "1" when SPFN during the most recent preceding S-PAD operation was negative. Cleared to "0" otherwise.	BGE, BGT (See Note)

NOTE

Indicates that the named op-code tests two or more conditions in order to determine status of branch conditions.

* I/O FLAGS Test by BRANCH GROUP Op-codes

The I/O DATA READY FLAG (IODRDY) and the INTERRUPT REQUEST FLAG (INTRQ)may be tested by appropriate op-codes within this group and branches made accordingly.

BRANCH - Related I/O Flags

AP I/O Flag	Branch Contingency	Related BRANCH Op-code
IODRDY	Set to "1"	BION
	Set to "0"	BIOZ
INTRQ	Set to "1"	BINTRQ

4.2.4.2 BRANCH TARGET ADDRESSING

The BRANCH TARGET ADDRESS is calculated by adding the current contents of the PROGRAM SOURCE ADDRESS REGISTER (PSA) with the value (BIAS removed) contained in the DISPlacement field of the instruction word. The biasing scheme used for calculating BRANCH TARGET ADDRESSES is explained fully in SPEC SUMMARY - Types of Branch Operations.

In assembly format, the BRANCH TARGET ADDRESS is specified in the following manner:

```
BIFZ        TARG
```

Where TARG may be a name or number specifying a target address. The BRANCH TARGET ADDRESS must be within a range from +17(octal) to -20(octal) locations relative to the current PROGRAM SOURCE ADDRESS (PSA). A relative address within the proper range may be specified directly by using the assembler symbol ".", which always has the value of the current location, i.e., BR .-3. (means branch three locations backwards from the current location).

NOTE

The following rules apply to ALL conditional branches:

1. Conditional branches test their particular condition as it existed during the previous instruction.

2. The branch determines what instruction will be executed NEXT. It has no effect upon execution of the current instruction.

See a more complete explanation in the SPEC section.

4.2.5 AP-120B Internal Status Register (APSTATUS) Summary

APSTATUS is a 16-bit read/write register containing status bits which may be used to monitor conditions pertinent to Floating-Point Arithmetic results, the S-Pad RESULT (SPFN), S-Pad CARRY, FFT and IFFT addressing functions, SUB-ROUTINE-RETURN-STACK OVERFLOW conditions, and optionally, MEMORY-PARITY.

Changes in APSTATUS bits are generated by either AP-120B hardware functions or programmed instruction, (whether via AP-120B or Host-CPU). APSTATUS is readable to the AP-120B via use of the RAPS instruction (see I/O, RDREG field). It is writable from the AP-120B via use of the LDAPS instruction (see I/O, LDREG field) except for PERR, which is "set-only"; PERR and PENB=0 if parity option is not present. Changes in STATUS-BITS become effective, with respect to subsequent testing operations, one AP cycle after the condition occurred that caused the change.

APSTATUS is cleared via a Panel Reset (To clear the I/O interface the programmer must also use the interface reset). PERR and PENB are also cleared by an interface reset.

Given below is the APSTATUS format and descriptions of the individual STATUS-BITS.

APSTATUS FORMAT

0	1	2	3	4	5	6	7	8	9	10	11	12	13 14 15
OVF	UNF	DIVZ	FZ	FN	Z	N	C	PERR	PENB	SRAO	IFFT	FFT	Bit Reverse

BIT	MNEMONIC	MEANING
0	OVF	OVERFLOW: Set one cycle after the current FADDR or FMULR RESULT (FA or FM) has OVERFLOWED. Set by AP-120B hardware. Remains set until cleared by programmed instruction (LDAPS) or panel RESET, (See Note 1).
1	UNF	UNDERFLOW: Set when current FADDR or FMULR RESULT (FA or FM) has UNDERFLOWED. Set by AP-120B hardware. Remains set until cleared by programmed instruction (LDAPS) or panel RESET, (See Note 2).
2	DIVZ	A DIVIDE-BY-ZERO has occurred. Set and cleared by programmed instruction (LDAPS)(See Note 3) or panel RESET.

3	FZ	FA-ZERO: Set to "1" one cycle after the current FADDR-RESULT (FA) equals 0.0. Cleared to "0" when FA does not equal 0.0 (Via AP-120B hardware or programmed instruction or panel RESET.)
4	FN	FA-NEGATIVE: Set to "1" one cycle after the current FADDR-RESULT (FA) <0.0. Cleared to "0" when FA> 0.0. (Via AP-120B hardware or programmed instruction or panel RESET.)
5	Z	SPFN-ZERO: Set to "1" one cycle after the current S-Pad function (SPFN) equals 0. Cleared to "0" when SPFN does not equal 0. (Via AP-120B hardware or programmed instruction or panel RESET.)
6	N	SPFN-NEGATIVE: Set to "1" one cycle after the current S-PAD function (SPFN) < 0. (Via AP-120B hardware or programmed instruction.)
7	C	S-PAD CARRY: If an S-PAD shift was specified, "C" reflects the last bit shifted off SPFN as a result of the shift operation. If a shift was not specified "C" reflects the state of S-PAD CARRY BIT or panel RESET.
8	PERR	PARITY-ERROR: (Optional). Set when a Main Data Memory (MD) PARITY-ERROR has occurred. Three parity-bits are used, one each for the EXPONENT, HIGH-MANTISSA, and LOW-MANTISSA portions of the memory word (See Note 4). If "PENB" is set, the AP-120B will halt on this error. Cleared only via panel or interface reset. Set only by LDAPS.
9	PENB	PARITY-HALT-ENABLE: (Optional). Enables halt on MEMORY PARITY ERROR. If set, the AP-120B will halt when a MEMORY-PARITY-ERROR is detected. Set or cleared by LDAPS.Cleared by panel or interface reset.
10	SRAO	SUB-ROUTINE-RETURN-STACK OVERFLOW. Set to "1" via AP-120B hardware if more than 16 levels nested sub-routine-calls have occurred, or if a "RETURN" is executed without a corresponding "JSR". (Cleared via programmed instruction or panel RESET.)

11	IFFT	INVERSE-FFT FLAG. (Set via programmed instruction.) When set in conjunction with the FFT FLAG (bit 12), this bit causes ROOTS-OF-UNITY table references to be interpreted as positive angles when set and negative angles when cleared. (Cleared via programmed instruction or panel RESET.)
12	FFT	FFT FLAG. Set and cleared via programmed instruction or panel RESET. When set, causes Table Memory Addresses to be interpreted as angles referencing the ROOTS-OF-UNITY table contained in Table Memory.
13-15	BIT-REVERSE	BIT-REVERSE SHIFT-VALUE. Three-bit field whose value controls the number of shifts accompanying an address BIT- REVERSING operation. See (S-PAD, BIT-REVERSE.) Shift value to be placed in this field is determined by the following equation:

$$\text{BIT-REVERSE} = 15 - (\log_2 N)$$

Where: N = length of complex-data-array to which BIT-REVERSE operation (&) is being applied.

NOTES

1. OVERFLOW occurs when EXPONENT of result is increased above APPARENT-VALUE of 1023 (TRUE-VALUE of 511). APMAX or APNMAX is forced as the result, depending on the sign of the MANTISSA.

2. UNDERFLOW occurs when EXPONENT of result is decreased below APPARENT-VALUE of 0 (TRUE-VALUE of -512.) ZERO (0.0) is forced as the result.

3. Result is set to the value of the dividend when a DIVIDE-BY-ZERO has occurred. (Via programmed instruction.) Used by AP-120B Math Library Divide Routines.

4. EXPONENT, HIGH-MANTISSA and LOW-MANTISSA PARITY-BITS occupy data-word bit positions 00, 01, and 40, respectively.

Selected STATUS-BITS within APSTATUS may be tested and branched upon (if the appropriate condition is satisfied) by the following AP-120B instructions:

INSTRUCTION WORD GROUP	SUB-GROUP	INSTRUCTION MNEMONIC	BRANCH CONDITION	RELATED BIT(S) in APSTATUS
BRANCH	COND	BFPE	Floating Point Error	UNF, OVF or DIVZ = 1
		BFEQ	FA=0.0	FZ=1
		BFNE	FA≠0.0	FZ=0
		BFGE	FA≥0.0	FN=0
		BFGT	FA>0.0	FN,FZ=0
		BEQ	SPFN=0	Z=1
		BNE	SPFN≠0	Z=0
		BGE	SPFN≥0	N=0
		BGT	SPFN>0	Z,N=0
SPEC	STEST	BFLT	FA<0.0	FN=1
		BLT	SPFN<0	N=1
		BNC	S-PAD-CARRY=1	C=1
		BZC	S-PAD-CARRY=0	C=0
		BIFN	IFFT Bit=1	IFFT=1
		BIFZ	IFFT Bit=0	IFFT=0

4.2.6 PERR and PENB, Theory of Operation

PERR, the Parity Error bit, will be set to a "one" by the Parity Option Logic any time a parity-error is detected on a read cycle from Main Data memory. The Parity-Option Logic checks parity on all read cycles whether from the AP processor, the Host Interface or other DMA device such as the IOP and the PIOP. When PERR is set to a "one" as a result of a detected parity error, the data word, parity bits, memory address and cycle acknowledge priority level of the failure are recorded in the logging registers. As a check of the Parity Error Registers, PERR can be set from the AP. If PERR is set via an LDAPS the logging registers will record the above information for the memory read cycle just completed.

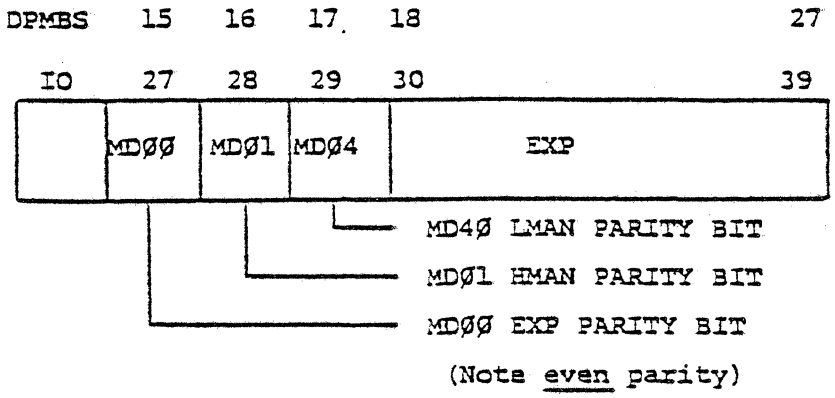
When PENB and PERR are both set, the AP will halt immediately. In fact, if an attempt is made to start or continue the AP when both of these bits are set, the AP will not go into the running state. Also, because of the overlap fetch/execute, the next instruction will have been set up. This may cause problems when restarting if this is not taken into account.

The immediate halt characteristic may cause lost interrupt problems with certain Host Interface/driver combinations. Thus, the recommended start-up procedure in the presence of DMA transfers overlapped with AP running is to clear PENB via a panel deposit to APSTATUS before starting the AP. Then to have the initializer code in the AP, set PENB when it clears OVF UNF and DIVZ before branching to the user-called routine. The "Set-only" characteristic of PERR protects the Parity Option from losing parity errors.

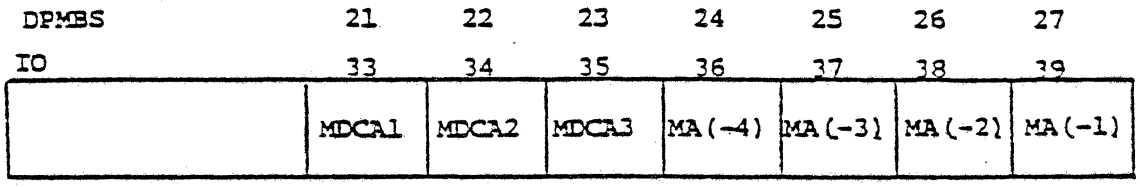
Note that contrary to common industry practice, the Parity Option generates even-parity. The highly inter-leaved nature of optimized, AP math-library routines requires many of them to read extra locations past the end of the arrays on which they are operating. Thus even-parity was selected so that a read from non-existent memory (all zeros on data and parity bits) would not cause spurious parity errors.

The following parity-error registers can be read via IN instructions at their respective Device Addresses. They reflect the first data-word, address and memory priority level that caused a parity error.

DA	REGISTER	DESCRIPTION
33	MDLMAN	Low Mantissa portion of failing location
34	MDHMAN	High Mantissa portion of failng location
35	MDEXP	Exponent and parity bits of failing location (Format below)



DA	REGISTER	DESCRIPTION
36	PEAD	Parity Error Address Address of Failing Location (least significant 16 bits)
37	MDCA	Main Data Cycle Acknowledge priority level and Address Extension. (Format below)



4.3 FLOATING ADDER SUMMARY (FADDR)

Discussion of the AP-120B FLOATING-POINT ADDER (FADDR) is presented in the following manner:

1. General description and theory of operation
2. FADDR single and double operand operations
3. FADDR Operands -- A1 and A2
4. FADDR Result -- FA
5. FADDR-associated test, branch, and error conditions
6. FADDR programming considerations

4.3.1 General Description, Theory of Operation

The AP-120B FLOATING-POINT ADDER (FADDR) is a two-stage arithmetic, logical, and format-conversion unit that uses 38-bit FLOATING-POINT NUMBERS as its operands.

* The Operands (A1, A2)

The operands (contained in the A1 and A2 registers) are selected by the octal-value contained in the respective A1 and A2 fields of the current instruction word.

The available inputs to the two FADDR REGISTERS are listed in the A1 and A2 summaries. (See Section 4.3.4).

* The Operations

The particular operation selected by the octal value in the FADD, or FADD1 fields is then performed. (See Section 4.3.2).

* The Result (FA)

The result of a specified FADDR operation is enabled onto the FLOATING ADDER OUTPUT BUS (FA) one cycle after the next FADDR operation is initiated. The result is either NORMALIZED and CONVERGENTLY-ROUNDED, or unnormalized and rounded, or unnormalized and TRUNCATED, depending on the operation specified. (See FLOATING POINT SUMMARY, for more details on NORMALIZATION and ROUNDING/TRUNCATION operations.)

Because of the unique configuration of the AP-120B FADDR, after the initial "pipeline" set-up requirements have been satisfied, the FADDR is capable of producing significant FLOATING-POINT ARITHMETIC results every instruction cycle (167 ns). Note that the AP-120B allows simultaneous FLOATING-POINT ADDER (FADDR) and FLOATING-POINT MULTIPLIER (FMULR) operations.

* Theory of Operation

The AP-120B FLOATING-POINT ADDER (FADDR) is essentially a two-stage pipeline which operates in the following manner:

Stage One

In the first stage, the EXPONENTS of the A1 and A2 OPERANDS are compared. The larger of the two EXPONENTS becomes the EXPONENT of the result. The MANTISSA of the smaller operand is correspondingly arithmetically right-shifted a number of places that reflects the magnitude of the difference between the two source-operand EXPONENTS.

The "aligned" mantissas then undergo the specified FADDR operation and the result is presented to the second-stage buffer latch.

When a subsequent FADDR operation is initiated, the preliminary result is "pushed" down into Stage Two.

Stage Two

In Stage Two, the preliminary result will be NORMALIZED or not and either CONVERGENTLY-ROUNDED or TRUNCATED, depending on the operation specified. (See Section 4.3.2).

This result becomes available onto the FA BUS one instruction cycle later. Appropriate bits of the APSTATUS REGISTER will be set according to the condition of this FADDR result and may be tested for significance one cycle after this result is enabled onto FA (two cycles after the second FADDR operation).

If the difference in exponents exceeds 31 (the number of significant bits in the AP FLOATING-POINT MANTISSA and FADDR GUARD BITS), a SHIFT-INHIBIT will occur, causing the operand with the SMALLER EXPONENT to be interpreted as positive 0.0. The result of the specified operation will reflect this interpretation.

NOTE

The APSTATUS bits will remain latched until one cycle after the result of a subsequent FADDR operation becomes available as FA.

As stated before, the result of an initial FADDR operation becomes available as FA only after a subsequent FADDR operation has been specified. In other words, the result of a given FADDR operation must be pushed down the "pipeline" and out onto the FA Bus by a successive FADDR operation -- the result of which, in turn, must be pushed out by yet another successive FADDR operation. Example:

	OPERATION (COMMENTS)	RESULT AVAILABLE AS FA
t1	FADD DPX,DPY	---------
t2	FADD TM,MD	---------
t3	NOP	(FA = DPX + DPY)
t4	NOP	(FA = DPX + DPY)
t5	FADD (PUSHES PIPELINE)	(FA = DPX + DPY)
t6	NOP	(FA = TM + MD)

4.3.2 FADDR Single and Double Operand Operations

The FLOATING ADDER Performs Floating Point:

* Arithmetic
* Logical
* Format Conversion, and
* Scaling Operations (floating to variable-width fixed-point conversion)

Instructions in the FADD field are double-operand operations which use the A1 and A2 fields to specify source-operands. Instructions in the FADD1 field use only a single-operand. When a FADD1 field op-code is specified (i.e. FADD = 0), the input selected as the source operand is determined by the octal-value in the A2 field.

Only one op-code from the following groups may be initiated during a given instruction cycle.

ARITHMETIC	OPCODE FIELD USED	OCTAL VALUE	SOURCE OPERAND FIELD(S) USED	OPERATION
FADD	FADD	3	<A1, A2>	(A1) + (A2)
FSUB	FADD	2	<A1, A2>	(A1) - (A2)
FSUBR	FADD	1	<A1, A2>	(A2) - (A1)

4.3.3 Floating Point Logical Operations

These instructions (FAND, FOR, FEQV) perform logical operations on floating-point numbers. Exponent alignment occurs as for a normal floating-point add. The two mantissas are then combined using the specified logical operations. The result is then normalized and rounded.

LOGICAL COMPARISON	OPCODE FIELD USED	OCTAL VALUE	SOURCE/OPERAND FIELD(S) USED	OPERATION
FEQV	FADD	4	<A1, A2>	(A1) EQV (A2)
FAND	FADD	5	<A1, A2>	(A1) AND (A2)
FOR	FADD	6	<A1, A2>	(A1) OR (A2)
FORMAT CONVERSION				
FIX	FADD1	1	A2	Convert (A2) to a 28-bit integer (rounded)
FIXT	FADD1	2	A2	Convert (A2) to a 28-bit INTEGER; TRUNCATE the RESULT (See Note 1)
FSM2C	FADD1	4	A2	Convert (A2): from SIGNED-MAGNITUDE to TWOS-COMPLEMENT
F2CSM	FADD1	5	A2	Convert (A2): from TWOS-COMPLEMENT to SIGNED-MAGNITUDE
FABS	FADD1	7	A2	Convert (A2): to its ABSOLUTE VALUE
SCALING				
FSCALE	FADD1	6	A2	Scale (A2) using SPFN as reference. Result rounded. (See Note 2)
FSCLT	FADD1	3	A2	Scale (A2) using SPFN as reference; RESULT TRUNCATED (See Notes 1 and 2)

<A1, A2> indicates that source operands need not be specified if a FADD "dummy" operation is used to push the result of the preceding FADDR operation down the pipeline.

NOTES

1. TRUNCATION - In an FSCLT or FIXT operation, the result will be TRUNCATED rather than CONVERGENTLY ROUNDED. (See FLOATING-POINT SUMMARY - ROUNDING/TRUNCATION.)

2. The current SPFN when the FSCALE operation is initiated. For correct results SPFN must equal maximum-comparison ($A1^{exponent}$ +1).

Note that the BIAS BIT of a given EXPONENT is automatically removed when the EXPONENT is transferred to S-PAD via an LDSPE instruction. (See SOP1). Accordingly, the value contained in a given S-PAD register following an LDSPE instruction will be the TRUE-VALUE of the EXPONENT. However, the BIAS BIT for a given EXPONENT is not removed if the transfer is via an LDSPI or LDSPNL instruction.

When the contents of a given S-PAD register are transferred to FADDR $A1^{exponent}$, $A2^{exponent}$ or to DB (via DB = SPFN), the BIAS is automatically added, thus producing an APPARENT-VALUE 512 greater than the TRUE-VALUE.

4.3.4 FADDR Operands (via A1, A2 Registers)

The FADDR uses A1 and A2 registers as the input-buffers for the source operand(s) specified for a given operation. The programmer may select one of six available sources to be used as the A1 REGISTER operand and one of eight sources for the A2 REGISTER operand.

The list below shows the sources available for each respective FADDR register.

A1 REGISTER SOURCE	A1 FIELD VALUE (IN OCTAL)	A2 REGISTER SOURCE	A2 FIELD VALUE (IN OCTAL)
NC*	0	NC*	0
FM	1	FA	1
DPX(idx)	2	DPX(idx)	2
DPY(idx)	3	DPY(idx)	3
TM	4	MD	4
ZERO**	5	ZERO**	5
		MDPX(idx)***	6
		EDPX(idx)***	7

NOTES

* NC: NO CHANGE — The input-buffers remain unchanged. This mnemonic is implied if no FADDR operands are specified.

** ZERO: Floating Point 0.0

*** MDPX: { Indicates split-word source. See A2 for a detailed

*** EDPX: { explanation of these sources.

Although the appropriate instruction-word fields and octal-values are listed above, the assembler format coding need only specify the desired operation and the operand source(s). Examples:

ASSEMBLER FORMAT	COMMENTS
FADD	"Dummy" FADD
FADD NC,NC	"Dummy" FADD
FSUB FM,FA	Subtract (FA) from (FM)
FSUBR FM,FA	Subtract (FM) from (FA)
MOV 5,5;FSCALE TM	Shift (TM) Right arithmetically a number of positions that is one less than the difference between SP(5) and the TRUE-VALUE of the EXPONENT of (TM)

4.3.5 The FADDR Result (FA)

The NORMALIZED and either CONVERGENTLY-ROUNDED or TRUNCATED RESULT* becomes available onto the FLOATING ADDER OUTPUT BUS (FA) one instruction cycle after the next FADDR operation is initiated. This FA will remain latched until replaced following the next FADDR operations.

The FLOATING ADDER OUTPUT (FA) may be directed to:

* The FLOATING-ADDER A2 REGISTER
* The FLOATING-MULTIPLIER M2 REGISTER
* DATA PAD X or DATA PAD Y, or
* to MAIN DATA MEMORY INPUT REGISTER (MI)

FA sets the appropriate bits of the APSTATUS REGISTER and these bits may be tested for significance on the instruction cycle after FA becomes valid.

*In case of FA OVERFLOW or UNDERFLOW, a signed-maximum or ZERO number is forced as the result. (See FLOATING-POINT SUMMARY-OVERFLOW/UNDERFLOW.)

4.3.6 FADDR Test, Branch, and Error Condition

The FADDR result (FA) sets or clears appropriate bits of the APSTATUS REGISTER. These bits may be tested and branches made on their condition one instruction cycle after the appropriate FADDR result is enabled onto FA.

FADDR RELATED BITS IN APSTATUS

APSTATUS BIT NAME	CONDITION	RELATED BRANCH OPCODE(S)
OVF (bit 0)	Set to "1" when the current FA or FM has OVERFLOWED (See note 2). OVF remains latched until cleared by the microprogram or HOST-CPU.	BFPE (See note 1)
UNF (bit 01)	Set to "1" when the current FA or FM has UNDERFLOWED (See note 2). UNF remains latched until cleared by the microprogram or HOST-CPU.	BFPE (See note 1)
FZ (bit 03)	Set to "1" when the current FA is equal to "0.0"; cleared to "0" when current FA is not equal to "0.0".	BFGE BFGT BFNE
FN (bit 04)	Set to "1" when the current FA is negative; cleared to "0" when current FA is non-negative.	BFLT

NOTES

1. Indicates that the named op-code tests two or more conditions in order to determine status of branch condition.

2. See FLOATING POINT SUMMARY-OVERFLOW.

4.3.7 Floating Point Adder Programming Considerations

4.3.7.1 Simple Examples

Any data source listed under A1 may be combined with any data source listed under A2. For example, to add a number from Data Pad X to another from Data Pad Y:

```
FADD DPX, DPY          "DPX + DPY
```

or to subtract a number read out of Data Memory from a constant in Table Memory:

```
FSUB TM,MD             "TM - MD
```

A reverse subtract changes the order of the subtraction, i.e.,

```
FSUBR TM,MD            "MD - TM
```

subtracts a constant from Table Memory from a number in Data Memory.

To negate a number from DPX:

```
FSUB ZERO, DPX         "0.0 - DPX = -DPX
```

To take the absolute value of a number from Data Memory:

```
FABS MD                "ABS(MD)
```

To fix (convert from floating-point to integer) a number from DPY:

```
FIX DPY                "FIX (DPY)
```

4.3.7.2 Pipelining Considerations

The Floating Adder is a two-stage pipeline. A "FADD" instruction loads the designated operands into the A1 and A2 registers. The previous contents of A1 and A2 are pushed down the pipeline to the Buffer register. One AP cycle later the new contents of Buffer have been normalized and rounded, and are then available for use or storage elsewhere.

The following instruction sequence illustrates how the Adder pipeline works, where A,B...G,H are floating-point numbers to be added:

Time	Cycle	Instruction	Adder Pipeline: A1, A2	Buffer	Adder Result (FA)
0	1.	FADD A,B	A,B	---	---
167ns	2.	FADD C,D	C,D	A,B	---
333ns	3.	FADD E,F	E,F	C,D	A+B
500ns	4.	FADD G,H	G,H	E,F	C+D
667ns	5.	FADD	---	G,H	E+F
833ns	6.	---	---	G,H	G+H

The "FADD" without arguments in cycle 5 is used only to push the last computation into the Buffer Register, and hence to the end of the pipeline. Thus, it is a dummy add in the sense that we don't care what its arguments are, since we will never use the results. In the above example we completed our floating-point adds in one microsecond. During cycles 2-4, while we kept the pipeline full, adds were being done every 167ns, the maximum rate.

The completed results, as they come out of the Adder pipeline, are referred to by the mnemonic "FA." FA is dynamic, in the sense that it must be used or stored elsewhere before being changed by the next floating-adder instruction. The programmer has, however, complete control over the pipeline. Arguments advance ONLY when pushed through the pipeline by floating-adder instructions.

4.3.7.3 Pipelining Example

A complete computational sequence is to do the vector sum Ax=Ax+Bx, i=0,1,2,3. Ax is stored in Data Pad X locations 0-3 and Bx is stored in Data Pad Y location 0-3.

1. FADD DPX(Ø), DPY(Ø) "Do A_0+B_0
2. FADD DPX(1), DPY(1) "Do A_1+B_1
3. FADD DPX(2), DPY(2); DPX(Ø)<FA "Do A_2+B_2, A_0+B_0 is now done, save it in A_0
4. FADD DPX(3), DPY(3); DPX(1)<FA "Do A_3+B_3, A_1+B_1 is now done, save it in A_1
5. FADD DPX(2)<FA "Push Adder; save A_2+B_2 in A_2
6. DPX(3)<FA "Save A_3+B_3 in A_3

Below is a chart of this computation, showing the state of the Adder pipeline and Data Pad after each instruction is executed.

	Adder Pipeline			Adder	Data Pad X:			
Cycle	A1,	A2	Buffer	Result	0	1	2	3
1.	A_0,	B_0	---	---	A_0	A_1	A_2	A_3
2.	A_1,	B_1	A_0, B_0	---	A_0	A_1	A_2	A_3
3.	A_2,	B_2	A_1, B_1	A_0+B_0	A_0+B_0	A_1	A_2	A_3
4.	A_3,	B_3	A_2, B_2	A_1+B_1	A_0+B_0	A_1+B_1	A_2	A_3
5.	---		A_3, B_3	A_2+B_2	A_0+B_0	A_1+B_1	A_2+B_2	A_3
6.	---		A_3, B_3	A_3+B_3	A_0+B_0	A_1+B_1	A_2+B_2	A_3+B_3

4.3.7.4 FADDR Branches Programming Considerations

The FADDR branches test "FA" one instruction cycle after it is ready for use. That is, an Adder result may be tested one cycle after it has come out of the Adder pipeline. An example:

```
1. FSUB DPX,DPY      "Do a computation
2. FADD              "Push the result out
3. DPX>FA            "Save the result
4. BFEQ LOOP         "Test the result here (branch
                     "to location "LOOP" if result
                     "was zero
```

Compound tests may be made also. Test MD to see if it is between a lower limit contained in DPX (1) and an upper limit in DPX (2), i.e., see if DPX(1)<MD<DPX(2):

```
1. FSUBR DPX(2), MD     "Do MD-DPX(2)
2. FSUB DPX(1), MD      "Do DPY(1)-MD
3. FADD                 "Push first test result
                        "out
4. BFGT BIG             "Was too big
5. BFGT SMALL           "Was too small
6. . . .                "OK
```

The branches are made relative to the current Program Source Address (PSA), with a 5-bit displacement value. This means that the conditional branch target address must be within -20(octal) to +17(octal) locations of the current instruction.

4.4 FLOATING MULTIPLIER (FMULR)

Discussion of the AP-120B FLOATING POINT MULTIPLIER (FMULR) is presented in the following manner:

1. General description and theory of operation
2. FMULR operation -- FMUL
3. FMULR operands -- M1 and M2
4. FMULR result -- FM
5. FMULR associated TEST, BRANCH, and ERROR conditions
6. FMUL Programming Considerations

4.4.1 General Description, Theory of Operation

The AP-120B FLOATING POINT MULTIPLIER (FMULR) is a three-stage multiplication unit using 38-bit FLOATING POINT NUMBERS as its operands.

4.4.1.1 The Operands (M1, M2)

The operands (contained in M1 and M2 registers) are selected by the value contained in the respective M1 and M2 fields of the current instruction word.

The available inputs to the two FMULR registers are described in detail in Section 4.4.3 and in M1, M2 of the instruction summary.

FMULR operands must not be unnormalized by more than one bit position or an unnormalized, and thus possibly inaccurate, product will result. With unnormalized operands the result will be incorrect in that it will be unnormalized by the sum of the number of unnormalized bit positions of the two input arguments.

Since the FMULR internally retains only 28 bits of MANTISSA Result (the full 56-bit product is generated in order to produce a clean 28-bit result), use of unnormalized operands can yield results with the loss of many, if not all, of the bits of significance expected for the result.

4.4.1.2 The Result (FM)

The NORMALIZED, CONVERGENTLY-ROUNDED result is enabled onto the FLOATING MULTIPLIER BUS (FM) one cycle after the second subsequent FMULR operation is initiated, (See Section 4.4.4)

After the initial "pipeline" set-up requirements have been satisfied, the FMULR can produce significant FLOATING POINT MULTIPLICATION results every instruction cycle (167 ns). Note that the AP-120B allows simultaneous FLOATING POINT MULTIPLIER (FMULR) and FLOATING POINT ADDER (FADDR) operations.

4.4.1.3 Theory of Operation

The process of multiplying two FLOATING POINT NUMBERS (FPN's) requires that the true EXPONENTS of both operands be added and the MANTISSAS of both operands be multiplied. The sum of the EXPONENTS becomes the

EXPONENT of the result.

The AP-120B FLOATING POINT MULTIPLIER (FMULR) is a THREE-STAGE pipeline which operates in the following manner:

* Stage One

In the first stage, the FMUL instruction loads the M1 and M2 operands. A partial multiplication is then performed on the two MANTISSAS and the EXPONENT true-values are added.

. . .

When a second FMUL is executed, the partial product and EXPONENT sum are "pushed" down into STAGE TWO by being latched in the second-stage buffer.

* Stage Two

In the second stage, the MANTISSA multiplication operation is completed.

. . .

When a third FMUL is executed, the preliminary result is "pushed" down into the STAGE THREE buffer.

* Stage Three

In the third stage, the result is NORMALIZED and CONVERGENTLY-ROUNDED.

This result becomes available as FM on the next instruction cycle after the third FMUL. THe UNDERFLOW or OVERFLOW bit of the APSTATUS REGISTER will be set according to the condition of this result and may be tested for significance one cycle after this result becomes available as FM (two cycles after the third FMUL). These two status bits, once set, remain set until they are cleared via LDAPS (see I/O group) or by a RESET operation from the Host to the panel reset. The interface reset, however, does not affect the APSTATUS register.

Stated again, the result of an initial FMUL becomes available as FM only after two subsequent FMULS have been initiated.

Example:

	OPERATION (COMMENTS)	RESULT AVAILABLE AS FM
t1	FMUL DPX,DPY	--------
t2	FMUL TM,MD	--------
t3	FMUL DPY(3),DPX(2)	--------
t4	ZZ	(FM = DPX * DPY)
t5	ZZ	(FM = DPX * DPY)
t6	FMUL (pushes pipeline)	(FM = DPY * DPY)
t7	ZZ	(FM = TM * MD)
t10	ZZ	(FM = TM * MD)

ZZ = Any non-FMUL instruction.

NOTE

The result of the FMUL operation initiated at t3 is at t10 still "hanging" in the pipeline, where it will remain until one cycle after another FMUL is initiated.

4.4.2 The FMULR Operation — FMUL

When the FM field of the instruction word = 1 and VALUE field is not used, a FLOATING MULTIPLY (FMUL) is initiated using the source operands selected by the octal value of the M1 and M2 fields of the instruction word. (M1 * M2)

4.4.3 FMULR Operands (via M1, M2 registers)

The FMULR uses M1 and M2 registers as the input buffers for the operands selected for the current FMUL operation. The programmer may select one of four available sources to be used as the M1 REGISTER operand and one of four available sources for the M2 REGISTER operand. Source operands must not be UNNORMALIZED by more than one bit position or an incorrect (unnormalized) result will be obtained.

The list below shows the sources available for each respective FMULR register. (See M1, M2 for detailed descriptions.)

M1 REGISTER SOURCE	M1 FIELD VALUE (IN OCTAL)	M2 REGISTER SOURCE	M2 FIELD VALUE (IN OCTAL)
FM	0	FA	0
DPX(idx)	1	DPX(idx)	1
DPY(idx)	2	DPY(idx)	2
TM	3	MD	3

For coding in ASSEMBLER FORMAT, the programmer need only specify FMUL with the desired operand sources. Examples:

ASSEMBLER FORMAT	COMMENTS
FMUL	"dummy" FMUL actually an FMUL FM,FA
FMUL FM, FA	Multiply: Current (FM) * current (FA)
FMUL DPY(-3),DPX	Multiply: (DPY(DPA-3)) * (DPX(DPA))

4.4.4 The FMULR Result (FM)

The FLOATING MULTIPLIER OUTPUT (FM) may be directed to:

1. FMULR M1 REGISTER
2. FADDR A1 REGISTER
3. MAIN DATA MEMORY INPUT REGISTER (MI)
4. DATA PAD Y (DPY), or
5. DATA PAD X (DPX)

FM can set the UNDERFLOW OR OVERFLOW bits of the APSTATUS REGISTER. These bits may be tested for significance one AP cycle later (after FM is valid).

NOTE

In the case of FM OVERFLOW or UNDERFLOW, a signed maximum or zero is forced as the result. (See FLOATING POINT SUMMARY, OVERFLOW/UNDERFLOW.)

4.4.5 FMUL Test, Branch, and Error Conditions

The FMUL results (FM) can set UNDERFLOW or OVERFLOW bits of the APSTATUS REGISTER. These bits may be tested and branches made on their condition one instruction cycle after the appropriate FADDR result is enabled onto FA.

APSTATUS BIT NAME	CONDITION	RELATED BRANCH OPCODE(S)
OVF (bit 0)	Set to "1" when the current FM has overflowed, (See Note 2), OVF remains latched until cleared by the micro-program or host computer.	BFPE (See Note 1)
UNF (bit 1)	Set to "1" when the current FM result has underflowed; (See Note 2), UNF remains latched until cleared by the microprogram or host computer.	BFPE (See Note 1)

NOTES

1. Indicates that the named op-code tests two or more conditions in order to determine status of branch condition.

2. See FLOATING POINT SUMMARY, OVERFLOW/ UNDERFLOW.

4.4.6 FMUL Programming Considerations

4.4.6.1 Simple Examples

Any of the data sources listed under M1 may be multiplied by any of the data sources in M2. For example, to multiply a number read from Data Memory by a constant from Table Memory:

```
FMUL TM,MD      "TM * MD
```

or, to multiply a number in Data Pad X by another number in Data Pad Y:

```
FMUL DPX,DPY    "DPX * DPY
```

4.4.6.2 Pipelining Considerations

The Floating Multiplier is a three-stage pipeline. An "FMUL" instruction loads the specified operands into the M1 and M2 registers. The two previous partially completed products are pushed down the pipeline to Buffer 2 and Buffer 3 respectively. One AP cycle later the new contents of Buffer 3 have been normalized and rounded, and are then available for use or storage elsewhere.

The following instruction sequence illustrates how the Multiplier pipeline works, where A,B...G,H are floating-point numbers to be multiplied together.

			Multiplier Pipeline			
Time	Cycle	Instruction	M1, M2	BUFFER 2	BUFFER 3	Multiplier Result (FM)
0	1.	FMUL A,B	A,B	---	---	---
167ns	2.	FMUL C,D	C,D	A,B	---	---
333ns	3.	FMUL E,F	E,F	C,D	A,B	---
500ns	4.	FMUL G,H	G,H	E,F	C,D	A*B
667ns	5.	FMUL	---	G,H	E,F	C*D
833ns	6.	FMUL	---	---	G,H	E*F
1.0us	7.	---	---	---	G,H	G*H

The "FMUL" in cycles 5 and 6 are dummy multiplies used to push the last two computations to the end of the pipeline. In the above example we completed four floating-point multiplies in 1.0us. During cycles 3-4, while the pipeline was full, products were being done every 167ns, the maximum rate.

The completed products as they come out of the Multiplier pipeline are referred to by the mnemonic "FM." FM is dynamic, in that it must be used or stored before being changed by the next "FMUL" instruction.

4.4.6.3 Pipelining Example

A computational example is to square the elements in a vector:

$A_i = A_i * A_i$, i=0,1,2,3. A_i is stored in Data Pad X.

```
1.  FMUL DPX(Ø),DPX(Ø)                      "Do A0^2
2.  FMUL DPX(1),DPX(1)                      "Do A1^2
3.  FMUL DPX(2),DPX(2)                      "Do A2^2
4.  FMUL DPX(3),DPX(3); DPX(Ø)<FM           "Do A3^2, save A0^2
5.  FMUL; DPX(1)<FM                         "Save A1^2
6.  FMUL; DPX(2)<FM                         "Save A2^2
7.  DPX(3)<FM                               "Save A3^2
```

Below is a chart of this computation, showing the state of the Multiplier pipeline and Data Pad X after each instruction is executed.

	Multiplier Pipeline				Data Pad X			
Cycle	M1,M2	Buffer 2	Buffer 3	Multiplier Result (FM)	0	1	2	3
1.	A_0, A_0	---	---	---	A_0	A_1	A_2	A_3
2.	A_1, A_1	A_0, A_0	---	---	A_0	A_1	A_2	A_3
3.	A_2, A_2	A_1, A_1	A_0, A_0	---	A_0	A_1	A_2	A_3
4.	A_3, A_3	A_2, A_2	A_1, A_1	A_0^2	A_0^2	A_1	A_2	A_3
5.	---	A_3, A_3	A_2, A_2	A_1^2	A_0^2	A_1^2	A_2	A_3
6.	---	---	A_3, A_3	A_2^2	A_0^2	A_1^2	A_2^2	A_3
7.	---	---	A_3, A_3	A_3^2	A_0^2	A_1^2	A_2^2	A_3^2

4.4.6.4 Multiply-Add Example

The full floating-point computational power of the AP-120B is utilized when we consider a process involving both multiplies and adds. Form the dot product of two eight-element vectors Ax Bx = AxBx, i = -4, -3, ... 1, 2, 3 where Ax is in Data Pad X and Bx is in Data Pad Y:

Fill the Multiplier Pipeline	1.	FMUL DPX (-4), DPY (-4)	"Do $A_{-4}B_{-4}$
	2.	FMUL DPX (-3), DPY (-3)	"Do $A_{-3}B_{-3}$
	3.	FMUL DPX (-2), DPY (-2)	"Do $A_{-2}B_{-2}$
Fill the Adder Pipeline	4.	FMUL DPX (-1), DPY (-1) FADD FM, ZERO	"Do $A_{-1}B_{-1}$. $A_{-4}B_{-4}$ is " now done, save it in " adder.
	5.	FMUL DPX (0), DPY (0); FADD FM, ZERO	"Do A_0B_0. $A_{-3}B_{-3}$ is now " done, save it in the " adder.
Both Pipelines full	6.	FMUL DPX (1), DPY (1) FADD FM, FA	"Do A_1B_1. $A_{-2}B_{-2}$ is now " coming out of the mul- " tiplier, and $A_{-4}B_{-4}$ " from the adder, add " them together.
	7.	FMUL DPX (2), DPY (2); FADD FM, FA	"Do A_2B_2. $A_{-1}B_{-1}$ is now " coming out of the mul- " tiplier, and $A_{-3}B_{-3}$ " from the adder, add " them together.
	8.	FMUL DPX (3), DPY (3); FADD FM, FA	"Do A_3B_3. A_0B_0 is now " coming out of the mul- " tiplier, and ($A_{-4}B_{-4}$ + " $A_{-2}B_{-2}$) from the adder, " add them together.
Empty the Multiplier Pipeline	9.	FMUL; FADD FM, FA	"A_1B_1 is coming out of the " multiplier, and ($A_{-3}B_{-3}$ " $+A_{-1}B_{-1}$) from the " adder, add them to- " gether.
	10.	FMUL; FADD FM, FA	"A_2B_2 is coming out of the " multiplier, and ($A_{-4}B_{-4}$ " $+A_{-2}B_{-2}+A_0B_0$) from the " adder, add them to- " gether.
	11.	FADD FM, FA	"A_3B_3 is coming out of " the multiplier, and " ($A_{-3}B_{-3}+A_{-1}B_{-1}+A_1B_1$) " from the adder, add " them together.
Empty the Adder Pipeline	12.	FADD; DPX (3)<FA	"($A_{-4}B_{-4}+A_{-2}B_{-2}+A_0B_0+A_2B_2$) " is coming out of the " adder, save it in DPX (3).
	13.	FADD DPX (3), FA	"($A_{-3}B_{-3}+A_{-1}B_{-1}+A_0B_0+A_2B_2$) " is coming out of the " adder, add it to " ($A_{-4}B_{-4}+A_{-2}B_{-2}+A_0B_0+$ " A_2B_2) which was saved " in DPX (3).
	14.	FADD	"Push result out of Adder
	15.	DPX (3)<FA	"The result: ($A_{-4}B_{-4}+$ " $A_{-3}B_{-3}+A_{-2}B_{-2}+A_{-1}B_{-1}+$ " $A_0B_0+A_1B_1+A_2B_2+A_3B_3$), " saved in DPX (3).

In accumulating the sum-of-products, the even term sum was kept in one half of the adder pipeline and the odd term sum in the other half. During cycles 5-7 when both pipelines were full, floating-point multiply-adds were being computed every 167ns. This is 12 million floating-point computations per second. A longer sum of products calculation, involving more terms, would maintain this maximum computation rate for nearly all of the computation loop. Here, in a short calculation, most of the time was spent filling and emptying pipelines. Even so, the seven adds and eight multiplies took 15 cycles (2.5us) to complete, or an overall rate of 333ns per floating-point multiply-add.

As a further aid in understanding the multiply-add interaction in the above sum-of-products computation, the chart below summarizes the computation:

	Multiplier:		Adder:			Data Pad:
Cycle	M1,M2	FM	A1,A2		FA	3
1.	A_{-4},B_{-4}	---	---		---	---
2.	A_{-3},B_{-3}	---	---		---	---
3.	A_{-2},B_{-2}	---	---		---	---
4.	A_{-1},B_{-1}	$A_{-4}*B_{-4}$	$A_{-4}B_{-4},\emptyset.\emptyset$		---	---
5.	A_0,B_0	$A_{-3}*B_{-3}$	$A_{-3}B_{-3},\emptyset.\emptyset$		---	---
6.	A_1,B_1	$A_{-2}*B_{-2}$	$A_{-2}B_{-2},A_{-4}B_{-4}$		$A_{-4}B_{-4}$	---
7.	A_2,B_2	$A_{-1}*A_{-1}$	$A_{-1}B_{-1},A_{-3}B_{-3}$		$A_{-3}B_{-3}$	---
8.	A_3,B_3	A_0*A_0	$A_0B_0,$	ES_2	ES_2	---
9.	---	A_1*A_1	$A_1B_1,$	OS_2	OS_2	---
10.	---	A_2*A_2	$A_2B_2,$	ES_3	ES_3	---
11.	---	A_3*A_3	$A_3B_3,$	OS_3	OS_3	---
12.	---	---	---		ES_4	ES_4
13.	---	---	$OS_4,$	ES_4	OS_4	ES_4
14.	---	---	---		---	ES_4
15.	---	---	---		OS_4+ES_4	OS_4+ES_4

ES is n terms of the even term Sum: $A_iB_i, i = -4,-2,0,2$
OS is n terms of the odd term Sum: $A_iB_i, i = -3,-1,1,3$

4.5 I/O GROUP

The Op-Codes available within the I/O group of the AP-120B instruction word provide the operations necessary for (1) DATA TRANSFERS between the AP-120B and the HOST COMPUTER INTERFACE (HOST-CPU I/F) or other ADDRESSABLE I/O DEVICES through the programmed I/O Section and (2) AP-120B INTERNAL REGISTER transfers to the PANEL BUS.

This summary is presented in sections related to the functional areas of the AP-120B I/O STRUCTURE: (1) the VIRTUAL FRONT PANEL (PANEL), AND (2) PROGRAMMED I/O OPERATIONS.

An outline of this summary is presented below:

I. AP-120B I/O OPERATIONS
    * General Overview
        * Panel Operations
        * Programmed I/O Operations
            * DEVICE ADDRESSING

II. AP-120B VIRTUAL FRONT PANEL (PANEL)
    * General Description
    * Panel Operations
        * General Rules

III. PROGRAMMED I/O
    a. HOST INTERFACE
        * Control Register
        * Formatter
        * Direct Memory Access operations and related transfer and control registers
        * Programmed Interrupts
    b. ADDRESSABLE I/O DEVICES
        * TMRAM
        * OTHER I/O DEVICES

IV. PROGRAMMING EXAMPLE

4.5.1 AP-120B I/O OPERATIONS

4.5.1.1 General Overview

The AP-120B I/O structure consists of two major areas of operation:

(1) VIRTUAL FRONT PANEL (PANEL) - through which the HOST-CPU may examine /or alter AP-120B INTERNAL REGISTERS and Data Memories.

(2) PROGRAMMED I/O - through which data transfers are accomplished between the AP-120B and addressable I/O DEVICES. The I/O DEVICES that may be addressed via programmed I/O operations are grouped in the following manner:

A. HOST INTERFACE
   * CONTROL REGISTER (CTL)
   * FORMATTER (FMT)
   * DIRECT MEMORY ACCESS REGISTERS:
     WORD COUNT REGISTER (WC)
     HOST MEMORY ADDRESS REGISTER (HMA, HHMA)
     AP-120B MEMORY ADDRESS REGISTER (APMA)

B. OTHER ADDRESSABLE I/O DEVICES
   * WRITABLE TABLE MEMORY (TMRAM)
   * MEMORY BANK SELECT
   * OTHER I/O DEVICES

4.5.1.2 Panel Operations

The PANEL is similar in function to the console of a stand-alone computer and consists of three 16-bit registers: LIGHTS (LITES), SWITCHES (SWR), AND FUNCTION (FN).

The PANEL is primarily under the control of the HOST-CPU. Through PANEL operations, the HOST-CPU may examine and modify internal AP-120B registers as well as dictate control functions related to AP-120B program execution. The AP- 120B may deposit into the LIGHTS REGISTER (LITES), and read the SWR.

Typically, the PANEL is used for bootstrap operations (loading and starting programs) and for debugging user software by using hardware breakpoints and/or by examining and modifying AP-120B registers and memory. PANEL OPERATIONS are controlled by the condition of control bits in the FUNCTION REGISTER (FN). (See Section 4.5.2 for more details on PANEL operations).

4.5.1.3 Programmed I/O Operations

The operations available through PROGRAMMED I/O will be discussed in detail in Section 4.5.3 of this summary. The purpose of this section is to present a general overview of the components and operations involved in executing basic PROGRAMMED I/O transfers between the AP-120B and ADDRESSABLE I/O DEVICES.

* Theory of Operations

In contrast to the customary use of a DEVICE CODE FIELD within a given I/O instruction word, the AP-120B uses a separate eight-bit register, termed the DEVICE ADDRESS REGISTER (DA), whose current contents designate the particular I/O DEVICE involved in the current AP-120B I/O operation. Because DA is a separate register, the programmer must properly condition the contents of DA at least one instruction before initiating an I/O operation to the desired I/O DEVICE.

Generally, the designated I/O DEVICE communicates its state of availability to the AP-120B (whether or not it is ready to receive or send data) by the current state of its I/O DATA READY FLAG (IODRDY). When IODRDY of the designated I/O DEVICE is equal to 1, then the I/O DEVICE is READY for the current I/O transfer operation.

Certain Op-Codes within the AP-120B instruction set are capable of testing the state of a device's IODRDY FLAG before executing a given I/O Op-Code. When these Op-Codes are used, the AP-120B will execute a SPIN operation (See note) until IODRDY(DA) is equal to 1, at which time the specified I/O transfer will be executed.

The particular DATA PATHS employed in an I/O operation depend on which DATA-TRANSFER MODE is being used. Generally, with the exception of the DIRECT-MEMORY ACCESS MODE (DMA), data to be OUTPUT from the AP-120B must be placed onto the DATA PAD BUS (DB) where it is automatically placed onto the INBS by the AP-120B I/O structure and OUTPUT to the addressed I/O DEVICE. Data input to the AP-120B must be concurrently transferred onto DB (via use of a DATA PAD GROUP OP-CODE) from the I/O BUS (INBUS) where it is placed by the sending I/O DEVICE.

NOTE

A "SPIN" will suspend all on-going program operations within the AP-120B until the IODRDY(DA) FLAG = 1. Note, however, that an infinite SPIN loop will occur if the IODRDY(DA) never equals 1. Do not use I/O SPINS during overlapped main data accesses (See Section 4.7).

* In/Out Operations

Basically, there are four types of PROGRAM CONTROL INPUT OR OUTPUT instructions. The PROGRAMMER may: (1) unconditionally transfer DATA (via IN, OUT); (2) SPIN until the I/O DEVICE is READY (IODRDY(DA)=1), and then transfer DATA (via SPININ, SPNOUT); (3) transfer DATA then change the contents of DA (INDA, OUTDA); or (4) SPIN until IODRDY(DA)=1, then transfer DATA, and then change the contents of DA (SPINDA, SPOTDA). (See IN/OUT field for more details.)

Examples of typical non-DMA PROGRAMMED I/O operations are given below:

Assume the S-PAD REGISTER 5 equals the desired I/O DEVICE ADDRESS.

TO INPUT DATA INTO AP-120B:

ASSEMBLER FORMAT	(MEANING)
MOV 5,5; LDDA;DB=SPFN	(Set up DEVICE ADDRESS)
IN; DPX(-3)<INBS	(INPUT data onto INBUS and transfer it to DPX(-3) via DB.)

or, TO INPUT BY TESTING IODRDY(DA)

MOV 5,5; LDDA;DB=SPFN	(Set-up DEVICE ADDRESS)
SPININ; DPX(-3)<INBS	(SPIN until IODRDY flag is set, then INPUT onto INBS and transfer it to DPX(-3) via DB.)

TO OUTPUT DATA AP-120B:

MOV 5,5;LDDA;DB=SPFN	(Set up DEVICE ADDRESS)
OUT;DB=TM	(Put DATA WORD onto DB, then OUTPUT. THE I/O LOGIC will place the data word onto INBS where it will be OUTPUT to the I/O DEVICE.)

TO OUTPUT BY TESTING IODRDY(DA)

MOV 5,5;LDDA;DB=SPFN	(Set-up DEVICE ADDRESS)
SPNOUT;DB=TM	(SPIN until I/O DEVICE is ready to accept the DATA WORD. When ready, the I/O logic will place the DATA WORD onto INBS where it will be OUTPUT to the I/O DEVICE.)

or, LDDA; DB=DEV	(Set device address)
OUTDA; MOV 5, 5	(Output to first device and set DA to SPFN.)

In order to execute an INPUT or OUTPUT operation, the object I/O DEVICE

must be selected by placing its DEVICE ADDRESS into the AP-120B I/O DEVICE ADDRESS REGISTER (DA). DA is an eight-bit register affording a range of 256 different I/O DEVICE ADDRESSES. The lower DEVICE ADDRESSES are dedicated in the following manner:

I/O DEVICE	DEVICE ADDRESS
HOST INTERFACE	
DMA REGISTERS:	
WORD COUNT REGISTER (WC)	0
HOST MEMORY ADDRESS REGISTER (HMA)	1
CONTROL REGISTER (CTL)(See Note 1)	2
AP-120B MEMORY ADDRESS REGISTER (APMA)	3
FORMATTER (FMT)	4
WRITABLE TABLE MEMORY (TMRAM)(See Note 2)	5
MEMORY ADDRESS EXTENSION (MAE)	30
APMA EXTENSION (APMAE)	31
MASK (including MODE and I/O)	32

ADDITIONAL DEVICE ADDRESSES are:

First IOP16	10-14
Second IOP16	20-24
Parity Option	33-37
First PIOP	100,101,110-117

NOTES

1. CTL register provides control functions for other HOST-INTERFACE functions besides DMA.

2. HHMA uses device address 5 in some systems. However, if both TMRAM and HHMA are needed, then HHMA is moved to another address which depends upon the system's configuration.

* DA Modification Instructions

The current contents of the DEVICE ADDRESS REGISTER (DA) determine which I/O DEVICE will be involved for a current AP-120B I/O operation. DA may be altered, via programmed instruction, by use of one of the following OP-CODES. (Effective as of the next instruction cycle).

OP-CODE	I/O SUB-FIELD	OCTAL VALUE	OPERATION
LDDA	LDREG	7	(DPBS)→ DA
OUTDA	INOUT	2	Perform output to current I/O DEVICE(DA) then (SPFN)→ DA
SPOTDA	INOUT	3	Spin until IODRDY(DA)=1, then OUTPUT DATA, then (SPFN)→ DA
INDA	INOUT	6	Perform INPUT from I/O DEVICE(DA), then (SPFN)→ DA
SPINDA	INOUT	7	Spin until IODRDY(DA)=1, then INPUT DATA, then (SPFN)→ DA
SNSADA	SENSE	2	Enable CONDITION "A" to IODRDY(DA), then (SPFN)→ DA
SPNADA	SENSE	3	Test loop for CONDITION "A": "A" is continually enabled into IODRDY(DA) while the AP-120B SPINS. When (IODRDY(DA)) = 1, then (SPFN)→ DA
SNSBDA	SENSE	6	Enable CONDITION "B" to IODRDY(DA), then (SPFN)→ DA
SPNBDA	SENSE	7	Test loop for CONDITION "B": "B" is continually enabled into IODRDY while the AP-120B SPINS. When (IODRDY(DA)) = 1, then (SPFN)→ DA

Additionally, certain OP-CODES within the SENSE field permit either of two condition lines (A,B) to be enabled into the IODRDY FLAG. (See SENSE field) Conditions "A" and "B" are device dependent and have no a prior descriptions.

4.5.2 Virtual Front Panel (PANEL)

4.5.2.1 General Description

The AP-120B I/O STRUCTURE contains a VIRTUAL FRONT PANEL (PANEL) consisting of three 16-bit registers - (1) SWITCHES (SWR), (2) LIGHTS (LITES), AND (3) FUNCTION (FN). The registers are under control of the HOST-CPU via HOST-INTERFACE. The HOST may examine and/or set these registers at any time, irrespective of the state of the AP-120B. The AP-120B, however, may only DEPOSIT into the LITES register and only READ the SWR (See HOSTPNL field of Special Operations group in instruction descriptions). A brief description of the PANEL registers are given below:

REGISTER	DESCRIPTION
SWITCHES (SWR) (16 Bits)	Used to enter DATA and ADDRESSES into the AP-120B. Can be read or written by HOST-CPU or read by AP-120B programmed instruction.
LIGHTS (LITES) (16 Bits)	Used to display the contents of internal AP-120B registers. Can be read by HOST-CPU and written by AP-120B programmed instruction.
FUNCTION (FN) (16 Bits)	Provides VIRTUAL FRONT PANEL control operations. Can be read or written by HOST-CPU only.

4.5.2.2 Panel Operations

The control bits of the PANEL FUNCTION REGISTER (FN) determine the current operation(s) of the VIRTUAL FRONT PANEL. The available PANEL operations may be classed into two groups - (1) AP-120B PROGRAM CONTROL, and (2) AP-120B INTERNAL REGISTER EXAMINATION and MODIFICATION.

A detailed explanation of available PANEL operations and related control bits of the FUNCTION REGISTER (FN) is given below. Unless otherwise noted, the active state of the appropriate FN control bit is a "1".

PANEL FUNCTION REGISTER FORMAT

Ø	1	2	3	4	5	6	7	8 9	10 11	12 13 14 15
STOP	START	CONT	STEP	RESET	EXAM	DEP	BREAK	INC	WORD	REGISTER SELECT

When the AP-120B is running only the STOP and RESET panel functions are valid. The other panel functions can only be exercised after the AP-120B has been halted. BREAK and REGISTER SELECT can be altered and have an effect.

PANEL OPERATIONS AND RELATED FN CONTROL BITS
AP-120B PROGRAM CONTROL

OPERATION	RELATED FN REGISTER BIT(s)	EFFECT
STOP/HALTED	BIT "0"	STOP AP-120B program execution upon completion of the current instruction. FN bit "0" reflects the current state of the AP-120B when examined by the HOST. ("1" equals AP-120B HALTED, "0" = AP-120B RUNNING). Setting this bit will stop the AP. Note that if AP-120B is currently executing a "SPIN" condition, the halt will be effective only after the "SPIN" has been completed.
START	BIT "1"	START AP-120B program execution at the program location specified by the contents of the SWITCHES (SWR) register. The preferred first instruction should be a NOP to to avoid timing sequence.
CONT	BIT "2"	CONTINUE AP-120B program execution at the program location pointed to by the current contents of the PROGRAM SOURCE ADDRESS REGISTER (PSA).
STEP	BIT "3"	Single STEP. The AP-120B instruction at the program location pointed to by the current contents of PSA will be executed. The PSA will then be advanced to point to the next program location.

RESET	BIT "4"	RESET and HALT the AP-120B immediately. Clear S-PAD Register "0" (SP(0)) and set current SPFN to SP(SPFN); clear APSTATUS Register, reset and clear MAIN DATA MEMORY TIMING. Inoperative if AP-120B caught in an I/O SPIN.
BREAK	BIT "7"	BREAKPOINT - valid only if PSA, MA, or TMA is specified in the REGISTER SELECT field (FN bits 12-15). This causes AP-120B program execution to halt when contents of selected register equal the value set into SWR register. The exact timing of a given BREAK operation is dependent on the ADDRESS REGISTER selected.

REGISTER SELECT	CORRESPONDING OCTAL VALUE IN REG SEL FIELD	EFFECT
PSA	0	Halt AP-120B program execution on next instruction following the one whose address is contained in SWR. PSA will point to the instruction following it.
MA TMA	2 3	Halt AP-120B program execution after executing the next instruction following the one that referenced the memory location whose address is contained in SWR. PSA will point to the second instruction following the one which caused the breakpoint.

AP-120B REGISTER EXAMINATION AND MODIFICATION

OPERATION	RELATED FN REGISTER BIT(s)	EFFECT
EXAM	BIT "5"	EXAMINE the register or memory selected by the octal value contained in the REGISTER SELECT field (FN bits 12-15).. Display to LITES that PORTION of the selected register or memory as determine by the WORD field (FN bits 10-11).
DEP	BIT "6"	DEPOSIT the contents of the SWITCHES (SWR) into the register register or memory selected by the value contained in the REGISTER SELECT field (FN bits 12-15). The deposit will occur to that portion of the object register or memory selected by the value contained in the WORD FIELD (FN bits 10-11).
INC	BITS 8-9	Increment either MA, TMA, or DPA- depending on current value- following completion of the other concurrently specified PANEL operations. This operation allows sequential memory locations to be examined or deposited. Note that MA pre-increments on the concurrent DEPOSIT into MD.
	INC OCTAL VALUE	ADDRESS REGISTER TO BE INCREMENTED
	0	NONE
	1	MA (MAIN DATA MEMORY ADDRESS REGISTER
	2	DPA (DATA PAD ADDRESS REGISTER)
	3	TMA (TABLE MEMORY ADDRESS REGISTER
WORD	BITS 10-11	Selects which portion of a register or memory is deposited or examined. The WORD field is used in conjunction with REGISTER SELECT (FN BITS 12-15). Portions are selected in the following manner.

WORD OCTAL VALUE IS:	REGISTER SELECTED BY CURRENT VALUE IN FN BIT 12-15 IS:		
	16 BITS OR LESS	38 BITS	64 BITS (PROGRAM SOURCE)
0	ALL	**	PS(QUARTER ZERO) (PS(Q0)); (PS(bits 0-15))
1	**	EXPONENT bits 02-11 right justified, zero filled.	PS(QUARTER ONE) (PS(Q1)); (PS (bits 16-31))
2	**	HIGH MANTISSA - MANTISSA bits 00-11; right justified, zero filled.	PS(QUARTER TWO) (PS(Q2)); (PS (bits 32-47))
3	**	LOW MANTISSA - MANTISSA bits 12-27.	PS(QUARTER THREE) (PS(Q3)); (PS (bits 48-63))

**Not applicable

REGISTER SELECT BITS 12-15 Selects which register, memory, or data path will be examined or deposited for the current PANEL OPERATION. Used in conjunction with WORD field, (FN bits 10-11) by OCTAL VALUE, in the following manner:

REGISTER SELECT OCTAL VALUE	REGISTER SELECTED (Field bit length)	COMMENTS
0	PSA (12)	PROGRAM SOURCE ADDRESS REGISTER
1	SPD (4)	S-PAD DESTINATION ADDRESS REGISTER
2	MA (16)	MAIN DATA MEMORY ADDRESS REGISTER
3	TMA (16)	TABLE MEMORY ADDRESS REGISTER
4	DPA (6)	DATA PAD ADDRESS REGISTER (DPA is 6 bits wide even though DPA has 32 registers.)

5	SPFN(16) (See Note 1)	S-PAD FUNCTION currently enabled. May be EXAMINED ONLY.
5	SP(SPD)(16) (See Note 2)	S-PAD DESTINATION REGISTER currently specified by SPD. May be DEPOSITED ONLY.
6	APSTATUS (16)	AP-120B INTERNAL STATUS REGISTER
7	DA (8)	DEVICE ADDRESS REGISTER
10	PS(TMA) (64)	PROGRAM SOURCE WORD specified by the ADDRESS currently contained in TMA.
11	INBS(DA) (38)	Read input data from I/O device specified by DA register. May be examined only.
12	0	Not applicable
13	DPX(DPA-4) (38)	DATA PAD X location specified by the current contents of DPA minus 4.
14	DPY(DPA-4) (38)	DATA PAD Y location specified by the current contents of DPA minus 4.
15	MD(MA) (16)	MAIN DATA MEMORY location specified by the current contents of MA.
16	0	Not applicable
17	TM(TMA)	TABLE MEMORY location specified by current contents of TMA. May be EXAMINED ONLY.Note one cycle delay in examination after change of TMA.

NOTES

1. Valid only during EXAMINATION operation (FN bit 5 = 1).

2. Valid only during DEPOSIT operation (FN bit 6 = 1).

General Rules - Panel Operations

* STARTING the AP-120B (via START or CONT)

The PANEL START function can be used to start the AP-120B program with the following restriction:

The FIRST INSTRUCTION executed by the AP-120B following a START command MUST NOT alter PSA in any manner other than to advance it to the next sequential instruction. The user should not use a BRANCH or JUMP instruction or any instruction within the SPEC or I/O groups of the instruction word. Accordingly, the preferred first instruction is a NOP.

The PANEL CONT function is recommended to start the AP-120B, in the following manner:

1. Set SWR to the starting address and execute a DEP into PSA.

2. Set SWR to a desired breakpoint (See note) and execute a CONT.

NOTE

This places the necessary breakpoint code into the user's program should he need to debug the program.

* STOPPING the AP-120B (via STOP or BREAK)

On all stopping operations except RESET, the AP120B will stop with PSA set to the ADDRESS of the next instruction. Since SPFN is current, it will be set according to the instruction that the PSA is currently pointing to. Otherwise, the instruction pointed to by PSA is not executed and will execute correctly when the user STEPS or CONTINUES (except for MD timing).

* MAIN DATA TIMING CYCLE

The MD memory timing is designed so as to preserve the current state of the timing sequence when a STOP or BREAK is executed. MD timing will be in its proper sequence when the user STEPS or CONTINUES.

Note that the user must not examine or alter either MD or TM, nor should he permit a DMA transfer while the AP-120B is stopped, if MD timing is suspended. TM will also be upset by any panel operation other than continue if one cycle after SETTMA, INCTMA, DECTMA or LDTMA.

To determine condition of MD memory timing: If PSA is pointing to either the first or second instruction location following a SETMA, INCMA, DECMA, or LDMA, the memory cycle is suspended in mid-operation and the restrictions stipulated in the note above apply.

* RESETTING the AP-120B (via RESET)

The user may not correctly STEP or CONT following execution of a PANEL RESET. MD memory timing is cleared and APSTATUS is reset. Also SP(0) is cleared and SPFN is set to SP(SPD) with SPD=0. Thus the S-PAD registers can be examined by setting SPD (DEP to SPD) and examining SPFN.

* STEPPING the AP-120B (via STEP)

The user may alter or examine any register or memory except for restrictions concerning MD MEMORY TIMING and TM timing as already discussed.

* EXAMINING or DEPOSITING

The user may examine and/or deposit into any available field within the restrictions concerning MD MEMORY and TM TIMING and the conditions listed below:

* The current contents of TMA are used as a pointer to indicate which PROGRAM SOURCE location is to be examined or deposited.

* TO EXAMINE SP(SPD) - since SP(SPD) may not be examined directly, the user may execute a PANEL RESET to force an SP(SPD) to SPFN and then examine SPFN (This clears SP(0)). Alternatively, this may be accomplished with the following micro-code sequence:

```
MOV #0,0
RSPFN; LDSPNL 0
HALT
NOP
```

This will set SPFN=SP(SPD) without clearing SP (0).

Note that a PANEL RESET will also clear MD TIMING and APSTATUS.

*TO EXAMINE OR DEPOSIT INTO MD - a MEMORY READ CYCLE is initiated by the user executing a DEP into MA. A MEMORY WRITE CYCLE is initiated by the user executing a DEP into MD. Therefore to deposit into a given MD location, and then examine that MD location, the following sequence must be performed.

PANEL REGISTER CONDITIONS

OPERATION	LITES	SWR	FN
(1) Set MA to location X	---	Set to address X	DEP (BIT 6) = 1 REG SEL (bits 12-15) = 2 (MA)
(2) Examine MD	Will reflect contents of MD(X)	---	Exam FN=1002(8) (bit 5) = 1 REG SEL = 15 WORD = 1, 2 or 3
(3) Deposit into MD(X) (This writes into MD at previously set MA)	--- MD(X)	Set to value desired to be deposited into WORD (bits 10-	DEP (bit 6) =1 REG SEL (bits 12-15) = 15(MD) 10-11) = 1, 2 or 3
(4) SET MA to location X	---	Set to address X	DEP (bit 6) = 1 REG SEL (bits 12-15) = 2(MA) WORD (bits 10-11) = 0

(5) Examine MD	Will reflect new contents of MD(X)	---	Exam (bit 5) = 1 REG SEL = 15 WORD = 1,2, or 3

* TO EXAMINE TM - because the TABLE MEMORY hardware requires two PANEL operations to retrieve a requested TM location, the following sequence must be performed.

PANEL REGISTER CONDITIONS

OPERATION	LITES	SWR	FN
(1) Set TMA (To initiate examine TM(X)	---	Address X	DEP (bit 6) = 1 REG SEL (bits 12-15) = 3(TMA)
(2) Set TMA ("Dummy PANEL operation to retrieve contents of TM(X))	---	Address X	Same as above
(3) Examine TM	(TM(X))	---	Exam (bit 5) = 1 REG SEL = 3 WORD = 1,2 or 3

NOTE

WRITABLE TABLE MEMORY (TMRAM) may be deposited through PANEL operations. The user must resort to program I/O Op-Codes in order to WRITE into TMRAM.

* USING THE INCREMENT FUNCTION (via INC)

This function is valid only when either Exam, Dep, RESET or START is selected in the FN. The timing of incrementation is dependent on the operation specified.

Assume INC = 1, 2, or 3

REGISTER SELECTED IN INC FIELD	PANEL OPERATION	EFFECT
DPA TMA	EXAM or DEP	DEP MD increments after the specified operation.
MA	DEP MD	Increments before executing current DEP operation.

4.5.3 PROGRAMMED I/O

ADDRESSABLE I/O DEVICES capable of I/O transfer operations via use of the AP-120B PROGRAMMED I/O may be classed in the following manner:

A. HOST INTERFACE and related REGISTERS

B. Other ADDRESSABLE I/O DEVICES

4.5.3.1 HOST-INTERFACE and related REGISTERS

* General Description

The HOST-INTERFACE acts as a buffer between AP-120B and HOST-CPU. The HOST-INTERFACE is capable of both programmed-control I/O (See Note 1) and DIRECT MEMORY ACCESS (DMA) operations. It consists of the following AP-120B ADDRESSABLE REGISTERS:

DEVICE	AP-120B DEVICE ADDRESS
WORD COUNT REGISTER (WC) (16 bits)	0
HOST MEMORY ADDRESS REGISTER (HMA) (16 bits)	1
CONTROL REGISTER (CTL) (16 bits)	2
AP-120B MEMORY ADDRESS REGISTER (APMA) (16 bits)	3
FORMATTER (FMT) (38 bits)	4
High Host Memory address (HHMA)(2-4 bits)(See Note 2)	5

NOTES

1. PROGRAMMED CONTROL I/O - I/O data transfer operations are accomplished through sequential program instruction execution. Although the DMA control registers may be accessed through PROGRAMMED CONTROL I/O, the actual DMA transfer operation when enabled is autonomous.

2. HHMA is applicable only for those HOST CPU's having more than 16 bits of DMA address.

* Control Register (CTL)

CTL is a 16-bit I/O ADDRESSABLE REGISTER (DA = 2). CTL contains the condition bits that control most HOST-INTERFACE-related I/O operations. CTL can be read anytime by the executing AP-120B program without regard to HOST-CPU activity. CTL can be written by either the HOST-CPU or AP-120B. However, if both attempt to write CTL at the same time, the HOST-CPU will be given priority.

The operations directed by CTL fall into the following groups:

1)Mode and direction of transfer (DMA or PROGRAMMED - CONTROL I/O)

2) Type of DATA FORMAT and I/O PATH SELECTION (via FORMATTER)

3) Transfer STATUS and ERROR BITS, and

4) PROGRAMMED-INTERRUPT enable bits

CONTROL REGISTER FORMAT

Ø	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
WC=Ø	INTR AP	IAP WC	IH HALT	IH WC	IH ENB	FERR	DLATE	CC	AP DMA	WRT HOST	DEC APMA	DEC HMA	FMT		HDMA START

A description of CTL bit operations is given below. Further details of particular CTL bit functions will be discussed in sections related to the I/O DEVICES which are controlled or monitored by the CTL register. All bits are READ/WRITE except as noted.

Bit 0 WC=0 Indicates that the Word Count Register is zero. Note that WC is decremented only during DMA transfers to/ from Host Memory. Bit 0 is a Read only bit. It should not be used to monitor DMA activity.

Bit 1 INTRAP Set the INTRQ (Interrupt Request) flag in the AP-120B.

Bit 2 IAPWC Set INTRQ (Interrupt Request) flag in the AP-120B when the DMA transfer is done.

Bit 3 IHALT Enable a Host Interrupt when the AP-120B halts.

Bit 4 IHWC Enable a Host Interrupt when the DMA transfer is done.

Bit 5 IHENB Interrupt Host Enable. Interrupt Host if AP-120B attempts to set this bit, or if AP-120B executes an "INTEN" instruction (See I/O field). This bit can actually be written only by the Host.

Bit 6 FERR Format Error. Indicates that EXPONENT UNDERFLOW or OVERFLOW occurred in conversion from AP-120B Format to Host Floating-Point Format. (Read only) Can be set only during transfers with CTL10=1 and FMT=2 or 3. Due to the Pipeline nature of the FORMATTER, the next two words past the end (before the beginning if CTL11= 1) of the DATA ARRAY will be loaded into the FORMATTER at the end of the transfer. In order to avoid spurious setting of FERR, the programmer must insure that these two words are either zero or are FPN's within the allowable dynamic range of formats 2 or 3.

Bit 7 DLATE Data Late. Indicates that the AP-120B did not empty/ load the FORMAT BUFFER before the Host attempted to reload/read it. On some Hosts this bit also indicates an attempt to access non-existent Host memory. In the latter case, the DMA transfer is terminated (PDP-11 only). Bit 7 is a read-only bit that is cleared by an interface reset or by setting HDMA START (CTL 15). It is not cleared by a panel Reset.

Bit 8 CC Consecutive Cycle. Block DMA transfers to/from Host memory will occur without interruption. On typical Hosts, the Host CPU will be locked out but other higher priority DMA devices will still have access to Host memory. Some Host interfaces are equipped with a hardware selectable consecutive cycle counter that limits the number of cycles stolen in a burst.

Bit 9 APDMA Allows the interface to perform DMA transfers to/ from AP-120B memory. Depending on the direction of transfer, a MAIN DATA MEMORY cycle is initiated every time the Host finishes reading or loading the format register, whether via DMA or program control. On the AP-120B side, the format register is loaded from the MAIN DATA BUS instead of the DATA PAD BUS.

Bit 10 WRTHOST Write to Host. This bit controls the direction of transfer. If set, data is read from the Ap-120B, passed through the FORMAT REGISTER, and written to the Host. If clear, the direction of transfer is reversed.

Bit 11 DECAPMA Decrement APMA. If set, APMA is decremented during DMA transfers to/from Ap-120B MAIN DATA MEMORY. If clear, APMA is incremented.

Bit 12 DECHMA Decrement HMA. If set, DMA is decremented during DMA transfers to/from Host memory. If clear, HMA is incremented.

Bits 13 & 14	FMT	Format Register Control. (See FORMATTER, next section).
Bit 15	HDMA start/ busy	Host DMA Start. Initiate DMA transfers to/from Host memory. When read, the state of this bit reflects the status of the Host DMA activity ("1" if active, "0" if inactive). Transfers continue until WC=0. Writing a "0" in this bit with DMA active has no effect on the transfer as long as the state of CTL (bits 8-14) is not changed. The programmer should not, under any circumstances, write a "1" in this bit with the DMA active. This is the bit that should be read in order to monitor DMA activity.

* Formatter (FMT)

The FORMATTER consists of the following:

* The necessary decoding logic to interpret the types of ongoing I/O transfer (via CTL bits 9, 10, 12, 13 and 14)

* A 38 bit double-buffered register, and

* An I/O DATA READY FLAG for AP-120B operations

FMT has an I/O DEVICE ADDRESS of 4

The FORMATTER is a HOST-INTERFACE transfer buffer which under the direction of selected bits of the CONTROL REGISTER (CTL), performs the following functions in order to coordinate the I/O transfer operations specified (whether via PROGRAMMED I/O or DMA):

1) FORMAT CONVERSION

2) INPUT BUS SELECTION

3) TRANSFER-DIRECTION CONTROL and ORDER of DATA-WORD ASSEMBLY

4) I/O TRANSFER TIMING COORDINATION via its IODRDY flag

(1) FORMAT CONVERSION

The following FORMAT CONVERSION operations may be performed for a given I/O TRANSFER between the AP-120B and the HOST-CPU. Note that the function selected is entirely dependent on the conditions of bits 13 and 14 of the HOST-INTERFACE CONTROL REGISTER (CTL). The programmer must insure that the CTL bits are configured in a manner consistent with the particular I/O transfer being executed.

VALUE in CTL Bits 13-14	FORMAT CONVERSION OPERATION
0	32-bit integer. No format conversion. Used to transfer integers of half-words.
1	16-bit integer. 16-bit integers from the Host are converted to unnormalized 38-bit AP-120B FPN's. Low 16-bits of AP-120B FPN are sent to Host.
2	Conversion of "SIGN-MAGNITUDE MANTISSA with BINARY EXPONENT" format to/from AP-120B Floating Point format. Includes logic to handle "Phantom Bit" and TWO's COMPLEMENT formats (See Notes 1 and 2).
3	Conversion of IBM 32-bit format to/from AP-120B format. IBM format can be specified to be either SIGN-MAGNITUDE or TWO's COMPLEMENT (See Notes 1 and 2).

NOTES

1. For format types 2 and 3, the FORMATTER has the necessary logic to detect OVERFLOW and UNDERFLOW on conversion from AP-120B format and to force a signed maximum quantity on OVERFLOW or Floating-Point Zero on UNDERFLOW.

2. Operation may vary depending on Host-CPU used. Generally, one of either format 2 or 3, will be adapted to the single precision Floating-Point format of the Host in question.

(2) INPUT BUS SELECTION

One of the following formatter input busses is selected by FMT depending on the I/O operation being performed (as determined by CTL bits 9 and 10).

CTL BIT 9	CTL BIT 10	INPUT BUS SELECTED BY FMT
0 OR 1	0	HOST DATA BUS
0	1	AP-120B I/O BUS (INBUS)
1	1	AP-120B MAIN DATA OUTPUT

(3) DIRECTION or TRANSFER and ORDER of DATA-WORD ASSEMBLY

Bit 10 of CTL (WRTHOST) determines which direction the FMT will transfer. (When WRTHOST = 1, the transfer is to HOST=CPU; when WRTHOST = 0, then the transfer is to AP-120B).

The FMT may assemble bi-directional 16-bit words from the HOST-CPU before transferring them to the AP-120B, (depending on the format selected via CTL bits 13 and 14). The order in which the half-words are assembled is dependent on the condition of CTL bit 12 (DECHMA). If DECHMA = 1, the HOST-CPU DMA I/F is assumed to be going backwards through host memory and the FMT will expect to receive a LOW WORD followed by a HIGH WORD and will assemble the two words accordingly. If DECHMA = 0, then the order of assembly is reversed (See Note).

NOTE

FLOATING POINT NUMBER ARRAYS are always expected to be stored in FORWARD ORDER; that is, with the LOW WORD portion of the DATA-WORD stored in the next higher numbered host memory location than the HIGH WORD portion.

(4) I/O TIMING COORDINATION via IODRDY

FMT will generate an I/O READY response (via IODRDY = 1) to an appropriate AP-120B IN/OUT or SENSE INSTRUCTION (with DA = 4). If IODRDY = 1, then the FMT is ready to either:

(1) Receive the DATA-WORD from the AP-120B (when CTL bit 10 (WRTHOST = 1), or

(2) Send FORMATTED DATA to the AP-120B (when CTL bit 10 (WRTHOST) = 0).

Note that to insure correct execution of a given I/O transfer operation, all functions of FMT must be properly selected. (i.e., all appropriate CTL bits must be correctly conditioned).

* Direct Memory Access (DMA)

The AP-120B is capable of both STANDARD-DMA and PARTIAL-DMA operations between the AP-120B and HOST-CPU. STANDARD DMA is defined as autonomous (outside of programmed - I/O) block transfers between AP-120B MAIN DATA MEMORY (MD) and HOST-CPU MEMORY, via HOST- INTERFACE FORMATTER (FMT). Partial DMA is a mixed mode of DMA and PROGRAMMED - I/O, available in the following combinations:

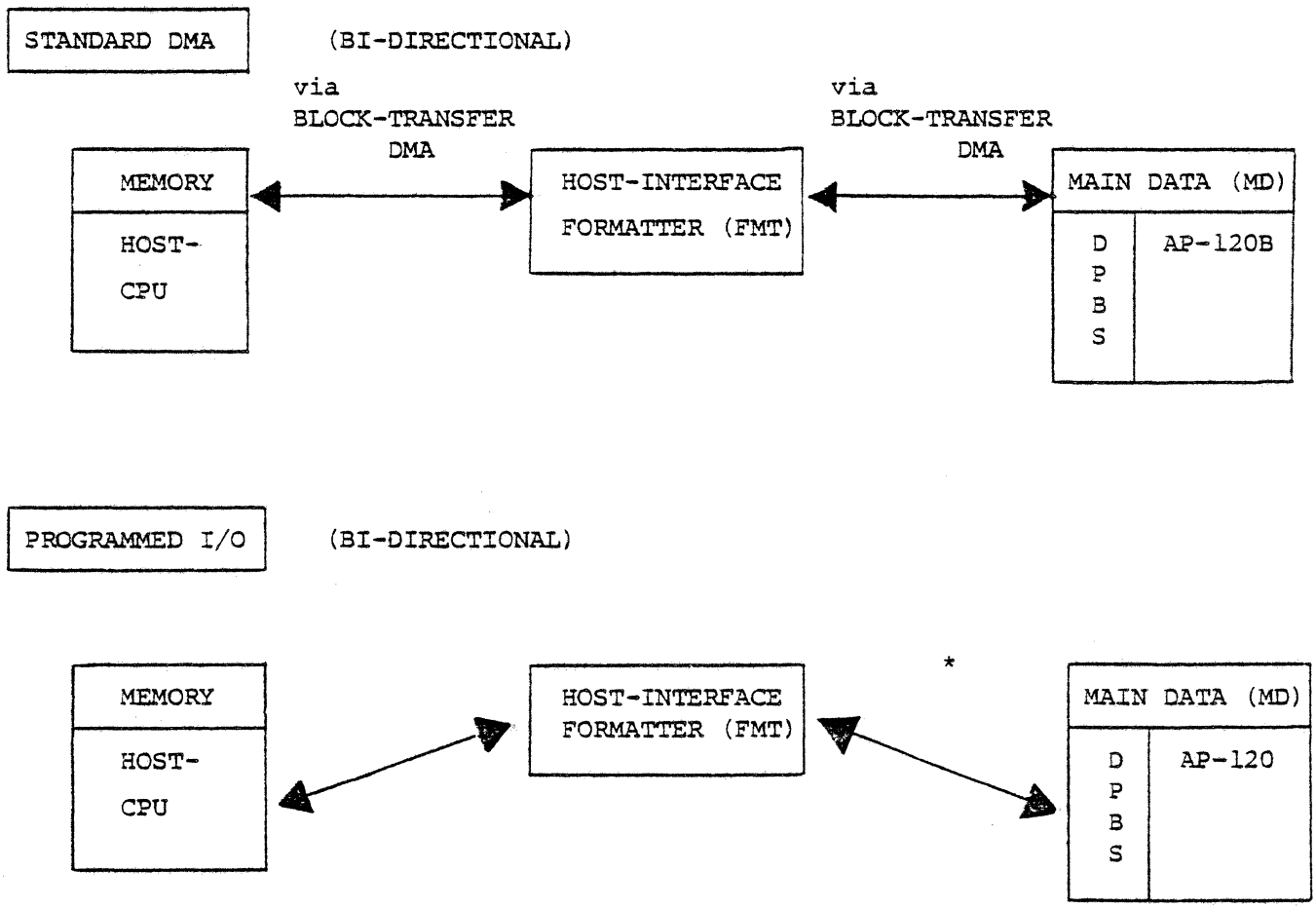
1) HOST-CPU MEMORY via DMA to FMT, via PROGRAMMED - I/O to AP-120B

2) HOST-CPU via PROGRAMMED I/O to FMT, via DMA to AP-120B MAIN DATA MEMORY (MD).

3) AP-120B MD via DMA to FMT, via PROGRAMMED I/O to HOST-CPU.

4) AP-120B PROGRAMMED I/O to FMT, via DMA to HOST-CPU MEMORY.

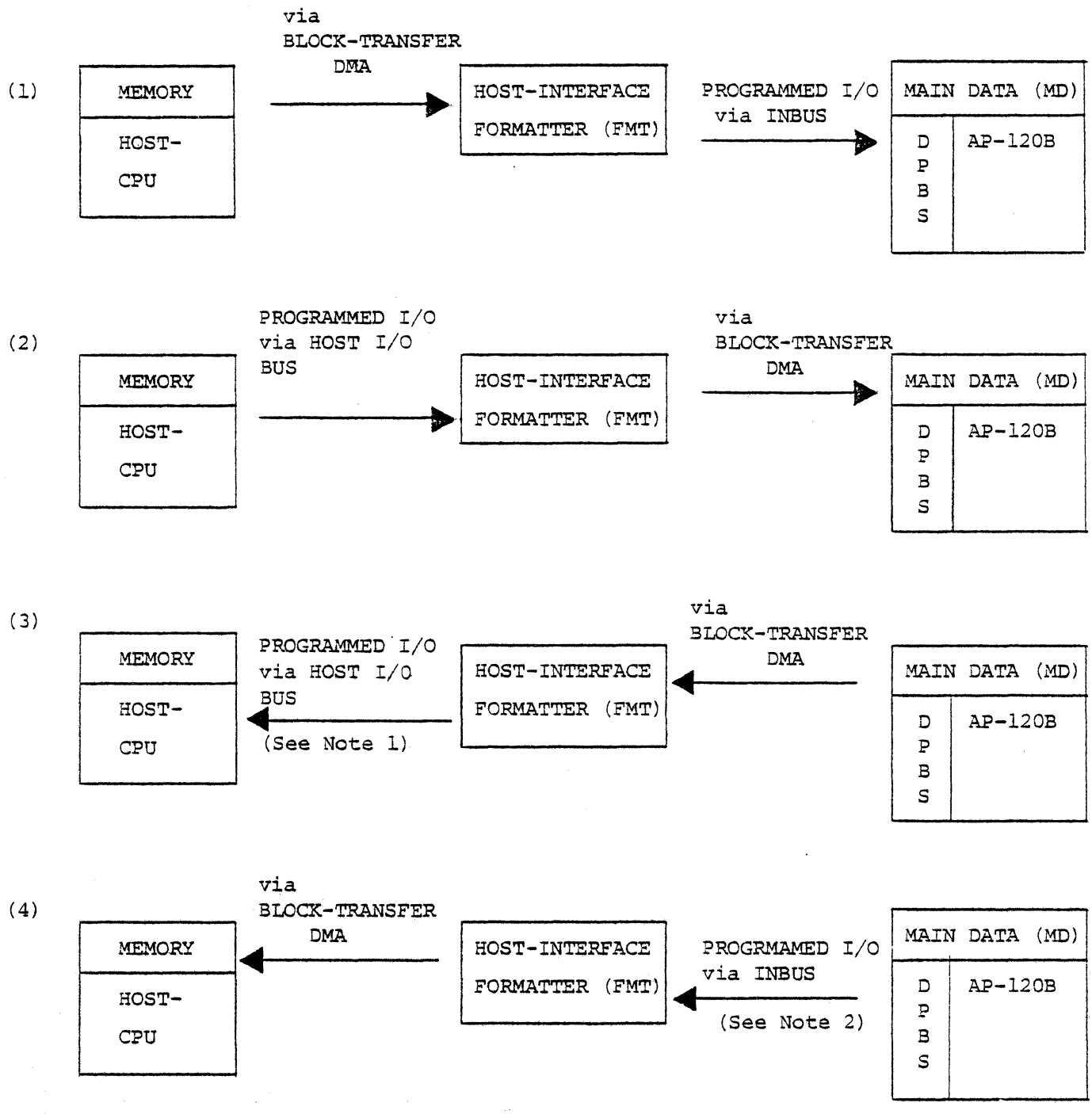
The particular mode executed depends on the configuration of bits 8-15 of the CTL register. (See, CONTROL register, this summary).



* At end of AP-120B-TO-HOST Transfer, AP must transfer two more words than Host.

PARTIAL DMA

FOUR TYPES:



NOTES

1. First two arguments read by host (32 bits total in Format 1, 64 bits total in Formats 0, 2, 3) must be discarded.

2. AP-120B must transfer two more words than word count at end of transfer.

DMA - Related Registers

The following AP-120B ADDRESSABLE Registers provide the necessary information and control bits for DMA execution. If the programmer wishes to control or initiate host DMA transfers from within the AP-120B, he must properly condition the applicable bits of these registers.

DMA-RELATED REGISTER	DEVICE ADDRESS	DMA-RELATED BITS
CONTROL (CTL) (See CONTROL REGISTER, this summary, for more details)	2	Bit 8 (CC) Bit 9 (APDMA) Bit 10 (WRTHOST) Bit 11 (DECAPMA) Bit 12 (DECHMA) Bit 13 & 14 (FMT) Bit 15 (HDMA START/BUSY)
WORD COUNT (WC)	0	All: Contains the value representing the length of block measured in Host CPU words to be transferred in current DMA operation. (1 to 65,535 Host words).
HOST MEMORY ADDRESS (HMA)	1	All: Contains the HOST MEMORY ADDRESS at which DMA transfer will begin (applicable only in STANDARD DMA or PARTIAL DMA types "1" and "4")
AP-120B MEMORY ADDRESS (APMA)	3	All: Contains AP-120B MEMORY ADDRESS at which DMA transfer will begin (applicable in STANDARD DMA and PARTIAL DMA types "2" and "3").

The specific operations required for HOST-CPU PROGRAMMED I/O depend entirely on the type of HOST-CPU being used. Programming examples for AP-120B PARTIAL-DMA (1 and 4 above) are presented in the PROGRAMMING EXAMPLE section of this summary. (e.g., BOOTSTRAP example).

DMA Operations

Generally, DMA operations are executed in the following manner:

(1) Bits 8-15 of CTL determine the particular mode, format, and direction of DMA operation to be performed.

(2) The value placed in WC determines the length of block to be transferred (number of words).

(3) Addresses placed in either HMA or APMA (or both, depending on mode of DMA selected) indicate respective MEMORY ADDRESSES at which the transfer is started.

Once the DMA is initialized (CTL bit 15 = "1"), the AP-120B will raise its DMA REQUEST to the HOST-CPU. Once the DMA REQUEST is granted, the AP-120B will transfer/receive in the following manner;

a. DMA will transfer one word (See Note), then -

b. If CC=0 (Bit 08), the AP-120B will drop its request for one HOST-CPU MEMORY cycle, allowing lower priority devices the opportunity to request and receive HOST-CPU channel priority.

c. One cycle later, the AP-120B will again raise its request to HOST-CPU (locking out all other lower priority devices) and will again transfer one word.

d. The process will continue until the DMA tranfer is completed (WC = 0) or until a programmed RESET prematurely terminates the operation. Additionally, FERR (CTL bit 06) will be set if EXPONENT OVERFLOW or UNDERFLOW occurred during the transfer operation.

NOTE

If DMA is the consecutive cycle mode (CTL bit 8 - "1") then DMA will transfer the number of words indicated by the value selected on the DMA "HEX" hardware switch, if applicable, before dropping its request for one cycle.

* Programmed Interrupts

The AP-120B is capable of generating the following interrupts:

a. AP-120B INTERNAL INTERRUPTS
   (1) UNCONDITIONAL
   (2) DMA INACTIVE

b. INTERRUPTS TO HOST-CPU
   (1) AP-120B HALTED
   (2) DMA INACTIVE
   (3) CONTROL BIT "5" (CTL05)

* AP-120B INTERNAL INTERRUPTS

AP-120B INTERNAL INTERRUPTS, when active, set the INTERRUPT REQUEST FLAG (INTRQ) to "1". INTRQ may then be tested and branches made by use of the BINTRQ Op-Code. (See BRANCH).

Both INTERNAL INTERRUPTS initially require an appropriate enabling bit set to "1" in the CONTROL REGISTER (See Note).

* The UNCONDITIONAL interrupt requires CTL bit "1" (INTRAP) be set. If INTRAP is set, the AP hardware will unconditionally set the INTRQ flag.

* The DMA INACTIVE interrupt requires CTL bit "2" (IAPWC) to be set as an enable. With IAPWC set to "1" when HDMA START=0 (indicating that DMA is inactive), the INTRQ flag will be set.

INTRQ will be true as long as the conditions for either interrupt stay valid.

INTERRUPT	CTL ENABLING BIT	OTHER CONDITIONS	EFFECT
UNCONDITION	Bit "1" = "1" (INTRAP)	none	Set INTRQ to "1"
DMA DONE	Bit "2" = "1" (IAPWC)	HOST DMA INACTIVE	Set INTRQ to "1"

NOTE

Additionally, AP-120B INTERNAL INTERRUPTS require PRIORITY GRANTED via the INTPIN enable line. (Applicable only if there are other AP-120B I/O devices in the hardware configuration). The host interface has the lowest interrupt priority.

Example:

To set UNCONDITIONAL INTERRUPT: (effectively using CTL01 as a program flag)

ASSEMBLER FORMAT	(MEANING)
LDDA; DB=2	(Set up DA for CTL)
OUT; DB=40000	(Write CTL bit "1"(See Note) INTRQ is set to "1")
o	
o	
o	
o	
BINTRQ SVCRT	(Branch to SVCRT (service routine) if INTRQ = 1)

To set DMA INACTIVE INTERRUPT:

LDDA; DB=2	(Set up DA for CTL)
OUT; DB=20000	(Write CTL bit "2"(See Note) - when DMA DONE, then INTRQ is set to "1").
o	
o	
o	
o	
BINTRQ SVCRT	(Branch to SVCRT (service routine) if INTRQ = 1) branch occurs only if DMA inactive.

NOTE

The PROGRAMMER may destroy existing status bits in CTL when writing new bits. To avoid this, he should first examine the bits of CTL and, if needed, rewrite existing bits except for CTL15 back into CTL along with the desired enable bit. This procedure is not advisable if the Host CPU may be writing CTL at the same time.

* Interrupts To Host-CPU

THE AP-120B is capable of generating three interrupts to HOST-CPU. Like the AP-120B INTERNAL INTERRUPTS, HOST-CPU interrupts require prior conditioning of the appropriate enable bits in the CONTROL REGISTER (CTL) in addition to the particular condition being true.

HOST-CPU INTERRUPTS

INTERRUPT	CTL ENABLING BIT	OTHER CONDITIONS	EFFECT (See Note 1)
AP-120B HALTED	Bit "3" = 1 (IHALT)	AP-120B HALTED	Cause HOST-CPU INTERRUPT
DMA INACTIVE	Bit "4" = 1 (IHWC)	DMA INACTIVE	Cause HOST-CPU INTERRUPT
CTL05 (See Note 2)	Bit "5" = 1 (IHENB)	AP attempted to set bit 5 or executed IHFNB	Cause HOST-CPU INTERRUPT

NOTES

1. The specific operation of interrupt generation depends on the particular HOST-CPU and HOST-INTERFACE employed.

2. CTL05 interrupt will generate a HOST-CPU INTERRUPT in the following manner:

    a. HOST-CPU must have previously set CTL05 to "1".

    b. The AP-120B attempts to set CTL05 by use of an INTEN Op-Code (See CONTROL) or by writing a 2000(8) into CTL.

    c. A HOST-CPU interrupt will be generated.

In order to facilitate repeated use of the CTL05 INTERRUPT, the method used to clear the CTL05 interrupt condition is for the host to load the FN register. In most HOST-CPU I/F's, any WRITE to FN automatically clears a CTL05 interrupt condition. The AP-120B can also clear the interrupt condition by trying to WRITE a "0" into CTL05. The state of CTL05 will not be changed by the AP-120B action.

Example:

CTL05 INTERRUPT APPLICATION

HOST-CPU PROGRAM	AP-120B PROGRAM	(MEANING)
Stipulated Condition	o	
(Sets CTL05)	o	
o	o	
o	o	
o	o	
o	o	
	INTEN	(INTEN attempts to set
	o	CTL05; generates HOST-
	o	CPU INTERRUPT)
	o	
	or,	
	LDDA, DB=2	"set up DA
	OUT; DB=2000	"attempt to set CTL05

Host-CPU Interrupt Service Routine

Handles interrupt; then clears CTL05 interrupt condition by an I/O output to FN, thus enabling another CTL05 interrupt. The suggested DATA-WORD to be written into FN is zero or one that restores FN bits 7 to 15.

Host Interrupt Handler Programming Considerations

In most host interfaces the following protocol will insure that all three AP to host interrupts (HALT, DMA DONE and CTL05) can be serviced without danger of a "lost" interrupt:

a) Poll the two visible interrupting conditions in a fixed order; e.g., AP halted and Halt interrupt enabled first followed by DMA inactive and DMA interrupt enabled. If the condition and the interrupt enable are both true then perform the required service.

b) If no service was required by the two visible conditions then the interrupt was due to the AP-120B having set CTL bit with IHENB set and this interrupt should be given its required service. This service should include a reset of the bit 5 condition by writing the FN register.

c) Clear all three interrupt enables in the CTL register.

d) Re-enable only those interrupts that were enabled at entry to the routine and were NOT serviced in step a) above. Be careful here not to set the HDMA START bit in the CTL register. Writing a zero into HDMA START will not affect a DMA transfer that is in progress as long as the other DMA control bits (CTL08 to CTL14) are not altered. IHENB can be re-enabled if step b) cleared the condition via a write FN register.

The net result of steps c) and d) is to clear the interrupt request logic thus allowing another interrupt to occur when another condition comes true. This technique will work even if the condition came true while in the service routine.

The routines that initiate DMA and AP-120B processor action will have to enable the appropriate interrupts AFTER starting the DMA or the AP-120B processor. This is now possible since the interrupt enables in the CTL register can now be safely modified while the DMA is running as long as a zero is written into HDMA START. Again, IHENB can be left on if the interrupt service routine uses a write to the FN registers to clear the CTL05 condition. If the service routine does not clear the CTL05 condition then IHENB can be set any time following the first write FN register command. The start routine must be careful to inhibit all AP-120B interrupts (set PDP-11 processor priority at or above level 4) when modifying the CTL register.

4.5.3.2 Addressable I/O Devices

The AP-120B I/O STRUCTURE may perform transfer operations with up to 256 I/O DEVICES addressable through use of the eight bit DEVICE ADDRESS REGISTER (DA). The first 24 addresses are dedicated to HOST- INTERFACE DEVICES (DA 0-4,6) to WRITABLE TABLE MEMORY-TMRAM (DA 5); MEMORY BANK SELECT (30,31), first IOP16, 10 to 14, and second IOP 16, 20 to 24.

* TMRAM

With respect to writing operations, TMRAM is essentially an I/O DEVICE whose address is 5. To WRITE into TMRAM, one must set DA to 5 at least one cycle before outputting data to TMRAM. For example, in order to WRITE the current FA into TMRAM location 100(8), the following program would be specified:

Assume S-PAD REGISTER 6 equals "100(octal)"

```
ASSEMBLER FORMAT                    (Meaning)
  o
  o
DPX(0)<FA                           Save FA in DPX(0)
LDDA; DB=5                          (Select TMRAM as I/O DEVICE(DA))
MOV 6,6; SETTMA; DB=DPX(0); OUT     (Set TMA to"100(octal), WRITE
  o                                 DPX(0) into TMRM 100(octal))
  o
  o
```

Because of the TM WRITE operation, two cycles later the out-put of TMREG is undefined. Three cycles later (assuming no change in TMA) the contents written into TMRAM will be available as the TM OUTPUT (TM).

* Other Addressable I/O Devices

A variety of I/O DEVICES may be added to the AP-120B, depending on user application and HOST-CPU to AP-120B CONFIGURATION.

4.5.4 Programming Example

The example given in this section shows how the AP-120B BOOTSTRAP program is loaded and used along with the HOST-DMA to store a given program into AP-120B PROGRAM SOURCE MEMORY. The example nicely illustrates the various functions of the AP-120B I/O structure.

AP-120B BOOTSTRAP

Essentially, the AP-120B BOOTSTRAP is loaded and executed in the following manner (See Table 4-2):

(1) The three-word BOOTSTRAP program is loaded into the AP-120B by the HOST-CPU using AP-120B PANEL operations.

(2) The BOOTSTRAP is started by HOST-CPU --again using the PANEL.

(3) With the AP-120B BOOTSTRAP running, a HOST-DMA is initiated. The HOST-DMA transfers 16-bit program quarter-words to the FORMATTER (FMT). FMT assembles them into 32-bit half-words and signals the AP-120B when it is ready. The AP-120B BOOTSTRAP then stores the half-word into the appropriate portion of the PROGRAM SOURCE MEMORY.

(4) The process continues. When the HOST-DMA is finished, the HOST-CPU is interrupted. (If interrupts are not used the HOST-CPU may test the CTL REGISTER-bit 15 to determine when the DMA is done.)

(5) The AP-120B is then reset (HOST ABORT function). The AP-120-B may then be started again by the HOST-CPU at the starting address of the newly loaded program.

```
        BOOT-LOADS INSTRUCTION HALF-WORDS FROM THE HOST DMA
                        INTO PROGRAM MEMORY

                                $LOC 0

000000    000003    BOOT:     LDDA; DB=4        "SET TO HOST DMA
          107000                                 FORMATTER
          002000
          000004

000001    011363    LOOP:     SPININ;           "SPIN (WAIT) UNTIL A
                                                  WORD IS READY
          145000              DB=INBS;          "PUT THE WORD ONTO DB
          001000              LPSLT             "STORE IT INTO THE
                                                  LEFT HALF OF PS(TMA)
          000000
```

```
000002    011367              SPININ;          "WAIT FOR THE DMA
          145117               DB=INBS;        "GET IT IN
          001000               LPSRT;          "STORE IT INTO THE
                                                 RIGHT HALF OF PS(TMA)
          000000               INCTMA;         "INCREMENT THE
                                                 POINTER (TMA)
                               BR LOOP         "BRANCH BACK FOR MORE

                    "
                    "
                    "
                    "
                    "CALLER-CALL AN AP-120B SUBROUTINE FROM THE HOST
                      COMPUTER

000003    000003    CALLER:   DPX<ZERO;       "GET ZERO INTO DPX
          174000              REFR            "MEMORY REFRESH SYNCH
          040004
          000000

000004    000001              FMUL DPX,DPX;   "CLEAR THE MULTIPLIER
          122000               FADD DPX,DPX   "AND THE ADDER
          000400
          012400

000005    000001              FMUL DPX,DPX;   "PUSH THE PIPELINES
          122000               FADD DPX,DPX
          000400
          012400

000006    011027              FMUL DPX,DPX;   "AND AGAIN FOR FMUL
          106000               LDAPS;         "CLEAR THE STATUS
                                                 REGISTER
          000400               DB=ZERO
          012400               JSRT           "GO DO THE SUBROUTINE

000007    000003              HALT            "HALT ON RETURN FROM
                                                 THE SUBROUTINE
          170000
          000000
          000000

000010    000000              NOP
          000000
          000000
          000000
                              $END
```

Table 4-2 LOADING AND EXECUTING AP-120B BOOTSTRAP

OPERATION	HOST-CPU	COMMENTS
Load AP-120B BOOTSTRAP via PANEL (See Note)	(1) 0→ SWR 1003(octal) → FN	Deposit 0 into TMA (TMA is the pointer for depositing into PS)
(BOOTSTRAP is loaded into PS locations 0, 1, and 2)	(2) (Bootstrap word bits 0-15) → SWR 1010(octal) → FN	(Deposit first quartile of bootstrap word into QUARTER ZERO of PS(TMA))
	(3) (Bootstrap word bits 16-31) → SWR 1030(octal) → FN	(Deposit second quartile of bootstrap word into QUARTER ONE of PS(TMA))
	(4) (Bootstrap word bits 32-47) → SWR 1050(octal) → FN	(Deposit third quartile of bootstrap word into QUARTER TWO of PS(TMA))
	(5) (Bootstrap word bits 32-47) → SWR 1370(octal) → FN	(Deposit fourth quartile of bootstrap word into QUARTER THREE of (PS)TMA. TMA is then incremented by "1" to point to the next PS location.
	{Repeat steps 2-5 for remaining bootstrap words}	

EXECUTING THE BOOTSTRAP

OPERATION	HOST-CPU	AP-120B	COMMENTS
Indicate to AP-120B where to start loading selected program. (Assume 200 is PS address where program is to be loaded)	200(OCTAL) → SWR 1003(octal) → FN		HOST-CPU deposits 200 into TMA (via panel)
Start AP-120B BOOTSTRAP PROGRAM	0 SWR 1000(octal)→ FN = Dep PSA 20000(octal) → FN = Continue	BOOTSTRAP is started-AP is waiting for first program half-words to be transferred.	Starts program at PS location 0 (BOOTSTRAP) via Panel Continue
Start PARTIAL-DMA from HOST MEMORY to HOST-INTERFACE FORMATTER (FMT) (assume program to be transferred to AP-120B is now stored in HOST MEMORY location 1000 and that the program to be loaded is 200(octal) AP program words (1000(octal) 16-bit host words).)	1000(octal) → HMA 1000(octal) → WC 4201(octal) → CTL	AP is waiting	Set HOST-DMA address to 1000 Set WORD COUNT to 1000 (assume 16-bit HOST-word) (Initialize HOST-DMA to FMT in consecutive cycle mode. When done, send interrupt to HOST-CPU when HOST-DMA is done.)

EXECUTING THE BOOTSTRAP (CONTINUED)

OPERATION	HOST-CPU		AP-120B	COMMENTS
HOST-CPU is transferring 16-bit words to FMT, via DMA. FMT is assembling them into 32-bit words and when ready, signals AP-120B BOOTSTRAP program. BOOTSTRAP then loads formatted word into PS at location TMA in the following manner: * First 32-bit word goes into PS - left half, * Second 32-bit word goes into PS - right half.	HOST-CPU is transferring 16-bit words from HOST-MEMORY to FMT.		AP BOOTSTRAP LDDA; DB=4	Set DEVICE ADDRESS to (FMT)
		LOOP:	SPININ, DB=INBS; LPSLT SPININ; DB=INBS; LPSRT; INCTMA; BR LOOP	Spin until first half word is ready, then store it into PS(LH)TMA. Spin until second half word is ready, then store it into PS(RH)TMA; then increment TMA and branch back to LOOP.

When HOST-DMA is completed, DMA INACTIVE INTERRUPT will be sent to HOST-CPU. HOST-CPU will respond to the interrupt by stopping the AP-120B BOOTSTRAP as follows:

RESET AP-120B		Reset AP-120B using I/O reset command
Set TMA to desired AP-120B starting address	200(octal) → SWR 1003(octal) → FN	Deposit starting location 200 into TMA
Start newly loaded program running in AP-120B	3(octal) → SWR 1000 (octal) → FN 20000(octal) → FN	Continue at AP-120B PROGRAM SOURCE location 3 4 using caller to clear pipelines and APSTATUS

NOTE

The specific OP-CODES usd by the HOST-CPU depends on the type of HOST-CPU used.

4.6 DATA PAD SUMMARY

Discussion of the DATA PAD GROUP (DP) will be presented in the following manner:

1. General Description and Theory of Operation
2. DP Group Operations
3. DP Addressing
4. Programming Examples

4.6.1 General Description, Theory of Operation

DATA PAD (DP) is a block consisting of two high-speed accumulator files, termed DATA PAD X (DPX) and Data PAD Y (DPY), respectively (See Figure 4-4).

Each file contains thirty-two, 38-bit accumulators. Both files share a common address pointer, termed the DATA PAD ADDRESS REGISTER (DPA). DPA, along with one of four BIASED index field XR=X Read Index, YR=Y Read, XW=X Write, YW=Y Write, is used to determine the EFFECTIVE ADDRESS (EFA) for any given DP READ or WRITE operation (See Section 4.6.4 for more details on DP addressing).

Data to be stored into DATA PAD may come from the currently available FLOATING ADDER result (FA), the currently available FLOATING MULTIPLIER result (FM), or one of eight possible sources enabled onto the DATA PAD BUS.

Data READ from DATA PAD may go to the FLOATING MULTIPLIER OPERAND REGISTERS (M1, M2), the FLOATING ADDER OPERAND REGISTERS (A1, A2), the MAIN DATA MEMORY INPUT REGISTER (MI), or onto the DATA PAD BUS (DB).

The DATA PAD REGISTERS behave like true accumulators in that the contents of one register can be read out and written into in the same instruction without conflict. The WRITE takes place at the very end of the instruction cycle so that the old contents can be read out for use during the cycle and the updated contents will be available during the immediately succeeding instruction cycle.

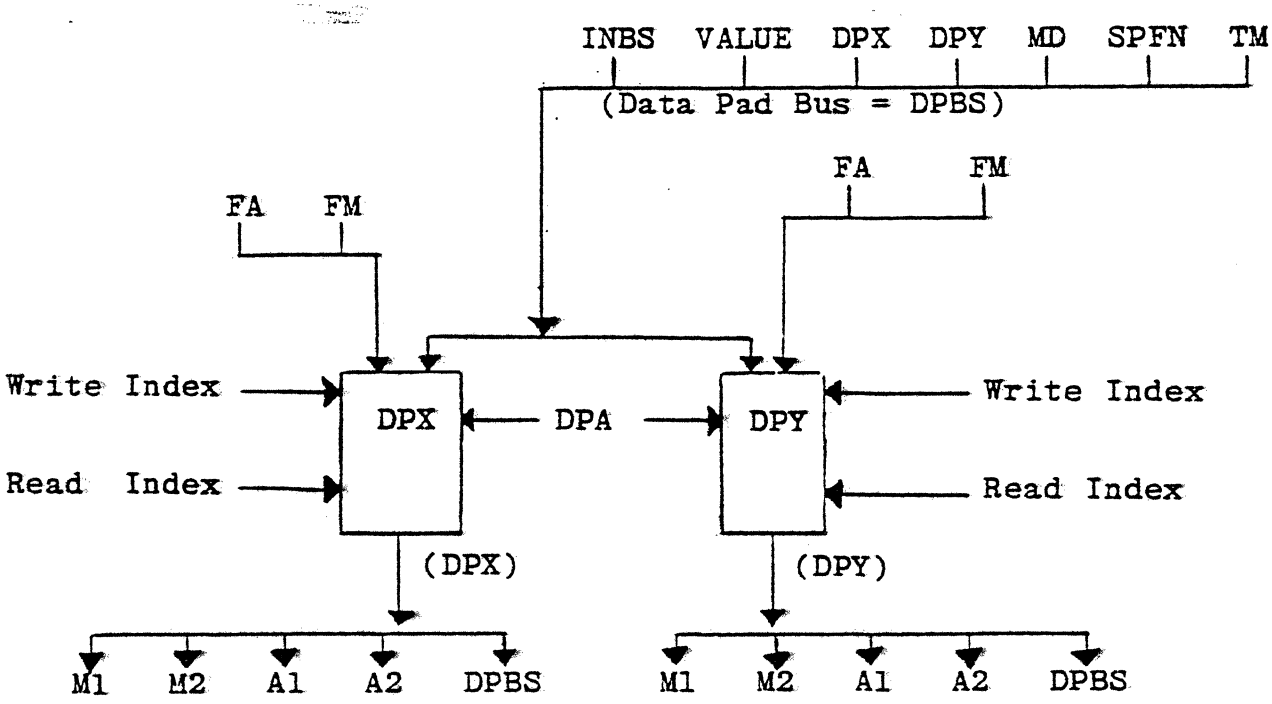


Figure 4-4 Data Pad

4.6.2 Data Pad Operations

Operations within the Data PAD GROUP may be classed in the following groups:

* DPX or DPY explicit WRITE operations
* DPX or DPY implicit READ operations
* DPBS source enabling operations

4.6.2.1 DPX or DPY Explicit WRITE Operations

One each DPX and DPY location may be written during a given instruction cycle, using either the DPX or DPY fields and either XW or YW fields of the instruction word.

MNEMONIC	INSTRUCTION WORD FIELD(S) USED	OCTAL VALUE	OPERATION
DPX(idx)<DB	DPX XW	1 0-7	Stores current DB into specified DATA PAD X location.
DPX(idx)<FA	DPX XW	2 0-7	Stores current FA into specified DATA PAD X location.
DPX(idx)<FM	DPX XW	3 0-7	Stores current FM into specified DATA PAD X location.
DPY(idx)<DB	DPY YW	1 0-7	Stores current DB into specified DATA PAD Y location.
DPY(idx)<FA	DPY YW	2 0-7	Stores current FA into specified DATA PAD Y location.
DPY(idx)<FM	DPY YW	3 0-7	Stores current FM into specified DATA PAD Y location.

NOTE

If VALUE field is used in the same disabled. The DPY location will be referenced by the XW field, instead.

4.6.2.2 DPX,DPY Implicit Read Operations

One each DPX and DPY location may be read during a given instruction cycle. Whenever a DATA PAD X or Y location is referenced as part of an operation in this or another group in the current instruction word, the XR and YR fields of this group are referenced. Example:

```
FADD  DPY(1),DPX(2)
```

The appropriate index fields in the DP group will be set when specifying the above operation. The YR field will be set to "5(octal)", and the XR field set to "6(octal)", since the fields are BIASED by four. The index is relative to the current contents of the DATA PAD ADDRESS REGISTER (DPA).

A complete summary of the DATA PAD addressing scheme is given in the following section.

4.6.2.3 DPBS Enabling Operations

One of eight sources may be enabled onto DPBS during a given instruction cycle, using the DPBS field of the current instruction word. Specifying a DPBS source has an immediate effect, that is, the DPBS source enabled during a given instruction will be the data currently used in all DPBS-related operations during the same instruction cycle.

Note that only one source may be enabled onto DPBS during any given instruction and that source will remain as DPBS only during this instruction.

MNEMONIC	INSTRUCTION WORD FIELD(S) USED	OCTAL VALUE	OPERATION
DB = ZERO	DPBS	0	Floating Point ZERO (0.0) is enabled as DPBS for the current instruction. This is the default selection for DPBS.
DB = INBS	DPBS	1	The data currently enabled onto INBS is enabled as DPBS for the current instruction.

DB = value (See Note)	DPBS VALUE	2	The contents of the value field are enabled as DPBS for the current instruction. (Partial word transfer, see instruction summary (DB = VALUE) for a detailed explanation.
DB = DPX(idx)	DPBS XR	3 0-7	The contents of the currently specified DATA PAD X location is enabled as DPBS for the current instruction.
DB = DPY(idx)	DPBS YR	4 0-7	The contents of the currently specified DATA PAD Y location is enabled as DPBS for the current instruction.
DB = MD	DPBS	5	The current contents of the MAIN DATA OUTPUT REGISTER (MDREG) are enabled as DPBS for the current instruction.
DB = SPFN	DPBS	6	The current SPFN is enabled as DPBS for the current instruction. (Partial-word transfer, see illustration summary (DB = SPFN) for a detailed explanation).
DB = TM	DPBS	7	The current contents of the TABLE MEMORY OUTPUT REGISTER (TMREG) are Enabled as DPBS for the current instruction.

NOTE

Value is an integer from decimal -32768 to 32767 or 0 to 177777 in octal or a label (constant).

In addition to the eight sources available in the DPBS field, three other sources (PS Left Half, PS Floating and PNLBS) can be enabled onto DPBS via operations in the SPEC group. These operations (e.g. RPSL, RPSF, SWDB) take precedence over the DPBS field.

4.6.3 Data Pad Addressing

The effective address of a location in DATA PAD to be read or written is determined by the following combination of elements:

* The current contents of the DATA PAD ADDRESS REGISTER (DPA), plus
* The value contained in the appropriate INDEX field (XW, XR, YW, YR) of the current instruction word, minus
* The BIAS (4(octal))

The INDEX field value is contained within the current instruction word.

The DPA contents may be changed by use of the following instructions effective one cycle after the appropriate instruction is executed.

INSTRUCTION	MEANING
INCDPA	Add "1" to DPA
DECDPA	Subtract "1" from DPA
SETDPA	Set current SPFN into DPA
LDDPA	Set current DPBS into DPA

DPA is circular in that incrementing DPA currently containing the maximum address (37octal) will produce a DPA of 0(octal) as of the next instruction cycle. Accordingly, decrementing a DPA of 0(octal) will produce a DPA of 37 (octal) as of the next instruction cycle. This circular effect applies equally to indexing. Example:

if DPA = 37(octal); then:
DPX(2) would indicate DPX location 1(octal).

if DPA = 0(octal); then
DPX(-2) would indicate DPX location 36(octal).

Note that instructions from LDREG field (LDDPA, LDTMA, MA) cause SETDPA, SETTMA, and SETMA to load from DPBS instead of SPFN.

Examples	Meaning
DB<DPX(1)	Place the contents of the DPX location pointed to by the current contents of the DPA plus one onto DPBS.

When specified in the above fashion, the ASSEMBLER will place a "5(octal)" into the XR field and a "3(octal)" into the DPBS field of the instruction word. The "5" in the XR field indicates that the DPX READ operation is "5" BIAS locations relative to current DPA. BIAS is "4(octal" for the DATA PAD INDEX FIELDS.

Accordingly, eight locations are available for any given DPX or DPY READ/WRITE operation, from +3 to -4 locations relative to the current DPA.

APPARENT VALUE (VALUE CONTAINED IN INDEX FIELD)	TRUE VALUE RELATIVE TO DPA	MEANING
7	+3	DPX or Y(DPA) +3
6	2	DPX or Y(DPA) +2
5	1	DPX or Y(DPA) +1
4	0	DPX or Y(DPA)
3	-1	DPX or Y(DPA) -1
2	-2	DPX or Y(DPA) -2
1	-3	DPX or Y(DPA) -3
0	-4	DPX or Y(DPA) -4

Of course, the programmer need only indicate the desired DATA PAD location in the following manner:

DPX (+3)	DPY (+3)
DPX (+2)	DPY (+2)
DPX (+1)	DPY (+1)
DPX (0)	DPY (0)
DPX (-1)	DPY (-1)
DPX (-2)	DPY (-2)
DPX (-3)	DPY (-3)
DPX (-4)	DPY (-4)

4.6.4 Programming Examples

Since four separate displacement fields (XW, XR, YW, YR) are provided within the instruction word, four separate locations in DATA PAD may be used in a given instruction. Example:

Assume DPA = 24(octal):

```
FADD   DPX(3),MD; FMUL TM,DPY(-2); DPX(-3)<FA; DPY(1)<FM
```

The above operation would:

* Use the contents of DPX location 27 in the FADDR operation,
* Use the contents of DPY location 22 in the FMULR operation,
* Write the currently available FA into DPX location 21, and
* Write the currently available FM into DPY location 23.

Note that only one DPX location may be written, only one DPY location may be written, only one DPX location may be read, and only one DPY location may be read during the same instruction cycle.

Using the DPBS to Write Into DATA PAD

Again, one may write into DATA PAD from FA, FM or from one of eight possible sources available on the DATA PAD Bus (DB).

Note, that only ONE source may be enabled onto DB for any given instruction. Example:

LEGAL

1. DPX<DB; DPY<DB; DB = MD	Meaning:	Write MD into both DPX and DPY via DB.
2. DPX<FA; DPY<DB; DB = TM	Meaning:	Write current FA to DPX, write current TM to DPY via DPBS.

ILLEGAL

3. DPX<DB; DB = MD; DPY<DB; DB = TM	Again, only ONE source may be enabled onto DB per instruction.

* DB Shorthand Notation

The assembler will automatically enable a source onto DB even though the programmer has not explicitly written the instruction. In other words, the above examples may be written as follows:

	Programmer Writes:	Assembler Inserts:
1.	DPX<MD;DPY<MD	DB = MD
2.	DPX<FA; DPY<TM	DB = TM
3.	DPX<MD; DPY<TM	Error: Assembler will flag as such.

As shown by the above examples, when using the shorthand notation, the programmer must bear in mind that he may only use one source as DPBS in a given instruction cycle.

4.7 MEMORY GROUP

The operations available within the MEMORY GROUP may be classed into the following functional groups:

* MAIN DATA MEMORY (MD) accessing operations
  MAIN DATA MEMORY ADDRESS REGISTER (MA) modification
* DATA PAD ADDRESS REGISTER (DPA) modification
* TABLE MEMORY (TM) accessing operations
  TABLE MEMORY ADDRESS (TMA) modification

Accordingly, this summary will be presented in the following manner:

1. MAIN DATA MEMORY (MI and MA fields)
   * General description
   * Addressing and memory cycle initialization
   * MD read and write operations

2. DATA PAD ADDRESS modification (DPA field)

3. TABLE MEMORY (TMA field)
   * General description
   * Addressing
   * Read and write operations

4.7.1 Main Data Memory (MD)

4.7.1.1 General Description

MAIN DATA (MD) is the main storage file, within the AP-120B for 41-bit data words. MD is a monolithic storage file available in 8K, 16K, 32K and 64K modules - 64K per page - up to a million words. MAIN DATA has two speed ranges: 333 or 167 ns if the interleaving capability is utilized and 500 or 333 ns if non-interleaving locations are accessed.

The MAIN DATA BLOCK (Figure 4-5) consists of the following component parts:

MEMORY INPUT REGISTER (MI)
MAIN DATA STORAGE FIELD (MD)
MAIN DATA MEMORY OUTPUT REGISTER (MDREG)
MAIN DATA MEMORY ADDRESS REGISTER (MA)

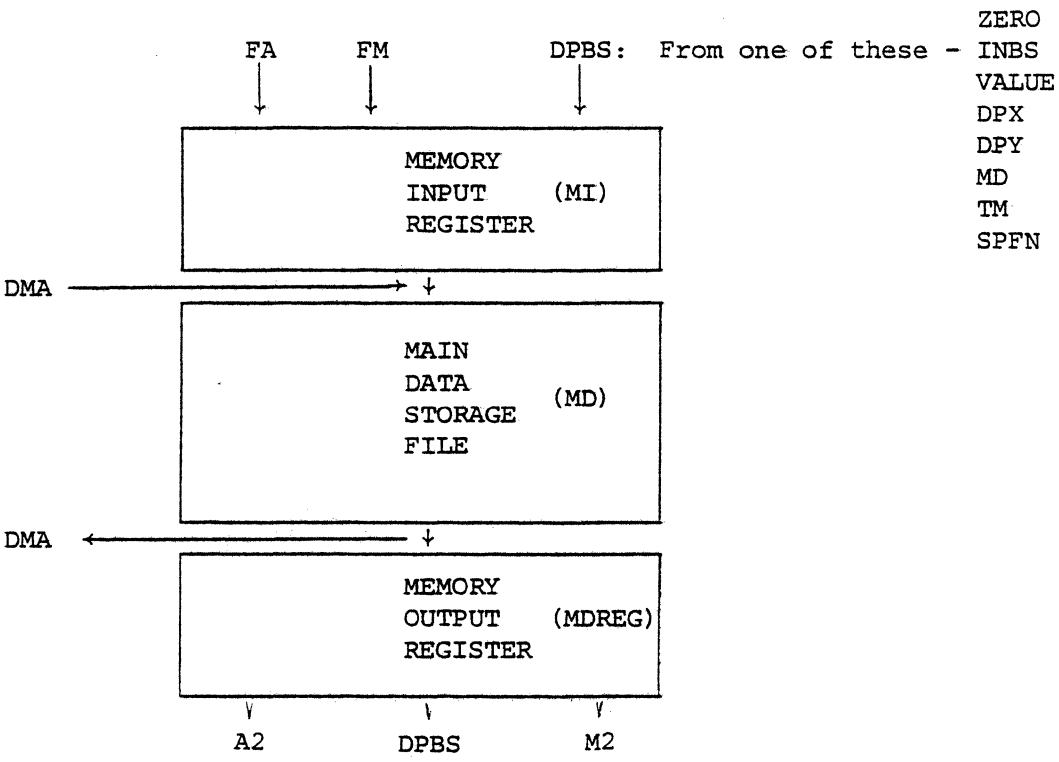
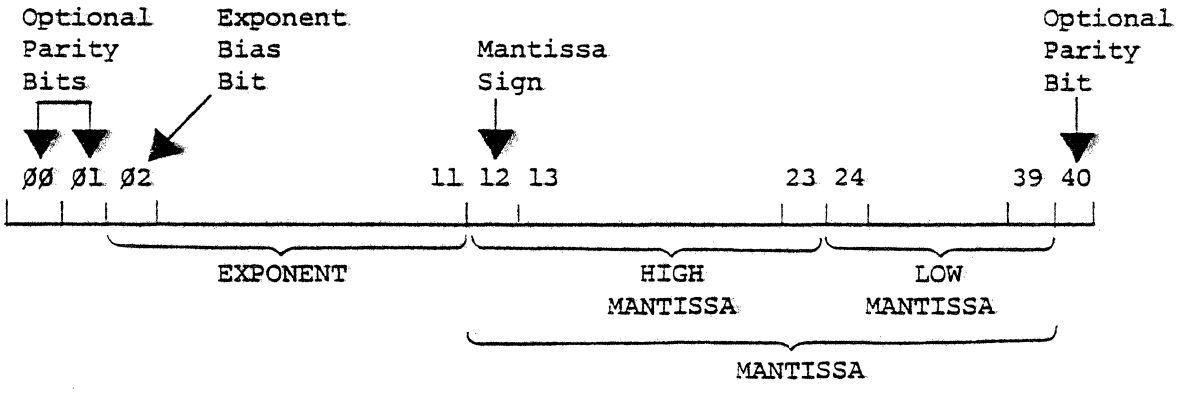


Figure 4-5 MAIN DATA BLOCK

MAIN DATA may be written from the FADDR output (FA), the FMULR output (FM), or from one of several sources enabled onto the DATA PAD BUS (DB). The MAIN DATA output, termed MD, is available as an input directly to the FADDR A2 operand register (A2), the FMULR M2 operand register (M2), or may be enabled onto the DB.

Additionally, MD may be accessed via DIRECT MEMORY ACCESS (DMA) from the Host and other I/O interfaces.

The format of a typical MD data word is presented below:



4.7.1.2 Addressing and Memory Cycle Initialization

The 16-bit MEMORY ADDRESS REGISTER (MA) is the pointer indicating which MD location is read or written during a given operation. Additionally, an MD memory cycle is initiated each time the contents of MA are altered by use of an INCMA, DECMA, SETMA or LDMA instruction.

MA MODIFICATION OP-CODES	EFFECT
INCMA	Increment MA by "1"; initiate an MD memory cycle.
DECMA	Decrement MA by "1"; initiate an MD memory cycle.
SETMA	Set MA from current SPFN (See Note); initiate an MD memory cycle.
LDMA	Set MA from current DB; initiate an MD memory cycle.

If an MI Op-Code is concurrently specified with an appropriate MA modification Op-Code, then an MD MEMORY WRITE CYCLE is initiated, using the new contents of MA to indicate the MD location to be written. If an MA modification instruction is specified without a concurrent MI Op-Code, then an MD MEMORY READ CYCLE is initiated using the new contents of MA to indicate the MD location to be read. Use of an MI Op-Code WITHOUT a concurrent MA modification op-code results in a NOP.

NOTE

If an Op-Code from the LDREG field (I/O) is used concurrently, then SETMA will set MA from current DB, not SPFN.

EXAMPLES:

INITIATING AN MD WRITE CYCLE	(MEANING)
MOV 5,5; SETMA; MA<FA	(Set MA to current SPFN, write current FA into MD location pointed to by the new contents of MA).
LDMA; DB=DPX(0); MI<FM	(Set MA to current DPX(0), write current FM into MD location pointed to by the new contents of MA).
INCMA; MI<DB	(Increment MA by "1", write current DB into MD location pointed to by new contents of MA).
INITIATING AN MD READ CYCLE	
MOV 5,5; SETMA	(Set MA to current SPFN, read MD location pointed to by new contents of MA).
DECMA	(Decrement MA by "1", read MD location pointed to by the new contents of MA).

Note, again, that an MA modification Op-Code executed WITH a concurrent MI Op-Code generates an MD WRITE CYCLE. An MA modification Op-Code WITHOUT an MI Op-Code generates an MD READ CYCLE.

4.7.1.3 Interleave

STANDARD speed MD MEMORY references the same bank of memory every three AP cycles (500 ns). In order to facilitate faster intervals between successive memory reference instructions, the AP 120B MAIN DATA MEMORY is divided into banks containing 4K or 16K words each. These banks are interleaved in pairs with odd number memory locations contained in one bank of the pair and even memory locations in the other bank of the pair. Memory reference instructions are allowable every other instruction cycle (333 ns) as long as INTERLEAVE (the sequential reference to different MD MEMORY BANKS) is not violated. Any attempt to reference the same bank of memory (NON-INTERLEAVING) before the minimum time constraint will cause the AP-120B hardware to generate a "SPIN" operation - effectively suspending all ongoing AP-120B program execution one cycle at a time until the memory is no longer busy and can execute the memory reference instruction which prompted the "SPIN" condition. This feature allows AP-120B programs to be written without concern for memory interleaving. But for maximum execution speed the interleaving feature should be used. Thus, for optimum coding of memory accesses, the programmer should be aware of the order in which he accesses memory banks.

For FAST MD MEMORY, memory reference may be made to different banks (INTERLEAVED) every AP cycle (167 ns) and to the same bank (NON-INTERLEAVED) every other AP cycle (333 ns.)

Interleave examples: locations 3 and 4 are interleaved by the odd-even interleave. Locations decimal 8190 and 8192 are interleaved because they reference different bank pairs (4-K bank size). For the 16-K bank size, the bank-pair size is 32K. Thus, for example, locations decimal 32766 and 32768 are in different bank pairs.

NOTE

For the 32K and 64K memory modules, the bank size is increased to 16K.

EXAMPLE:

MEMORY ADDRESS SEQUENCE (OCTAL)	MEMORY BANK SEQUENCE	STANDARD MEMORY REFERENCE TIMING	FAST MEMORY REFERENCE TIMING
101	1		
102	0	Every two	Every
103	1	AP cycles	AP cycle
104	0	(Interleaved)	(Interleaved)
100	0		
102	0	Every three	Every two
104	0	AP cycles	AP cycles
106	0	(Non-interleaved)	(Non-interleaved)
234	0		
10374	2	Every two	Every
233	1	AP cycles	AP cycle
10376	2	(Interleaved)	(Interleaved)

4.7.1.4 MD Read and Write Operations

Three AP clock cycles after an MD memory read cycle has been initiated, the data from the selected memory location becomes available as MD to the DB, A2 operand register, and M2 operand register.

STANDARD MEMORY		FAST MEMORY	
AP CODE	LOCATION AVAILABLE AS MD	AP CODE	LOCATION AVAILABLE AS MD
1. INCMA	-----	1. INCMA	-----
2. NOP	-----	2. INCMA	-----
3. INCMA	-----	3. INCMA	-----
4. NOP	MD101	4. NOP	MD101
5. INCMA	MD101	5. NOP	MD102
6. NOP	MD102	6. NOP	MD103
7. NOP	MD102	7. NOP	MD103
8. NOP	MD103	8. NOP	MD103

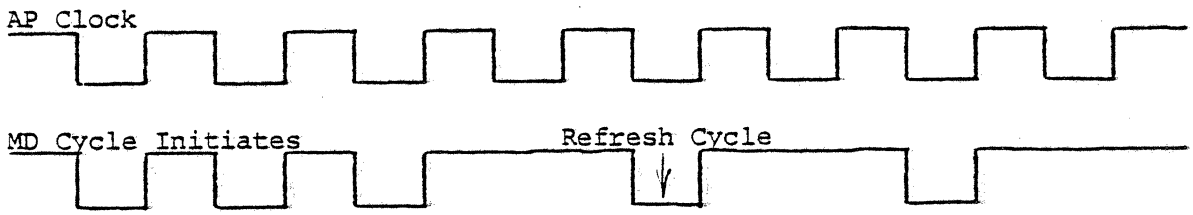
Once an MD MEMORY WRITE CYCLE is initiated, the data specified by the concurrent MI Op-Code is loaded into the MEMORY INPUT REGISTER (MI) and written into MD at the location pointed to by the MA register.

Data may be written INTERLEAVED every other AP cycle and NON-INTERLEAVED every three AP cycles for standard memory and INTERLEAVED every cycle and NON-INTERLEAVED every other cycle for fast MD.

Note that the example for standard memory with NOP's between the INCMA's will execute correctly on Fast Memory; however, the reverse is not true. Both examples will execute correctly with the INCMA's replaced by SETMA's even if the memory accesses are NON-INTERLEAVED. NON-INTERLEAVED accesses merely result in a slowdown of instruction execution; they do not change the timing relationships between instructions. Writing MD leaves the contents of MDREG unchanged. This allows a memory Read cycle to be initiated and the data to be used later irrespective of intervening MD write cycles.

4.7.1.5 AP-120B DMA and Refresh Time Penalties

1. REFRESH

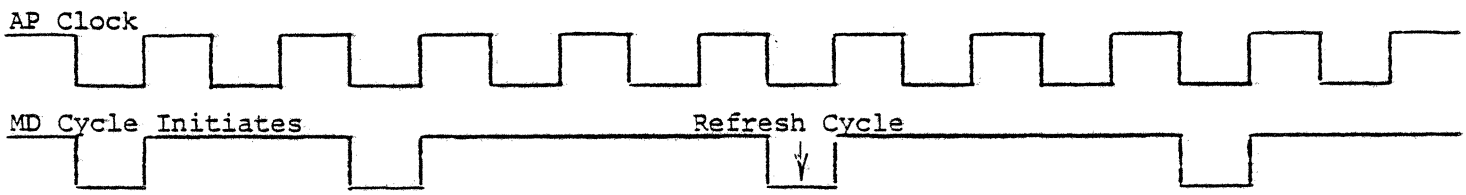


FAST MD

a) Refresh costs three clocks = 500ns if AP running interleaved cycles (worst case).

b) Refresh costs only two clocks if non-interleaved cycles in AP program.

STANDARD MD



c) Refresh costs four clocks = 667ns if AP running interleaved cycles (worst case).

d) Refresh costs only three clocks if AP program running non-interleaved cycles.

2. DMA interference (assuming AP running interleaved cycles).

	Time to be added to AP execution per DMA memory cycle:	
	Fast MD	Std MD
a) Host interface DMA interference:		
DMA address in separate bank-pair from AP accesses	1 clock	2 clocks
Same bank-pair	2 clocks	3 clocks
b) Second DMA channel (IOP,PIOP):		
Separate bank-pair	2 clocks	3 clocks
Same bank-pair	2 1/2 clocks	3 1/2 clocks

Subtract one clock if AP running non-interleaved cycles in one bank. One clock = 167ns

4.7.1.6 Programming Examples

* Read Example

Load a vector Ax,i = 0,2 stored in Memory Locations 101, 102, 103 into DPX locations 10, 11, 12. We will assume that MA was set to 100 and DPA was set to 10 before we started.

```
1.   INCMA                      "Fetch A0 from Memory
2.   ---
3.   INCMA                      "Fetch A1 from Memory
4.   DPX<MD; INCDPA             "Store A0 into DPX location 10
                                "  and bump DPA pointer to 11.
5.   INCMA;                     "Fetch A2 from Memory
6.   DPX<MD; INCDPA             "Store A1 into DPX location 11
                                "  and bump DPA pointer to 12.
7.   ---
8.   DPX<MD                     "Store A2 into DPX location 12.
```

Below is a chart of the above transfer, showing the state of each component after each instruction.

	Memory		Data Pad			
Cycle	MA	MD	DPA	DPX_{10}	DPX_{11}	DPX_{12}
1.	101	---	10	---	---	---
2.	101	---	10	---	---	---
3.	102	---	10	---	---	---
4.	102	A_0	10	A_0	---	---
5.	103	A_0	11	A_0	---	---
6.	103	A_0	11	A_0	A_1	---
7.	103	A_1	12	A_0	A_1	---
8.	103	A_2	12	A_0	A_1	A_2

* Write Example

Square the elements of a vector Ax,i = 0, 1, 2, in DPX locations 10, 11, 12 and store the results into Data Memory locations 101, 102, 103. We will assume that MA was set to 100 and DPA was set to 10 before we started.

```
1.  FMUL DPX, DPX; INCDPA     "Square A0, bump DPA pointer
                              "  to 11.
2.  FMUL                      "Push down the multiplier
                              "  pipeline.
3.  FMUL DPX, DPX; INCDPA     "Square A1, bump DPA pointer
                              "  to 12.
4.  FMUL; MI<FM; INCMA        "Write A0^2 into memory location
                              "  101.
```

Below is a chart of this computation:

		Multiplier		Memory	
Cycle	DPA	M1,M2	FM	MA	MI
1.	10	A_0,A_0	--	--	--
2.	11	---	--	--	--
3.	11	A_1,A_1	--	--	--
4.	12	---	A_0^2	101	A_0^2
5.	12	A_2,A_2	--	101	A_0^2
6.	12	---	A_1^2	102	A_1^2
7.	12	---	--	102	A_1^2
8.	12	---	A_2^2	103	A_2^2

* Memory Interleave

Data Memory is divided into 16 banks of 4K words each using MA00-MA02 and MA15 as a memory bank select. (These are the three highest-order bits and the least-significant bit of MA.) Memory references to different banks may be made every 2 AP cycles, while references to the same bank may be made every 3 AP cycles. For some possible memory addressing sequences we have:

Memory Address Sequence (Octal)	Memory Bank Sequence	Memory Reference Timing
101, 102, 103, 104,...	1, 0, 1, 0,...	every 2 AP cycles
166, 165, 164, 163,...	0, 1, 0, 1,...	every 2 AP cycles
100, 102, 104, 106,...	0, 0, 0, 0,...	every 3 AP cycles
233, 10374, 234, 10376,...	1, 2, 0, 2,...	every 2 AP cycles

Thus references to successive sequential memory locations may be made every other AP cycle, but references to successive-odd or successive-even locations must be three cycles apart.

4.7.2 TABLE MEMORY (TMA)

4.7.2.1 General Description

The TABLE MEMORY FILE (TM) is a separate 38-bit wide storage file used to store standard constants and other slowly changing coefficients. Addressing for TM locations is achieved by use of the TABLE MEMORY ADDRESS REGISTER (TMA).

TM is available in two types -- READ ONLY MEMORY (TMROM) and RANDOM ACCESS MEMORY (TMRAM). TMRAM is capable of both being WRITTEN and READ, while TMROM is capable only of being READ. In both types of memory, the contents of the TM location pointed to by the current contents of TMA become available as TM two cycles after an instruction that alters TMA, with one exception; that is--for a TMRAM WRITE operation, the contents of TMREG are undefined two cycles later, and the new contents of the TM location written can become available as TM three cycles later if TMA is not changed.

TABLE MEMORY (Figure 4-6) consists of the following component parts:

1. TABLE MEMORY STORAGE FILE (TM)
2. TABLE MEMORY ADDRESS REGISTER (TMA)
3. TABLE MEMORY OUTPUT REGISTER (TMREG)
4. TABLE MEMORY INPUT REGISTER (TMIREG)

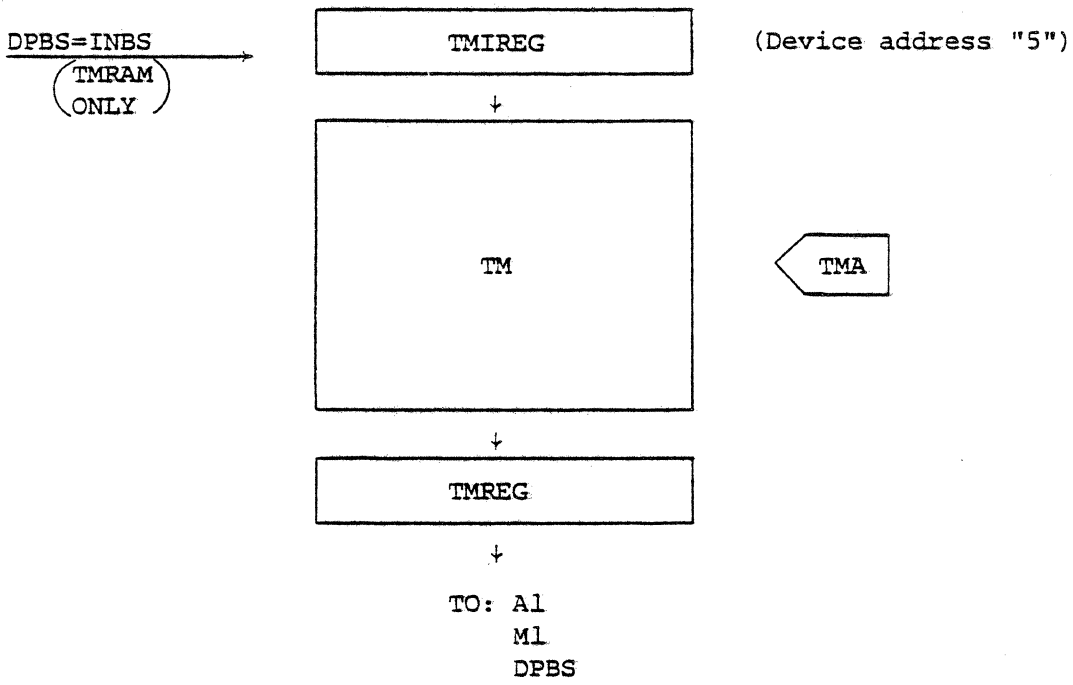


Figure 4-6 Table Memory

Values stored in Table Memory are read by setting the Table Memory Address (TMA) register to the address of the desired Table Memory location. This is done by the instructions:

```
INCTMA    "Increment TMA by 1
DECTMA    "Decrement TMA by 1
SETTMA    "Set TMA to the current S-Pad
          #function (SPFN)
```

Each of the above initiates a fetch from the Table Memory Location pointed at by the new contents of TMA. Two AP cycles later the contents of the desired location are available for use. A new location may be fetched every Ap cycle. One instruction must be placed between a TMA address modification instruction and the use of TM contents.

Therefore,

```
        INCTMA                  "points to TM address X
        INCTMA                  "points to TM address Y
        FMUL TM, MD             "Multiplies contents of TM address
                                " X with MD
        FMUL TM, DPX(0)         Multiplies contents of TM address
                                " Y with DPX(0)
```

In TMRAM, data may be written into TM from the DATA PAD BUS (DB), via an I/O instruction. Data may be read from TMRAM or TMROM directly to the FLOATING ADDER A1 OPERAND REGISTER (A1), to the FLOATING MULTIPLIER M1 OPERAND REGISTER (M1), or onto the DATA PAD BUS (DB).

With respect to writing operations, TMRAM is an I/O DEVICE whose address is 5. To write into TMRAM, one must set DA to 5 at least one cycle before outputting data to TMRAM. For example, in order to write the current FA into TMRAM location 100(octal), the following program would be specified:

Assume S-PAD REGISTER 6 equals "100(octal)"

```
ASSEMBLER FORMAT                    (Meaning)
  o
  o
  o
  LDDA;DB = 5                       (Select TMRAM as IODEVICE (DA))
  MOV 6,6; SETTMA; DB<FA; OUT       (Set TMA to "100(octal)", write
                                    current FA into TMRAM 100(Octal)
  o
  o
  o
```

Because the TABLE MEMORY INPUT REGISTER can accept data in 167 ns, it does not have an IORDY flag associated with it. Therefore, the SPNOUT instruction should not be used to load TMRAM. Data placed on DB during the OUT instruction is written into TMRAM at the address contained in TMA following the OUT instruction. Thus, if an OUT instruction also modifies TMA, the data is written at the TMA address specified by that OUT instruction.

Example:

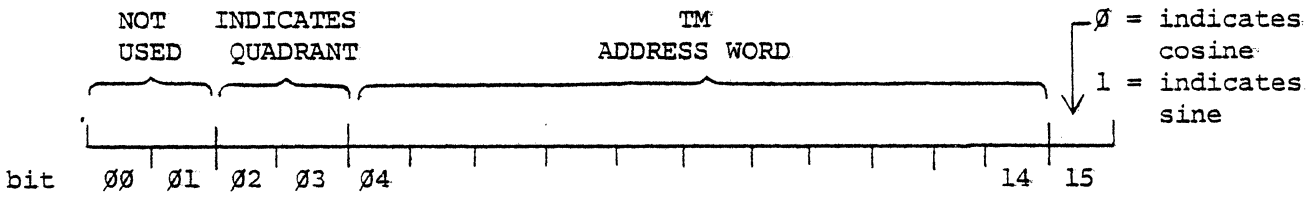
```
 Location
 Counter        INSTRUCTION          COMMENTS

    0           LDDA; DB=5           " SET DA=5

    1           OUT; DB=DPX(0);      " Send DPX(0) to TMRAM
                     INCTMA          " at TMA+1

    2           NOP                  " Other user code
                                     " Previous TM output OK
                                     " here

    3           NOP                  " TM output invalid
                                     " during this instruction

    4           FADD TM, DPX(1)      " Use of TMRAM
                                     " Output OK here
                                     " User gets value
                                     " Written by the OUT
                                     " If TMA not changed
                                     " At instruction 2
```

It is possible to write TMRAM on every instruction. TMRAM behaves in all other respects exactly as does TABLE MEMORY ROM (TMROM). Different TMRAM locations can be read on every instruction (167 ns cycle) and the output data is available for use during the second following instruction (167 ns access). It is only after TMRAM WRITE that the TMRAM output is invalid during the second following instruction.

4.7.2.2 Addressing

Addressing of appropriate TM locations is achieved by reference to the current contents of the 16-bit TABLE MEMORY ADDRESS (TMA). When used strictly as an address pointer, TMA is capable of addressing 64K of TABLE MEMORY.

However, TMA is alternatively used as both an address pointer and as a quadrant/sine/cosine indicator for constants used in FFT or IFFT operations. This alternate addressing is enabled by the FFT Bit in APSTATUS (Bit 12) and is modified by the IFFT Bit (Bit 11). In this case, certain bits in TMA also perform control functions for referencing and properly preparing the raw constant stored in TM. Given below is a sample FFT-related format of TMA for a 2048 word cosine table (8K max FFT size).



This division of the TMA register into three sections allows the programmer to address a table of real cosine values lying between angles 0 and 90 degrees as if it were a full circle of complex exponential values with the real part in even numbered locations and the imaginary part in odd numbered locations.

TMA quadrant, sine, cosine theory of operation:

The complex exponential function and the sine and cosine functions are related via the following equation;

$$e^{\pm jX} = \text{cosine } (X) \pm j \text{ sine } (X)$$

$$\text{EXP } (\pm X) - \text{COS } (X) \pm j \text{ SIN } (X).$$

Thus, the real part of the complex exponential of plus or minus a positive real number X is the cosine of X and the imaginary part is plus or minus the sine of X. A forward FFT requires complex exponentials of negative arguments CEXP(-X), while an inverse FFT (IFFT) requires complex exponentials of positive arguments, CEXP(X). Thus, the TMA logic is set up so that the normal mode of operation (FFT = "1", IFFT = "0" in APSTATUS) is to produce complex exponentials of minus X. With FFT = "1" and IFFT = "1", the TMA logic produces complex exponentials of plus X. With FFT = "0" and IFFT = "0" or "1", the logic treats TMA as an unmodified 16-bit address.

Complete understanding of the TMA complex exponential addressing requires knowledge of the following trigonometric identities with 0 degrees $\leq$ X < 90 degrees.

0^{th} quadrant	COS (X)	= COS (X)
	SIN (X)	= COS (90^{o} - X)
1^{st} quadrant	COS (X+90^{o})	= -SIN (X)
		= -COS (90^{o} - X)
	SIN (X+90^{o})	= COS (X)
2^{nd} quadrant	COS (X+180^{o})	= -COS (X)
	SIN (X+180^{o})	= -SIN (X)
		= -COS (90^{o} - X)
3^{rd} quadrant	COS (X+270^{o})	= SIN (X)
		= COS (90^{o} - X)
	SIN (X+270^{o})	= -COS (X)

Thus, these eight equations reduce the desired sine and cosine values in any quadrant to cosine values in the 0th quadrant (0 degrees $\leq$ X < 90 degrees) plus the one value COS (90 degrees) = 0 for X = 0 in COS (90 degrees - X).

This last value is achieved by inhibiting the output of TM when COS (90 degrees) is desired. Since the equations do require negative values of the cosine and the table contains only positive values, the logic in TMREG contains a two's complement generator that is capable of negating the mantissa of the output of TM, thus producing a negative floating-point result when -COS(X) or -COS(90 degrees - X) is required.

For use with the FFT or IFFT, the table size required is always a power of two since the FFT algorithm works with the power of two number of data points. The FFT requires a full circle of complex exponentials that has the same number of complex values in it as the largest number of data points desired (number of real points for real FFT's, number of complex points for complex FFT's). Since the above equations reduce the full circle of complex exponentials to one quadrant of cosines, we see that the cosine table must contain one-fourth the number of points for the largest desired FFT.

Thus, for the typical case of an 8192-point maximum size FFT, there will be 2048 values of the cosine function in table memory. The values are computed for the angles $(n \cdot 90^{o}) \div 2048$ where $n = 0$ to 2047 . Thus each increment in TM address, represents an angle increment of $90^{o} \div 2048$.

Since an angle of 90 degrees corresponds to a table memory address of 2048, the problem of generating the addresses for the angles 90 degrees - X required by the above equations reduces to a question of simply two's complementing the TM address (Bits 04 to 14) portion of the TMA register output and masking the result back to 11 bits.

$$(90^{\circ} - x) = \frac{2048-90^{\circ}}{2048} - \frac{n \cdot 90^{\circ}}{2048} = (2048 - n) \cdot \frac{90^{\circ}}{2048}$$

In 11-bit binary arithmetic, (2048 - n) is an equivalent definition of the two's complement of n.

Thus, the hardware logic for the complex exponential generation also provides the ability to steer bits 04 to 14 of the TMA register, onto bits 05 to 15 of the TMA address bus, while placing zeros on bits 00 to 04, and to conditionally two's complement bits 04 to 14 of TMA, when the angle (90 degrees - X) is required.

Note that the op-code RTMA reads the value on the TMA address bus, while the op-codes in the SPEC OPER field that use TMA (JSRT, JMPT, LPSLT, etc.) use the unaltered 16-bit contents of the TMA register. An FN register breakpoint on TMA also uses the value on the TMA address bus.

Table 4-3 shows the Table Memory CEXP truth table.

Table 4-3 Table Memory CEXP Truth Table

QUADRANT		SIN/COS	FFT	IFFT	TM ADDRESS	TM OUTPUT
			0		Full 16 bits	+
0	0	0	1	0/1	+n	+
0	0	1	1	0/1	-n	-/+
0	1	0	1	0/1	-n	-
0	1	1	1	0/1	+n	-/+
1	0	0	1	0/1	+n	-
1	0	1	1	0/1	-n	+/-
1	1	0	1	0/1	-n	+
1	1	1	1	0/1	+n	+/-

n = bits 04 to 14 of TMA register for a 2048 point cosine table.

Maximum points in FFT = 8192.

quadrant = Bits 02 and 03 of TMA register for a 2048 point cosine table

SIN/COS = Bit 15 of TMA register

FFT = Bit 12 of APSTATUS

IFFT = Bit 11 of APSTATUS

Hardware strap options allow n to be widened and the quadrant bits moved to the left, thus allowing alternative cosine table sizes of 4K (16K FFT) and 8K (32K FFT). Cosine table sizes of 16K and 32K are possible. However, the quadrant determination logic will not work properly and thus special FFT micro-code would be required.

4.7.2.3 Read and Write Operations

In READ OPERATIONS the contents of the TM location pointed to by the current address contained in TMA will become available as TM two instruction cycles later.

In order to change TMA, one may specify an INCTMA, DECTMA, SETTMA op-code from the TMA field of the MEMORY portion of the instruction word, or specify a LDTMA op-code from the LDREG field of the I/O portion of the instruction word. The contents of TMA as altered by these op-codes are available one cycle later for op-codes that use it as an address (JSRT) or data (RDTMA). The available TMA modification op-codes are given below:

INSTRUCTION WORD FIELD	OCTAL VALUE	OP-CODE	OPERATION
TMA	1	INCTMA	Increment current TMA address by "1"
TMA	2	DECTMA	Decrement current TMA address by "1"
TMA	3	SETTMA	Replace TMA address with current SPFN (See Note)
LDREG	3	LDTMA	Replace TMA address with current DB

NOTE

If an Op-Code from LDREG field (I/O) is used concurrently, then SETTMA will replace the TMA address with the current DB, not SPFN.

4.7.2.4 Programming Examples

* Example 1

Do the vector sum Ax = B+K, i = 0, 1, 2, where Ax is in DPX locations 10-12, Bx is in DPY 10-13, and Kx is a series of constants stored in Table Memory location 235-237. Ax will be stored back into DPX. We will assume that DPA was set to 10 and TMA was set to 234 before we start.

```
1.   INCTMA                              "Fetch K0
2.   INCTMA                              "Fetch K1
3.   INCTMA; FADD TM, CPY; INCDPA        "Do K0 + B0, bump DPA to 11
4.   FADD TM, DPY; INCDPA                "Do K1 + B1, bump DPA to 12
5.   FADD TM, DPX (0); DPX (-2)<FA       "Do K2 + B2, store A in DPX10
6.   FADD: DPX (-1)<FA                   "Store A1 in DPX11
7.   DPX (0)<FA                          "Store A2 in DPX12
```

The following charts the above computation:

Cycle	Table Memory TMA	Memory TM	Adder A1,A2	FA	Data DPA	Pad X 10	11	12
1.	235	---	---	---	10	---	---	---
2.	236	---	---	---	10	---	---	---
3.	237	K_0	K_0,B_0	---	10	---	---	---
4.	237	K_1	K_1,B_1	---	11	---	---	---
5.	237	K_2	K_2,B_2	K_0+B_0	12	A_0	---	---
6.	237	K_2	---	K_1+B_1	12	A_0	A_1	---
7.	237	K_2	---	K_2+B_2	12	A_0	A_1	A_2

* Example 2 - A Complex Multiply

An example using both memories is a complex multiply from the FFT (Fast Fourier Transform) algorithm. The multiply is between a complex signal point held in Data Memory and a complex exponential value (a root of unity, xxx) fetched from Table Memory. The computation is:

$$X_R = C_R * W_R - C_I * W_I$$

$$X_I = C_R * W_I + C_I W_R$$

Where C is the data point and W is the complex exponential "R" and "I" denote real and imaginary parts respectively. C is in Main Data Memory, and W is in Table Memory.

Fetch the	1.	INCMA	"Fetch C_R from Data Memory
4 arguments	2.	INCTMA	"Fetch W_R from Table Memory
	3.	INCMA; INCTMA	"Fetch C_I fetch W_I
	4.	FMUL TM, MD	"Do $C_R * W_R$
Do the	5.	FMUL TM, MD; DECTMA	"Do $C_R * W_I$ fetch W_I
multiplies	6.	FMUL TM, MD	"Do $C_I * W_I$
	7.	FMUL TM, MD; DPX (0)<FM	"Do $C_I * W_R$, Save $C_R W_R$, In DPX
	8.	FMUL; DPX (1)<FM	"Save $C_R W_I$ in DPX
Do the 2	9.	FMUL; FSUBR FM, DPX (0)	"Do $X_R + C_R W_R - C_I W_I$
adds.	10.	FADD FM, DPX (1)	"Do $X_I = C_R W_I + C_I W_R$
	11.	DPX (0)<FA; FADD	X_R is ready, save in DPX
	12.	DPX (1)<FA	X_I is ready, save in DPX

The total elapsed time is 12 cycles or 2us. In practice, however, we can overlap all but cycles 4-7 with the preceding and following computations. The complex multiply then takes us only 667ns, when mixed in with other computations.

Below is a summary chart of the complex multiply:

	Memories		Multiplier		Adder		Data Pad	
Cycle	TM	MD	M1,M2	FM	A1,A2	FA	0	1
1.	---	---	---	---	---	---	---	---
2.	---	---	---	---	---	---	---	---
3.	---	---	---	---	---	---	---	---
4.	W_R	C_R	W_R,C_R	---	---	---	---	---
5.	W_I	C_R	W_I,C_R	---	---	---	---	---
6.	W_I	C_I	W_I,C_I	---	---	---	---	---
7.	W_R	C_I	W_R,C_I	W_R*C_R	---	---	$W_R C_R$	---
8.	---	---	---	W_I*C_R	---	---	$W_R C_R$	$W_I C_R$
9.	---	---	---	W_I*C_I	$W_I C_I$,$W_R C_R$	---	$W_R C_R$	$W_I C_R$
10.	---	---	---	W_R*C_I	$W_R C_I$,$W_I C_R$	X_R	$W_R C_R$	$W_I C_R$
11.	---	---	---	---	---	X_I	X_R	$W_I C_R$
12.	---	---	---	---	---		X_R	X_I

CHAPTER 5

HOW TO PROGRAM THE AP-120B

5.1 MEET THE AP.....AGAIN

5.1.1 Introduction

The purpose of this chapter is to illustrate the way to use the AP most efficiently, i.e., to write good loops. It assumes that the reader has already read the Software Development Package Manual (APAL, Sections 2 and 3) and has at least a passing acquaintance with the AP instruction set.

This chapter presents a short review of the basic elements of the Array Processor from the programmer's point of view, covers methods and techniques of writing loops and suggests some common pitfalls to avoid.

It reviews some of the basic AP instructions; it is not meant to be all-inclusive but to briefly cover the most-often-used things.

This chapter assumes the use of the AP's 333 ns interleaved memory.

5.1.2 Basic Overview

5.1.2.1 Arithmetic

Both the Floating Adder and Floating Multiplier need explicit instructions (e.g., FADD and FMUL, respectively) to push their respective answers out of the pipelines. Given these "pushers", the Floating Adder result (FA) will be available two cycles after the original instruction, and the Floating Multiplier result (FM) will be available three cycles after the original instruction:

```
0.  FADD DPX, DPY   "add              0.  FMUL DPX, DPY   "multiply
1.  FADD            "push             1.  FMUL            "push
2.  DPX(1)<FA       "store answer     2.  FMUL            "push
                                      3.  DPY(1)<FM       "store answer
```

The empty FADD and FMUL "pushers" can also be read Adder or Multiplier operations, thus producing new answers each cycle.

If the "pushers" do not directly follow the original instructions, FA will come out one cycle after the first FADD pusher, and FM will come out one cycle after the second FMUL pusher. Both FA and FM will remain available for succeeding cycles until a new FA or FM is pushed out.

The arguments for Adder and Multiplier instructions consist of one from column A and one from column B, (in that order):

COLUMN A (A1 or M1)	COLUMN B (A2 or M2)
FM	FA
TM	MD
DPX	DPX
DPY	DPY

The Adder has additional arguments of ZERO and NC (no change), which can be used in either or both columns.

5.1.2.2 Main Data Memory

Reading from memory requires one of the following instructions: SETMA, INCMA, DECMA, or LDMA. In practice, it is usually done by the SETMA instruction. The result, MD, comes out three cycles later and is also available for succeeding cycles until a new MD comes out. No "pushers" are needed. Writing into memory requires one of the above instructions plus MI<source, where source is FA, FM or DB. This goes on the same line as SETMA, and gets done in that cycle. Memory can be referenced every two cycles, for either a read or write.

5.1.2.3 Table Memory

Table memory is usually referenced by the SETTMA or LDTMA instruction. Two cycles later, TM is available and remains so until two cycles after the next instruction affecting TMA. Such instructions can occur in every cycle, producing a new TM every cycle.

5.1.2.4 Data Pad

DPX and DPY each contain 32 registers, eight of which are accessible from any given DPA. That is, one can reference DPX from DPX(DPA-4) to DPX(DPA+3), and similarily for DPY.

The Data Pad Bus is usually used to store data from memory or from one Data Pad register into another, or to utilize a value, e.g., in conjunction with a load operation:

```
DPX(1)<DB; DB=DPY(-2)          This can be shortened to
                               DPX(1)<DPY(-2).)

DPX<DB; DB=MD                  (Or DPX<MD)

LDDPA; DB=3                    (This sets DPA=3)
```

Storing into Data Pad from FA or FM does not use the Data Pad Bus. This is important, as it leaves DB free for other uses.

5.1.2.5 S-Pad

S-Pad registers are usually used as address pointers or counters, and thus to pass parameters to a program. An S-Pad operation must accompany a SETMA (or SETDPA, SETTMA, etc.) instruction. An S-Pad operation must also precede a conditional branch (BGT, BNE, etc.) by one cycle. That is, conditional branches are based on the S-Pad Function (SPFN) of the S-Pad operation in the previous cycle.

The fastest way to get an integer into S-Pad is to use the LDSPI instruction:

```
LDSPI COUNT; DB=5
```

This puts 5 into an S-Pad register called COUNT. The value is assumed to be octal unless a decimal point is added. DB=15. (note point) is equivalent to DB=17 (octal), or to DB=0FX (hex). Hexadecimal numbers must start with a numeric digit and end with "X".

Although the Floating Adder operation FSUB A1, A2 will do A1-A2, the S-Pad operation SUB subtracts in the opposite direction, i.e., SUB PIECE, TOTAL will do:

(contents of S-Pad TOTAL) minus (contents of S-Pad PIECE).

5.1.3 Referencing Memory

In order to read something out of memory, or write into it, the location in memory where this will occur must be provided. The SETMA instruction gets this necessary information from the S-Pad Function (SPFN) of the same cycle. Therefore, one needs to construct an S-Pad operation which will result in a pointer to the appropriate memory location. Generally, this takes the form of adding increments to pointers. For example, if there was a 4-element vector in memory locations 100, 102, 104, 106, one would need an S-Pad register (say, APTR) containing the base address (100), and another S-Pad register (AINC) containing the increment between elements (2). Then, if one wanted to read the element in location 102, the appropriate instruction would be ADD AINC, APTR; SETMA. Now APTR would contain 102. If one wrote another ADD AINC, APTR; SETMA the contents of memory location 104 would be read.

Consider the following instruction: MOV APTR, APTR. This doesn't seem to accomplish much, but in the light of the above discussion, it can be seen that its SPFN could be useful for a SETMA. This is how one would get the first element of a vector.

All of the above is correspondingly true for writing into memory.

5.1.4 S-Pad Mnemonics

S-Pad names such as APTR, AINC, N are really only temporary names for the 16 S-Pad registers. A statement such as DEC N will not mean anything to the assembler unless the program has equated the mnemonic "N" with a specific S-Pad register, such as S-Pad 0. This is done by the following assembler pseudo-op: N $EQU 0. All S-Pad names used in a program must be declared in this manner before using them in an instruction. Thus, programs generally begin with lists like:

```
APTR $EQU 0
AINC $EQU 1
BPTR $EQU 2
BINC $EQU 3
N    $EQU 4
     .
     .
     .
```

These S-Pad numbers should not be confused with the contents of the S-Pads. ADD BINC, BPTR would not add 3 to 2 (using the above list), but would add the contents of S-Pad 3 to the contents of S-Pad 2.

There can be more than one name for an S-Pad register. If you had two different vectors, A and B, and wished to use the mnemonics AINC and BINC for their increments, you could use the same S-Pad register if the increment for both is the same in all cases, by declaring:

```
AINC $EQU 1
BINC $EQU 1
```

5.1.5 Other Pseudo-Ops

Besides the $EQU pseudo-op, the typical program includes $TITLE and $ENTRY pseudo-ops at the very beginning, and an $END at the very end. A basic program with one loop would have the following form:

```
                                    $TITLE name
                                    $ENTRY name
                    S-Pad mnemonic    $EQU  0
                                      .     1
                                      .     2
                                      .     .
                                      .     .
                                      .     .
name:   (code)
          "
          "                 ("intro" to loop and any initializations
          "                   and pointer adjustments)
          "

loop:   (code)
          "
          "
          "

                                            $END
```

See the software manual for explanations of these pseudo-ops.

5.2 LOOPS

5.2.1 A Poor Loop

The loop is where the potential of the AP comes into full bloom. For example, one way (lengthy but workable) to write a dot product program is as follows:

Given: Vectors A and B in Main Data Memory, with elements of each vector in equally spaced locations in memory (e.g., even-numbered locations).

Produce:

$$c = \sum_{m=1}^{N} A(m).B(m)$$

Parameters passed in S-Pad:

S-Pad Name	Contains:
APTR	base address of vector A
BPTR	base address of vector B
XINC	increment (number of locations from one element to the next) (same for both vectors)
N	number of elements in each vector
CPTR	address of answer

```
DOTPROD:  SUB XINC, APTR (See Note below)
          SUB XINC, BPTR (See Note below)
          FADD ZERO, ZERO          "initialize FA=0
          FADD
LOOP:     ADD XINC, APTR; SETMA    "get mth element of vector A
          NOP                      "  from memory
          NOP
          DPX<MD                   "MD=A(m), store into DPX
          ADD XINC< BPTR; SETMA    "get mth element of vector B
          NOP
          NOP
          FMUL DPX, MD             "MD=B(m), do A(m).B(m)
          FMUL
          FMUL
          FADD FM, FA              "add product to sum of products
          FADD
          DEC N                    "decrement counter
          BGT LOOP                 "branch back if not done yet
                                   "(i.e. if N>0)
DONE:     MOV CPTR, CPTR; SETMA; MI<FA
                                   "otherwise, store answer
```

NOTE

This is so that the first time through the loop, ADD XINC, APTR and ADD XINC< BPTR will not move the pointer to the second element, passing up the first altogether.

To begin with, this program can certainly be shortened by combining instructions and overlapping memory fetches. Thus:

```
DOTPROD:  FADD ZERO, ZERO; SUB XINC, APTR
          FADD; SUB XINC, BPTR
LOOP:     ADD XINC, APTR; SETMA     "get A(m)
          NOP
          ADD XINC, BPTR; SETMA     "get B(m)
          DPX<MD                    "store A(m) in DPX
          NOP
          FMUL DPX, MD              "do A(m).B(m)
          FMUL
          FMUL
          FADD FM, FA; DEC N        "add product to sum of products
                                    "  and decrement counter
          FADD; BGT LOOP            "test if done. If not, branch
                                    "          to LOOP
DONE:     MOV CPTR, CPTR; SETMA; MI<FA
                                    "if so, store answer
```

Note the extra FMUL's and FADD's, described as "pushers". These push the answers through the pipelines, so that FM and FA will contain what they are intended to contain. This is pointed out because the beginning AP programmer is likely to forget to put "pushers" in his code.

Now the loop of the evolving dot product program is ten cycles long. This means that each new pair of elements costs ten more cycles. Although better than the initial example, which had a 14-cycle loop, this can actually be cut down to a mere four cycles!

5.2.2 Determining Length of Loop

One might suppose that the length of a program loop depends on what one is trying to do. This is true, but not in the way one would think. The AP programmer decides ahead of time how many cycles his loop should contain, and then fits everything into that framework. How does he pick the magic number? Most commonly, loops are memory-limited. Recall that one can reference memory (to read or to store) every two cycles. If one has two memory references to do (e.g., "get A" and "get B"), then the loop will be at least four cycles long (two per memory reference). And, unless one has more than four different FMUL's, four different FADD's, or four different S-Pad operations to do, the loop should be, at MOST, four cycles. A lot can be done in four cycles when one can do a Floating Multiplier operation, a Floating Adder operation, an S-Pad operation, a branch, a memory reference, a Data Pad Bus transfer, etc., in EACH cycle.

5.2.3 Writing A Real Memory-Limited Loop

Before continuing with the transformation of the dot product program, another example will be utilized.

Given: Vectors A and B in Main Data memory, length=N elements

Produce: Vector C (in memory), where C(m)=A(m)2+B(m) for m=1 to N

Parameters:

S-Pad Name	Contains
APTR	base address of A
BPTR	base address of B
CPTR	base address of C
XINC	increment (same for all vectors in this example)
N	number of elements

Note that there should be three memory references in the loop: "get A", "get B", and "store C". (Unlike the dot product which accumulated a running sum in the Adder, this program needs to store an answer after each set of computations. For the dot product, storing was not a repeated process, and hence not included in its loop.) Three memory references, one every other cycle, means the loop would be six cycles long. It would start like this:

```
1) ---(nothing here, but count a cycle)
2) ADD XINC, APTR; SETMA      get A
3) ---
4) ADD XINC, BPTR; SETMA      "get B
5) DPX<MD                     "store A in DPX
6) FMUL DPX, MD               "do A*A
```

(The reason for starting on the second line will be explained later.)

Now it has run out of cycles, but there is still more to do, so it starts back up at the first cycle, which is where the end will branch to, when it gets around to testing if it's done.

```
LOOP:   1) ---                  FMUL                    "B is available
                                                         here, but
                                                        "not needed yet

        2) ADD XINC, APTR;      FMUL
           SETMA

        3) ---                  FADD FM, MD             "add B to A(2)

        4) ADD XINC, BPTR;      FADD
           SETMA

        5) DPX<MD               DEC N                   "answer is avail-
                                                           able here
                                                        "but can't re-
                                                         ference memory
                                                        "yet to store it

        6) FMUL DPX, MD         ADD XINC, CPTR; SETMA; MI<FA;
                                 BGT LOOP               "store answer and
                                                        "test if done
```

This is the entire loop. In its proper form, taking out lines and adding semicolons, it looks like this:

```
LOOP:  FMUL
       ADD XINC, APTR; SETMA; FMUL
       FADD FM, MD
       ADD XINC, BPTR; SETMA; FADD
       DPX<MD; DEC N
       FMUL DPX, MD; ADD XINC, CPTR; SETMA; MI<FA; BGT LOOP
```

5.2.4 Writing Intros

Notice, however, that if the program goes right into this loop, after initial overhead such as SUB XINC, APTR
SUB XINC, BPTR
SUB XINC, CPTR
it picks up the first element of A and B as it's supposed to, but it also stores something into C before it's ready to, and decrements the counter too early. IT GOES THROUGH BOTH COLUMNS AT THE SAME TIME. What is desired, however, is that computations in the second column continue from the first column. The only way it can do this is to continue from what the first column did in the PREVIOUS time through the loop. And the FIRST time, there was no previous time. Hence the need for additional microcode before getting into the loop.

Exactly what needs to go before the loop? In order for the second column of the loop to be doing what it's supposed to when the program gets to it, the first column must precede it. Essentially, one rewrites the first column as an "intro" to the loop. Thus:

```
PROGRAM:                    MOV APTR, APTR; SETMA   "get first element
                                                    "of A

                            SUB XINC, CPTR          "to offset ADD in
                                                        loop

                            MOV BPTR, BPTR; SETMA   "get first element
                                                    "of B

                            DPX<MD                  "store A(1) in DPX

                            FMUL DPX, MD            "do A(1)2

LOOP:                       FMUL

  ADD XINC, APTR; SETMA;    FMUL                    "get A(m+1)

                            FADD FM, MD             "do A(m)2+B(m)

  ADD XINC, BPTR; SETMA;    FADD                    "get B(m+1)

  DPX<MD;                   DEC N                   "store A(m+1)

  FMUL DPX, MD;             ADD XINC, CPTR; SETMA; MI<FA; BGT LOOP
                                                    "do A(m+1)2, store
                                                    "C(m), test if done


DONE:    RETURN
```

To clear up a loose end regarding the structure of memory-limited loops, one might notice that since the branch must be in the last cycle, the DEC N instruction must be in the second-to-last cycle. DEC is an S-Pad operation and cannot be in the same cycle as another S-Pad operation, such as ADD XINC, XPTR. A memory-limited loop has SETMA's (requiring S-Pad operations) on every other line. Since the DEC N operation will go on an odd-numbered line of the loop, the SETMA's must go on even-numbered lines. This is why the first thing to do,ADD XINC, APTR; SETMA (See section 5.2.3), was put on line 2.

5.2.5 Dot Product Program

It is now possible to write the four cycle dot product. Using the technique outlined above, the loop should be constructed as follows:

```
1) ---

2) ADD XINC, APTR; SETMA  "get A

3) ---

4) ADD XINC, BPTR; SETMA  "get B

                      then

1) ---                    DPX<MD        "store A

2) ADD XINC, APTR; SETMA  ---

3) ---                    FMUL DPX, MD  "do A.B

4) ADD XINC, BPTR; SETMA  FMUL

                      then

1) ---                    DPX<MD          FMUL

2) ADD XINC, APTR; SETMA  ---             FADD FM,FA "add A.B to sum
                                                     "of products

3) ---                    FMUL DPX, MD    FADD; DEC N"decrement
                                                      counter

4) ADD XINC, BPTR; SETMA  FMUL            BGT LOOP   "test if done
```

The intro to this three column loop will consist of the first column alone, then the first and second column together. Other overhead, such as initializing FA to 0, can be mixed in with the intro.

To generalize, an N-column loop would require an intro consisting of column 1 followed by columns 1 and 2 together, followed by columns 1, 2 and 3 together....followed by columns 1, 2,...,N-1 together.

```
$TITLE DOTPROD
$ENTRY DOTPROD

APTR $EQU 0
BPTR $EQU 1
CPTR $EQU 2
XINC $EQU 3
N    $EQU 4

DOTPROD:  MOV APTR, APTR; SETMA; FADD ZERO, ZERO  "get A(1) and
                                                  "initialize FA=0

          MOV BPTR, BPTR; SETMA; FADD             "get B(1)

          DPX<MD                                  "store A(1)

          ADD XINC, APTR; SETMA                   "get A(2)

          FMUL DPX, MD                            "do A(1)*B(1)

          ADD XINC, BPTR; SETMA; FMUL             "get B(2)

LOOP:     DPX<MD; FMUL                            "store A(m+1)

          ADD XINC, APTR; SETMA; FADD FM, FA      "get A(m+2), add
                                                  "A(m)B(m) to sum

          FMUL DPX, MD; FADD; DEC N               "do A(m+1)B(m+1)
                                                  "decrement counter

          ADD XINC, BPTR; SETMA; FMUL; BGT LOOP   "get B(m+2), test if
                                                  "done

DONE:     MOV CPTR, CPTR; SETMA; MI<FA;  RETURN   "if so, store answer

$END
```

Now each new pair of elements will cost four more cycles, because every four cycles a new pair is being fetched; every four cycles another product is added to the sum. The longer overhead is no disadvantage as it is only done once, and even if the program was called with N containing 1, making the streamlined loop unnecessary, it takes no longer than the unstreamlined program.

Note that there are two SETMA's in a row at the beginning and again at the end of the program. This will not cause any problems except to make memory spin, which is the memory's way of putting in the NOP's the programmer leaves out. The timing is still the same, and this way there are two less locations of Program Source used up.

It might be mentioned that if one were getting Vectors A and B out of Data Pad instead of memory, the dot product could be written with a one-cycle loop! This will be demonstrated later.

5.2.6 Notation

A few words about notation are in order. The "---" used when writing loops in column form simply denotes a blank spot, indicating a cycle goes by while awaiting the results of a memory fetch or while looking for a more propitious spot to use the results of the Adder or Multiplier, etc. Normally, something else will eventually go on the same line, in a different column.

Example: This takes vector A, multiplies it by a constant in DPX, and stores it in vector B.

```
1) ---                        FMUL DPX, MD

2) ADD XINC, APTR; SETMA      FMUL

3) ---                        FMUL; DEC N

4) ---                        ADD XINC, BPTR; SETMA; MI<FM; BGT LOOP
```

Since the length of the loop was already decided by the number of SETMA's, these blank spots cause no harm to the speed. It is the number of cycles in the loop, not the number of columns, which determines speed. Extra columns simply mean longer intros, which the program only goes through once anyway unless it's part of a nested loop.

In loops with several Adder or Multiplier operations, it often happens that one such instruction will be a "pusher" for another in another column.

```
1)  (code)                FMUL DPY, MD   (code)
2)    " FMUL DPX, DPY     FMUL                "
3)    " FMUL              FMUL                "
4)    " FMUL              DPY<FM              "
5)    " DPX(1)<FM         "
6)    "                                       "
```

In column 2, lines 2 and 3 are illegal, as those lines already contain FMUL's (which will do the pushing for column 2 as well as column 1). However, it may be advantageous to the programmer to note to himself somehow that FMUL's do belong there, in case things in the first column get moved around for some reason. This is the purpose of such notation as (fmul) or (fadd).

Thus:

```
1)  (code)                FMUL DPY, MD   (code)
2)    " FMUL DPX, DPY     (fmul)              "
3)    " FMUL              (fmul)              "
4)    " FMUL              DPY<FM              "
5)    " DPX(1)<FM                             "
6)    "                                       "
```

Now, if pieces of the first column were moved down a couple of lines for some reason,

```
1)  (code)              FMUL DPY, MD  (code)  DPX(1)<FM
2)    "                 (fmul)          "
3)    "                 (fmul)          "
4)    "FMUL DPX, DPY    DPY<FM          "
5)    "FMUL                             "
6)    "FMUL                             "
```

the programmer would be reminded to put real FMUL's back on those lines.

When writing loops with a small number of cycles, these reminders can also help one keep track of the columns, as in:

```
---                     ---  DPY<MD         FMUL    FADD FM, FA; DEC N
ADD XINC, APTR; SETMA   ---  FMUL, DPY, MD (fmul)  FADD; BGT LOOP
```

This gets a vector from memory, squares each element and adds the squares together (sort of a dot product between vector A and itself). The seemingly empty columns, which disappear when the loop is written in proper form (see below), are necessary in order to write the intro properly. If one left out the second column, for example, his intro would start with:

```
MOV APTR, APTR; SETMA
DPY<MD
ADD XINC, APTR; SETMA; FMUL DPY, MD
          .
          .
          .
```

Clearly, the first MD will not be the first element fetched. By the time it gets down to FADD FM, FA in the loop, something which doesn't belong will be added in.

This is what the intro and loop should look like:

```
        MOV APTR, APTR; SETMA
        FADD ZERO, ZERO          "initialize FA=0
        ADD XINC, APTR; SETMA; FADD
        DPY<MD
        ADD XINC, APTR; SETMA; FMUL DPY, MD
        DPY<MD; FMUL
        ADD XINC, APTR; SETMA; FMUL DPY, MD

LOOP:   DPY<MD; FMUL; FADD FM, FA; DEC N
        ADD XINC; APTR; SETMA; FMUL DPY, MD; FADD; BGT LOOP

        (answer)<FA
```

5.2.7 Dropping Out One Early

```
1) ---                          (code)                    (code)
2) ADD XINC, APTR; SETMA          "                         "
3) (code)                         "                         "
4) ADD XINC, BPTR; SETMA          "                         "
5) (code)                         "                         "
6)   "                          ADD XINC, CPTR; SETMA       "
7)   "                          (code)                      "  DEC N
8)   "                            "                       ADD XINC, DPTR; SETMA
                                                             MI<DPX; BGT LOOP
```

Here, there are two memory reads in the first column, one read in the second column, and a store in the last column. When writing the intro, the pointers should be taken care of as follows:

```
MOV APTR, APTR; SETMA
SUB XINC, DPTR; (code)
MOV BPTR, BPTR; SETMA
(code)
  .
  .
  .
ADD XINC, APTR; SETMA; (code)
(code)
ADD XINC, BPTR; SETMA;   "
(code)                   "
  "                    MOV CPTR, CPTR; SETMA
  "                    (code)
  "                      "
```

If the memory reference in the second column of the loop was a store instead of a read, the problem would become more complicated. By the time the counter went down to zero and the last result was stored at DPTR, an extra C would have been stored, possibly over a valuable piece of data, such as the beginning of vector D. Or if instead of ADD XINC, CPTR; SETMA; MI<DPY in the second column, we had DPY<FA (where FA is cumulative, as in the dot product) and later stored DPY into CPTR after getting out of the loop, an extra FA would have been computed and DPY would contain an incorrect answer. In this case, it would be wise to drop out of the loop one time early. One would put an extra DEC N somewhere in the intro, so that the loop would be done N-1 times. Then after the loop, write just the last column (not including DEC and the branch, of course), which is all that remains to be done from the loop anyway.

Example: This does a dot product of vectors A and B, and also outputs the square of each updated sum into vector D.

```
---                          FMUL DPX, MD  ---
ADD XINC, APTR; SETMA        (fmul)        FMUL DPY, DPY
---                          FMUL          (fmul )
ADD XINC, BPTR; SETMA        FADD FM, FA   FMUL
DPX<MD                       FADD          DEC N
---                          DPY<FA        ADD XINC, DPTR; SETMA; MI<FM;
                                             BGT LOOP
```

When it is going through the loop for the last time and storing the very last thing in D (column 3), it is also simultaneously doing extra executions of columns 1 and 2. Normally, that doesn't matter, but in this case, something extra is being added to the cumulative sum of the dot product (column 2), which was completed the previous time through the loop. By dropping out of the loop before its last time around, this error is avoided:

```
   MOV APTR, APTR; SETMA
   DEC N                    "to cause dropping out early
   MOV BPTR, BPTR; SETMA
   DPX<MD
   SUB XINC, DPTR           "to nullify the first ADD XINC, DPTR

                            FMUL DPX, MD
   ADD XINC, APTR; SETMA;   FMUL
                            FMUL
   ADD XINC, BPTR; SETMA    FADD FM, FA
   DPX<MD                   FADD
                            DPY<FA

 LOOP:                      FMUL DPX, MD
   ADD XINC, APTR; SETMA                    FMUL DPY, DPY
                            FMUL
   ADD XINC, BPTR; SETMA    FADD FM, FA     FMUL
   DPX<MD                   FADD;           DEC N
                            DPY<FA;         ADD XINC, DPTR; SETMA; MI<FM;
                                              BGT LOOP

 OUT:                                       FMUL DPY, DPY
                            MOV CPTR, CPTR; SETMA; MI<DPY; FMUL
                                            FMUL
                                            ADD XINC, DPTR; SETMA; MI<FM;
                                              RETURN
```

Notice that the (fmul) in column 2 became a real FMUL in the intro. OUT starts just the last column. The next line stores the completed dot product.

One might wish to come out one early even if one doesn't strictly need to, if the loop is long and there are only a couple of lines in the last column:

```
 1)  (code)       (code)
 2)    "  SETMA     "
 3)    "    "       " SETMA; MI<DPX
 4)    "
 5)    "
 6)    "  SETMA
 7)    "
 8)    "  SETMA
 9)    "
10)    "  SETMA
11)    "  DEC N
12)    "  SETMA   BGT LOOP
```

In this case, coming out of the loop one time early and adding on the last four lines afterward would save going through eight cycles for nothing.

5.2.8 Interaction Between Columns

In order to fit things into complicated loops without creating op-code conflicts, the AP programmer takes advantage of results (e.g. MD, FA) which are the same for one or more cycles after first available. Sometimes he will purposely delay the pushing of an answer through a pipeline by leaving out "pushers". But he must be careful of the way the columns interact with each other within the loop.

```
1) FMUL DPX, DPY        .
2)     .              FMUL DPY(3), DPX(2)
3)     .              FMUL
4) FMUL                 .
5) FMUL                DPY(1)<FM
6) DPX(1)<FM            .
```

The FMUL's in column 2 will act as "pushers" for the FMUL DPX, DPY in column 1, whose answer will come out on line 4 instead of line 6 as desired and will disappear forever when replaced by a new FM on line 5. Notice the FMUL on line 4 in column 1 acts as a pusher for column 2, which was planned for.

Another example:

```
1) (code)                   DPX<MD          (code)
2) ADD XINC, APTR; SETMA    (code)          DPX<FA
3) (code)                   FADD FM, DPX    (code)
4)   "                      (code)            "
```

The DPX of column 2, line 3 will not be the same as what was stored into it in column 2, line 1. It will be FA from column 3, line 2.

5.2.9 Changing DPA

Because one can access things in Data Pad much faster than things in memory, it makes sense to store things from memory into Data Pad if they will be used again. For example, if one is going to use an N-element vector for several different computations, one could store it in DPX(0), DPX(1),....,DPX(N-1). Because the Data Pad indices can only be accessed from -4 to +3 with a static DPA, it becomes useful to leave the index alone and change DPA.

Storing vector A in DPX is basically the repeated operation of DPX<MD; INCDPA. If DPA is initially set to zero, then the first element will be stored into DPX(0). INCDPA will increase DPA for the NEXT instruction.

```
    Thus:  DPX<MD; INCDPA    "refers to DPX(0)
           DPX<MD            "refers to DPX(1)
```

The ways to set DPA to zero:

```
    CLR# (S-Pad name); SETDPA              "uses up S-Pad field
                    or
           DB=ZERO; LDDPA                  "uses up Adder field
```

This loop will read a vector from memory into Data Pad X:

```
 ---                        ---   DPX<MD; INCDPA; DEC N
 ADD XINC, APTR; SETMA      ---   BGT LOOP
```

With intro:

```
           MOV APTR, APTR; SETMA
           CLR# APTR; SETDPA
           ADD XINC, APTR; SETMA
    LOOP:  DPX<MD; INCDPA; DEC N
           ADD XINC, APTR; SETMA; BGT LOOP
```

5.2.10 Non-Memory-Limited Loops

A non-memory-limited loop is a loop in which two times the number of memory references is less than the number of same-op-code-field operations required. For example, if there are five Floating Adder operations to be done (FADD, FSUB, FSUBR, etc.) but only two memory references (a fetch and a store), the five Adder operations cannot fit into four cycles.

Incidentally, "pushers" don't count in figuring out how many cycles are needed. In a five-cycle loop with five different Adder operations, the Adder instructions become each other's pushers.

Recall that in memory-limited loops, the first instruction in column 1 usually starts on line 2, to avoid S-Pad conflicts on the next-to-last line. (See last paragraph of Section 5.2.4). This is not necessary in non-memory-limited loops.

The following loop will test whether each element of a vector in DPY is within the range between a maximum limit and minimum limit. If so, the element is added to a cumulative sum. The maximum limit is conveniently located in MD, and the minimum limit in FM, by the grace of whatever program uses this loop. Neither FM nor MD change during this loop's execution.

```
FSUB DPY, MD      BFGT BIGGER          (fadd)
FSUB FM, DPY      BFGT SMALL           DPX<FA; DEC N
(fadd) INCDPA     FADD DPY(-1), DPX    BGT LOOP
```

Note that the BFGT instruction tests FA of the previous cycle.

5.2.11 A One-Cycle Loop

For the one-cycle dot product, it is assumed that the vectors are already in Data Pad, starting at DPX(0) and DPY(0) (where DPA=0). Obviously, vectors longer than 32 elements cannot be handled this way (or can only be handled in segments of 32 or less).

This is what the loop really looks like:

```
FMUL DPX, DPY; INCDPA  (fmul)  (fmul)  FADD FM, FA; DEC N  (fadd) BGT
                                                            LOOP
```

The FMUL and FADD instructions become their own "pushers".

```
$TITLE DOTPROD
$ENTRY DOTPROD

N $EQU 0        "number of elements in each vector
CPTR $EQU 1     "where to store answer

DPTPROD:  CLR# N; SETDPA                  "DPA=0
          FMUL DPX, DPY;                  "do A(1)*B(1)
             INCDPA;                      "DPA to 1
             DEC N                        "set drop out early
          FMUL DPX, DPY;                  "do A(2)*B(2)
             INCDPA                       "DPA to 2
          FMUL DPX, DPY;                  "do A(3)*B(3)
             INCDPA;                      "DPA to 3
             FADD ZERO, ZERO              "init. FA=0
          FMUL DPX, DPY;                  "do A(4)*B(4)
             INCDPA;                      "DPA to 4
             FADD FM, ZERO;               "A(1)B(1) in Adder
            DEC N                         "decrement counter
LOOP:     FMUL DPX< DPY;                  "do A(m)*B(m)
             INCDPA;                      "DPA to DPA+1
             FADD FM, FA;                 "add A(m-3)B(m-3) to sum
             DEC N;                       "decrement counter
             BGT LOOP                     "test if done

OUT:      DPX<FA; FADD                    "store cumulative FA
          FADD DPX, FA                    "add it to other cumulative FA
          FADD
          MOV CPTR, CPTR; SETMA; MI<FA;   "store answer
             RETURN
$END
```

This particular sort of loop has a problem with the Floating Adder, in that a cumulative FA needs at least two cycles to accumulate each new addition. Hence, the one-cycle loop is actually operating with two mutually exclusive cumulative FA's, interwoven with each other:

```
FADD FM, FA
FADD FM, FA
FADD FM, FA
FADD FM, FA
FADD FM, FA
```

At the end of all this, they (the two strings of sums) need to be added to each other. (see OUT, the label after LOOP).

This also illustrates the practice of dropping out of the loop one time early. If it didn't drop out early, the last (unneeded) FADD FM, FA of the loop would push out one of the two cumulative FA's. By the next cycle it would be gone forever. By dropping out early, DPX<FA can be done before it's too late.

This line of reasoning can eventually lead one to the idea that the last column of the loop (see beginning of Section 4.2.16) is unnecessary, since there is no way for the Adder result to come out in time for the next FADD FM, FA. The FADD FM, FA of each of the two strings of cumulative FA's will push out the other string. So the loop need only be of the form:

```
FMUL DPX, DPY; INCDPA   (fmul)   (fmul) DEC N   FADD FM, FA; BGT LOOP
```

This is one column less than before, which means that there will be one column's worth (in this case, one line) less to put in the intro. It will also not be necessary to come out of the loop one time early, as there is no extra FADD FM, FA to push away something needed. It is still necessary to add the two cumulative FA's together at the end.

```
$TITLE DOTPROD
$ENTRY DOTPROD

N     $EQU 0
CPTR  $EQU 1

DOTPROD:  CLR# N; SETDPA                  "DPA=0
          FMUL DPX, DPY;                  "A(1) * B(1)
             INCDPA;                      "   DPA to 1
             FADD. ZERO, ZERO                initialize cum. FA=0
          FMUL DPX, DPY;                  "A(2) *B(2)
             INCDPA;                      "   DPA to 2
             FADD ZERO, ZERO              "   initialize other cum. FA=0
          FMUL DPX, DPY;                  "A(3) * B(3)
             INCDPA;                      "   DPA to 3
             DEC N
LOOP:     FMUL DPX, DPY;                  "A(m) * B(m)
             INCDPA;                      "   DPA to DPA+1
             DEC N;                       "   decrement counter
             FADD FM, FA;                 "   add A(m-3)B(m-3) to cum. FA
             BGT LOOP                     "   test if done
OUT:      DPX<FA; FADD                    "store first cumulative FA
          FADD DPX, FA                    "add it to other cumulative FA
          FADD
          MOV CPTR, CPTR; SETMA; MI<FA;"store answer
             RETURN
$END
```

5.3 CAVEAT PROGRAMMER (LET THE PROGRAMMER BEWARE)

5.3.1 Calling Another Sub-Routine

The JSR instruction allows one program to utilize another program, for example the divide sub-routine (DIV). In order to do this, one must declare DIV external ($EXT DIV) so that the assembler and linker will know what to do with the otherwise undefined symbol. One must also save everything he will need when program execution gets back to his main program. Depending upon what was used in the called sub-routine, some things may remain untouched. Commonly one should not count on being able to leave things in the Adder or Multiplier. Parts of Data Pad may also be changed, or DPA may change. S-Pad will probably not remain inviolate. (Remember, it's the S-Pad register number, not name, which is important.) These things need to be checked before doing a JSR.

5.3.2 Illegal Instruction Sequences (not caught by APAL)

The following sequences of instructions have been found to work improperly. They are not flagged as errors by APAL and thus the programmer must be very careful to avoid them.

SEQUENCE	CODE EXAMPLE	PROBLEM
1) Two consecutive RETURN instructions.	JSR RETURN	Stack pointer out of step.
2) A two-cycle instruction (PS, PSODD, or PSEVEN field) followed by a JSR instruction.	RPSF JSR	Stack pointer out of step.
3) An instruction from the PS field followed by a BDBN or BDBZ instruction.	LPSL BDBN	The branch occurs based on DB=0.
4) HALT followed by JSR.	HALT JSR	Causes stack pointer overflow if repeated (stack pointer out of step).
5) Panel breakpoint before a JSR.		Stack pointer out of step.
6) Set flag Branch flag	SFL0 BFL0	Wait one instruction after setting before branch.
7) LDAPS BR APSTATUS	LDAPS BEQ	Wait one instruction after setting before branch.

5.3.3 Other Things To Watch Out For (caught by APAL)

The rest of this section consists of various short examples, cautions, and reminders.

```
DPX<MD; DPY<DPX(1)
```
Illegal. Data Bus is assigned twice. (The above is really `DPX<DB; DB=MD; DPY<DB; DB=DPX(1).`)

```
DPX<MD; DPY<MD is legal.  (DB=MD; DPX<DB; DPY<DB)
```

```
DPX<FA; DPY<FM
```
Legal. FA and FM don't use the Data Bus.

```
DPX<FA; DPY<FM; DPX(1)<MD
```
Illegal. Data Pad X is being written into twice (different indices). Within each cycle, there should be no more than one of each of the following:

- write into DPX
- write into DPY
- read from DPX
- read from DPY

The exception is when reading out of Data Pad more than once but using the same index:

- `FADD DPX, FA; FMUL DPX, FA; DPY(1)<DPX` is legal.
- `FADD DPY, DPY; FMUL DPY, DPY` is legal.
- `FADD DPX, FA; FMUL DPX(1), FA` is not legal.

```
FADD DPX, DPY; DPY<MD
```
The old value of DPY, before MD replaces it, is used in the sum.

```
DB=4; LDSPI XINC; LDDPA; DPX(2)<FM
```
Both DPA and the contents of XINC will become 4, but the previous DPA is used in referencing DPX(2).

```
SUB# XINC, APTR; BGT OUT
```
Illegal. The # uses the condition field (branch).

APPENDICES

APPENDIX A

GLOSSARY

A	I.O Device Condition "A" Flag
A1	Floating Adder Input Register #1
A2	Floating Adder Input Register #2
ALU	Arithmetic - Logic Unit
APAL	A.P. Assembly Language (S/W package)
APARTH	A.P. Arithmetic Test (S/W package)
APBUG	A.P. Debugger (S/W package)
APEX	A.P. Executive (S/W package)
APLINK	A.P. Linker (S/W package)
APMA	AP-120B Memory Address Register
APMATH	A.P. Math Library (S/W package)
APMAX	Maximum-Positive Floating-Point Number
APNMAX	Maximum-Negative Floating-Point Number
APPATH	A.P. Path Tester (S/W package)
APSIM	A.P. Simulator (S/W package)
APSTATUS	A.P. Status Register
APTEST	A.P. Tester (S/W package)
B	I/O Device Condition "B" Flag
CB	Control Buffer (command)
CTL	Control Register
DA	I/O Device Address
DB	Data Pad Bus
DMA	Direct Memory Access
DP	Data Pad Group
DPA	Data Pad Address Register
DPBS	Data Pad Bus Field, Data Pad Bus
DPX	Data Pad X Registers
DPY	Data Pad Y Registers
DST	Destination Register
EFA	Effective Address
FA	Floating Adder Output Register, Floating Adder Result
FADDR	Floating Adder
FM	Floating Multiplier Output Register, Floating Multiplier Output
FFT	Fast Fourier Transform
FMT	Formatter
FMULR	Floating Multiplier

FN	Function Register
FPN	Floating Point Number
HHMA	High Host Memory Address
HMA	Host Memory Address Register
IFFT	Inverse Fast Fourier Transform
INBS	I/O Input Bus
INTRQ	Interrupt Request Flag
IODRDY	I/O Data Ready Flag
JSRS	Jumps To Subroutines
LIFO	Last-In--First-Out
LITES	Lights Register
LSB	Least Significant Bit
M1	Floating Multiplier Input Register #1
M2	Floating Multiplier Input Register #2
MA	Memory Address Register
MAE	Memory Address Extension
MD	Data Memory
MI	Memory Input
MSB	Most Significant Bit
PANEL	AP Virtual Front Panel
PNLBS	Panel Bus
PS	Program Source Memory
PSA	Program Source Address Register
SP	Scratch Pad Register
SP(SPD)	S-Pad Destination
SP(SPS)	S-Pad Source
S-PAD	Scratch Pad (See also SP)
SPD	Scratch Pad Destination Address Register
SPFN	S-Pad Function
SRA	Subroutine Return Address
SRAO	Subroutine Return Address Overflow
SRC	Source Register
SRS	Subroutine Return Stack
SVCRT	Service Routine
SWR	Panel Switch Register
TM	Table Memory
TMA	Table Memory Address Register
TMRAM	Writable Table Memory (Random Access)
VALUE	Command Buffer Value
WC	Word Count Register

APPENDIX B

LIST OF TERMS AND USAGE

The following terms and abbreviations are used throughout Part III to facilitate instruction descriptions.

TERM/ABBREVIATION	MEANING	USAGE
<>	Optional Instruction Mnemonic	Elements or operands contained within are optional. e.g. ADD < sh > < # > < & > 5,6 If no <> , then listed operands are mandatory.
&	BIT-REVERSE Operator	When specified, this symbol immediately preceeds the SP_{SPS} operand. e.g. ADD &sub, base (See S-PAD SUMMARY, SPFN MODIFIERS)
sh	S-PAD Shift Operator	Indicates one of 4 S-PAD shift options: e.g. SUB < sh > 5,6 (See S-PAD SUMMARY, SPFN MODIFIERS)
#	S-PAD NO-LOAD Operator	Indicates that normal (SPFN)→ SP_{SPD} operation in inhibited. e.g. MOV# 5,5
sps	S-PAD Source Register	Indicates currently designated S-PAD DESTINATION REGISTER. e.g. ADD sps, spd (See S-PAD SUMMARY, S-PAD OPERANDS)

targ	BRANCH TARGET ADDRESS	An address contained in the DISP field of the instruction word within a range of -20_8 to $+17_8$ locations relative to the current program location; by name, number, or expression. e.g. BR targ (See SPEC SUMMARY, Test, Branch, and Jump Operations).
adr	ADDRESS	Address contained in Value field of instruction word. May be used for both absolute and relative addressing operations. (See SPEC SUMMARY, Test, Branch, and Jump Operations). e.g. JMP adr
idx	DATA-PAD Index	DATA-PAD location specifier; by name, number, or expression, within -4 to +3 locations relative to current DPA. e.g. DPY(idx)<FA
val	value	Numeric value contained in the VALUE field of the current instruction word.
a1	A1 source operand	Indicates currently selected A1 REGISTER source operand.
a2	A2 source operand	Indicates currently selected A2 REGISTER source operand. e.g. FADD a1, a2

m1	M1 source-operand	Indicates currently selected M1 register source-operand.
m2	M2 source-operand	Indicates currently selected M2 register source-operand e.g. FMUL m1,m2

APPENDIX C

LIST OF FUNCTIONS

FUNCTION OPERATOR	FUNCTION	MEANING
+	PLUS	Additive; A plus B
-	MINUS	Subtractive; A plus B
EQV	EQUIVALENCE (NOT Exclusive OR)	If A=B, then C=1; if A=B, then C=0.
AND	AND	Indicates that both elements must be present to satisfy the test condition.
OR	OR	Indicates that any one or both element(s) present satisfies the test condition.
( )	Contents of a Register or Data currently enabled onto a bus.	Used to describe Register Transfers in the following manner: (A2) DPX(idx) Meaning: The contents of A2 register replace the contents of DPX(idx).
superscripts	indicates portion of larger elements.	Example: $Q1_{PS}$ Meaning: Quarter-one (bits 16-31) of the PROGRAM-SOURCE word. Note: To avoid double-superscripting, the following convention is observed: $Q1_{PS}$ bits 23-27 Meaning: Bits 23 through 27 of the QUARTER-ONE portion of the PROGRAM-SOURCE word.

arg subscripts	logical complement "as addressed by"	e.g., SP(SPS) Example: PS TMA Meaning: PROGRAM-SOURCE word at the location pointed to by the current contents of TABLE MEMORY ADDRESS REGISTER (TMA). Note: For addressing subscripts, the ( ) operator is implied.

APPENDIX D

INSTRUCTION FIELD LAYOUT AND SUMMARY

AP-120B INSTRUCTIONS

Unconditional Fields

Each of the following fields may be used in any given instruction word.

Octal Code	Field Name										Octal Code
	B	SOP	SOP1	SH	SPS	SPD	FADD	FADD1	A1	A2	
Ø	NOP	SOP1	NOP	NOP	(S-Pad	(S-Pad	FADD1	NOP	NC	NC	Ø
1	&	SPEC	WRTEXP	L	Source	Dest.	FSUBR	FIX	FM	FA	1
2		ADD	WRTHMN	RR	Reg.)	Reg.)	FSUB	FIXT	DPX	DPX	2
3		SUB	WRTLMN	R			FADD	FSCLT	DPY	DPY	3
4		MOV	NOP		(0-17)	(0-17)	FEQV	FSM2C	TM	MD	4
5		AND	NOP				FAND	F2CSM	ZERO	ZERO	5
6		OR	NOP				FOR	FSCALE	ZERO	MDPX	6
7		EQV	NOP				IO	FABS	ZERO	EDPX	7
1Ø			CLR								1Ø
11			INC								11
12			DEC								12
13			COM								13
14			LDSPNL								14
15			LDSPE								15
16			LDSPI								16
17			LDSPT								17

Octal Code	Field Name										Octal Code
	COND	DISP	DPX	DPY	DPBS	XR	YR	XW	YW	FM	
Ø	NOP	(Branch	NOP	NOP	ZERO	(DPX	(DPY	(DPX	(DPY	NOP	Ø
1	#	Displa-	DB	DB	INBS	Read	Read	Write	Write	FMUL	1
2	BR	cement)	FA	FA	VALUE*	Index)	Index)	Index)	Index)		2
3	BINTRQ	(0-37)	FM	FM	DPX						3
4	BION				DPY	(0-7)	(0-7)	(0-7)	(0-7)		4
5	BIOZ				MD						5
6	BFPE				SPFN						6
7	RETURN				TM						7
1Ø	BFEQ										1Ø
11	BFNE										11
12	BFGE										12
13	BFGT										13
14	BEQ										14
15	BNE										15
16	BGE										16
17	BGT										17

Octal Code	Field Name						Octal Code
	M1	M2	MI	MA	DPA	TMA	
Ø	FM	FA	NOP	NOP	NOP	NOP	Ø
1	DPX	DPX	FA	INCMA	INCDPA	INCTMA	1
2	DPY	DPY	FM	DECMA	DECDPA	DECTMA	2
3	TM	MD	DB	SETMA	SETDPA	SETTMA	3

* This instruction uses a 16-bit immediate VALUE as a constant or address (in bits 48-63 of this instruction). The YW, FM, M1, M2, MI, TMA and DPA fields are then disabled for this instruction word.

SPEC Fields

One of the SPEC Fields may be used per instruction word. The S-PAD Fields (B, SOP, SOP1, SH, SPS, and SPD) are then disabled for this instruction.

Octal Code	Field Name								Octal Code
	SPEC	STEST	HOSTPNL	SETPSA	PSEVEN	PSODD	PS	SETEXIT	
Ø	STEST	BFLT	PNLLIT	JMPA*	RPSØA*	RPS1A*	RPSLA*	NOP	Ø
1	HOSTPNL	BLT	DBELIT	JSRA*	RPS2A*	RPS3A*	RPSFA*	SETEXA*	1
2	SPMDA	BNC	DBHLIT	JMP*	RPSØ*	RPS1*	RPSL*	NOP	2
3	NOP	BZC	DBLLIT	JSR*	RPS2*	RPS3*	RPSF*	SETEX*	3
4	NOP	BDBN	NOP	JMPT	RPSØT	RPS1T	RPSLT	NOP	4
5	NOP	BDBZ	NOP	JSRT	RPS2T	RPS3T	RPSFT	SETEXT	5
6	NOP	BIFN	NOP	JMPP	NOP	NOP	RPSLP	NOP	6
7	NOP	BIFZ	NOP	JSRP	NOP	NOP	RPSFP	SETEXP	7
1Ø	SETPSA	NOP	SWDB	NOP	WPSØA*	WPS1A*	LPSLA*	NOP	1Ø
11	PSEVEN	NOP	SWDBE	NOP	WPS2A*	WPS3A*	LPSRA*	NOP	11
12	PSODD	NOP	SWDBH	NOP	WPSØ*	WPS1*	LPSL*	NOP	12
13	PS	NOP	SWDBL	NOP	WPS2*	WPS3*	LPSR*	NOP	13
14	SETEXIT	BFLØ	NOP	NOP	WPSØT	WPS1T	LPSLT	NOP	14
15	NOP	BFL1	NOP	NOP	WPS2T	WPS3T	LPSRT	NOP	15
16	NOP	BFL2	NOP	NOP	NOP	NOP	LPSLP	NOP	16
17	NOP	BFL3	NOP	NOP	NOP	NOP	LPSRP	NOP	17

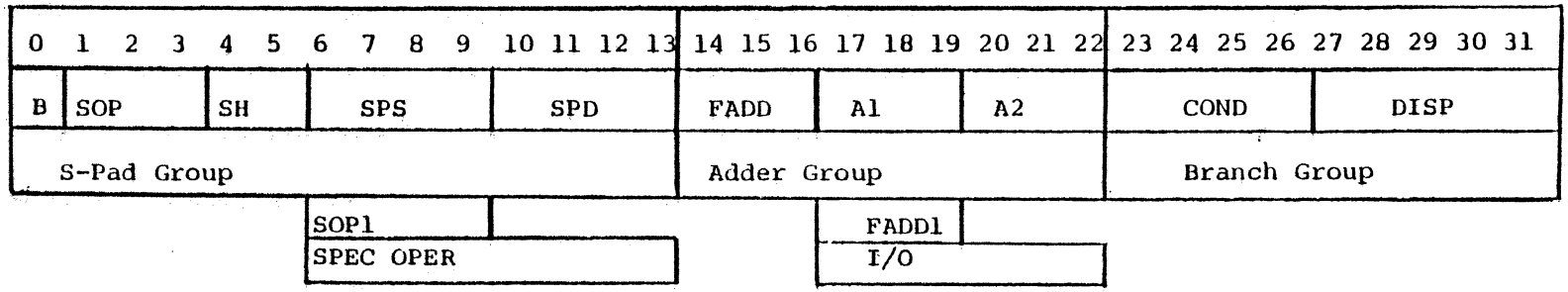
I/O Fields

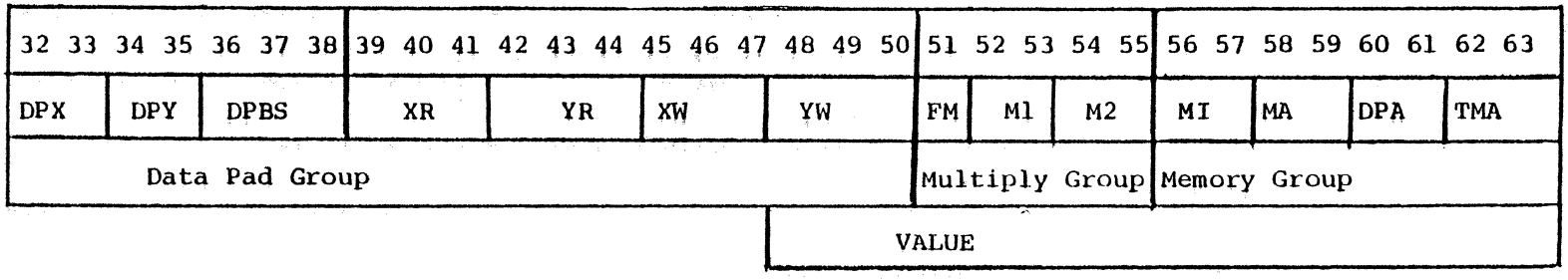
One of the I/O fields may be used per instruction word. The Floating Adder Fields (FADD, FADD1, A1, and A2) are then disabled for this instruction word.

Octal Code	Field Names							Octal Code
	IO	LDREG	RDREG	INOUT	SENSE	FLAG	CONTROL	
Ø	LDREG	NOP	RPSA	OUT	SNSA	SFLØ	HALT	Ø
1	RDREG	LDSPD	RSPD	SPNOUT	SPINA	SFL1	IORST	1
2	SPMDAV	LDMA	RMA	OUTDA	SNSADA	SFL2	INTEN	2
3	REXIT	LDTMA	RTMA	SPOTDA	SPNADA	SFL3	INTA	3
4	INOUT	LDDPA	RDPA	IN	SNSB	CFLØ	REFR	4
5	SENSE	LDSP	RSPFN	SPININ	SPINB	CFL1	WRTEX	5
6	FLAG	LDAPS	RAPS	INDA	SNSBDA	CFL2	WRTMAN	6
7	CONTROL	LDDA	RDA	SPINDA	SPNDBA	CFL3	NOP	7

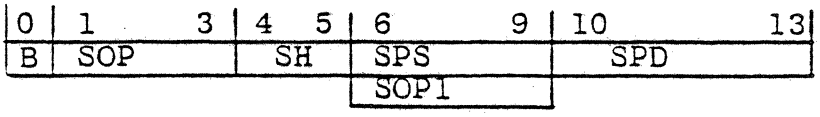
* This instruction uses a 16-bit integer VALUE (in bits 48-63 of the instruction word). The YW, FM, M1, M2, MI, MA, TMA, and DPA Fields are then disabled for this instruction word.

AP-120B Instruction Field Layout



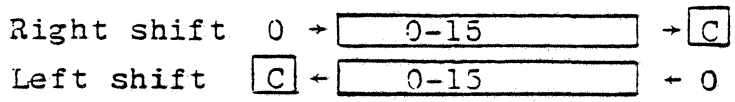


S-PAD GROUP



Field	Octal Code	Mnemonic	Effect
B	Ø	-	No-Op
	1	&	Use SP_{SPS} (bit-reversed)
SOP	Ø	-	See SOP1 field
	1	-	See Special Operations Group
	2	ADD	$(SP_{SPD})+(SP_{SPS})\rightarrow SPFN$
	3	SUB	$(SP_{SPD})-(SP_{SPS})\rightarrow SPFN$
	4	MOV	$(SP_{SPS})\rightarrow SPFN$
	5	AND	$(SP_{SPD})AND(SP_{SPS})\rightarrow SPFN$
	6	OR	$(SP_{SPD})OR(SP_{SPS})\rightarrow SPFN$
	7	EQV	$(SP_{SPD})\overline{XOR}(SP_{SPS})\rightarrow SPFN$
*SH	Ø	-	No-Op
	1	L	SPFN*2→SPFN (left shift)
	2	RR	SPFN÷4→SPFN (double right shift)
	3	R	SPFN÷2→SPFN (right shift)
SPS	$0\text{-}17_8$	$0\text{-}17_8$	S-Pad Source Operand Address
SPD	$0\text{-}17_8$	$0\text{-}17_8$	S-Pad Destination Address, $SPFN\rightarrow SP_{SPD}$ unless inhibited by No Load (COND=1)

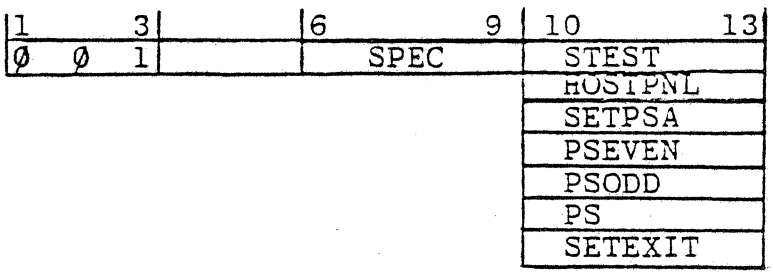
*Note: These are logical shifts:



Field	Octal Code	Mnemonic	Effect
SOP1	Ø	-	No-Op
	1	WRTEXP	Restricts DPX, DPY & MI fields to Write Exponent Only
	2	WRTHMN	Restricts DPX, DPY & MI fields to Write High Mantissa Only (Bits 00-11)
	3	WRTLMN	Restricts DPX, DPY & MI fields to Write Low Mantissa Only (Bits 12-27)
	4	-	-
	5	-	-
	6	-	-
	7	-	-
	10	CLR	Ø→SPFN
	11	INC	$(SP_{SPD})+1 \rightarrow SPFN$
	12	DEC	$(SP_{SPD})-1 \rightarrow SPFN$
	13	COM	$\overline{(SP_{SPD})} \rightarrow SPFN$
	14	LDSPNL	$SP_{SPD} \rightarrow SPFN$, $PNLBS \rightarrow SP_{SPD}$
	15	LDSPE	$SP_{SPD} \rightarrow SPFN$, $DPBS^{E} - 512 \rightarrow SP_{SPD}$
	16	LDSPI	$SP_{SPD} \rightarrow SPFN$, $DPBS^{ML} \rightarrow SP_{SPD}$
	17	LDSPT	$SP_{SPD} \rightarrow SPFN$, $DPBS^{MT} \rightarrow SP_{SPD}$

MH=Mantissa High=Mantissa bits 00-11
ML=Mantissa Low=Mantissa bits 12-27
MT=Mantissa bits for table lookups=Mantissa bits 02-08
E=Exponent

SPECIAL OPERATIONS GROUP



Field	Octal Code	Mnemonic	Effect
SPEC	Ø	-	See STEST Field (B-6)
	1	-	See HOSTPNL Field (B-7)
	2	SPMDA	Spin until MD available
	3	-	-
	4	-	-
	5	-	-
	6	-	-
	7	-	-
	10	-	See SETPSA Field, inhibit TEST except No Load (B-8)
	11	-	See PSEVEN Field (B-9)
	12	-	See PSODD Field (B-10)
	13	-	See PS Field (B-11)
	14	-	See SETEXIT Field (B-12)
	15	-	-
	16	-	-
	17	-	-

Field	Octal Code	Mnemonic	Effect
STEST	Ø	BFLT	Branch if FA<Ø.Ø
	1	BLT	Branch if SPFN<Ø
	2	BNC	Branch if S-Pad carry bit=1
	3	BZC	Branch if S-Pad carry bit=Ø
	4	BDBN	Branch if DPBS<Ø.Ø
	5	BDBZ	Branch if DPBS positive and unnormalized
	6	BIFN	Branch if Inverse FFT flag=1
	7	BIFZ	Branch if Inverse FFT flag=0
	10	--	--
	11	--	--
	12	--	--
	13	--	--
	14	BFLØ	Branch if Flag Ø=1
	15	BFL1	Branch if Flag 1=1
	16	BFL2	Branch if Flag 2=1
	17	BFL3	Branch if Flag 3=1

If the above specified condition is true OR the condition specified in the COND field is true, a branch occurs to (PSA)+DISP-20

Field	Octal Code	Mnemonic	Effect
HOSTPNL	Ø	PNLLIT	PNLBS →LITES
	1	DBELIT	$DPBS^{E}$→PNLBS→LITES
	2	DBHLIT	$DPBS^{MH}$→PNLBS→LITES
	3	DBLLIT	$DPBS^{ML}$→PNLBS→LITES
	4	-	-
	5	-	-
	6	-	-
	7	-	-
	10	SWDB	(SWR) →PNLBS→DPBS
	11	SWDBE	(SWR) →PNLBS→$DPBS^{E}$ and WRTEXP*
	12	SWDBH	(SWR) →PNLBS→$DPBS^{MH}$ and WRTHMN *
	13	SWDBL	(SWR) →PNLBS→$DPBS^{ML}$ and WRTLMN *
	14	-	-
	15	-	-
	16	-	-
	17	-	-

*Restrict DPS, DPY and MI to:

WRTEXP: Write Exponent only
WRTHMN : Write High Mantissa only (bits 00-11)
WRTLMN : Write Low Mantissa only (bits 12-27)

MH=Mantissa High=Mantissa bits 00-11
ML=Mantissa Low =Mantissa bits 12-27
E=Exponent

Field	Octal Code	Mnemonic	Effect
SETPSA	Ø	JMPA	VALUE→PSA
	1	JSRA	(SRA)+1→SRA,(PSA)+1→SRS_{SRA}, VALUE→PSA
	2	JMP	VALUE+(PSA)→PSA
	3	JSR	(SRA)+1→SRA,(PSA)+1→SRS_{SRA}, VALUE +(PSA)→PSA
	4	JMPT	(TMA)→PSA
	5	JSRT	(SRA)+1→SRA,(PSA)+1→SRS_{SRA}, (TMA)→PSA
	6	JMPP	(SWR)→PNLBS→PSA
	7	JSRP	(SRA)+1→SRA,(PSA)+1→SRS_{SRA}, (SWR)→PNLBS→PSA

VALUE=Bits 48-63 of this instruction (CB48-CB63)

Field	Octal Code	Mnemonic	Effect
PSEVEN	Ø	RPSØA	(PS$^{QØ}_{VALUE}$)→PNLBS→ LITES
	1	RPS2A	(PS$^{Q2}_{VALUE}$)→PNLBS→ LITES
	2	RPSØ	(PS$^{QØ}_{VALUE+PSA}$)→PNLBS→ LITES
	3	RPS2	(PS$^{Q2}_{VALUE+PSA}$)→PNLBS→ LITES
	4	RPSØT	(PS$^{QØ}_{TMA}$)→PNLBS→ LITES
	5	RPS2T	(PS$^{Q2}_{TMA}$)→PNLBS→ LITES
	6	-	-
	7	-	-
	10	WPSØA	(SWR) →PNLBS→PS$^{QØ}_{VALUE}$
	11	WPS2A	(SWR) →PNLBS→PS$^{Q2}_{VALUE}$
	12	WPSØ	(SWR) →PNLBS→PS$^{QØ}_{VALUE+PSA}$
	13	WPS2	(SWR) →PNLBS→PS$^{Q2}_{VALUE+PSA}$
	14	WPSØT	(SWR) →PNLBS→PS$^{QØ}_{TMA}$
	15	WPS2T	(SWR) →PNLBS→PS$^{Q2}_{TMA}$
	16	-	-
	17	-	-

This field requires 2 cycles to execute

VALUE = Bits 48-63 of this instruction (CB48-CB63)
QØ = Quarter zero of Program Source Word (PSØØ-PS15)
Q2 = Quarter two of Program Source Word (PS31-PS47)

Field	Octal Code	Mnemonic	Effect
PSODD	Ø	RPS1A	(PS^{Q1}_{VALUE})→PNLBS→ LITES
	1	RPS3A	(PS^{Q3}_{VALUE})→PNLBS→ LITES
	2	RPS1	$(PS^{Q1}_{VALUE+PSA})$→PNLBS→ LITES
	3	RPS3	$(PS^{Q3}_{VALUE+PSA})$→PNLBS→ LITES
	4	RPS1T	(PS^{Q1}_{TMA})→PNLBS→ LITES
	5	RPS3T	(PS^{Q3}_{TMA})→PNLBS→ LITES
	6	-	-
	7	-	-
	10	WPS1A	(SWR) →PNLBS→PS^{Q1}_{VALUE}
	11	WPS3A	(SWR) →PNLBS→PS^{Q3}_{VALUE}
	12	WPS1	(SWR) →PNLBS→$PS^{Q1}_{VALUE+PSA}$
	13	WPS3	(SWR) →PNLBS→$PS^{Q3}_{VALUE+PSA}$
	14	WPS1T	(SWR) →PNLBS→PS^{Q1}_{TMA}
	15	WPS3T	(SWR) →PNLBS→PS^{Q3}_{TMA}
	16	-	-
	17	-	-

This field requires 2 cycles to execute.

VALUE=Bits 48-63 of this instruction (CB48-CB63)
Q1=Quarter one of Program Source Word (PS16-PS31)
Q3=Quarter three of Program Source Word (PS48-PS63)

Field	Octal Code	Mnemonic	Effect
PS	Ø	RPSLA	(PS$^{LH}_{VALUE}$)→DPBS
	1	RPSFA	(PS$^{FP}_{VALUE}$)→DPBS
	2	RPSL	(PS$^{LH}_{VALUE+PSA}$)→DPBS
	3	RPSF	(PS$^{FP}_{VALUE+PSA}$)→DPBS
	4	RPSLT	(PS$^{LH}_{TMA}$)→DPBS
	5	RPSFT	(PS$^{FP}_{TMA}$)→DPBS
	6	RPSLP	(PS$^{LH}_{PNLBS}$)→DPBS
	7	RPSFP	(PS$^{FP}_{PNLBS}$)→DPBS
	10	LPSLA	DPBS→PS$^{LH}_{VALUE}$
	11	LPSRA	DPBS→PS$^{RH}_{VALUE}$
	12	LPSL	DPBS→PS$^{LH}_{VALUE+PSA}$
	13	LPSR	DPBS→PS$^{RH}_{VALUE+BA}$
	14	LPSLT	DPBS→PS$^{LH}_{TMA}$
	15	LPSRT	DPBS→PS$^{RH}_{TMA}$
	16	LPSLP	DPBS→PS$^{LH}_{PNLBS}$
	17	LPSRP	DPBS→PS$^{RH}_{PNLBS}$

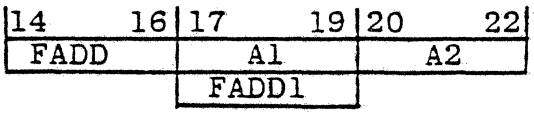
This field requires 2 cycles to execute.

VALUE=Bits 48-63 of this instruction (CB48-CB63)
LH=Left half of Program Source Word (Bits 00-31)
RH=Right half of Program Source Word (Bits 32-63)
FP=Program Source bits 26-63, used for floating-point literals

Field	Octal Code	Mnemonic	Effect
SETEXIT	Ø	-	-
	1	SETEXA	VALUE→SRS_{SRA}
	2	-	-
	3	SETEX	VALUE+(PSA)→SRS_{SRA}
	4	-	-
	5	SETEXT	TMA→SRS_{SRA}
	6	-	-
	7	SETEXP	PSA+1 →SRS_{SRA}

Sets the current subroutine return address as indicated above.
SRA does not change.
VALUE=Bits 48-63 of this instruction.

FLOATING ADDER GROUP



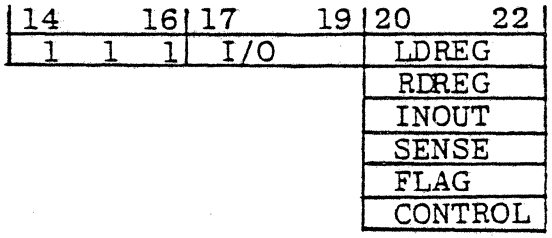
Field	Octal Code	Mnemonic	Effect
FADD	Ø	-	See FADD1 field
	1	FSUBR	Subtract: (A2) - (A1)
	2	FSUB	Subtract: (A1) - (A2)
	3	FADD	Add: (A1) + (A2)
	4	FEQV	Logical Equivalence: (A1) $\overline{\text{XOR}}$ (A2)
	5	FAND	Logical and: (A1) AND (A2)
	6	FOR	Logical or: (A1) OR (A2)
	7	-	See I/O Group
A1	Ø	NC	(A1)→A1
	1	FM	FM→A1
	2	DPX(1DX)	$(DPX_{DPA+1DX})$→A1 Where XR=1DX+4
	3	DPY(1DX)	$(DPY_{DPA+1DX})$→A1 Where YR=1DX+4
	4	TM	(TM)→A1
	5	ZERO	Ø.Ø → A1
	6	-	-
	7	-	-

Note: All floating adder op-codes:

1. Align exponents
2. Perform the specified arithmetic, logical, or shift operation
3. Normalize
4. Convergently round

Field	Octal Code	Mnemonic	Effect
A2	Ø	NC	(A2)→A2
	1	FA	FA→A2
	2	DPX (1DX)	$(DPX_{DPA+1DX})\rightarrow A2$, Where XR=1DX+4
	3	DPY (1DX)	$(DPY_{DPA+1DX})\rightarrow A2$, Where YR=1DX+4
	4	MD	(MD)→A2
	5	ZERO	Ø.Ø→A2
	6	MDPX (1DX)	$SPFN+512\rightarrow A2^{E}$, $(DPX^{M}_{DPA+1DX})\rightarrow A2^{M}$
	7	EDPX (1DX)	$(DPX^{E}_{DPA+1DX})\rightarrow A2^{E}$, $SPFN\rightarrow A2^{M}$(ØØ-Ø1), $Ø\rightarrow A2^{M}$(Ø2-27)
FADD1	Ø	-	No-Op
	1	FIX	Convert (A2) to an integer
	2	FIXT	Convert (A2) to an integer (result truncated)
	3	FSCLT	Shift (A2) right and increment $A2^{E}$ until $A2^{E}$ =(SPFN+511) (result truncated).
	4	FSM2C	Convert (A2), from signed Magnitude to 2's complement.
	5	F2CSM	Convert (A2) from 2's complement to signed magnitude.
	6	FSCALE	Shift (A2) right and increment $A2^{E}$ until $A2^{E}$=SPFN+511.
	7	FABS	Take the absolute value of (A2).

I/O GROUP



Field	Octal Code	Mnemonic	Effect
I/O	Ø	-	See LDREG field
	1	-	See RDREG field
	2	SPMDAV	Spin until MD available
	3	REXIT	SRS (SRA) → PNLBS (See Note)
	4	-	See INOUT field
	5	-	See SENSE field
	6	-	See FLAG field
	7	-	See CONTROL field
LDREG	Ø	-	No-Op
	1	LDSPD	DPBS→SPD
	2	LDMA	DPBS→MA
	3	LDTMA	DPBS→TMA
	4	LDDPA	DPBS→DPA
	5	LDSP	SP_{SPD}→SPFN, DPBS→SP_{SPD}
	6	LDAPS	DPBS→APSTATUS
	7	LDDA	DPBS→DA

Note: This is generally used with a LDSPNL to load SRS into S-Pad register. It doesn't affect SRA.

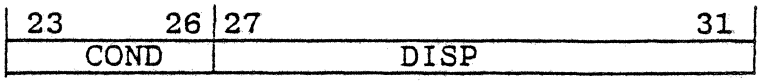
Field	Octal Code	Mnemonic	Effect
RDREG	Ø	RPSA	(PSA)→PNLBS
	1	RSPD	(SPD)→PNLBS
	2	RMA	(MA)→PNLBS
	3	RTMA	(TMA)→PNLBS
	4	RDPA	(DPA)→PNLBS
	5	RSPFN	SPFN→PNLBS
	6	RAPS	(APSTATUS)→PNLBS
	7	RDA	(DA)→PNLBS
INOUT	Ø	OUT	DPBS→$IODEVICE_{DA}$
	1	SPNOUT	SPIN if $IODRDY_{DA}$=Ø DPBS→$IODEVICE_{DA}$
	2	OUTDA	DPBS→$IODEVICE_{DA}$, SPFN→DA
	3	SPOTDA	SPIN if $IODRDY_{DA}$=Ø, SPFN→DA DPBS→$IODEVICE_{DA}$
	4	IN	($IODEVICE_{DA}$)→INBS
	5	SPININ	SPIN if $IODRDY_{DA}$=Ø ($IODEVICE_{DA}$)→INBS
	6	INDA	($IODEVICE_{DA}$)→INBS, SPFN→DA
	7	SPINDA	SPIN if $IODRDY_{DA}$=Ø, SPFN→DA ($IODEVICE_{DA}$)→INBS

Field	Octal Code	Mnemonic	Effect
SENSE	Ø	SNSA	A_{DA}→IODRDY Flag
	1	SPINA	A_{DA}→IODRDY, SPIN if IODRDY=Ø
	2	SNSADA	A_{DA}→IODRDY, SPFN→DA
	3	SPNADA	A_{DA}→IODRDY, SPIN if IODRDY=Ø, SPFN→DA
	4	SNSB	B_{DA}→IODRDY Flag
	5	SPINB	B_{DA}→IODRDY, SPIN if IODRDY=Ø
	6	SNSBDA	B_{DA}→IODRDY, SPFN→DA
	7	SPNBDA	B_{DA}→IODRDY, SPIN if IODRDY=Ø, SPFN→DA
A and B are I/O device dependent conditions, either 1 or 0			
FLAG	Ø	SFLØ	1→$FLAG_Ø$
	1	SFL1	1→$FLAG_1$
	2	SFL2	1→$FLAG_2$
	3	SFL3	1→$FLAG_3$
	4	CFLØ	Ø→$FLAG_Ø$
	5	CFL1	Ø→$FLAG_1$
	6	CFL2	Ø→$FLAG_2$
	7	CFL3	Ø→$FLAG_3$

Field	Octal Code	Mnemonic	Effect
CONTROL	0	HALT	Halt
	1	IORST	I/O reset (See note)
	2	INTEN	Interrupt enable - generates CTL05 Interrupt to Host
	3	INTA	Interrupt acknowledge. Device Address of interrupting device put into DPBS.
	4	REFR	Memory refresh sync
	5	WRTEX	Restricts DPX, DPY & MI to Write exponent only
	6	WRTMAN	Restricts DPX, DPY & MI to Write Mantissa Only (Bits 0-27)
	7	Not Used	---

Note: This also clears flags.

BRANCH GROUP



Field	Octal Code	Mnemonic	Effect
COND	Ø	-	No-Op
	1	#	Inhibit load of SPFN→SP_{SPD}
	2	BR	Branch always
	3	BINTRQ	Branch if INTRQ (Interrupt Request) flag=1
	4	BION	Branch if $IODRDY_{DA}$ flag=1
	5	BIOZ	Branch if $IODRDY_{DA}$ flag=0
	6	BFPE	Branch on floating-point arithmetic error (overflow, underflow, or divide by zero)
	7	RETURN	(SRS_{SRA})→PSA, (SRA)-1→SRA (Subroutine return jump.

NOTE: "RETURNS" may not be made in two successive instructions.

	10	BFEQ	Branch if FA-Ø.Ø
	11	BFNE	Branch if FA≠Ø.Ø
	12	BFGE	Branch if FA≥Ø.Ø
	13	BFGT	Branch if FA>Ø.Ø
	14	BEQ	Branch if SPFN=Ø
	15	BNE	Branch if SPFN≠Ø
	16	BGE	Branch if SPFN≥Ø
	17	BGT	Branch if SPFN>Ø

Note: FA and SPFN are tested as to their state for the previous instruction.

DISP	Ø to 37	If branch condition is true,(PSA) +DISP-20→PSA

Thus the effective Branch Range is -20 to +17 relative to the current instruction.

DATA PAD GROUP

32 33	34 35	36 38	39 41	42 44	45 47	48 50
DPX	DPY	DPBS	XR	YR	XW	YW

Field	Octal Code	Mnemonic	Effect
DPX	Ø	-	No-Op
	1	DPX(1DX)<DB	DPBS→*$DPX_{DPA+1DX}$, Where XW=1DX+4
	2	DPX(1DX)<FA	FA→*$DPX_{DPA+1DX}$, Where XW=1DX+4
	3	DPX(1DX)<FM	FM→*$DPX_{DPX+1DX}$, Where XW=1DX+4
DPY	Ø	-	No-Op
	1	DPY(1DX)<DB	DPBS→*$DPY_{DPA+1DX}$ Where YW=1DX+4
	2	DPY(1DX)<FA	FA→*$DPY_{DPA+1DX}$ Where YW=1DX+4
	3	DPY(1DX)<FM	FM→*$DPY_{DPA+1DX}$ Where YW=1DX+4

*All bits written unless WRTEXP, WRTHMAN or WRTLMAN set. See SOP1 and HOSTPNL field.

DPBS	Ø	DB=ZERO	Ø.Ø→DPBS
	1	DB=INBS	INBS→DPBS
	2	DB=VALUE	VALUE→$DPBS^{E}$, VALUE→$DPBS^{ML}$, sign extended into $DPBS_{MH}$
	3	DB=DPX(1DX)	$(DPX_{DPA + 1DX})$→DPBS, Where XR=1DX+4
	4	DB=DPY(1DX)	$(DPY_{DPA + 1DX})$→DPBS, Where YR=1DX+4
	5	DB=MD	(MD) DPBS
	6	DB=SPFN	SPFN + 512→$DPBS^{E}$, SPFN→$DPBS^{ML}$, sign extended into $DPBS_{MH}$
	7	DB=TM	(TM)→DPBS

DPBS forced to Ø if HOSTPNL field=10 to 13
ML=Mantissa Low (Mantissa Bits 12-27)
MH=Mantissa High (Mantissa Bits 00-11)
E=Exponent
VALUE is a 16-bit 2's complement number, contained in bits 48-63 of the instruction word.

Field	Octal Code	Mnemonic	Effect
XR	Ø to 7		DPX Read EFA is (DPA)+XR-4
YR	Ø to 7		DPY Read EFA is (DPA)+YR-4
XW	Ø to 7		DPX Write EFA is (DPA)+XW-4
YW	Ø to 7		DPY Write EFA is (DPA)+YW-4, YW=XW if VALUE is used in another field

FLOATING MULTIPLIER GROUP

51	52 53	54 55
FM	M1	M2

Field	Octal Code	Mnemonic	Effect
FM	Ø	-	No-Op
	1	FMUL	Multiply: (M1)*(M2)
M1	Ø	FM	FM→M1
	1	DPX (1DX)	$(DPX_{DPA+ 1DX})\rightarrow M1$, Where XR=1DX+4
	2	DPY (1DX)	$(DPY_{DPA+ 1DX})\rightarrow M1$, Where YR=1DX+4
	3	TM	(TM)→M1
M2	Ø	FA	FA→M2
	1	DPX (1DX)	$(DPX_{DPA+ 1DX})\rightarrow M2$, Where XR=1DX+4
	2	DPY (1DX)	$(DPY_{DPA+ 1DX})\rightarrow M2$, Where YR=1DX+4
	3	MD	(MD)→M2

Note: These fields are not in effect if VALUE is used in another field.

Arguments that are unnormalized by more than one position will produce incorrect results.

MEMORY GROUP

56 57	58 59	60 61	62 63
MI	MA	DPA	TMA

Field	Octal Code	Mnemonic	Effect
MI	Ø	-	No-Op
	1	MI<FA	FA→MI, write MI into Data Memory**
	2	MI<FM	FM→MI, write MI into Data Memory**
	3	MI<DB	DPBS→MI, write MI into Data Memory**

**All bits written unless WRTEXP, WRTHMAN or WRTLMAN is set. See SOP1 and HOSTPNL fields.

MA	Ø	-	No-Op
	1	INCMA	(MA)+1→MA, intitate a Data Memory cycle
	2	DECMA	(MA)-1→MA, initiate a Data Memory cycle
	3	SETMA	*SPFN→MA, initiate a Data Memory cycle

*DPBS is used in place of SPFN if LDREG field is used.

DPA	Ø	-	No-Op
	1	INCDPA	(DPA)+1→DPA
	2	DECDPA	(DPA)-1→DPA
	3	SETDPA	*SPFN→DPA

*DPBS is used in place of SPFN if LDREG field is used.

Note: These fields are not in effect if a value is used by another field. Changes made in MA, TMA, or DPA do not affect the values of these registers used by other fields during the current instruction.

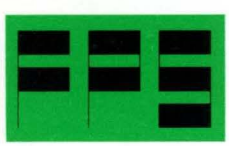
Field	Octal Code	Mnemonic	Effect
TMA	Ø	-	No-Op
	1	INCTMA	(TMA)+1→TMA, initiate a read from Table Memory
	2	DECTMA	(TMA)+1→TMA, initiate a read from Table Memory
	3	SETTMA	*SPFN→TMA, initiate a read from Table Memory

*DPBS is used in place of SPFN if LDREG field is used.

Note: These fields are not in effect if a VALUE is used by another field. Changes made in MA, TMA, or DPA do not affect the values of these registers used by other fields during the current instruction.

0 to 63	NOP	Assembler recognizes this mnemonic and will insert an all zeros instruction which is a NOP.

INDEX



FLOATING POINT
SYSTEMS, INC.

CALL TOLL FREE 800-547-1445
P.O. Box 23489, Portland, OR 97223
(503) 641-3151, TLX: 360470 FLOATPOINT PTL