To the Graduate Council:

I am submitting herewith a thesis written by Susan Vickrey entitled "Implementation of the Am9511 Arithmetic Processing Unit Into a Typical Microprocessor System." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Ronald A. Belz, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

Thesis
60
.V524
cop. 2

IMPLEMENTATION OF THE Am9511 ARITHMETIC PROCESSING UNIT INTO A TYPICAL MICROPROCESSOR SYSTEM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Susan Vickrey

June 1980

3041892

ACKNOWLEDGMENTS

The author wishes to express her sincere appreciation to her major advisor, Dr. Ronald A. Belz, for his constant support and contributions. The author also wishes to thank Dr. Terry Feagin and Dr. F. W. Donaldson for their support.

The author also wishes to express much love and thanks to her parents for their unfailing support and encouragement.

ABSTRACT

The Advanced Micro Devices' Am9511 Arithmetic Processing Unit (APU) and its implementation into a typical microprocessor system, in this case, the Motorola M6800 System, are described. A description of the Am9511, covering its internal configuration, the commands it can execute, the data formats required by these commands, and how to address the APU within the system are presented. The M6800 Central Processing Unit (CPU) is covered, along with a description of the 32K RAM memory board and the logic which connects the Am9511 to the system. The routines written for program interface with the Am9511 are described. These include utility routines to perform all stack load and unload, stack manipulation, and command execution operations individually, and the complete-operation routines to stack the necessary operand(s), execute the desired command, retrieve the result, and perform limited error analysis and recovery, all in one routine. A problem was encountered when a log of zero was attempted. The required fix-up procedure is described. Detailed information on the APU commands and the access routines are also included, with listings of each command, its hexadecimal code, its function, input requirements, possible errors, and effect on the APU stack; flow charts

of each routine written to access the APU are included; and a listing of each routine, written in the M6800 instruction set, is presented.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

CHAPTER I

INTRODUCTION

Over the incredibly brief era of the microprocessor, an advance in the reduction of the size and cost of microprocessor chips and associated memory has made the use of a single computer dedicated to one function or one problem economical. However, a programming bottleneck has developed in the use of any type of calculation. The need for multi-byte words to express very large and very small numerical values complicates simple arithmetic, requiring complex routines to carry out the operations. Derived functions have always been a problem, requiring lengthy and complicated software approximations for solutions.

The Advanced Micro Devices' Am9511 Arithmetic Processing Unit (APU) is a great advance in this area. The arithmetic functions and Tchebyshev approximations of the derived functions are programmed into the chip itself, so that they are now part of the hardware instead of the software, thereby speeding up each operation's execution time.

A microprocessor system which includes the Am9511 APU needs a set of routines for using the APU available to its users. A typical system, based upon the Motorola M6800 Central Processing Unit (CPU) will be described.

Chapter II describes the APU internal architecture, the addressing of the APU, and the kind of data handled by

the APU. Chapter III presents the microprocessor system in which the APU is incorporated. Chapter IV describes the two sets of routines written for the system and the basic ASCII conversion routines. Chapter V will summarize the investigation.

CHAPTER II

Am9511 ARITHMETIC PROCESSING UNIT DESCRIPTION

The Am9511 Arithmetic Processing Unit (APU) provides high performance fixed and floating point basic arithmetic operations and transcendental derived functions (trigonometric and inverse trigonometric, square root, logarithm, and exponentiation), as well as floating point-to-fixed and fixed-to-floating point conversions and stack control operations.

A. Internal Configuration

The internal configuration of the Am9511 is illustrated in Fig. 1. The chip is in a standard 24-pin package.

Data Stack and Command and Status Registers

Three parts of the APU internal structure are accessible to the programmer. These are the Data Stack, the Command Register, and the Status Register. These three entities are accessed by the microprocessor as locations in system memory.

In the microcomputer, the Data Stack is located at system address $D01C_{16}$. The stack may be considered to be an 8 by 16-bit stack for single-precision operands, or a 4 by 32-bit stack for double-precision and floating point

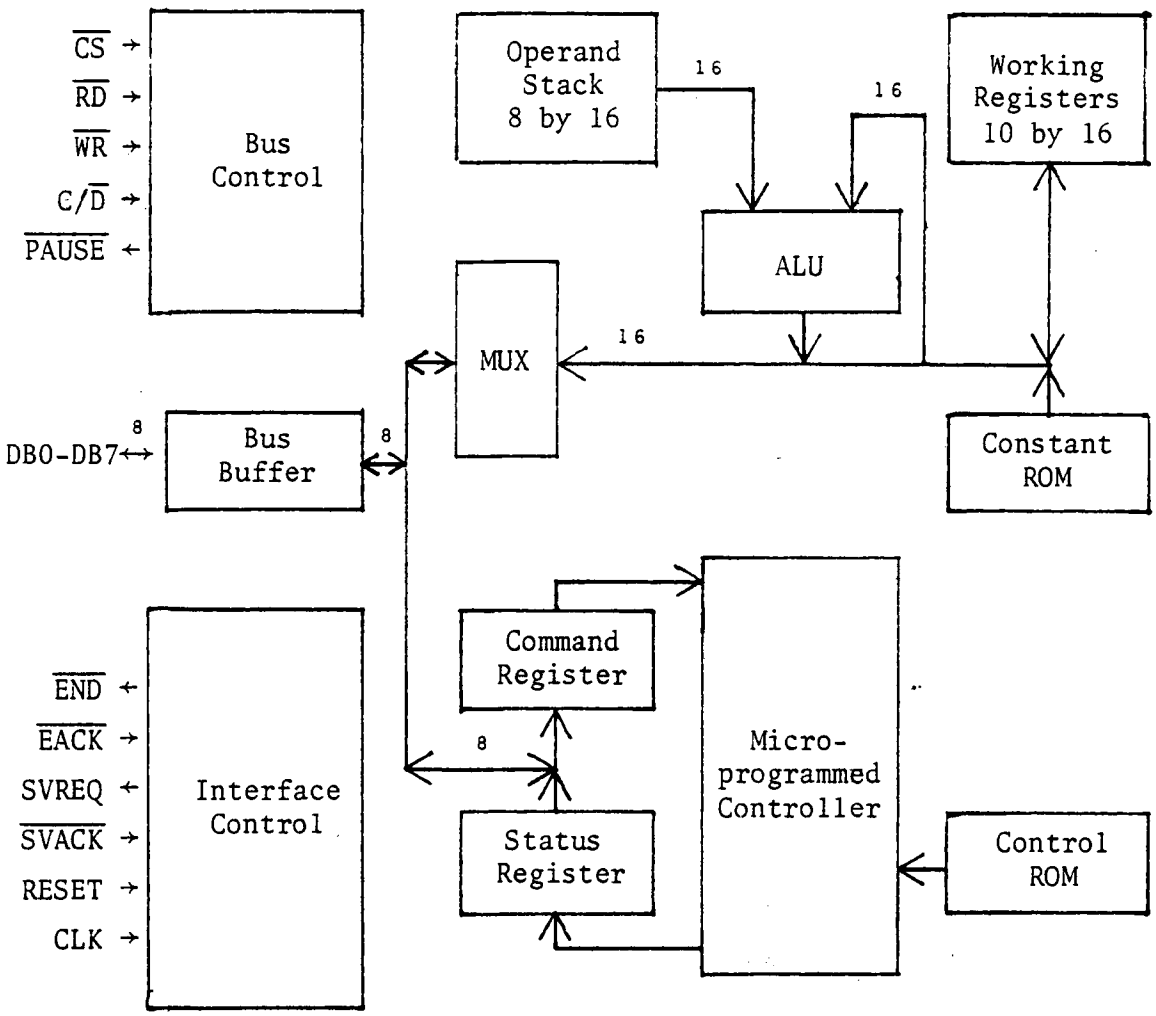


Figure 1. Am9511 internal configuration.

operands, as shown in Fig. 2. A memory read operation at $D01C_{16}$ will read the top byte of the Data Stack, causing those eight bits to be physically removed from the stack. A memory write operation to $D01C_{16}$ will write a byte onto the top of the stack. For convenience, the top operand on the stack is referred to as TOS (Top of Stack) and the operand below it is NOS (Next on Stack).

The Command Register is an 8-bit register accessed by the system at memory address $D01E_{16}$. When a command word is written into $D01E_{16}$, execution of that particular operation begins. The command byte format is shown in Fig. 3. A listing of the various command codes and descriptions is presented in Table 1. A full description of each command is contained in Appendix A.

The Status Register is also an 8-bit register accessed by the system at memory address $D01E_{16}$. A memory read at this address will access the status byte. The Status Register may be read when the APU is active or idle, without affecting any operation in progress, the contents of the Data Stack, or changing the value of the Status Register. When the busy bit (bit 7) is set, the values of the other bits are meaningless. When the busy bit is clear, the result of the operation may be decoded, according to which bits are set or cleared, as shown in Fig. 4.

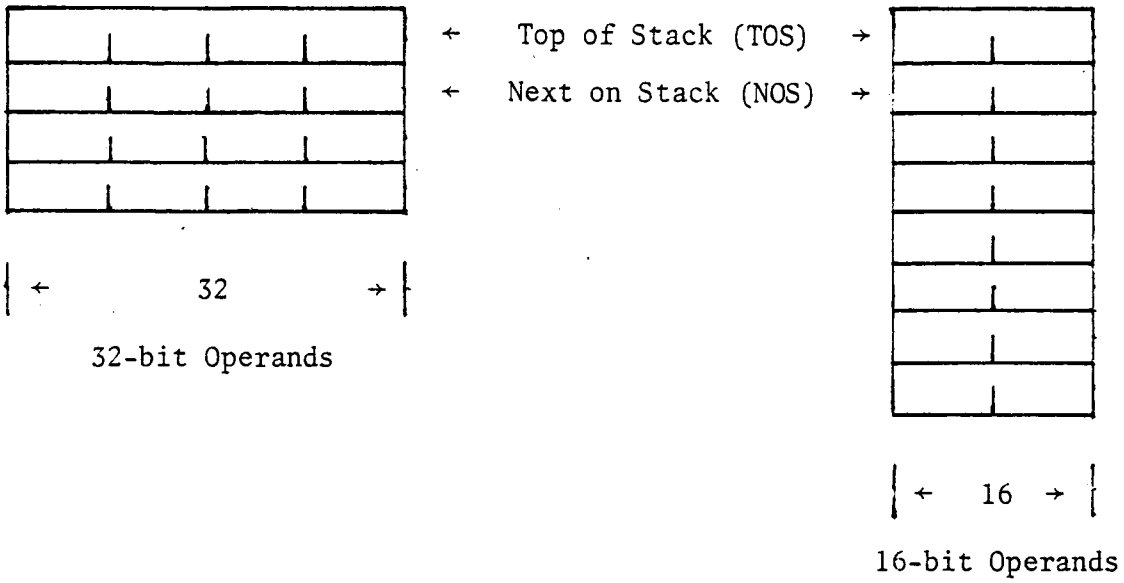


Figure 2. Am9511 data stack.

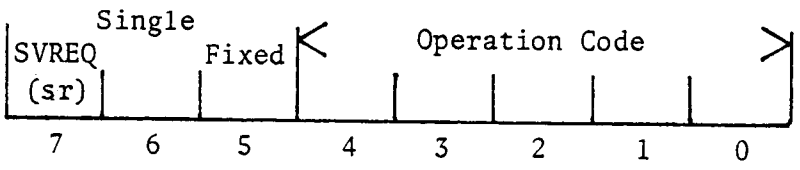


Figure 3. Am9511 command byte.

Table 1. Am9511 Command Description

Command Mnemonic	Hex Code (sr = 0)	Hex Code (sr = 1)	Summary Description
		16-bit Fixed Point Operations	
SADD	6C	EC	Add TOS to NOS
SSUB	6D	ED	Subtract TOS from NOS
SMUL	6E	EE	Multiply NOS by TOS; lower result
SMUU	76	F6	Multiply NOS by TOS; upper result
SDIV	6F	EF	Divide NOS by TOS
		32-bit Fixed Point Operations	
DADD	2C	AC	Add TOS to NOS
DSUB	2D	AD	Subtract TOS from NOS
DMUL	2E	AE	Multiply NOS by TOS; lower result
DMUU	36	B6	Multiply NOS by TOS; upper result
DDIV	2F	AF	Divide NOS by TOS
		32-bit Floating Point Primary Operations	
FADD	10	90	Add TOS to NOS
FSUB	11	91	Subtract TOS from NOS
FMUL	12	92	Multiply NOS by TOS
FDIV	13	93	Divide NOS by TOS

Table 1 (continued)

Command Mnemonic	Hex Code (sr = 0)	Hex Code (sr = 1)	Summary Description
		32-bit Floating Point Derived Operations	
SQRT	01	81	Square root of TOS
SIN	02	82	Sine of TOS
COS	03	83	Cosine of TOS
TAN	04	84	Tangent of TOS
ASIN	05	85	Inverse sine of TOS
ACOS	06	86	Inverse cosine of TOS
ATAN	07	87	Inverse tangent of TOS
LOG	08	88	Common logarithm of TOS
LN	09	89	Natural logarithm of TOS
EXP	0A	8A	e raised to power in TOS
PWR	0B	8B	NOS raised to power in TOS
		Data and Stack Manipulation Operations	
NOP	00	80	No operation
FIXS	1F	9F	Convert TOS from floating point to fixed point format
FIXD	1E	9E	
FLTS	1D	9D	Convert TOS from fixed point to floating point format
FLTD	1C	9C	
CHSS	74	F4	Change sign of fixed point TOS
CHSD	34	B4	
CHSF	15	95	Change sign of floating point TOS
PTOS	77	F7	Push stack
PTOD	37	B7	
PTOF	17	97	
POPS	78	F8	Pop stack
POPD	38	B8	
POPF	18	98	
XCHS	79	F9	Exchange TOS and NOS
XCHD	39	B9	
XCHF	19	99	
PUPI	1A	9A	Push floating point constant π onto TOS

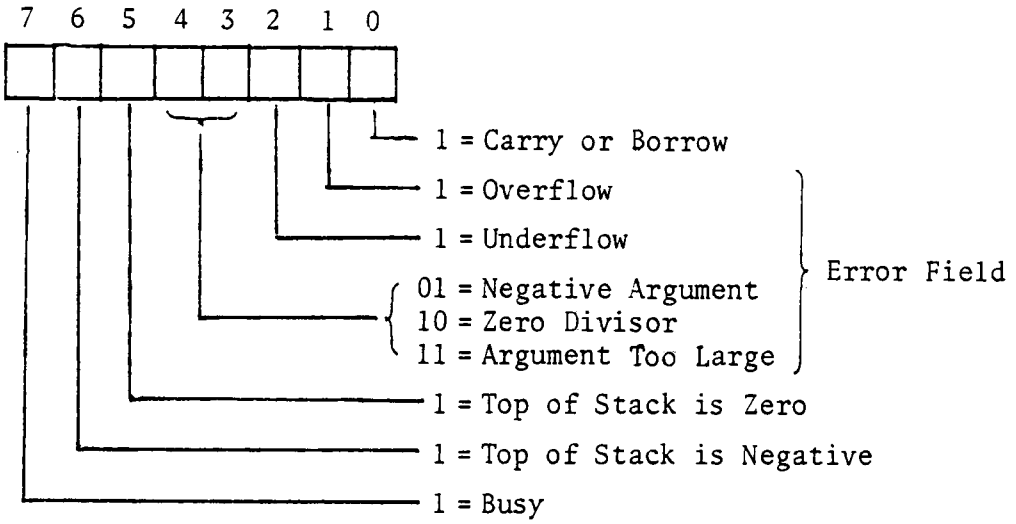


Figure 4. Am9511 status byte.

Arithmetic Processing Unit Lines

All operand, result, status, and command information is transferred over the 8-bit bidirectional data bus (DB0-DB7). The Chip Select line ($\overline{CS}$) enables the chip. The Command/Data line (C/$\overline{D}$) in conjunction with the Read ($\overline{RD}$) and Write ($\overline{WR}$) lines signal whether the data bus is connected to the Data Stack, the Command Register, or the Status Register. Table 2 describes the relationship of these three lines.

Table 2. Am9511 Register/Stack Addressing

C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	Function
0	1	0	Enter data byte onto stack
0	0	1	Read data byte from stack
1	1	0	Enter command
1	0	1	Read status

The Pause ($\overline{PAUSE}$) signal is an output which indicates the APU is busy executing an operation, transferring data to or from the Data Stack, or having the status read.

Five interface control signals allow the CPU to initialize the APU and check and control its status. RESET is an active high input signal that initializes the APU. On this signal, all operations in progress are terminated and the status register is cleared, leaving the Data Stack contents unchanged. The APU then goes into the idle state.

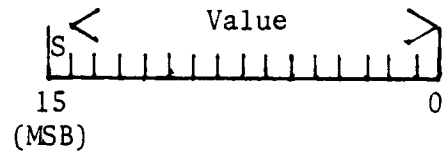
Since there is no internal power-on reset, this line must be initially brought high by the system during the initialization process. $\overline{END}$, the End of Execution signal, is an active low output indicating that the previously entered command has finished execution. $\overline{END}$ may be used as an interrupt request and is cleared by the End Acknowledge signal ($\overline{EACK}$), by RESET, or by any read or write to the APU. The Service Request signal, SVREQ, indicates that command execution is complete and that the previous command byte requested post-execution service by having a bit 7 value of 1. This signal may be cleared by the Service Acknowledge signal ($\overline{SVACK}$), by RESET, or by the completion of a subsequent command not requesting service (bit 7 clear). SVREQ may be used as an interrupt to the CPU. The Clock signal, CLK, must be supplied by an external timing source.

B. Data Format Specification

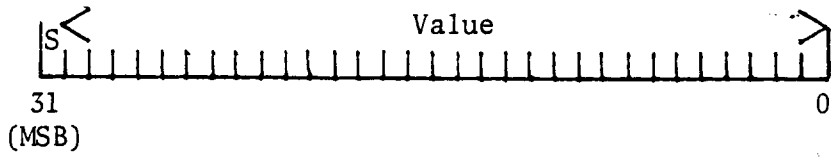
The Am9511 executes operations using 16-bit (single-precision) and 32-bit (double-precision) fixed point and 32-bit floating point binary data formats.

Fixed Point Formats

The single-precision fixed point format has a 16-bit operand in the range of -32,768 to +32,767 (Fig. 5). The sign is located in the Most Significant Bit (MSB). Negative values are in two's complement format and have a one in the



a. 16-bit Fixed Point Format



b. 32-bit Fixed Point Format

Figure 5. Am9511 single- and double-precision fixed point formats.

sign bit; positive values and zero have a zero in the sign bit.

As an example,

$$72_{10} = 0000\ 0000\ 0100\ 1000_2 = 00\ 48_{16} ,$$

and

$$-72_{10} = 1111\ 1111\ 1011\ 1000_2 = FF\ B8_{16} .$$

Data are presented in the double-precision fixed point format as 32-bit values representing binary two's complement integers and having a range of -2,147,483,648 to +2,147,483,647 (Fig. 5). The format for double-precision is generally the same as for single-precision, as described above.

Floating Point Formats

The floating point format shown in Fig. 6 is represented by a 32-bit binary operand, composed of a 24-bit mantissa, a 6-bit exponent, an exponent sign, and a mantissa sign. Decimal numbers in the range of $\pm(2.7 \times 10^{-20}$ to $9.2 \times 10^{18})$ can be expressed in the form of a mantissa, multiplied by two raised to a power. The mantissa is a 24-bit fractional binary number occupying the least significant three bytes of the operand. It is always normalized to a value between 0.5 and 1 (by shifting left or right while adjusting the exponent value until the left-most bit of the mantissa is a one). The exception to this

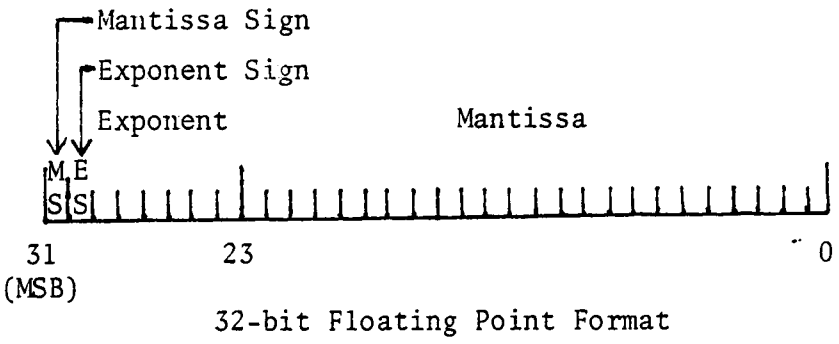


Figure 6. Am9511 floating point format.

is the number zero, which is represented by all zeros in the operand. The exponent is a 7-bit binary two's complement number, ranging in value from -64 to +63, with the most significant bit holding the value for the sign (0 = positive/1 = negative). The sign bit for the mantissa is the most significant bit of the operand, with 0 indicating a positive value, and 1, a negative value.

A decimal number is converted to the floating point representation in the following manner, using the decimal number 11.6875 as an example. The number is first converted to binary:

$$11.6875_{10} = 1011.1011_2 ,$$

working on the principal that any decimal number can be converted to a binary number of the format

$$\ldots + a \cdot 2^2 + b \cdot 2^1 + c \cdot 2^0 + d \cdot 2^{-1} + e \cdot 2^{-2} + \ldots .$$

The binary number is then converted to the base 2 exponent notation:

$$1011.1011_2 = 0.10111011 \times 2^{+4} .$$

Remembering that the mantissa is 24 bits long, this gives a mantissa value of

$$1011\ 1011\ 0000\ 0000\ 0000\ 0000_2\ (BB\ 00\ 00_{16}) .$$

The exponent value, +4, is converted to its binary equivalent in two's complement integer form:

$$+4_{10} = 0000100_2 .$$

Since the value is positive, the mantissa sign bit has a value of zero. Therefore, the 32-bit floating point equivalent of 11.6875_{10} is

$$0\ 000\ 0100\ \ 1011\ 1011\ 0000\ 0000\ 0000\ 0000_2 ,$$

which, in hexadecimal notation, is

$$04\ BB\ 00\ 00_{16} .$$

For a negative number, the same procedure is used, with the exception that the mantissa sign bit is set to one. For example,

$$-11.6875_{10} = 1\ \ 000\ 0100\ \ 1011\ 1011\ 0000\ 0000\ 0000\ 0000_2$$
$$= 84\ BB\ 00\ 00_{16} .$$

To convert a fractional number such as 0.359375_{10} to the floating point format, the number is converted to fractional binary:

$$0.35975_{10} = 0.010111_2 .$$

The result is normalized, as before:

$$0.010111_2 = 0.10111_2 \times 2^{-1} .$$

The exponent, -1, is converted to two's complement form:

$$-1_{10} = -(0000001_2) = 1111111_2 .$$

Since the mantissa is positive, the sign bit is a zero, so the 32-bit floating point format is:

$$0.359375_{10} = 0\ \ 111\ 1111\ \ 1011\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

$$= 7F\ B8\ 00\ 00_{16} .$$

CHAPTER III

CENTRAL PROCESSING UNIT DESCRIPTION

The Am9511 was incorporated into a microcomputer system based upon the Motorola 6800 Microprocessing Unit (MPU) (Fig. 7). The Central Processing Unit (CPU) board has five functional subsystems:

1. MPU, clock circuitry, and related operational control logic.
2. Tri-state address and data line buffer/drivers with timing decode logic.
3. Read Only Memory (ROM), masked with operating system and external decode logic.
4. Random Access Memory (RAM) and associated control circuitry.
5. Peripheral Interface Adapter (PIA), for onboard serial input/output (I/O) with timing control and baud rate select logic.

The microprocessor system also includes 32K of memory on a RAM board, located at addresses 0000_{16} - $7FFF_{16}$. The Am9511 is on a third board, with the Data Stack located at address $D01C_{16}$, and the Command and Status Registers located at $D01E_{16}$. All three boards are hooked up to an S100 bus.

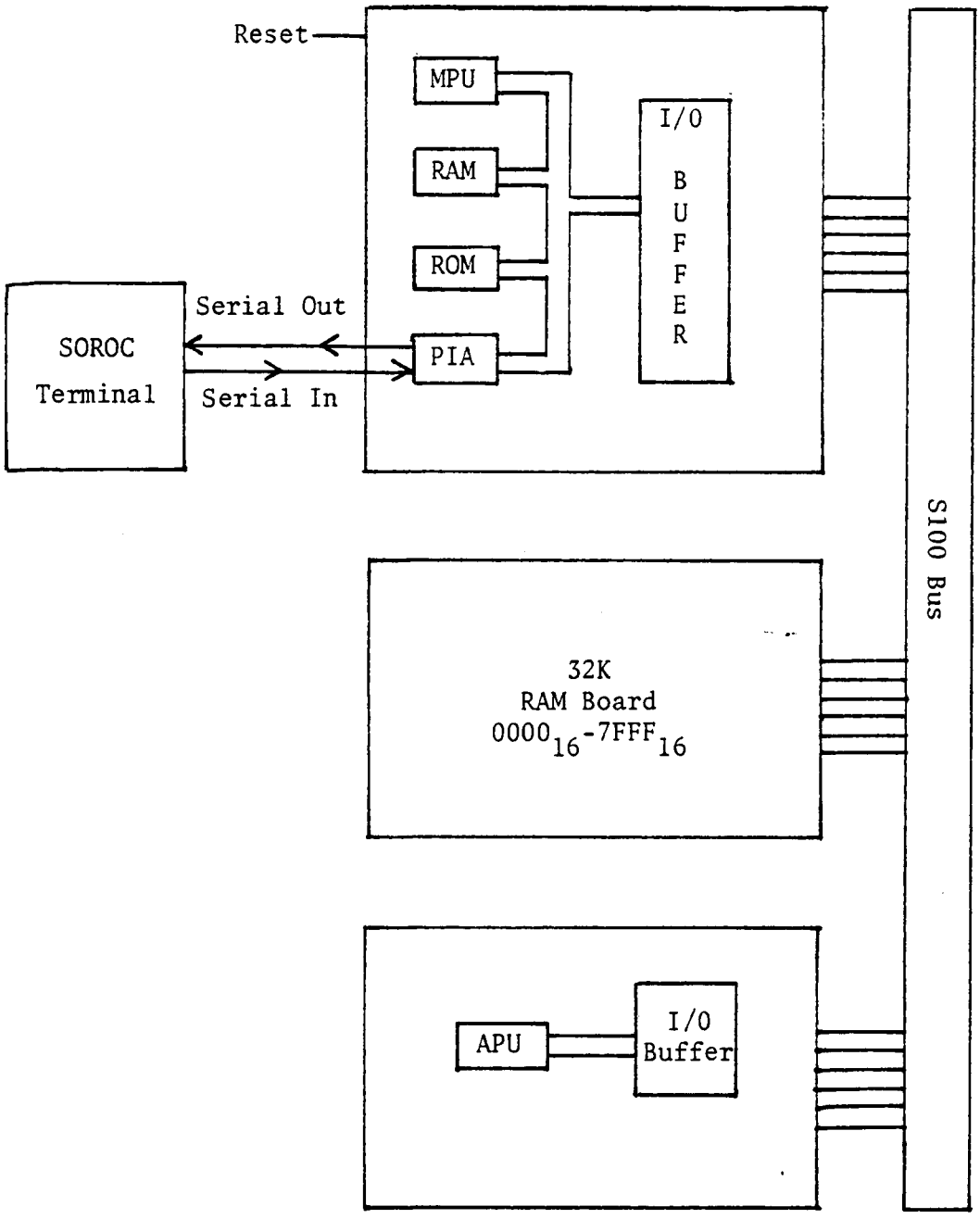


Figure 7. System configuration.

A. Central Processing Unit

The five functional subsystems are depicted in block form in Fig. 7. The system has a clock rate of 1.84 MHz. The output of the clock is also initially divided by an MC14411 baud rate generator, which generates baud rates for the serial I/O. Various control/decode logic, power-on auto restart, the clock signals for the MPU, the system buses, dynamic and slow memory interfacing, the board clock outputs, and memory refresh are provided on the S100 compatible CPU board.

The 6800 MPU, shown in Fig. 8, is an 8-bit parallel processor. It has six registers, which include two 8-bit accumulators (A and B), an Index Register, a Program Counter, a Stack Pointer, and a Condition Code Register. The instruction set includes 72 variable-length instructions and 7 addressing modes. The 16 address lines allow direct addressing of 65,536 words of memory. The MPU also has eight bidirectional data lines, over which all data and instruction bytes are transferred. The MPU requires two input clock signals (ϕ_1 and ϕ_2), but only one power supply (+5V). The eight data lines are connected to bidirectional data bus transceivers, which break up the bidirectional data bus attached to the MPU into the 16 separate input and output lines required by the S-100 bus. The control signals used in conjunction with the data bus are Read/Write (R/$\overline{W}$), which determines the direction of

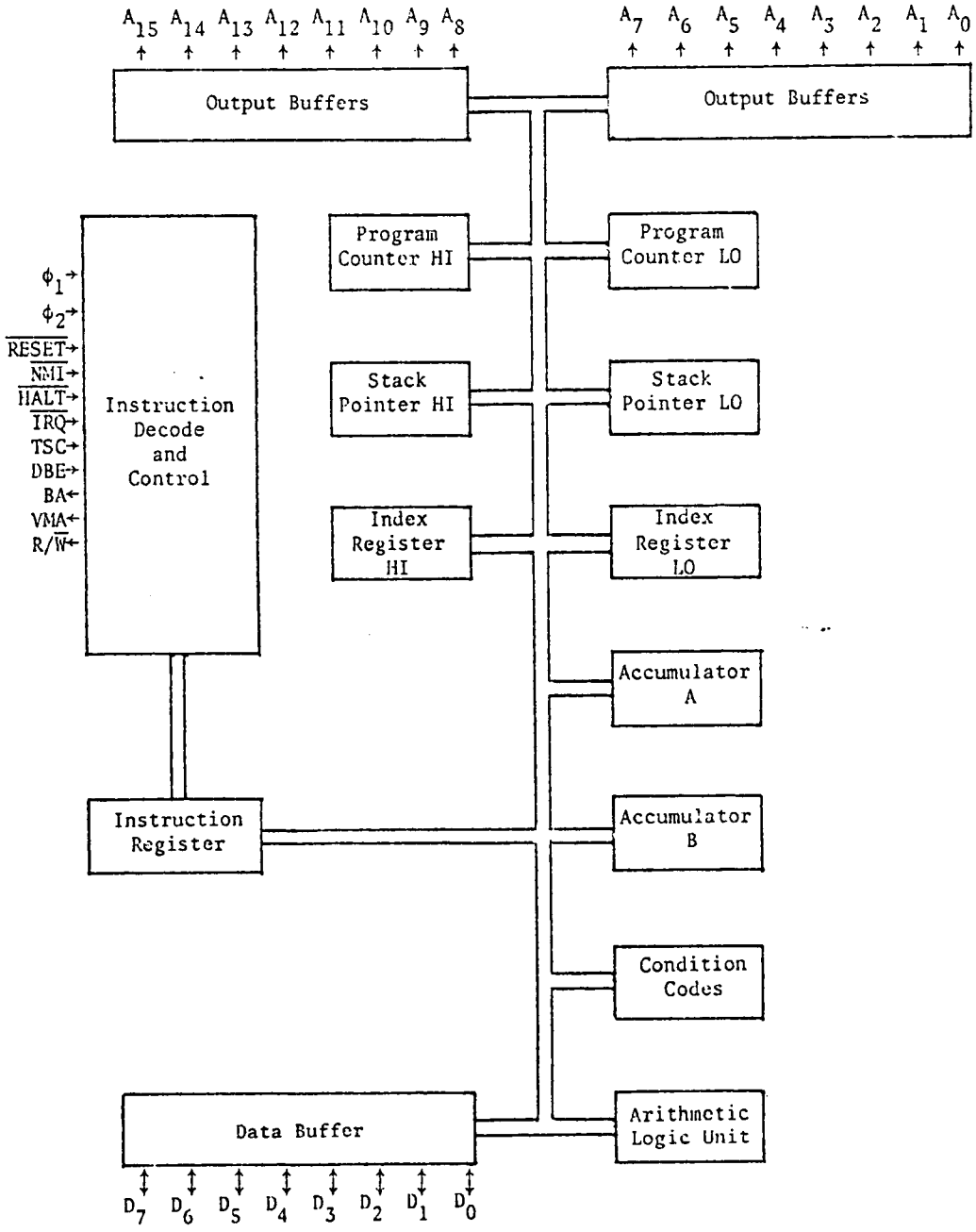
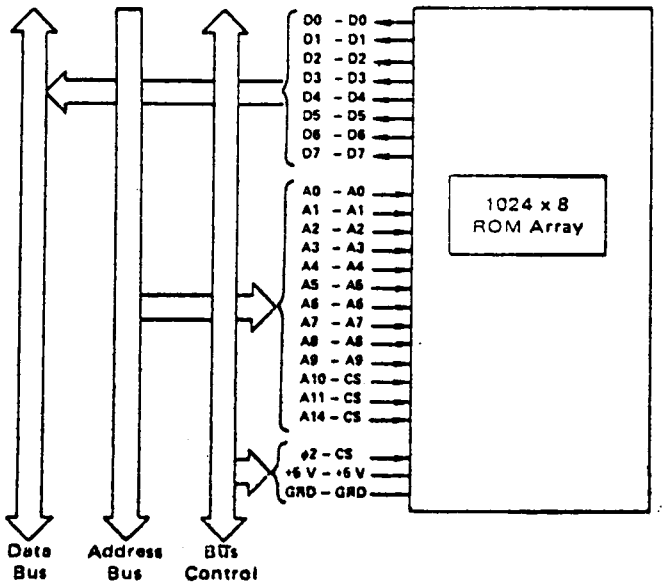


Figure 8. M6800 internal configuration.

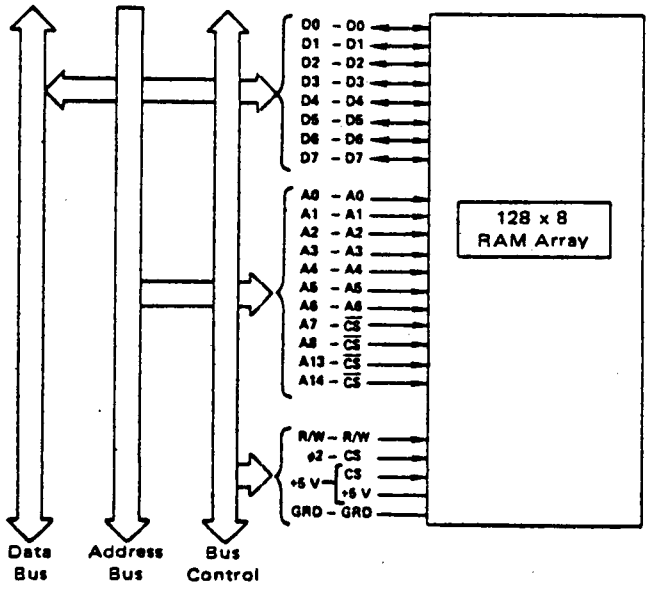
data transfer; Bus Available (BA), which determines whether the MPU or an external device is in control of the data bus; and Valid Memory Address (VMA), which signals when a valid memory address is present on the address lines.

A 6830 Read Only Memory (ROM), organized into 1,024 bytes, contains the operating system for the MPU. The ROM is located at addresses $E000_{16}$ to $E3FF_{16}$. A 6810 Random Access Memory (RAM), organized as 128 bytes, provides stack area and scratchpad facilities for the operating system located on the 6830. It is located at addresses $A000_{16}$ to $A07F_{16}$. The four chip selects on the 6830 provide sufficient partial address decoding to distinguish the ROM. The six chip selects on the 6810 provide similar decoding for the RAM (Fig. 9).

A 6820 Peripheral Interface Adapter (PIA) provides serial I/O with the peripheral device, which in this system is a video terminal. The PIA is enabled by VMA, RESET, and ϕ_2; it is addressed by system addresses 8xx4 to 8xx7. The R/$\overline{W}$ line directs the direction of data transfer. The PIA is organized into two ports (A and B). Each port handles eight separately programmable I/O lines and two control lines, and each is addressable in the system memory (Fig. 10).



a. ROM Bus Interface



b. RAM Bus Interface

Figure 9. M6800 ROM and RAM bus interfaces.

Source: Branko Soucek, Microprocessors and Microcomputers (New York: John Wiley and Sons, Inc., 1976), pp. 323-324.

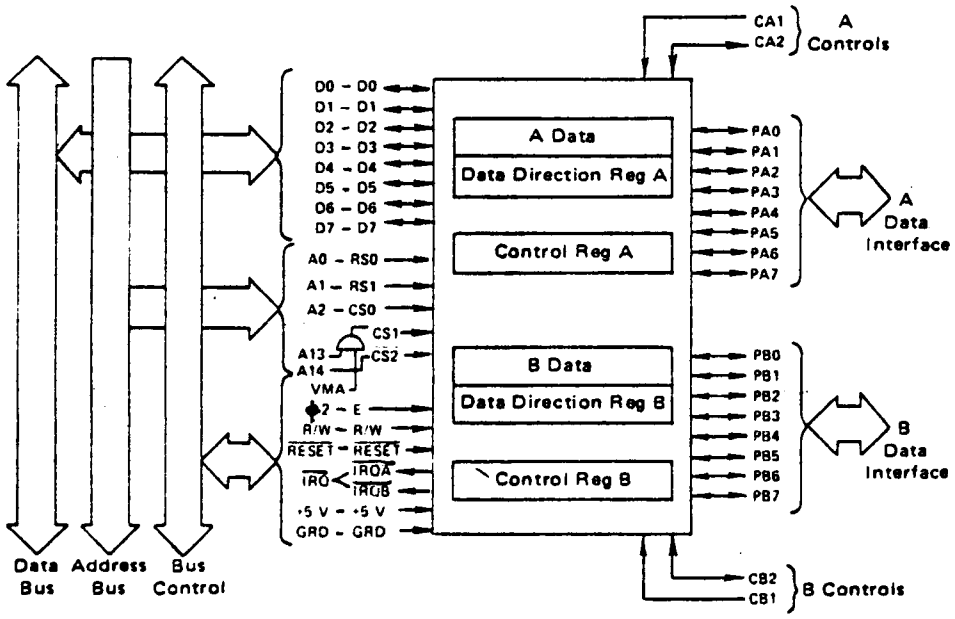


Figure 10. M6800 PIA bus interface.

Source: Branko Soucek, Microprocessors and Microcomputers (New York: John Wiley and Sons, Inc., 1976), p. 326.

B. Arithmetic Processing Unit and Memory

The Am9511 is located on an S100 board (Fig. 11) connected to the CPU at addresses $D01E_{16}$ (Command/Status Registers) and $D01C_{16}$ (Data Stack). Address lines A0 and A2 to A15 determine the Chip Select ($\overline{CS}$) line. The Command/Data (C/$\overline{D}$) line is tied to A1. The read ($\overline{RD}$) line is activated by the MPU R/$\overline{W}$ line and the Chip Select; the Write ($\overline{WR}$) line is also activated by the MPU R/$\overline{W}$ and the Chip Select. The 8-bit Data Bus (DB0-DB7) is tied to the Data In (DI0-DI7) and Data Out (DO0-DO7) lines. When $\overline{RD}$ is active, the DI lines are enabled, and when $\overline{WR}$ is active, the DO lines are enabled. The RESET line is connected to the Power-on-Clear ($\overline{POC}$) and the Clock (CLK) line is connected to the CPU signal ϕ_2.

The system also has 32K of dynamic RAM, contained on a single board connected to the S100 bus. This RAM is located at memory locations 0000_{16} - $7FFF_{16}$. The RAM is cleared when power is shut off. At this time, the system has no permanent memory storage.

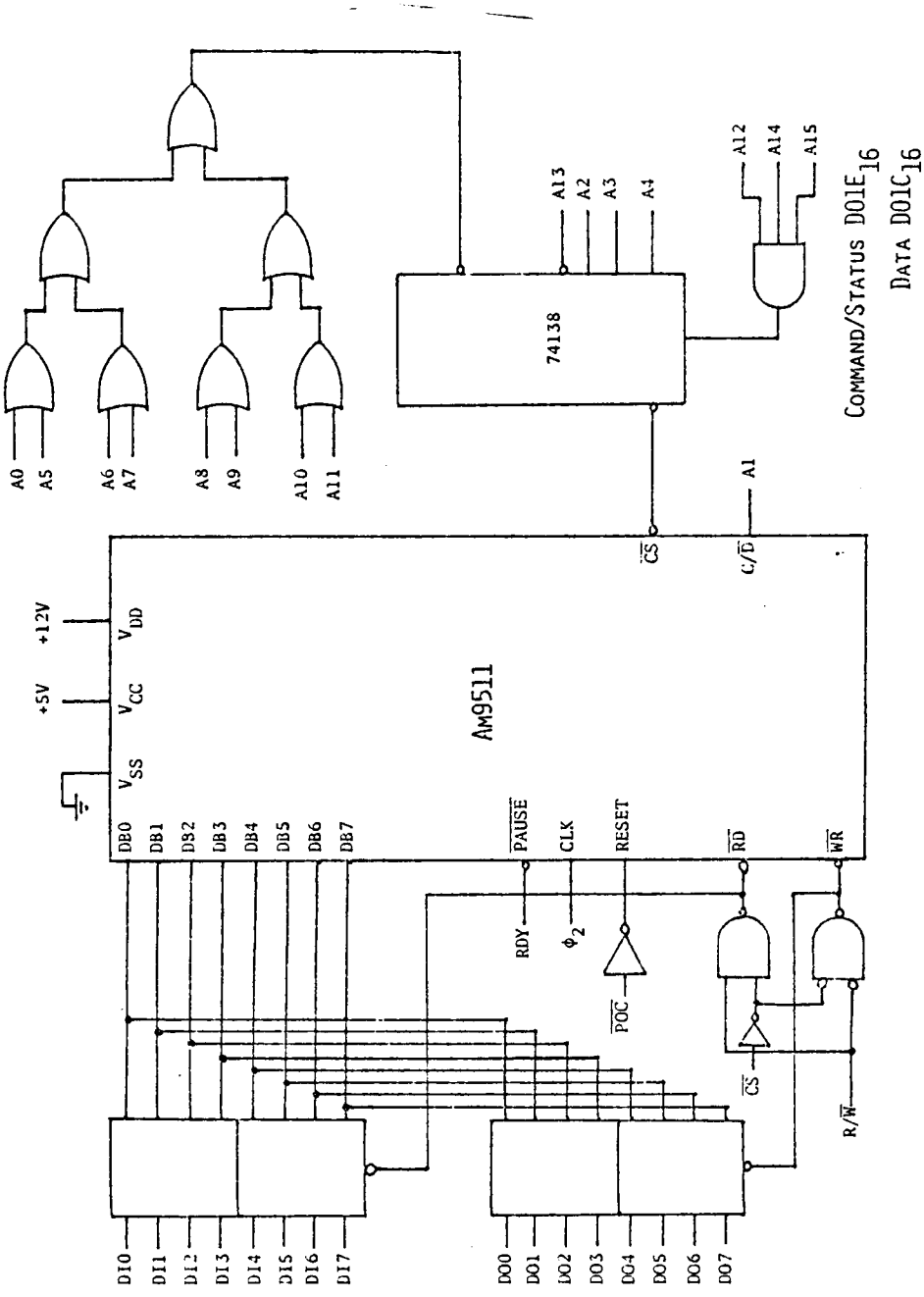


Figure 11. Am9511 logic.

CHAPTER IV

PROGRAM DESCRIPTION

Two distinct sets of programs were written for the Am9511. The programmer may choose from either or both sets in his program, according to his needs. The first is a set of utility routines with which the programmer performs all of the various operations individually. These include all chip Data Stack load and unload operations and all Data Stack manipulations, as well as all arithmetic and derived function calls. The other set of routines combines into each command the loading of the Data Stack, command execution, retrieval of the result from the Data Stack, error analysis, and a limited degree of error recovery.

In addition to the two sets of Arithmetic Processing Unit (APU) command routines, routines are also provided for ASCII-to-floating point and floating point-to-ASCII conversion.

A. Utility Routines

Stack and Unstack Routines

The routines which load data onto the APU stack are called STK2 and STK4. These routines respectively load two and four bytes of data from a memory location specified by the Index Register. The routines UNSTK2 and UNSTK4

respectively unload two and four bytes of data from the APU Data Stack, storing them in the memory locations specified by the Index Register of the MPU.

Command Routines

The command routines are named in such a manner as to reflect the function to be executed. In essence, the APU function name is preceded by the letter U (for example, for the inverse sine command ASIN, the routine UASIN would be called). After loading the necessary operand(s) into the APU Data Stack with the stack routines described above, the programmer calls the desired command routine. Each command routine loads the proper command byte into the MPU accumulator A, and then branches to the UXEQ routine, which sends the command stored in accumulator A to the APU Command Register (located at system address $D01E_{16}$). The status routine, USTAT, is then executed. USTAT reads the status byte now available at $D01E_{16}$ and checks the value of the busy bit (see Chapter II). If the busy bit contains a one, the routine branches back to read the status word again. When the busy bit is clear (0), the remainder of the status word is valid and is stored by USTAT in the program-defined memory location named ERRSTAT, which makes it available for further use.

B. Complete-Operation Routines

The complete-operation routines require the programmer to establish the addresses of his operands in predefined memory locations. The programmer then calls the specific command routine, which stacks the operand(s), executes the operation, unstacks the result, checks for error conditions, and provides error recovery where necessary.

Stack and Unstack Routines

Before calling an operation routine, the operator should store the address of operand A in the program-defined memory location OPA. If the command is a two-operand function, the address of operand B should be stored in the program-defined memory location OPB. The address of the location in which the result should be placed must be stored in the program-defined memory location OPR. This method of data reference provides easy access to the data for the complete-operation routines.

Command Routines

The complete-operation routines execute the various chip commands using the APU command names (for example, to call the inverse sine function ASIN, call ASIN). Not all of the APU commands can be performed through this set of routines. Table 3 lists the complete-operation routines. All Data Stack manipulation commands and the no-operation

Table 3. Complete-Operation Routines

Command	Description	Hex Code	Number of Operands	Error Codes
ACOS	Arc cosine	06	1	1100
ASIN	Arc sine	05	1	1100
ATAN	Arc tangent	07	1	None
CHSD	Change sign double	34	1	xx01
CHSF	Change sign floating	15	1	None
CHSS	Change sign single	74	1	xx01
COS	Cosine	03	1	None
DADD	Double add	2C	2	xx01
DDIV	Double divide	2F	2	xx01,1000
DMUL	Double multiply lower	2E	2	xx01
DMUU	Double multiply upper	36	2	xx01
DSUB	Double subtract	2D	2	xx01
EXP	Exponentiation (e^X)	0A	1	1100
FADD	Floating add	10	2	xx01,xx10
FDIV	Floating divide	13	2	xx01,xx10,1000
FIXD	Fix double	1E	1	xx01
FIXS	Fix single	1F	1	xx01
FLTD	Float double	1C	1	None
FLTS	Float single	1D	1	None
FMUL	Floating multiply	12	2	xx01,xx10
FSUB	Floating subtract	11	2	xx01,xx10
LOG	Common logarithm	08	1	0100
LN	Natural logarithm	09	1	0100
PUPI	Push π	1A	0	None
PWR	Power (Y^X)	0B	2	xx01,xx10,1100,0100
SADD	Single add	6C	2	xx01
SDIV	Single divide	6F	2	xx01,1000
SIN	Sine	02	1	None
SMUL	Single multiply lower	6E	2	xx01
SMUU	Single multiply upper	76	2	xx01
SQRT	Square root	01	1	0100
SSUB	Single subtract	6D	2	xx01
TAN	Tangent	04	1	xx01,1000

command have been omitted, as their primary purpose is to manipulate data on the stack. Between routine executions in this set of routines, no data operands are left on the stack.

For each of the 14 arithmetic commands that are associated with addition, subtraction, multiplication, and division, two operands are required. The value referenced by the address in OPB is put onto the Next On Stack (NOS) location and the value referenced by the address in OPA is put in the Top of Stack (TOS) location. For the derived functions and the format conversion routines, only one operand is required. It is referenced by the address in OPA and loaded into the TOS position. After command execution has been completed, the result is in the TOS position. It is retrieved and placed into the system location referenced by the address in OPR. Each command routine then checks the error condition (Table 4). If no error is reported, the routine returns to the calling program. For the cosine (COS), sine (SIN), inverse tangent (ATAN), floating 32-bit fixed point (FLTD), floating 16-bit fixed point (FLTS), floating point sign change (CHSF), and push π (PUPI) commands, no error code is generated by the APU, and the routine returns automatically to the calling program.

Table 4. Am9511 Error Code Definitions

Error Code	Description
0000	No error
1100	Input argument out of range
1000	Divide by zero
0100	Argument of logarithm or square root negative
xx10	Underflow
xx01	Overflow

Error Detection and Recovery

The Am9511 has a total of five error conditions which are listed in Table 4. In some cases the APU itself takes corrective action; for others, the actions are included in the complete-operation routines.

When a divide by zero has been attempted, error code 1000 is generated by the APU. This occurs for the operations of division in single-precision (SDIV), double-precision (DDIV), and floating point (FDIV) formats, as well as for the tangent (TAN) operation. In the division operations, when operand A is zero, the APU sets the result to the value of operand B. The written routines, however, change the result to the maximum value possible in the particular data format with the sign of the value of operand B. This is in acknowledgment of the fundamental

relationships of

$$\lim_{x \to 0} \frac{+y}{x} = +\infty$$

and

$$\lim_{x \to 0} \frac{-y}{x} = -\infty \ .$$

The tangent operation gets the error code as a result of the internal APU operation of sine/cosine (SIN/COS) when operand A is an odd multiple of $\pm\pi/2$ and the cosine is therefore zero. In this case, the result is set to the maximum floating point value with the sign of the sine of A:

$$\lim_{X \to 0} \frac{\pm \sin X}{\cos X} = \pm\infty \ .$$

When the square root (SQRT) or the logarithm of a nonpositive number has been attempted, the error code 0100 is generated by the APU and the APU returns a meaningless result. To prevent this from tying up the program that would use this operation, the complete-operation routine changes the sign of the operand and re-executes the operation. For the common logarithm (LOG) and the natural logarithm (LN) commands, when the input is negative the same action is taken as for SQRT. When the input value for either logarithm is a zero, however, it was discovered that when the APU attempts to compute the result, it goes into an unending, internal microprogram loop. For the logarithm commands, the value of operand A must be checked <u>first</u>,

before command execution. If the input is a zero, the error is flagged and the result is set to zero, without using the APU at all.

When the input values of the inverse sine (ASIN), inverse cosine (ACOS), and e^x (EXP) operations are outside a defined range of values, the error code 1100 is generated. When this occurs for the inverse sine and inverse cosine operations, which have an input range of -1.0 to +1.0, a new operand of 1.0 with the sign of the original operand is entered and the command is executed again. If this occurs for e^x, which has a range of input values between -1.0×2^5 to $+1.0 \times 2^5$, when the input value exceeds the negative bound, the result is set to zero,

$$\lim_{x \to -\infty} e^x = 0 \ ,$$

and when the input value exceeds the positive bound, the result is set to the positive maximum floating point value,

$$\lim_{x \to +\infty} e^x = +\infty \ .$$

When an underflow has occurred during an operation, the error code xx10 is generated. This happens for the floating point arithmetic operations of addition (FADD), subtraction (FSUB), multiplication (FMUL), and division (FDIV). As the minimum value of the floating point format is on the order of 10^{-20}, the result on an underflow condition is set to zero by the written routines.

When an overflow has occurred during an operation, the error code xx01 is generated. Table 5 lists the operations which can achieve this condition.

For each of the following operations on overflow, the written routines flag the error and allow the action taken by the APU to stand. For the single-precision change sign (CHSS) and double-precision change sign (CHSD) operations, overflow occurs when the input value is the most negative value possible in the format ($80\ 00_{16}$ and $80\ 00\ 00\ 00_{16}$, respectively). The input value is returned unchanged by the APU. For the fix floating point to single-precision (FIXS) and fix floating point to double-precision (FIXD) operations, overflow occurs when the integer portion of the floating point operand A is greater than 15 bits and 31 bits, respectively. On that condition, the APU returns the input value unchanged as the result. For the fixed point add and subtract (DADD, DSUB, SADD, and SSUB) operations, when overflow occurs, the least significant 32 bits and 16 bits, respectively, are returned by the APU. For the multiplication commands (DMUL and SMUL), which return the lower half of the product of operands A and B (32 bits and 16 bits, respectively), overflow occurs (1) when the discarded upper half is non-zero (the least significant 32 bits or 16 bits are returned by the APU and the upper half of the product can be found by the programmer executing DMUU or SMUU), or (2) when either of the

Table 5. Am9511 Commands Which May Overflow

Command	Description
CHSD	Double-precision change sign
CHSS	Single-precision change sign
DADD	Double-precision addition
DSUB	Double-precision subtraction
DMUL	Double-precision multiplication, lower
DMUU	Double-precision multiplication, upper
DDIV	Double-precision division
FADD	Floating point addition
FSUB	Floating point subtraction
FMUL	Floating point multiplication
FDIV	Floating point division
FIXD	Fix floating point to double-precision
FIXS	Fix floating point to single-precision
SADD	Single-precision addition
SSUB	Single-precision subtraction
SMUL	Single-precision multiplication, lower
SMUU	Single-precision multiplication, upper
TAN	Tangent

operands has the maximum allowable negative value ($80\ 00\ 00\ 00_{16}$ and $80\ 00_{16}$, respectively). (The result is set to that value by the written routines.) Overflow occurs for the multiplication operations DMUU and SMUU only when either of the operands is the most negative value. The APU returns that value as the result.

For the following operations, the results from an overflow condition returned by the APU are replaced by corrections from the written routines. For a tangent (TAN) operation, overflow occurs when the value of operand A is very near $\pm\pi/2$ radians. The corrective action taken in the written program is to set the result to the floating point maximum value with the sign of sin A (using the same reasoning as for a zero divide correction for TAN). For the floating point addition (FADD), subtraction (FSUB), multiplication (FMUL), and division (FDIV) operations on overflow, the APU returns an accurate mantissa value with the exponent offset by -128. The error recovery by the written routines for this condition in these cases is to set the result to the maximum floating point value with the same sign as the result generated by the APU logic. For the fixed point division commands (DDIV and SDIV), overflow occurs if either operand is the most negative value ($80\ 00\ 00\ 00_{16}$ and $80\ 00_{16}$, respectively). The result is set to that negative value by the complete-operation routine.

The power (PWR) command, Y^X, is set aside for separate discussion because this command is capable of generating four of the five error conditions. The value is calculated by the APU using the following formula:

$$B^A = e^{A \times \ell n\ B} .$$

The error conditions possible are: logarithm of a nonpositive number (0100); argument of e^X out of range (1100); floating point underflow (xx10); and floating point overflow (xx01). For the error condition 0100 (natural logarithm of a nonpositive number), operand B must be tested before command execution. If B is zero, the result is set to zero and the complete-operation routine returns to the calling program without the chip being called to execute. If B is negative, the sign is changed and then the command is executed. For the error condition 1100 (argument of e^X out of range), which occurs when the value of $A \times \ell n\ B$ is outside the range of -1.0×2^5 to $+1.0 \times 2^5$, the corrective action taken is similar to the action for e^X (EXP) operation. For the error condition xx10 (floating point underflow), where the underflow is the result of $A \times \ell n\ B$, the complete-operation routine sets the result of the multiplication to zero and sets the result of the command to $e^0 = 1$. For the condition xx01 (floating point overflow), the corrective action taken is the same as

for an argument exceeding the positive bound of e^x: the result is set to the positive floating point maximum.

C. ASCII Conversion Routines

In order to prepare data for use for the Am9511, routines for conversion from ASCII to floating point and floating point to ASCII were written. The routine for ASCII to floating point conversion, INASCII, takes a string of ASCII characters and decodes the string into the equivalent floating point value. If the user wishes the value to be in either of the single-precision or double-precision formats, use of the FIXS and FIXD routines will accomplish the conversion. A floating point value is converted to an equivalent ASCII string in scientific notation by the routine OUTASCII. For output of single-precision and double-precision values, the user must first use the FLTS and FLTD routines.

The routine for ASCII to floating point conversion, INASCII, takes a string of ASCII characters in integer, decimal, or scientific notation and decodes the string into the equivalent floating point value. The routine requires the address for the result to be stored in the two bytes located at the program-defined memory location, MANTADDR, and the address of the leading character in the string to be in the Index Register. The routine first checks for a sign character and records it for later use (where there is no sign, the value is assumed to be positive). The routine

then checks for integer digits. Each digit is stripped of its ASCII code and added to 10 times the previously accumulated mantissa. If a decimal point is encountered, the routine continues to add all digits to the mantissa but counts the number of decimal digits. The result is a value equivalent to the input digits: an integer mantissa times 10 raised to the negative value of the counter of decimal digits. If the input string is in scientific notation, the "E" is deciphered, the exponent sign is flagged, and the one- or two-digit exponent value is computed (the exponent will have the count of decimal digits subtracted from it). Then the value of 10 raised to the power expressed by the exponent minus the counter is computed and multiplied by the integer mantissa. The sign of the input string is recalled and the value is adjusted accordingly; the value is then stored in the address defined in MANTADDR. If the user wishes the value to be in single- or double-precision format, the FIXS and FIXD routines will accomplish the conversion.

The routine for floating point to ASCII conversion, OUTASCII, takes a floating point value and decodes the bits for assembling an equivalent ASCII string. The routine requires the address of the value to be converted to be in the Index Register. The routine first checks for a value of zero. If it is zero, the output string is set to

Ø0ØØØØØØØØØØ .

If the value is not zero, the mantissa bits are decoded, one byte at a time, by the subroutine ASCACC. This routine makes use of a table, 24 × 9 bytes, which contains a decimal value for each bit of the mantissa (the most significant bit, bit 23, which represents 2^{-1}, has a table value of $05\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00_{16}$; bit 22, 2^{-2}, has a table value of $02\ 05\ 00\ 00\ 00\ 00\ 00\ 00\ 00_{16}$; and so forth). The table value corresponding to each set bit of the mantissa is added to an accumulator. When the mantissa has been converted, the sign is then decoded from the most significant bit of the value (bit 31) and set in the string. If the value has a positive exponent, the accumulator is doubled and the exponent decremented until it is zero (due to the exponent being a power of two). If the exponent value is negative, the accumulator is halved and the exponent is incremented until it is zero. When the exponent is zero, the accumulator is rounded to seven digits. The digits are then encoded to ASCII characters, and a decimal point is added into the string. The power of 10 is computed by incrementing during the doubling process whenever the accumulator has to be shifted to the right, and by decrementing during the halving process when the accumulator is shifted to the left. It is decoded, its sign is determined, and it is added onto the end of the ASCII string. If the power of 10 is a zero, the exponent

field is set to four spaces. The final form of the ASCII string is:

sn.nnnnnnEsnn .

If a single- or double-precision value needs to be output, it should first be converted to floating point by the routines FLTS and FLTD.

CHAPTER V

SUMMARY

The Am9511 was easily adapted into the M6800 Central Processing Unit (CPU) operating system. Implementation into any other microprocessor operating system should be straightforward. All that is required from a programming standpoint is a method to access the Arithmetic Processing Unit (APU) Command and Status Registers and the Data Stack. The flow diagrams in Appendix B can be applied to any microprocessor system, with suitable adaptations for that microprocessor's instruction set.

The use of the data formats peculiar to the Am9511 APU should pose no problem to any user wishing to include the APU in his system. Previously, the data format has been affected by the programmed arithmetic and derived function routines and the capabilities of the microprocessor in use; for the Am9511, the data formats are independent of the microprocessor.

Two sets of routines were developed. The first set, the utility routines, is intended for a user working primarily in an assembler language. Knowledge of the effect of each operation on the APU Data Stack permits the user to stack operands, execute operations, perform manipulations on the Data Stack, and unstack operands in the order most convenient to his programming. The second set,

the complete-operation routines, is intended for adaptation into a language interpreter or compiler, providing error analysis and recovery. These routines can be adapted into a compiler for breakdown of equations to simple operations of the form

$$R = B \text{ op } A$$

and

$$R = \text{op } A \ ,$$

where op represents any operation. For example, instead of having a compiler call a software routine for R = log A, the compiler would store the address of A in OPA, the address of R in OPR, and call LOG. The error handling can also be used during a run of the compiled program.

Few problems were encountered in developing the routines. The most noteworthy was the discovery that an attempted logarithm of the value of zero causes the APU to go into an infinite microprogram loop. This was not covered in the manufacturer's specification sheets. A test code was required for the ULOG, ULN, UPWR, LOG, LN, and PWR routines. For the logarithm routines, operand A had to be tested for zero; for the power routines, operand B was tested. If a zero was detected, the result, R, was set to zero, and the routines returned to the calling program.

BIBLIOGRAPHY

BIBLIOGRAPHY

Am9511 Arithmetic Processing Unit Specification Sheets. Sunnyvale, California: Advanced Micro Devices, 1978.

M6800 Programming Reference Manual. Phoenix: Motorola, Inc., 1978.

Osborne, Adam, with Susanna Jacobson and Jerry Kane. An Introduction to Microcomputers. Vol. II. June 1977 Revision. Berkeley, California: Adam Osborne and Associates, Inc., 1977.

Parker, Richard O., and Joseph H. Kroeger. Algorithm Details for the Am9511 Arithmetic Processing Unit. Sunnyvale, California: Advanced Micro Devices, 1978.

Soucek, Branko. Microprocessors and Microcomputers. New York: John Wiley and Sons, Inc., 1976.

APPENDICES

APPENDIX A

Am9511 COMMANDS

This appendix contains the manufacturer's algorithm details for each of the Am9511 commands (see Fig. A-1).

ACOS

32-BIT FLOATING-POINT INVERSE COSINE

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	1	1	0

Hex Coding: 86 with sr = 1
06 with sr = 0

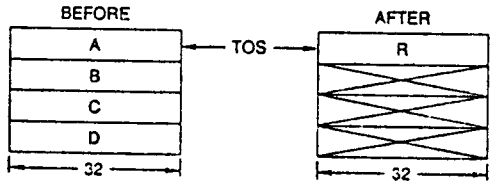
Execution Time: 6304 to 6284 clock cycles

Description:
The 32-bit floating-point operand A at the TOS is replaced by the 32-bit floating-point inverse cosine of A. The result R is a value in radians between 0 and π. Initial operands A, B, C and D are lost.
ACOS will accept all input data values within the range of −1.0 to +1.0. Values outside this range will return an error code of 1100 in the status register.

Accuracy: ACOS exhibits a maximum relative error of 2.0 x 10^{-7} over the valid input data range.

Status Affected: Sign, Zero, Error Field

STACK CONTENTS



ASIN

32-BIT FLOATING-POINT INVERSE SINE

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	1	0	1

Hex Coding: 85 with sr = 1
05 with sr = 0

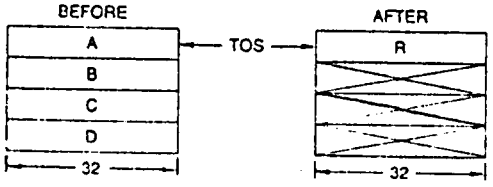
Execution Time: 6230 to 7938 clock cycles

Description:
The 32-bit floating-point operand A at the TOS is replaced by the 32-bit floating-point inverse sine of A. The result R is a value in radians between $-\pi/2$ and $+\pi/2$. Initial operands A, B, C and D are lost.
ASIN will accept all input data values within the range of −1.0 to +1.0. Values outside this range will return an error code of 1100 in the status register.

Accuracy: ASIN exhibits a maximum relative error of 4.0 x 10^{-7} over the valid input data range.

Status Affected: Sign, Zero, Error Field

STACK CONTENTS



ATAN

32-BIT FLOATING-POINT INVERSE TANGENT

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	1	1	1

Hex Coding: 87 with sr = 1
07 with sr = 0

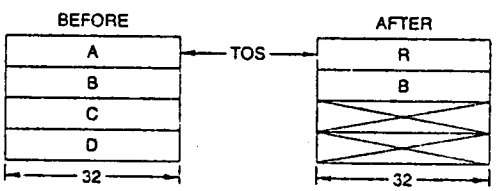
Execution Time: 4992 to 6536 clock cycles

Description:
The 32-bit floating-point operand A at the TOS is replaced by the 32-bit floating-point inverse tangent of A. The result R is a value in radians between $-\pi/2$ and $+\pi/2$. Initial operands A, C and D are lost. Operand B is unchanged.
ATAN will accept all input data values that can be represented in the floating point format.

Accuracy: ATAN exhibits a maximum relative error of 3.0 x 10^{-7} over the input data range.

Status Affected: Sign, Zero

STACK CONTENTS



CHSD

32-BIT FIXED-POINT SIGN CHANGE

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	1	0	1	0	0

Hex Coding: B4 with sr = 1
34 with sr = 0

Execution Time: 26 to 28 clock cycles

Description:
The 32-bit fixed-point two's complement integer operand A at the TOS is subtracted from zero. The result R replaces A at the TOS. Other entries in the stack are not disturbed.
Overflow status will be set and the TOS will be returned unchanged when A is input as the most negative value possible in the format since no positive equivalent exists.

Status Affected: Sign, Zero, Error Field (overflow)

STACK CONTENTS

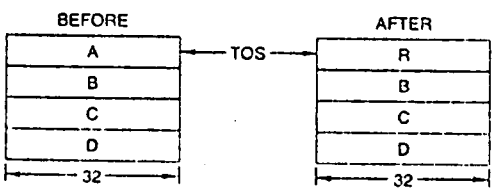


Figure A-1. Command descriptions for the Am9511.

Source: Richard O. Parker and Joseph H. Kroeger, Algorithm Details for the Am9511 Arithmetic Processing Unit (Sunnyvale, California: Advanced Micro Devices, 1978), pp. 9-23.

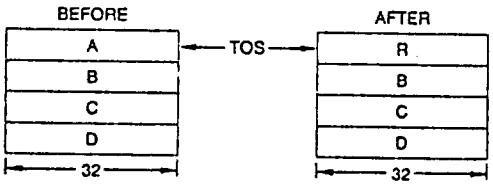
CHSF

32-BIT FLOATING-POINT SIGN CHANGE

Binary Coding:	7	6	5	4	3	2	1	0
	sr	0	0	1	0	1	0	1

Hex Coding: 95 with sr = 1
15 with sr = 0
Execution Time: 16 to 20 clock cycles
Description:
The sign of the mantissa of the 32-bit floating-point operand A at the TOS is inverted. The result R replaces A at the TOS. Other stack entries are unchanged.
If A is input as zero (mantissa MSB = 0), no change is made.
Status Affected: Sign, Zero

STACK CONTENTS



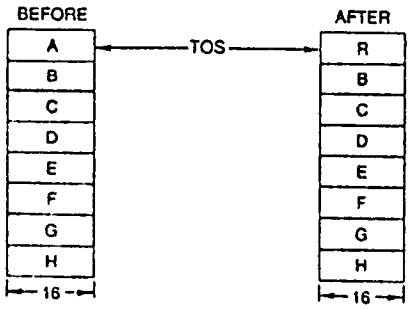
CHSS

16-BIT FIXED-POINT SIGN CHANGE

Binary Coding:	7	6	5	4	3	2	1	0
	sr	1	1	1	0	1	0	0

Hex Coding: F4 with sr = 1
74 with sr = 0
Execution Time: 22 to 24 clock cycles
Description:
16-bit fixed-point two's complement integer operand A at the TOS is subtracted from zero. The result R replaces A at the TOS. All other operands are unchanged.
Overflow status will be set and the TOS will be returned unchanged when A is input as the most negative value possible in the format since no positive equivalent exists.
Status Affected: Sign, Zero, Overflow

STACK CONTENTS



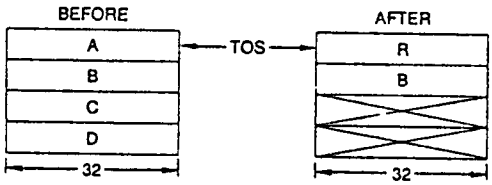
COS

32-BIT FLOATING-POINT COSINE

Binary Coding:	7	6	5	4	3	2	1	0
	sr	0	0	0	0	0	1	1

Hex Coding: 83 with sr = 1
03 with sr = 0
Execution Time: 3840 to 4878 clock cycles
Description:
The 32-bit floating-point operand A at the TOS is replaced by R, the 32-bit floating-point cosine of A. A is assumed to be in radians. Operands A, C and D are lost. B is unchanged.
The COS function can accept any input data value that can be represented in the data format. All input values are range reduced to fall within an interval of $-\pi/2$ to $+\pi/2$ radians.
Accuracy: COS exhibits a maximum relative error of 5.0×10^{-7} for all input data values in the range of -2π to $+2\pi$ radians.
Status Affected: Sign, Zero

STACK CONTENTS



DADD

32-BIT FIXED-POINT ADD

Binary Coding:	7	6	5	4	3	2	1	0
	sr	0	1	0	1	1	0	0

Hex Coding: AC with sr = 1
2C with sr = 0
Execution Time: 20 to 22 clock cycles
Description:
The 32-bit fixed-point two's complement integer operand A at the TOS is added to the 32-bit fixed-point two's complement integer operand B at the NOS. The result R replaces operand B and the Stack is moved up so that R occupies the TOS. Operand B is lost. Operands A, C and D are unchanged. If the addition generates a carry it is reported in the status register.
If the result is too large to be represented by the data format, the least significant 32 bits of the result are returned and overflow status is reported.
Status Affected: Sign, Zero, Carry, Error Field

STACK CONTENTS

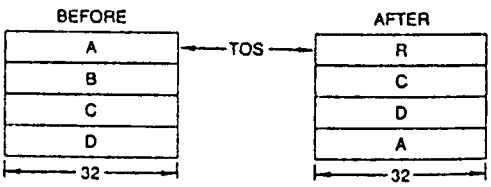


Figure A-1 (continued)

DDIV

32-BIT FIXED-POINT DIVIDE

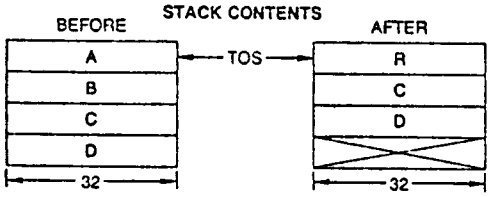
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	0	1	1	1	1

Hex Coding: AF with sr = 1
2F with sr = 0

Execution Time: 196 to 210 clock cycles when A ≠ 0
18 clock cycles when A = 0.

Description:
The 32-bit fixed-point two's complement integer operand B at NOS is divided by the 32-bit fixed-point two's complement integer operand A at the TOS. The 32-bit integer quotient R replaces B and the stack is moved up so that R occupies the TOS. No remainder is generated. Operands A and B are lost. Operands C and D are unchanged.
If A is zero, R is set equal to B and the divide-by-zero error status will be reported. If either A or B is the most negative value possible in the format, R will be meaningless and the overflow error status will be reported.

Status Affected: Sign, Zero, Error Field



DMUL

32-BIT FIXED-POINT MULTIPLY, LOWER

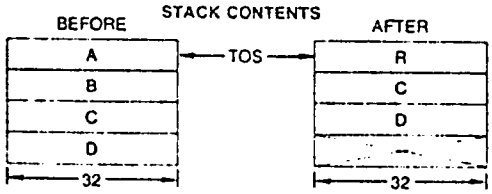
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	0	1	1	1	0

Hex Coding: AE with sr = 1
2E with sr = 0

Execution Time: 194 to 210 clock cycles

Description:
The 32-bit fixed-point two's complement integer operand A at the TOS is multiplied by the 32-bit fixed-point two's complement integer operand B at the NOS. The 32-bit least significant half of the product R replaces B and the stack is moved up so that R occupies the TOS. The most significant half of the product is lost. Operands A and B are lost. Operands C and D are unchanged.
The overflow status bit is set if the discarded upper half was non-zero. If either A or B is the most negative value that can be represented in the format, that value is returned as R and the overflow status is set.

Status Affected: Sign, Zero, Overflow



DMUU

32-BIT FIXED-POINT MULTIPLY, UPPER

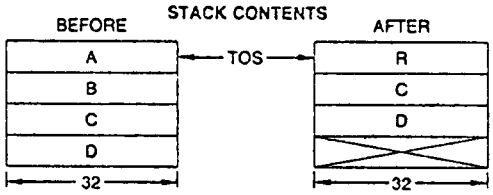
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	1	0	1	1	0

Hex Coding: B6 with sr = 1
36 with sr = 0

Execution Time: 182 to 218 clock cycles

Description:
The 32-bit fixed-point two's complement integer operand A at the TOS is multiplied by the 32-bit fixed-point two's complement integer operand B at the NOS. The 32-bit most significant half of the product R replaces B and the stack is moved up so that R occupies the TOS. The least significant half of the product is lost. Operands A and B are lost. Operands C and D are unchanged.
If A or B was the most negative value possible in the format, overflow status is set and R is meaningless.

Status Affected: Sign, Zero, Overflow



DSUB

32-BIT FIXED-POINT SUBTRACT

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	0	1	1	0	1

Hex Coding: AD with sr = 1
2D with sr = 0

Execution Time: 38 to 40 clock cycles

Description:
The 32-bit fixed-point two's complement operand A at the TOS is subtracted from the 32-bit fixed-point two's complement operand B at the NOS. The difference R replaces operand B and the stack is moved up so that R occupies the TOS. Operand B is lost. Operands A, C and D are unchanged.
If the subtraction generates a borrow it is reported in the carry status bit. If A is the most negative value that can be represented in the format the overflow status is set. If the result cannot be represented in the data format range, the overflow bit is set and the 32 least significant bits of the result are returned as R.

Status Affected: Sign, Zero, Carry, Overflow

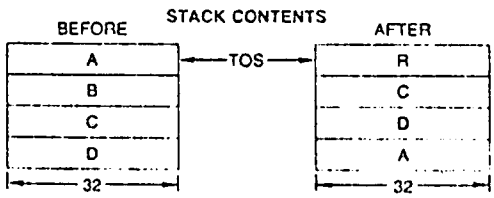


Figure A-1 (continued)

EXP

32-BIT FLOATING-POINT e^X

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	1	0	1	0

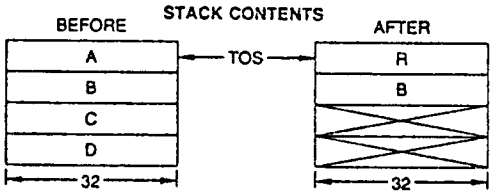
Hex Coding: 8A with sr = 1
0A with sr = 0

Execution Time: 3794 to 4878 clock cycles for $|A| \leq 1.0 \times 2^5$
34 clock cycles for $|A| > 1.0 \times 2^5$

Description:
The base of natural logarithms, e, is raised to an exponent value specified by the 32-bit floating-point operand A at the TOS. The result R of e^A replaces A. Operands A, C and D are lost. Operand B is unchanged.
EXP accepts all input data values within the range of $-1.0 \times 2^{+5}$ to $+1.0 \times 2^{+5}$. Input values outside this range will return a code of 1100 in the error field of the status register.

Accuracy: EXP exhibits a maximum relative error of 5.0×10^{-7} over the valid input data range.

Status Affected: Sign, Zero, Error Field



FADD

32-BIT FLOATING-POINT ADD

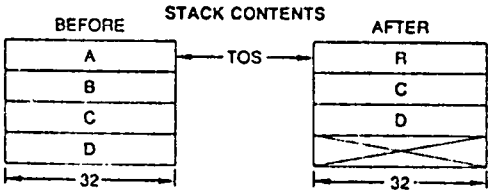
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	0	0	0	0

Hex Coding: 90 with sr = 1
10 with sr = 0

Execution Time: 54 to 368 clock cycles for A ≠ 0
24 clock cycles for A = 0

Description:
32-bit floating-point operand A at the TOS is added to 32-bit floating-point operand B at the NOS. The result R replaces B and the stack is moved up so that R occupies the TOS. Operands A and B are lost. Operands C and D are unchanged.
Exponent alignment before the addition and normalization of the result accounts for the variation in execution time. Exponent overflow and underflow are reported in the status register, in which case the mantissa is correct and the exponent is offset by 128.

Status Affected: Sign, Zero, Error Field



FDIV

32-BIT FLOATING-POINT DIVIDE

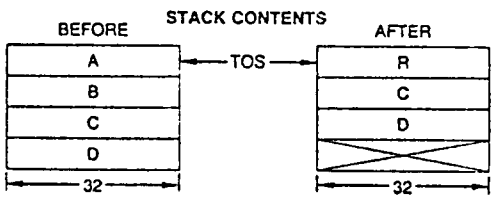
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	0	0	1	1

Hex Coding: 93 with sr = 1
13 with sr = 0

Execution Time: 154 to 184 clock cycles for A ≠ 0
22 clock cycles for A = 0

Description:
32-bit floating-point operand B at NOS is divided by 32-bit floating-point operand A at the TOS. The result R replaces B and the stack is moved up so that R occupies the TOS. Operands A and B are lost. Operands C and D are unchanged.
If operand A is zero, R is set equal to B and the divide-by-zero error is reported in the status register. Exponent overflow or underflow is reported in the status register, in which case the mantissa portion of the result is correct and the exponent portion is offset by 128.

Status Affected: Sign, Zero, Error Field



FIXD

32-BIT FLOATING-POINT TO 32-BIT FIXED-POINT CONVERSION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	1	1	0

Hex Coding: 9E with sr = 1
1E with sr = 0

Execution Time: 90 to 336 clock cycles

Description:
32-bit floating-point operand A at the TOS is converted to a 32-bit fixed-point two's complement integer. The result R replaces A. Operands A and D are lost. Operands B and C are unchanged.

If the integer portion of A is larger than 31 bits when converted, the overflow status will be set and A will not be changed. Operand D, however, will still be lost.

Status Affected: Sign, Zero Overflow

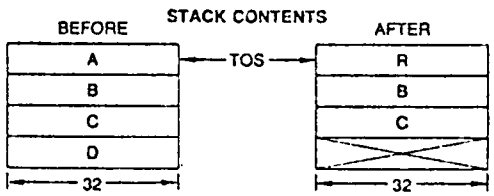


Figure A-1 (continued)

FIXS

32-BIT FLOATING-POINT TO 16-BIT FIXED-POINT CONVERSION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	1	1	1

Hex Coding: 9F with sr = 1
1F with sr = 0

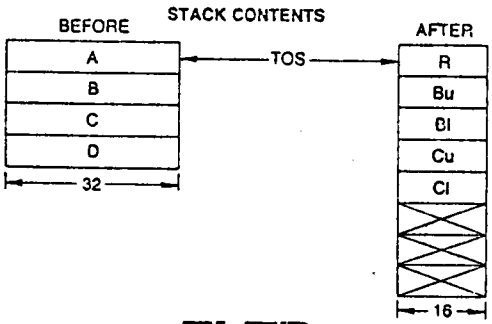
Execution Time: 90 to 214 clock cycles

Description:

32-bit floating-point operand A at the TOS is converted to a 16-bit fixed-point two's complement integer. The result R replaces the lower half of A and the stack is moved up by two bytes so that R occupies the TOS. Operands A and D are lost. Operands B and C are unchanged, but appear as upper (u) and lower (l) halves on the 16-bit wide stack if they are 32-bit operands.

If the integer portion of A is larger than 15 bits when converted, the overflow status will be set and A will not be changed. Operand D, however, will still be lost.

Status Affected: Sign, Zero, Overflow



FLTD

32-BIT FIXED-POINT TO 32-BIT FLOATING-POINT CONVERSION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	1	0	0

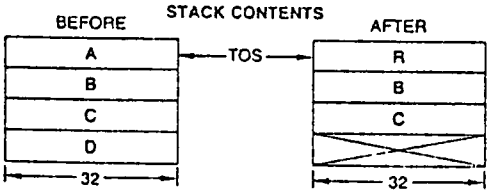
Hex Coding: 9C with sr = 1
1C with sr = 0

Execution Time: 56 to 342 clock cycles

Description:

32-bit fixed-point two's complement integer operand A at the TOS is converted to a 32-bit floating-point number. The result R replaces A at the TOS. Operands A and D are lost. Operands B and C are unchanged.

Status Affected: Sign, Zero



FLTS

16-BIT FIXED-POINT TO 32-BIT FLOATING-POINT CONVERSION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	1	0	1

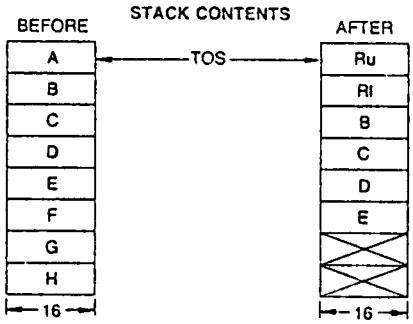
Hex Coding: 9D with sr = 1
1D with sr = 0

Execution Time: 62 to 156 clock cycles

Description:

16-bit fixed-point two's complement integer A at the TOS is converted to a 32-bit floating-point number. The lower half of the result R (Rl) replaces A, the upper half (Ru) replaces H and the stack is moved down so that Ru occupies the TOS. Operands A, F, G and H are lost. Operands B, C, D and E are unchanged.

Status Affected: Sign, Zero



FMUL

32-BIT FLOATING-POINT MULTIPLY

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	0	0	1	0

Hex Coding: 92 with sr = 1
12 with sr = 0

Execution Time: 146 to 168 clock cycles

Description:

32-bit floating-point operand A at the TOS is multiplied by the 32-bit floating-point operand B at the NOS. The normalized result R replaces B and the stack is moved up so that R occupies the TOS. Operands A and B are lost. Operands C and D are unchanged.

Exponent overflow or underflow is reported in the status register, in which case the mantissa portion of the result is correct and the exponent portion is offset by 128.

Status Affected: Sign, Zero, Error Field

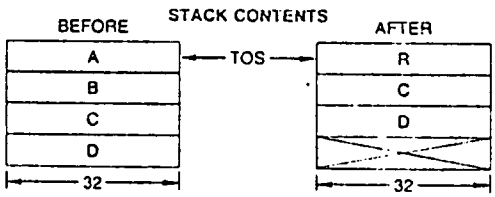


Figure A-1 (continued)

FSUB

32-BIT FLOATING-POINT SUBTRACTION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	0	0	0	1

Hex Coding: 91 with sr = 1
11 with sr = 0

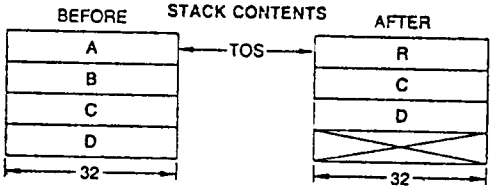
Execution Time: 70 to 370 clock cycles for A ≠ 0
26 clock cycles for A = 0

Description:
32-bit floating-point operand A at the TOS is subtracted from 32-bit floating-point operand B at the NOS. The normalized difference R replaces B and the stack is moved up so that R occupies the TOS. Operands A and B are lost. Operands C and D are unchanged.

Exponent alignment before the subtraction and normalization of the result account for the variation in execution time.

Exponent overflow or underflow is reported in the status register in which case the mantissa portion of the result is correct and the exponent portion is offset by 128.

Status Affected: Sign, Zero, Error Field (overflow)



LOG

32-BIT FLOATING-POINT COMMON LOGARITHM

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	1	0	0	0

Hex Coding: 88 with sr = 1
08 with sr = 0

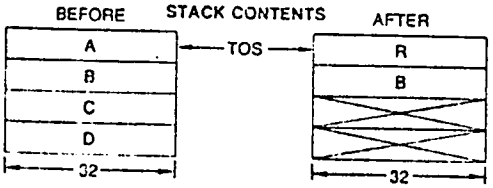
Execution Time: 4474 to 7132 clock cycles for A > 0
20 clock cycles for A ≤ 0

Description:
The 32-bit floating-point operand A at the TOS is replaced by R, the 32-bit floating-point common logarithm (base 10) of A. Operands A, C and D are lost. Operand B is unchanged.

The LOG function accepts any positive input data value that can be represented by the data format. If LOG of a non-positive value is attempted an error status of 0100 is returned.

Accuracy: LOG exhibits a maximum absolute error of 2.0×10^{-7} for the input range from 0.1 to 10, and a maximum relative error of 2.0×10^{-7} for positive values less than 0.1 or greater than 10.

Status Affected: Sign, Zero, Error Field



LN

32-BIT FLOATING-POINT NATURAL LOGARITHM

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	1	0	0	1

Hex Coding: 89 with sr = 1
09 with sr = 0

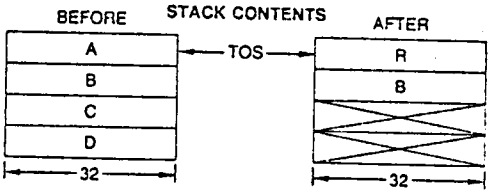
Execution Time: 4298 to 6956 clock cycles for A > 0
20 clock cycles for A ≤ 0

Description:
The 32-bit floating-point operand A at the TOS is replaced by R, the 32-bit floating-point natural logarithm (base e) of A. Operands A, C and D are lost. Operand B is unchanged.

The LN function accepts all positive input data values that can be represented by the data format. If LN of a non-positive number is attempted an error status of 0100 is returned.

Accuracy: LN exhibits a maximum absolute error of 2×10^{-7} for the input range from e^{-1} to e, and a maximum relative error of 2.0×10^{-7} for positive values less than e^{-1} or greater than e.

Status Affected: Sign, Zero, Error Field



NOP

NO OPERATION

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	0	0	0

Hex Coding: 80 with sr = 1
00 with sr = 0

Execution Time: 4 clock cycles

Description:
The NOP command performs no internal data manipulations. It may be used to set or clear the service request interface line without changing the contents of the stack.

Status Affected: The status byte is cleared to all zeroes.

Figure A-1 (continued)

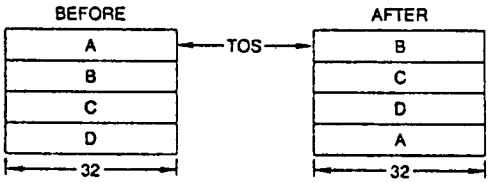
POPD

32-BIT
STACK POP

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	1	1	0	0	0

Hex Coding: B8 with sr = 1
38 with sr = 0
Execution Time: 12 clock cycles
Description:
The 32-bit stack is moved up so that the old NOS becomes the new TOS. The previous TOS rotates to the bottom of the stack. All operand values are unchanged. POPD and POPF execute the same operation.
Status Affected: Sign, Zero

STACK CONTENTS



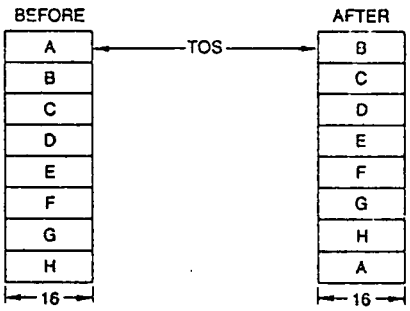
POPS

16-BIT
STACK POP

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	1	1	0	0	0

Hex Coding: F8 with sr = 1
78 with sr = 0
Execution Time: 10 clock cycles
Description:
The 16-bit stack is moved up so that the old NOS becomes the new TOS. The previous TOS rotates to the bottom of the stack. All operand values are unchanged.
Status Affected: Sign, Zero

STACK CONTENTS



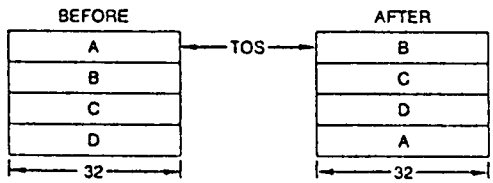
POPF

32-BIT
STACK POP

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	0	0	0

Hex Coding: 98 with sr = 1
18 with sr = 0
Execution Time: 12 clock cycles
Description:
The 32-bit stack is moved up so that the old NOS becomes the new TOS. The old TOS rotates to the bottom of the stack. All operand values are unchanged. POPF and POPD execute the same operation.
Status Affected: Sign, Zero

STACK CONTENTS



PTOD

PUSH 32-BIT
TOS ONTO STACK

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	1	0	1	1	1

Hex Coding: B7 with sr = 1
37 with sr = 0
Execution Time: 20 clock cycles
Description:
The 32-bit stack is moved down and the previous TOS is copied into the new TOS location. Operand D is lost. All other operand values are unchanged. PTOD and PTOF execute the same operation.
Status Affected: Sign, Zero

STACK CONTENTS

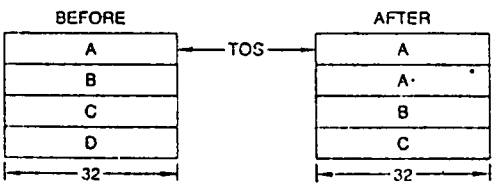


Figure A-1 (continued)

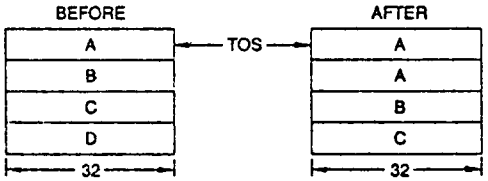
PTOF

PUSH 32-BIT
TOS ONTO STACK

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	0	1	1	1

Hex Coding: 37 with sr = 1
17 with sr = 0
Execution Time: 20 clock cycles
Description:
The 32-bit stack is moved down and the previous TOS is copied into the new TOS location. Operand D is lost. All other operand values are unchanged. PTOF and PTOD execute the same operation.
Status Affected: Sign, Zero

STACK CONTENTS



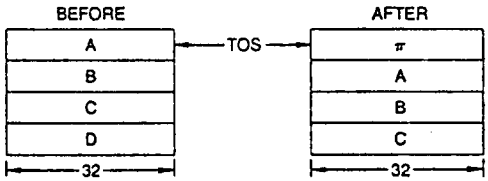
PUPI

PUSH 32-BIT
FLOATING-POINT π

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	0	1	0

Hex Coding: 9A with sr = 1
1A with sr = 0
Execution Time: 16 clock cycles
Description:
The 32-bit stack is moved down so that the previous TOS occupies the new NOS location. 32-bit floating-point constant π is entered into the new TOS location. Operand D is lost. Operands A, B and C are unchanged.
Status Affected: Sign, Zero

STACK CONTENTS



PTOS

PUSH 16-BIT
TOS ONTO STACK

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	1	0	1	1	1

Hex Coding: F7 with sr = 1
77 with sr = 0
Execution Time: 16 clock cycles
Description:
The 16-bit stack is moved down and the previous TOS is copied into the new TOS location. Operand H is lost and all other operand values are unchanged.
Status Affected: Sign, Zero

STACK CONTENTS

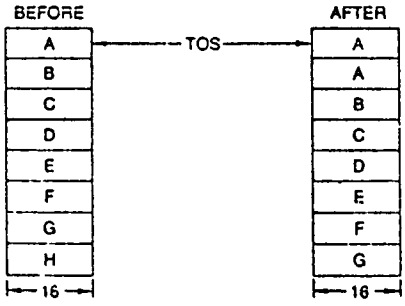


Figure A-1 (continued)

PWR

32-BIT
FLOATING-POINT X^Y

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	1	0	1	1

Hex Coding: 8B with sr = 1
0B with sr = 0

Execution Time: 8290 to 12032 clock cycles

Description:

32-bit floating-point operand B at the NOS is raised to the power specified by the 32-bit floating-point operand A at the TOS. The result R of B^A replaces B and the stack is moved up so that R occupies the TOS. Operands A, B, and D are lost. Operand C is unchanged.

The PWR function accepts all input data values that can be represented in the data format for operand A and all positive values for operand B. If operand B is non-positive an error status of 0100 will be returned. The EXP and LN functions are used to implement PWR using the relationship $B^A = EXP\ [A(LN\ B)]$. Thus if the term $[A(LN\ B)]$ is outside the range of $-1.0 \times 2^{+5}$ to $+1.0 \times 2^{+5}$ an error status of 1100 will be returned. Underflow and overflow conditions can occur.

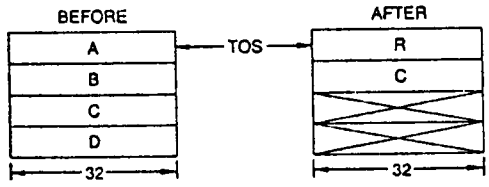
Accuracy: The error performance for PWR is a function of the LN and EXP performance as expressed by:

$|(\text{Relative Error})_{PWR}| \approx |(\text{Relative Error})_{EXP} + |A(\text{Absolute Error})_{LN}|$

The maximum relative error for PWR occurs when A is at its maximum value while $[A(LN\ B)]$ is near 1.0×2^5 and the EXP error is also at its maximum. For most practical applications the relative error for PWR will be less than 7.0×10^{-7}.

Status Affected: Sign, Zero, Error Field

STACK CONTENTS



SADD

16-BIT
FIXED-POINT ADD

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	0	1	1	0	0

Hex Coding: EC with sr = 1
6C with sr = 0

Execution Time: 16 to 18 clock cycles

Description:

16-bit fixed-point two's complement integer operand A at the TOS is added to 16-bit fixed-point two's complement integer operand B at the NOS. The result R replaces B and the stack is moved up so that R occupies the TOS. Operand B is lost. All other operands are unchanged.

If the addition generates a carry bit it is reported in the status register. If an overflow occurs it is reported in the status register and the 16 least significant bits of the result are returned.

Status Affected: Sign, Zero, Carry, Error Field

STACK CONTENTS

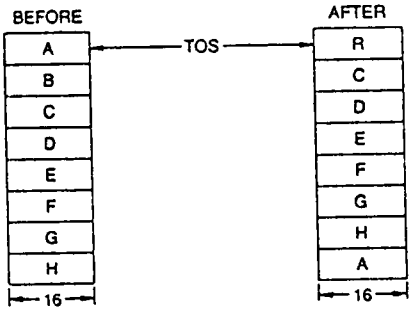


Figure A-1 (continued)

SDIV

16-BIT
FIXED-POINT DIVIDE

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	0	1	1	1	1

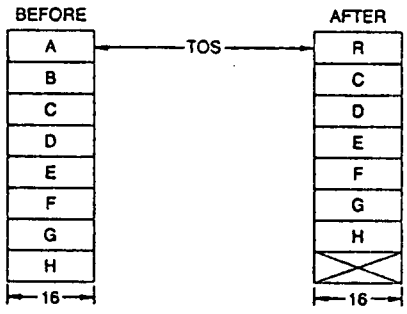
Hex Coding: EF with sr = 1
6F with sr = 0

Execution Time: 84 to 94 clock cycles for A ≠ 0
14 clock cycles for A = 0

Description:
16-bit fixed-point two's complement integer operand B at the NOS is divided by 16-bit fixed-point two's complement integer operand A at the TOS. The 16-bit integer quotient R replaces B and the stack is moved up so that R occupies the TOS. No remainder is generated. Operands A and B are lost. All other operands are unchanged.
If A is zero, R will be set equal to B and the divide-by-zero error status will be reported.

Status Affected: Sign, Zero, Error Field

STACK CONTENTS



SIN

32-BIT
FLOATING-POINT SINE

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	0	1	0

Hex Coding: 82 with sr = 1
02 with sr = 0

Execution Time: 3796 to 4808 clock cycles for $|A| > 2^{-12}$ radians
30 clock cycles for $|A| \leq 2^{-12}$ radians

Description:
The 32-bit floating-point operand A at the TOS is replaced by R, the 32-bit floating-point sine of A. A is assumed to be in radians. Operands A, C and D are lost. Operand B is unchanged.

The SIN function will accept any input data value that can be represented by the data format. All input values are range reduced to fall within the interval $-\pi/2$ to $+\pi/2$ radians.

Accuracy: SIN exhibits a maximum relative error of 5.0×10^{-7} for input values in the range of -2π to $+2\pi$ radians.

Status Affected: Sign, Zero

STACK CONTENTS

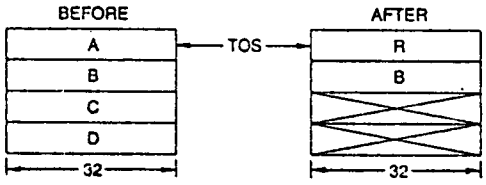


Figure A-1 (continued)

SMUL

16-BIT FIXED-POINT
MULTIPLY, LOWER

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	0	1	1	1	0

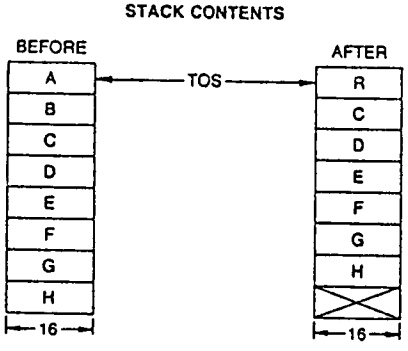
Hex Coding: EE with sr = 1
6E with sr = 0

Execution Time: 84 to 94 clock cycles

Description:

16-bit fixed-point two's complement integer operand A at the TOS is multiplied by the 16-bit fixed-point two's complement integer operand B at the NOS. The 16-bit least significant half of the product R replaces B and the stack is moved up so that R occupies the TOS. The most significant half of the product is lost. Operands A and B are lost. All other operands are unchanged. The overflow status bit is set if the discarded upper half was non-zero. If either A or B is the most negative value that can be represented in the format, that value is returned as R and the overflow status is set.

Status Affected: Sign, Zero, Error Field



SMUU

16-BIT FIXED-POINT
MULTIPLY, UPPER

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	1	0	1	1	0

Hex Coding: F6 with sr = 1
76 with sr = 0

Execution Time: 80 to 98 clock cycles

Description:

16-bit fixed-point two's complement integer operand A at the TOS is multiplied by the 16-bit fixed-point two's complement integer operand B at the NOS. The 16-bit most significant half of the product R replaces B and the stack is moved up so that R occupies the TOS. The least significant half of the product is lost. Operands A and B are lost. All other operands are unchanged.

If either A or B is the most negative value that can be represented in the format, that value is returned as R and the overflow status is set.

Status Affected: Sign, Zero, Error Field

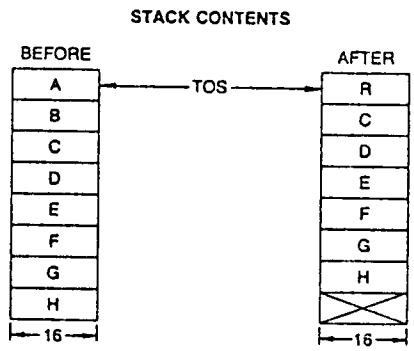


Figure A-1 (continued)

SQRT

32-BIT FLOATING-POINT SQUARE ROOT

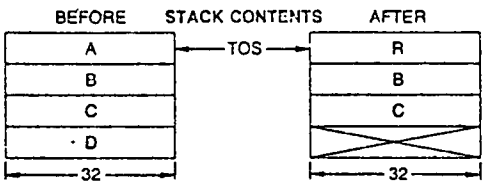
	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	0	0	1

Hex Coding: 81 with sr = 1
01 with sr = 0

Execution Time: 782 to 870 clock cycles

Description:
32-bit floating-point operand A at the TOS is replaced by R, the 32-bit floating-point square root of A. Operands A and D are lost. Operands B and C are not changed.
SQRT will accept any non-negative input data value that can be represented by the data format. If A is negative an error code of 0100 will be returned in the status register.

Status Affected: Sign, Zero, Error Field



SSUB

16-BIT FIXED-POINT SUBTRACT

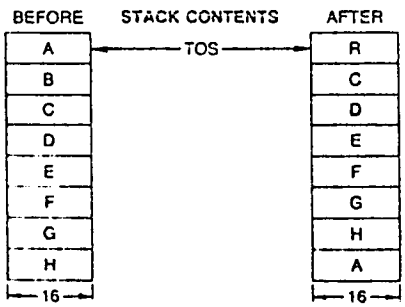
	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	0	1	1	0	1

Hex Coding: ED with sr = 1
6D with sr = 0

Execution Time: 30 to 32 clock cycles

Description:
16-bit fixed-point two's complement integer operand A at the TOS is subtracted from 16-bit fixed-point two's complement integer operand B at the NOS. The result R replaces B and the stack is moved up so that R occupies the TOS. Operand B is lost. All other operands are unchanged.
If the subtraction generates a borrow it is reported in the carry status bit. If A is the most negative value that can be represented in the format the overflow status is set. If the result cannot be represented in the format range, the overflow status is set and the 16 least significant bits of the result are returned as R.

Status Affected: Sign, Zero, Carry, Error Field



TAN

32-BIT FLOATING-POINT TANGENT

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	0	0	1	0	0

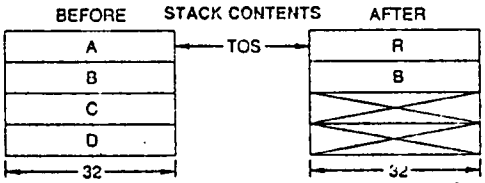
Hex Coding: 64 with sr = 1
04 with sr = 0

Execution Time: 4894 to 5886 clock cycles for $|A| > 2^{-12}$ radians
30 clock cycles for $|A| \leq 2^{-12}$ radians

Description:
The 32-bit floating-point operand A at the TOS is replaced by the 32-bit floating-point tangent of A. Operand A is assumed to be in radians. A, C and D are lost. B is unchanged.
The TAN function will accept any input data value that can be represented in the data format. All input data values are range-reduced to fall within $-\pi/4$ to $+\pi/4$ radians. TAN is unbounded for input values near odd multiples of $\pi/2$ and in such cases the overflow bit is set in the status register. For angles smaller than 2^{-12} radians, TAN returns A as the tangent of A.

Accuracy: TAN exhibits a maximum relative error of 5.0 x 10^{-7} for input data values in the range of -2π to $+2\pi$ radians except for data values near odd multiples of $\pi/2$.

Status Affected: Sign, Zero, Error Field (overflow)



XCHD

EXCHANGE 32-BIT STACK OPERANDS

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	1	1	1	0	0	1

Hex Coding: B9 with sr = 1
39 with sr = 0

Execution Time: 26 clock cycles

Description:
32-bit operand A at the TOS and 32-bit operand B at the NOS are exchanged. After execution, B is at the TOS and A is at the NOS. All operands are unchanged. XCHD and XCHF execute the same operation.

Status Affected: Sign, Zero

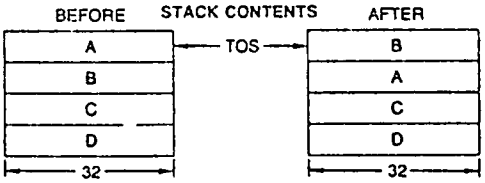


Figure A-1 (continued)

XCHF

EXCHANGE 32-BIT
STACK OPERANDS

	7	6	5	4	3	2	1	0
Binary Coding:	sr	0	0	1	1	0	0	1

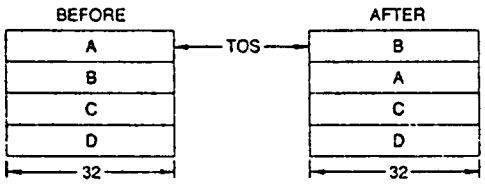
Hex Coding: 99 with sr = 1
19 with sr = 0

Execution Time: 26 clock cycles

Description:
32-bit operand A at the TOS and 32-bit operand B at the NOS are exchanged. After execution, B is at the TOS and A is at the NOS. All operands are unchanged. XCHD and XCHF execute the same operation.

Status Affected: Sign, Zero

STACK CONTENTS



XCHS

EXCHANGE 16-BIT
STACK OPERANDS

	7	6	5	4	3	2	1	0
Binary Coding:	sr	1	1	1	1	0	0	1

Hex Coding: F9 with sr = 1
79 with sr = 0

Execution Time: 18 clock cycles

Description:
16-bit operand A at the TOS and 16-bit operand B at the NOS are exchanged. After execution, B is at the TOS and A is at the NOS. All operand values are unchanged.

Status Affected: Sign, Zero

STACK CONTENTS

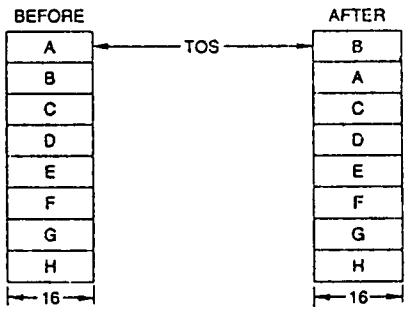


Figure A-1 (continued)

APPENDIX B

CODE LISTINGS AND FLOW CHARTS

In this appendix, the M6800 assembly code listings of the routines described in this investigation are provided. In addition, flow charts are provided in a generalized form so that a user may adapt these routines to other microcomputer systems.

List of Symbols

AccA	M6800 accumulator A
AccB	M6800 accumulator B
h	Indicates hexadecimal number
IX	Index register
[nnnn]	Refers to contents of address nnnn
•	Boolean and
+	Add
⊕	Boolean exclusive or
-	Subtract
←	Assign
addr	Address
cmmc	Command code
dec	Decrement
inc	Increment

$D01C_{16}$	Address of Am9511 data stack
$D01E_{16}$	Address of Am9511 command and status registers
APU NOS	APU next on stack
APU TOS	APU top of stack
MSB	Most significant byte
LSB	Least significant byte
FLT1	A floating point 1
FLT10	A floating point 10
OPA	Contains address of APU operand A
OPB	Contains address of APU operand B
OPR	Contains address of APU result R
ERRSTAT	Status byte
ERRTOT	Total errors
ERRxx01	Overflow
ERRxx10	Underflow
ERR0100	Negative argument of log or square root
ERR1000	Zero divisor
ERR1100	Argument out of input range
MANT	Mantissa for INASCII
EXPNT	Exponent for INASCII
EXPCNT	Decimal counter for INASCII
FLAG	Sign flag for INASCII
STNGAD	Address save location for INASCII
MANTAD	Address save location for INASCII
SIGN	Sign for OUTASCII
ACC	Digit string for OUTASCII
ACCF1	Digit string for OUTASCII

ACCLST	E character for OUTASCII
ACCTRL	E sign for OUTASCII
PWR10	Power of 10 for OUTASCII
EDGT	Digit for E for OUTASCII
TAG	Delimiter
SAVACC	Address save location for OUTASCII
SAVF	Address save location for OUTASCII
SAVX	Address save location for OUTASCII
FRCLST	Addresses for FRCTBL
FRCTBL	Values for OUTASCII conversion
~	Not

```
OPA       DB    2          Location of addr of operand A
OPB       DB    2          Location of addr of operand B
OPR       DB    2          Location of addr for result R

STKB4     LDX   OPB        Load IX with addr of operand B
          BRA   STK4       Go to stacking routine
STKA4     LDX   OPA        Load IX with addr of operand A
STK4      INX              Adjust IX to addr LSB of operand
          INX
          INX
STKIT4    BSR   STKIT2     Stack two bytes
STKIT2    BSR   STKIT1     Stack one byte
STKIT1    LDA   B,X        Transfer byte at [IX] to APU TOS
          STA   B,D01Ch      via AccB
          DEX              Address preceding byte of operand
          RTS              Return
STKB2     LDX   OPB        Load IX with addr of operand B
          BRA   STK2       Go to stacking routine
STKA2     LDX   OPA        Load IX with addr of operand A
STK2      INX              Adjust IX to addr LSB of operand
          BRA   STKIT2     Stack
```

(see Fig. B-1)

```
UNSTKR2   LDX   OPR        Load IX with addr of result R
          BRA   UNSTK2     Go to 2-byte unstack
UNSTKR4   LDX   OPR        Load IX with addr of result R
UNSTK4    BSR   UNSTK2     Unstack two bytes
UNSTK2    BSR   UNSTK1     Unstack one byte
UNSTK1    LDA   A,D01Ch    Transfer byte at APU TOS to [IX]
          STA   A,X          via AccA
          INX              Address next byte of result
          RTS              Return
```

(see Fig. B-2)

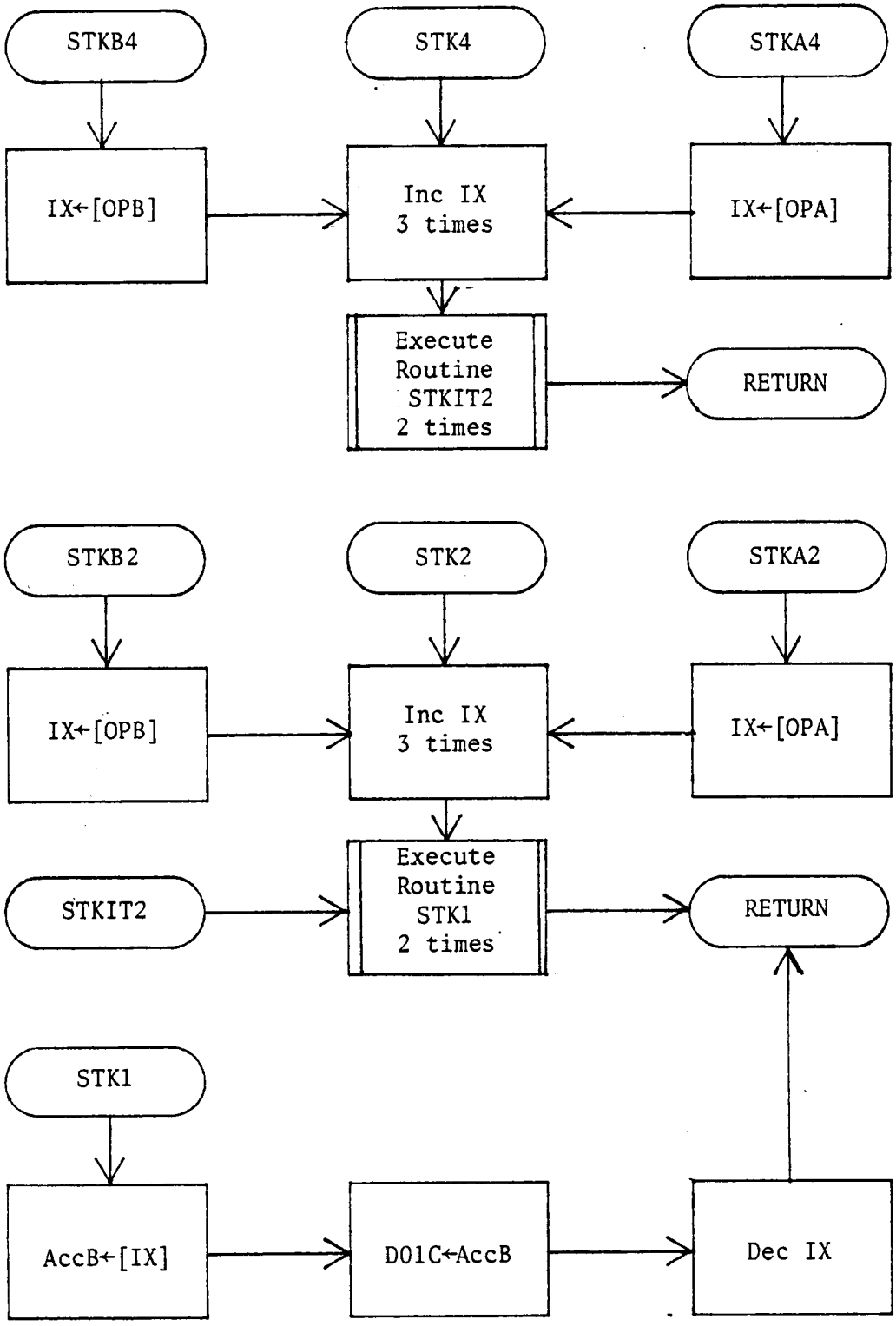


Figure B-1. Stack routines.

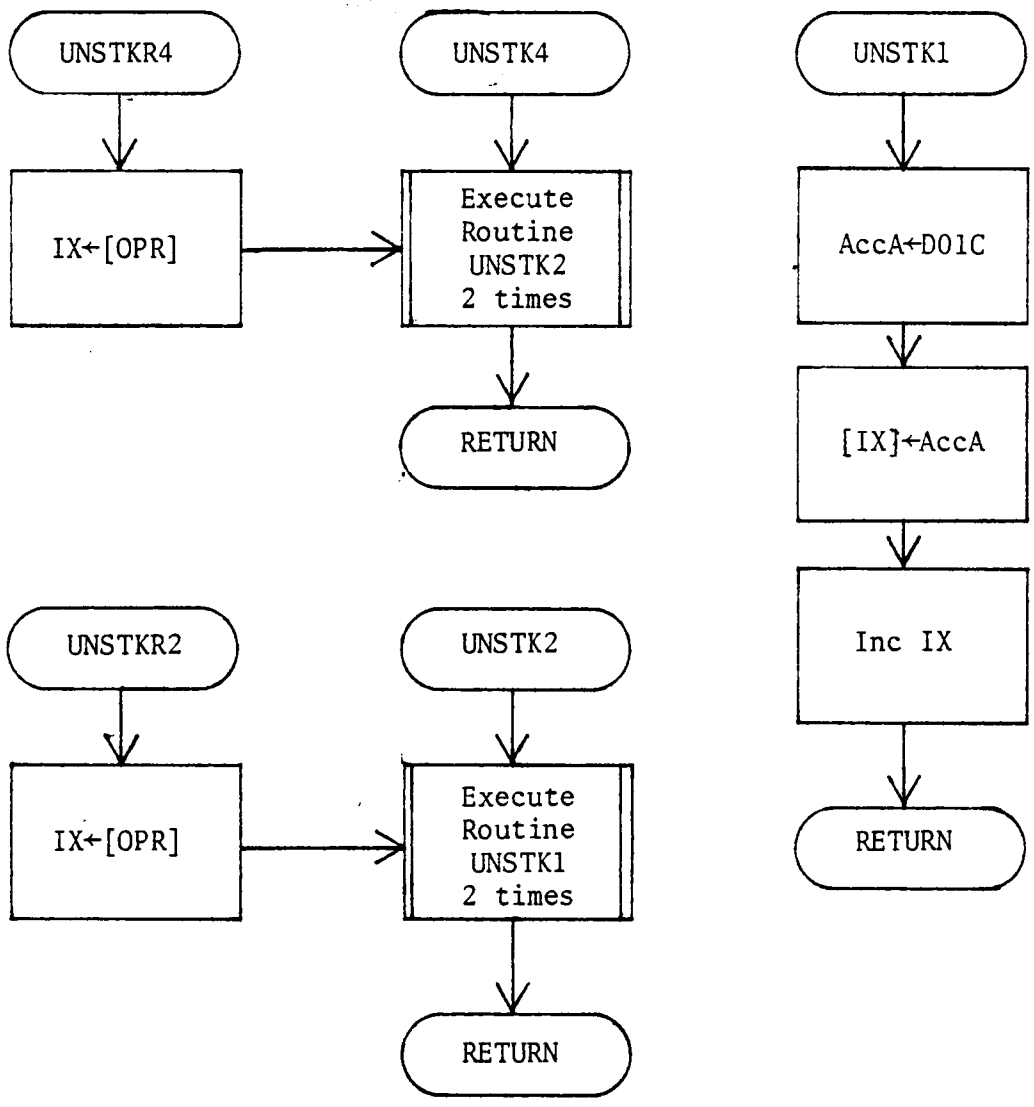


Figure B-2. Unstack routines.

```
ERRSTAT  DB   1           Location for APU status word
UXEQ     STA  A,D01Eh     Send cmmc to APU command register
USTAT    LDA  B,D01Eh     Read status from APU status
                            register
         BMI  USTAT       If busy (bit 7), read status again
         STA  B,ERRSTAT   Save status byte
         RTS              Return
UNOP     CLR  A           Load cmmc for no-operation
         BRA  UXEQ        Go to execution code
USQRT    LDA  A,01h       Load cmmc for square root
         BRA  UXEQ        Go to execution code
USIN     LDA  A,02h       Load cmmc for sine
         BRA  UXEQ        Go to execution code
UCOS     LDA  A,03h       Load cmmc for cosine
         BRA UXEQ         Go to execution code
UTAN     LDA  A,04h       Load cmmc for tangent
         BRA  UXEQ        Go to execution code
UASIN    LDA  A,05h       Load cmmc for arcsine
         BRA  UXEQ        Go to execution code
UACOS    LDA  A,06h       Load cmmc for arccosine
         BRA  UXEQ        Go to execution code
UATAN    LDA  A,07h       Load cmmc for arctangent
         BRA  UXEQ        Go to execution code
UEXP     LDA  A,0Ah       Load cmmc for exponentiation (e^x)
         BRA  UXEQ        Go to execution code
UFADD    LDA  A,10h       Load cmmc for floating add
         BRA UXEQ         Go to execution code
UFSUB    LDA  A,11h       Load cmmc for floating subtract
         BRA  UXEQ        Go to execution code
UFMUL    LDA  A,12h       Load cmmc for floating multiply
         BRA  UXEQ        Go to execution code
UFDIV    LDA  A,13h       Load cmmc for floating divide
         BRA  UXEQ        Go to execution code
UCHSF    LDA  A,15h       Load cmmc for change sign floating
         BRA  UXEQ        Go to execution code
UPTOF    LDA  A,17h       Load cmmc for push stack floating
         BRA  UXEQ        Go to execution code
UPOPF    LDA  A,18h       Load cmmc for pop stack floating
         BRA  UXEQ        Go to execution code
UXCHF    LDA  A,19h       Load cmmc for exchange operands
                            floating
         BRA  UXEQ        Go to execution code
UPUPI    LDA  A,1Ah       Load cmmc for push π
         BRA  UXEQ        Go to execution code
```

```
UFLTD   LDA  A,1Ch    Load cmmc for float double
        BRA  UXEQ     Go to execution code
UFLTS   LDA  A,1Dh    Load cmmc for float single
        BRA  UXEQ     Go to execution code
UFIXD   LDA  A,1Eh    Load cmmc for fix double
        BRA  UXEQ     Go to execution code
UFIXS   LDA  A,1Fh    Load cmmc for fix single
        BRA  UXEQ     Go to execution code
UDADD   LDA  A,2Ch    Load cmmc for double add
        BRA  UXEQ     Go to execution code
UDSUB   LDA  A,2Dh    Load cmmc for double subtract
        BRA  UXEQ     Go to execution code
UDMUL   LDA  A,2Eh    Load cmmc for double multiply
                        lower
        BRA  UXEQ     Go to execution code
UDDIV   LDA  A,2Fh    Load cmmc for double divide
        BRA  UXEQ     Go to execution code
UCHSD   LDA  A,34h    Load cmmc for change sign double
        BRA  UXEQ     Go to execution code
UDMUU   LDA  A,36h    Load cmmc for double multiply
                        upper
        BRA  UXEQ     Go to execution code
UPTOD   LDA  A,37h    Load cmmc for push stack double
        BRA  UXEQ     Go to execution code
UPOPD   LDA  A,38h    Load cmmc for pop stack double
        BRA  UXEQ     Go to execution code
UXCHD   LDA  A,39h    Load cmmc for exchange operands
                        double
        BRA  UXEQ     Go to execution code
USADD   LDA  A,6Ch    Load cmmc for single add
        BRA  UXEQ     Go to execution code
USSUB   LDA  A,6Dh    Load cmmc for single subtract
        BRA  UXEQ     Go to execution code
USMUL   LDA  A,6Eh    Load cmmc for single multiply
                        lower
        BRA  UXEQ     Go to execution code
USDIV   LDA  A,6Fh    Load cmmc for single divide
        BRA  UXEQ     Go to execution code
UCHSS   LDA  A,74h    Load cmmc for change sign single
        BRA  UXEQ     Go to execution code
USMUU   LDA  A,76h    Load cmmc for single multiply
                        upper
        BRA  UXEQ     Go to execution code
UPTOS   LDA  A,77h    Load cmmc for push stack single
        BRA  UXEQ     Go to execution code
UPOPS   LDA  A,78h    Load cmmc for pop stack single
        BRA  UXEQ     Go to execution code
UXCHS   LDA  A,79h    Load cmmc for exchange operands
                        single
        BRA  UXEQ     Go to execution code
```

```
ULOG    LDA  A,08h       Load cmmc for common logarithm
        BRA  ULJMP       Go to log test
ULN     LDA  A,09h       Load cmmc for natural logarithm
ULJMP   BSR  UTSTZ       Test for APU TOS = 0
UPL     STA  B,ERRSTAT   Save status
        BEQ  UXEQ        If APU TOS ≠ 0, go to execution
                           code
        RTS              Return
UPWR    BSR  UXCHF       Exchange APU TOS and NOS
        BSR  UTSTZ       Test for APU TOS = 0
        PSH  B           Save APU TOS status
        BSR  UXCHF       Restore APU TOS and NOS operands
        PUL  B           Recall APU status
        LDA  A,0Bh       Load cmmc for power (y^x)
        BRA  UPL         Go to status check
UTSTZ   LDA  B,D01Eh     Read APU TOS status
        AND  B,20h       Check for APU TOS = 0
        BEQ  UTZ1        If APU TOS ≠ 0, return
        LDA  B,28h       Set error status
UTZ1    RTS              Return
```

(see Fig. B-3)

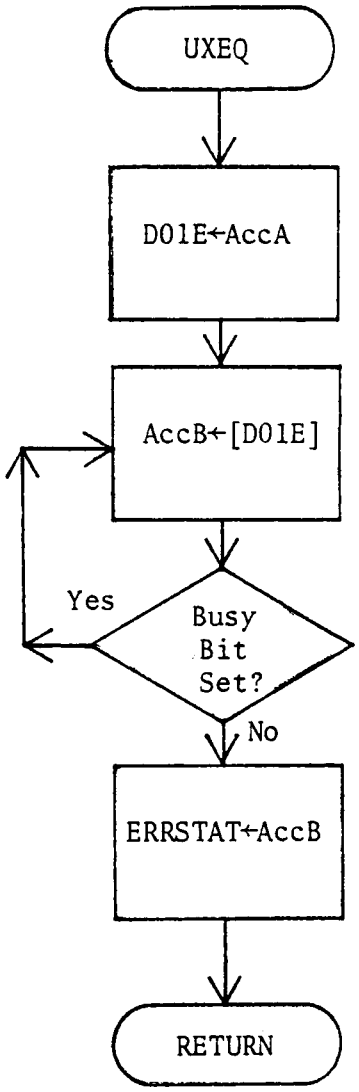


Figure B-3. Utility routines.

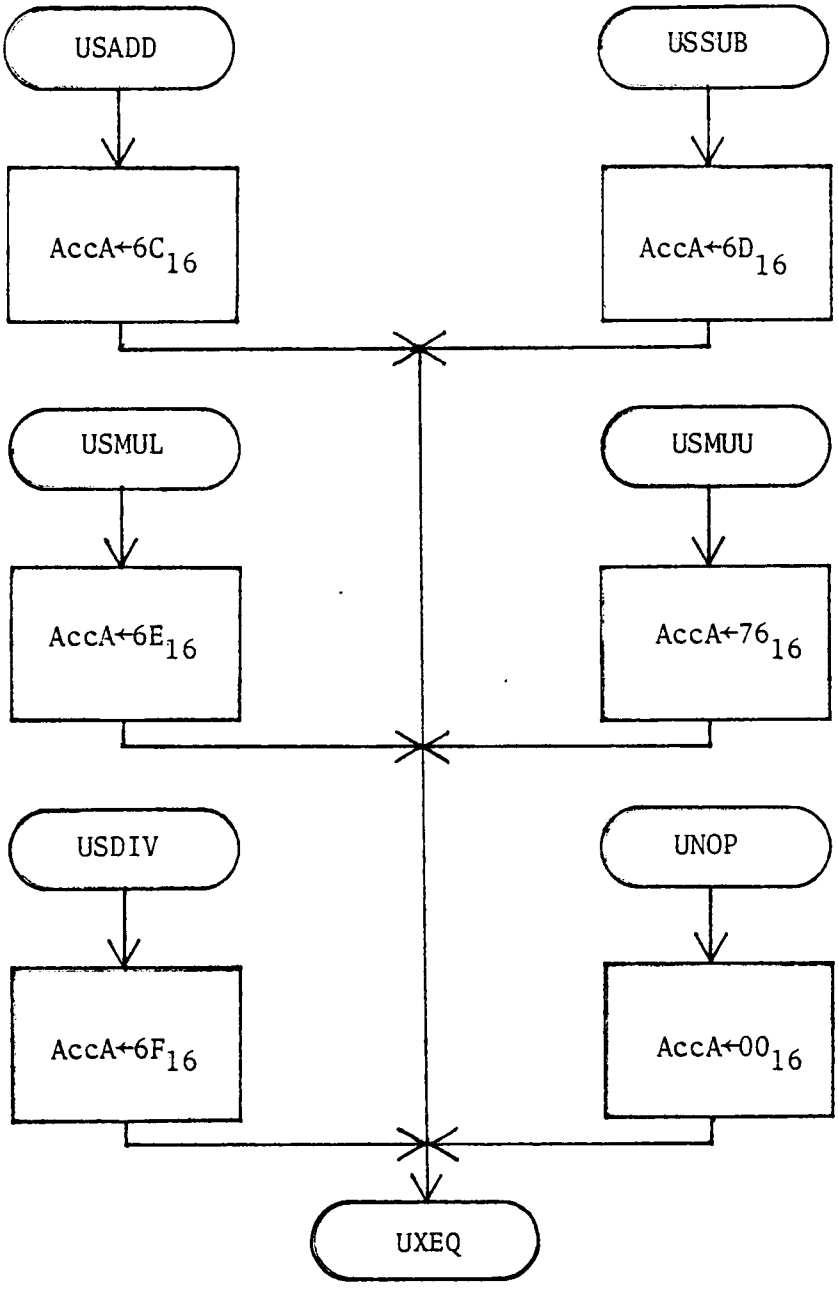


Figure B-3 (continued)

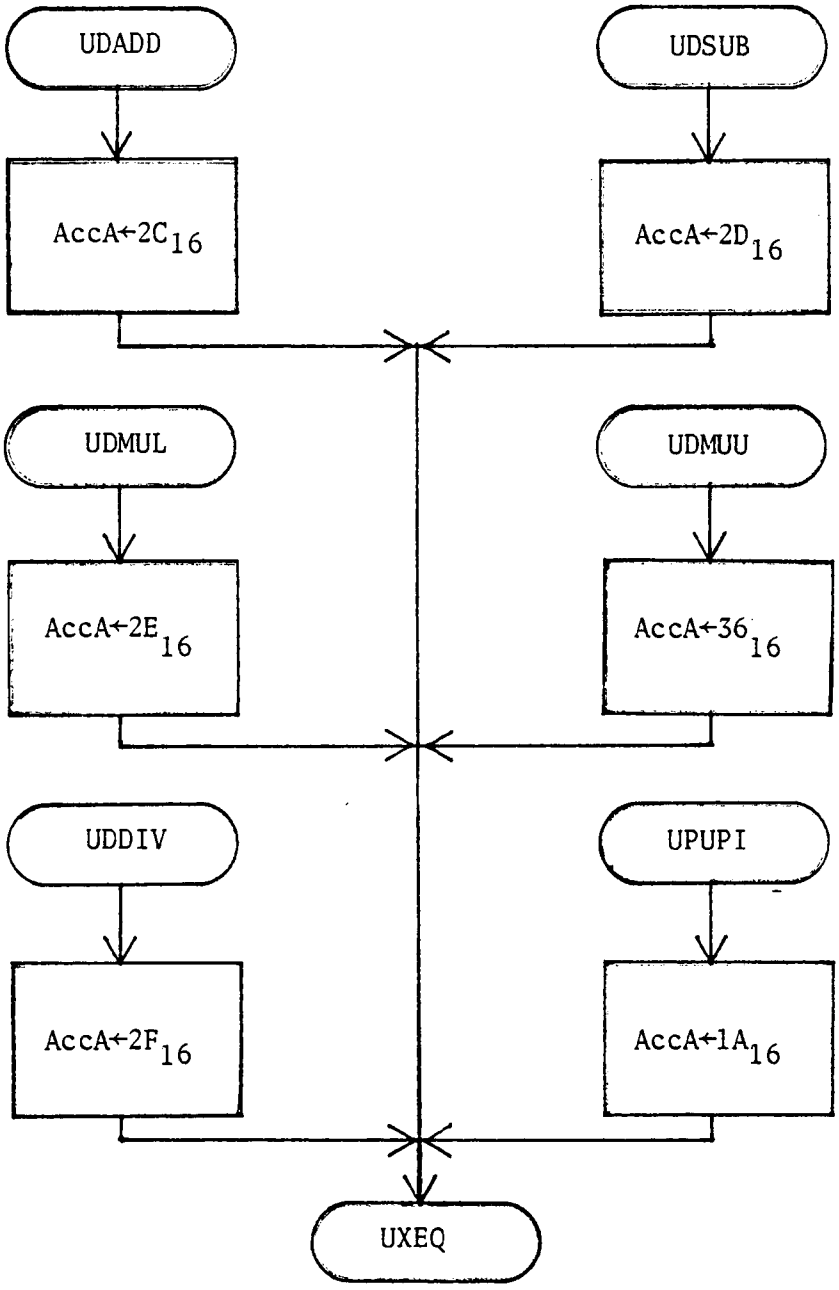


Figure B-3 (continued)

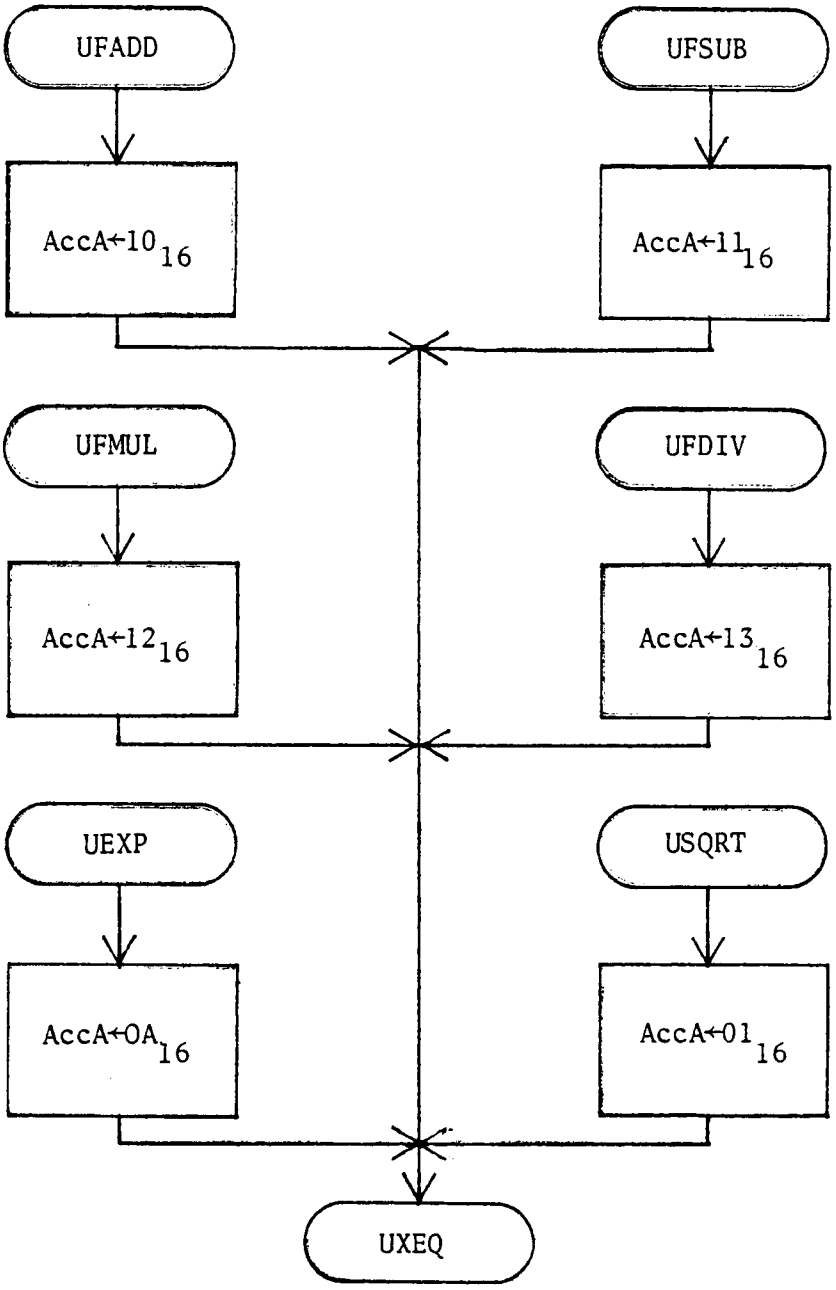


Figure B-3 (continued)

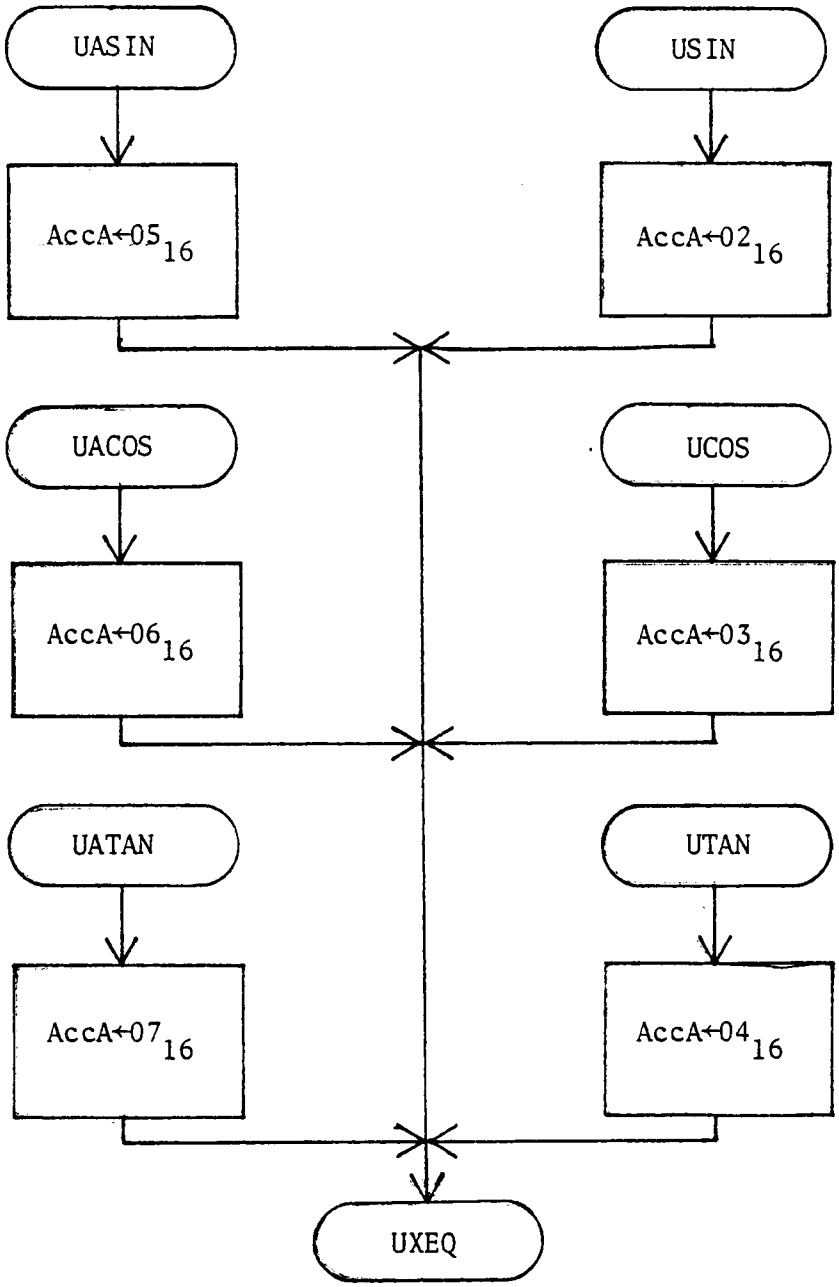


Figure B-3 (continued)

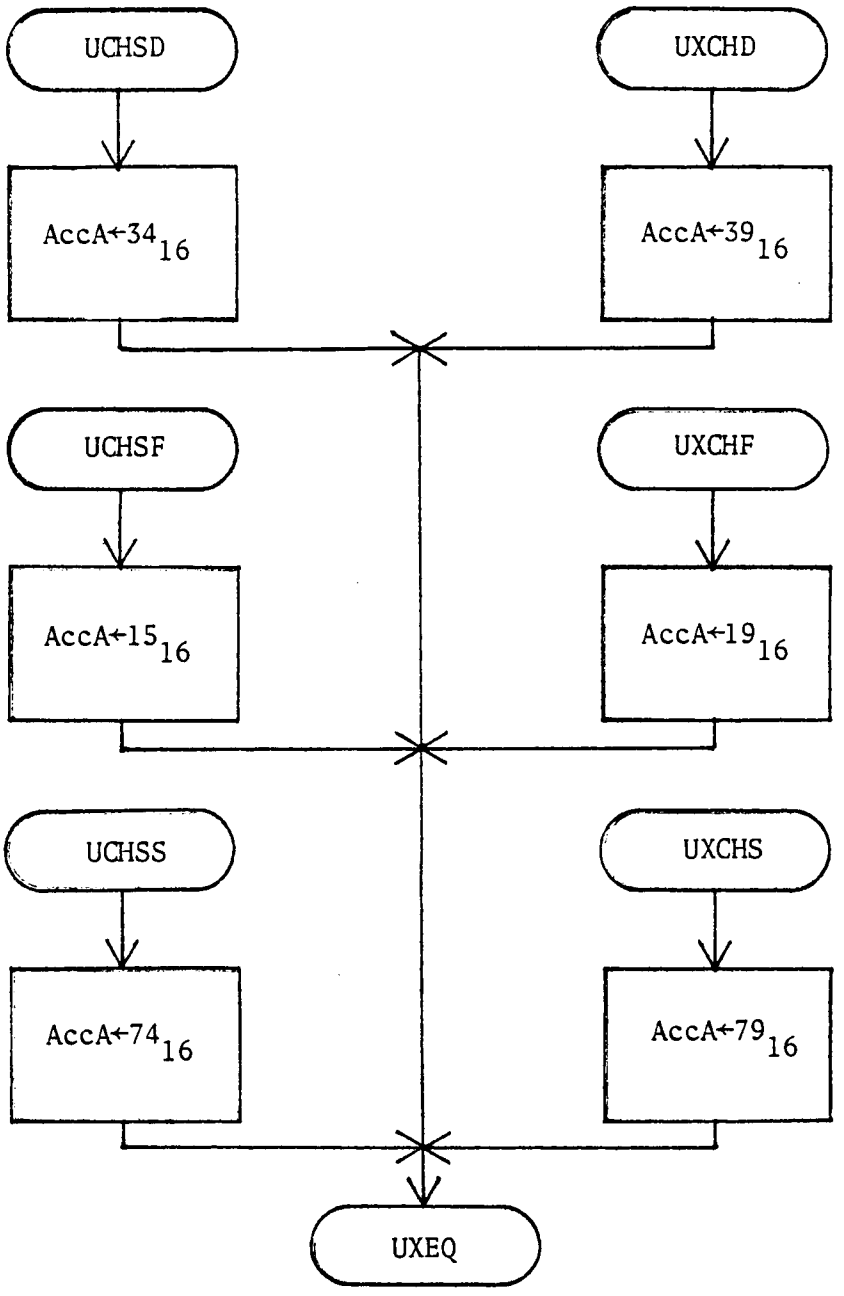


Figure B-3 (continued)

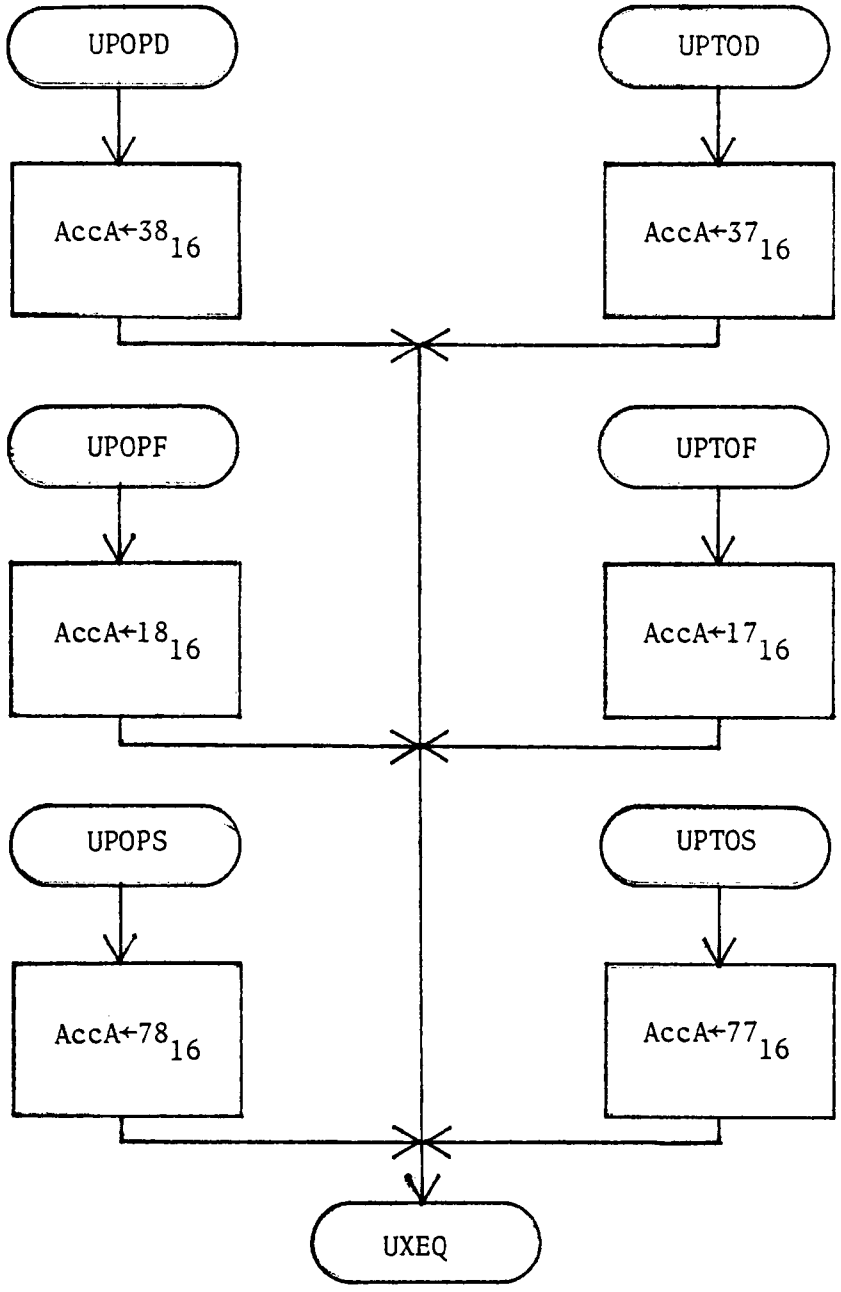


Figure B-3 (continued)

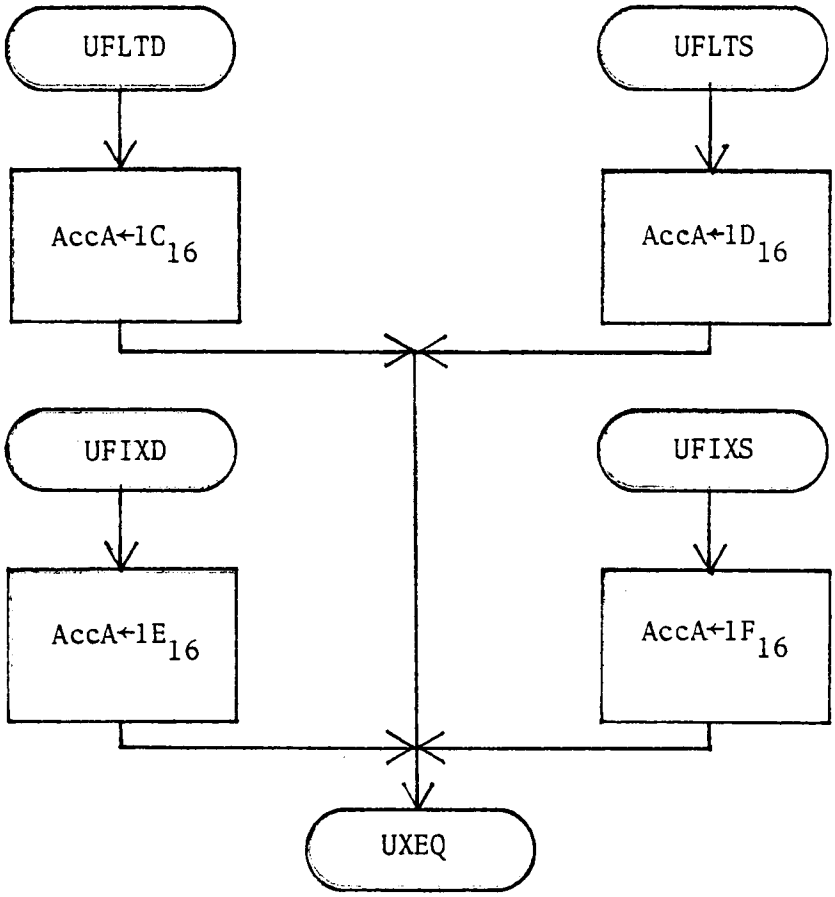


Figure B-3 (continued)

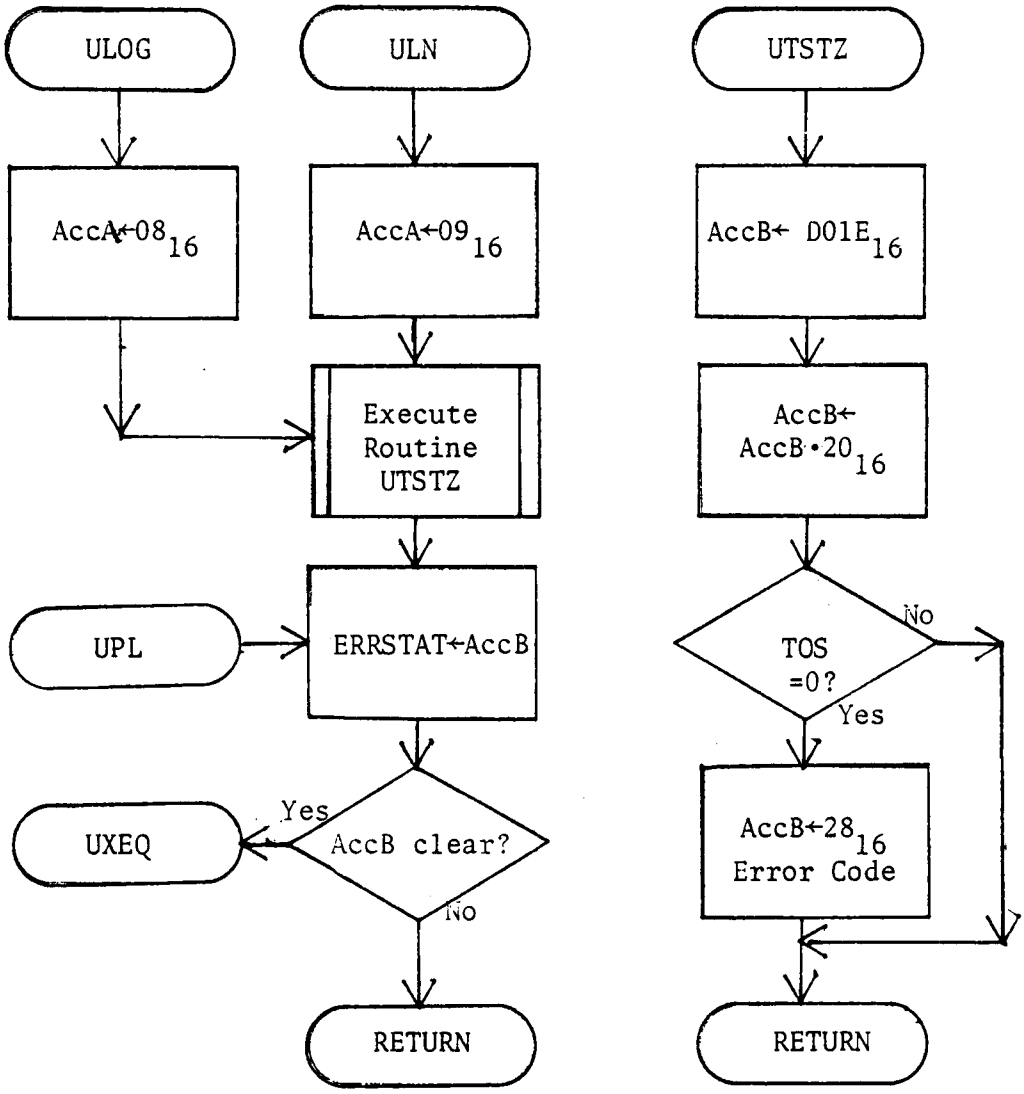


Figure B-3 (continued)

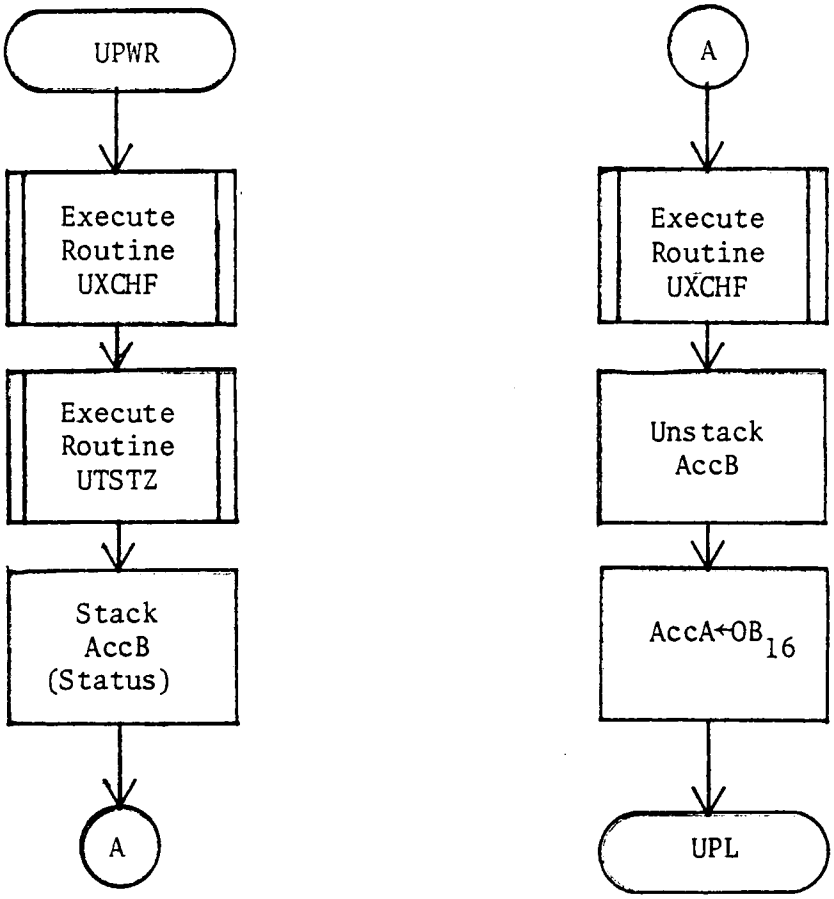


Figure B-3 (continued)

```
DXEQ2    JSR  STKB4      Stack 4-byte APU NOS
DXEQ1    JSR  STKA4      Stack 4-byte APU TOS
DXEQ0    PSH  A          Save cmmc in stack
         JSR  UXEQ       Execute operation ordered by AccA
         JSR  UNSTKR4    Unstack 4-byte result at APU TOS
DSX      PUL  A          Recall cmmc from stack
         AND  B,1Eh      Isolate error field
RETURN   RTS             Return
SXEQ2    JSR  STKB2      Stack 2-byte APU NOS
SXEQ1    JSR  STKA2      Stack 2-byte APU TOS
SXEQ0    PSH  A          Save cmmc in stack
         JSR  UXEQ       Execute operation ordered by AccA
         JSR  UNSTKR2    Unstack 2-byte result at APU TOS
         BRA  DSX        Branch to same ending as DXEQ group
```

(see Fig. B-4)

```
CHSF     LDA  A,15h      Load cmmc for change sign floating
         BRA  DXEQ1      Go to execution code
FLTD     LDA  A,1Ch      Load cmmc for float double
         BRA  DXEQ1      Go to execution code
COS      LDA  A,03h      Load cmmc for cosine
         BRA  DXEQ1      Go to execution code
SIN      LDA  A,02h      Load cmmc for sine
         BRA  DXEQ1      Go to execution code
ATAN     LDA  A,07h      Load cmmc for arctangent
         BRA  DXEQ1      Go to execution code
PUPI     LDA  A,1Ah      Load cmmc for push π
         BRA  DXEQ0      Go to execution code
FLTS     JSR  STKA2      Stack 2-byte APU TOS
         LDA  A,1Dh      Load cmmc for float single
         BRA  DXEQ0      Go to execution code
```

(see Fig. B-5)

```
FIXS     JSR  STKA4      Stack 4-byte APU TOS
         LDA  A,1Fh      Load cmmc for fix single
         JSR  SXEQ0      Execute
         BEQ  RETURN     If no error, return
         JSR  UNSTK2     Retrieve lower two bytes of
                           unchanged operand
         JMP  INCxx01    Flag overflow
CHSD     LDA  A,34h      Load cmmc for change sign double
         BRA  DPA        Go to execution code
FIXD     LDA  A,1Eh      Load cmmc for fix double
DPA      JSR  DXEQ1      Execute
         BEQ  RETURN     If no error, return
         JMP  INCxx01    Flag overflow
```

(see Fig. B-6)

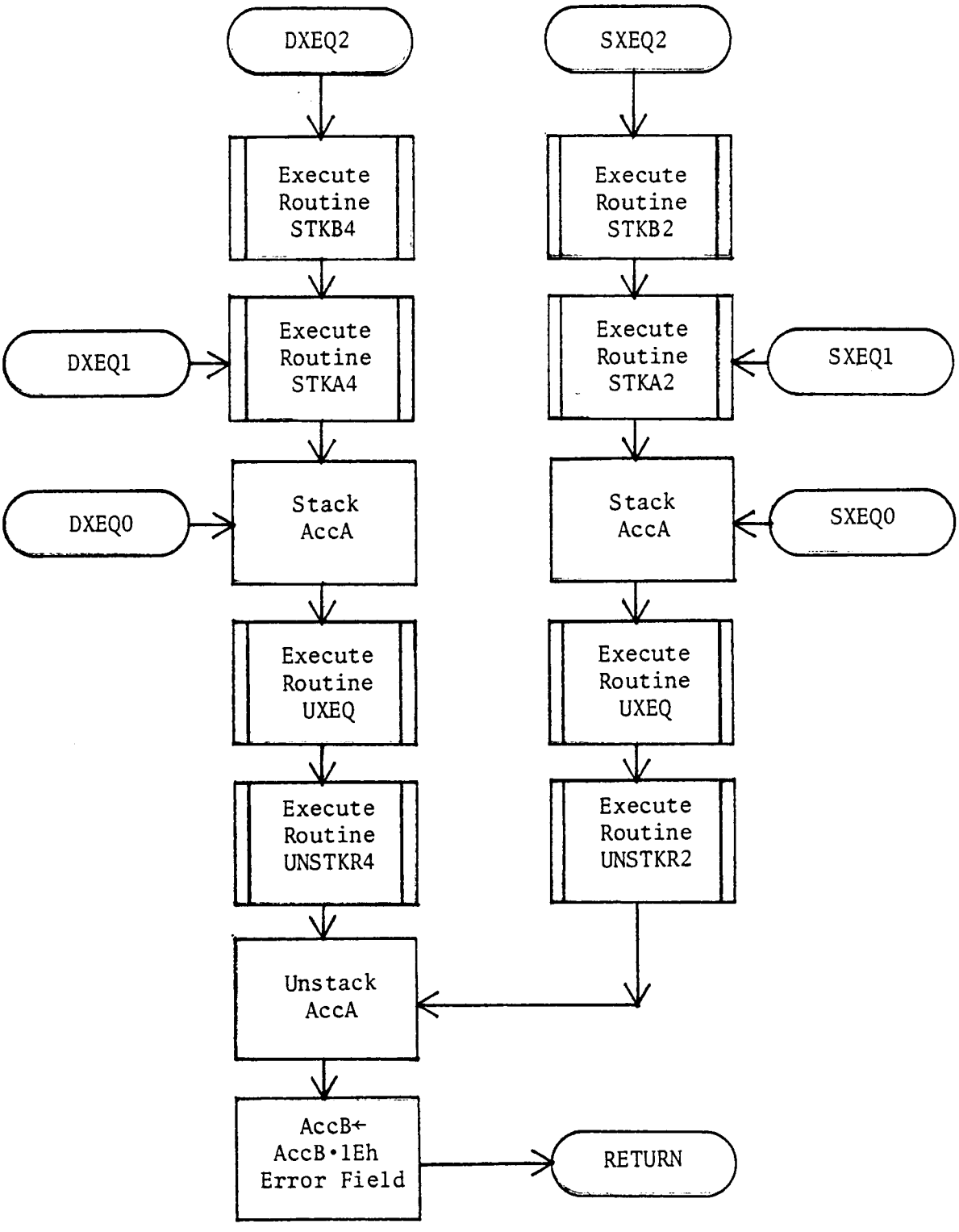


Figure B-4. Complete-operation execution routines.

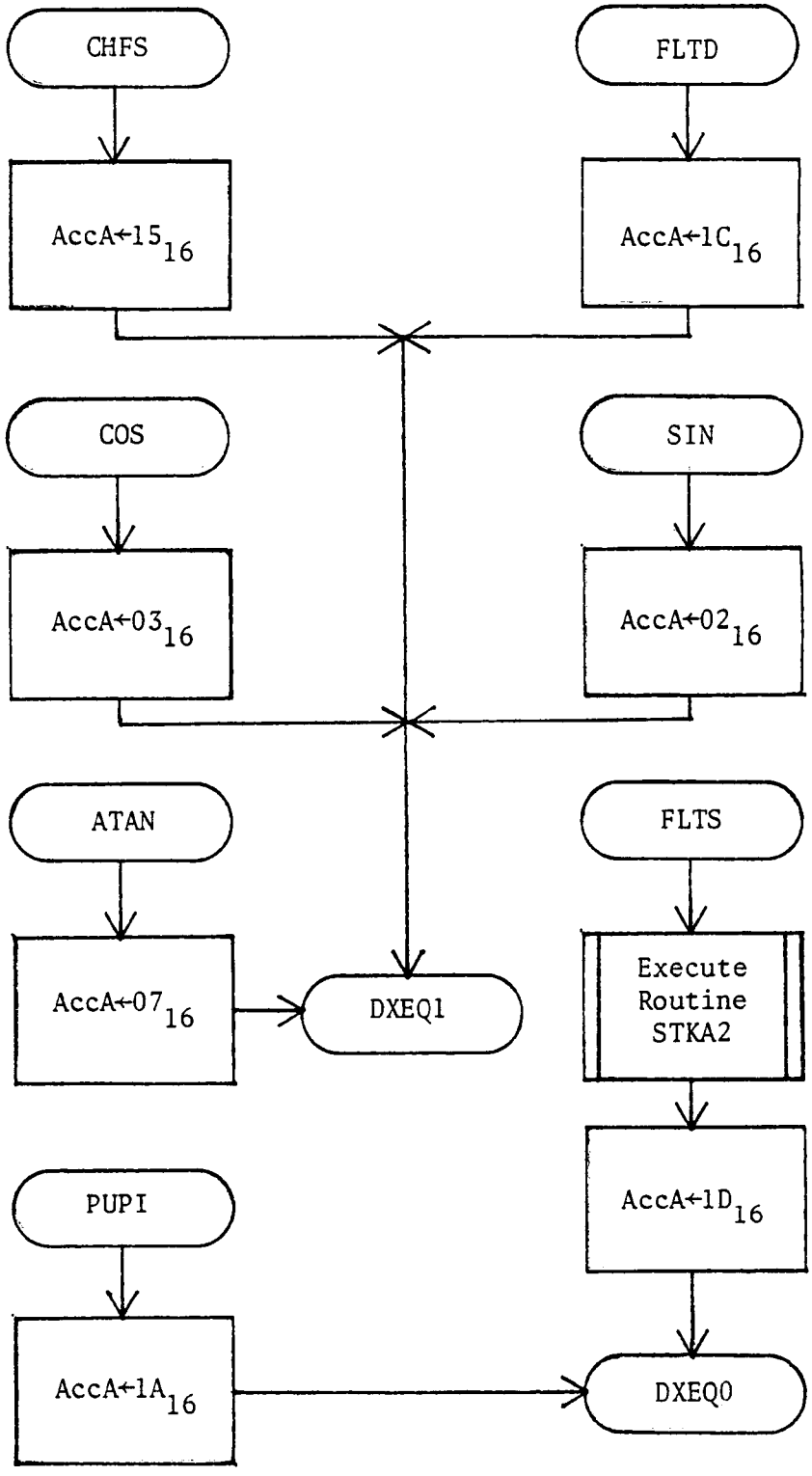


Figure B-5. Complete-operation routines; floating point change sign, cosine, sine, inverse tangent, push π, float single, and double fixed point.

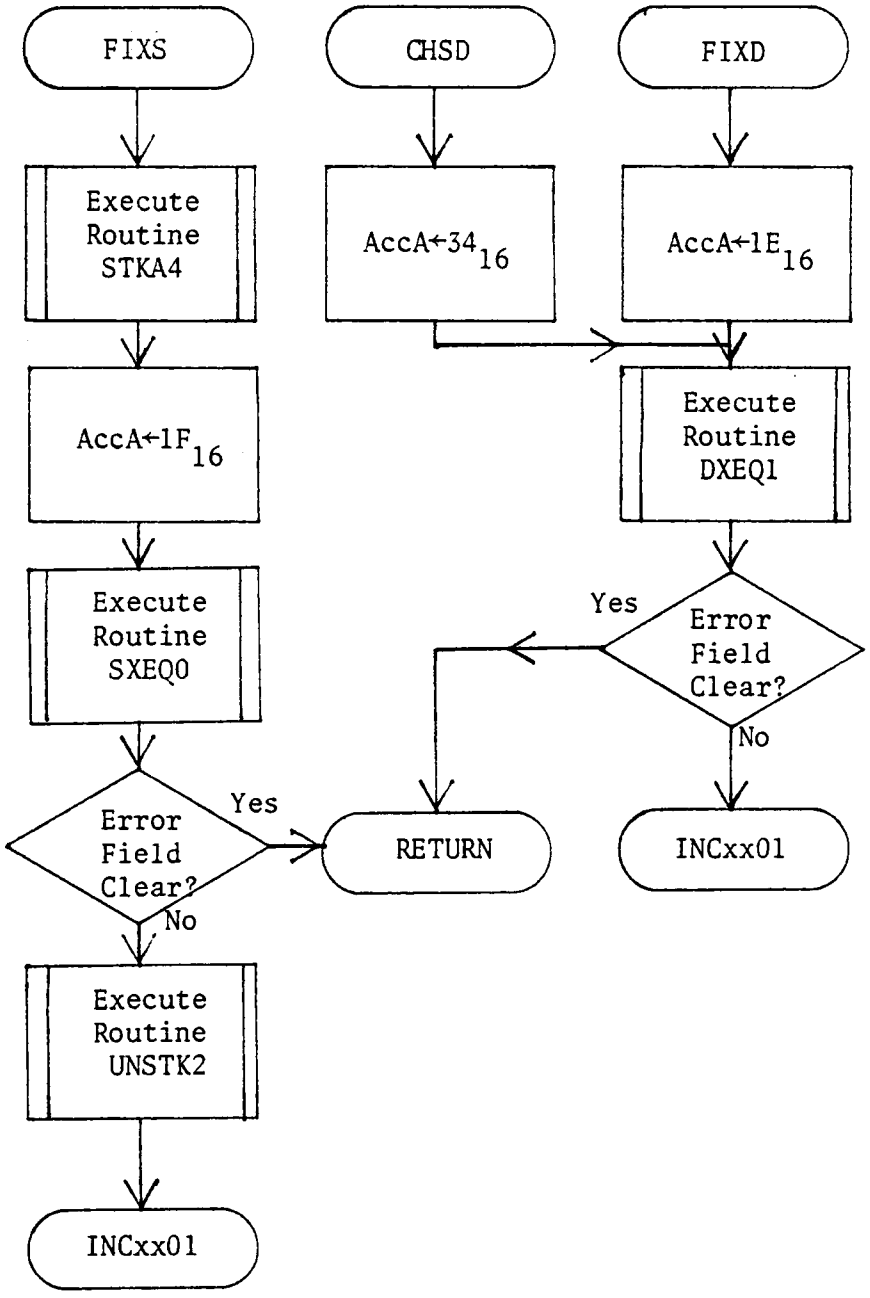


Figure B-6. Complete-operation routines; change sign double and fix floating point to single and double.

```
FLT1    DC   4          '01 80 00 00' floating 1
EXP     LDA  A,0Ah      Load cmmc for exponentiation (e^X)
        JSR  DXEQ1      Execute
        BEQ  ERTS       If no error, return
        LDX  OPA        Load IX with addr of APU TOS
        JSR  INC1100    Flag error (argument out of range)
        TST  X          Find sign of operand A
        BPL  FLTMAX+    If positive, set R to 3F FF FF FF16
        BRA  R0         If negative, set R to 0
ERTS    RTS
```

(See Fig. B-7)

```
TAN     LDA  A,04h      Load cmmc for tangent
        JSR  DXEQ1      Execute
        BEQ  ERTS       If no error, return
        JSR  INC1000    Flag error zero-divide
        JMP  FLTMAXR    Set R to floating maximum
```

(See Fig. B-8)

```
ASIN    LDA  A,05h      Load cmmc for arcsine
        BRA  ARC        Go to execution code
ACOS    LDA  A,06h      Load cmmc for arccosine
ARC     JSR  DXEQ1      Execute
        BEQ  ERTS       If no error, return
        JSR  INC1100    Flag error (argument out of range)
        LDX  #FLT1      Set APU TOS to 1 × sign of (A)
        JSR  STK4
        LDX  OPA
        TST  X
        BPL  ARCGO
        PSH  A
        JSR  UCHSF
        PUL  A
ARCGO   JMP  DXEQ0      Execute command again
```

(see Fig. B-9)

```
SQRT    LDA  A,01h      Load cmmc for square root
        BRA  LSQ        Go to execution code
LOG     LDA  A,08h      Load cmmc for common logarithm
        BRA  LJMP       Go to execution code
LN      LDA  A,09h      Load cmmc for natural logarithm
LJMP    JSR  FTSTA      Test for operand A = 0
LSQ     JSR  DXEQ1      Execute
        BEQ  ERTS       If no error, return
        PSH  A          Save cmmc in stack
        JSR  STKA4      Stack 4-byte operand A
        JSR  UCHSF      Change sign of APU TOS
        PUL  A          Recall cmmc
        JSR  INC0100    Flag error (negative input)
        JMP  DXEQ0      Go to execution code
```

(see Fig. B-10)

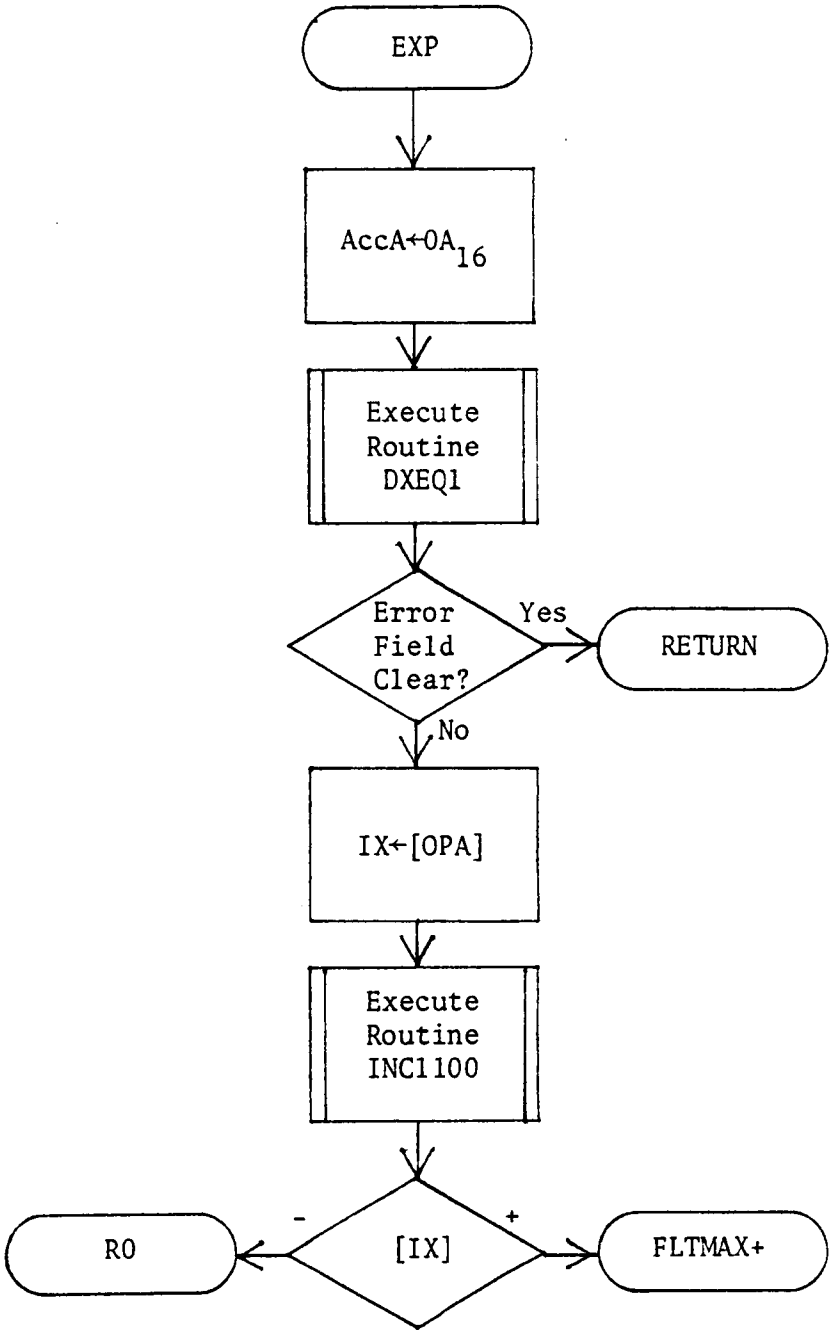


Figure B-7. Complete-operation routines; exponentiation (e^X).

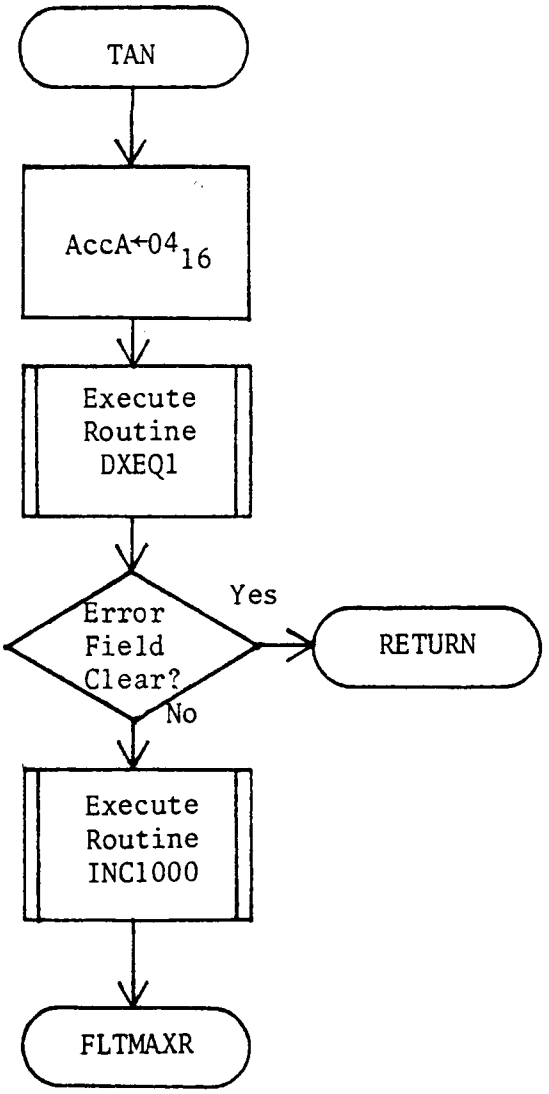


Figure B-8. Complete-operation routines; tangent.

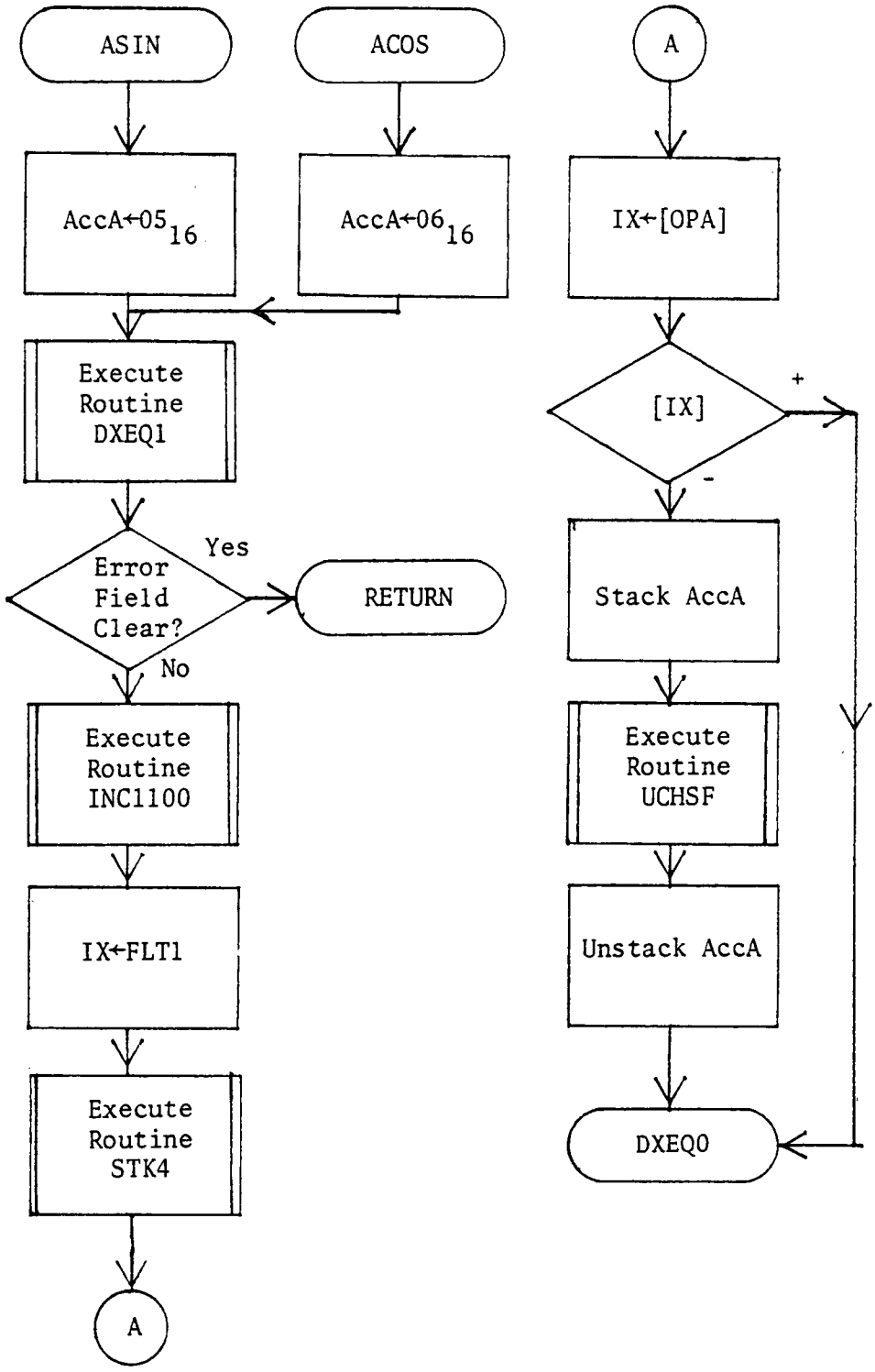


Figure B-9. Complete-operation routines; inverse sine and inverse cosine.

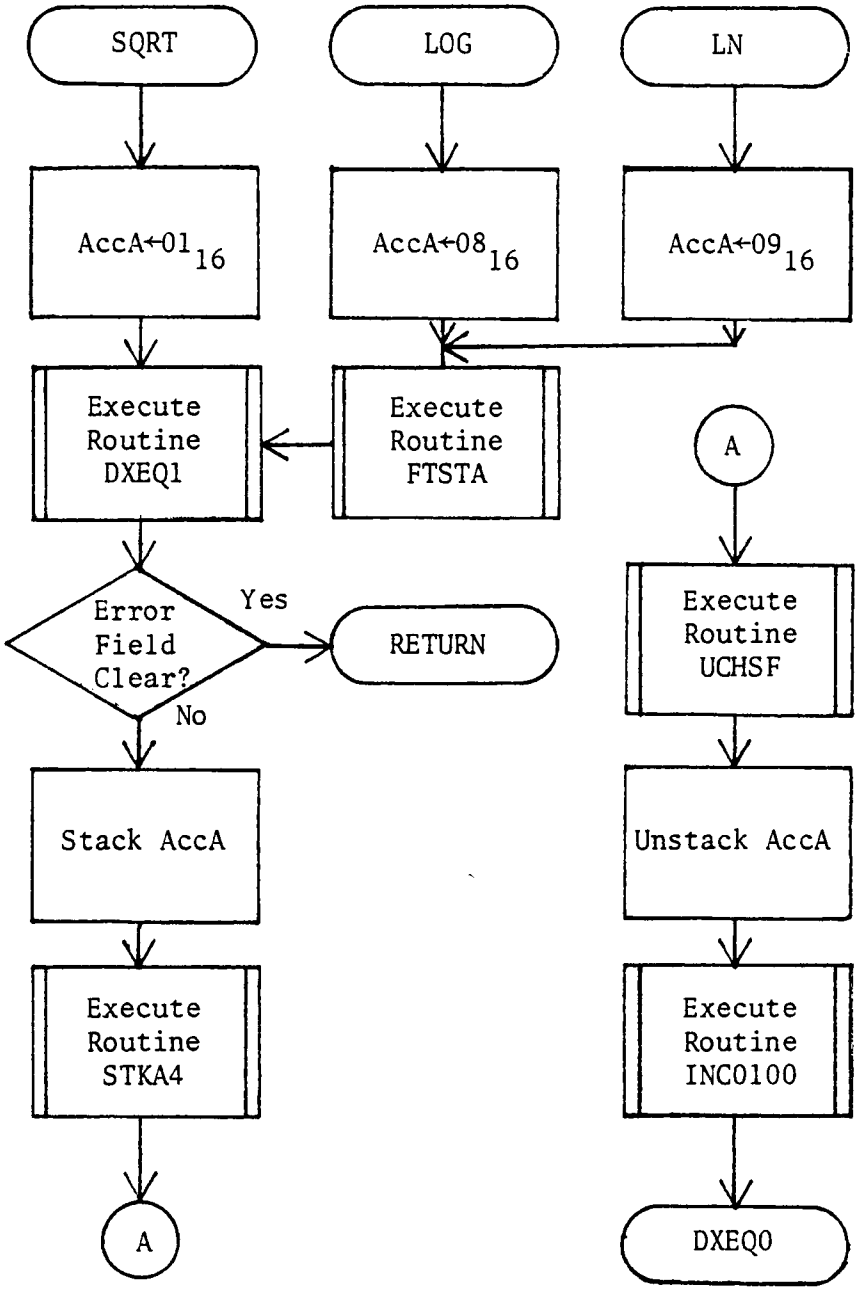


Figure B-10. Complete-operation routines; square root, common logarithm, and natural logarithm.

```
FTSTB    LDX  OPB          Load IX with addr of APU NOS
         BRA  FTST         Go to test code
FTSTA    LDX  OPA          Load IX with addr of APU TOS
FTST     TST  X+1          Is value 0?
         BNE  FTSTR        If not 0, return
         LDA  B,28h        Load status of zero bit set,
                             error = 0100
         STA  B,ERRSTAT    Save status
         JSR  R0           Set result to 0
         INS               Adjust stack to return up two
         INS                 levels, instead of one
         JMP  INC0100      Flag error (zero argument)
FTSTR    RTS
```

(see Fig. B-11)

```
PWR      JSR  FTSTB        Test for APU NOS value = 0
         LDA  A,0Bh        Load cmmc for power (y^x)
         JSR  DXEQ2        Execute
PWR0     BEQ  FTSTR        If no error, return
PWR1     CMP  B,02h        If not overflow, check another
         BNE  PWR2
         JSR  INCxx01      Flag overflow
         BRA  PWR41        Go to same fixup as for error 1100
PWR2     CMP  B,08h        If not negative argument, check
                             another
         BNE  PWR3
         JSR  INC0100      Flag error (negative argument)
         JSR  STKB4        Stack APU NOS
         JSR  UCHSF        Change sign of APU NOS
         LDA  A,0Bh        Load cmmc for power
         JSR  DXEQ1        Execute
         BRA  PWR0         Go to error test
PWR3     CMP  B,04h        If not underflow, check another
         BNE  PWR4
         JSR  INCxx10      Flag underflow
PWR31    LDX  #FLT1        Set result R to 1
         JSR  STK4
         JMP  UNSTKR4
PWR4     JSR  INC1100      Flag error (argument out of range)
PWR41    JSR  STKB4        Stack APU NOS
         JSR  ULN          Take natural logarithm of APU NOS
         AND  B,40h        Find sign of ln(NOS)
         ASL  B            Set AccB to sign of ln(NOS)
         LDX  OPA          Load IX with addr of operand A
         EOR  B,X          Find sign of A × lnB
         BMI  PWR31        If negative, set R to 1
         JMP  FLTMAX+      If positive, set R to 3F FF FF FF16
```

(see Fig. B-12)

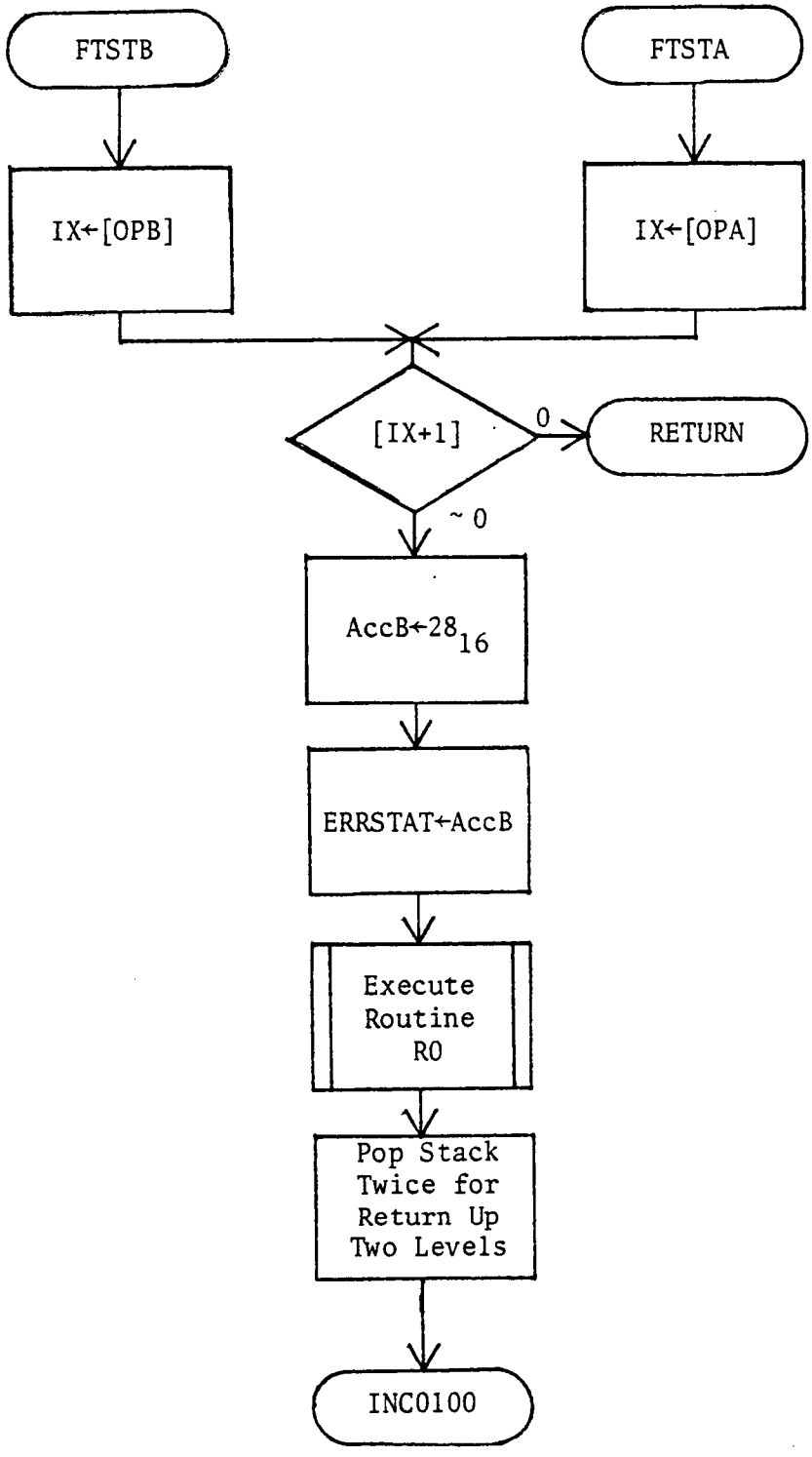


Figure B-11. Complete-operation routines; test operand of logarithm.

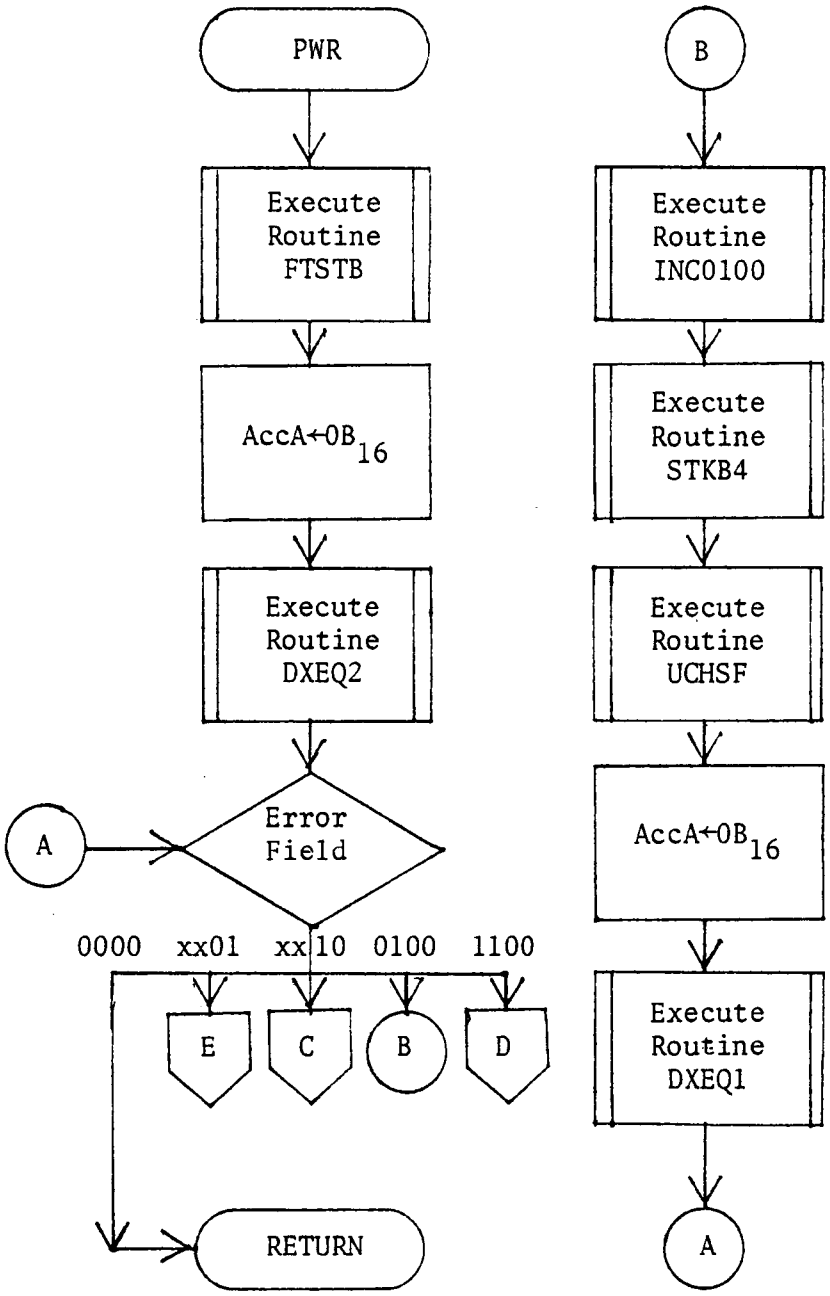


Figure B-12. Complete-operation routines; power (y^x).

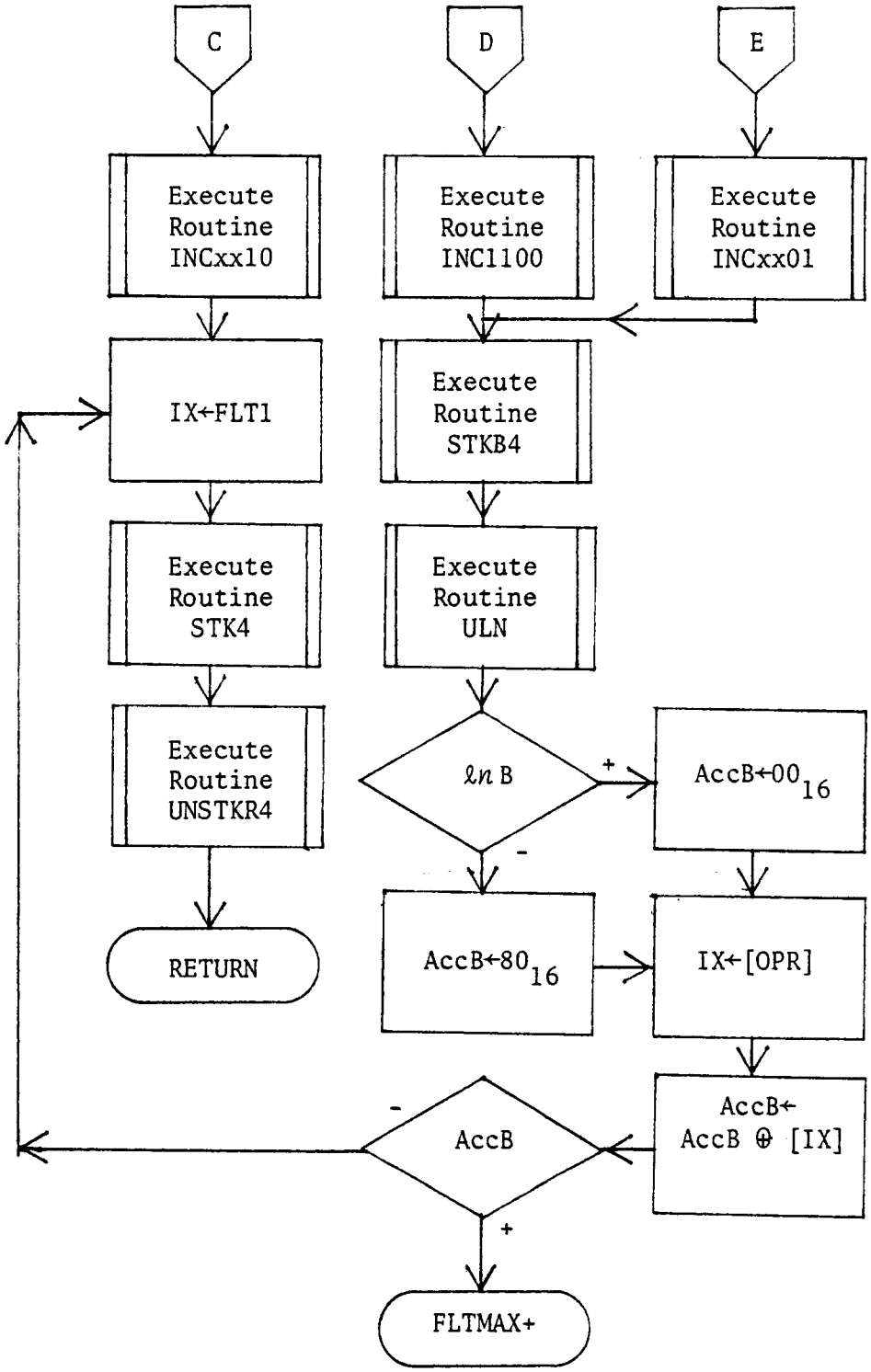


Figure B-12 (continued)

```
FADD    LDA  A,10h     Load cmmc for floating add
        BRA  FLTATH    Go to execution code
FSUB    LDA  A,11h     Load cmmc for floating subtract
        BRA  FLTATH    Go to execution code
FMUL    LDA  A,12h     Load cmmc for floating multiply
FLTATH  JSR  DXEQ2     Execute
        BEQ  FLTRTS    If no error, return
        CMP  B,02h     If not overflow, try underflow
        BNE  FLTUND
        JSR  FLTMAXR   On overflow set R to floating
                         maximum
        JMP  INCxx01   Flag overflow
FLTUND  JSR  R0        Set R to 0
        JMP  INCxx10   Flag underflow
FLTRTS  RTS
```

(see Fig. B-13)

```
DADD    LDA  A,2Ch     Load cmmc for double add
        BRA  DPATH     Go to execution code
DSUB    LDA  A,2Dh     Load cmmc for double subtract
        BRA  DPATH     Go to execution code
DMUL    LDA  A,2Eh     Load cmmc for double multiply
                         lower
        BRA  DPATH     Go to execution code
DMUU    LDA  A,36h     Load cmmc for double multiply
                         upper
DPATH   JSR  DXEQ2     Execute
        BEQ  FLTRTS    If no error, return
        JSR  INCxx01   Flag overflow
        CMP  A,36h     If not DMUU, return; else fixup
        BNE  FLTRTS
        JMP  DPMAXN    Set R to 80 00 00 00(16)
```

(see Fig. B-14)

```
DDIV    LDA  A,2Fh     Load cmmc for double divide
        BRA  DIVIDE    Go to execution code
FDIV    LDA  A,13h     Load cmmc for floating divide
DIVIDE  JSR  DXEQ2     Execute
        BEQ  FLTRTS    If no error, return
        CMP  B,04h     If not underflow, check another
        BNE  DIV1
        JSR  INCxx10   Flag underflow
        JMP  R0        Set R to 0
DIV1    CMP  B,10h     If not zero divide, check another
        BNE  DIV2
        JSR  INC1000   Flag zero divide
        BRA  DIV3      Same fixup as for overflow
DIV2    JSR  INCxx01   Flag overflow
DIV3    CMP  A,2Fh
        BEQ  DPMAXR    If DDIV, set R to double maximum
        BRA  FLTMAXR   If FDIV, set R to floating maximum
```

(see Fig. B-15)

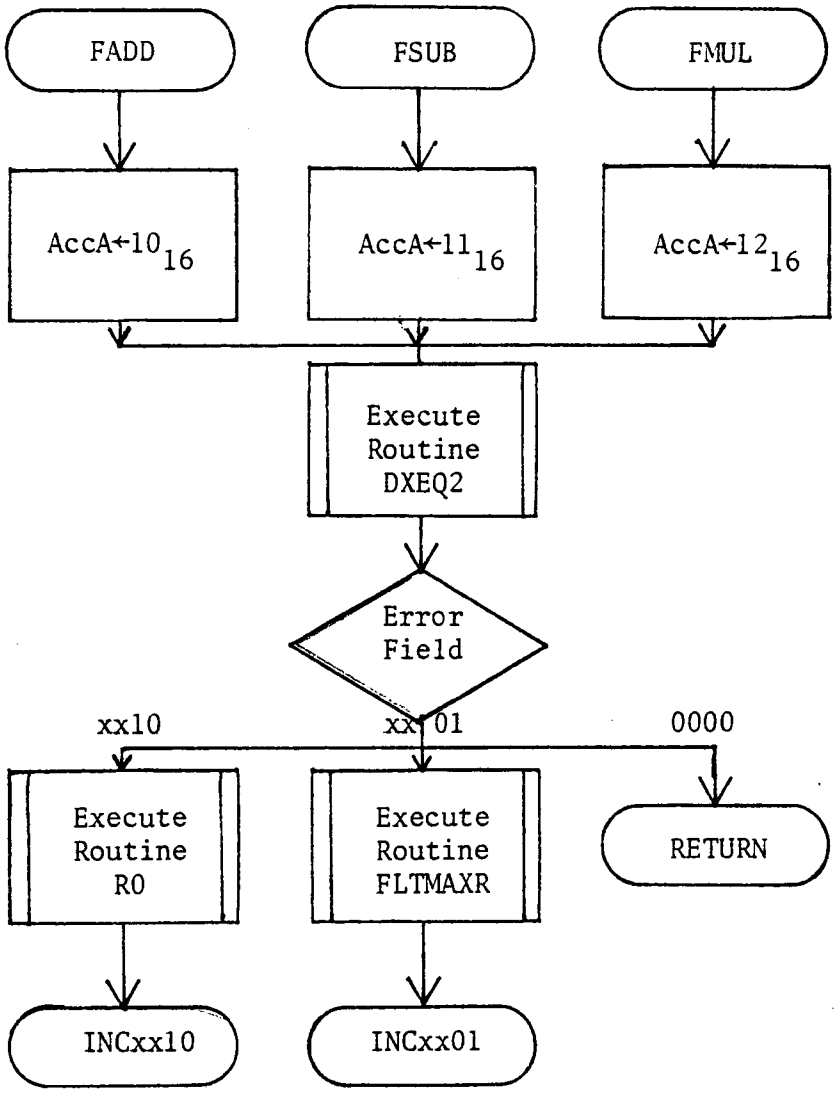


Figure B-13. Complete-operation routines; floating add, subtract, and multiply.

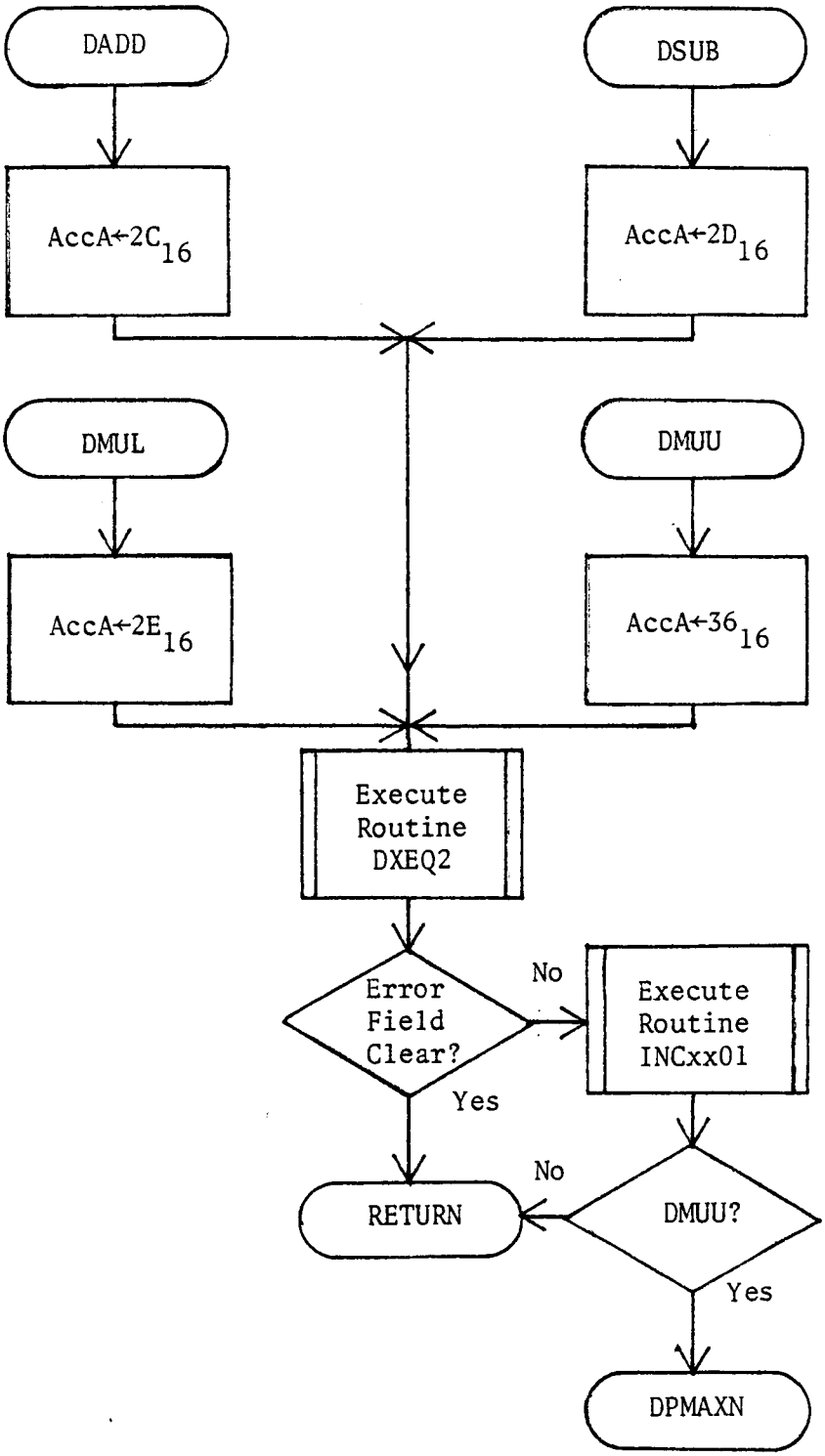


Figure B-14. Complete-operation routines; double add, subtract, and multiply.

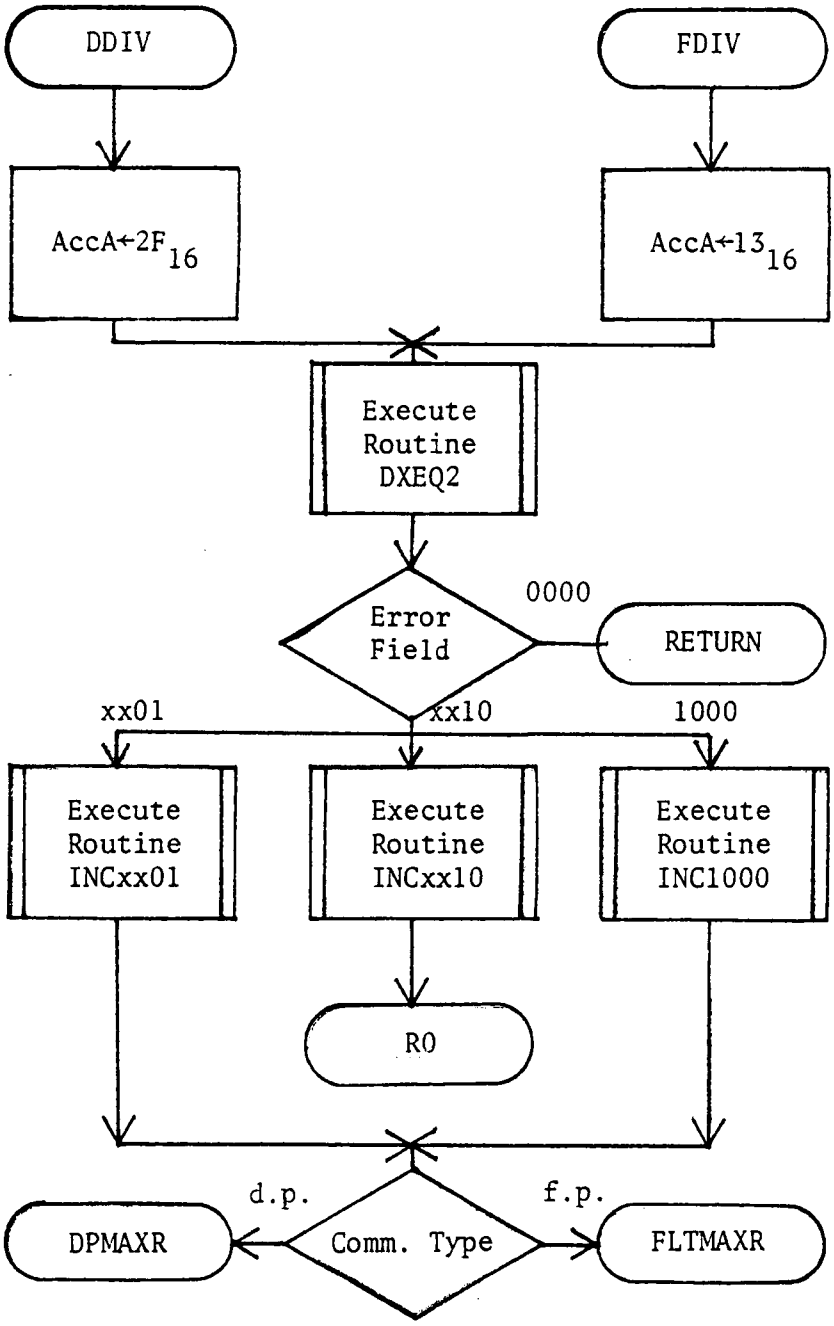


Figure B-15. Complete-operation routines; double and floating divide.

```
CHSS    LDA  A,74h      Load cmmc for change sign single
        JSR  SXEQ1      Execute
        BNE  INCxx01    If overflow, flag
SPRTS   RTS             Return
SADD    LDA  A,6Ch      Load cmmc for single add
        BRA  SPATH      Go to execution code
SSUB    LDA  A,6Dh      Load cmmc for single subtract
        BRA  SPATH      Go to execution code
SMUL    LDA  A,6Eh      Load cmmc for single multiply
                          lower
        BRA  SPATH      Go to execution code
SMUU    LDA  A,76h      Load cmmc for single multiply
                          upper
SPATH   JSR  SXEQ2      Execute
        BEQ  SPRTS      If no error, return
        JMP  INCxx01    Flag overflow
```

(see Fig. B-16)

```
SDIV    LDA  A,6Fh      Load cmmc for single divide
        JSR  SXEQ2      Execute
        BEQ  SPRTS      If no error, return
        LDX  OPR        Load IX with addr of result R
        CMP  B,10h      If not zero divide, check another
        BNE  SD1
        JSR  INC1000    Flag zero divide
        TST  X          Find sign of R
        BMI  SD2        If R negative, set to 80 00(16)
        LDA  B,7Fh      If R positive, set to 7F FF(16)
        LDA  A,FFh
        BRA  SD3
SD1     JSR  INCxx01    Flag overflow
SD2     LDA  B,80h
        CLR  A
SD3     STA  B,X
        STA  A,X+1
        RTS
```

(see Fig. B-17)

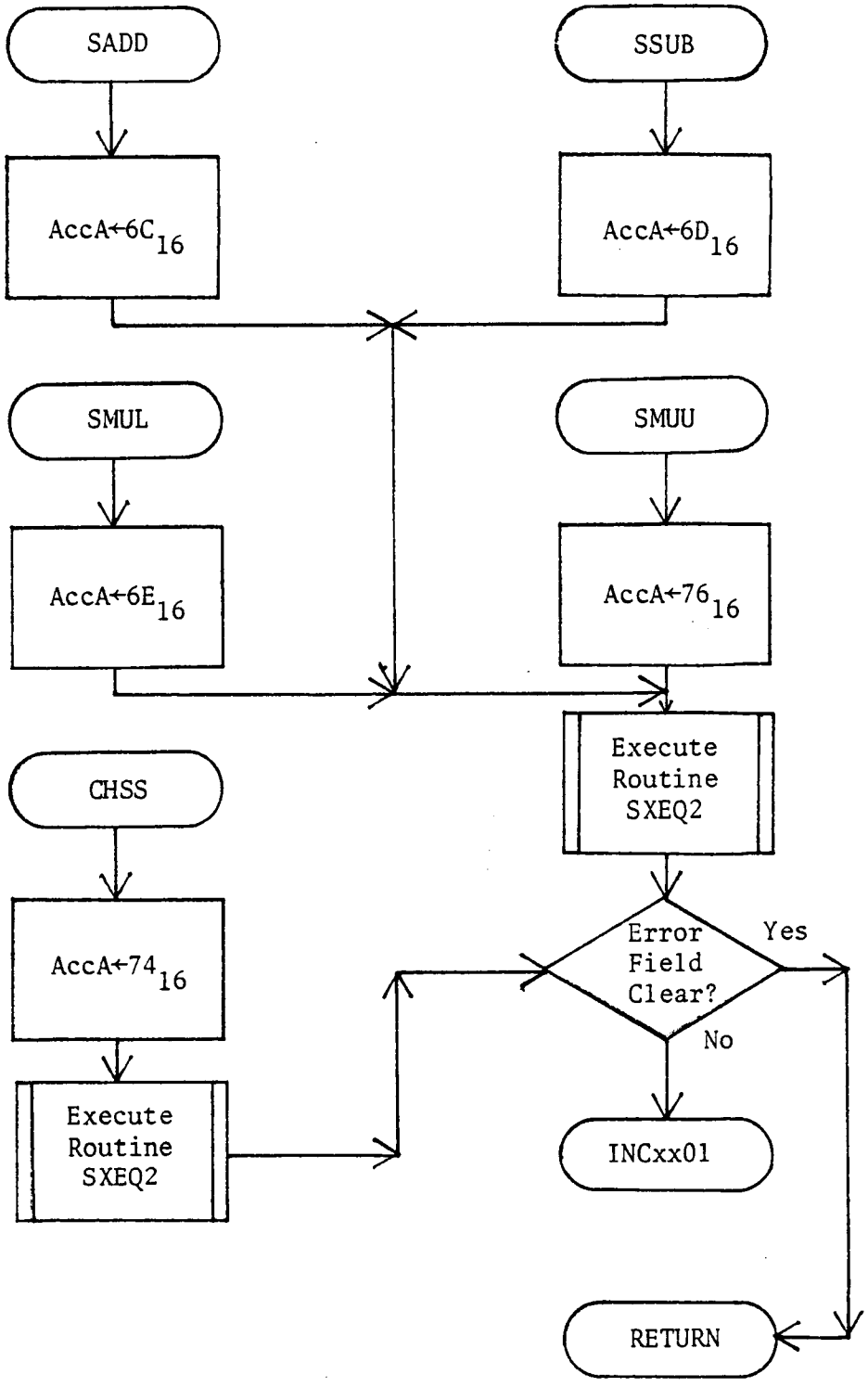


Figure B-16. Complete-operation routines; single add, subtract, multiply, and sign change.

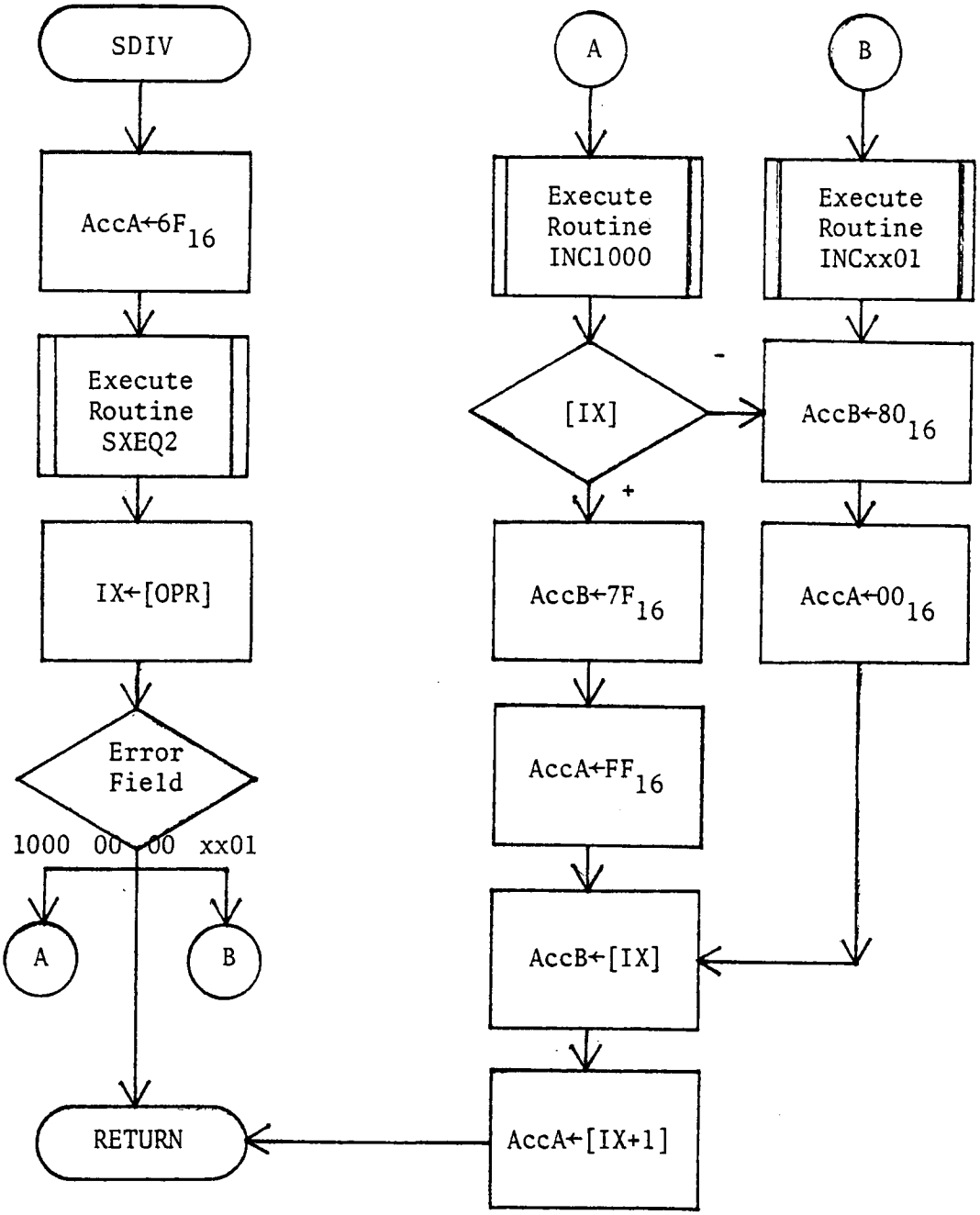


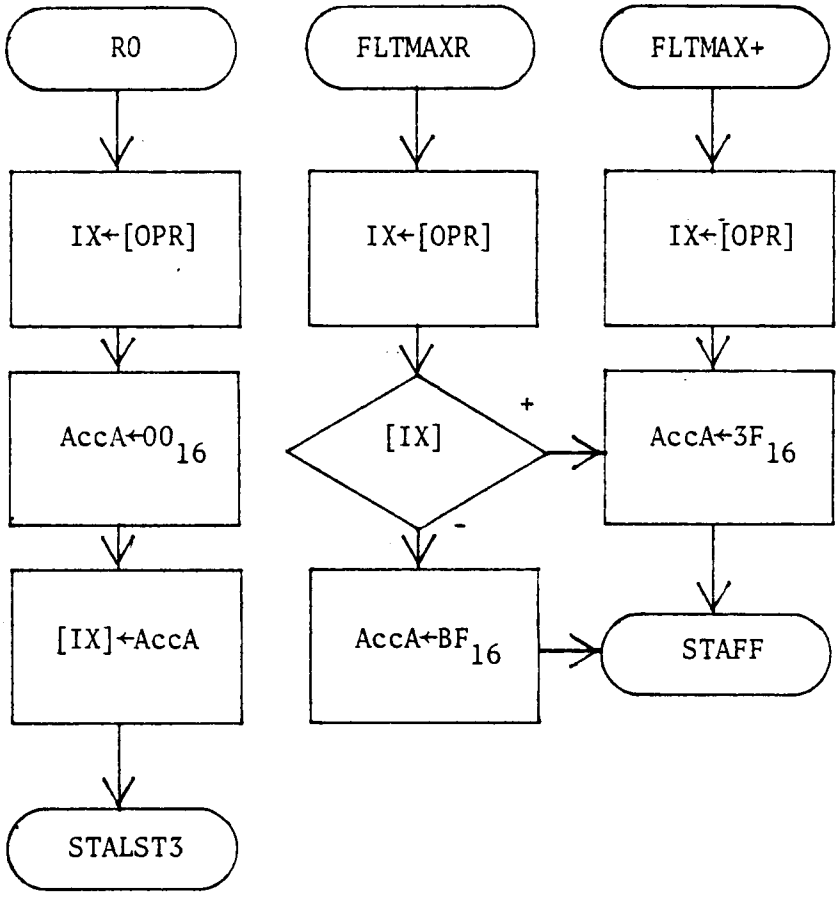
Figure B-17. Complete-operation routines; single divide.

```
ERRTOT   DB   1          Flag for all error
ERR1100  DB   1          Flag for argument out of range
ERR1000  DB   1          Flag for zero divide
ERR0100  DB   1          Flag for negative argument
ERRxx10  DB   1          Flag for underflow
ERRxx01  DB   1          Flag for overflow
FLTMAXR  LDX  OPR        Load IX with addr of result R
         TST  X          Find sign of R
         BPL  FLT+       If positive, set R to 3F FF FF FF16
         LDA  A,BFh      Set R to BF FF FF FF16
         BRA  STAFF
FLTMAX+  LDX  OPR        Load IX with addr of result R
FLT+     LDA  A,3Fh      Set R to 3F FF FF FF16
STAFF    STA  A,X
         LDA  A,FFh
STALST3  STA  A,X+1
         STA  A,X+2
         STA  A,X+3
         RTS
R0       LDX  OPR        Load IX with addr of result R
         CLR  A          Set R to 0
         STA  A,X
         BRA  STALST3
DPMAXR   LDX  OPR        Load IX with addr of result R
         TST  X          Find sign of R
         BMI  DPN1       If R negative, set R to
                           80 00 00 0016
         LDA  A,7Fh      If R positive, set R to
                           7F FF FF FF16
         BRA  STAFF
DPMAXN   LDX  OPR        Load IX with addr of result R
DPN1     LDA  A,80h      Set R to 80 00 00 0016
         STA  A,X
         CLR  A
         BRA  STALST3
```

(see Fig. B-18)

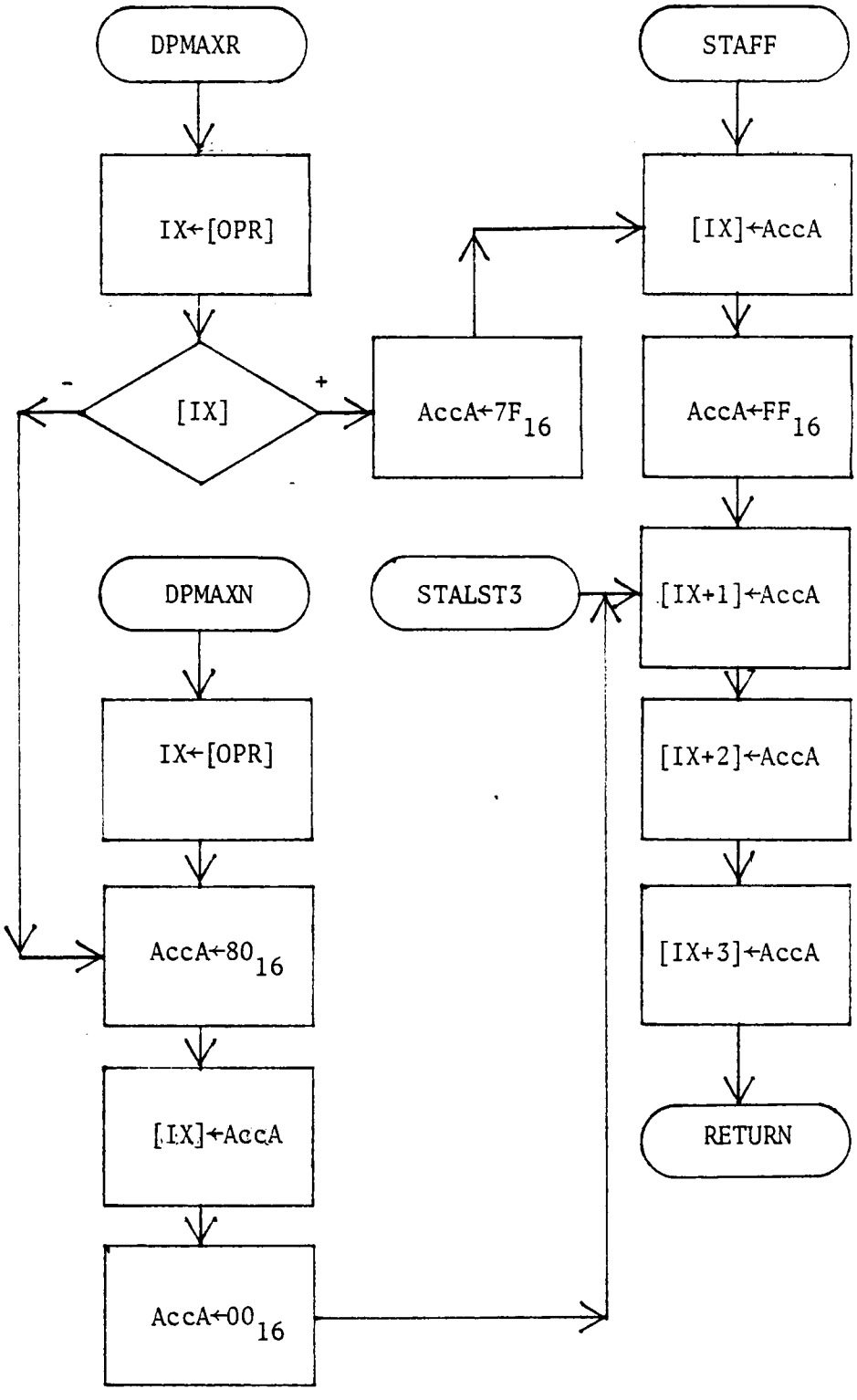
```
INC1100  INC  ERR1100    Flag argument out of range
                           (EXP, ACOS, ASIN)
         BRA  INCTOT
INC1000  INC  ERR1000    Flag zero divide
         BRA  INCTOT
INC0100  INC  ERR0100    Flag negative argument
                           (LOG, LN, SQRT)
         BRA  INCTOT
INCxx10  INC  ERRxx10    Flag underflow
         BRA  INCTOT
INCxx01  INC  ERRxx01    Flag overflow
INCTOT   INC  ERRTOT     Flag error total
         RTS             Return
```

(see Fig. B-19)



a. Set Result to Zero and Floating Maximum

Figure B-18. Complete-operation routines; set result to zero, floating maximum, double maximum, and store values via index register.



b. Set Result to Double Maximum and Store Values via Index Register

Figure B-18 (continued)

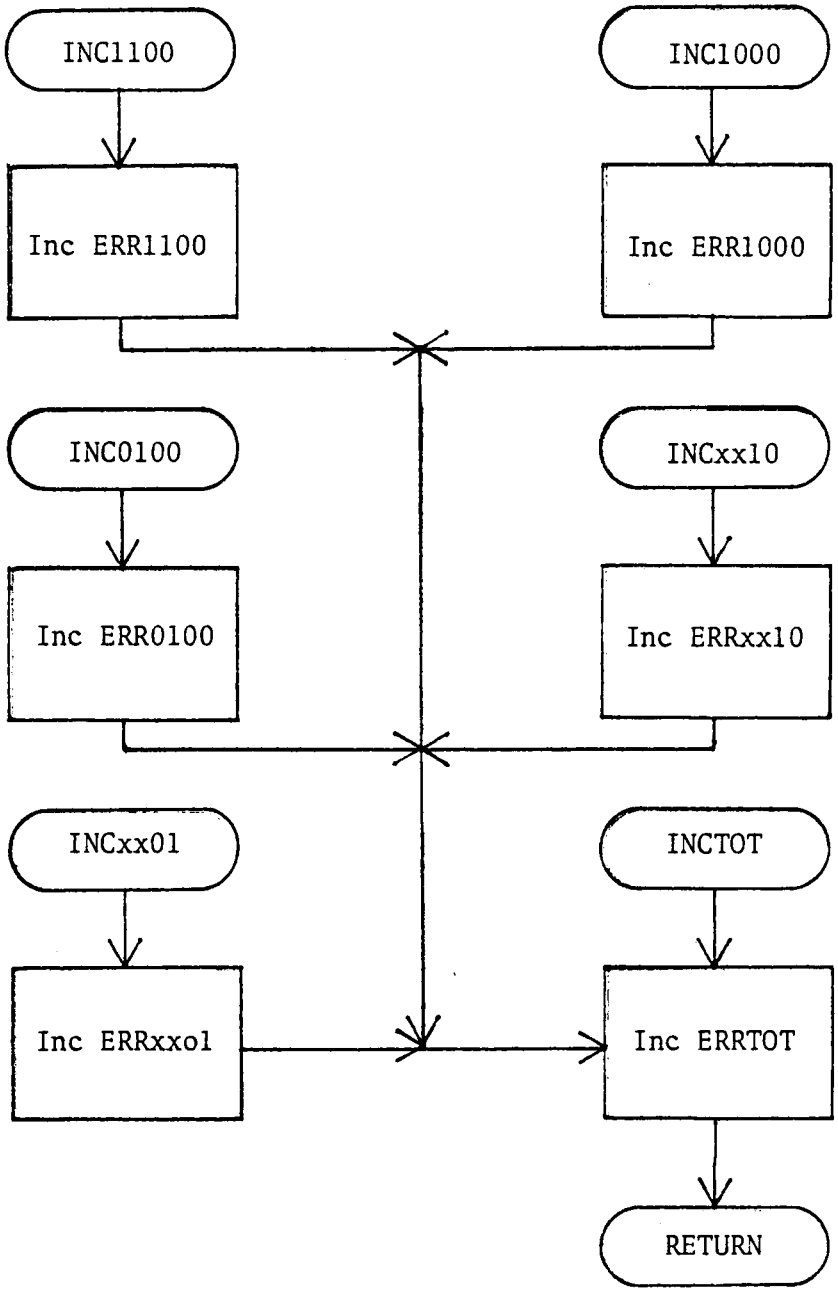


Figure B-19. Complete-operation routines; flag errors.

```
FLT10     DB   4          '04 A0 00 00' floating 10
STNGADDR  DB   2          Scratch addr
MANTAD    DB   2          Address of final result
MANT      DB   4          Scratch mantissa
EXPNT     DB   1          Scratch exponent
EXPCNT    DB   1          Scratch decimal count
FLG       DB   1          Scratch sign flag
INASCII   STX  STNGADDR   Save addr of ascii string to be
                            converted
          LDX  #MANT      Initialize mantissa, expnt, expcnt,
                            and flag
          LDA  B,07h      Set counter for initialization
N1        CLR  X
          INX
          DEC  B
          BNE  N1
          DEX
          INC  X          Set up flag for mantissa and
                            exponent signs
          LDX  STNGADDR   Recall addr of ascii string
          LDA  A,X        Access first byte of string
          INX
          CMP  A,2Dh      '-'?
          BNE  N2         If not, check for +
          INC  FLG        Flag a negative mantissa
          BRA  N3
N2        CMP  A,2Bh      '+'?
          BNE  N4         If not, assume positive mantissa
N3        LDA  A,X        Access next byte
          INX
N4        CMP  A,30h      Digit?  If not, try decimal point
          BLT  N5
          CMP  A,39h
          BGT  N5
          JSR  ADDMANT    Update mantissa value
          BRA  N3         Test next byte
N5        CMP  A,2Eh      Decimal point:  If not, try 'E'
          BNE  N7
N6        LDA  A,X        Access next byte
          INX
          CMP  A,30h      Digit?  If not, try 'E'
          BLT  N7
          CMP  A,39h
          BGT  N7
          JSR  ADDMANT    Update mantissa
          INC  EXPCNT     Count digit
          BRA  N6
```

```
N7      CMP   A,45h      'E'?
        BNE   N11        If not, try carriage return
        LDA   A,X        Access next byte
        INX
        CMP   A,2Dh      '-'?
        BNE   N8         If not, try +
        COM   FLG        Flag a negative power of 10
        BRA   N9
N8      CMP   A,2Bh      '+'?
        BNE   N10        If not, assume positive
N9      LDA   A,X        Access next byte
        INX
N10     CMP   A,30h      Digit?  If not, error
        BLT   ERR
        CMP   A,39h
        BGT   ERR
        AND   A,0Fh      Strip ascii digit
        STA   A,EXPNT    Store exponent
        LDA   A,X
        INX
        CMP   A,30h      Digit?  If not, try carriage return
        BLT   N11
        CMP   A,39h
        BGT   N11
        PSH   A          Multiply exponent by 10 and add
                           second digit
        LDA   A,EXPNT
        ASL   A          (Double)
        TAB
        ASL   A          (Double)
        ASL   A          (Double)
        ABA              (Original exponent now multiplied
                           by 10)
        PUL   B          Recall one's digit
        ABA              Add to exponent
        STA   A,EXPNT
        LDA   A,X        Access next byte
        INX
N11     TST   A          Carriage return?
        BNE   ERR        If not, invalid character
        LDA   A,EXPNT    Adjust exponent for any decimal
                           digits
        STA   A,D01Ch    Send to APU TOS
        CLR   A
        STA   A,D01Ch
        TST   FLG        If flag negative, exponent is
                           negative
        BPL   N12
        COM   FLG        Set flag up for mantissa sign test
        JSR   UCHSS      Change sign of exponent
```

```
N12       LDA   A,EXPCNT    Subtract decimal count from
                              exponent
          STA   A,D01Ch
          CLR   A
          STA   A,D01Ch
          JSR   USSUB
          JSR   UFLTS
          LDX   #FLT10      Raise 10 to the power expressed by
          JSR   STK4          exponent
          JSR   UXCHF
          JSR   UPWR
          BSR   STKMANT     Stack mantissa
          DEC   FLG         Find mantissa sign
          BEQ   N13         If positive, skip
          JSR   UCHSF
N13       JSR   UFMUL       Multiply mantissa by 10 to the
                              power
          LDX   MANTADDR    Unload converted value into
          JMP   UNSTK4        address contained in mantaddr
ERR       LDX   MANTADDR    Set value to 0
          CLR   A
          STA   A,X
          JMP   STALST3

ADDMANT   PSH   A           Save digit to be converted
          BSR   STKMANT     Stack accumulated mantissa
          LDX   #FLT10      Multiply mantissa by 10
          JSR   STK4
          JSR   UFMUL
          PUL   A           Recall digit
          AND   A,0Fh       Strip ascii value
          STA   A,D01Ch     Add digit to mantissa
          CLR   A
          STA   A,D01Ch
          JSR   UFLTS
          JSR   UFADD
          LDX   #MANT
          JMP   UNSTK4

STKMANT   LDX   #MANT       Load mantissa address
          JMP   STK4        Stack mantissa in APU
```

(see Fig. B-20)

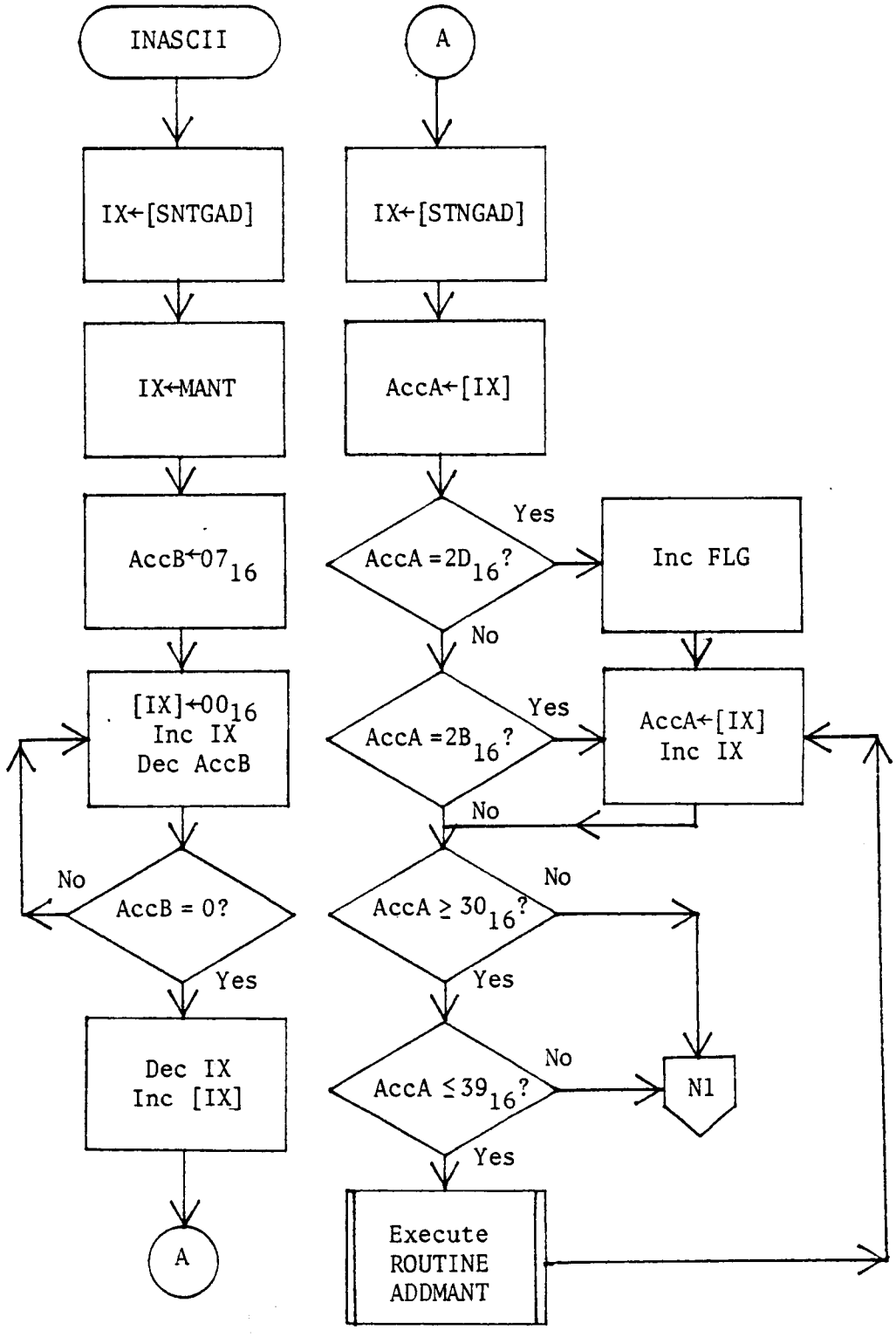


Figure B-20. Conversion of ASCII to floating point.

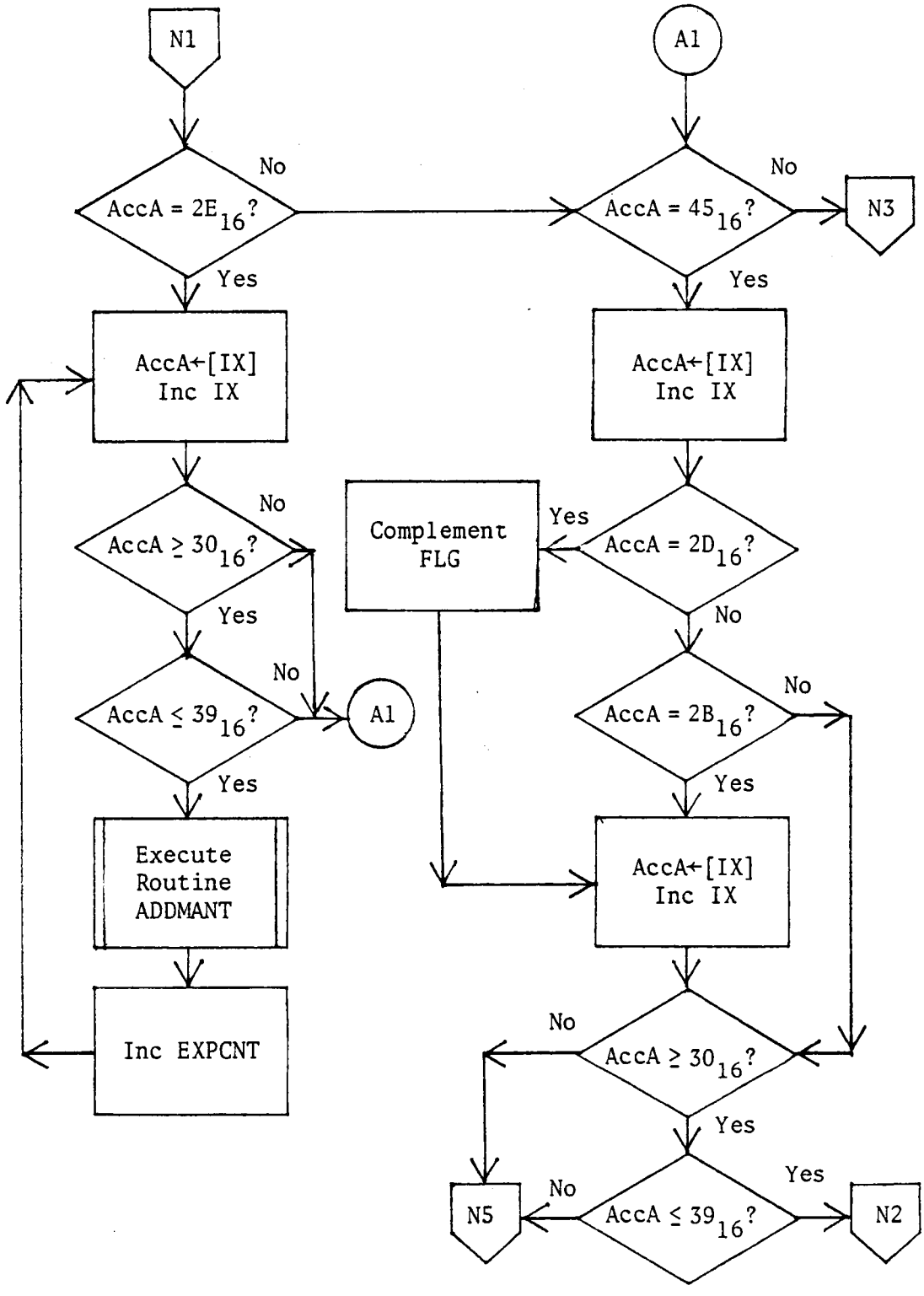


Figure B-20 (continued)

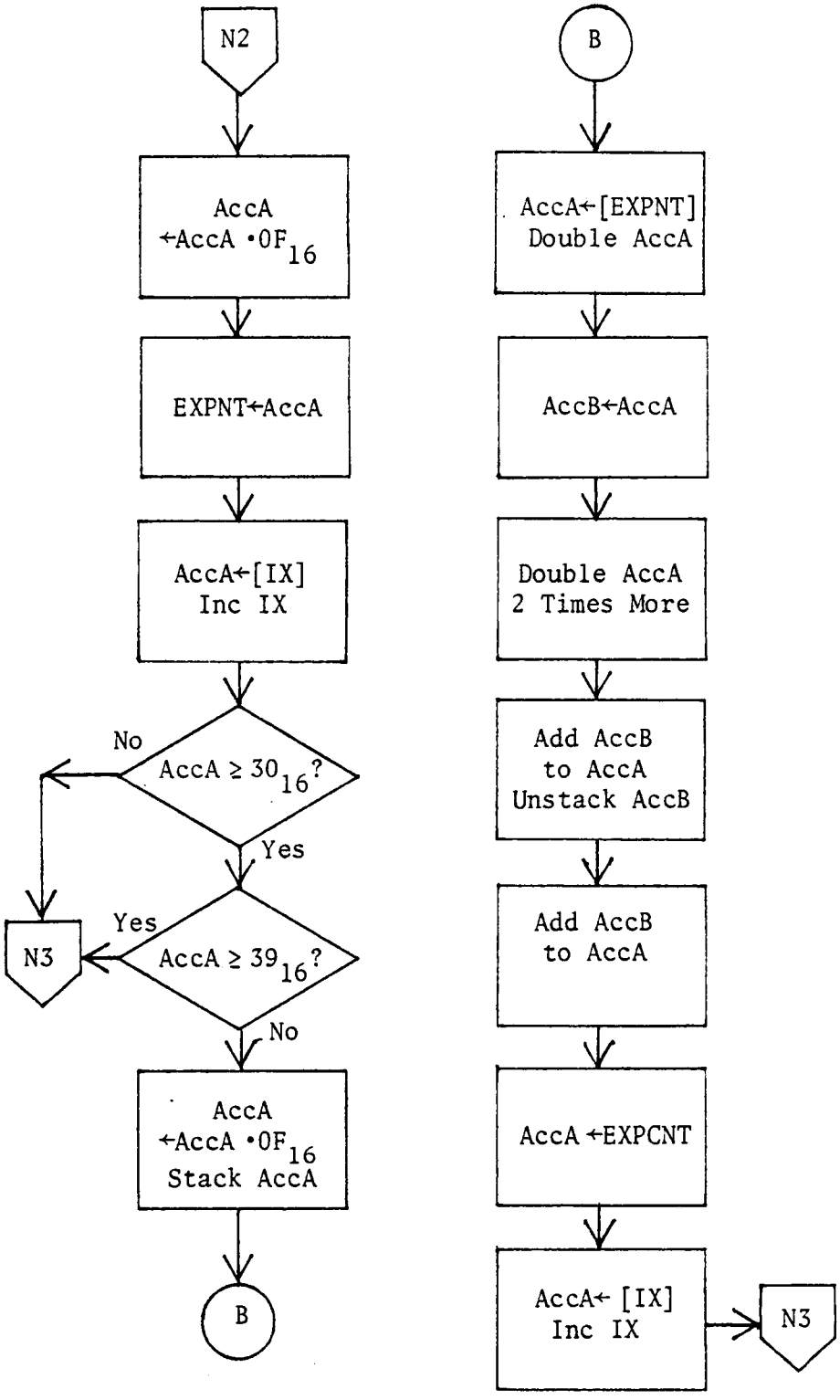


Figure B-20 (continued)

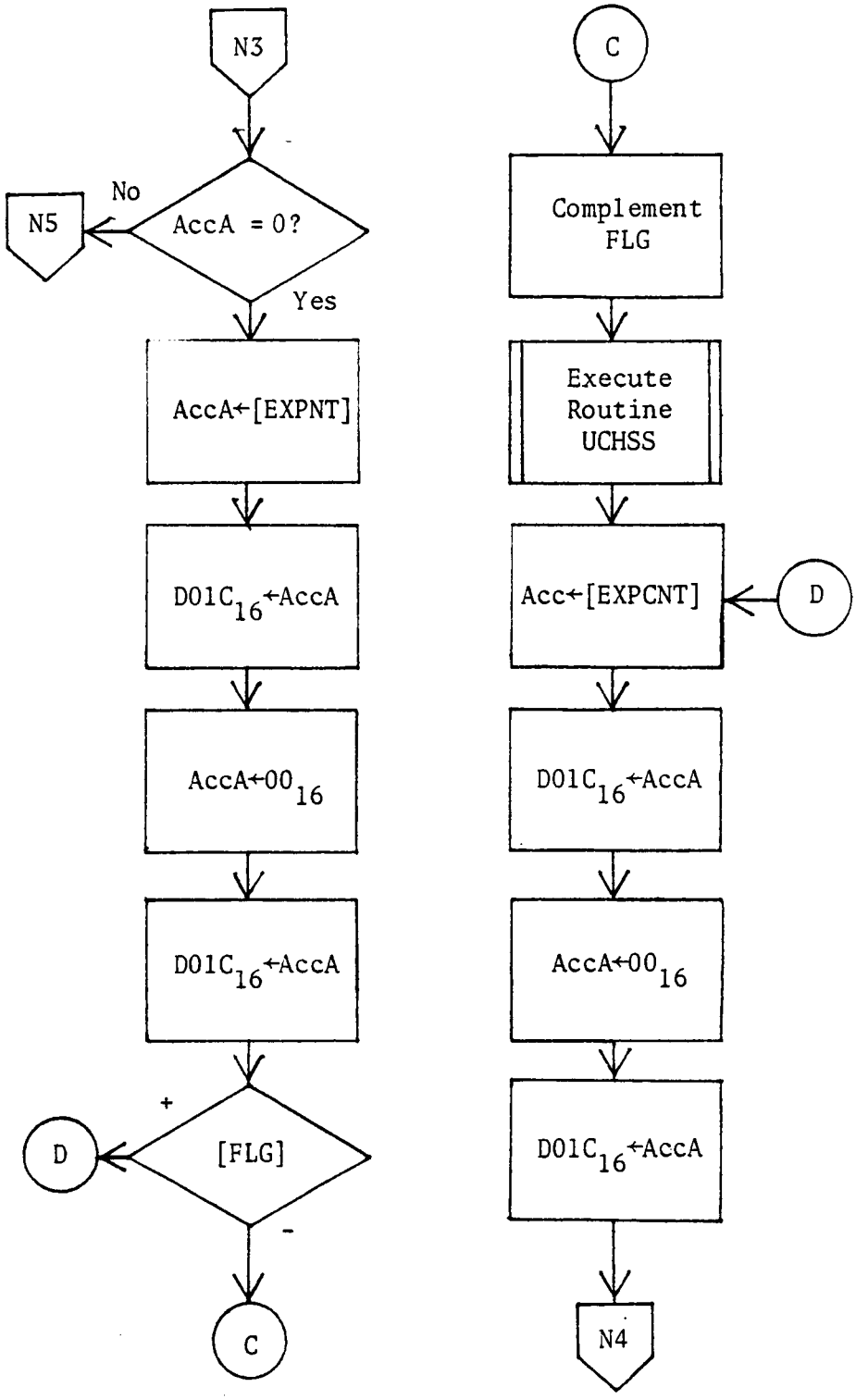


Figure B-20 (continued)

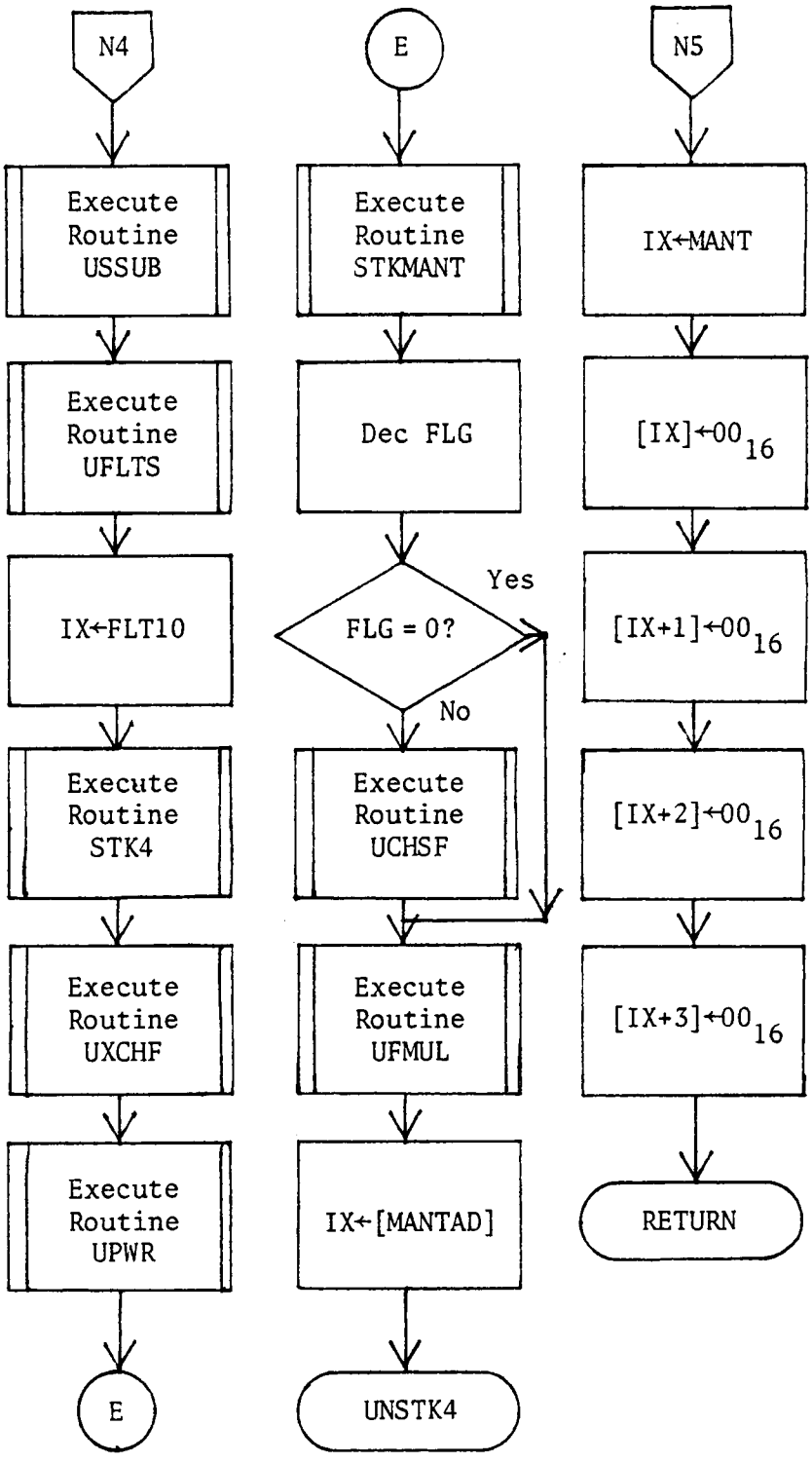


Figure B-20 (continued)

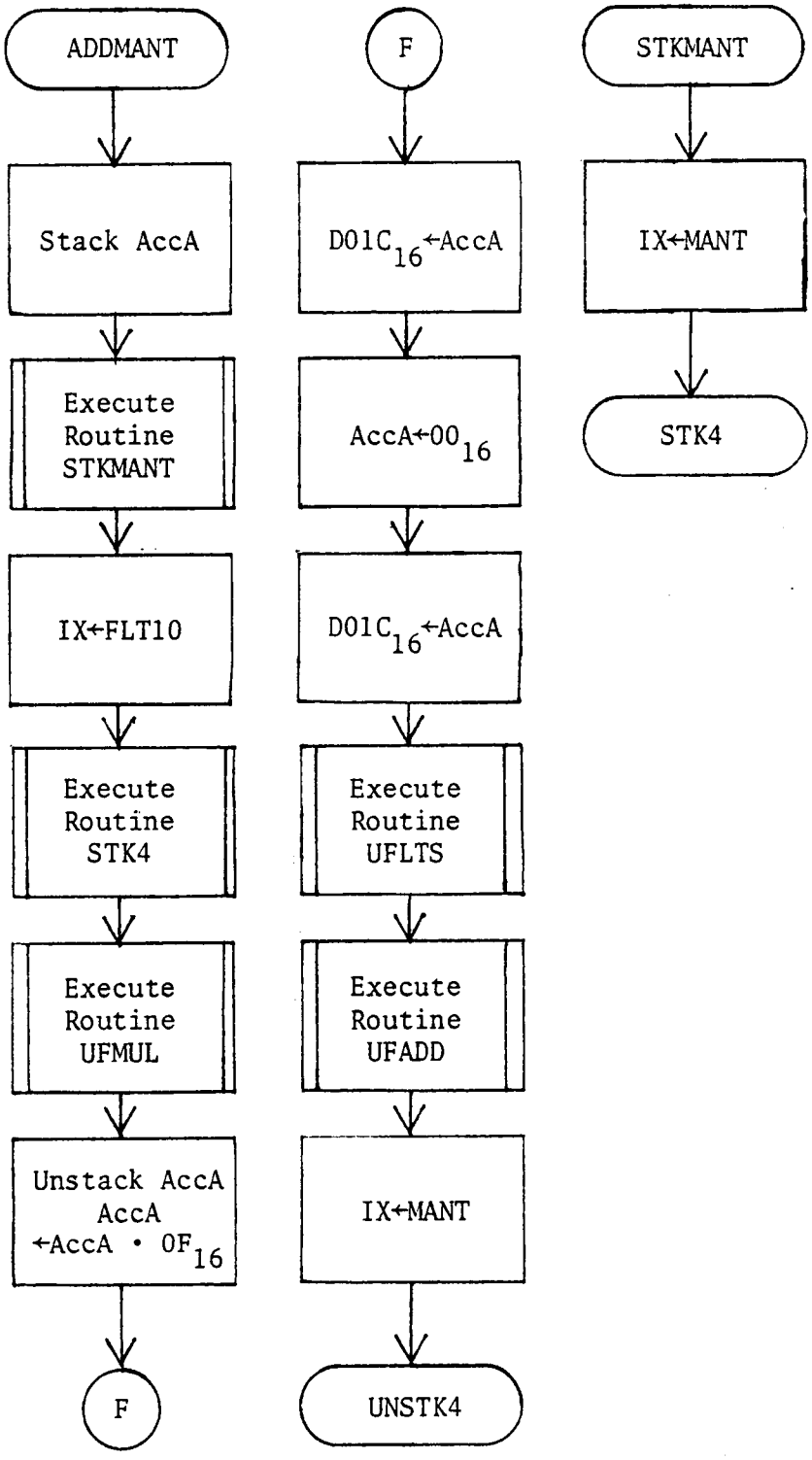


Figure B-20 (continued)

```
SIGN     DB   1        Sign of output value
ACC      DB   1}
ACCF1    DB   8}       Output mantissa
ACCLST   DB   1}
ACCTRL   DB   1}       Rounding digits and output for 'E'
                         and exponent sign
EDGT     DB   2        Output for exponent
TAG      DB   1        End of string
PWR10    DB   1        Counter for exponent
SAVX     DB   2        Scratch memory for output value
                         addr
SAVACC   DB   2        Scratch memory for accumulator
                         addr
SAVF     DB   2        Scratch memory for FRCTBL addr


FRCLST   DC   48       Table to address 24 parts of
                         FRCTBL
```

```
FRCTBL    DC    216        '05 00 00 00 00 00 00 00 00';2**-1
                           '02 05 00 00 00 00 00 00 00';2**-2
                           '01 02 05 00 00 00 00 00 00';2**-3
                           '00 06 02 05 00 00 00 00 00';2**-4
                           '00 03 01 02 05 00 00 00 00';2**-5
                           '00 01 05 06 02 05 00 00 00';2**-6
                           '00 00 07 08 01 02 05 00 00';2**-7
                           '00 00 03 09 00 06 02 05 00';2**-8
                           '00 00 01 09 05 03 01 02 05';2**-9
                           '00 00 00 09 07 06 05 06 03';2**-10
                           '00 00 00 04 08 08 02 08 01';2**-11
                           '00 00 00 02 04 04 01 04 01';2**-12
                           '00 00 00 01 02 02 00 07 00';2**-13
                           '00 00 00 00 06 01 00 03 05';2**-14
                           '00 00 00 00 03 00 05 01 08';2**-15
                           '00 00 00 00 01 05 02 05 09';2**-16
                           '00 00 00 00 00 07 06 02 09';2**-17
                           '00 00 00 00 00 03 08 01 05';2**-18
                           '00 00 00 00 00 01 09 00 07';2**-19
                           '00 00 00 00 00 00 09 05 04';2**-20
                           '00 00 00 00 00 00 04 07 07';2**-21
                           '00 00 00 00 00 00 02 03 08';2**-22
                           '00 00 00 00 00 00 01 01 09';2**-23
                           '00 00 00 00 00 00 00 06 00';2**-24
```

```
OUTASCII TST  X+1       Test for output value of 0
         BNE  OUT2      If not zero, go to conversion
         LDX  #SIGN     Prepare output string of
                          ' 0 '
         LDA  B,0Eh     Prepare counter
         LDA  A,20h     Prepare ascii value of a space
OUT1     STA  A,X
         INX
         DEC  B
         BNE  OUT1
         STA  B,X       (TAG) the trailer with the ascii
         LDA  A,30h       for carriage return
         STA  A,ACC     Output string is ' 0             '
         RTS
OUT2     STX  SAVX      Save address of output value
         LDX  #ACC      Initialize accumulator, power 10,
         LDA  B,0Fh       and string
OUT3     CLR  X
         INX
         DEC  B
         BNE  OUT3
         LDX  SAVX      Load IX with addr of output value
         LDA  A,X+1     Access first byte of mantissa
         BSR  ASCACC    Translate all set bits to
                          accumulated value
         LDA  A,X+2     Access second byte of mantissa
         BEQ  OUT4      If second byte all zeros, try
                          third byte
         LDA  B,08h     Set counter for second byte
                          translation
         BSR  ASCACC    Convert second byte
OUT4     LDA  A,X+3     Access third byte
         BEQ  OUT5      If third byte is 0, do not convert
         LDA  B,10h     Set counter for third byte
                          conversion
         BSR  ASCACC    Convert third byte
OUT5     LDA  A,X       Find sign of mantissa
         BPL  MPLUS
         LDA  B,2Dh     For negative mantissa,
         BRA  MMINUS      set SIGN to '-'
MPLUS    LDA  B,2Bh     For positive mantissa,
                          set SIGN to '+'
MMINUS   STA  B,SIGN
         ASL  A         Set all eight bits to the exponent
         ASR  A           value
         BEQ  ROUNDIT   If exponent is 0, go to rounding
                          code
         BPL  DOUBLE    If exponent is +, adjust
                          accumulator
```

```
HALVE   PSH  A          Save exponent value
        LDX  #ACCF1     Address accumulator
        LDA  B,0Ah      Set counter
        CLC             Clear carry bit for logical shift
                          right
H0      LDA  A,X        Load byte of accumulator
        BCC  H1         If no carry from previous byte,
                          skip
        ADD  A,0Ah      Add 10 before halving
H1      LSR  A          Halve accumulator byte
        STA  A,X        Store adjusted accumulator byte
        INX             Address next byte
        DEC  B          Adjust counter
        BNE  H0         If not last byte, halve next byte
        TST  ACCF1      If MSB of accumulator not zero,
        BNE  H3           skip
        LDX  #ACCF1     Address accumulator
        DEC  PWR10      Adjust counter for powers of 10
        LDA  B,0Ah      Set counter for shifting
                          accumulator left
H2      LDA  A,X+1      Shift
        STA  A,X
        INX
        DEC  B
        BNE  H2
H3      PUL  A          Recall exponent value
        INC  A          Adjust to indicate halving
        BNE  HALVE      If not zero, halve again
        BRA  ROUNDIT    Go to roundoff code


DOUBLE  PSH  A          Save exponent value
        LDX  #ACCTRL    Access LSB of accumulator
        LDA  B,0Ah      Set counter for doubling
        CLC             Clear carry bit for rotate left
D0      LDA  A,X        Load byte of accumulator
        ROL  A          Double accumulator
        CMP  A,0Ah      If greater than 10, adjust carry
        CLC
        BLT  D1
        SUB  A,0Ah      Subtract 10
        SEC             Set carry
D1      STA  A,X        Store adjusted byte of accumulator
        DEX             Address preceding byte
        DEC  B          Adjust counter
        BNE  D0         If not end, loop back
        BCC  D2         If carry is clear, skip
        BSR  SLPRGHT    Shift accumulator to right
D2      PUL  A          Recall exponent value
        DEC  A          Adjust exponent
        BNE  DOUBLE     If not zero, double again
```

```
ROUNDIT   LDX   #ACCTRL     Addr LSB of accumulator for
                              round off
          LDA   A,05h       Add 5 to LSB
          ADD   A,X
          LDA   B,09h       Set counter
          DEX               Address next byte of accumulator
          CMP   A,0Ah       Is byte greater than or equal
                              to 10?
          BLT   RNDNXT      If not, skip
          INC   X           Adjust next significant
                              accumulator byte
RNDNXT    LDA   A,05h       Add 5 to next byte
RNDLOOP   ADD   A,X
          CMP   A,0Ah       If greater than or equal to 10,
                              adjust
          BLT   RNDFIN      If not, rounding is finished
          SUB   A,0Ah
          STA   A,X
          DEX
          LDA   A,01h
          DEC   B
          BNE   RNDLOOP
          STA   A,X         If all bytes are rounded, set MSB
          BSR   SLPRGHT       to 1 and shift accumulator
RNDFIN    LDX   #ACCLST-1   Adjust LSB of number string
          LDA   B,07h       Replace all trailing 0's with
                              spaces
ASCSP     TST   X           If not a zero, go to number
          BNE   ASCNUM        conversion
          STA   A,X         Replace 0 with space
          DEX               Address next byte
          DEC   B           Adjust counter
          BNE   ASCSP
          BRA   DECPT       If all decimal digits, 0, skip
                              number conversion
ASCNUM    LDA   A,X         Convert number to ascii character
          ORA   A,30h
          STA   A,X
          DEX               Convert all other numbers to ascii
          DEC   B
          BNE   ASCNUM
DECPT     LDA   A,X         Get digit to precede decimal point
          ORA   A,30h       Convert to ascii
          DEX               Adjust IX to preceding byte
          STA   A,X
          LDA   A,2Eh       Character for '.'
          STA   A,X+1       Store '.' (sn.nnnnnnn)
```

```
         LDX   #ACCLST     Address place for 'E'
         LDA   B,PWR10     Find value of power of 10
         DEC   B           Adjust for shifting character to
                             left of '.'
         BEQ   SP4         If power of 10 zero, set E field
                             to spaces
         LDA   A,45h       Character for 'E'
         STA   A,X
         TST   B           Find sign of power of 10
         BPL   EPLUS
         LDA   A,2Dh       Set to '-'
         NEG   B           Adjust power of 10 for final
         BRA   EMINUS        conversion
EPLUS    LDA   A,2Bh       Set to '+'
EMINUS   STA   A,X+1
         CLR   A           Prepare AccA for E digit
                             conversion
EDGT     CMP   B,0Ah       Is value greater than 9?
         BLT   ESTORE      If not, store value
         INC   A           Adjust 10's digit
         SUB   B,0Ah       Adjust 1's digit
         BRA   EDGT
ESTORE   ORA   A,30h       Ascii character for 10's digit
         ORA   B,30h       Ascii character for 1's digit
         STA   A,X+2       Store 10's digit
         STA   A,X+3       Store 1's digit
         RTS
SP4      LDA   A,20h       Store spaces
         STA   A,X
         JMP   STALST3
SLPRGHT  LDD   #ACCLST     Shift accumulator one byte to the
                             right
         LDA   B,09h       Set counter
SR1      LDA   A,X         Shift
         STA   A,X+1
         DEX
         DEC   B
         BNE   SR1
         INC   B           Store a 1 in the MSB accumulator
         STA   B,X+1
         INC   PWR10       Adjust power of 10
         RTS               Return
```

```
BITO     INC  B          Adjust counter
ASCACC   ASL  A          Entry point, find value of next
                           bit
         BCC  BITO       If not set, adjust counter and
                           try next
         LDX  #ACCLST    Address accumulator
         STX  SAVACC     Save accumulator address
         PSH  A          Save mantissa byte
         LDX  #FRCLST    Find address of representation of
                           this bit
         PSH  B          AccB determines which mantissa bit
         TST  B
         BEQ  FCRCT
FFIX     INX             FRCLST is a table of address for
         INX               the conversion of all 24 bits
         DEC  B            in the mantissa
         BNE  FFIX
FCRCT    LDX  X          Load IX with address in location
                           ref by IX
         LDA  B,09h      Set counter
ACCUM    LDA  A,X        Load byte from constant
         DEX             Adjust addressing for fractional
                           value
         STX  SAVF       Save address of fractional value
         LDX  SAVACC     Load address of accumulator
         ADD  A,X        Add to accumulator
         DEX             Adjust accumulator address
         CMP  A,0Ah      If greater than 10, adjust and
         BLT  LT10         carry
         SUB  A,0Ah
         INC  X
LT10     STA  A,X+1      Store sum in accumulator
         STX  SAVACC     Save accumulator address
         LDX  SAVF       Access fractional value address
         DEC  B          Adjust counter
         BNE  ACCUM      If not finished, repeat
         PUL  B          Unstack mantissa counter
         PUL  A          Recall mantissa byte
         TST  A          If mantissa byte not zero, repeat
         BNE  BITO
         LDX  SAVX       Recall address of value
         RTS             Return
```

(see Fig. B-21)

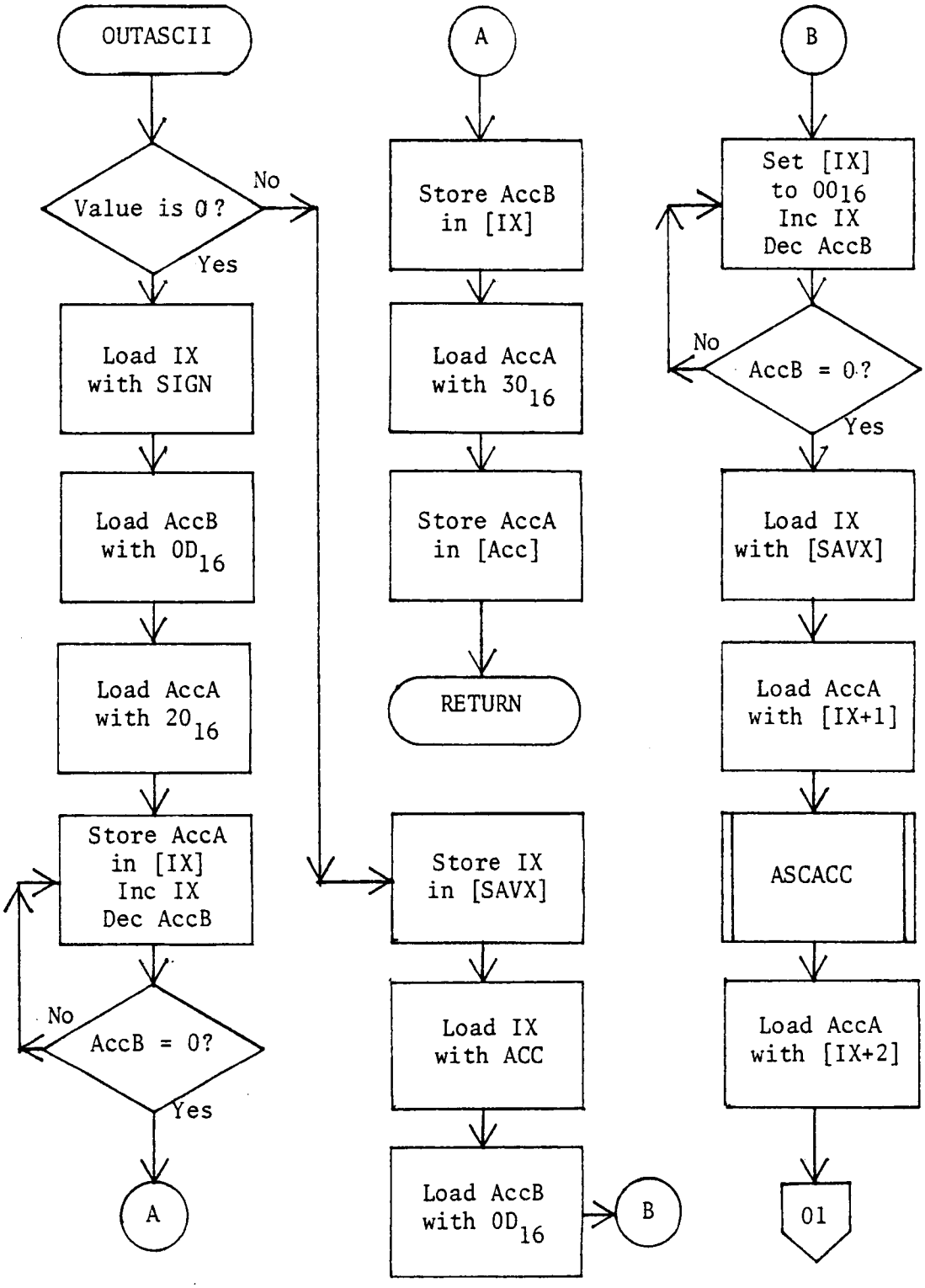


Figure B-21. Conversion of floating point to ASCII.

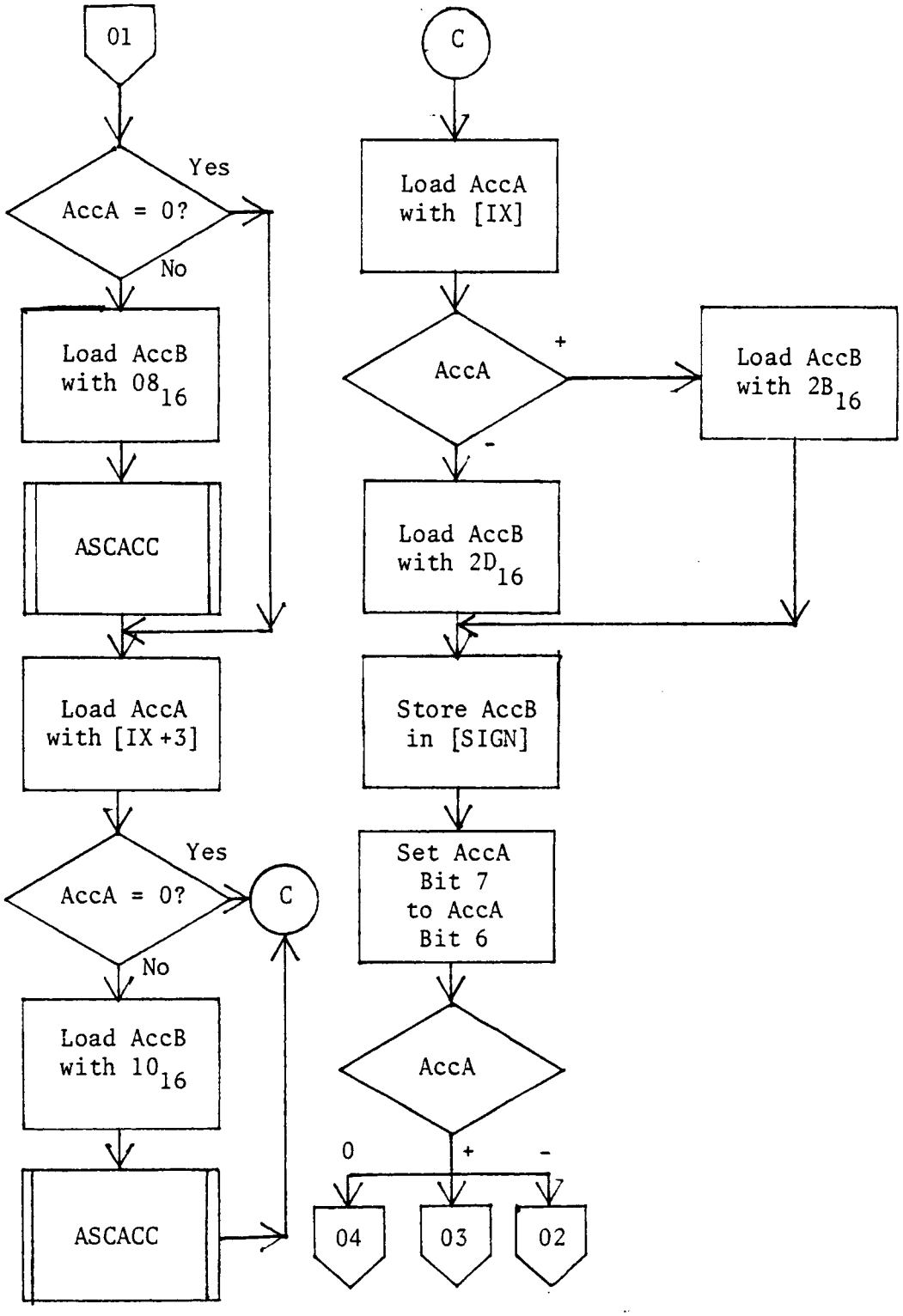


Figure B-21 (continued)

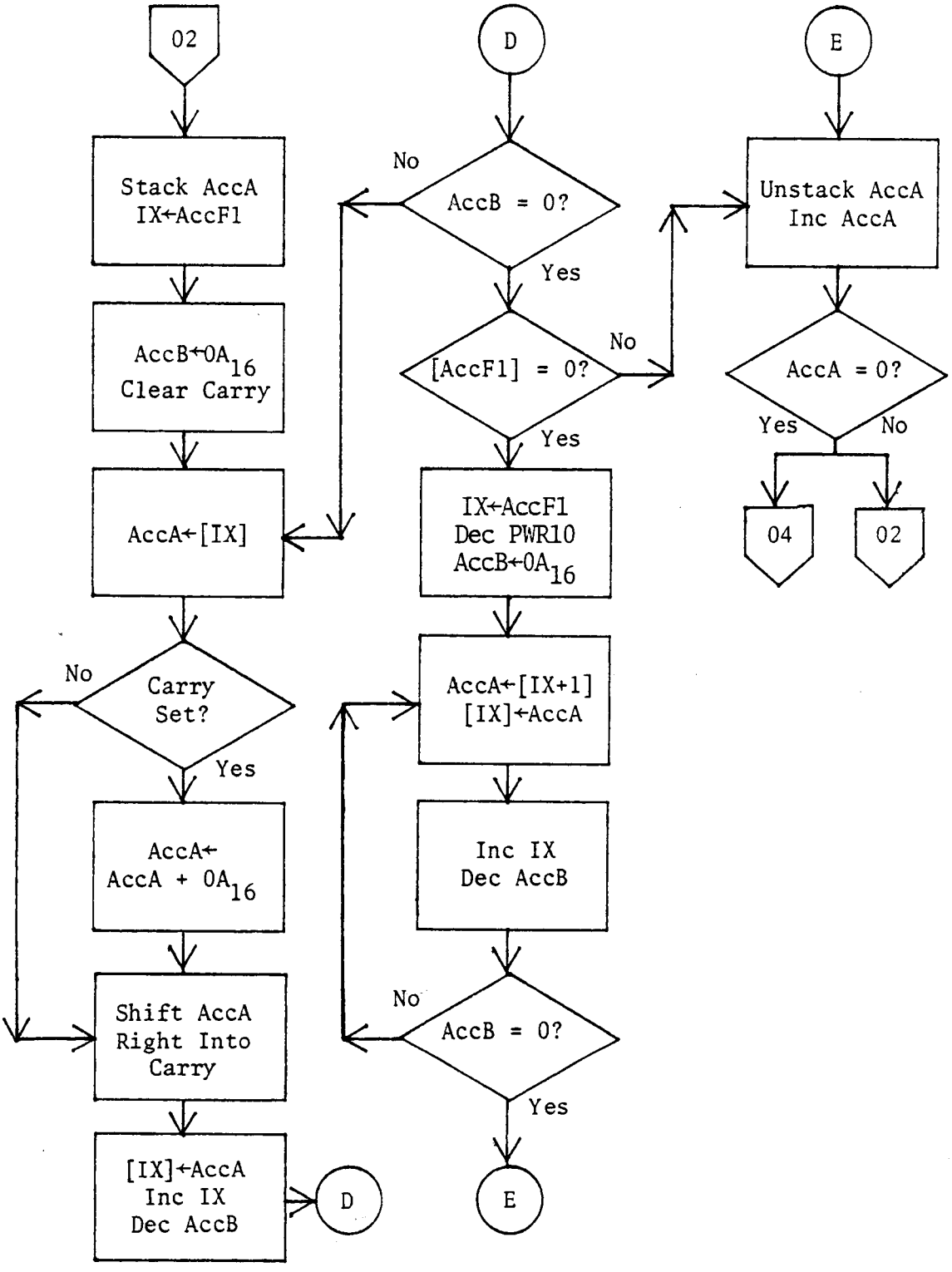


Figure B-21 (continued)

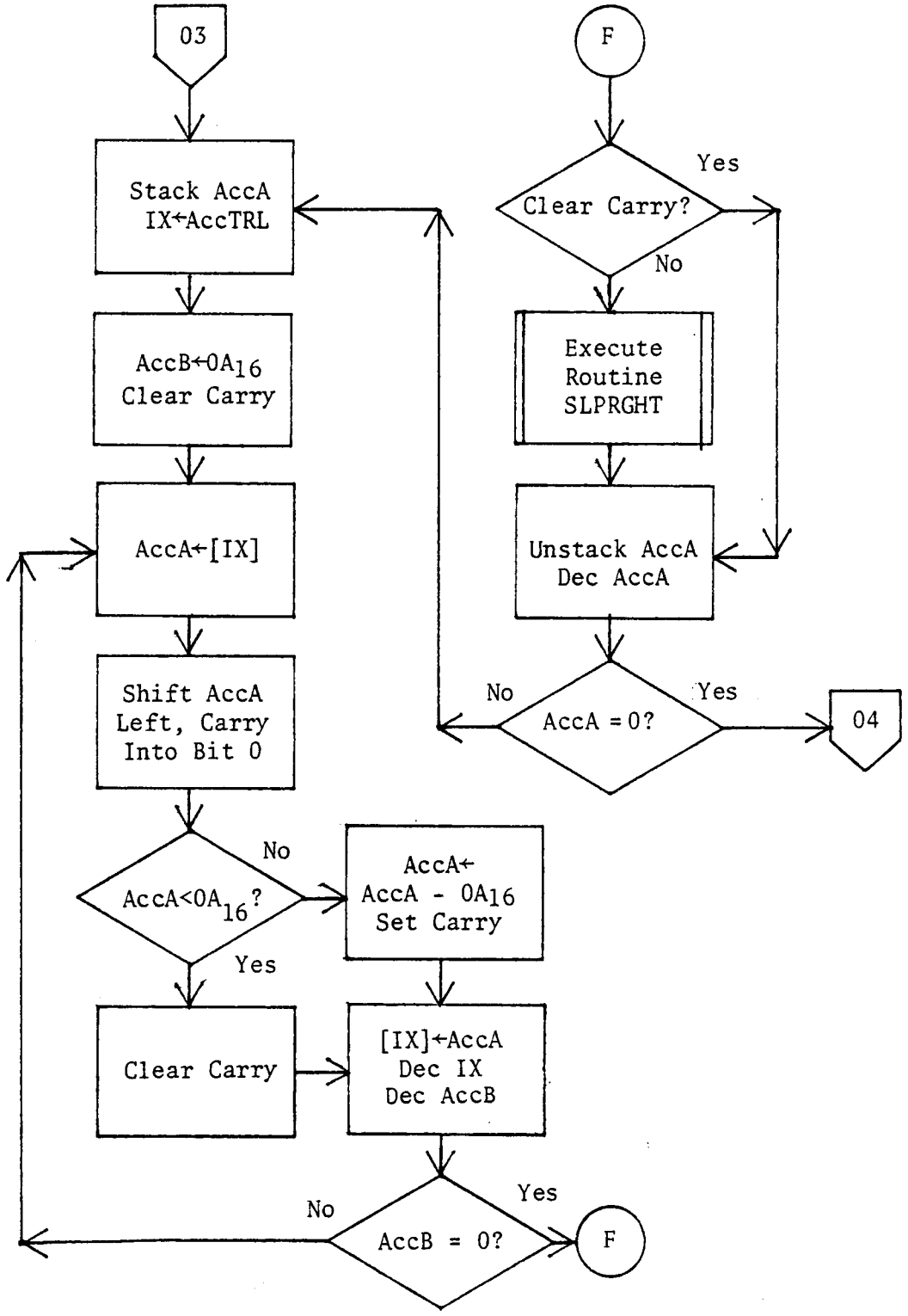


Figure B-21 (continued)

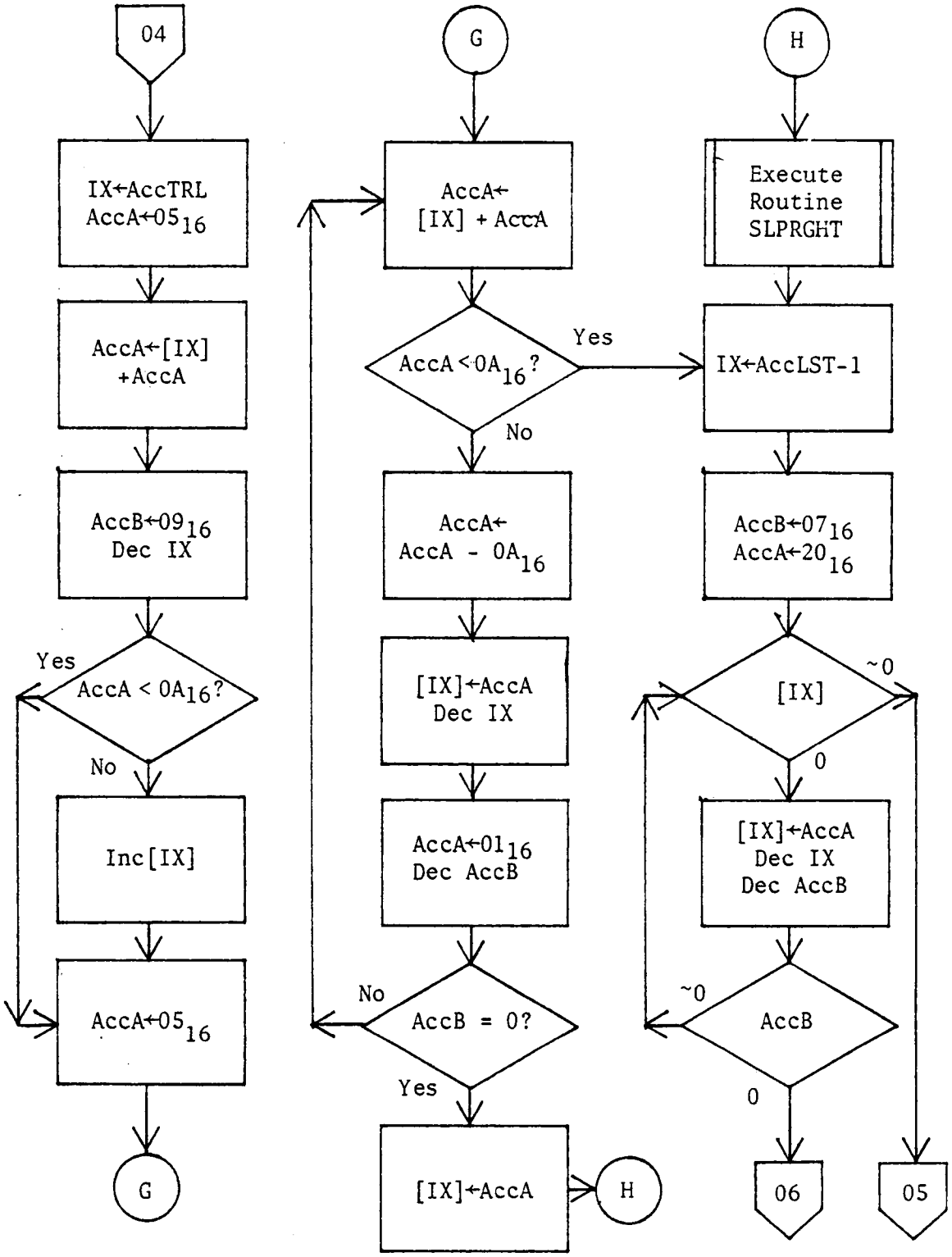


Figure B-21 (continued)

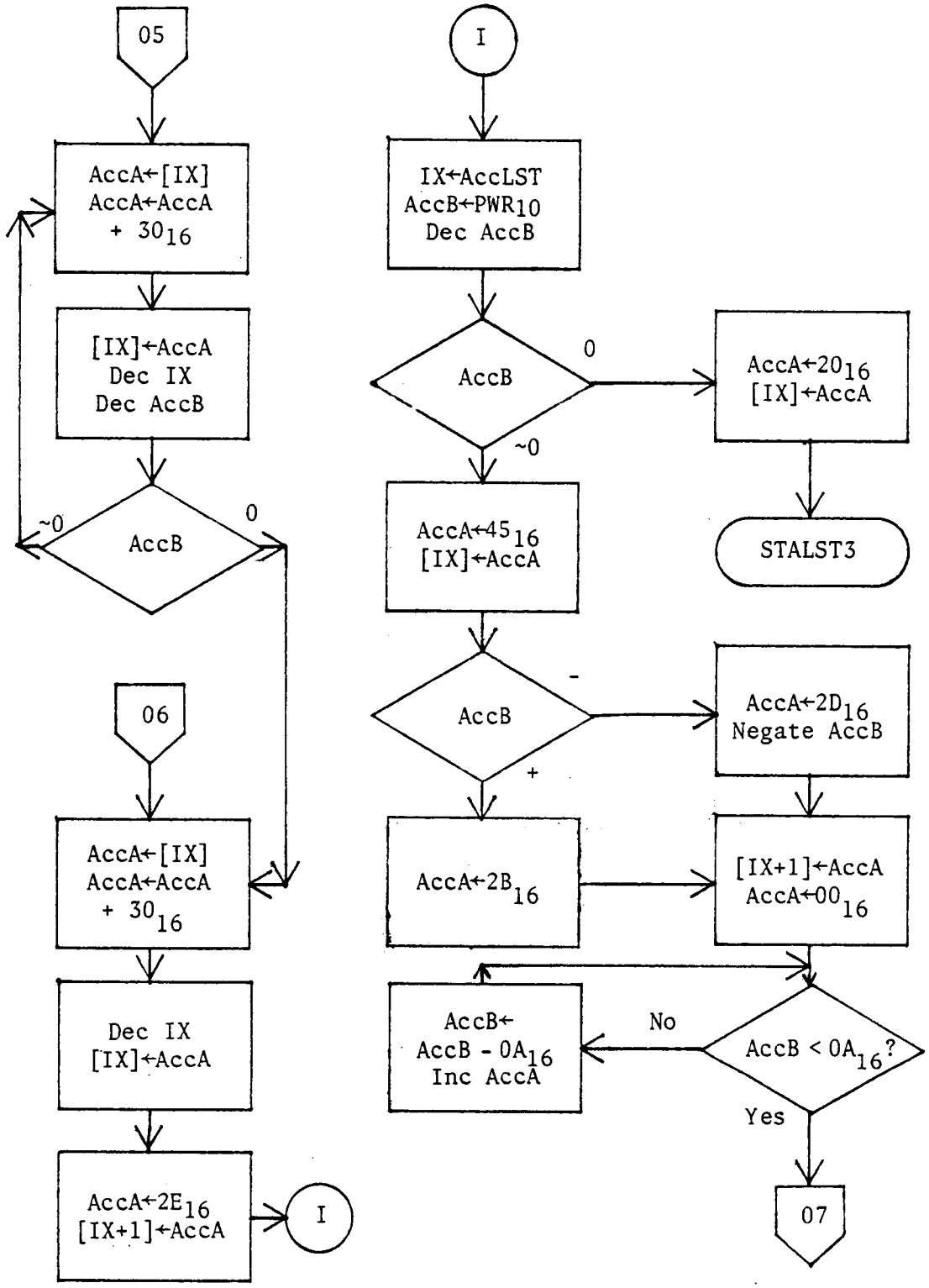


Figure B-21 (continued)

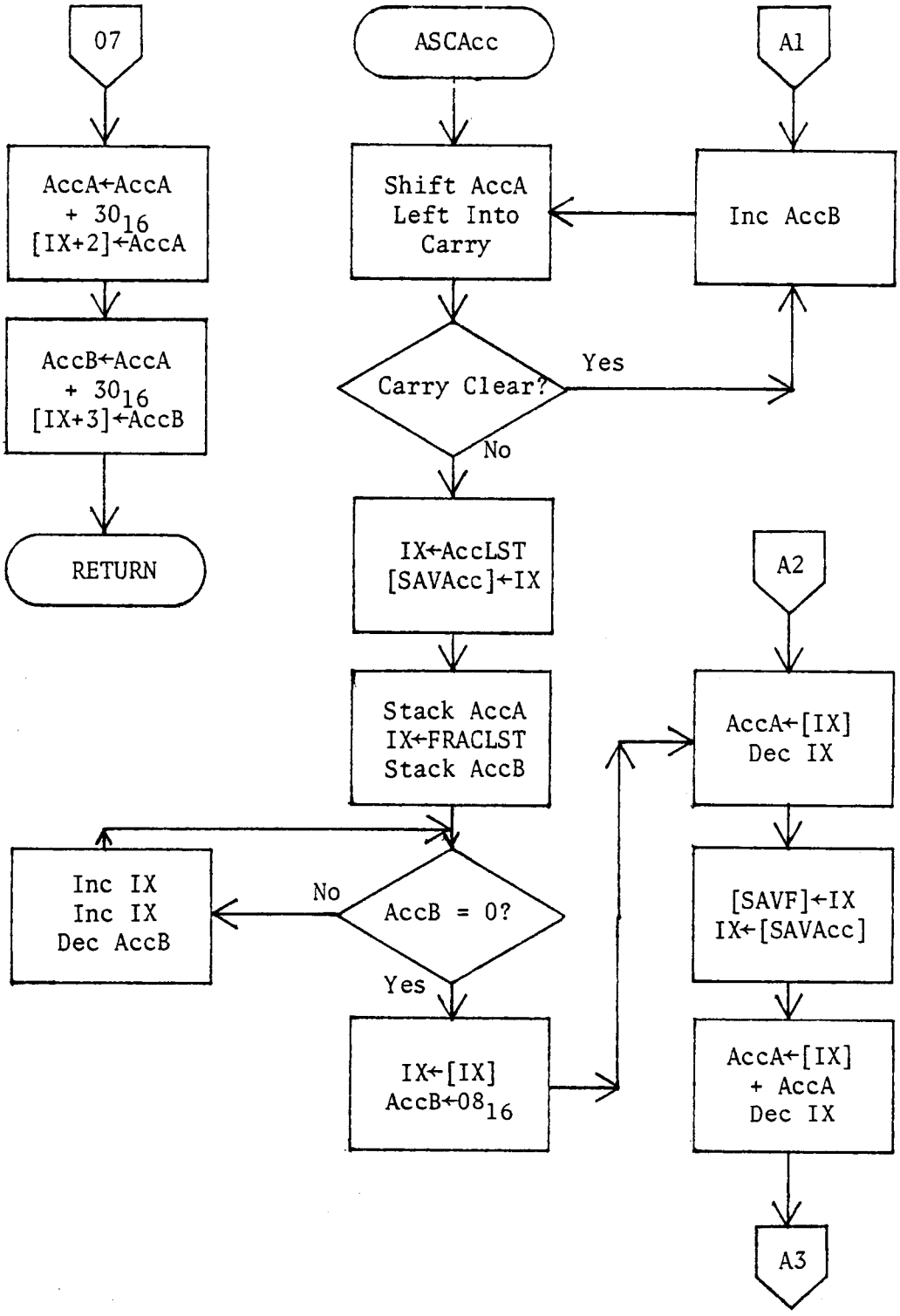


Figure B-21 (continued)

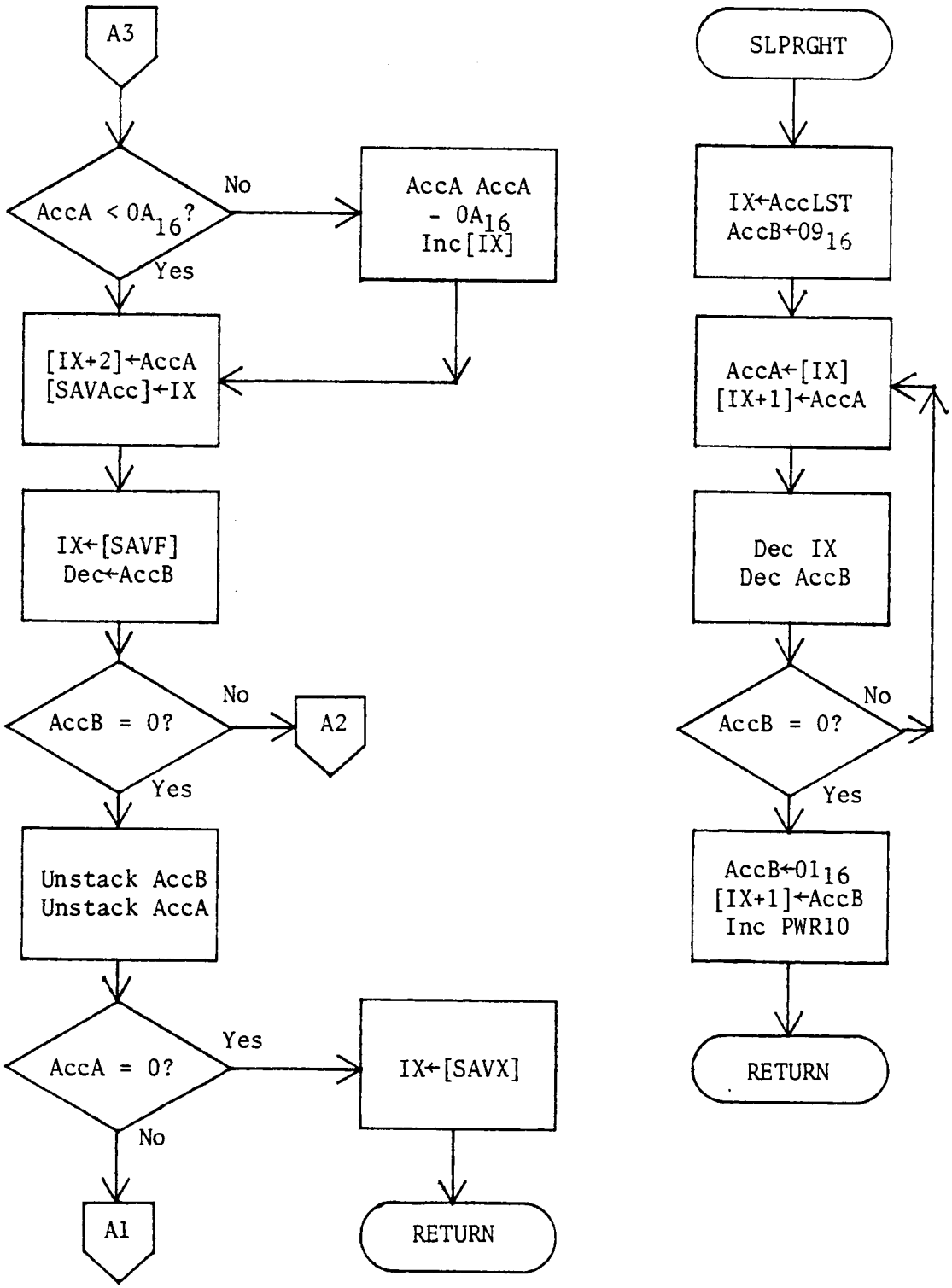


Figure B-21 (continued)

VITA

Susan Vickrey was born in Ardmore, Oklahoma on June 28, 1956. She was graduated from John Overton High School in Nashville, Tennessee in May 1974. She was awarded a National Merit Scholarship at Central Missouri State University in Warrensburg, Missouri and attended that university from September 1974 until May 1977 when she graduated magna cum laude with a Bachelor of Science degree in Computer Science.

In June 1977, Ms. Vickrey accepted a graduate research assistantship and entered The University of Tennessee Space Institute near Tullahoma, Tennessee. She received the Master of Science degree with a major in Computer Science in June 1980.

Ms. Vickrey is a member of Phi Kappa Phi, Kappa Mu Epsilon, and the American Institute of Aeronautics and Astronautics. She has accepted employment with Digital Equipment Corporation in Colorado Springs, Colorado.