



Advanced Micro Devices

Am9511A/Am9512 Floating Point Processor Manual

By Steven Cheng

The International Standard of Quality
guarantees these electrical AQLs on all
parameters over the operating tempera-
ture range: 0.1% on MOS RAMs & ROMs;
0.2% on Bipolar Logic & Interface; 0.3%
on Linear, LSI Logic & other memories.

© 1981 Advanced Micro Devices, Inc.

Advanced Micro Devices reserves the right to make changes in its products without notice in order to improve design or performance characteristics. The company assumes no responsibility for the use of any circuits described herein.

901 Thompson Place, P.O. Box 453, Sunnyvale, California 94086
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

Printed in U.S.A. 5/81 RMC-615

TABLE OF CONTENTS

CHAPTER 1 – AN INTRODUCTION TO FLOATING POINT

- 1.1 What Is a Floating Point Number?
- 1.2 When Should Floating Point Be Used?

CHAPTER 2 – FLOATING POINT FORMATS

- 2.1 Commonly Used Floating Point Bases
- 2.2 Comparisons of the Three Commonly Used Bases
- 2.3 Different Exponent Formats
- 2.4 "Implied 1"

CHAPTER 3 – FLOATING POINT ARITHMETIC

- 3.1 Introduction
- 3.2 Floating Point Add and Subtract
- 3.3 Floating Point Multiply
- 3.4 Floating Point Divide

CHAPTER 4 – DATA CONVERSION

- 4.1 Introduction
- 4.2 Binary Fixed Point to Floating Point
- 4.3 Floating Point to Binary Fixed Point
- 4.4 Decimal to Binary Floating Point Conversion
- 4.5 Binary to Decimal Floating Point Conversion

CHAPTER 5 – SINGLE-CHIP FLOATING POINT PROCESSORS

- 5.1 Introduction
- 5.2 Am9511A Arithmetic Processor
- 5.3 Am9512 Floating Point Processor

CHAPTER 6 – SOME INTERFACE EXAMPLES

- 6.1 Introduction
- 6.2 Am9080A to Am9511A Interface
- 6.3 Am9080A to Am9512 Interface
- 6.4 Am8085A to Am9511-1 Interface
- 6.5 Am8085A to Am9512-1 Interface
- 6.6 Z80 to Am9511A Interface
- 6.7 Z80 to Am9512 Interface
- 6.8 MC6800 to Am9511A Interface
- 6.9 MC6800 to Am9512 Interface
- 6.10 AmZ8002 to Am9511A Interface
- 6.11 AmZ8002 to Am9512 Interface

CHAPTER 7 – Am9511A INTERFACE METHODS

- 7.1 Introduction
- 7.2 Demand/Wait
- 7.3 Poll Status
- 7.4 Interrupt Driven
- 7.5 DMA Transfer

CHAPTER 8 – FLOATING POINT EXECUTION TIMES

- 8.1 Introduction
- 8.2 Floating Point Add/Subtract Execution Times
- 8.3 Floating Point Multiply/Divide Execution Times
- 8.4 Double-Precision Floating Point Execution Times

CHAPTER 9 – TRANSCENDENTAL FUNCTIONS OF Am9511A

- 9.1 Introduction
- 9.2 Chebyshev Polynomials
- 9.3 The Function CHEBY and ENTIER
- 9.4 Sine
- 9.5 Cosine
- 9.6 Tangent
- 9.7 Arcsine
- 9.8 Arccosine
- 9.9 Arctangent
- 9.10 Exponentiation
- 9.11 Natural Logarithm
- 9.12 Logarithm to Base 10 (Common Logarithm)
- 9.13 X to the Power of Y

TABLE OF CONTENTS (Cont.)

- 9.14 Square Root
- 9.15 Derived Function Error Performance

REFERENCES

- APPENDIX A. Am9511A DATA SHEET
- APPENDIX B. Am9512 DATA SHEET

ILLUSTRATIONS

FIG. TITLE

- 3.1 Floating Point Add/Subtract Flowchart
- 3.2 Floating Point Multiply Flowchart
- 3.3 Floating Point Divide Flowchart
- 4.1 Fix To Float Conversion Flowchart
- 4.2 Float To Fix Conversion Flowchart
- 4.3 Fix To Float/Float To Fix Conversion Subroutines
- 4.4 Decimal To Binary Floating Point Conversion Flowchart
- 4.5 Decimal to Binary Floating Point Conversion Programs
- 4.6 Binary To Decimal Floating Point Conversion Flowchart
- 4.7 Binary To Decimal Floating Point Conversion Programs
- 6.1 Am9080A To Am9511A Interface
- 6.2 Am9080A To Am9512 Interface
- 6.3 Am8085A To Am9511-1 Interface
- 6.4 Am8085A To Am9512-1 Interface
- 6.5 Z80 To Am9511A Interface
- 6.6 Z80 To Am9512 Interface
- 6.7 MC6800 To Am9511A Interface
- 6.8 MC6800 To Am9512 Interface
- 6.9 AmZ8002 To Am9511A Interface
- 6.10 AmZ8002 To Am9512 Interface
- 7.1 Demand/Wait Programming
- 7.2 Status Poll Programming Interface
- 7.3 Interrupt Driven Programming
- 7.4 DMA Interface Programming
- 7.5 High-Performance Configuration
- 9.1 Tangent
- 9.2 e^x
- 9.3 Natural Logarithm
- 9.4 Square Root
- 9.5 Square Root Computation

TABLES

- 8.1 Am9511A vs LLL BASIC Floating Point Add/Subtract Execution Time Comparison
- 8.2 Am9512 vs Intel FPAL LIB Floating Point Add/Subtract Execution Time Comparison
- 8.3 Am9511A vs LLL BASIC Floating Point Multiply/Divide Execution Time Comparison
- 8.4 Am9512 vs Intel FPAL LIB Floating Point Multiply/Divide Execution Time Comparison
- 8.5 Am9512 Double Precision Add/Subtract Execution Times
- 8.6 Am9512 Double Precision Multiply/Divide Execution Times

CHAPTER 2 FLOATING POINT FORMATS

2.1 COMMONLY USED FLOATING POINT BASES

The following three number bases are commonly used in floating point number systems:

- 1) Binary – The base is 2.
- 2) Binary Code Decimal (BCD) – The base is 10.
- 3) Hexadecimal – The base is 16.

2.2 COMPARISONS OF THE THREE COMMONLY USED BASES

Binary –

The main advantages of the binary floating point format are relative ease of hardware implementation and maximum accuracy for a given number of bits. On the negative side, the conversion of an ASCII (American Standard Code for Information Interchange) decimal string to and from a binary floating number is difficult and time consuming. In commercial applications where input and output are always decimal character strings, the binary floating point numbers will have an inherent rounding error because numbers such as 0.1_{10} cannot be represented exactly with a binary floating point number.

BCD –

The advantages and disadvantages of the BCD floating point numbers are just the opposite of the binary floating point numbers. BCD floating point is most commonly used in commercial applications where the computations involved are usually simple and input/output is always in the form of decimal ASCII strings.

Hexadecimal –

The hexadecimal floating point numbers have similar advantages and disadvantages as the binary floating point when compared with the BCD floating point format. When the same number of bits of exponent and mantissa are used, the hexadecimal floating point gives a considerably larger dynamic range than the binary floating point format. For example, for a 7-bit exponent, the largest positive number that can be represented in the hexadecimal floating point is approximately 16^{64} (approximately 1.16×10^{77}). The smallest non-zero positive number that can be represented is 16^{-64} (approximately 8.64×10^{-78}). By comparison, the largest and smallest positive numbers that can be represented in a 7-bit exponent binary system are approximately 1.84×10^{19} and 5.42×10^{-20} respectively.

An advantage of the hexadecimal floating point system over the binary point system is that during normalization and denormalization of the floating point numbers the hexadecimal system requires far fewer shifts compared with the binary system, because the hexadecimal system shifts four places at a time and most binary systems shift only one place at a time. For more sophisticated systems where normalization and denormalization can be done in one operation, this advantage does not exist. Most present-day systems do not fall in this category.

This disadvantage of the hexadecimal system is the loss of precision as compared with the binary system when the number of mantissa bits are the same. Since the three most significant bits could be zero when the first digit of the hexadecimal is a 1, this leads to a loss of 3 bits of accuracy in the worst case. However, assuming uniform distribution of numbers, the average loss of accuracy is only 11/15 bits. The above comparison assumes the binary system does not use an "implied 1" (Section 2.4). The loss of accuracy in a hexadecimal system compared with a binary system using an "implied 1" and same number of bits of mantissa is 4 bits in the worst case and 1 and 11/15 bits on the average.

2.3 DIFFERENT EXPONENT FORMATS

Two types of exponents used in floating point number systems are the biased exponent and the unbiased exponent. An unbiased exponent has a two's complement number. An exponent said to be biased by N (or excess N notation), means that the coded exponent is formed by adding N to the actual exponent in two's complement form. Any overflow generated from the addition is ignored. The result becomes an unsigned number. Most common floating point systems use a biased exponent. Biased exponents are used to simplify floating point hardware. During floating point computations, arithmetic operations such as add and subtract need to be performed on the exponents of the operands. If a biased exponent is used, the arithmetic logic unit (ALU) needs only to perform unsigned arithmetic. If an unbiased exponent is used, the ALU must perform two's complement arithmetic, and overflow conditions are more difficult to detect.

2.4 "IMPLIED 1"

Most floating point numbers must always be presented to the computer in "normalized" form (i.e., the most significant digit of the mantissa is always non-zero, except if the number is zero). For a binary floating point system, this would mean the leading binary bit of the mantissa is always 1 (except when the number is zero). In some floating point number systems, such as Am9512 format, this 1 bit is not represented on input or output to the floating point processor. The extra bit can be used for one more bit of precision or one more bit of exponent range.

CHAPTER 3 FLOATING POINT ARITHMETIC

3.1 INTRODUCTION

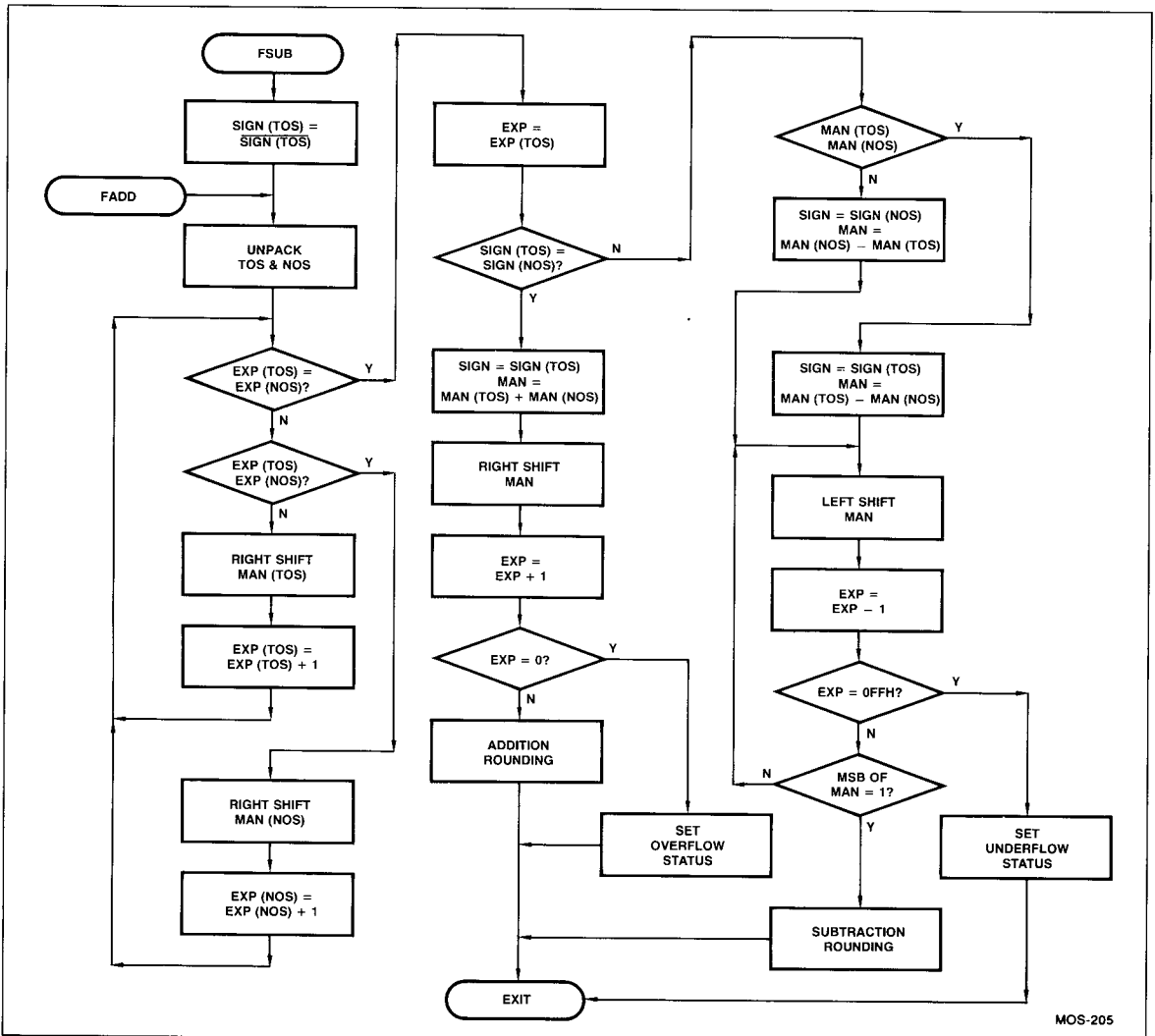
This chapter describes the basic principles of performing arithmetic with floating point numbers. First, the internal mechanism of floating point is analyzed. The following discussion uses the Am9512 single precision format although the discussion can apply to other formats with only minor modifications. The operands are assumed to be located in a stack. The first operand is called TOS (top of stack) and the second operand is called NOS (next on stack).

3.2 FLOATING POINT ADD AND SUBTRACT

Floating point add and subtract use essentially the same algorithm. The only difference is that floating point subtract changes the sign of the floating point number at top of stack and then performs the floating point add.

The following is a step-by-step description of a floating point add algorithm (Figure 3.1):

- a. Unpack TOS and NOS.
- b. The exponent of TOS is compared to the exponent of NOS.
- c. If the exponents are equal, go to step f.
- d. Right-shift the mantissa of the number with the smaller exponent.
- e. Increment the smaller exponent and go to step b.
- f. Set sign of result to sign of larger number.
- g. Set exponent of result to exponent of larger number.
- h. If sign of the two numbers are not equal, go to m.
- i. Add mantissas.
- j. Right-shift resultant mantissa by 1 and increment exponent of result by 1.



MOS-205

Figure 3.1. Floating Point Add/Subtract Flowchart

- k. If the most significant bit (MSB) of exponent changes from 1 to 0 as a result of the increment, set overflow status.
- l. Round if necessary and exit.
- m. Subtract smaller mantissa from larger mantissa.
- n. Left-shift mantissa and decrement exponent of result.
- o. If MSB of exponent changes from 0 to 1 as a result of the decrement, set underflow status and exit.
- p. If the MSB of the resultant mantissa = 0, go to n.
- q. Round if necessary and exit.

- d. Convert EXP (TOS) and EXP (NOS) to unbiased form:
 $EXP(TOS) = EXP(TOS) - 127_{10}$
 $EXP(NOS) = EXP(NOS) - 127_{10}$
- e. Add exponents:
 $EXP = EXP(TOS) + EXP(NOS)$
- f. If MSB of EXP (TOS) = MSB of EXP (NOS) = 0 and MSB of EXP = 1, then set overflow status and exit.
- g. If MSB of EXP (TOS) = MSB of EXP (NOS) = 1 and MSB of EXP = 0, then set underflow status and exit.
- h. Convert exponent back to biased form:
 $EXP = EXP + 127_{10}$
- i. If sign of TOS = sign of NOS, set sign of result to 0; otherwise, set sign of result to 1.
- j. Multiply mantissas.
- k. If MSB of resultant mantissa = 1, right-shift mantissa by 1 and increment exponent of resultant.
- l. If MSB of exponent changes from 1 to 0 as a result of the increment, set overflow status.
- m. Round if necessary and exit.

3.3 FLOATING POINT MULTIPLY

Floating point multiply basically involves the addition of the exponents and multiplication of the mantissas. The following is a step-by-step description of a floating point multiplication algorithm (Figure 3.2):

- a. Check if TOS or NOS = 0.
- b. If either TOS or NOS = 0, Set result to 0 and exit.
- c. Unpack TOS and NOS.

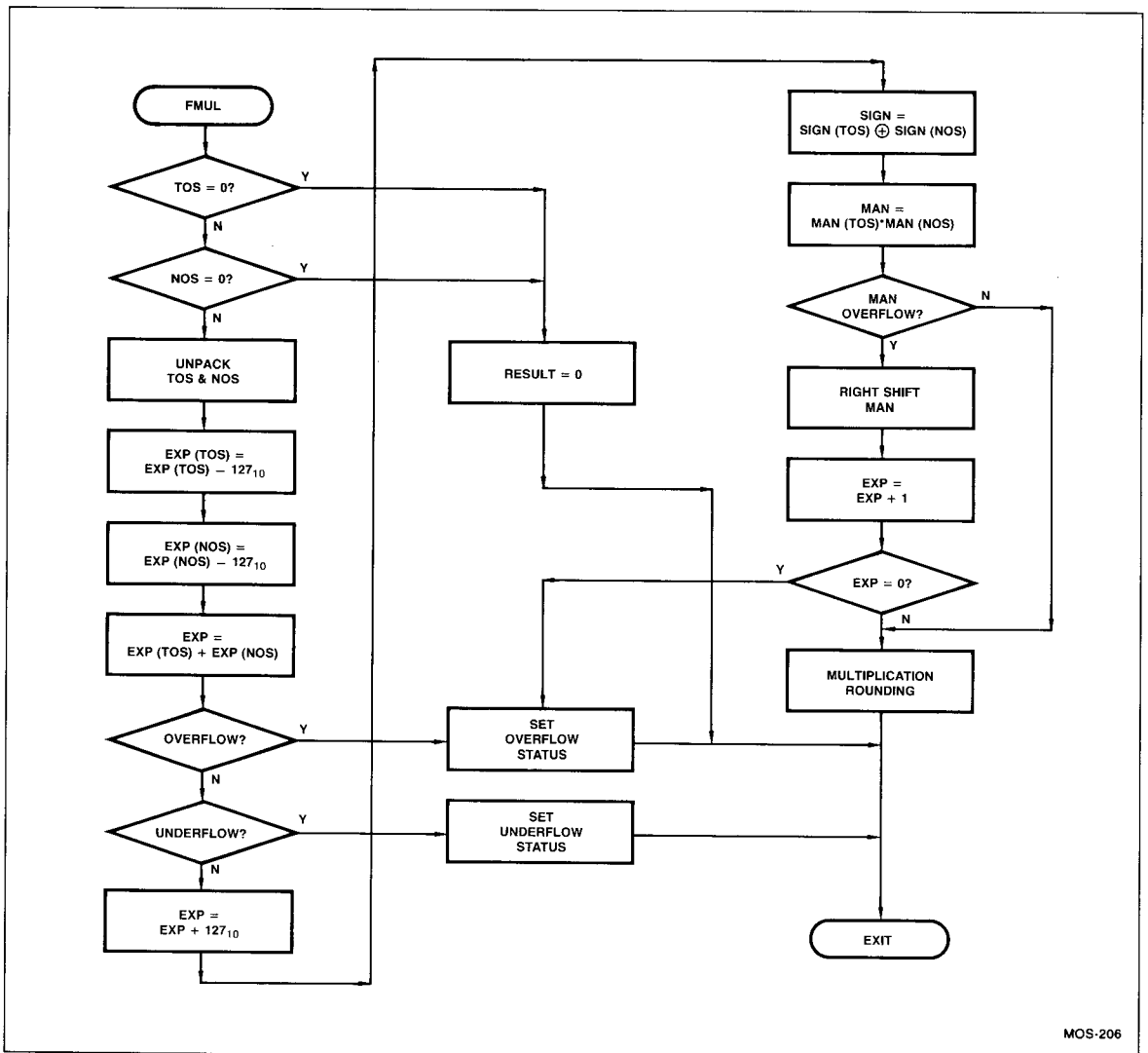


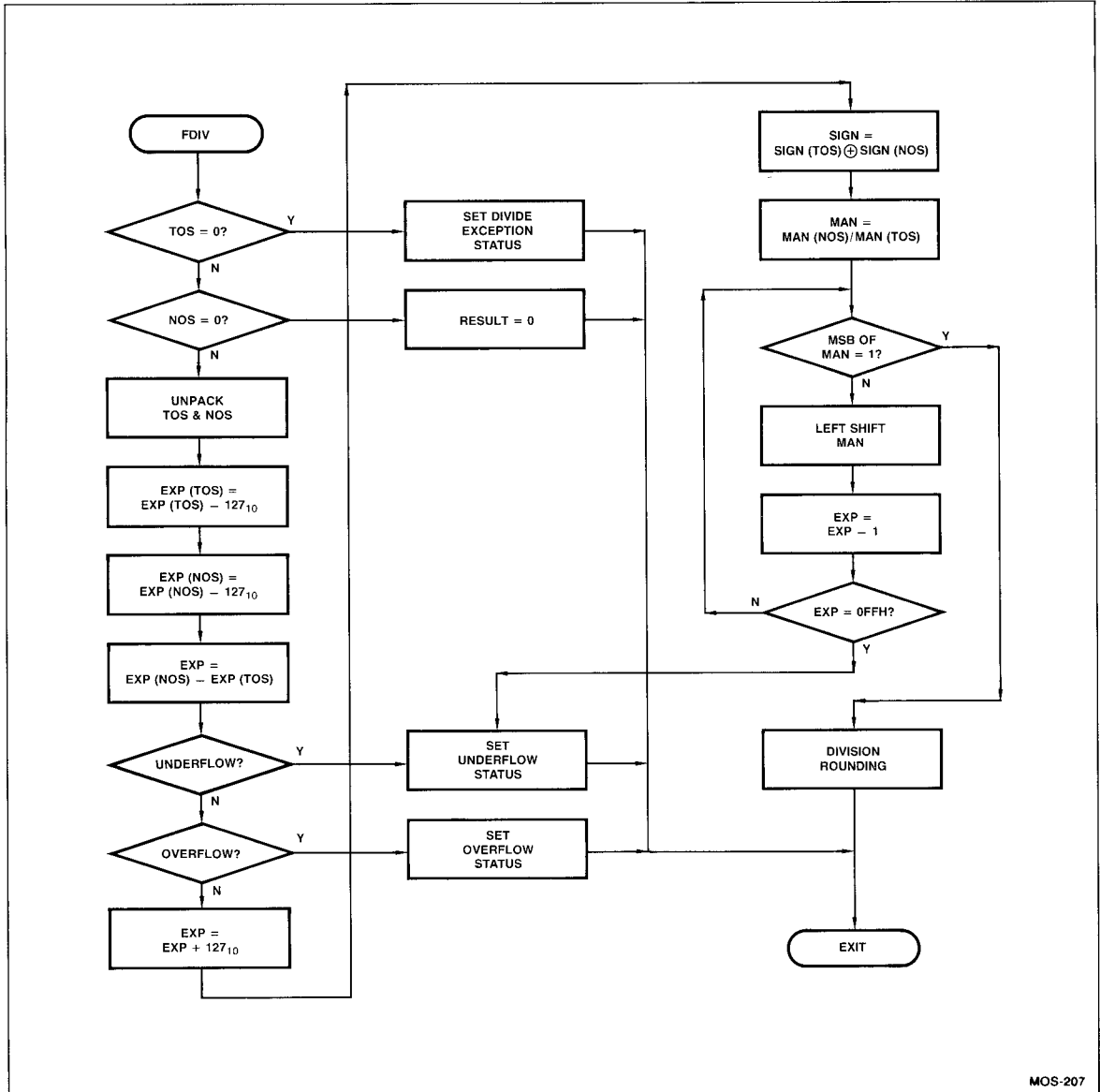
Figure 3.2. Floating Point Multiply Flowchart

3.4. FLOATING POINT DIVIDE

The floating point divide basically involves the subtraction of exponents and the division of mantissas. The following is a step-by-step description of a division algorithm (Figure 3.3):

- If TOS = 0, set divide exception error and exit.
- If NOS = 0, set result to 0 and exit.
- Unpack TOS and NOS.
- Convert EXP (TOS) and EXP (NOS) to unbiased form:
 $EXP(TOS) = EXP(TOS) - 127_{10}$
 $EXP(NOS) = EXP(NOS) - 127_{10}$
- Subtract exponent of TOS from exponent of NOS:
 $EXP = EXP(NOS) - EXP(TOS)$
- If MSB of EXP (NOS) = 0, MSB of EXP (TOS) = 1, and MSB of EXP = 1, then set overflow status and exit.

- If MSB of EXP (NOS) = 1, MSB of EXP (TOS) = 0, and MSB of EXP = 0, then set underflow status and exit.
- Add bias to exponent of result:
 $EXP = EXP + 127_{10}$
- If sign of TOS = sign of NOS, set sign of result to 0, else set sign of result to 1.
- Divide mantissa of NOS by mantissa of TOS
- If MSB = 0, left-shift mantissa and decrement exponent of resultant, or else go to n.
- If MSB of exponent changes from 0 to 1 as a result of the decrement, set underflow status.
- Go to k.
- Round if necessary and exit.



MOS-207

Figure 3.3. Floating Point Divide Flowchart

CHAPTER 4 DATA CONVERSION

4.1 INTRODUCTION

This chapter describes how to convert fixed point binary integer to floating point, floating point to fixed point binary integer, decimal ASCII (American Standard Code for Information Interchange) string to floating point and floating point to decimal ASCII string. These conversion methods are useful because few real-world inputs and outputs are in floating point format. When human interface is involved, the real-world interface is usually a decimal ASCII string. If the data are collected through some automatic means such as an A/D converter, counters, etc., the input is usually in the form of fixed point binary or BCD integers. In this chapter, the floating point format is assumed to be the Am9512 single precision format.

4.2 BINARY FIXED POINT TO FLOATING POINT

The input to this routine is assumed to be a 32-bit two's complement number and the output is a binary floating point number of

Am9512 format. Figure 4.1 shows the flow chart of such a program and Figure 4.2 shows an Am9080A assembly language subroutine that accomplishes this task.

The data format used in the assembly language conversion is as follows:

Fixed Point –

Two's complement number that occupies 4 consecutive memory locations with the most significant byte residing in low memory. To address the number, the pointer points to the low address.

Floating Point –

Am9512 floating point format that occupies 4 consecutive memory locations. The sign and 7 bits of the exponent resides in the low address. To address the number, the pointer points to the low address.

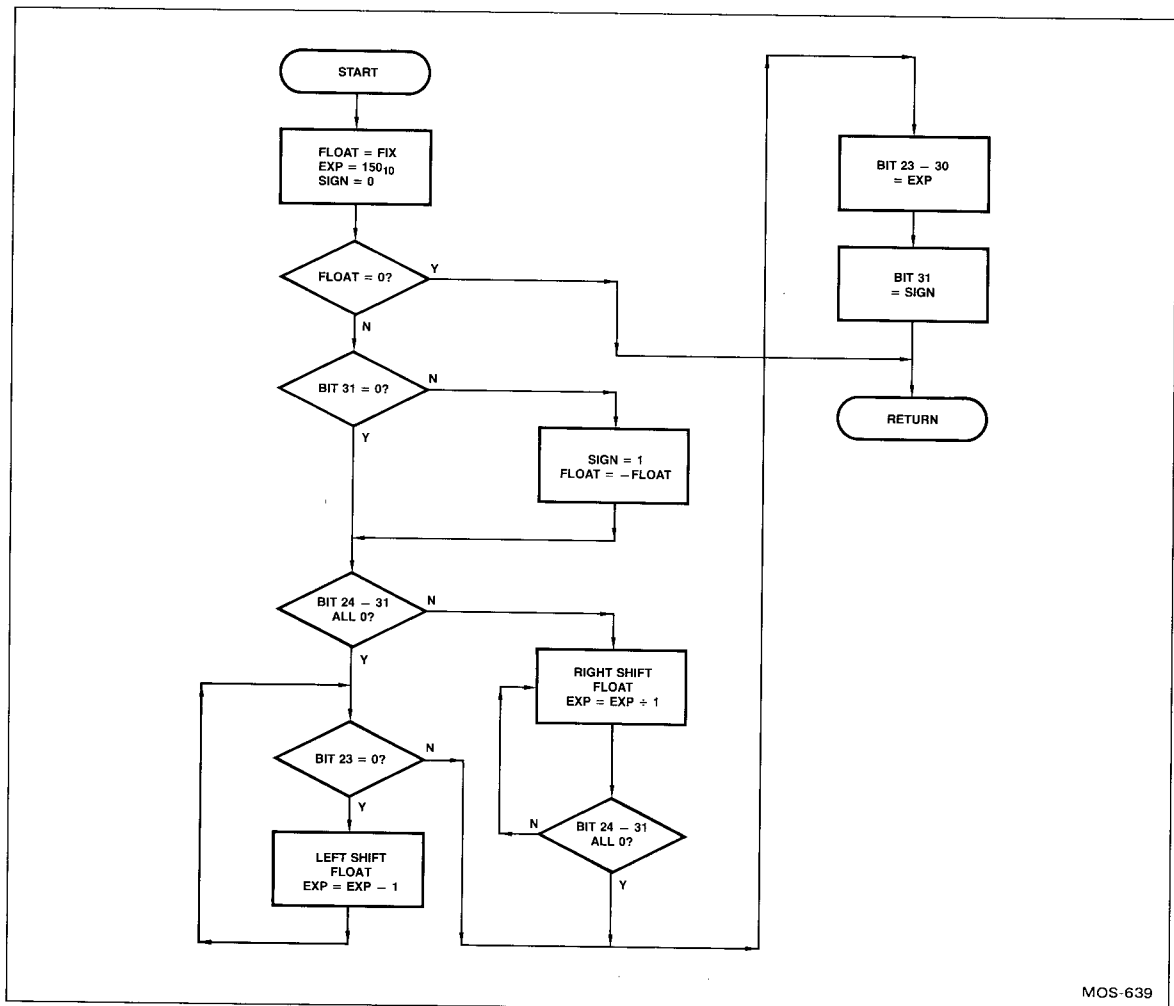


Figure 4.1. Fix to Float Conversion Flowchart

```

LOC  OBJ          LINE    SOURCE STATEMENT
                                1 $    PAGEWIDTH(80)  MACROFILE
                                2 ;
                                3 ;    *****
                                4 ;
                                5 ;    SUBROUTINES TO CONVERT FIX TO FLOAT
                                6 ;    AND FLOAT TO FIX POINT FORMATS
                                7 ;
                                8 ;    *****
                                9 ;
                               10 ;    NAME CONV
                               11 ;
                               12 ;    PUBLIC FXTOFL,FLTOFX
                               13 ;
                               14 ;    EXTRN QMOVE,QTEST,QNEG,QLSL,QLSR,QCLR
                               15 ;
                               16 ;    CSEG PAGE
                               17 ;
                               18 ;    FIX TO FLOAT CONVERSION ROUTINE
                               19 ;    TO CALL THE PROGRAM,
                               20 ;    HL = POINTER TO THE FIXED POINT NUMBER
                               21 ;    DE = POINTER TO THE FLOATING POINT NUMBER
                               22 ;    ACC AND PSW ARE ALTERED BY THE SUBROUTINE
                               23 ;    ALL OTHER REGISTERS ARE NOT DISTURBED
                               24 ;
0000  C5          25  FXTOFL:  PUSH B          ;SAVE BC REGISTER PAIR
0001  D5          26          PUSH D          ;SAVE DESTINATION POINTER
0002  E5          27          PUSH H          ;SAVE SOURCE POINTER
0003  CD0000     E  28          CALL QMOVE     ;COPY FIXED PT NO. INTO FLOAT
0006  EB          29          XCHG          ;PUT FLOAT POINTER IN HL
0007  CD0000     E  30          CALL QTEST    ;TEST IF NO. = 0?
000A  CA4D00     C  31          JZ RETN      ;YES - JUMP
                               32 ;
                               33 ;    THE NUMBER IS NOT ZERO, INIT. SIGN AND EXP
                               34 ;
000D  0600       35          MVI B,0      ;P REG = SIGN
000F  0E96       36          MVI C,23+127 ;C REG = EXPONENT + BIAS
                               37 ;
                               38 ;    TEST IF THE NUMBER IS NEGATIVE
                               39 ;
0011  7E         40          MOV A,M      ;GET MSB FROM FLOAT
0012  B7         41          ORA A      ;SET FLAGS
0013  F21B00     C  42          JP FX10     ;JUMP IF NO. IS POSITIVE
                               43 ;
                               44 ;    THE FIXED POINT NUMBER IS NEGATIVE
                               45 ;    NEGATE NUMBER AND SET SIGN = 1
                               46 ;
0016  0680       47          MVI B,80H    ;SET SIGN TO 80H
0018  CD0000     E  48          CALL QNEG    ;NEGATE NUMBER IN FLOAT
                               49 ;
                               50 ;    TEST IF MOST SIGNIFICANT BYTE OF FLOAT = 0
                               51 ;
001E  7E         52  FX10:  MOV A,M      ;GET MSB OF FLOAT
001C  B7         53          ORA A      ;SET FLAGS
001D  CA2C00     C  54          JZ FX20     ;JUMP IF MSB = 0

```

Figure 4.2. Float to Fix Conversion Flowchart

LOC	OBJ	LINE	SOURCE STATEMENT
		55 ;	
		56 ;	MSB NOT ZERO, RIGHT SHIFT REQUIRED
		57 ;	
0020	0C	58	FX15: INR C ;INC. EXP BY 1
0021	CD0000	59	CALL QISR ;LOGICAL SHIFT RIGHT OF FLOAT
0024	7E	60	MOV A,M ;TEST IF MSB = 0
0025	B7	61	ORA A ;SET FLAGS
0026	C22000	62	JNZ FX15 ;NOT ZERO, SHIFT SOME MORE
0029	C33B00	63	JMP FX30 ;ZERO, SHIFT COMPLETE
		64 ;	
		65 ;	MSB = 0, TEST IF LEFT SHIFT REQUIRED
		66 ;	
002C	54	67	FX20: MOV D,H
002D	5D	68	MOV E,L ;PUT FLOAT POINTER INTO DE
002F	13	69	INX D ;POINT TO NEXT MSB OF FLOAT
002F	1A	70	FX25: IDAX D ;GET NEXT MSB
0030	F7	71	ORA A ;SET FLAGS
0031	FA3B00	72	JM FX30 ;DONE IF BIT 23 = 1
0034	0D	73	DCR C ;DEC. EXP BY 1
0035	CD0000	74	CALL QISL ;LOGICAL LEFT SHIFT OF FLOAT
0038	C32F00	75	JMP FX25 ;TRY AGAIN
		76 ;	
		77 ;	SHIFT COMPLETE, MANTISSA FORMED IN FLOAT
		78 ;	
003B	1A	79	FX30: LDAX D ;GET NEXT MSB OF FLOAT
003C	E67F	80	ANI 7FH ;STRIP OFF HIDDEN "1"
003E	12	81	STAX D ;PUT IT BACK IN MEMORY
003F	79	82	MOV A,C ;GET EXPONENT
0040	0F	83	PRC ;ROTATE RIGHT
0041	4F	84	MOV C,A ;PUT ROTATED EXP. BACK IN C
0042	E680	85	ANI 80H ;EXTRACT LSB OF EXPONENT
0044	EB	86	XCHG ;PUT NEXT MSB POINTER IN HL
0045	B6	87	ORA M ;COMBINE MSB OF MANTISSA WITH EX
		P	
0046	77	88	MOV M,A
0047	EB	89	XCHG ;RESTORE POINTERS
0048	79	90	MOV A,C ;GET ROTATED EXPONENT
0049	E67F	91	ANI 7FH ;STRIP OF LSB
004B	B0	92	ORA B ;COMBINE EXP WITH SIGN
004C	77	93	MOV M,A ;SET MSB OF FLOAT
		94 ;	
		95 ;	CONVERSION COMPLETE, RETURN TO CALLER
		96 ;	
004D	E1	97	RETN: POP H ;RESTORE ALL REGISTERS
004E	D1	98	POP D
004F	C1	99	POP B
0050	C9	100	RET ;RETURN TO CALLER
		101 ;	
		102 ;	FLOAT TO FIX CONVERSION ROUTINE
		103 ;	TO CALL THE PROGRAM
		104 ;	HL = POINTER TO THE FLOATING POINT NUMBER
		105 ;	DE = POINTER TO THE FIXED POINT NUMBER
		106 ;	ON RETURN
		107 ;	A REG = 0 AND Z FLAG = 1 IF NO ERROR
		108 ;	A = 1 AND Z FLAG = 0 IF OVERFLOW ERROR

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
		109 ;	OTHER REGISTERS ARE NOT DISTURBED
		110 ;	
0051	C5	111	FLTOFX: PUSH B ;SAVE ALL REGISTERS
0052	D5	112	PUSH D
0053	E5	113	PUSH H
0054	CD0000	E 114	CALL QMOVE ;COPY FLOAT TO FIX
0057	CD0000	E 115	CALL QTEST ;TEST IF INPUT NO. = 0?
005A	CAA200	C 116	JZ FL40 ;RETURN IF INPUT IS ZERO
		117 ;	
		118 ;	EXTRACT SIGN AND EXPONENT FROM FLOATING PT NO.
		119 ;	
005D	EB	120	XCHG ;HL POINTS TO FIX
005E	7E	121	MOV A,M ;GET MSB
005F	E680	122	ANI 80H ;EXTRACT SIGN BIT
0061	47	123	MOV B,A ;SAVE SIGN IN B
0062	7E	124	MOV A,M ;GET MSB AGAIN
0063	07	125	RLC ;MULTIPLY BY 2
0064	E6FE	126	ANI 0FEH ;STRIP OF LSB
0066	4F	127	MOV C,A ;SAVE IN C
0067	23	128	INX H ;POINT TO NEXT MSB
0068	7E	129	MOV A,M ;GET NEXT MSB
0069	07	130	RLC ;MOVE LSB OF EXP INTO CARRY
006A	D26E00	C 131	JNC \$+4 ;SKIP IF NO CARRY
006D	0C	132	INR C ;PROPAGATE CARRY INTO EXP
006E	7E	133	MOV A,M ;GET NEXT MSB
006F	F680	134	ORI 80H ;SET HIDDEN BIT
0071	77	135	MOV M,A ;RESTORE NEXT MSB
0072	2B	136	ECX H ;NOW HL POINTS TO MSB AGAIN
0073	3600	137	MVI M,0 ;CLEAR MSB
0075	79	138	MOV A,C ;GET BIASED EXPONENT
0076	D67F	139	SUI 127 ;STRIP OFF BIAS
0078	FAA700	C 140	JM ZERO ;EXP < 0, RETURN ZERO AS RESULT
007B	FE1F	141	CPI 31 ;CHECK IF EXP > 31
007D	D2AD00	C 142	JNC OVFL ;JUMP IF NUMBER IS TOO LARGE
0080	D617	143	SUI 23 ;SUBTRACT EXP BY 23
0082	CA9A00	C 144	JZ FL30 ;NO SHIFT REQUIRED, CHECK SIGN
0085	4F	145	MOV C,A ;SAVE SHIFT COUNT
0086	DA9300	C 146	JC FL20 ;COUNT < 0, RIGHT SHIFT
		147 ;	
		148 ;	COUNT > 0, LEFT SHIFT REQUIRED
		149 ;	
0089	CD0000	E 150	FL10: CALL QLSL ;LOGICAL SHIFT LEFT
008C	0D	151	DCR C
008D	C28900	C 152	JNZ FL10
0090	C39A00	C 153	JMP FL30
		154 ;	
		155 ;	COUNT < 0, RIGHT SHIFT REQUIRED
		156 ;	
0093	CD0000	E 157	FL20: CALL QLSR ;LOGICAL SHIFT RIGHT
0096	0C	158	INR C
0097	C29300	C 159	JNZ FL20
		160 ;	
		161 ;	SHIFT COMPLETE, CHECK SIGN AND EXIT
		162 ;	
009A	78	163	FL30: MOV A,B ;GET SIGN

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
009E	B7	164	ORA A ;SET FLAGS
009C	F2A200	C 165	JP FL40 ;PLUS SIGN, SKIP NEGATION
009F	CD0000	E 166	CALL QNEG ;MINUS SIGN, NEGATE NUMBER
		167 ;	
		168 ;	CLEAR ERROR FLAG AND RETURN
		169 ;	
00A2	AF	170 FL40:	XRA A
00A3	E1	171	POP H ;RESTORE ALL REGISTERS
00A4	D1	172	POP D
00A5	C1	173	POP B
00A6	C9	174	RET
		175 ;	
		176 ;	ZERO FIX POINT NUMBER AND RETURN
		177 ;	
00A7	CD0000	E 178 ZERO:	CALL QCLR ;CLEAR FIX POINT NUMBER
00AA	C3A200	C 179	JMP FL40 ;RETURN
		180 ;	
		181 ;	SET OVERFLOW FLAG AND RETURN
		182 ;	
00AD	3E01	183 OVFL:	MVI A,1 ;SET A REG
00AF	B7	184	ORA A ;SET Z FLAG
00B0	C3A300	C 185	JMP FL40+1 ;RESTORE REG. AND RETURN
		186	END

PUBLIC SYMBOLS

FLTOFX C 0051 FXTOFL C 0000

EXTERNAL SYMBOLS

QCLR E 0000 QLSL F 0000 QLSR E 0000 QMOVE E 0000
 QNEG E 0000 QTEST E 0000

USER SYMBOLS

FL10 C 0089 FL20 C 0093 FL30 C 009A FL40 C 00A2
 FLTOFX C 0051 FX10 C 001B FX15 C 0020 FX20 C 002C
 FX25 C 002F FX30 C 003B FXTOFL C 0000 OVFL C 00AD
 QCLR E 0000 QLSL E 0000 QLSR E 0000 QMOVE E 0000
 QNEG E 0000 QTEST E 0000 RETN C 004D ZERO C 00A7

ASSEMBLY COMPLETE, NO ERRORS

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

```

LCC OBJ          LINE          SOURCE STATEMENT
1  §            1  PAGEWIDTH(80) MACROFILE
2  ;            2  ;
3  ;            3  ;*****
4  ;            4  ;
5  ;            5  QUADRUPLE PRECISION SUBROUTINES
6  ;            6  ;
7  ;            7  ;*****
8  ;            8  ;
9  ;            9  PUBLIC QMOVE,QTEST,QNEG,QLSL,QLSR,QCLR
10 ;           10 ;
11 ;           11 CSEG
12 ;           12 ;
13 ;           13 MOVE 4 BYTES POINTED TO BY HL
14 ;           14 TO 4 BYTES POINTED BY DE
15 ;           15 M(DE) = M(HL)
16 ;           16 ;
0000 C5         17 QMOVE:  PUSH B           ;SAVE ALL REGISTERS
0001 D5         18         PUSH D
0002 E5         19         PUSH H
0003 0604       20         MVI B,4
0005 7E         21 QM10:  MOV A,M           ;GET BYTE FROM M(HL)
0006 12         22         STAX D           ;STORE BYTE IN M(DE)
0007 23         23         INX H           ;BUMP SOURCE POINTER
0008 13         24         INX D           ;BUMP DESTINATION POINTER
0009 05         25         DCR B
000A C20500    C 26         JNZ QM10        ;UNTIL 4 TIMES
000D E1         27         POP H           ;RESTORE ALL REGISTERS
000E D1         28         POP D
000F C1         29         POP B
0010 C9         30         RET
31 ;           31 ;
32 ;           32 TEST 4 BYTES POINTED TO HL FOR 0
33 ;           33 M(HL) = 0?
34 ;           34 ;
0011 E5         35 QTEST:  PUSH H           ;SAVE HL
0012 7E         36         MOV A,M           ;GET FIRST BYTE
0013 23         37         INX H
0014 B6         38         ORA M           ;COMBINE WITH 2ND BYTE
0015 23         39         INX H
0016 B6         40         ORA M           ;COMBINE WITH 3RD BYTE
0017 23         41         INX H
0018 B6         42         ORA M           ;COMBINE WITH 4TH BYTE
0019 E1         43         POP H           ;RESTORE HL
001A C9         44         RET
45 ;           45 ;
46 ;           46 NEGATE THE QUAD PRECISION NUMBER POINTED TO BY H
L             47 M(HL) = - M(HL)
48 ;           48 ;
001B C5         49 QNEG:  PUSH B           ;SAVE BC
001C 23         50         INX H           ;MOVE HL TO LSB
001D 23         51         INX H
001E 23         52         INX H
001F 0604       53         MVI B,4

```

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
0021	B7	54	ORA A ;CLEAR CARRY
0022	3E00	55	QN10: MVI A,0 ;CLEAR A WITHOUT AFFECTING CARRY
0024	9E	56	SBB M
0025	77	57	MOV M,A
0026	2B	58	DCX H
0027	05	59	DCR B
0028	C22200	60	JNZ QN10
002B	23	61	INX H ;RESTORE HL
002C	C1	62	POP B ;RESTORE BC
002D	C9	63	RET
		64 ;	
		65 ;	
		66 ;	LOGICAL SHIFT LEFT 4 BYTES POINTED TO HL
		67 ;	M(HL) = LSL(M(HL))
002E	C5	68	QLSL: PUSH B ;SAVE BC
002F	23	69	INX H ;MOVE POINTED TO LSB
0030	23	70	INX H
0031	23	71	INX H
0032	0604	72	MVI B,4
0034	B7	73	ORA A ;CLEAR CARRY
0035	7E	74	QLSL10: MOV A,M
0036	17	75	RAL
0037	77	76	MOV M,A
0038	2B	77	DCX H
0039	05	78	DCR B
003A	C23500	79	JNZ QLSL10
003D	23	80	INX H ;RESTORE HL
003E	C1	81	POP B ;RESTORE BC
003F	C9	82	RET
		83 ;	
		84 ;	
		85 ;	LOGICAL RIGHT SHIFT OF 4 BYTES POINTED TO BY HL
		86 ;	M(HL) = LSR(M(HL))
0040	C5	87	QLSR: PUSH B ;SAVE BC
0041	E5	88	PUSH H ;SAVE HL
0042	0604	89	MVI B,4
0044	B7	90	ORA A ;CLEAR CARRY
0045	7E	91	QLSR10: MOV A,M
0046	1F	92	RAR
0047	77	93	MOV M,A
0048	23	94	INX H
0049	05	95	DCR B
004A	C24500	96	JNZ QLSR10
004D	E1	97	POP H ;RESTORE HL
004E	C1	98	POP B ;RESTORE BC
004F	C9	99	RET
		100 ;	
		101 ;	
		102 ;	CLEAR 4 BYTES POINTED TO BY HL
		103 ;	M(HL) = 0
0050	E5	104	QLCL: PUSH H
0051	AF	105	XRA A
0052	77	106	MOV M,A
0053	23	107	INX H
0054	77	108	MOV M,A

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

```

LOC  OBJ          LINE          SCURCE STATEMENT
0055  23          109          INX  H
0056  77          110          MOV  M,A
0057  23          111          INX  H
0058  77          112          MOV  M,A
0059  E1          113          POP  H
005A  C9          114          RET
                   115 ;
                   116          END

PUBLIC SYMBOLS
QCLR  C 0050      QLSL  C 002E      QLSR  C 0040      QMOVE C 0000
QNEG  C 001E      QTEST C 0011

EXTERNAL SYMBOLS

USER SYMBOLS
QCLR  C 0050      QLSL  C 002E      QLS10 C 0035      QLSR  C 0040
QLSR10 C 0045      QM10  C 0005      QMOVE C 0000      QN10  C 0022
QNEG  C 001B      QTEST  C 0011

ASSEMBLY COMPLETE, NO ERRORS

```

Figure 4.2. Float to Fix Conversion Flowchart (Cont.)

The following is a step-by-step description of the algorithm used in the conversion example:

- a. Copy the fixed point number into the location of the floating point number.
- b. Test the floating point number to see if it is zero.
- c. Return to caller if the number is zero.
- d. The sign is defaulted to 0 (plus).
- e. Default the actual exponent to 23. This is the exponent that would be valid if no shift is required, i.e., the most significant 1 is in bit position 23. Since the Am9512 format has a bias of 127_{10} the bias is added to the default value to make the default exponent $23_{10} + 127_{10} = 150_{10}$.
- f. If bit 31 in the floating point register = 1, then the input number is a negative number. The number in the floating point register is negated (two's complement negation) and the sign is set to 1.
- g. If bits 24-31 of the floating point register are all zeroes, then the input number has an exponent less than or equal 23. The program transfers to step j for possible left shifts. Otherwise the program falls through to h.
- h. Bits 24-31 are not all zeroes. This means the magnitude of the fixed point number is greater than 2^{23} . The floating point register is right-shifted one place and the exponent is incremented by 1.
- i. Test bits 24-31 again for all zeroes. If they are not all zeroes, repeat step h. If bits 24-31 are all zeroes, shifting is complete and the program transfers to step l.
- j. Bits 24-31 are all zero. If bits 23 = 1, no more shifting is required and the program transfers to step l.
- k. Left-shift floating point register. Decrement exponent by 1 and repeat step j.
- l. Shifting is complete. The exponent is stored into bits 23-30. (The original bit 23, the "hidden 1" is overwritten).
- m. Store the sign into bit 31 of the floating point register.
- n. Return to caller.

4.3 FLOATING POINT TO BINARY FIXED POINT

Figure 4.2 shows the flowchart of a floating point to fixed point conversion flowchart. An Am9080A assembly language subroutine that implements to flowchart is shown in Figure 4.3. The following is a step-by-step description of the algorithm:

- a. Copy the floating point number into the fixed point register.
- b. If the floating number is zero, return to caller.
- c. Unpack the floating point number from the fixed point register by removing the exponent and sign. The exponent (in the unbiased form) and the sign are stored in CPU registers. The "Hidden 1" is restored in the fixed point register.
- d. If exponent is less than 0, zero fixed point register and exit.
- e. If exponent is larger than 31, set overflow flag and exit.
- f. Subtract 23 from exponent to derive the shift count.
- g. If the adjusted exponent is greater than zero, the original

- exponent is greater than 23, the program transfers to step j to left shift fixed point register, or else it falls through to step h.
- h. If the exponent = 0, shift is complete and the program transfers to step l.
- i. Right-shift the fixed point number one position and increment the exponent by 1. Repeat step h.
- j. Left-shift the fixed point number by one position and decrement the exponent by 1.
- k. If the exponent is not zero, repeat step j; or else, the program falls through to step l.
- l. Test the original sign of the floating point number. If sign is positive skip step m.
- m. If the sign is negative, negate the number in the fixed point register (two's complement).
- n. Return to caller.

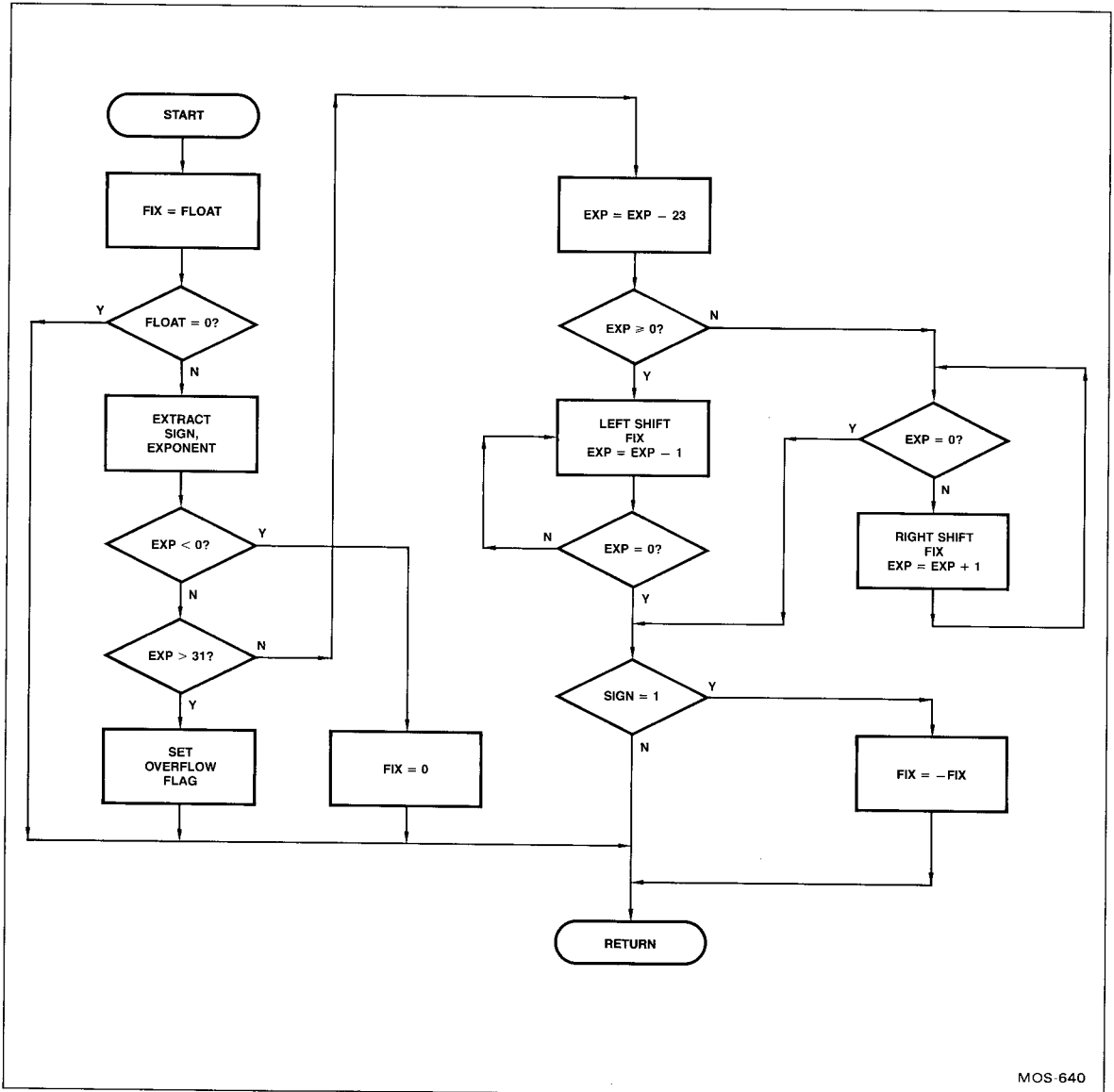


Figure 4.3. Fix to Float/Float to Fix Conversion Subroutines

4.4 DECIMAL TO BINARY FLOATING POINT CONVERSION

When a programmer works with binary floating point numbers, it is often necessary to convert decimal numbers into binary floating point notation to enter the desired numbers into the machine. Figure 4.4 shows the flowchart of such a conversion program and Figure 4.5 shows a BASIC program that does the conversion.

The program uses an array A of 32 elements. Each element of the array corresponds to one bit of the floating point number: A(31) is the sign bit, A(30) to A(23) represent the exponent and A(22) to A(0) represent the mantissa. Other variables used are as follows:

- D – The decimal number entered from console
- E – The exponent of the binary floating point number
- H – An index to the hexadecimal string with range 0-15
- HS – An ASCII string of all hexadecimal characters used for hexadecimal output

- I – An integer used for loop index
- J – A number used for comparison when unpacking the exponent and the mantissa
- M – The mantissa of the binary floating point number

The following equation converts a floating point number from one base to another:

- Let E_2 = Exponent of new number
- M_2 = Mantissa of new number
- B_2 = Base of new number
- N_1 = Original number

Given N_1 and B_2 , the equations used to solve E_2 and M_2 are:

$$E_2 = \text{INT} (\text{LOG} (N_1) / \text{LOG} (B_2))$$

$$M_2 = N_1 / (B_2^{**} E_2)$$

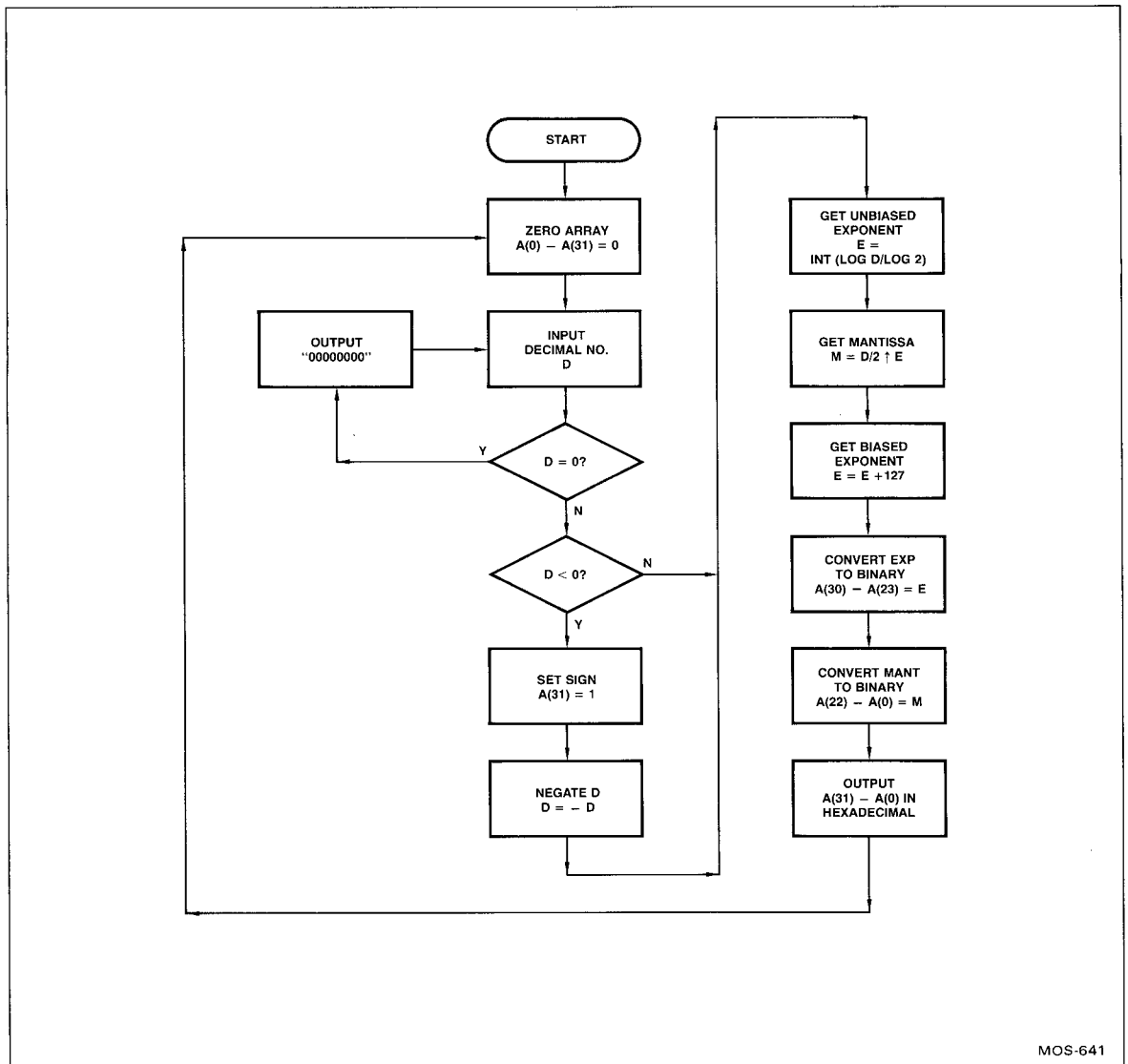


Figure 4.4. Decimal to Binary Floating Point Conversion Flowchart

```

10 REM
20 REM
30 DIM A(32)
40 H$ = "0123456789ABCDEF"
50 PRINT "INPUT DECIMAL NO. ";
60 INPUT D
70 REM CLEAR BINARY ARRAY
80 FOR I = 0 TO 31
90 A(I) = 0
100 NEXT I
110 IF D = 0 THEN 450
120 IF D < 0 THEN A(0) = 1
130 D = ABS(D)
140 REM FIND THE EXPONENT
150 E = INT(LOG(D)/LOG(2)) + 1
160 M = D/2^E
170 REM FORM BINARY ARRAY FOR EXPONENT
180 IF E < 1 THEN 250
190 J = 128
200 FOR I = 1 TO 7
210 J = J/2
220 IF E >= J THEN A(I) = 1 : E = E - J
230 NEXT I
240 GOTO 320
250 REM E IS LESS THAN 1
260 A(1) = 1
270 J = - 64
280 FOR I = 2 TO 7
290 J = J/2
300 IF E >= J THEN A(I) = 1 ELSE E = E - J
310 NEXT I
320 REM FORM BINARY ARRAY FOR MANTISSA
330 J = 1
340 FOR I = 8 TO 31
350 J = J/2
360 IF M >= J THEN A(I) = 1 : M = M - J
370 NEXT I
380 REM FORM HEXADECIMAL NUMBER AND OUTPUT IT
390 FOR I = 0 TO 31 STEP 4
400 H = 8*A(I) + 4*A(I+1) + 2*A(I+2) + A(I+3)
410 PRINT MID$(H$,H+1,1);
420 NEXT I
430 PRINT
440 GOTO 50
450 PRINT "00000000"
460 GOTO 50

```

a) Decimal String to Am9511A Floating Point Format

Figure 4.5. Decimal to Binary Floating Point Conversion Programs

```

10 REM
20 REM
30 REM
40 REM
50 DEFINT A,I,H
60 DIM A(32)
70 H$ = "0123456789ABCDEF"
80 REM
90 REM CLEAR BINARY ARRAY A(0) TO A(31)
100 REM
110 FOR I = 0 TO 31
120 A(I) = 0
130 NEXT I
140 REM
150 REM INPUT A DECIMAL NUMBER FROM CONSOLE
160 REM
170 PRINT
180 INPUT "ENTER DECIMAL NUMBER";D
190 REM
200 REM CHECK IF INPUT NUMBER IS ZERO
210 REM
220 IF D <> 0 THEN 280
230 PRINT "00000000"
240 GOTO 180
250 REM
260 REM INPUT IS NOT ZERO, CHECK IF IT IS NEGATIVE
270 REM
280 IF D < 0 THEN A(31) = 1 : D = -D
290 REM
300 REM FIND THE UNBIASED EXPONENT
310 REM
320 E = INT(LOG(D)/LOG(2))
330 REM
340 REM FIND THE MANTISSA
350 REM
360 M = D/2^E
370 REM
380 REM FIND THE BIASED EXPONENT
390 REM
400 E = E + 127
410 REM
420 REM FORM BINARY ARRAY FOR EXPONENT
430 REM
440 J = 256
450 FOR I = 30 TO 23 STEP - 1
460 J = J/2
470 IF E >= J THEN A(I) = 1 : E = E - J
480 NEXT I
490 REM
500 REM FORM BINARY ARRAY FOR MANTISSA
510 REM
520 M = M - 1 : REM STRIP OFF "HIDDEN 1"
530 J = 1
540 FOR I = 22 TO 0 STEP -1
550 J = J/2
560 IF M >= J THEN A(I) = 1 : M = M - J
570 NEXT I
580 REM
590 REM FORM HEXADECIMAL NUMBER AND OUPUT TO CONSOLE
600 REM
610 FOR I = 31 TO 0 STEP -4
620 H = 8*A(I) + 4*A(I-1) + 2*A(I-2) + A(I-3)
630 PRINT MID$(H$,H+1,1);
640 NEXT I
650 GOTO 110

```

b) Decimal String to Am9512 Floating Point Format

Figure 4.5. Decimal to Binary Floating Point Conversion Programs (Cont.)

```

10 REM
20 REM
30 REM
40 REM
50 DEFINT H,I,S : DIM H(8)
60 REM
70 REM   INPUT BINARY FLOATING POINT IN HEXADECIMAL
80 REM
90 INPUT "ENTER AN 8 DIGIT HEXADECIMAL NUMBER";H$
100 REM
110 REM   UNPACK HEXADECIMAL NUMBER INTO A BINARY ARRAY
120 REM
130 FOR I = 0 TO 7
140 C$ = MID$(H$,I+1,1)
150 H(I) = ASC(C$)
160 IF (H(I) < 48 OR H(I) > 70) THEN 530
170 IF (H(I) > 57 AND H(I) < 65) THEN 530
180 H(I) = H(I) - 48
190 IF H(I) > 9 THEN H(I) = H(I) - 7
200 NEXT I
210 REM
220 REM   FIND THE SIGN OF THE NUMBER
230 REM
240 S = 0
250 IF H(0) > 7 THEN S = 1
260 REM
270 REM   FIND THE EXPONENT OF THE NUMBER
280 REM
290 E = 32*(H(0) AND 7) + 2*H(1) + (H(2) AND 8)/8 - 127
300 REM
310 REM   FIND THE MANTISSA OF THE NUMBER
320 REM
330 H(2) = H(2) AND 7
340 M = 1
350 FOR I = 2 TO 7
360 M = M + H(I)/2^(3+4*(I-2))
370 NEXT I
380 REM
390 REM   FIND THE NUMBER BY COMBINING EXPONENT & MANTISSA
400 REM
410 N = (2^E) * M
420 REM
430 REM   CHECK SIGN TO SEE IF NEGATION REQUIRED
440 REM
450 IF S = 1 THEN N = -N
460 REM
470 REM   OUTPUT DECIMAL NUMBER
480 REM
490 PRINT N : GOTO 90
500 REM
510 REM   ILLEGAL INPUT DETECTED, ABORT
520 REM
530 PRINT "INPUT ERROR, UNKNOWN CHARACTER '" ; C$ ; "' : GOTO 90

```

b) Hexadecimal Floating Point

Figure 4.5. Binary to Decimal Floating Point Conversion Program

```

10 REM
20 REM
30 REM
40 DEFINIT A,I
50 DEFDBL B-H,J-Z
60 DIM A(64)
70 H$ = "0123456789ABCDEF"
80 INPUT "ENTER DECIMAL NUMBER";D
90 REM CLEAR BINARY ARRAY
100 FOR I = 0 TO 63
110 A(I) = 0
120 NEXT I
130 IF D = 0 THEN 540
140 IF D < 0 THEN A(0) = 1
150 D = ABS(D)
160 REM FIND THE UNBAISED EXPONENT
170 E = INT(LOG(D)/LOG(2))
180 REM USE ITERATIVE LOOP TO FIND 2^E BECAUSE
190 REM EXPONENTIATION IS NOT EXACT T = 2^E
200 T = 1
210 IF E = 0 THEN 320
220 IF E > 0 THEN 280
230 REM THE EXPONENT IS NEGATIVE
240 FOR I = -1 TO E STEP -1
250 T = T/2
260 NEXT I
270 GOTO 320
280 FOR I = 1 TO E
290 T = 2*T
300 NEXT I
310 REM FIND THE MANTISSA AND BIASED EXPONENT
320 M = D/T
330 E = E + 1023
340 REM FORM BINARY ARRAY FOR EXPONENT
350 J = 2048
360 FOR I = 1 TO 11
370 J = J/2
380 IF E >= J THEN A(I) = 1 : E = E - J
390 NEXT I
400 REM FORM BINARY ARRAY FOR MANTISSA
410 M = M - 1#
420 J = 1
430 FOR I = 12 TO 63
440 J = J/2
450 IF M >= J THEN A(I) = 1 : M = M - J
460 NEXT I
470 REM FORM HEXADECIMAL NUMBER AND OUTPUT IT
480 FOR I = 0 TO 63 STEP 4
490 H = 8*A(I) + 4*A(I+1) + 2*A(I+2) + A(I+3)
500 PRINT MID$(H$,H+1,1);
510 NEXT I
520 PRINT
530 GOTO 80
540 PRINT "0000000000000000"
550 GOTO 80

```

c) Decimal String to Am9512 Floating Point - Double Precision Format

Figure 4.5. Decimal to Binary Floating Point Conversion Programs (Cont.)

```

10 REM
20 REM
30 DEFDBL A-G,K-Z
35 DEFINT I,J
40 DIM C(16)
50 INPUT "INPUT 16 DIGIT HEXADECIMAL NUMBER ";H$
60 REM UNPACK HEXADECIMAL NUMBER INTO A BINARY ARRAY
70 FOR I = 0 TO 15
80 C$ = MID$(H$,I+1,1)
90 C(I) = ASC(C$) - 48
100 IF C(I) < 0 THEN 290
110 IF C(I) > 10 THEN C(I) = C(I) - 7
120 IF C(I) > 15 THEN 290
130 NEXT I
140 REM FIND SIGN OF NUMBER
150 S = 0
160 IF C(0) > 7 THEN S = 1
170 REM FIND EXPONENT OF NUMBER
180 E = 256*(C(0) AND 7) + 16*C(1) + C(2) - 1023
190 REM FIND MANTISSA OF NUMBER
200 C(2) = C(2) AND 7
210 M = 1
220 FOR I = 3 TO 15
230 M = M + C(I)/2^(4*(I-2))
240 NEXT I
250 N = (2^E) * M
260 IF S = 1 THEN N = -N
270 PRINT N
280 GOTO 50
290 PRINT "INPUT ERROR"
300 GOTO 50

```

c) Double Precision Decimal Number

Figure 4.5. Binary to Decimal Floating Point Conversion Program (Cont.)

4.5 BINARY TO DECIMAL FLOATING POINT CONVERSION

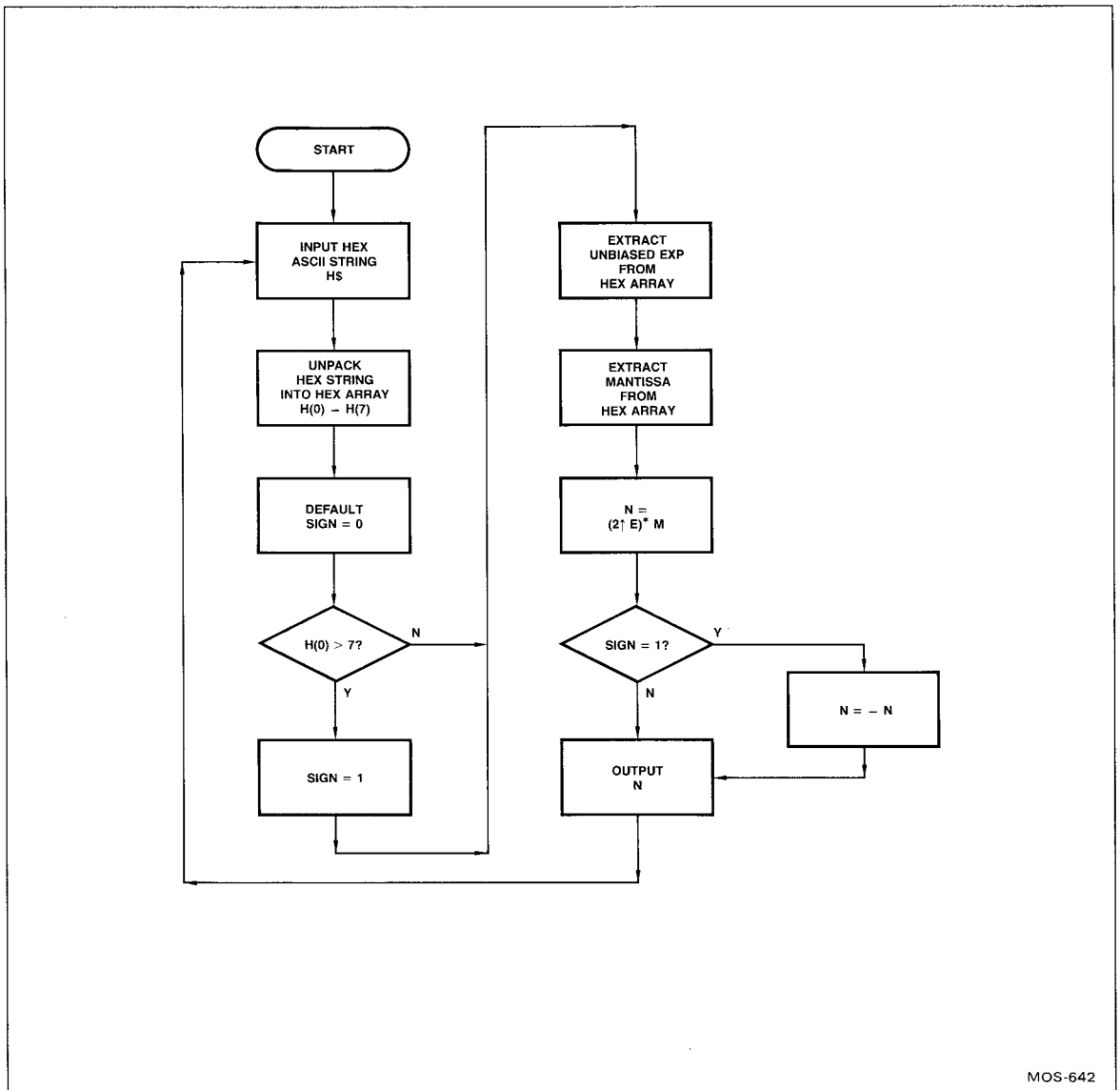
In order to read the value of a binary floating point number stored in a computer, it is often useful to convert it to a decimal number so a person can visualize the number. The conversion from binary to decimal is somewhat simpler than from decimal to binary. The following is an algorithm to convert a binary number into a decimal number:

- Unpack the binary floating point number into sign (S), unbiased exponent (E) and mantissa (M).
- Obtain the decimal value of the exponent using an integer binary to decimal conversion routine.
- Obtain the decimal value of the mantissa using a fractional binary to decimal conversion routine.
- Obtain the decimal value using

$$(-1)^S \times 2^E \times M$$

The flowchart in Fig. 4.6 and the basic program in Fig. 4.7 illustrate an example of such a conversion. The following is a description of the variables used in the basic program:

- C\$ – A single ASCII character used during unpacking of the input string.
- E – The exponent of the binary floating point number.
- H(0)-H(7) – Each element of the array represents the value of each hexadecimal ASCII character entered. That is, each element has the value 0 to 15.
- H\$ – The input string, which should be an 8-digit hexadecimal number. Characters entered after the eighth character are ignored.
- I – An integer used for loop index.
- M – The mantissa of the binary floating point number.
- N – The decimal floating point number.



MOS-642

Figure 4.6. Binary to Decimal Floating Point Conversion Flowchart


```

10 REM
20 REM
30 REM
40 DIM C(8)
50 PRINT "INPUT 8 DIGIT HEXADECIMAL NUMBER: ";
60 INPUT H$
70 REM UNPACK HEXADECIMAL NUMBER INTO BINARY ARRAY
80 FOR I = 0 TO 7
90 C$ = MID$(H$,I+1,1)
100 REM CHECK IF INPUT IS ZERO
110 IF H$ <> "00000000" THEN 140
120 PRINT "0"
130 GOTO 50
140 C(I) = ASC(C$) - 48
150 IF C(I) < 0 THEN 370
160 IF C(I) > 10 THEN C(I) = C(I) - 7
170 IF C(I) > 15 THEN 370
180 NEXT I
190 REM CHECK IF INPUT IS NORMALIZED
200 IF (C(2) AND 8) > 0 THEN 230
210 PRINT "INPUT NOT NORMALIZED FLOATING POINT NO."
220 GOTO 50
230 REM FIND SIGN OF NUMBER
240 S = 0
250 IF C(0) > 7 THEN S = 1
260 REM FIND EXPONENT OF NUMBER
270 E = 16*(C(0) AND 7) + C(1)
280 REM FIND MANTISSA OF NUMBER
290 M = 0
300 FOR I = 2 TO 7
310 M = M + C(I)/2^(4*(I-1))
320 NEXT I
330 N = (2^E) * M
340 IF S = 1 THEN N = -N
350 PRINT N
360 GOTO 50
370 PRINT "INPUT ERROR"
380 GOTO 50

```

Figure 4.7. Binary to Decimal Floating Point Conversion Programs

CHAPTER 5 SINGLE-CHIP FLOATING POINT PROCESSORS

5.1 INTRODUCTION

Until recently, floating point computation has been implemented either in software or in hardware with MSI/SSI (medium-scale integration/small-scale integration) devices. The former method involves considerable programming effort and the resulting product is usually very slow. It also consumes valuable main memory space for the floating point routines. The latter method involves using hundreds of ICs, which requires considerable development effort, and the resulting product is expensive to manufacture and requires considerable power and space. With the advent of LSI (large-scale integration) technology in recent years, it becomes possible to put a complete hardware floating point processor into a single IC.

The advantages of the single-chip LSI floating point processor compared to previous hardware implementation are as follows:

Low development cost –

The cost of developing an interface to a single-chip floating point processor should be less than 10 percent of the cost of developing a complete hardware floating point processor.

Low production cost –

The cost of producing and testing of hardware floating point boards is at least several hundred dollars whereas the cost of a single-chip processor is only a small fraction of that cost.

Improved reliability –

Most electronic failures occur at the interface level. By combining all the logic inside a single device, the number of connections in the system is drastically reduced. Hence reliability is increased.

Less power consumption –

The single-chip processor typically draws less than 5 percent of the power of an MSI/SSI implementation.

Less space –

The single-chip processor usually fits on the same board as the CPU, thus requiring one or two fewer boards than the MSI/SSI solution.

Get product to market sooner –

Due to less effort required both for development and production, using single-chip processors will shorten the design cycle of a new product.

The advantages of the single-chip LSI floating point processor over software floating point computation methods are:

Enhanced execution speed –

Hardware floating point processors typically execute floating point arithmetic five to 50 times faster than software. If the floating point processor allows concurrent CPU execution, the overall throughput is even further enhanced for applications

where the CPU can do other meaningful tasks during a floating point computation.

Low development cost –

The cost of developing a comprehensive software floating point package often involves many manmonths of programming effort. With a hardware processor, programming is drastically reduced because the floating point computation algorithm is precoded inside the hardware processor.

Less main memory required –

Since the floating point processors contain the computation algorithm on chip (often in microcode), it could save a few thousand bytes of main memory. This should be important in applications where CPU has limited addressing space.

Improved portability –

With the advent of new microprocessors in rapid frequency, software often must be rewritten when upgrading from one CPU to another. When using the hardware processors, rewriting the floating point routines is eliminated.

The first LSI single-chip floating processors available commercially were introduced by Advanced Micro Devices. AMD introduced the Am9511 Arithmetic Processor unit in 1977 and the Am9512 Floating Point Processor unit in 1979.

5.2 Am9511A ARITHMETIC PROCESSOR

This pioneer single-chip arithmetic processor interfaces with most popular 8-bit microprocessors such as Am9080A, Am8085, MC6800 by Motorola and Z80 by Zilog. It can also be used for 16-bit microprocessors such as AmZ8000,* but its performance with such 16-bit microprocessors is somewhat hindered by its 8-bit external data bus.

Although the external interface is only 8 bits wide, the Am9511A internally is a 16-bit microprogrammed, stack-oriented floating point machine. It includes not only floating point operations but fixed point as well. In addition to the basic add, subtract, multiply and divide operations, transcendental derived functions are also included. A data sheet of Am9511A is included in Appendix A.

5.3 Am9512 FLOATING POINT PROCESSOR

The Am9512 is a follow-up to the Am9511A. Although the hardware interface between the two chips is similar, the data formats are different.

The Am9512 supports two data types: 32-bit binary floating point and 64-bit binary floating point. The formats adopted are compatible with one of the proposed IEEE formats. Unlike the Am9511A, the Am9512 does not have any of the derived transcendental functions. A description of the Am9512 is included in Appendix B.

*Z8000 is a trademark of Zilog, Inc.

Am9080A. The Am9080A stores the current program counter on the stack and jumps to location 38H to execute the interrupt handling routine. By pulling the EACK HIGH, the END output will stay LOW until the first read/write operation is performed on the Am9511A, thus clearing the interrupt request.

6.3 Am9080A TO Am9512 INTERFACE

Figure 6.2 illustrates an example of interfacing the Am9512 to the Am9080A. The principal timing difference between the Am9511A and the Am9512 is that the PAUSE follows RD or WR in the Am9511A whereas the PAUSE follows CS in the Am9512.

Two additional gates (74LS08 and 74LS32) are inserted in the PAUSE to RDYIN line. Otherwise, during a memory cycle in which the memory address bits 1 to 7 match the I/O address of the Am9512, the PAUSE will go LOW. Since there will be no IOR or IOW in that cycle to reset the PAUSE, the system will be dead-locked. The additional gates allow the PAUSE to pass through only if the current cycle is an I/O cycle. Strobing the chip select decoder with IOR or IOW will not work because that will create a negative chip select to RD or WR setup time, which is not permitted with the Am9512. Other considerations about the chip-select decoding are the same as discussed in Section 6.2.

The 74LS32 gate shown at the top of Figure 6.2 allows either END or ERR to interrupt to CPU. The CPU can read the status register of the Am9512 to determine the source of the interrupt.

6.4 Am8085A to Am9511-1 INTERFACE

In a typical Am8085A system, the system clock rate is 3MHz. The Am9511-1 is selected because the Am9511-1 has as a maximum clock rate of 3 MHz. The Am8085A has an earlier ready setup window compared with the Am9080A. If the PAUSE signal is connected directly to the READY input to the Am8085A, the ready line will be pulled down too late for the Am8085A to go into the WAIT state. The 74LS74 is used for forcing one WAIT state when the Am9511-1 is accessed. After the first WAIT state, the 74LS74 Q output is reset to HIGH and the PAUSE of the Am9511-1 controls any additional wait states if necessary. The chip-select decoder is strobed with IO/M signal to prevent Am9511-1 responding to memory accesses when bits 9 to 15 of the memory address coincides with Am9511-1 I/O address.

6.5 Am8085A TO Am9512-1 INTERFACE

The Am9512 is designed specifically to interface to Am8085A. The interface is straightforward and no additional logic is required. The Am9512-1 is used instead of Am9512 because the typical Am8085A system runs at 3 MHz.

The ERR output and END output are connected to separate interrupt inputs so that the CPU can identify the source of interrupt without reading the status register of the Am9512-1.

Since the chip-select decoder is strobed with the IO/M signal, a transition is guaranteed with each I/O operation without the concern of insufficient address decode as in the Am9080A to Am9511A or Am9512 interfaces.

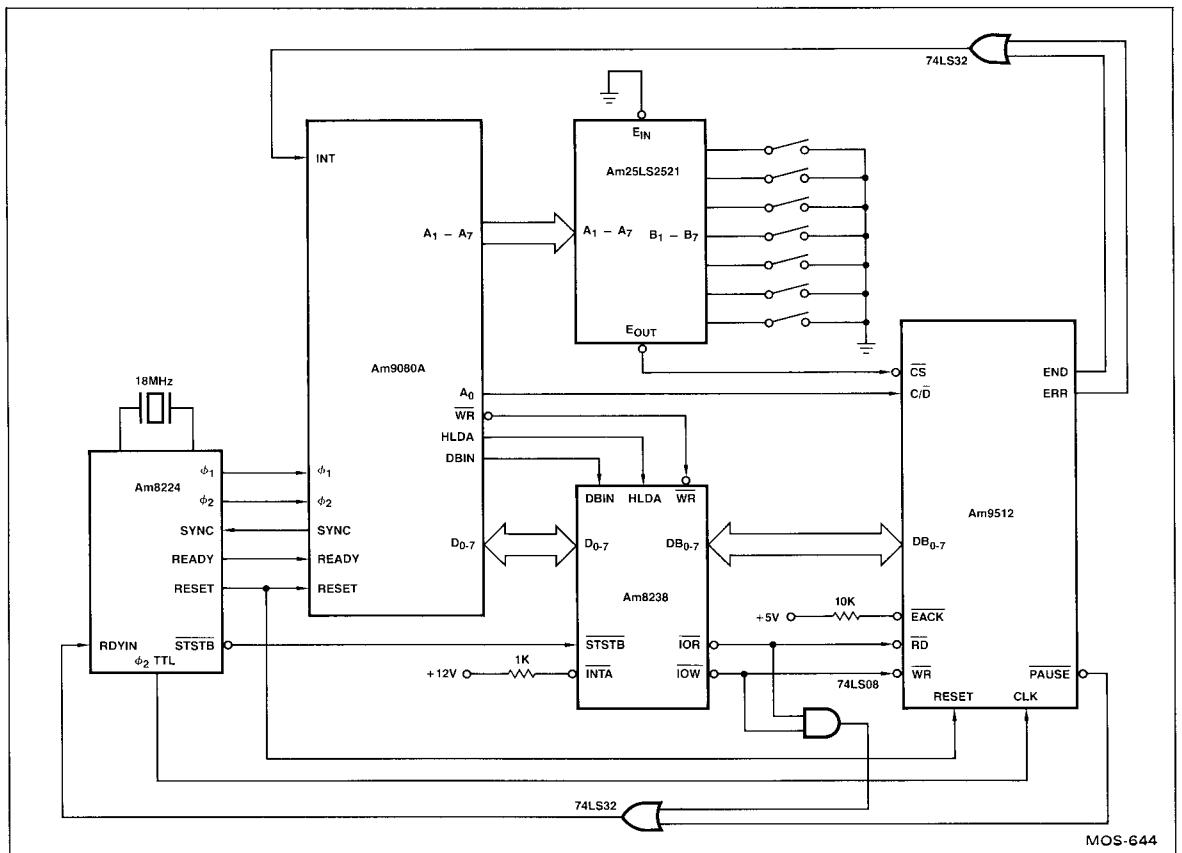


Figure 6.2. Am9080A to Am9512 Interface

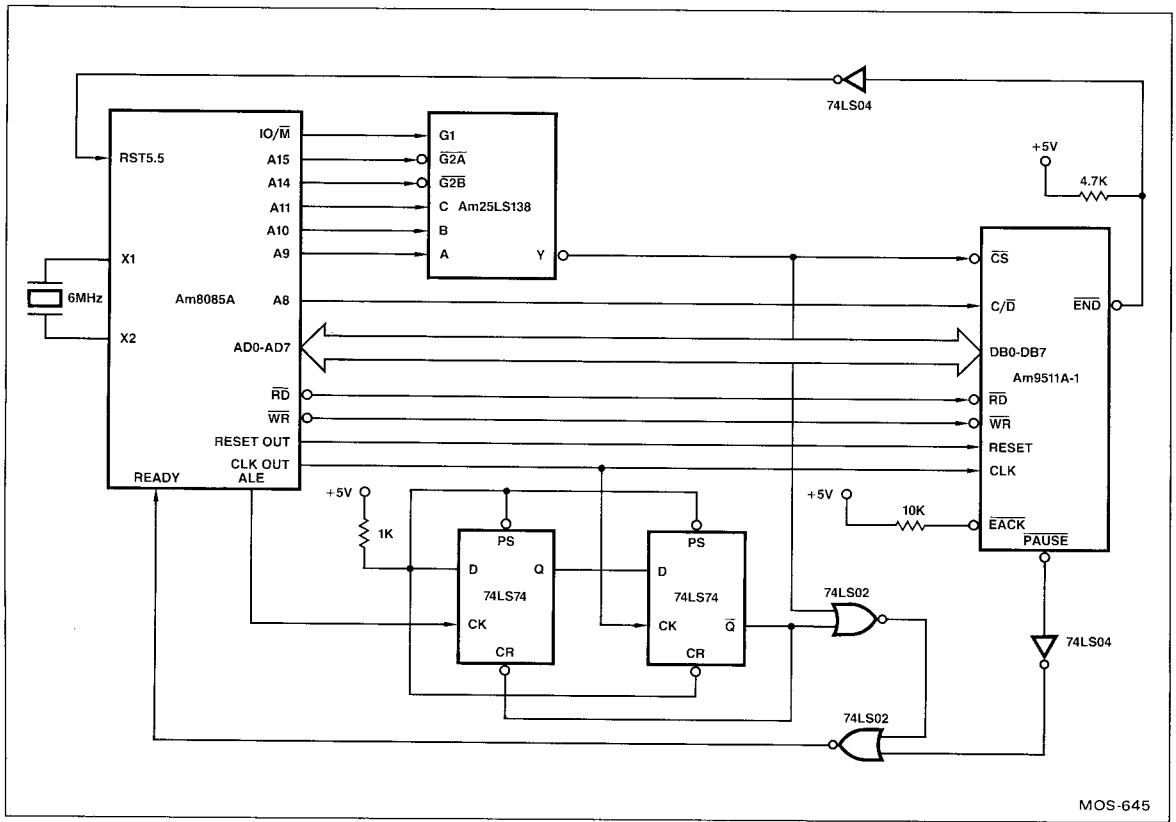


Figure 6.3. Am8085A to Am9511-1 Interface

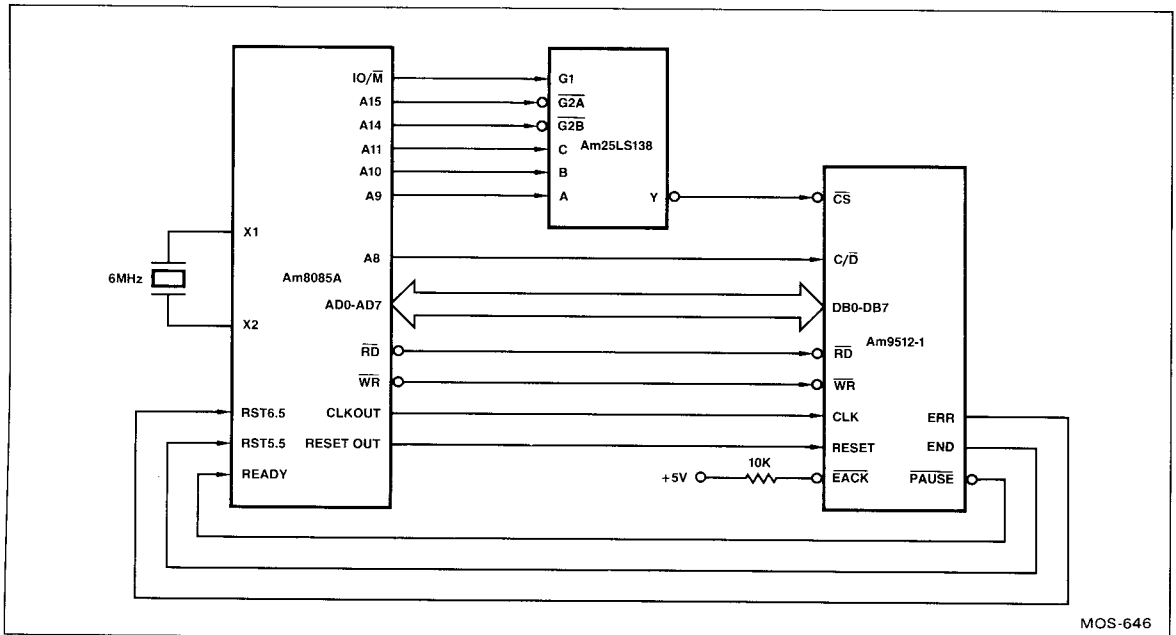


Figure 6.4. Am8085A to Am9512-1 Interface

6.6 Z80 TO Am9511A INTERFACE

Figure 6.5 illustrates a programmed I/O interface technique for Am9511A with a Z80 CPU.

The Chip Select (CS) signal is a decode of Z80 address lines A1-A7. This assigns the Am9511A to two consecutive addresses, an even (Data) address, and the next higher odd (Command) address. Selection between the Data (even) and the Command/Status (odd) ports is by the least significant address bit A0.

The IORQ (Input/Output Request) from the Z80 is an enable input to the Am25LS139 decoder. The WR and RD from the Z80 are the two inputs to the decoder. The outputs Y1 and Y2 are tied to WR and RD of the Am9511A. The PAUSE output of the Am9511A is connected to WAIT line of Z80. The Am9511A outputs a LOW on PAUSE 150ns (max) after RD or WR has become active. The PAUSE remains LOW for 3.5 TCY + 50ns (min) for data read and is LOW for 1.5 TCY + 50ns (min) for status read from Am9511A where TCY is the clock period at which Am9511A is running. Therefore, Z80 will insert one to two extra WAIT states. The Am9511A PAUSE output responds to a data read, data write, or command write request received while the Am9511A is still occupied (executing a previous command) by pulling the PAUSE output LOW. Since PAUSE and WAIT are tied together, as soon as Z80 tries to interfere with APU execution, Z80 enters the WAIT state.

6.7 Z80 TO Am9512 INTERFACE

The Am9512 interface to Z80 (Fig. 6.6) requires two more gates than the Am9511A interface to Z80. An inverter is added to the interrupt request line because the sense of the END/ERR signals

are different. The 74LS32 is added in the wait line because the Am9512 PAUSE will go LOW whenever chip select on the Am9512 goes LOW. In Fig. 6.6 the chip-select input can go LOW during second or third cycles of an instruction when the memory address matches the Am9512 I/O addressed. If the 74LS32 OR-gate is omitted, the WAIT input on the Z80 will go LOW and the system will be deadlocked. Strobing the chip-select decoder will not work because this would cause a negative chip select to RD or WR time on the Am9512.

The chip select decoder in this example is strobed with M1. This accomplishes a dual purpose. It not only guarantees a chip select transition on every I/O cycle, it also prevents the chip select to go LOW during an interrupt acknowledge cycle. This is vital because IORQ is also LOW during that cycle. Without the M1 strobe, CS might go LOW and cause PAUSE to go LOW which will again cause the system to deadlock.

6.8 MC6800 TO Am9511A INTERFACE

Figure 6.7 shows interface of a Motorola MC6800 microprocessor to an Am9511A. The MC6800 has no explicit I/O instructions. All I/O devices are treated as memory locations. Therefore the chip-select input of the Am9511A is derived from a decode of address lines A₁ to A₁₅. The decoder is strobed by VMA (Valid Memory Address) to produce a glitch-free output. The C/D input of the Am9511A is connected directly to the A₀ of the MC6800 so that the even address selects the data port and odd address selects the status or command port. The RD and WR inputs to the Am9511A is derived by demultiplexing the O₂ and VMA and the R/W signals.

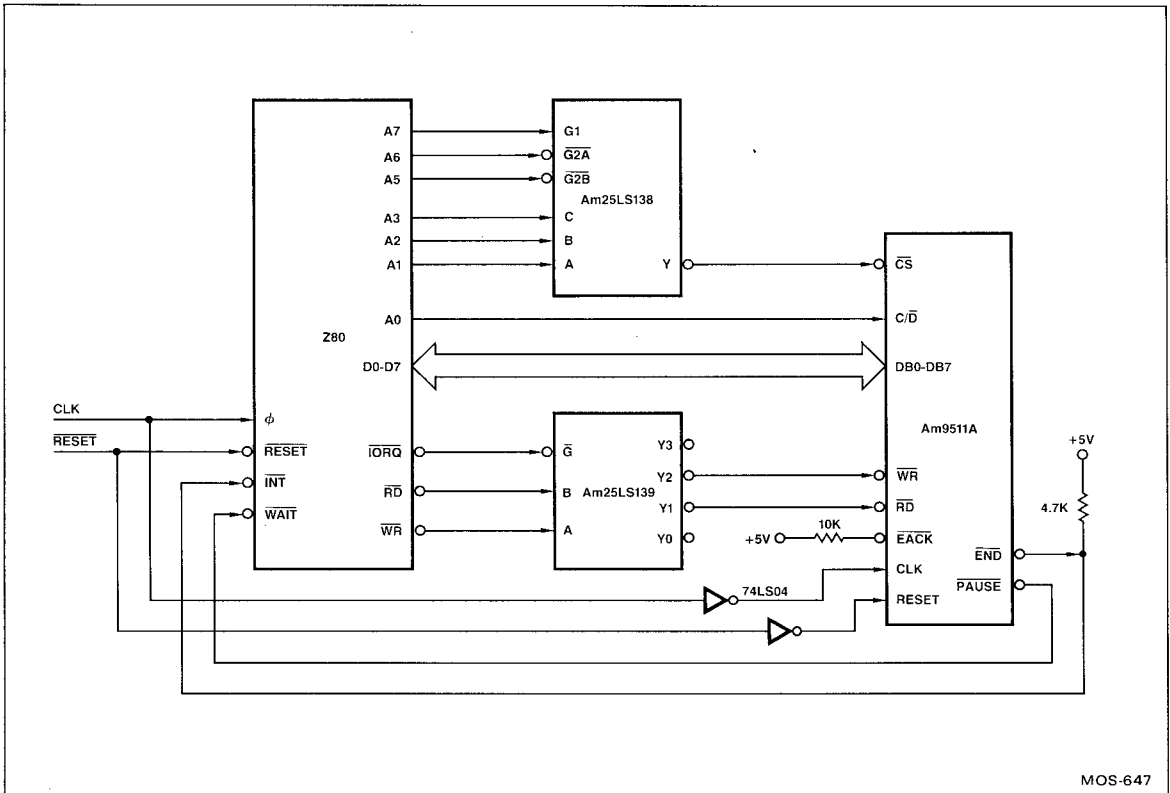


Figure 6.5. Z80 to Am9511A Interface

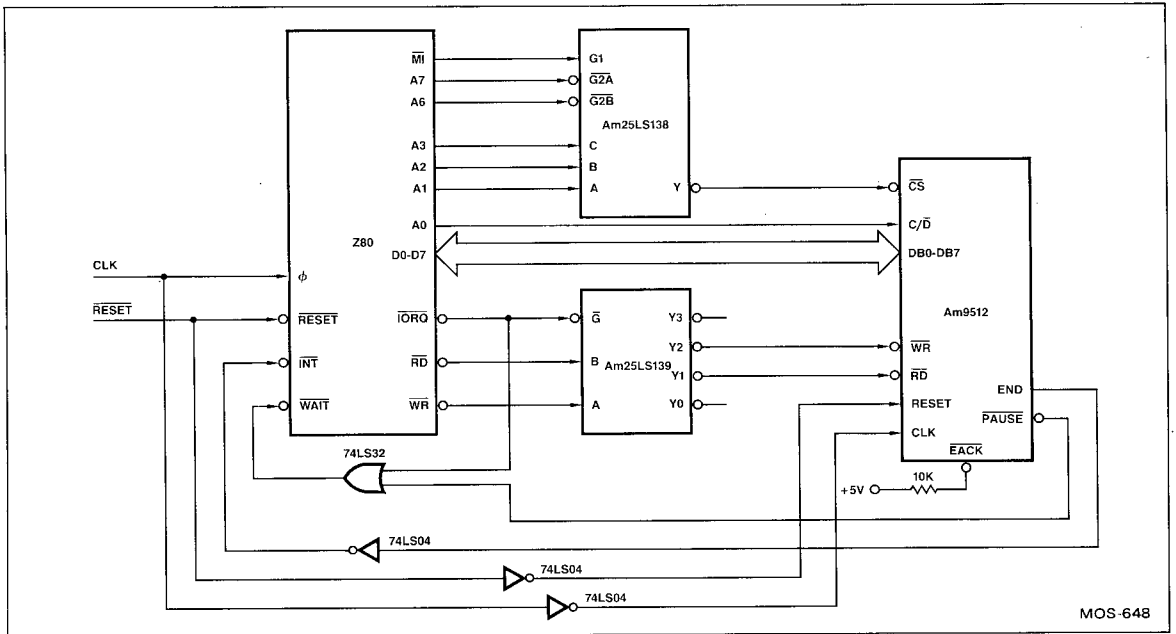


Figure 6.6. Z80 to Am9512 Interface

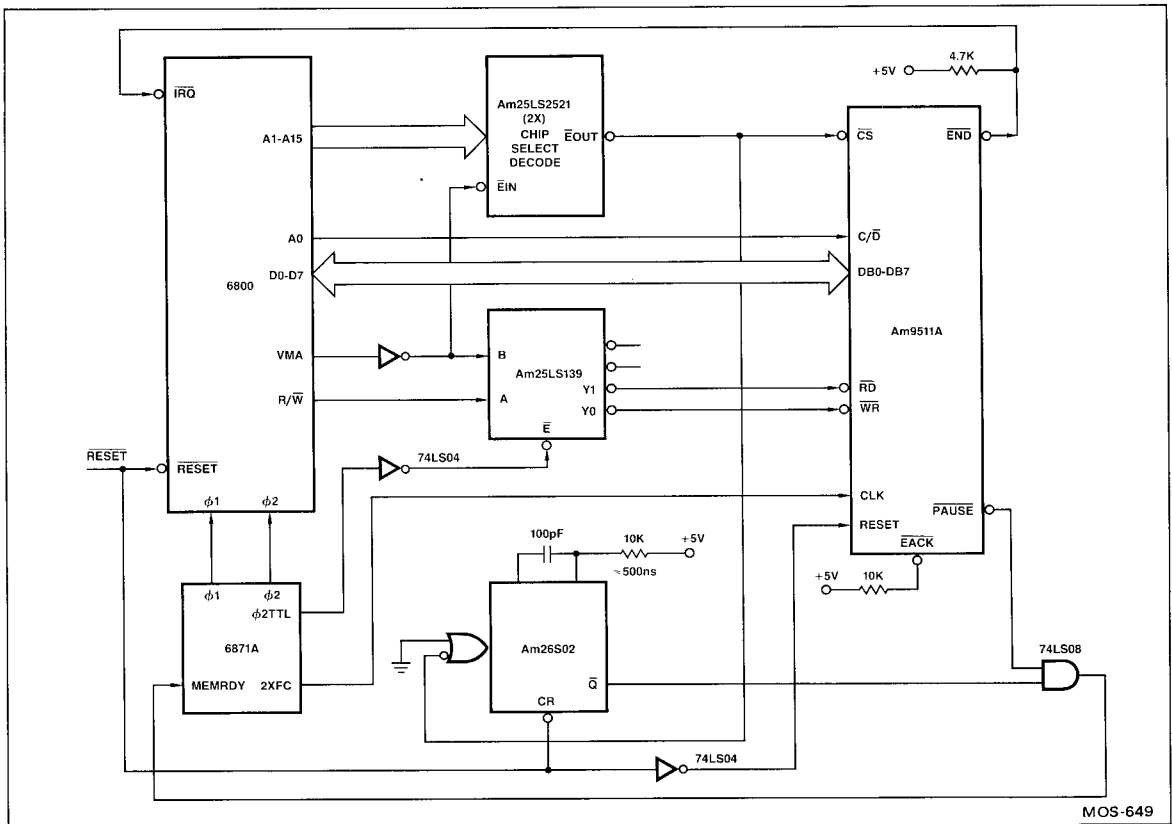


Figure 6.7. MC6800 to Am9511A Interface

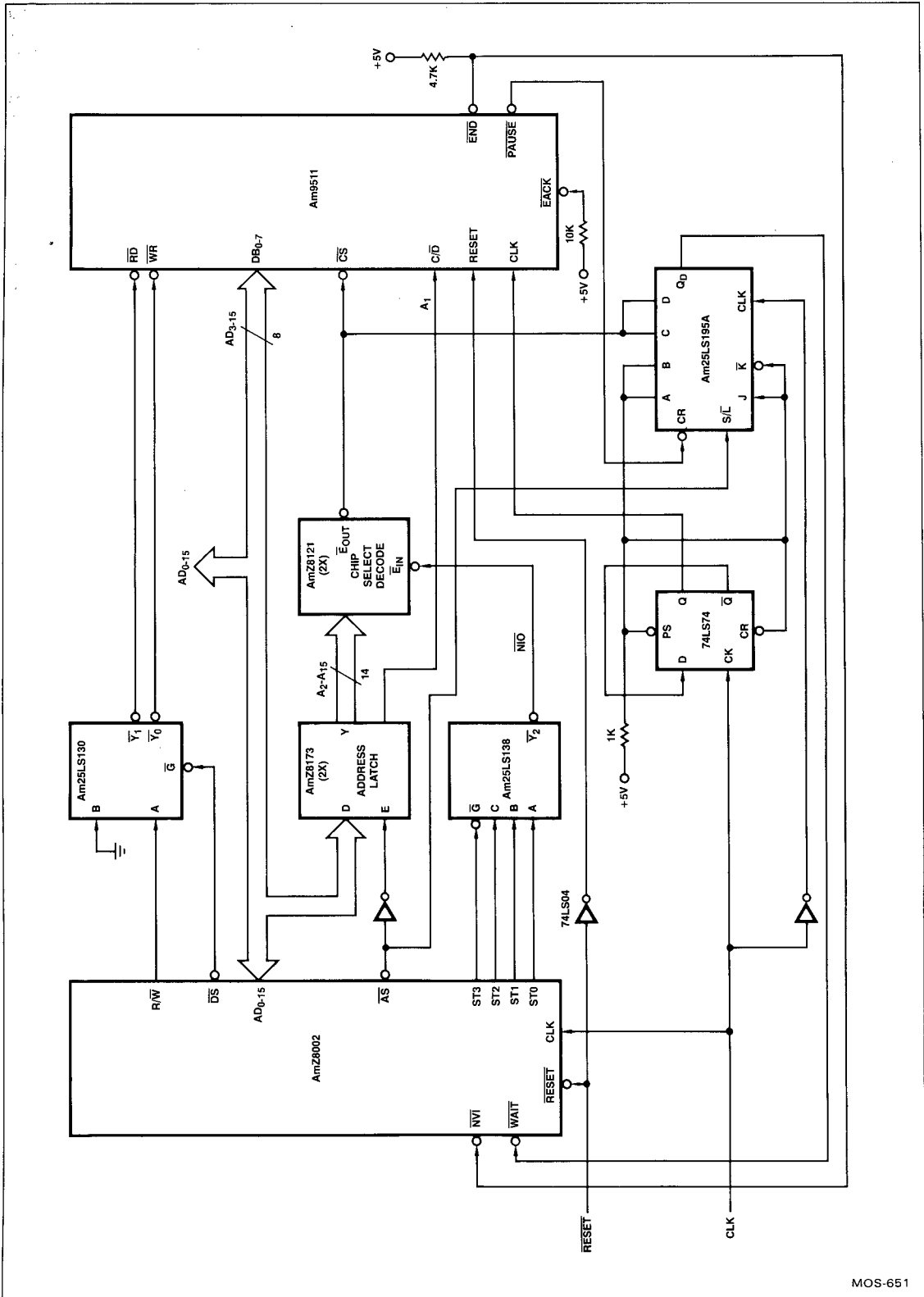


Figure 6.9. AmZ8002 to Am9511A Interface

CHAPTER 7

Am9511A INTERFACE METHODS

7.1 INTRODUCTION

Interfacing the Am9080A to the Am9511A can be accomplished in one of the following ways:

1. Demand/wait
2. Poll status
3. Interrupt driven
4. DMA transfer

The various tradeoffs of these methods are discussed below. Although only the Am9080A and Am9511A are used as an example, the principle applies to any of the processors discussed in Chapter 6.

7.2 DEMAND/WAIT

This interface is the simplest both in terms hardware and software. The connection is shown in Fig. 6.1, except that the interrupt input to the Am9080A need not be connected to the END output of the Am9511A. When this interface is used, the programmer can regard the Am9511A as always ready for READ and WRITE operations. If the Am9511A is not ready, the PAUSE will go LOW to put Am9080A in the WAIT state. When the Am9511A has completed the current operation, the PAUSE will go HIGH and the suspended READ and WRITE will proceed. Figure 7.1 shows an example of a program that loads the data into the Am9511A, executes a command and retrieves the data from the Am9511A.

The drawback of this method is that concurrent processing by the CPU is not allowed, and the CPU also cannot respond to other interrupts or DMA requests in the system while it is in the WAIT state. In systems where above considerations are not important, this would be the preferred method. This interface is not applicable to MC6800 systems because the clock of the MC6800 may not be stretched beyond 5 microseconds.

7.3 POLL STATUS

The hardware interface of this method is the same as demand/wait. The software (Fig. 7.2) is slightly more complicated. When the CPU wants to READ or WRITE to the Am9511A, the status register is first read. If the most significant bit is a 1, the Am9511A is executing a command. The CPU should refrain from performing any operation on the Am9511A except loop back for another status read. When the MSB of the status is a 0, the Am9511 has finished executing the command and the program can fall through to perform a READ or WRITE to the Am9511A.

This method does not allow the CPU to perform useful concurrent tasks, but it does allow the CPU to respond to interrupts and DMA requests when it is in the status poll loop.

7.4 INTERRUPT DRIVEN

The hardware configuration of the interrupt driven method is shown in Fig. 6.1. The CPU would first load the APU data stack and then issue a command. During the command execution, the CPU would be able to perform other useful tasks in the system. When the Am9511A has finished the command, the END output goes LOW to issue an interrupt request. When the interrupt request is acknowledged by the CPU, the CPU executes a routine to fetch from the Am9511A data stack and, if necessary, load up the data stack and issue another command.

This method is most suitable for real-time multitasking systems because concurrent execution of the CPU and APU is allowed. Figure 7.3 shows an example interrupt handler for Am9511A.

7.5 DMA TRANSFER

If ultimate system performance is required, the Am9511A data stack can be loaded and unloaded by a DMA controller such as the Am9517. To achieve maximum throughput, two channels of the Am9517 DMA controller are used in the configuration shown. Channel 2 is used to load the Am9511A and channel 3 is used to unload the Am9511 result into the main memory. For real-time interrupt driven systems, an interrupt controller such as the Am9519A should also be used. Figure 7.4 shows the connection diagram of such a system and Fig. 7.5 shows a sample program to drive such a system.

The following is the initializing sequence required only after power up or system reset:

1. The Command Register
 - Bit 0 = Don't care (applies to memory to transfer option)
 - Bit 1 = Don't care (applies to memory option)
 - Bit 3 = 0, Enable DMA controller
 - Bit 4 = 0, Normal timing
 - Bit 5 = 1, Extended write
 - Bit 6 = 0, DREQ active HIGH
 - Bit 7 = 0, DACK active LOW
2. The mode register of channel 2:
 - Read mode, auto initialize, address decrement, block mode
3. The mode register of channel 3:
 - Write mode, auto initialize, address increment, block mode
4. The word count register of channel 2:
 - Initialized to a count of 8
5. The word count register of channel 3:
 - Initialized to a count of 4
6. Mask register:
 - Channels 2 and 3 cleared

The word count registers may need to be modified later if the word count desired is not the default value.

The following is a sequence of operations required for each Am9511A operation:

1. The operand address is written to the base address register of channel 2 of the Am9517.
2. If the word count of the operand is different from the previous operation, the new word count is written to channel 2 of the Am9517.
3. The address of the result is written to the channel 3 base address register.
4. A software request is sent to channel 2.
5. The CPU performs other tasks.
6. An interrupt is received from channel 2 end of operation signal.
7. The CPU writes the command word into the command register with MSB of the command word set to 1 to indicate DMA service required at end of operation.
8. The CPU is free to perform other tasks.
9. An interrupt is received from channel 3 end of operation signal. The result is now is the desired location in main memory.

The above method offers maximum concurrent operation of an Am9080A and Am9511A system. If Am9511 or Am9512 is used instead of Am9511A, the mode of transfer of the Am9517 must be in single transfer mode to obtain a transition at the chip select input of the Am9511 or Am9512.

```

LOC  OBJ          LINE      SOURCE STATEMENT
      1 $          1          PAGEWIDTH(80) MACROFILE NOOBJECT
      2 ;
      3 ;          *****
      4 ;
      5 ;          PROGRAMS FOR CHAPTER 7 OF
      6 ;          FLOATING POINT TUTORIAL
      7 ;
      8 ;          *****
      9 ;
     10          NAME CHAP7
     11 ;
     12 ;          AM9511A ARITHMETIC PROCESSING UNIT
     13 ;          I/O PORT ASSIGNMENT
     14 ;
00C0  15 APUDR      EQU 0C0H          ;AM9511A DATA PORT
00C1  16 APU R      EQU APUDR+1        ;AM9511A STATUS PORT
00C1  17 APUCR      EQU APUSR          ;AM9511A COMMAND PORT
     18 ;
     19 ;          AM9517A MULTIMODE DMA CONTROLLER
     20 ;          I/O PORT ASSIGNMENT
     21 ;
00B0  22 DMAC        EQU 0B0H          ;AM9517A BASE ADDRESS
00B4  23 CH2ADR      EQU DMAC+4        ;CHANNEL 2 ADDRESS
00B5  24 CH2CNT      EQU DMAC+5        ;CHANNEL 2 BYTE COUNT
00B6  25 CH3ADR      EQU DMAC+6        ;CHANNEL 3 ADDRESS
00B7  26 CH3CNT      EQU DMAC+7        ;CHANNEL 3 BYTE COUNT
00B8  27 CMD17       EQU DMAC+8        ;COMMAND REGISTER
00B9  28 REQ17       EQU DMAC+9        ;REQUEST REGISTER
00BB  29 MOD17       EQU DMAC+0BH       ;MODE REGISTER
00BD  30 CLR17       EQU DMAC+0DH       ;MASTER CLEAR
00BF  31 MSK17       EQU DMAC+0FH       ;MASK REGISTER
     32 ;
     33 ;          AM9519 UNIVERSAL INTERRUPT CONTROLLER
     34 ;          I/O PORT ASSIGNMENT
     35 ;
00C2  36 UICDR      EQU 0C2H          ;AM9519 DATA PORT
00C3  37 UIC R      EQU UICDR+1        ;AM9519 STATUS PORT
00C3  38 UICCR      EQU UICSR          ;AM9519 COMMAND PORT
     39 ;
     40          CSEG
     41 ;
     42 ;          PROGRAM EXAMPLE FOR DEMAND WAIT INTERFACE
     43 ;          ***** FIGURE 7.1 *****
     44 ;
     45 ;          TO CALL THE FOLLOWING PROGRAM,
     46 ;          ON ENTRY:
     47 ;          HL = POINTER TO THE FIRST OPERAND (NOS)
     48 ;          DE = POINTER TO THE SECOND OPERAND (TOS)
     49 ;          BC = POINTER TO THE RESULT
     50 ;          A = THE 2 OPERAND OPCODE
     51 ;
     52 ;          ON RETURN:
     53 ;          A = THE STATUS REGISTER OF AM9511A
     54 ;          ALL POINTERS ARE DESTROYED

```

Figure 7.1. Demand/Wait Programming

LOC	OBJ	LINE	SOURCE STATEMENT
		55 ;	
0000	C5	56	DEMAND: PUSH B ;SAVE RESULT POINTER
0001	F5	57	PUSH PSW ;SAVE OPCODE
0002	010300	58	LXI B,3
0005	09	59	DAD B ;MOVE SOURCE POINTER TO LSB
		60 ;	
		61 ;	PUSH OPERAND #1 ONTO APU DATA STACK
		62 ;	
0006	0604	63	MVI B,4 ;INIT LOOP1 COUNTER
0008	7E	64	DLOOP1: MOV A,M ;FETCH A BYTE FROM OPER 1
0009	D3C0	65	OUT APUDR ;PUSH ONTO APU DATA STACK
000B	2B	66	DCX H ;DEC. BYTE POINTER
000C	05	67	DCR B ;DEC. LOOP COUNTER
000D	C20800	68	JNZ DLOOP1
		69 ;	
0010	EB	70	XCHG ;PUT OPERAND 2 POINTER IN HL
0011	010300	71	LXI B,3
0014	09	72	DAD B ;MOVE POINTER TO LSB
		73 ;	
		74 ;	PUSH OPERAND #2 ONTO APU DATA STACK
		75 ;	
0015	0604	76	MVI B,4
0017	7E	77	DLOOP2: MOV A,M ;FETCH A BYTE FROM OPER 2
0018	D3C0	78	OUT APUDR ;PUSH ONTO APU DATA STACK
001A	2B	79	DCX H ;DEC. BYTE POINTER
001B	05	80	DCR B ;DEC. LOOP COUNTER
001C	C21700	81	JNZ DLOOP2
		82 ;	
		83 ;	OPERAND LOAD COMPLETE, WRITE COMMAND
		84 ;	
001F	F1	85	POP PSW ;RETRIEVE COMMAND OPCODE
0020	D3C1	86	CUT APUCR ;WRITE TO APU COMMAND PORT
		87 ;	
		88 ;	READ DATA FROM STACK
		89 ;	IF THE APU IS NOT READY, THE PAUSE
		90 ;	SIGNAL WILL PUT AM9080A INTO THE
		91 ;	"WAIT" STATE UNTIL THE DATA IS READY
		92 ;	
0022	C1	93	POP B ;RETRIEVE RESULT POINTER
0023	1E04	94	MVI E,4 ;INIT LOOP3 COUNTER
0025	DBC0	95	DLOOP3: IN APUDR ;READ APU STACK
0027	02	96	STAX B ;STORE RESULT IN MEMORY
0028	03	97	INX B
0029	1D	98	DCR E
002A	C22500	99	JNZ DLOOP3
		100 ;	
		101 ;	RETURN STATUS IN A
		102 ;	
002D	DBC1	103	IN APUSR
002F	C9	104	RET
		105 \$	EJECT

Figure 7.1. Demand/Wait Programming (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
		106 ;	
		107 ;	SUBROUTINE FOR POLL STATUS INTERFACE
		108 ;	**** FIGURE 7.2 ****
		109 ;	
0030	C5	110	POLL: PUSH B ;SAVE RESULT POINTER
0031	F5	111	PUSH PSW ;SAVE OPCODE
0032	010300	112	IXI B,3
0035	09	113	DAD B ;MOVE POINTER TO LSB
		114 ;	
		115 ;	CHECK IF AM9511A IS READY TO ACCEPT DATA
		116 ;	
0036	DBC1	117	CHK1: IN APUSR ;READ APU STATUS
0038	E7	118	ORA A ;SET CPU FLAGS
0039	FA3600 C	119	JM CHK1 ;LOOP BACK IF NOT READY
		120 ;	
		121 ;	THE AM9511A IS READ IF FALLEN THROUGH
		122 ;	
003C	0604	123	MVI B,4 ;INIT LOOP1 COUNTER
003E	7E	124	PLOOP1: MOV A,M ;FETCH FROM OPERAND 1
003F	D3C0	125	OUT APUDR ;PUSH ONTC APU DATA STACK
0041	2B	126	DCX H ;DEC. BYTE POINTER
0042	05	127	DCR B ;DEC. LOOP COUNTER
0043	C23E00 C	128	JNZ PLOOP1
		129 ;	
0046	EB	130	XCHG ;PUT OPERAND 2 POINTER IN HL
0047	010300	131	LXI B,3
004A	09	132	DAD B ;MOVE POINTER TO LSB
		133 ;	
		134 ;	PUSH OPERAND #2 ONTO APU DATA STACK
		135 ;	
004B	0604	136	MVI B,4 ;INIT LOOP2 COUNTER
004D	7E	137	PLOOP2: MOV A,M ;FETCH FROM OPERAND 2
004E	D3C0	138	OUT APUDR ;PUSH ONTO APU DATA STACK
0050	2B	139	DCX H ;DEC. BYTE POINTER
0051	05	140	DCR B ;DEC. LOOP COUNTER
0052	C24D00 C	141	JNZ PLOOP2
		142 ;	
		143 ;	OPERANDS LOADED, WRITE COMMAND
		144 ;	
0055	F1	145	POP PSW ;RETRIEVE OPCODE
0056	D3C1	146	CUT APUCR ;WRITE COMMAND TO APU
		147 ;	
		148 ;	SET UP RESULT POINTER AND LOOP3 COUNTER
		149 ;	
0058	C1	150	POP B ;RETRIEVE RESULT POINTER
0059	1E04	151	MVI E,4 ;INIT LOOP3 COUNTER
		152 ;	
		153 ;	WAIT UNTIL AM9511A FINISH EXECUTION
		154 ;	
005B	DBC1	155	CHK2: IN APUSR ;READ APU STATUS PORT
005D	B7	156	ORA A ;SET STATUS FLAGS
005E	FA5B00 C	157	JM CHK2 ;LOOP BACK IF NOT READY
0061	F5	158	PUSH PSW ;SAVE APU STATUS
		159 ;	
		160 ;	THE AM9511A HAS FINISHED EXECUTION

Figure 7.2. Status Poll Programming Interface

LOC	OBJ	LINE	SOURCE STATEMENT
		161 ;	READ RESULT
		162 ;	
0062	DBC0	163	PLOOP3: IN APUDR ;READ APU DATA STACK
0064	02	164	STAX B ;STORE RESULT IN MEMORY
0065	03	165	INX B ;INC. MEMORY POINTER
0066	1D	166	DCR E ;DEC. LOOP COUNTER
0067	C26200	C 167	JNZ PLOOP3
		168 ;	
		169 ;	EXECUTION COMPLETE, RESTORE STATUS IN A
		170 ;	
006A	F1	171	POP PSW ;RESTORE APU STATUS
006B	C9	172	RET
		173 \$	EJECT

Figure 7.2. Status Poll Programming Interface (Cont.)

```

LOC  OBJ          LINE      SOURCE STATEMENT
                                174 ;
                                175 ;      SUBROUTINES FOR INTERRUPT DRIVEN INTERFACE
                                176 ;      ***** FIGURE 7.3 *****
                                177 ;
                                178 ;      LOCATE INTERRUPT HANDLER IN RST 7 LOCATION
                                179 ;
                                180      ASEG
0038          181      ORG 38H
                                182 ;
0038 F5       183 RST7:  PUSH PSW          ;SAVE ALL REGISTERS USED
0039 C5       184      PUSH B
003A E5       185      PUSH H
003B 0604     186      MVI B,4          ;INIT LOOP COUNTER
003D 2A0000   D 187      LHLD RSTPTR      ;FETCH RESULT POINTER
                                188 ;
0040 DBC0     189 ILOOP1: IN APUDR          ;READ RESULT FROM APU
0042 77       190      MOV M,A          ;STORE IT IN MEMORY
0043 23       191      INX H          ;BUMP MEMORY POINTER
0044 05       192      DCR B          ;DEC. LOOP COUNTER
0045 C24000   193      JNZ ILOOP1
                                194 ;
                                195 ;      DONE, SET DONE FLAG AND RESTORE REGISTERS
                                196 ;
0048 3E01     197      MVI A,1
004A 320200   D 198      STA DONE
004D E1       199      POP H
004E C1       200      POP B
004F F1       201      POP PSW
0050 C9       202      RET
                                203 ;
                                204 ;      SUBROUTINE TO LOAD APU STACK AND SEND
                                205 ;      COMMAND WORD
                                206 ;
                                207 ;      CALLING SQUENCE:
                                208 ;      ON ENTRY  HL = POINTER TO MSB OF 8 BYTES
                                209 ;                  OF OPERAND
                                210 ;                  DE = POINTER TO 4 BYTES OF RESULT
                                211 ;                  A = EXECUTION OPCODE
                                212 ;
                                213 ;      ON RETURN:  ALL REGISTER ARE NOT AFFECTED,
                                214 ;                  DONE FLAG CLEARED.
                                215 ;
                                216      CSEG
                                217 ;
006C E5       218 LOAD:  PUSH H          ;SAVE OPERAND POINTER
006D D5       219      PUSH D          ;SAVE RESULT POINTER
006E F5       220      PUSH PSW       ;SAVE OPCODE
                                221 ;
006F 110800   222      LXI D,8          ;OPER. OFFSET, E = LOOP2 CTR
0072 19       223      DAD D          ;MOVE OPERAND POINTER TO LSB
                                224 ;
                                225 ;      CHECK AM9511A STATUS
                                226 ;
0073 DBC1     227 LLOOP1: IN APUSR          ;READ AM9511 STATUS REG.
0075 B7       228      ORA A          ;TEST FOR BUSY

```

Figure 7.3. Interrupt Driven Programming

LOC	OBJ		LINE	SOURCE STATEMENT
0076	FA7300	C	229	JM LLOOP1 ;WAIT UNTIL NOT BUSY
			230 ;	
			231 ;	LOAD AM9511 STACK
			232 ;	
0079	2B		233	LLOOP2: DCX H ;DEC. OPERAND POINTER
007A	7E		234	MOV A,M ;FETCH 1 BYTE OF OPERAND
007E	D3C0		235	CUT APUDR ;LOAD APU DATA STACK
007D	1D		236	DCR E ;DEC. LOOP COUNTER
007E	C27900	C	237	JNZ LLOOP2
			238 ;	
0081	F1		239	POP PSW ;GET OP CODE
0082	D3C1		240	OUT APUCR ;WRITE TO APU COMMAND REG.
0084	210200	D	241	LXI H,DONE
0087	3600		242	MVI M,0 ;CLEAR DONE FLAG
0089	E1		243	POP H ;GET RESULT POINTER
008A	220000	D	244	SHLD RSTPTR ;STORE IN RESULT POINTER
008D	EB		245	XCHG ;RESTORE DE REG. PAIR
008E	E1		246	P P H ;RESTORE HL
008F	C9		247	RET
			248 ;	
			249 ;	RAM AREA
			250 ;	
			251	DSEG
			252 ;	
0000			253	RSTPTR: DS 2 ;RESULT POINTER
0002			254	DONE: DS 1 ;DONE FLAG, 1 = DONE
			255	§ EJECT

Figure 7.3. Interrupt Driven Programming (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
		256 ;	
		257 ;	HIGH PERFORMANCE INTERFACE WITH
		258 ;	AM9517A AND AM9519
		259 ;	**** FIGURE 7.4 ****
		260 ;	
		261	CSEG
		262 ;	
		263 ;	AM9517A INITIALIZATION ROUTINE
		264 ;	CALLING SEQUENCE:
		265 ;	NO PARAMETERS REQUIRED ON ENTRY.
		266 ;	SOURCE OPERANDS ASSUMED TO BE 8 BYTES AND
		267 ;	RESLUT OPERAND ASSUMED TO BE 4 BYTES
		268 ;	
		269 ;	ON RETURNED: NO REGISTER AFFECTED
		270 ;	
0090	F5	271	INIT17: PUSH PSW ;SAVE PSW
0091	D3BD	272	OUT CLR17 ;MASTER CLEAR
0093	3E20	273	MVI A,00100000B ;LOAD COMMAND WORD
0095	D3B8	274	OUT CMD17 ;WRITE TO COMMAND REG.
0097	3EBA	275	MVI A,10111010B ;LOAD CH 2 MODE WORD
0099	D3BB	276	OUT MOD17 ;INIT CHANNEL 2 MODE
009B	3E97	277	MVI A,10010111B ;LOAD CH 3 MODE WORD
009D	D3BB	278	OUT MOD17 ;INIT CHANNEL 3 MODE
009F	3E08	279	MVI A,8 ;LOAD CH 2 BYTE COUNT
00A1	D3B5	280	OUT CH2CNT ;INIT CH 2 LOW BYTE COUNT
00A3	AF	281	XRA A
00A4	D3B5	282	OUT CH2CNT ;INIT CH 2 HIGH BYTE COUNT
00A6	3E04	283	MVI A,4 ;LOAD CH 3 BYTE COUNT
00A8	D3B7	284	OUT CH3CNT ;INIT CH 3 LOW BYTE COUNT
00AA	AF	285	XRA A
00AB	D3B7	286	OUT CH3CNT ;INIT CH 3 HIGH BYTE COUNT
00AD	3E03	287	MVI A,00000011B ;LOAD MASK REGISTER PATTERN
00AF	D3B7	288	OUT MSK17 ;INIT MASK REGISTER
00B1	F1	289	POP PSW ;RESTORE PSW
00B2	C9	290	RET
		291 ;	
		292 ;	SUBROUTINE TO INITIALIZE AM9519
		293 ;	CALLING SEQUENCE:
		294 ;	ON ENTRY: HL = STARTING ADDRESS OF WRITE
		295 ;	COMMAND SUBROUTINE
		296 ;	DE = STARTING ADDRESS OF SET
		297 ;	DONE FLAG SUBROUTINE
		298 ;	ON RETURN: NO REGISTERS ARE AFFECTED
		299 ;	
00B3	F3	300	INIT19: DI ;DISABLE ALL CPU INTERRUPTS
00B4	F5	301	PUSH PSW ;SAVE PSW
00B5	AF	302	XRA A
00B6	D3C3	303	OUT UICCR ;SOFTWARE RESET AM9519
00B8	3E88	304	MVI A,10001000B ;MODE WORD FOR M0-M4
00BA	D3C3	305	OUT UICCR ;SET M0-M4
00BC	3EC0	306	MVI A,11000000B ;SELECT AUTO CLEAR REG
00BE	D3C3	307	OUT UICCR
00C0	3E03	308	MVI A,00000011B ;SELECT CH 0 & 1 FOR AUTO CLR
00C2	D3C2	309	OUT UICDR
00C4	3EB0	310	MVI A,10110000B ;SELECT MASK REGISTER

Figure 7.4. DMA Interface Programming

LOC	OBJ	LINE	SOURCE STATEMENT
00C6	D3C3	311	OUT UICCR
00C8	3EFC	312	MVI A,11111100B ;CLR CH 0 & 1 MASK REG.
00CA	D3C2	313	OUT UICDR
00CC	3EF0	314	MVI A,11110000B ;SEL CH 0 FOR 3 BYTES
00CE	D3C3	315	OUT UICCR
00D0	3ECD	316	MVI A,0CDH ;9080A 'CALL' OPCODE
00D2	D3C2	317	CUT UICDR
00D4	7B	318	MOV A,E ;GET CH 0 LOW ADDRESS
00D5	D3C2	319	OUT UICDR
00D7	7A	320	MOV A,D ;GET CH 0 HIGH ADDRESS
00D8	D3C2	321	OUT UICDR
00DA	3EF1	322	MVI A,11110001B ;SEL CH 1 FOR 3 BYTES
00DC	D3C3	323	OUT UICCR
00DE	3ECD	324	MVI A,0CDH ;9080A 'CALL' OPCODE
00E0	D3C2	325	OUT UICDR
00E2	7D	326	MOV A,L ;GET CH 1 LOW ADDRESS
00E3	D3C2	327	OUT UICDR
00E5	7C	328	MOV A,H ;GET CH 1 HIGH ADDRESS
00E6	D3C2	329	OUT UICDR
00E8	3EA1	330	MVI A,10100001B ;ARM AM9519
00EA	D3C3	331	OUT UICCR
00EC	F1	332	POP PSW ;RESTORE PSW
00ED	FB	333	EI ;ENABLE CPU INTERRUPTS
00EE	C9	334	RET
		335 ;	
		336 ;	
		337 ;	SUBROUTINE TO PERFORM AN EXECUTION WITH
		338 ;	8 BYTES OF OPERANDS AND 4 BYTES OF RESULT
		339 ;	CALLING SEQUENCE:
		340 ;	0 ENTRY: HL = ADDRESS OF OPERANDS
		341 ;	DE = ADDRESS OF RESULT
		342 ;	A = OPCODE
		343 ;	ON RETURN: ALL REGISTERS ARE NOT AFFECTED
		344 EXEC:	
00EF	F5	345	PUSH PSW ;SAVE OPCODE
00F0	320300	D 346	STA OPCODE ;INIT OPCODE STORAGE
00F3	AF	347	XRA A
00F4	320400	D 348	STA DONE2 ;CLEAR DONE FLAG
00F7	7D	349	MOV A,L
00F8	D3E4	350	OUT CH2ADR ;INIT CH 2 LOW ADDR
00FA	7C	351	MOV A,H
00FB	D3E4	352	OUT CH2ADR ;INIT CH 2 HIGH ADDR
00FD	7B	353	MOV A,E
00FE	D3B6	354	CUT CH3ADR ;INIT CH 3 LOW ADDR
0100	7A	355	MOV A,D
0101	D3B6	356	OUT CH3ADR ;INIT CH 3 HIGH ADDR
0103	3E06	357	MVI A,00000110B
0105	D3B9	358	OUT REQ17 ;SOFTWARE REQ TO CH 2
0107	F1	359	POP PSW ;RESTORE PSW
0108	C9	360 ;	RET
		361 ;	
		362 ;	INTERRUPT HANDLER #1 TO WRITE COMMAND WORD
		363 ;	TO AM9511A WHEN AM9517A HAS FINISHED
		364 ;	LOADING THE OPERANDS
0109	F5	365 INTR1:	PUSH PSW ;SAVE PSW

Figure 7.4. DMA Interface Programming (Cont.)

LOC	OBJ	LINE	SOURCE STATEMENT
010A	3A0300	D 366	LDA OPCODE ;GET OPCODE
010D	D3C1	367	OUT APUCR ;WRITE TO COMMAND REGISTER
010F	F1	368	POP PSW ;RESTORE PSW
0110	FB	369	EI ;RE-ENABLE CPU INTERRUPTS
0111	C9	370	RET
		371 ;	
		372 ;	INTERRUPT HANDLER #2 TO SET DONE FLAG
		373 ;	TO INDICATE OPERATION IS COMPLETE
		374 ;	
0112	F5	375 INTR2:	PUSH PSW ;SAVE PSW
0113	3E01	376	MVI A,1
0115	320400	D 377	STA DONE2 ;SET DONE FLAG
0118	F1	378	POP PSW ;RESTORE PSW
0119	FB	379	EI ;RE-ENABLE CPU INTERRUPTS
011A	C9	380	RET
		381 ;	
		382 ;	RAM AREA
		383 ;	
		384	DSEG
		385 ;	
0003		386 OPCODE: DS 1	;APU OPCODE SAVE AREA
0004		387 DONE2: DS 1	;DONE FLAG
		388 ;	
		389	END

PUBLIC SYMBOLS

EXTERNAL SYMBOLS

USER SYMBOLS

APUCR A 00C1	APUDR A 00C0	APUSR A 00C1	CH2ADR A 00B4
CH2CNT A 00B5	CH3ADR A 00B6	CH3CNT A 00B7	CHK1 C 0036
CHK2 C 005B	CLR17 A 00BD	CMD17 A 00B8	DEMAND C 0000
DLOOP1 C 0008	DLOOP2 C 0017	DLOOP3 C 0025	DMAC A 00B0
DONE D 0002	DONE2 D 0004	EXEC C 00EF	ILOOP1 A 0040
INIT17 C 0090	INIT19 C 00B3	INTR1 C 0109	INTR2 C 0112
LLOOP1 C 0073	LLOOP2 C 0079	LOAD C 006C	MOD17 A 00BB
MSK17 A 00FF	OPCODE D 0003	PLOOP1 C 003E	PLOOP2 C 004D
PLOOP3 C 0062	POLL C 0030	REQ17 A 00B9	RST7 A 0038
RSTPTR D 0000	UICCR A 00C3	UICDR A 00C2	UICSR A 00C3

ASSEMBLY COMPLETE, NO ERRORS

Figure 7.4. DMA Interface Programming (Cont.)

CHAPTER 8 FLOATING POINT EXECUTION TIMES

8.1 INTRODUCTION

This chapter offers some numerical values of comparing execution times between Am9511A, Am9512 and their software counterparts. The software packages selected are the Intel FPAL LIB^(R) floating point library and the Lawrence Livermore Laboratory BASIC (LLL BASIC). These two software packages are selected because the Intel format is the same as the Am9512 single precision format and the LLL BASIC format is the same as the Am9511A floating point format. This should offer a reasonably comprehensive comparison.

In the execution-time cycles tables, the cycles given for the Am9511A and Am9512 are from the issue of the command to the completion of the command execution. The times for loading and unloading the operands are not included because these times depend on external hardware and also depend on whether the calculation is a chain calculation. Similarly, the software cycles are counted from the "Call" instruction to the "Ret" instruction of the floating point package. Operand setup time is also not counted.

The measurement is conducted on an Intel MDS 800^(R) system with an Advanced Micro Computers 95/6011 APU board and 95/6012 FPU board. The host is a 2-MHz 8080A. The clock for the 95/6011 or 95/6012 board is derived from the 9.8304-MHz bus clock divided by five to achieve a frequency of 1.96608 MHz. Because the main memory of the MDS 800 is dynamic, there is approximately $\pm 0.5\%$ uncertainty of software timing measurements. Because the bus clock is asynchronous to the CPU clock and the internal clock of the Am9511A and Am9512 is a two-phase clock derived from the single phase bus clock, there is a ± 2 -clock uncertainty in the hardware measurements.

8.2 FLOATING POINT ADD/SUBTRACT EXECUTION TIMES

Floating point add and subtract usually share the same routine. Floating point subtract is merely a change of sign of the subtrahend and is performed as floating point add. For the sake of discussion in this chapter, we assume the two operands are of like signs. If the operands are different signs, the discussion about addition will apply to subtraction and vice versa.

The execution time of floating point addition is mostly dependent on exponent alignment time of the two operands, maximum of

one shift would be required for post-normalization. If the addend and the augend have the same exponent, no exponent alignment time is required. If the magnitude of the addend and the augend are fairly close, only a few alignment shifts are required. If the addend and augend are very different, the number of required shifts is large, hence longer execution time.

The execution time of floating point subtraction not only has the same exponent alignment time as in the floating point addition, it also has a post-normalization time. Like floating point addition, the execution time lengthens as the magnitude of the minuend diverges from the magnitude of the subtrahend. Unlike the floating point add routine, the execution time also lengthens as the subtrahend approaches the value of the minuend. This is due to the number of left shifts required to produce a normalized result.

Table 8.1 shows the cycle times of Am9511A and LLL BASIC floating point add and subtract routines. Table 8.2 shows the cycle time of Am9512 and Intel floating point library execution times. The software execution times given have been normalized for a 2-MHz 8080A.

8.3 FLOATING POINT MULTIPLY/DIVIDE EXECUTION TIMES

Unlike floating point add or subtract, the execution times of floating point multiply or divide falls within a relatively narrow range and is not dependent on the relative magnitudes of the operands. Most multiplication algorithms use a shift and add method. For such algorithms, the execution time dependency is mainly on the number of 1's in the multiplier. The number of 1's in the multiplicand would not affect the execution time. The division execution time dependency is more complicated because of the number of division algorithms in use. In general, there is no simple way to predict the division execution time of a particular pair of operands (Tables 8.3 and 8.4).

8.4 DOUBLE-PRECISION FLOATING POINT EXECUTION TIMES

The Am9512 supports a double-precision (64-bit) floating point format. No known 64-bit floating point library routines are available at this time. Some sample execution times are given. The operands are selected over a representative range to give a comprehensive average (Tables 8.5 and 8.6).

TABLE 8.1. Am9511A vs LLL BASIC FLOATING POINT ADD/SUBTRACT EXECUTION TIME COMPARISON

OPERAND #1		OPERAND #2		AM9511		LLIBASIC	
DEC.	HEX.	DEC.	HEX.	FADD	FSUB	FADD	FSUB
5	03A00000	.0006	769D4951	214	228	3395	3884
5	03A00000	.006	79C49BA4	179	192	3000	3506
5	03A00000	.06	7CF5C28E	143	156	2608	3088
5	03A00000	.6	00999999	95	108	2100	2578
5	03A00000	6	03C00000	57	91	1826	2105
5	03A00000	60	06F00000	116	120	2362	2281
5	03A00000	600	0A960000	153	169	2540	2805
5	03A00000	6000	0DBE8000	189	204	2945	3186
123	07F60000	456	09F40000	103	108	2215	2137
.123	7DFBE76C	456	09E40000	213	227	3220	3467
123	07F60000	.456	7FE978D4	154	169	2748	3241
12345	0EC0E400	67890	11849900	106	131	2030	2460
1.3579	01ADCFAA	24680	0FC0D000	238	253	3469	3727
.000012	70C9539A	340000	13A60400	344	347	4783	5025
234	08EA0000	-678	8AA98000	118	96	2605	1920
-1.234	819DF3B6	12345	0EC0E400	238	229	3890	3367
			TOTAL	2660	2828	45736	48777
			AVERAGE	166.2	176.8	2858.5	3048.6

TABLE 8.2. Am9512 vs INTEL FPAL LIB FLOATING POINT ADD/SUBTRACT EXECUTION TIME COMPARISON

OPERAND #1		OPERAND #2		AM9512		FPAL.LIB	
DEC.	HEX.	DEC.	HEX.	SADD	SSUB	FADD	FSUB
5	40A00000	.0006	3A1D4952	254	275	2351	2568
5	40A00000	.006	3FC49BA6	229	217	1914	2152
5	40A00000	.06	3D75C28F	171	178	2506	2724
5	40A00000	.6	3F19999A	98	119	1954	2178
5	40A00000	6	40C00000	58	89	1430	1734
5	40A00000	60	42700000	128	123	2002	2165
5	40A00000	600	44160000	169	177	2455	2712
5	40A00000	6000	45BB8000	212	219	1866	2159
123	42F60000	456	43E40000	114	109	1844	2036
.123	3DFBE76D	456	43E40000	264	283	2145	2424
123	42F60000	.456	3FE978D4	192	183	1651	1878
12345	4640E400	67890	47849900	114	140	1889	2279
1.3579	3FADCFAB	24680	46C0D000	300	309	2435	2715
.000012	3749539B	340000	48A60400	475	477	1953	2231
234	436A0000	-678	C4298000	124	101	2155	1911
-1.234	BF9DF3B6	12345	4640E400	284	297	2564	2284
			TOTAL	3186	3296	33114	36150
			AVERAGE	199.1	206.0	2069.6	2259.4

TABLE 8.3. Am9511A vs LLL BASIC FLOATING POINT MULTIPLY/DIVIDE EXECUTION TIME COMPARISON

OPERAND #1		OPERAND #2		AM9511		LLLBASIC	
DEC.	HEX.	DEC.	HEX.	FMUL	FDIV	FMUL	FDIV
5	03A00000	.0006	769D4951	174	157	8451	13013
5	03A00000	.006	79C49BA4	174	178	8441	12856
5	03A00000	.06	7CF5C28E	149	177	8264	12867
5	03A00000	.6	00999999	174	157	8407	13302
5	03A00000	6	03C00000	173	178	8423	12835
5	03A00000	60	06F00000	148	179	8218	12892
5	03A00000	600	0A960000	173	155	8415	12214
5	03A00000	6000	0DBB8000	175	179	8437	13020
123	07F60000	456	09E40000	148	156	8939	12713
.123	7DFEE76C	456	09E40000	148	157	10948	13373
123	07F60000	.456	7FE978D4	149	155	8965	12878
12345	0EC0E400	67890	11849900	173	157	9163	14305
1.3579	01ADCFAA	24680	0FC0D000	147	179	10591	13149
.000012	70C9539A	340000	13A60400	149	157	10018	13395
234	08EA0000	-678	8AA98000	148	156	8781	13509
-1.234	819DF3B6	12345	0EC0E400	175	178	10971	12952
			TOTAL	2577	2655	145432	209273
			AVERAGE	161.1	165.9	9089.5	13079.6

TABLE 8.4. Am9512 vs INTEL FPAL LIB FLOATING POINT MULTIPLY/DIVIDE EXECUTION TIME COMPARISON

OPERAND #1		OPERAND #2		AM9512		FPAL.LIB	
DEC.	HEX.	DEC.	HEX.	SMUL	SDIV	FMUL	FDIV
5	40A00000	.0006	3A1D4952	234	250	3206	7757
5	40A00000	.006	3PC49BA6	256	235	3252	7905
5	40A00000	.06	3D75C28F	198	247	3088	7975
5	40A00000	.6	3F19999A	234	248	3245	7708
5	40A00000	6	40C00000	220	232	3052	7955
5	40A00000	60	42700000	200	246	2897	7999
5	40A00000	600	44160000	220	248	3072	7799
5	40A00000	6000	45BE8000	220	246	3137	7853
123	42F60000	456	43E40000	201	248	2903	7820
.123	3DFBE76D	456	43E40000	199	243	3087	7834
123	42F60000	.456	3EE978D4	219	236	3072	7822
12345	4640E400	67890	47849900	242	249	3124	7585
1.3579	3FADCFAB	24680	46C0D000	253	240	3139	7854
.000012	3749539B	340000	48A60400	219	228	3131	7776
234	436A0000	-678	04298000	201	234	2925	7721
-1.234	BF9DF3B6	12345	4640E400	223	227	3314	7852
			TOTAL	3539	3857	49644	125215
			AVERAGE	221.2	241.1	3102.8	7825.9

TABLE 8.5. Am9512 DOUBLE PRECISION ADD/SUBTRACT EXECUTION TIMES

OPERAND #1		OPERAND #2		AM9512	
DEC.	HEX.	DEC.	HEX.	DADD	DSUB
5	4014000000000000	.0006	3F43A92A30553261	1273	1310
5	4014000000000000	.006	3F789374BC6A7EF9	1174	1211
5	4014000000000000	.06	3FAEB851EB851EB8	1038	1105
5	4014000000000000	.6	3FE3333333333333	868	891
5	4014000000000000	6	4018000000000000	720	773
5	4014000000000000	60	404E000000000000	951	922
5	4014000000000000	600	4082C00000000000	1091	1107
5	4014000000000000	6000	40B7700000000000	1229	1244
123	405EC00000000000	456	407C800000000000	906	877
.123	3FFB7CED916872B0	456	407C800000000000	1233	1280
123	405EC00000000000	.456	3FDD2F1A9FBE76C8	1072	1103
12345	40C81C8000000000	67890	40F0932000000000	907	960
1.3579	3FF5F9F559B3D07C	24680	40D81A0000000000	1322	1352
.000012	3EE92A737110E453	340000	4114C08000000000	2158	2232
234	406D400000000000	-678	0085300000000000	914	861
-1.234	BFF3BE76C8B43958	12345	40C81C8000000000	1309	1290
			TOTAL	18165	18518
			AVERAGE	1135.3	1157.4

TABLE 8.6. Am9512 DOUBLE PRECISION MULTIPLY/DIVIDE EXECUTION TIMES

OPERAND #1		OPERAND #2		AM9512	
DEC.	HEX.	DEC.	HEX.	DMUL	DDIV
5	4014000000000000	.0006	3F43A92A30553261	1810	4857
5	4014000000000000	.006	3F789374BC6A7EF9	1814	4983
5	4014000000000000	.06	3FAEB851EB851EB8	1779	5048
5	4014000000000000	.6	3FE3333333333333	1841	5007
5	4014000000000000	6	4018000000000000	1785	4700
5	4014000000000000	60	404E000000000000	1751	4699
5	4014000000000000	600	4082C00000000000	1787	4618
5	4014000000000000	6000	40B7700000000000	1786	4702
123	405EC00000000000	456	407C800000000000	1750	4671
.123	3FBF7CED916872B0	456	407C800000000000	1756	4748
123	405EC00000000000	.456	3FDD2F1A9FEE76C8	1744	4936
12345	40C81C8000000000	67890	40F0932000000000	1807	4696
1.3579	3FF5F9F559B3D07C	24680	40D81A0000000000	1762	4788
.000012	3EE92A737110E453	340000	4114C08000000000	1755	4764
234	406D400000000000	-678	C085300000000000	1750	4670
-1.234	BFF3FE76C8B43958	12345	40C81C8000000000	1802	4768
			TOTAL	28479	76655
			AVERAGE	1779.9	4790.9

CHAPTER 9 TRANSCENDENTAL FUNCTIONS OF Am9511A

9.1 INTRODUCTION

The word "transcendental" is defined as "a function that cannot be expressed by a finite number of algebraic operations." Three examples of such functions are sine, logarithmic and exponentiation. The Am9511A performs a number of such functions, and this chapter describes the algorithms adopted by the device.

9.2 CHEBYSHEV POLYNOMIALS

Computer approximations of transcendental functions are often based on some form of polynomial equations, such as

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \dots$$

The most well-known polynomial for evaluating transcendental functions is the Taylor series

$$f(x) = f(a) + \frac{f'(a)(X-a)^k}{k!}$$

Where $f'(a)$ is the k^{th} derivative of the function f . Taylor series usually works well when $(x-a)$ is a small number. When the value of $(x-a)$ is large, the number of Taylor series terms required to evaluate to a given accuracy becomes large. To avoid this shortcoming, there is a set of approximating functions that not only minimizes the maximum error but also provides an even distribution of errors within the selected data representation interval. These are known as Chebyshev polynomial functions and are based upon the cosine functions. The Chebyshev polynomials $T(x)$ are defined as follows

$$T_n(x) = \cos(ncos^{-1}x)$$

The various terms of the Chebyshev series can be computed as

$$T_0(x) = \cos(0) = 1$$

$$T_1(x) = \cos(cos^{-1}x) = x$$

$$T_2(x) = \cos(2cos^{-1}x) = 2cos^2(cos^{-1}x) - 1 = 2x^2 - 1$$

in general

$$T_n(x) = 2x(T_{n-1}(x)) - T_{n-2}(x) \text{ for } n \geq 2$$

the terms $T_3(x)$, $T_4(x)$, $T_5(x)$ and $T_6(x)$ are given below for reference

$$T_3(x) = 4x^3 - 3x$$

$$T_4(x) = 8x^4 - 8x^2 + 1$$

$$T_5(x) = 16x^5 - 20x^3 + 5x$$

$$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1$$

It is not the intent of this book to go into the detailed derivation of the Chebyshev series. For readers interested in the formal derivation, references 1 and 3 are recommended. The Chebyshev series is given as follows:

$$f(x) = \frac{1}{2} C_0 + \sum_{n=1}^{\infty} C_n T_n(x)$$

where

$$C_n = \frac{2}{\pi} \int_{-1}^1 \frac{f(x) T_n(x)}{\sqrt{1-x^2}} dx$$

For a given accuracy, only a finite number of terms is required. The Am9511A selects the number of terms required by different functions to provide a mean relative error of about one part in 10^7 . The coefficients C_n are all precalculated and stored in the constant ROM.

Each of the transcendental functions in the Am9511A uses the Chebyshev polynomial series except the square root function. Each function is a three-step process as follows:

Range Reduction –

The input argument of the function is transformed to fall within a range of values for which the function can be computed to a valid result. For example, since functions like sine and cosine are periodic for multiples of radians, input arguments for these functions are converted to lie within a range of

$$0 \text{ to } \pi \text{ or } -\frac{\pi}{2} \text{ to } +\frac{\pi}{2}$$

Chebyshev polynomial evaluation –

This step is the same for all functions. The algebraic sum of the appropriate number of terms of the Chebyshev series is computed.

Postprocessing –

Some functions, such as sine and cosine, need postprocessing of the result such as sign correction.

The following sections give a detailed function-by-function description of each transcendental function in the Am9511A.

9.3 THE FUNCTIONS CHEBY AND ENTIER

Two functions are used in the following sections. The first one is CHEBY. This function evaluates the Chebyshev polynomial series

$$f(x) = 1/2C_0 + \sum_{k=1}^{n-1} C_k T_k(x)$$

The function is called by CHEBY (x, c, n) where x is the input argument after any necessary preprocessing; c is the coefficient list for the given function; and n is the number of Chebyshev polynomial terms used.

The FORTRAN program to implement the cheby function is as follows:

```
FUNCTION CHEBY (X, C, N)
```

```
  DIMENSION C(12), T(12)
```

```
  T(1) = 1
```

```
  T(2) = X
```

```
  CHEBY = 0.5 * X(1) + C(2) * T(2)
```

```
  DO 100 I = 3, N
```

```
    T(I) = 2 * X * T(I-1) - T(I-2)
```

```
  100 CHEBY = CHEBY + C(I) * T(I)
```

This program is not written to minimize execution time or code space but for its clarity. A program that improves execution speed but is somewhat more obscure is as follows:

```
FUNCTION CHEBY (X, C, N)
```

```
  DIMENSION C(12), T(12)
```

```
  B = 0
```

```
  D = C(N)
```

```
  X2 = 2 * X
```

```
  DO 100 I = N, 2, -1
```

```
    A = B
```

```
    B = D
```

```
  100 D = X2 * B - A + C(I-1)
```

```
  CHEBY = (D - A)/2
```

```
  END
```

The second function is called ENTIER. Entier is the French word for integer. The entier function is similar to the FORTRAN integer function, except the integer function rounds down to the nearest integer closer to zero whereas the entier function rounds down to the nearest integer of a lower value. In other words, if the number is greater than or equal to zero, both functions are identical. If the number is negative, such as -2.5, INT (-2.5) = -2, ENTIER (-2.5) = -3.

A FORTRAN program to implement the entier function is as follows:

```
FUNCTION ENTIER (X)
  IF (X.LT.0) X = X - 1
  ENTIER = INT (X)
END
```

9.4 SINE

Any argument of the sine function can be reduced to a value from $-\pi/2$ to $+\pi/2$. Hence the range reduction is

```
X = X * 2/π
X = X - 4 * Entier ((X + 1)/4)
If (X.GT.1) X = 2 - X
```

This reduces the input argument to a range from -1 to +1. The Chebyshev polynomial evaluation is

$\text{Sin}(X) = X * \text{CHEBY}(2X^2 - 0.1, \text{Csin}, \text{Nsin})$

there Csin is an array of precalculated Chebyshev coefficients for sine, and Nsin is the number of Chebyshev polynomial series used. In the case of Am9511A

```
Nsin = 6
Csin0 = 2.5525579
Csin1 = -0.2852616
Csin2 = 9.118016 x 10-3
Csin3 = -1.365875 x 10-4
Csin4 = 1.184962 x 10-6
Csin5 = -6.702792 x 10-9
```

9.5 COSINE

Any argument of cosine function can be reduced to a range from 0 to π . Hence, the formulas for cosine range reduction are

```
X = X * 2/π
X = 4 * Entier ((X + 2)/4) - X + 1
If (X.GT.1) X = 2 - X
```

The cosine function is now evaluated the same way as the sine function

$\text{cos}(x) = X * \text{CHEBY}(2x^2 - 1, \text{Csin}, \text{Nsin})$

where Csin and Nsin are the same as the sine function

9.6 TANGENT

Any argument for tangent can be reduced to a value from $-\pi/2$ to $+\pi/2$. This is the same range reduction algorithm as the sine function (Figure 9.1).

```
X = X * 2/π
X = X - 4 * Entier ((X + 1)/4)
Y = X
If (Y.GT.1) X = 2 - X
```

The Chebyshev polynomial evaluation is

$\text{Tan}(X) = X * \text{CHEBY}(2X^2 - 1, \text{Ctan}, \text{Ntan})$

A postprocessing step is also required

If (Y.GT.1)Tan(X) = 1/Tan(X)

The constants used in the Am9511A are as follows:

```
Ntan = 9
Ctan0 = 1.7701474
Ctan1 = 1.0675393 x 10-1
Ctan2 = 7.5861016 x 10-3
Ctan3 = 5.4417038 x 10-4
Ctan4 = 3.9066370 x 10-5
Ctan5 = 2.8048161 x 10-6
Ctan6 = 2.0137658 x 10-7
Ctan7 = 1.4458187 x 10-8
Ctan8 = 1.0380510 x 10-9
```

9.7 ARCSINE

The argument of arcsine must be less than or equal to 1, or else an input error is detected. Hence, range reduction is not necessary.

There are two different Chebyshev polynomial expansion used depending on the initial value of X. If $X^2 \leq 1/2$ then the following formula is used

$\text{Asin}(X) = x * 2 * \text{CHEBY}(4x^2 - 1, \text{Casin}, \text{Nasin})$

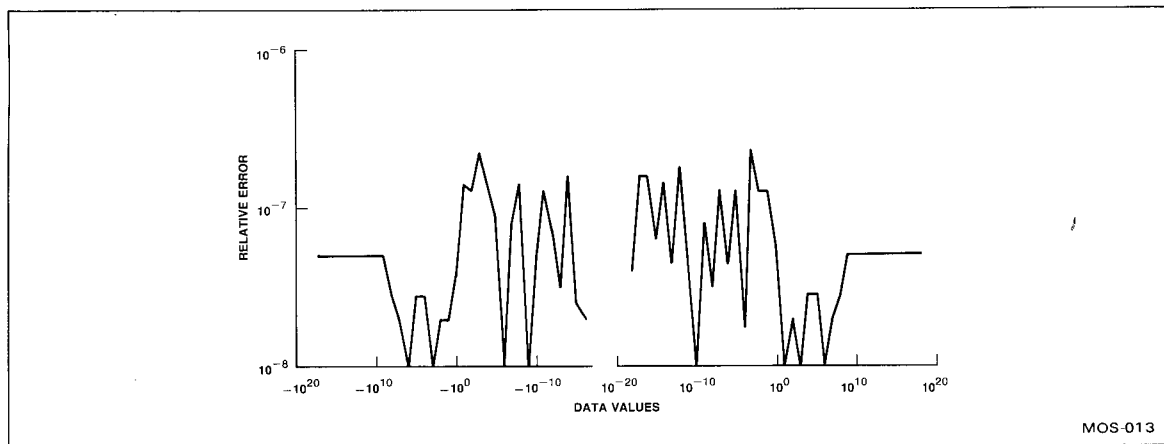


Figure 9.1. Tangent

If $1/2 < x^2 \leq 1$ then

$$\text{Asin}(X) = \text{sign}(X) * \frac{\pi}{2} * \sqrt{2 - 2x^2} * \text{CHEBY}(3 - 4x^2, \text{Casin}, \text{Nasin})$$

Where sign(X) is the sign of X. The values of Casin and Nasin used in the Am9511A are as follows:

- Nasin = 10
- Casin₀ = 1.4866665
- Casin₁ = 3.8853034 x 10⁻²
- Casin₂ = 2.8854414 x 10⁻³
- Casin₃ = 2.8842183 x 10⁻⁴
- Casin₄ = 3.3223672 x 10⁻⁵
- Casin₅ = 4.1584779 x 10⁻⁶
- Casin₆ = 5.4965045 x 10⁻⁷
- Casin₇ = 7.5500784 x 10⁻⁸
- Casin₈ = 1.0671938 x 10⁻⁸
- Casin₉ = 1.5421800 x 10⁻⁹

9.8 ARCCOSINE

The arccosine is obtained from arcsine by using the trigonometric identity.

$$\text{Arccosine}(x) = \frac{\pi}{2} - \text{arcsine}(x)$$

9.9 ARCTANGENT

The range reduction of the arctangent function involves taking the reciprocal of the input argument if the absolute value of the input argument is greater than 1.

$$U = X$$

$$\text{If } (\text{ABS}(U) \cdot \text{GT.}1) X = 1/X$$

The Chebyshev polynomial evaluation is

$$\text{Atan}(X) = X * \text{Cheby}(2X^2 - 1, \text{Catan}, \text{Natan})$$

The postprocessing requirement is

$$\text{If } (U \cdot \text{GT.}1) \text{Atan}(X) = \pi/2 - \text{Atan}(X)$$

$$\text{If } (U \cdot \text{LT.}-1) \text{Atan}(X) = -\pi/2 - \text{Atan}(X)$$

The value of Natan and Catan used in the Am9511A are:

- Natan = 11
- Catan₀ = 1.7627472
- Catan₁ = -1.0589292 x 10⁻¹
- Catan₂ = 1.1135842 x 10⁻²
- Catan₃ = -1.3811950 x 10⁻³
- Catan₄ = 1.8574297 x 10⁻⁴
- Catan₅ = -2.6215196 x 10⁻⁵
- Catan₆ = 3.8210366 x 10⁻⁶
- Catan₇ = -5.6991862 x 10⁻⁷
- Catan₈ = 8.6488779 x 10⁻⁸
- Catan₉ = -1.3303384 x 10⁻⁸
- Catan₁₀ = 2.0685060 x 10⁻⁹
- Catan₁₁ = -3.2448600 x 10⁻¹⁰

9.10 EXPONENTIATION (Figure 9.2)

The range reduction for the exponentiation function is performed by the following formulas

$$X = X * \text{Log}_2 e$$

$$N = 1 + \text{Entier}(X)$$

The Chebyshev polynomial evaluation is

$$\text{Exp}(X) = 2^N * \text{Cheby}(2*(N - X) - 1, \text{Cexp}, \text{Nexp})$$

No postprocessing is required for the exponentiation function. The values of Nexp and Cexp used by Am9511A are:

- Nexp = 8
- Cexp₀ = 1.4569999
- Cexp₁ = -2.4876243 x 10⁻¹
- Cexp₂ = 2.1446556 x 10⁻²
- Cexp₃ = -1.2357141 x 10⁻³
- Cexp₄ = 5.3453058 x 10⁻⁵
- Cexp₅ = -1.8506907 x 10⁻⁶
- Cexp₆ = 5.3411877 x 10⁻⁸
- Cexp₇ = -1.3215160 x 10⁻⁹

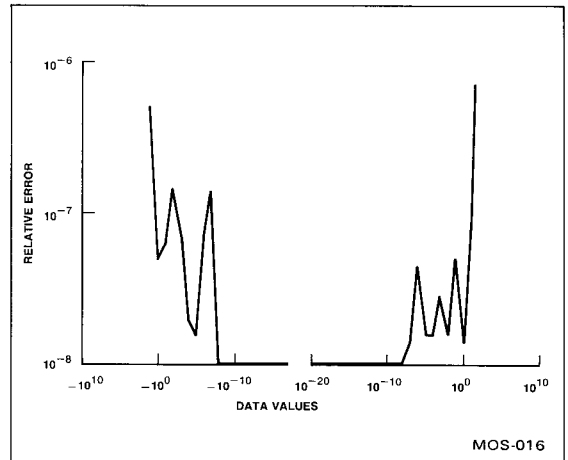


Figure 9.2. e^x

9.11 NATURAL LOGARITHM (Figure 9.3)

Any input argument to a logarithm function that is less than or equal to zero will be returned as an error input. No preprocessing or postprocessing is necessary for all positive input X.

$$\text{LN}(X) = \text{CHEBY}(4 * \text{Mant}(X) - 3, \text{CLN}, \text{NLN}) + (\text{Expo}(X) - 1) * \text{LN}2$$

Where Mant(X) is the mantissa value of X and expo(X) is the exponent value of X.

The value of NLN and CLN used in the Am9511A are:

- NLN = 11
- CLN₀ = 7.5290563 x 10⁻¹
- CLN₁ = 3.4314575 x 10⁻¹
- CLN₂ = -2.9437253 x 10⁻²
- CLN₃ = 3.3670893 x 10⁻³
- CLN₄ = -4.3327589 x 10⁻⁴
- CLN₅ = 5.9470712 x 10⁻⁵
- CLN₆ = -8.5029675 x 10⁻⁶
- CLN₇ = 1.2504674 x 10⁻⁶
- CLN₈ = -1.8772800 x 10⁻⁷
- CLN₉ = 2.8630251 x 10⁻⁸
- CLN₁₀ = -4.4209570 x 10⁻⁹

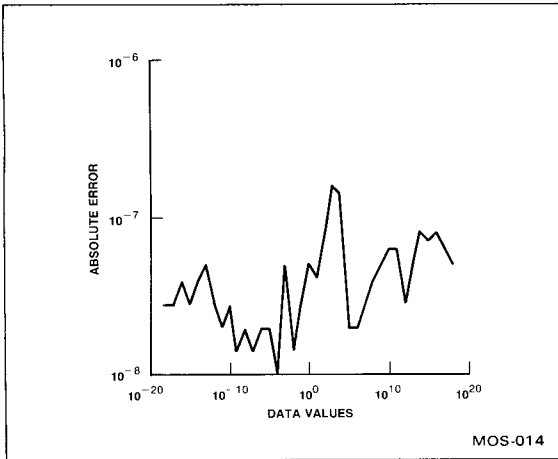


Figure 9.3. Natural Logarithm

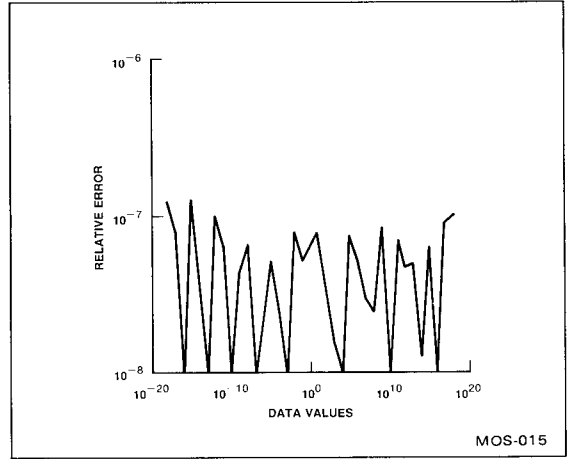


Figure 9.4. Square Root

9.12 LOGARITHM TO BASE 10 (COMMON LOGARITHM)

The common logarithm is derived from the natural logarithm by the equation

$$\text{LOG}(X) = \text{LN}(X) * \text{LOG}_{10}e$$

where

$$\text{LOG}_{10}e = 0.4342945$$

9.13 X TO THE POWER OF Y

The function X to the power of Y is derived from the following equation

$$X^Y = e^{(Y * \text{LN}(X))}$$

9.14 SQUARE ROOT

The square root function (Figure 9.4) in the Am9511A is the only derived function that does not use the Chebyshev polynomials. It uses a combination of linear approximation and the Newton-Rafelson successive approximation methods. The square root algorithm adopted is divided into three parts:

(a) Range reduction –

The input argument is divided into the exponent and the mantissa. If the exponent is odd, the exponent is incremented by 1 and the mantissa is divided by 2. If the input exponent is even, the above step is skipped.

(b) Linear Approximation –

The mantissa is now a number greater than or equal to 1/4 and less than 1. The curve line in Figure 9.5 represents the square root of all numbers between 1/4 and 1. The straight line represents the first-order approximation for the square root of the number. To select the best straight line, we must minimize the maximum relative error between the straight line and the curve line. This would reduce the worst case error to a minimum. This line is known as the minimax line.

The method used to compute the best linear approximation line is as follows:

- Let m = Slope of the minimax line
- Let b = Y intercept of the minimax line
- Let Y = The function of the minimax line

such that

$$Y = mx + b$$

The relative error between the actual square root value and the first-order approximation is

$$E(X) = \frac{mx + b - \sqrt{x}}{\sqrt{x}}$$

Figure 9.5 shows that the absolute value of E(x) is a maximum at the two extremities (x = 1/4 and x = 1) and at a point where the slope of the curve E(x) = 0, or dE/dx = 0.

$$\frac{dE}{dx} = \frac{d}{dx} \left(\frac{mx + b - \sqrt{x}}{\sqrt{x}} \right) \quad (9.1)$$

$$= \frac{d}{dx} mx^{1/2} + \frac{d}{dx} bx^{-1/2} - \frac{d}{dx} (1)$$

$$= m \frac{d}{dx} x^{1/2} + b \frac{d}{dx} x^{-1/2} - 0$$

$$= \frac{1}{2} mx^{-1/2} - \frac{1}{2} bx^{-3/2} = 0$$

therefore

$$mx^{1/2} = bx^{-3/2}$$

$$x = \frac{b}{m}$$

The relative errors at the extremities are given by

$$E\left(\frac{1}{2}\right) = \frac{\frac{m}{4} + b - \sqrt{\frac{1}{4}}}{\sqrt{\frac{1}{4}}} = \frac{\frac{m}{4} + b - \frac{1}{2}}{\frac{1}{2}} = \frac{m}{2} + 2b - 1 \quad (9.2)$$

$$\Sigma(1) = \frac{m + b - \sqrt{1}}{\sqrt{1}} = m + b - 1 \quad (9.3)$$

The minimax line requires these maximum errors to be equal

$$\frac{m}{2} + 2b - 1 = m + b - 1$$

$$b - \frac{m}{2} = 0$$

$$\frac{b}{m} = \frac{1}{2} \quad (9.4)$$

$$m = 2b \quad (9.5)$$

from equations 9.1 and 9.4

$$x = \frac{b}{m} = \frac{1}{2}$$

Therefore, the maximum error in the middle occurs when $X = 1/2$. The minimax line requires these errors to be equal in magnitude. Thus

$$E\left(\frac{1}{4}\right) = E(1) = -E\left(\frac{1}{2}\right)$$

$$E\left(\frac{1}{2}\right) = \frac{\frac{m}{2} + b - \sqrt{\frac{1}{2}}}{\sqrt{\frac{1}{2}}} \quad (9.6)$$

Since $m = 2b$ from equation 9.5

$$E\left(\frac{1}{2}\right) = \frac{2b - \sqrt{\frac{1}{2}}}{\sqrt{\frac{1}{2}}} \quad (9.7)$$

From equations 9.3 and 9.5

$$E(1) = 3b - 1 \quad (9.8)$$

From equations 9.6, 9.7 and 9.8

$$2b - \sqrt{\frac{1}{2}} = -(3b - 1) = 1 - 3b$$

$$\sqrt{\frac{1}{2}}$$

$$2\sqrt{2}b - 1 = 1 - 3b$$

$$b = \frac{2}{2\sqrt{2} + 3} = 0.34314575$$

From 9.5

$$m = 2b = 0.6829150$$

Therefore, the minimax line is given by

$$Y = 0.68629150X + 0.34314575$$

This is the equation used in Am9511A for the first-order linear approximation. Therefore

$$X_0 = 0.68629150X + 0.34314575$$

(c) Newton-Ralfson successive approximation -

After the first-order approximation (X_0) is obtained, the Am9511A executes two iterations of the Newton-Ralfson approximation

$$X_1 = (X/X_0 + X_0)/2$$

$$X_2 = (X/X_1 + X_1)/2$$

And the result is given by

$$\text{SQRT}(X) = x_2 * 2^{E/2}$$

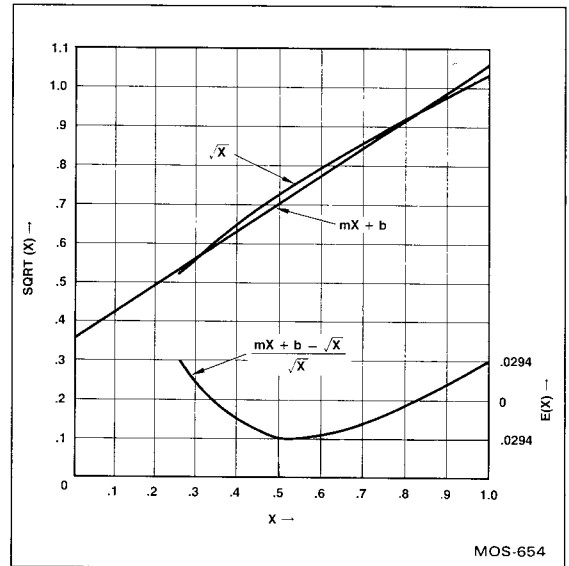


Figure 9.5. Square Root Computation

A FORTRAN function to illustrate the above algorithm is given below:

```

FUNCTION ROOT (X)
INTEGER EXPO, LSB
REAL MANT, X0, X1, X2
EXPO = INT (LOG(X)/LOG(2)) + 1
MANT = X/2**EXP
LSB = MOD(EXPO, 2)
IF (LSB.EQ.0) GOTO 100
EXPONENT IS ODD
EXPO = EXPO + 1
MANT = MANT/2.0
100 X0 = 0.68629150* MANT + 0.34314575
X1 = (X/X0 + X0)/2.0
X2 = (X/X1 + X1)/2.0
Root = (2**(EXPO/2))*X2
End
    
```

9.15 DERIVED FUNCTION ERROR PERFORMANCE

Since each of the derived functions is an approximation of the true function, results computed by the Am9511A are not always exact. In order to quantify the error performance of the component more comprehensively, the following graphs have been prepared. Each function has been executed with a statistically significant number of diverse data values, spanning the allowable input data range, and resulting errors have been tabulated. Absolute errors (that is, the number of bits in error) have been converted to relative errors according to the following equation:

$$\text{Relative Error} = \frac{\text{Absolute Error}}{\text{True Result}}$$

This conversion permits the error to be viewed with respect to the magnitude of the true result. This provides a more objective measurement of error performance since it directly translates to a measure of significant digits of algorithm accuracy.

For example, if a given absolute error is 0.0001 and the true result is also 0.0001, it is clear that the relative error is equal to 1.0 (which implies that even the first significant digit of the result is wrong). However, if the same absolute error is computed for a true result of 10000.0, then the first six significant digits of the result are correct ($0.001/10000 = 0.0000001$).

Each of the following graphs was prepared to illustrate relative algorithm error as a function of input data range. Natural logarithm is the only exception; since logarithms are typically additive, absolute error is plotted for this function.

Two graphs have not been included in the following figures: common logarithms and the power function (X^Y). Common logarithms are computed by multiplication of the natural logarithms by the conversion factor 0.43429448 and the error function is therefore the same as that for natural logarithm. The

power function is realized by combination of natural log and exponential functions according to the equation

$$X^Y = e^{Y \ln(X)}$$

The error for the power function is a combination of that for the logarithm and exponential functions. Specifically, the relative error for PWR is expressed as

$$RE_{PWR} = RE_{EXP} + X(AE_{\ln})$$

where

RE_{PWR} = relative error for power function

RE_{EXP} = relative error for exponential function

AE_{\ln} = absolute error for natural logarithm

X = value of independent variable in X^Y

REFERENCES

1. Pennington, Ralph H. *Introduction to Computer Methods and Numerical Analysis*. Macmillan Company, 1970.
2. Clenshaw, Miller and Woodger. "Algorithms for Special Functions (I and II)," *Numerische Mathematic*, 1963.
3. Parker, Richard O. and Joseph H. Kroeger. *Algorithm Details for the Am9511A Arithmetic Processing Unit*. Advanced Micro Devices, 1978.