The ability of a digital computer to handle mathematical operations combined with its ability to manipulate text gives the machine a unique combination of power that partially accounts for its growing popularity. Programming a computer using machine language to perform mathematical functions is perhaps a bit more complicated than having it perform routine text manipulations. But, it is not as difficult as some people tend to think before being introduced to the subject. Like most other programming tasks, the key to success is organization of the program into small routines that can be built upon to form more powerful functions.

The instruction set of the '8008' and similar CPU's contain a number of primary mathematical instructions that are the basis for developing mathematical programs. The groups used most often include the ADDITION, SUBTRACTION and ROTATE instructions. (Do you recall that rotating a binary number to the left effectively doubles, or multiplies the original value by two, and rotating it to the right essentially divides the original value in half?)

Dealing with numbers of small magnitude using a microprocessor is simplicity itself. For instance, if one wanted to add the numbers '2' and '7,' one could load one number into register B in the CPU and load the other into the accumulator. The simple directive:

```
ADB
```

would result in the value '011' (octal) being left in the accumulator. Subtraction is just as easy. If one placed '7' in the accumulator and '2' in register B and executed a:

```
SUB
```

instruction the value '5' would be left in the accumulator.

Multiplication of small numbers may be readily accomplished using a simple algorithm. That is to add the multiplicand to itself the number of times dictated by the multiplier. Suppose one desired to have the computer multiply '2' times '3.' Placing the value '2' in register B and '3' in register C and executing the following instruction sequence:

```
START,  XRA
MULTIP, ADB
        DCC
        JFZ MULTIP
STOP,   HLT
```

would result in the value '6' ending up in the accumulator. As shall be discussed further on, the above algorithm is not very efficient when the numbers become large. More efficient multiplication algorithms are based on rotate operations which effectively multiply a number by a power of two. For instance, multiplying a number by '32' (decimal) would require 32 loops through the above routine. It would only require five rotate operations! However, the above routine illustrates how a number can be multiplied even though the computer does not have a specific "multiply" instruction.

One may also divide small valued numbers that have integer results using a similarly simple algorithm that subtracts instead of adds. For instance, a reverse of the previous example would be to divide the number '6' by the value '2.' The subtraction algorithm could appear as:

```
START,  LCI 000
DIVIDE, NDA
        JTZ STOP
        SUB
        INC
        JMP DIVIDE
STOP,   HLT
```

In the algorithm just presented, the routine starts with the number '6' in the accumulator. The divisor is in register B. Register C is used as a counter to count how many times the value in B can be subtracted before the contents of the accumulator reaches zero. As pointed out previously, the algorithm only works properly if the result is an integer value. Division is perhaps the most difficult basic mathematical function to perform on a digital computer because of mathematical peculiarities (involving the manipulation of fractional values). However, as will be illustrated later, there are ways around the above limitation. The above illustration is merely to give the novice encouragement. It illustrates that such operations are possible even though a specific divide command is not part of a typical microprocessor's instruction set!

The discussion so far has been limited to numbers of relatively small magnitude. Specifically, numbers small enough to be contained in a single eight bit binary register or memory location in a microprocessor. Many user's who want to use the digital computer to perform mathematical operations seem to get stumped when first coming across a requirement to manipulate numbers that are too large in magnitude to fit in one memory word or CPU register. With an '8008' based machine, and indeed most microcomputers available at present, such a requirement typically arrives shortly after one has started operating their machine! The reason is simply that the largest valued number that can be placed in an 'N-bit' register is the value (2**N)-1. Since most microprocessors use but eight bits in a word, the largest number that can be represented in a single word if all the bits are used is a mere 255 (decimal). If one desires to maintain the "sign" (whether the value is greater or less than zero) and uses one bit in a register for that purpose, then the largest number that can be represented in a single word is a paltry 127 (decimal). That is hardly enough to bother using a computer to manipulate!

But, the secret to rapidly increasing the magnitudes of the numbers that can be handled by a digital computer is held in that formula just presented; (2**N)-1. That formula states that the size of the number that can be stored in a binary register doubles for every bit added to the register. Thus, if one were to store a number using the available bits in two registers or memory words in an 8-bit-per-word system, one would be able to represent numbers as large as (2**16)-1 or 65,535 (decimal). If one of those 16 bits was reserved for a sign indicator, the magnitude would be limited to (2**15)-1 which is 32,767 (decimal). That is certainly a lot more than the value of 127 that can be held in just one word! But, why stop at holding a number in two words? There is no need to, one may keep adding words to build up as many bits as desired. Three words of eight bits, leaving one bit out for a sign indicator, would leave 23 bits. That number of bits could represent numbers as large as (2**23)-1 which is about 8,388,607 (decimal). Four words would allow representing a signed number up to (2**31)-1 which is roughly 1,107,483,647! One could add still more words if required.. Generally, however, one selects the number of significant digits that will be important in the calculations that are to be performed and uses enough words to ensure that the "precision" or number of significant digits required for the operations, can be represented in the total number of bits available within the grouped words. The use of more than one computer word or normal sized register to store and manipulate numbers as though they were in one large continuous register is commonly referred to as "multiple-precision" arithmetic. One often hears computer technologists speaking of "double-precision" or "triple-precision" arithmetic. This simply means that the machine is using techniques (generally programming techniques) that enable it to handle numbers stored in two or three registers as though they were one number in a very large register.

The '8008' CPU is capable of multiple-precision arithmetic. In fact it does it quite nicely because the designers of the CPU took particular care to include some special in-

structions for just such operations. (Such as the ADD and SUBTRACT with CARRY instructions.) Multiple-precision arithmetic is not difficult. It takes a little extra consideration when organizing a program to handle and store numbers that are contained in multiple words in memory. But, with the use of effective subroutines (and "chaining") the task can be handled with relative ease.

In order to effectively deal with multiple-precision arithmetic one must establish a convention for storing the sections of one large number in several locations. For the purposes of the current discussion, it will be assumed that triple-precision arithmetic is to be performed. Numbers will be stored in three consecutive memory locations according to the following arrangement.

Location N = Least significant 8 bits
Location N+1 = Next significant 8 bits
Location N+2 = Most significant 7 + sign bit

Thus, the three words in memory could be mentally viewed as being one continuous large register containing 23 binary bits plus a sign bit as illustrated in the diagram below.

LOC N+2	LOC N+1	LOC N
sx xxx xxx	xx xxx xxx	xx xxx xxx
MSB's	NSB's	LSB's

Of course, one could reverse the above sequence, and store the least significant bits in memory location 'N,' the next group in 'N+1' and the most significant bits plus sign bit in memory location 'N+2.' It makes little difference as long as one remains consistent within a program. However, the convention illustrated will be the one used for the discussion in this section.

Also, as has been pointed out, it is not necessary to limit the storage to just three words. Additional words may be used if additional precision is required. For most of the discussion in this chapter, three words will be used for storing numbers. Using three words in the above fashion will allow numbers up to a value of 8,388,647 to be stored. This means that six to seven significant digits (decimal) can be maintained in calculations.

The first multiple-precision routine to be illustrated will be an addition routine that will add together two multiple-precision numbers and leave the result in the location formerly occupied by one of the numbers. The routine to be presented has been developed as a general purpose routine in that, by properly setting up memory address pointers and loading a CPU register with a precision value prior to calling the routine, the same routine may be used to handle multiple-precision addition of numbers varying in length from '1' to 'N' registers. (As long as the registers containing a number are in consecutive order in memory, and with the restriction that all the registers containing a number are on one page. That limits 'N' to 255 (decimal) words, which is a limitation few programmers will find cumbersome!)

The key element in the addition routine to be illustrated is the use of the ACM add with carry instruction. The essential difference between an add with carry (ACM) instruction and an ADM (add without carry) command is as follows.

An ADM instruction simply adds the contents of the accumulator and the contents of the memory location pointed to by the H and L registers. During the addition process, the status of the carry flag is ignored. However, if at the end of the process, an overflow has occured, the carry flag will be set to a logic one condition. For example, adding the following binary numbers would yield:

```
          1 0 1 0 1 0 1 0
          0 1 0 1 0 1 0 1
          ---------------
CARRY = 0 : 1 1 1 1 1 1 1 1
```

An example illustrating a carry occuring is shown next.

```
                  11 111 111
                  00 000 001
                  ----------
CARRY = 1 :       00 000 000
```

Remember in the above examples that the CARRY FLAG is only affected by an overflow condition after the operation has occured. The original condition of the carry flag will have no effect on the final results of the calculation.

An ACM command, on the other hand, examines the contents of the CARRY FLAG prior to the addition operation and considers it as an operator on the least significant bit position. At the end of the addition process, the carry flag is again set or cleared depending on whether or not an overflow occured. (As in the case for the ADM instruction discussed above.) For example, adding the following binary numbers yields results that differ depending on the INITIAL status of the carry flag.

```
CASE 1A:                    0 : 'C' FLAG initially cleared
                  10 101 010
                  01 010 101
                  ----------
CARRY = 0 :       11 111 111

CASE 1B:                    1 : 'C' FLAG initially set
                  10 101 010
                  01 010 101
                  ----------
CARRY = 1 :       00 000 000

CASE 2A:                    0 : 'C' FLAG initially cleared
                  11 111 111
                  00 000 001
                  ----------
CARRY = 1 :       00 000 000

CASE 2B:                    1 : 'C' FLAG initially set
                  11 111 111
                  00 000 001
                  ----------
CARRY = 1 :       00 000 001
```

In summary, one can see that an ACM type of instruction makes multiple-precision addition extremely easy. This is because the carry bit acts as a link between any carry from the most significant bit of one addition operation into the least significant bit of the next addition operation. This allows one to proceed just as though the addition operation was being performed in one long register instead of several short registers. To discern this clearly, examine the example provided next which first illustrates an addition operation being performed in a hypothetical 16 (decimal) bit register, then shows the same result when two ACM operations are performed on two eight bit registers "linked" by the special capabilities of the ACM instruction.

ADDITION IN HYPOTHETICAL 16 BIT REGISTER

```
                    11 111 111  10 101 010
                    00 000 000  11 010 101
                    -----------------------
CARRY = 1 :         00 000 000  01 111 111
```

SAME OPERATION USING ACM INSTRUCTION & TWO 8 BIT REGISTERS

```
                                        0 : 'C' FLAG assumed cleared
FIRST ACM OPERATION:        10 101 010
                            11 010 101
                            ----------
CARRY = 1 :                 01 111 111  : LSB's in memory location N

                                        1 : 'C' FLAG set by above add
SECOND ACM OPERATION:       11 111 111
                            00 000 000
                            ----------
CARRY = 1 :                 00 000 000  : MSB's in location N+1
```

Placing the results of the two eight bit registers side-by-side after using the ACM type of instruction yields the same result as though the operation had been performed in a sixteen bit register. The concept can be applied to as many eight bit registers as desired.

Armed with the knowledge of how the powerful ACM type of instruction operates, one may proceed to develop a N'th precision addition subroutine. Examine the following routine.

```
ADDER,  NDA          Always clear carry flag at routine entry
ADDMOR, LAM          Get first number into accumulator
        CAL SWITCH   Change pointers to second number
        ACM          Perform ADDITION with CARRY
        LMA          Place result back into memory
        DCB          Decrement the "precision" counter
        RTZ          Exit routine when counter reaches '0'
        INL          Advance second number pointer
        CAL SWITCH   Change pointer back to first number
        INL          Advance first number pointer
        JMP ADDMOR   Repeat process for next precision
```

Note that the above ADDER subroutine requires that a number of the CPU registers be setup prior to calling the routine. The H and L registers must contain the address of the least significant bits register (memory location) for the first multi-word number. Registers D and E similarly must be setup to contain the address of the least significant part of the second multi-word number that is to be added to the first. Finally, register B must be initialized to the "precision," or number of memory words used to hold a multi-word number. Suppose, for example, that a number in triple-precision format is stored in three words starting at location 100 on page 00 and that a second number in similar format is stored at location 200 on page 01. The following instructions would be used

to setup the CPU registers prior to calling the ADDER subroutine just described.

```
INIT,   LHI 000      Set page for LSW of first number
        LLI 100      Set location on page for LSW of 1'st number
        LDI 001      Set page for LSW of second number
        LEI 200      Set location on page for LSW of 2'nd number
        LBI 003      Set "precision" value (three words this case)
        CAL ADDER    Call the N'th precision addition routine
```

Note too, that the ADDER subroutine is destructive to the original value of the second number because the answer is left in those locations. If, for some reason, the user wanted to save the original value of the second number, then it would have to be saved elsewhere in memory prior to performing the multi-precision addition operation.

Just as there are two classes of instructions for performing addition with an '8008' CPU, one of which (ACM category) is suited for multiple-precision arithmetic, there are two classes of subtract commands. The SUM (SUBTRACT WITHOUT CARRY) and the SBM (SUBTRACT WITH CARRY, or more appropriately, SUBTRACT WITH BORROW). The SBM type works similarly to the ACM type previously discussed. The CPU first checks the status of the carry flag before performing the subtraction operation. It is thus an easy matter to perform multiple-precision subtraction operations. In fact, one can set up aan almost identical routine to that just described for addition. As in the addition example, one would first setup CPU registers as pointers to the least significant portions of the multiple-precision numbers and load register B with the number of memory words (N) occupied by a N'th precision number.

The routines presented here only utilize the ACM or SBM instructions because the algorithms have been developed as general purpose routines to handle strings of numbers in memory. The reader is reminded that there are a whole group of instructions that have similar functions for working with data while it is in the various CPU registers (such as the ACB, ACC, ACD.... instructions). In addition, there is also the ACI instruction for performing an addition operation with an IMMEDIATE data word. The reader may review the appropriate section in Chapter 1 for a summary of the variations possible when using an '8008' CPU.

EXAMPLE OF AN N'th PRECISION SUBTRACTION SUBROUTINE

```
SUBBER,  NDA          Always clear carry flag at start of routine
SUBTRA,  LAM          Get first number into accumulator
         CAL SWITCH   Change pointers to second number
         SBM          Subtract second from first with borrow
         LMA          Place result back into memory
         DCB          Decrement the "precision" counter
         RTZ          Exit subroutine when counter is '0'
         INL          Advance second number pointer
         CAL SWITCH   Change pointer back to first number
         INL          Advance first number pointer
         JMP SUBTRA   Repeat process for next part of number
```

One thing a person dealing with mathematical functions on a computer will soon

have to be concerned with is what happens when a larger number is subtracted from a smaller number. The answer is naturally a minus or negative number. As was initially discussed in the chapter on fundamental programming skills, most microprocessors handle negative numbers utilizing the "two's complement" convention. The reader may want to review the first few pages of that section at this time.

If, for instance, (using single-precision arithmetic) the number '8' (decimal) was subtracted from '6,' the result would appear in the accumulator as shown here:

```
6 decimal = 0 0 0 0 0 1 1 0 binary
8 decimal = 0 0 0 0 1 0 0 0 binary
subtracted ---------------------
    is      1 1 1 1 1 1 1 0 binary
```

Note that the most significant bit in the register containing the minus answer is a '1.' By establishing a "two's complement" convention and always ensuring that the magnitude of any number handled does not interfere with the most significant bit, one may quickly determine whether a number in a register (or series of registers in the case of multiple-precision formatting) is positive or negative. This may be accomplished by testing to see if the most significant bit is a '1' (for a negative number) or '0' (for a positive) value. This is done in an '8008' or similar microprocessor by testing the SIGN FLAG with a JFS, CTS, or similar type CONDITIONAL instruction.

Remember too, that a number may be subtracted from another number by forming the two's complement of the number to be subtracted, then performing an addition operation. Thus:

```
+8 decimal = 0 0 0 0 1 0 0 0 binary

2's comp   = 1 1 1 1 1 0 0 0 binary

          consequently

6 decimal  = 0 0 0 0 0 1 1 0 binary
2's comp of 8 1 1 1 1 1 0 0 0 binary
when added  ---------------------
    is       1 1 1 1 1 1 1 0 binary
```

It is often desirable to perform a straight two's complement operation on a number in order to change it from a positive to a negative number (or the reverse). One easy way to accomplish this is to simply subtract the number from a value of zero. For multiple-precision work one could simply load one string of memory locations (the first number) with zeroes and place the number to be "negated" in the second string of memory locations. Then simply call the previously illustrated SUBBER subroutine. However, there may be cases where one does not want to disturb values in memory locations or perform the transfer operations necessary to setup the numbers for the SUBBER subroutine. What is needed is a two's complement routine that will operate on a value in the location(s) in which it resides. The following subroutine will accomplish that objective.

```
COMPLM,  LAM        Get least significant bits in first word
         XRI 377    Exclusive OR yields pure complement
         ADI 001    Now add '1' to form two's complement
MORCOM,  LMA        Return 2's complement value to memory
         RAR        Get the carry bit status
         LDA        And save the carry bit status
         DCB        Now decrement the precision counter
         RTZ        Finished subroutine when counter is zero
         INL        If not done, advance memory pointer
         LAM        And fetch the next group of bits
         XRI 377    Produce a pure complement
         LEA        Save pure complement temporarily
```

```
LAD         Get previous carry back into accumulator
RAL         And shift it back out to the carry flag
LAI 000     Do a load so does not disturb carry flag
ACE         Add complemented value with any carry
JMP MORCOM  Go on to do next word in string
```

Notice that in the above COMPLM subroutine it was necessary to save the status of the CARRY FLAG (carry bit) in a CPU register. This was because an XRI or any other BOOLEAN LOGIC instruction in an '8008' CPU automatically clears the carry flag to the zero state and would cause it to "lose" any previous logic one condition.

As in the ADDER and SUBBER subroutines it is also necessary to do some preliminary setting up before calling the COMPLM subroutine. The H and L registers must be set to the first word (least significant bits) of the multi-precision number. Register B must indicate how many words are used to hold the multi-word number.

It will also be pointed out here, that as the programmer gets into developing more and more complicated routines that utilize a lot of subroutines, the programmer must maintain strict control over which CPU registers are affected. The programmer must make sure that the use of selected CPU registers by one routine (especially when it CALLS another subroutine) do not interfere with the over-all operation of a program. The best rule of thumb is to try and leave a subroutine with all the CPU registers, except those transferring information to the next routine, in a FREE or "don't care" state. This is not always possible. When it is not, the programmer must keep track of which registers are being used for a specific purpose and not allow them to be unintentionally altered. For instance, the above COMPLM routine requires that three of the CPU registers be setup prior to entry. The H, L and B registers. When it leaves the subroutine those registers are essentially free for use by the next portion of the program. It also uses the A, D and E registers for operations that it performs. It does not care about the status of those registers when it starts operations because it loads them itself. It also leaves those registers essentially free when the routine is exited. (All the critical operations in the COMPLM subroutine are done with locations in memory.) However, the fact that the routine uses certain CPU registers, such as registers D and E, would be very important to remember if one was using other routines that maintained, say, memory pointers in registers D and E. The novice programmer (and a lot of times the not-so-novice ones) will often find some very strange operations occuring in a newly created program because of problems related to just this aspect!

The ADDER and SUBBER subroutines previously presented could be used by themselves to handle the addition and subtraction of large numbers. However, a restriction on the types of numbers they could handle would be that the numbers have to be whole numbers. Also, as the magnitudes of the numbers to be handled increased, the number of words used to store a value in multi-precision format would have to be increased. As was pointed out earlier, when one starts dealing with numbers of large magnitude, one is primarily concerned with a certain number of SIGNIFICANT DIGITS in a calculation. For instance, one could represent the value ONE MILLION as '1,000,000.' To store this number in multi-precision format requires the use of three memory words in an eight bit microprocessor. However, the number '1,000,000' only contains one significant digit. The number could just as easily be represented as '1' raised to the sixth power of ten. Or, 1 E+6 in what is often termed FLOATING POINT FORMAT. Note that if the number was stored in such format, one would only need to use one memory register in which to hold the single significant digit, plus a sepa-

rate register in which to hold the power to which the significant digit was to be raised. Floating point format also makes it easy to handle the task of processing fractional numbers. Up to this point, no discussion on representing non-integer numbers has been presented. This will be done shortly. As an introduction, note that the decimal number '0.1' could be represented in floating point format as '1' raised to the 'minus one' power of ten, or 1 E-1.

The reader has now been introduced to multi-precision arithmetic. Hopefully the reader now has an understanding of how large numbers can be stored in several small registers. The term large numbers may be interpreted as meaning numbers containing more than a couple of significant digits. The reader should understand that increasing the number of significant digits requires an increase in the number of binary bits required to store a number. It thus increases the number of memory words required when the number is stored in multiple-precision format. Also, when the format described up to now is used, increasing the magnitude of a number (by adding zeros to the right of the significant digits) rapidly increases the number of words of memory required to hold a number. Finally, just storing a number in a register, without regard to a decimal point location, makes it impossible to properly manipulate fractional numbers.

However, the idea that numbers can be represented as a series of significant digits raised to a power presents a solution to the limitations mentioned. Handling numbers in such a fashion is generally termed "floating-point" arithmetic. The remainder of this chapter will be devoted to developing routines for a FLOATING POINT mathematical program for general purpose applications.

Before proceeding into the development of floating-point routines, it will be necessary to discuss a matter that has been left aside up to this point. That is how to represent fractional numbers utilizing the language of the digital computer, binary arithmetic.

In the decimal numbering system which virtually everyone has been educated in, fractions of a number are represented by digits placed to the right of a decimal point. Each position to the right of such a point represents units of decreasing powers of 10. Thus the number:

0 . 1 2 5

actually represents:

	1 . .	Tenth (10 E-1)
Plus:	2 .	Hundredths (10 E-2)
Plus:	5	Thousandths (10 E-3)

The concept is exactly the same for binary arithmetic except that now each position to the right of the decimal point represents units of decreasing powers of two! Thus the number:

0 . 1 1 1

represents:

	1 . .	Half (2**-1)
Plus:	1 .	Quarter (2**-2)
Plus:	1	Eighth (2**-3)

Thus the above binary number '0.111' represents a fractional number which when converted to decimal is equal to:

1/2 + 1/4 + 1/8 = 7/8 or 0.875 (decimal)

The manner in which fractional binary numbers are represented brings out an interesting point which many readers may have heard of, but not truly understood. That is the introduction of errors into calculations done on a digital computer due to the manipulation of fractions that can not be finalized. As an analogy, there are similar cases in decimal arithmetic. One such case occurs when the number '1' is divided by '3.' The answer is:

0.333333333333333.....

which is a non-ending series of '3's after the

decimal point. The accuracy or precision with which a calculation involving such a number can be carried out is determined by how many significant digits are used in further calculations involving the fraction. For instance, theoretically, if the number one is divided by three and then multiplied by three, one would get back one as a result. However, if the result of the division is actually multiplied by three, the answer is not actually one, but approaches that value as the number of significant bits used in the calculation is increased. Observe.

```
  0.3   (one significant digit used)
X 3
-----
 .9   (answer is off by 10%)

  0.33  (two significant digits used)
X  3
--------
 .99   (answer is off by 1%)

  0.333  (three significant digits used)
X   3
----------
 .999   (answer is off by 0.1%)
```

A similar situation exists with binary arithmetic except there are now many more cases where the non-ending fraction situation can occur. For instance, the value '0.1' is truly represented in the decimal system. But, in the binary system, the decimal value '0.1' can only be approximated. As for the decimal case discussed above, the more binary digits used, the closer the value approaches the true value of '0.1.' Observe.

0.0001 (binary) = 1/16 =.0625 (decimal)
Which is off by 37.5%!

0.000110011 = 1/16 + 1/32 + 1/256
+ 1/512 = .0996
Off by just 0.4%!

Note too, that the binary representation is a non-ending series:

0.1 decimal = 0.0001100110011001100....
..... binary

and can not reach the theoretical true value of '0.1' as in the decimal system. Thus, if '0.1' as represented in the binary system is multiplied by, say, '10,' (which can be truly represented in the binary system) the theoretical value of '1.0' can only be approached. The more bits used to hold the binary equivalent, the closer one will approach the true answer. Thus, one may see another reason for using multiple-precision arithmetic in a digital computer, even if one does not want to handle large numbers. This is because, the more bits available to store a fractional number, the more precision one can maintain in performing calculations. One should now also realize, that the more complex a series of mathematical operations, in other words, the more times a number that can not be truly represented is multiplied or divided, the wider will become the margin of error in the final answer!

Now that one has a grasp of how binary digits can represent fractional numbers when placed to the right of a decimal point, one may proceed to investigate floating point arithmetic using a digital computer.

FLOATING POINT ARITHMETIC

Just as one can represent decimal numbers in floating point format, that is, by a string of significant digits raised to a power of ten, one may treat binary numbers in a similar manner as a string of binary digits raised to a power of two.

When handling numbers in floating point format the number is represented in two parts. The significant digits portion is referred to as the MANTISSA. The power to which the significant digits are raised is referred to as the EXPONENT. In decimal floating point format the number '5' could be expressed as:

5.0 E+0 = 5 X 1 = 5

OR 50.0 E-1 = 50 X 1/10 = 5

OR 0.5 E+1 = 0.5 X 10 = 5

While in binary floating point format the number could be expressed as:

101.0 E+0 = 5 X 1 = 5

OR 101000.0 E-3 = 40 X 1/8 = 5

OR 0.101 E+3 = 5/8 X 8 = 5

It should be remembered that in the decimal example above the EXPONENT represents a power of TEN. In the binary example it represents a power of TWO.

Note that the mechanics of the correspondence between the exponent and the location of the decimal point in the mantissa is the same for both numbering systems. If the significant digits in the mantissa are moved to the right of the decimal point then the exponent is increased one unit for each position the mantissa is shifted. If the digits in the mantissa are shifted to the left, then the exponent is decreased. The only difference between the two systems is that the exponent in the decimal system is specified for powers of ten, while in the binary system it is for powers of two.

The reader may now see that it can be quite a simple matter to handle binary numbers using floating point format if one register (or several registers) is used to hold the mantissa portion, and another register is used to maintain the exponent. Furthermore, a very simple relationship can be maintained between the mantissa and the exponent to facilitate keeping track of a decimal point. Once one has selected a given position as a reference in the mantissa portion, one has only to observe the following procedures for manipulating the number and keeping track of the decimal point:

Each time the MANTISSA is shifted RIGHT

INCREMENT the EXPONENT!

Each time the MANTISSA is shifted LEFT

DECREMENT the EXPONENT!

For the remainder of this chapter, a convention for storing numbers in floating point format will be established and maintained. Numbers will be stored in four consecutive words in memory. The first word in a group will be used to store the EXPONENT with the most significant bit in the word used to represent the SIGN of the EXPONENT. A '1' in the most significant bit position means the number is NEGATIVE. The next three words will hold the MANTISSA portion in triple-precision format. The first bit in the first (most significant word) of the mantissa will be used as the mantissa sign bit. The remaining bits in that word will be the most significant bits of the mantissa. The remaining two words in the mantissa group will hold the less significant bits of the mantissa. Furthermore, there will be an IMPLIED DECIMAL POINT immediately to the right of the sign bit in the mantissa. The format is illustrated here:

...EXPONENT...	..... MSW ..... MANTISSA		LSW .....
S E E E E E E E	S.M M M M M M M	M M M M M M M M	M M M M M M M M
MEM LOC N+3	MEM LOC N+2	MEM LOC N+1	MEM LOC N

Note the order of the memory addresses assigned to the storage of a number. The order of storage is an arbitrary assignment. However, once it has been assigned it must be adhered to within a program. The order shown is the one that will be used in the discussion and program examples for the remainder of this section.

Note too, that a convention has been established that will consider a decimal point (actually, perhaps it should be termed a binary point) to be located to the right of the designated sign bit for the mantissa. This means that all numbers stored in floating point format will be represented as a fractional number! Also, the reader may observe that with one bit out of the three words used to hold the sign of the mantissa, that 23 (decimal) bits are left to hold the actual magnitude of the mantissa. Similarly, the exponent has seven bits in which to represent the magnitude of its value. The eighth bit being used to represent the sign of the exponent. Furthermore, an exponent must be an integer value as there is no implied decimal point in the exponent register.

FLOATING POINT NORMALIZATION

NORMALIZATION may be considered as a standardizing process that will place a number into a fixed position as a reference point from which to commence operations. For the purposes of this discussion, the term NORMALIZATION will mean to place a number into its storage registers so that the mantissa will have a value that is greater than or equal to ONE HALF (1/2) but less than ONE (1). Put another way, this means that any number to be manipulated by a floating point routine will first be shifted so that the most significant binary digit is next to the IMPLIED BINARY POINT in the most significant word of the MANTISSA storage registers. For instance, if a binary number such as:

101.0 E+0 (5 decimal)

was received by an input routine to a floating point program, the number would be NORMALIZED when it was placed in the form:

0.101 E+3 (5/8 X 5 = 5 decimal)

Similarly, if after, say, a binary division operation in which the number '1' had been divided by 10 (decimal) and one had the answer:

0.000110011001100... E+0 (0.1 decimal)

the number would be considered normalized when it was placed in the format:

0.110011001100110... E-3 (0.1 decimal)

Note that normalizing a number is a pretty easy matter. In the first example the number was normalized by shifting the original number to the right until the most significant bit was just to the right of the decimal point. During this procedure, the value of the exponent was incremented for each shifting operation in the mantissa. In the second example, the number is normalized by shifting the original value of the mantissa to the left while decrementing the exponent for each shifting operation in the mantissa.

There are several reasons for wanting to NORMALIZE a number when working with a floating point program. The first has to do with the fact that generally numbers will originate from a human who will be using the computer to manipulate numbers in decimal format. Therefore, the computer will have to convert numbers from say, decimal floating point format, to the binary floating point format used by the computer. There will be more discussion on this matter later in this chapter after a number of binary floating point operations have been presented. The second reason for normalizing numbers, and a very important one, is because the process will allow more significant digits to be retained in a fixed length register. This may be seen by observing in the above example (the case where '0.1' decimal is normalized)

that shifting the binary number to the left three places would allow several more LSB's to be placed in a fixed length register for the non-ending binary series 0.110011001100..... and thus allow more accuracy in the binary calculations that might follow!

A routine for normalizing binary numbers will be presented next. In the routine for normalizing numbers, and various other mathematical routines in this chapter, various locations on PAGE 00 will be used for storing numbers that are to be manipulated by the routines as well as for holding COUNTERS and POINTERS used in the routines. A list of the locations reserved for such use on PAGE 00 will be provided later. Also, before getting into the actual binary floating point routines, the reader should be informed that in the following routines, references will be made to a FLOATING POINT ACCUMULATOR and a FLOATING POINT OPERAND. The floating point accumulator and operand will be separate groups of registers consisting of four consecutive memory words on PAGE 00 used to store the active numbers that are manipulated by the floating point routines. They will, of course, be arranged in the format described earlier. That is, a single-word EXPONENT and then a triple-word MANTISSA. The FLOATING POINT ACCUMULATOR will be the focal point for any floating point routine as all the results of floating point calculations will be placed there. The FLOATING POINT OPERAND will be used primarily for holding and manipulating the number that the floating point accumulator operates on. For abbreviation in further discussions, the floating point accumulator will be shortened to FPACC and the operand to FPOP.

```
FPNORM,  LAB        Check register B for special case
         NDA        Set flags after load operation
         JTZ NOEXCO If B was '0' then do standard normalization
         LLI 127    Otherwise set EXPONENT of FPACC
         LMB        To value found in B at start of routine
NOEXCO,  LLI 126    Set pointer to MSW of FPACC MANTISSA
         LAM        And get MSW of FPACC MANTISSA into ACC
         LLI 100    Change pointer to SIGN storage address
         NDA        Set flags after previous LAM operation
         JTS ACCMIN If MSB in MSW equals '1' then have minus number
         XRA        If MSB = '0' then have positive value mantissa
         LMA        So set SIGN storage to 000 value
         JMP ACZERT Proceed to see if FPACC = zero
ACCMIN,  LMA        Original FPACC = negative number, set SIGN
         LBI 004    Set precision counter to four (using extra word)
         LLI 123    And pointer to FPACC LSW-1 (using extra word)
         CAL COMPLM Two's complement FPACC (using extra word)
ACZERT,  LLI 126    Check to see if FPACC contains zero
         LBI 004    Set a counter
 LOOK0,  LAM        Get a part of FPACC
         NDA        Set flags after load operation
         JFZ ACNONZ If find anything then FPACC is not zero
         DCL        Otherwise move pointer to next part
         DCB        Decrement the loop counter
         JFZ LOOK0  And if not finished check next part
         LLI 127    If reach here FPACC was zero
         XRA        So make sure EXPONENT of FPACC is zero
         LMA        By placing zero in it
         RET        Can then exit NORMALIZATION routine
```

```
ACNONZ,  LLI 123      If FPACC has value, set up pointer and
         LBI 004      Precision value ('4' to handle special cases)
         CAL ROTATL   Then rotate FPACC to the LEFT
         LAM          Now get MSB of MSW from MANTISSA
         NDA          Set flags after load operation
         JTS ACCSET   If MSB = '1' then have found MSB in FPACC
         INL          If not, advance pointer to FPACC EXPONENT
         CAL CNTDWN   And decrement the value of the EXPONENT
         JMP ACNONZ   Then continue in the rotating left loop
ACCSET,  LLI 126      Compensate for last rotate left when MSB
         LBI 003      Found to leave room for SIGN in MSB of the
         CAL ROTATR   FPACC MANTISSA by doing one rotate RIGHT
         LLI 100      Set pointer to original SIGN storage
         LAM          Get original SIGN indicator value
         NDA          Set flags after load operation
         RFS          Finished if value in FPACC is POSITIVE
         LLI 124      Original SIGN is negative, so set pointer to
         LBI 003      LSW of FPACC and also set precision counter
         CAL COMPLM   Now two's complement the NORMALIZED FPACC
         RET          That is all for the FP NORMALIZATION ROUTINE
```

There are several items in the above routine that might confuse the reader if not explained. First of all, the routine checks CPU register B when it is entered. If B contains '0' then the routine will proceed directly on to a new section in the program. If B contains some value, then the value it contains will be placed in the EXPONENT portion of the FPACC. This is done so that the FPNORM subroutine can process numbers that are not initially in floating point form. For instance, when a number is first received from an INPUT device it will generally be in a form such as shown in the example below depicting the binary equivalent of 5 decimal:

00 000 000 00 000 000 00 000 101

As it would appear in standard triple-precision format. Now, the above standard format could be converted to floating point format by assuming that a BINARY POINT existed to the right of the least significant bit, and shifting the entire number to the right while incrementing the binary exponent register. However, the technique would cause a slight problem. How could one tell where the most significant bit of the binary number resided? A way around that problem is to simply shift the registers to the LEFT until the first '1' (MSB) is in the desired position. If this is done, one must first set the EXPONENT portion of the floating point number to the highest possible value that could be contained in the registers. Then, that value is decremented each time the magnitude portion of the number is shifted to the LEFT. In the example presentation, there are 23 decimal bits available for storing the mantissa when triple-precision formatting is being used (24 bits less one which is used to represent the sign of the number). Thus, one would simply load register B with the octal equivalent of 23 decimal which is 27 before calling the FPNORM subroutine whenever one wanted to convert a number in standard form to floating point format! The following illustrations should help clarify the presentation:

ORIGINAL BINARY NUMBER WHEN IT IS IN STANDARD FORMAT

00 000 000 00 000 000 00 000 101

DESIRED FLOATING POINT FORMAT

SE EEE EEE
(exponent followed by mantissa):
S.I III III II III III II III III

NOW ORIGINAL NUMBER PLACED IN FPACC and EXPONENT SET TO 27 (OCTAL)

00 010 111
(exponent followed by mantissa):
0.0 000 000 00 000 000 00 000 101

ORIGINAL NUMBER IS THEN NORMALIZED BY ROTATING LEFT

00 000 011
(exponent followed by mantissa):
0.1 010 000 00 000 000 00 000 000

Since the exponent was decremented each time the number was rotated left the final exponent value is the same as if the number had been rotated to the right to accomplish the normalization while incrementing the exponent from a value of zero!

The reader should also note that the FPNORM subroutine checks to see if the number to be normalized is negative. If it is, the routine keeps track of that fact and makes the number positive in order to accomplish the normalization procedure. If it did not, the normalization routine would not work as may be seen when one recalls what a number such as minus five appears like in its two's complement form:

11 111 111 11 111 111 11 111 011

After the number has been normalized in its positive form, it is converted back to the negative form so that the number minus five would appear when normalized as:

00 000 011
(exponent followed by mantissa):
1.0 110 000 00 000 000 00 000 000

The reader should work through the procedure using pencil and paper to make sure the process is understood when processing negative numbers as it may be confusing at first glance. Note that the normalized minus value has the most significant bit position in the mantissa set to a '1' to indicate a negative value!

Another point of interest in the FPNORM subroutine is that the routine tests to see if the FPACC contains zero. Note that if this test was not made and appropriate action taken to exit the subroutine on such a condition, that the program could become trapped in the rotate left loop as it would fail to ever see a '1' appear in the most significant bit position! When a zero condition is found in the mantissa, the routine sets the exponent part of the FPACC to zero as an additional safety measure.

Finally, the reader may note that the first part of the normalization routine assumes the mantissa uses four memory words. This was done so that the subroutine could handle some special cases that can occur after operations such as multiplication where it may be necessary to have some additional precision. In cases where the feature is not needed, the extra memory word should be set to zero before using the FPNORM subroutine.

The ROTATL and ROTATR subroutines called by FPNORM are short routines that have been set up for N'th-precision operation as with other algorithms discussed in this chapter. Before entering the routines the calling program sets the starting address of the string of memory words to be processed in the H and L CPU registers. It should also set the number of words in the string in register B. The two subroutines are shown next.

```
ROTATL,  NDA        Clear carry flag at this entry point
ROTL,    LAM        Fetch word from memory
         RAL        Rotate LEFT (with carry)
         LMA        Restore rotated word to memory
         DCB        Decrement precision counter
         RTZ        Return to calling routine when done
         INL        Otherwise advance pointer to next word
         JMP ROTL   And rotate across the memory word string

ROTATR,  NDA        Clear carry flag at this entry point
ROTR,    LAM        Fetch word from memory
         RAR        Rotate RIGHT (with carry)
         LMA        Restore rotated word to memory
         DCB        Decrement precision counter
         RTZ        Return to calling routine when done
         DCL        Going other way so decrement memory pointer
         JMP ROTR   And rotate across the memory word string
```

FLOATING POINT ADDITION

Floating point addition is quite straight forward. In fact, one may use the ADDER subroutine already developed earlier in this chapter for the mantissa portion of a set of floating point numbers. However, there are a few other parameters that must be considered in developing the overall routine.

When two numbers are to be added it will be assumed that they have been positioned in the FPACC and the FPOP memory storage areas. A few items that should be considered in developing the basic floating point addition routine include the following.

Suppose either the FPOP or FPACC contain zero? Or, they both contain zero? In the latter case the routine could be immediately exited as the answer is sitting in the FPACC. If the FPACC is zero, but the FPOP is not, then one has merely to place the contents of the FPOP into the FPACC (as the convention was established earlier that the result of an operation would always be left in the FPACC). For the case where the FPACC contains a value, but the FPOP is zero, one may immediately exit the routine.

But, as will more likely be the case when the floating point ADD routine is called, both the FPACC and the FPOP will contain some non-zero value. Thus one could immediately proceed to perform the addition operation, right? WRONG! Since floating point operations allow the manipulation of numbers with large magnitudes, because of the exponent method of maintaining magnitudes, it is quite possible that an operator might ask for an addition of a very small number to a very large number. (This also might occur in the middle of a complex calculation where an operator was not monitoring the intermediate results.) Readers know that if the difference between the two numbers to be added is so great that there can be no change in the significant digits during the calculation then there is no need to perform the addition process. So, the next step in the floating point addition routine would be to check to see whether or not the magnitudes of the numbers are within significant range of one another. If they are not, then the largest value should be placed in the FPACC as the answer!

If the magnitudes of the two numbers are within significant range then the two numbers may be added. Before this can be done, they must first be ALIGNED by shifting one of the numbers until the exponent is equal in value to that of the second number. The alignment

is accomplished by finding out which exponent is the smallest and shifting the mantissa of that number to the right (while incrementing the exponent for each shift) until it is properly aligned. The shifting procedure is quite straightforward since it can be handled by a N'th-precision register rotate subroutine. However, there is one special consideration for the case of a negative number being shifted to the right. One must insert a '1' into the most significant bit position each time such a shift is made in order to maintain the minus value properly (to keep the sign bit in its proper state). This can be accomplished easily as the reader may observe in the following FPADD subroutine by inserting a '1' into the carry bit, then calling the ROTR subroutine. (This is simply another entry point to the ROTATR subroutine presented earlier. The entry point at ROTR avoids the NDA instruction which would cause the carry bit to be cleared to a '0' condition if executed.)

One more consideration that the reader may note in the following FPACC subroutine is that the two numbers to be added are shifted to the right once before the addition is performed so that any overflow from the addition will stay within the FPACC. This will allow normalization to be handled by the previously presented routine instead of having to be concerned with the status of the carry flag at the end of the operation. Because of this shifting operation, an additional memory word is used by both the FPACC and FPOP and the addition is performed using quad-precision. At the end of the addition process the result is normalized and left in the FPACC.

```
FPADD,  LLI 126      Set pointer to MSW of FPACC
        LBI 003      Set loop counter
CKZACC, LAM          Fetch part of FPACC
        NDA          Set flags after loading operation
        JFZ NONZAC   Finding anything means FPACC not zero
        DCB          If that part equals zero, decrement loop counter
        JTZ MOVOP    If FPACC equals zero, move FPOP into FPACC
        DCL          Not finished checking, decrement pointer
        JMP CKZACC   And test next part of FPACC
MOVOP,  CAL SWITCH   Save pointer to LSW of FPACC
        LHD          Set H equal to zero for sure
        LLI 134      Set pointer to LSW of FPOP
        LBI 004      Set a loop counter
        CAL MOVEIT   Move FPOP into FPACC as answer
        RET          Exit FPADD subroutine
NONZAC, LLI 136      Set pointer to MSW of FPOP
        LBI 003      Set loop counter
CKZOP,  LAM          Get MSW of FPOP
        NDA          Set flags after load operation
        JFZ CKEQEX   If not zero then have a number
        DCB          If zero, decrement loop counter
        RTZ          Exit subroutine if FPOP equals zero
        DCL          Else decrement pointer to next part of FPOP
        JMP CKZOP    And continue testing for zero FPOP
CKEQEX, LLI 127      Check for equal exponents
        LAM          Get FPACC exponent
        LLI 137      Change pointer to FPOP exponent
        CPM          Compare exponents
        JTZ SHACOP   If same can setup for ADD operation
```

```
         XRI 377      If not same, then two's complement
         ADI 001      The value of the FPACC exponent
         ADM          And add in FPOP exponent
         JFS SKPNEG   If + then go directly to alignment test
         XRI 377      If negative perform two's complement
         ADI 001      On the result
SKPNEG,  CPI 030      Now see if result greater than 27 octal
         JTS LINEUP   If not can perform alignment
         LAM          If not alignable, get FPOP exponent
         LLI 127      Set pointer to FPACC exponent
         SUM          Subtract FPACC exponent from FPOP exponent
         RTS          FPACC exponent greater so just exit routine
         LLI 124      FPOP was greater, set pointer to FPACC LSW
         JMP MOVOP    Go put FPOP into FPACC and then exit routine
LINEUP,  LAM          Align FPACC and FPOP, get FPOP exponent
         LLI 127      Change pointer to FPACC exponent
         SUM          Subtract FPACC exponent from FPOP exponent
         JTS SHIFTO   FPACC greater so go to shift operand
         LCA          FPOP greater so save difference
MORACC,  LLI 127      Pointer to FPACC exponent
         CAL SHLOOP   Call shift loop subroutine
         DCC          Decrement difference counter
         JFZ MORACC   Continue aligning if not done
         JMP SHACOP   Setup for ADD operation
SHIFTO,  LCA          Shift FPOP routine, save difference count (negative)
MOROP,   LLI 137      Set pointer to FPOP exponent
         CAL SHLOOP   Call shift loop subroutine
         INC          Increment difference counter
         JFZ MOROP    Shift again if not done
SHACOP,  LLI 123      First clear out extra room, setup pointer
         LMI 000      to FPACC LSW+1 and set it to zero
         LLI 127      Now prepare to shift FPACC right once
         CAL SHLOOP   Set pointer and then call shift loop routine
         LLI 137      Shift FPOP right once, first set pointer
         CAL SHLOOP   Call shift loop subroutine
         LDH          Setup pointers, set D equal to zero for sure
         LEI 123      Pointer to LSW of FPACC
         LBI 004      Set precision counter
         CAL ADDER    Add FPACC to FPOP using quad-precision
         LBI 000      Set B for standard normalization procedure
         CAL FPNORM   Normalize the result of the addition
         RET          Exit FPADD subroutine with result in FPACC
SHLOOP,  LBM          Shifting loop for alignment
         INB          Fetch exponent into B and increment
         LMB          Return increment value to memory
         DCL          Decrement the pointer
         LBI 004      Set a counter
FSHIFT,  LAM          Get MSW of floating point number
         NDA          Set flags after loading operation
         JTS BRING1   If number is minus, need to shift in a '1'
         CAL ROTATR   Otherwise perform N'th-precision rotate
```

```
        RET         Exit FSHIFT subroutine
BRING1, RAL         Save '1' in carry bit
        CAL ROTR    Do ROTATE RIGHT without clearing carry bit
        RET         Exit FSHIFT subroutine
MOVEIT, LAM         Fetch a word from memory string 'A'
        INL         Advance 'A' string pointer
        CAL SWITCH  Switch pointer to string 'B'
        LMA         Put word from string 'A' into string 'B'
        INL         Advance B string pointer
        CAL SWITCH  Switch pointer back to string 'A'
        DCB         Decrement counter
        RTZ         Return to calling routine when counter is zero
        JMP MOVEIT  Otherwise continue moving operation
```

FLOATING POINT SUBTRACTION

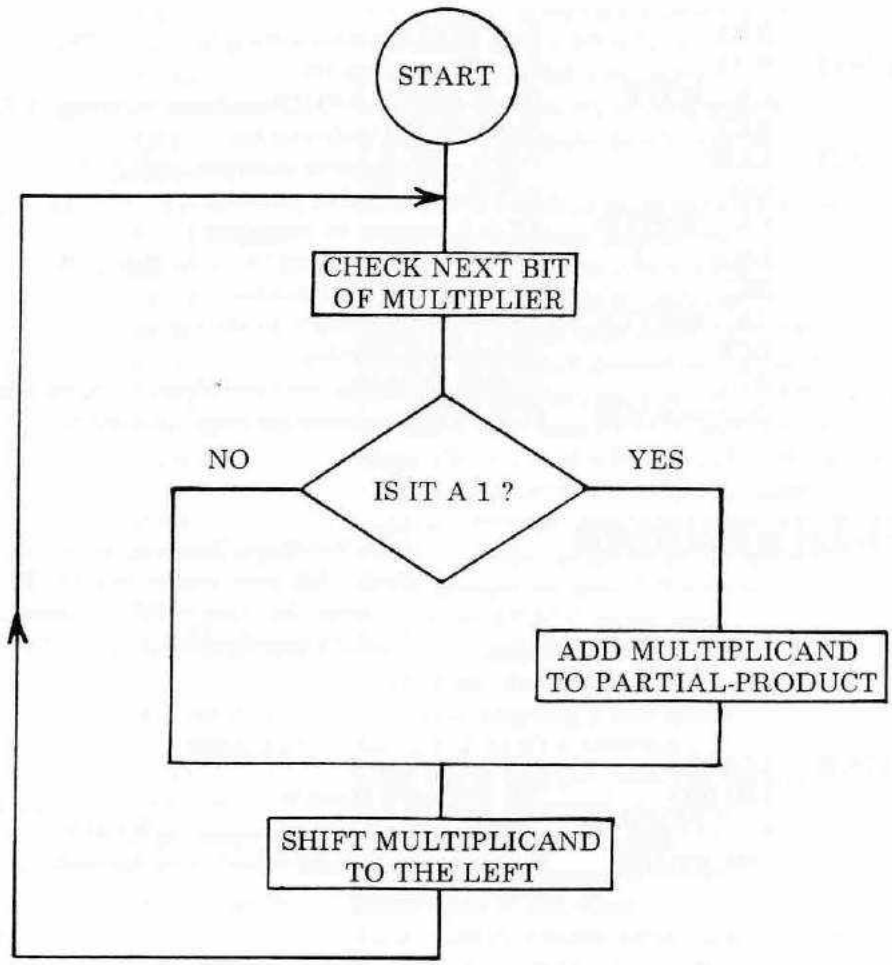
Now that one has a floating point addition routine, floating point subtraction is a snap. All one really has to do is negate the number in the FPACC and jump to the floating point addition routine!

```
FSUB,  LLI 124     Set pointer to LSW of FPACC
       LBI 003     Set precision counter
       CAL COMPLM  Perform two's complement on FPACC
       JMP FPADD   Subtraction accomplished now by adding!
```

FLOATING POINT MULTIPLICATION

Floating point multiplication can be accomplished by utilizing a shifting and adding algorithm for the mantissa portion of the numbers. As pointed out earlier, shifting a binary number to the LEFT serves to essentially DOUBLE its value. An algorithm that takes advantage of that fact can be described as follows.

Consider the two numbers as a MULTIPLIER and a MULTIPLICAND. Examine the least significant bit of the MULTIPLIER. If it is a one, add the current value of the MULTIPLICAND to a third register (which initially starts with a value of zero). Now, shift the MULTIPLICAND one position to the LEFT. Examine the next bit to the LEFT of the least significant bit in the MULTIPLIER. If it is a one, add the current value of the MULTIPLICAND to the third register (which could be called the PARTIAL-PRODUCT register). Shift the MULTIPLICAND to the LEFT again. Continue the process by examining all the bits in the MULTIPLIER for a one condition. Whenever the MULTIPLIER contains a ONE add the current value of the MULTIPLICAND to the PARTIAL-PRODUCT register. After each examination of a bit in the multiplier (and addition of the multiplier to the partial-product register if a '1' was observed) shift the multiplicand LEFT. Continue until all bits in the multiplier have been examined. The result of the multiplication will be in the partial-product register at the completion of the above process. The algorithm can perhaps be seen a little more clearly by studying the flow chart presented next.



The reader may verify the algorithm by following the example below for two small numbers, the number '3' as the multiplicand and the number '5' as the multiplier.

00 000 011	(Multiplicand at start of operations.)
00 000 101	(Multiplier.)
---	
00 000 000	(Partial-product before operations start.)
00 000 011	(Multiplicand when first bit of multiplier is examined.)
00 000 101	(Least significant bit of multiplier = '1.')
---	
00 000 011	(Multiplicand is added to partial-product.)

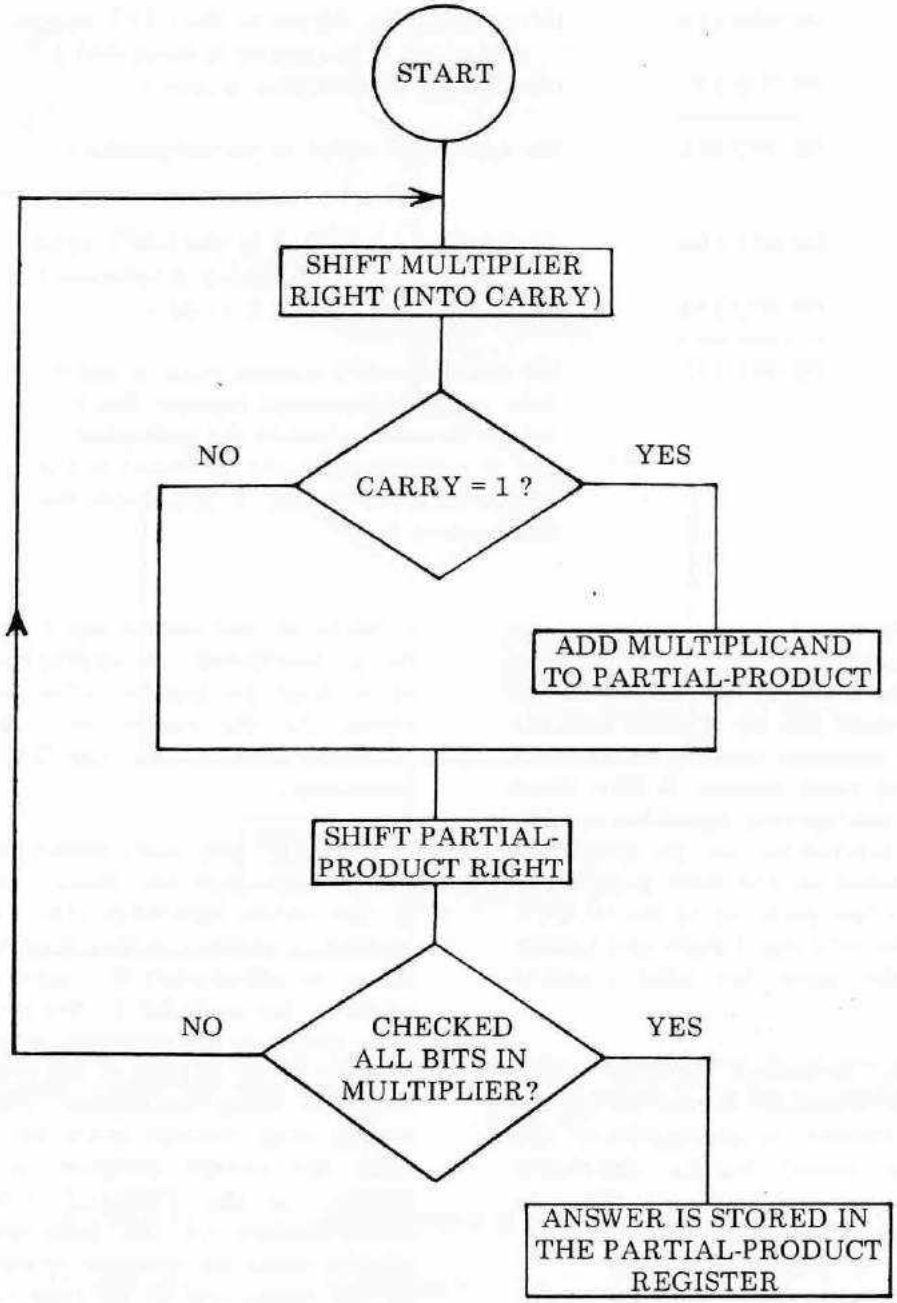
```
00 000 110      (Multiplicand is shifted to the LEFT before
                  second bit of multiplier is examined.)
00 000 101      (Second bit of multiplier is zero.)
-----------
00 000 011      (So nothing is added to partial-product.)


00 001 100      (Multiplicand is shifted to the LEFT again
                  before next bit of multiplier is examined.)
00 000 101      (Third bit of multiplier is a one.)
-----------
00 001 111      (So multiplicand's current value is added
                  into the partial-product register. Since
                  all the remaining bits in the multiplier
                  are '0' nothing more will be added to the
                  partial-product register. It thus holds the
                  final answer!)
```

While the algorithm just presented was designed for multiplying numbers that are in standard format, with just a little variation, the basic procedure can by applied towards multiplying the mantissa portion of numbers stored in floating point format. A flow chart of the mantissa multiplying algorithm used in the FPMULT subroutine to be presented shortly is illustrated on the next page. Note that it is easy to test each bit of the MULTIPLIER by simply rotating it right and testing the status of the carry flag after a rotate operation.

Handling the exponent portion when multiplying two numbers stored in binary floating point format is accomplished the same way one would handle exponents in decimal floating point format. The exponents are simply added together.

There are several other parameters to consider when multiplying numbers. First, the algorithm presented may only be used when the numbers are positive in value. Thus, any negative numbers must first be negated before using the algorithm. Furthermore, the reader knows that if two numbers of the same sign are multiplied together the answer will be a positive value, but, if the signs are different, the answer will be a negative number. Therefore, one must take account of the initial signs of the numbers being multiplied. If appropriate, the final value must be negated after using the algorithm. As the reader may observe in the FPMULT subroutine, handling this task is quite easy.

Secondly, the alert reader may have observed that since the multiplicand is shifted in the above algorithm (the partial-product register is shifted in the floating point algorithm to accomplish the same purpose) one position for each bit in the multiplier, then it is necessary to maintain working registers that are twice as long as the original numbers that are being multiplied. Thus, the final answer may contain more bits of precision than the overall program is designed to handle. In the FPMULT subroutine, the multiplication of the mantissas is accomplished using six memory words per register. At the conclusion of the routine, the twenty-third binary bit is rounded off (depending on the status of the twenty-fourth least significant bit) and the answer is normalized back to a 23 bit binary number which is the largest number of bits the package being discussed is designed to normally manipulate. The method allows maximum precision to be maintained during the multiplication process without over-burdening the rest of the floating point routines.



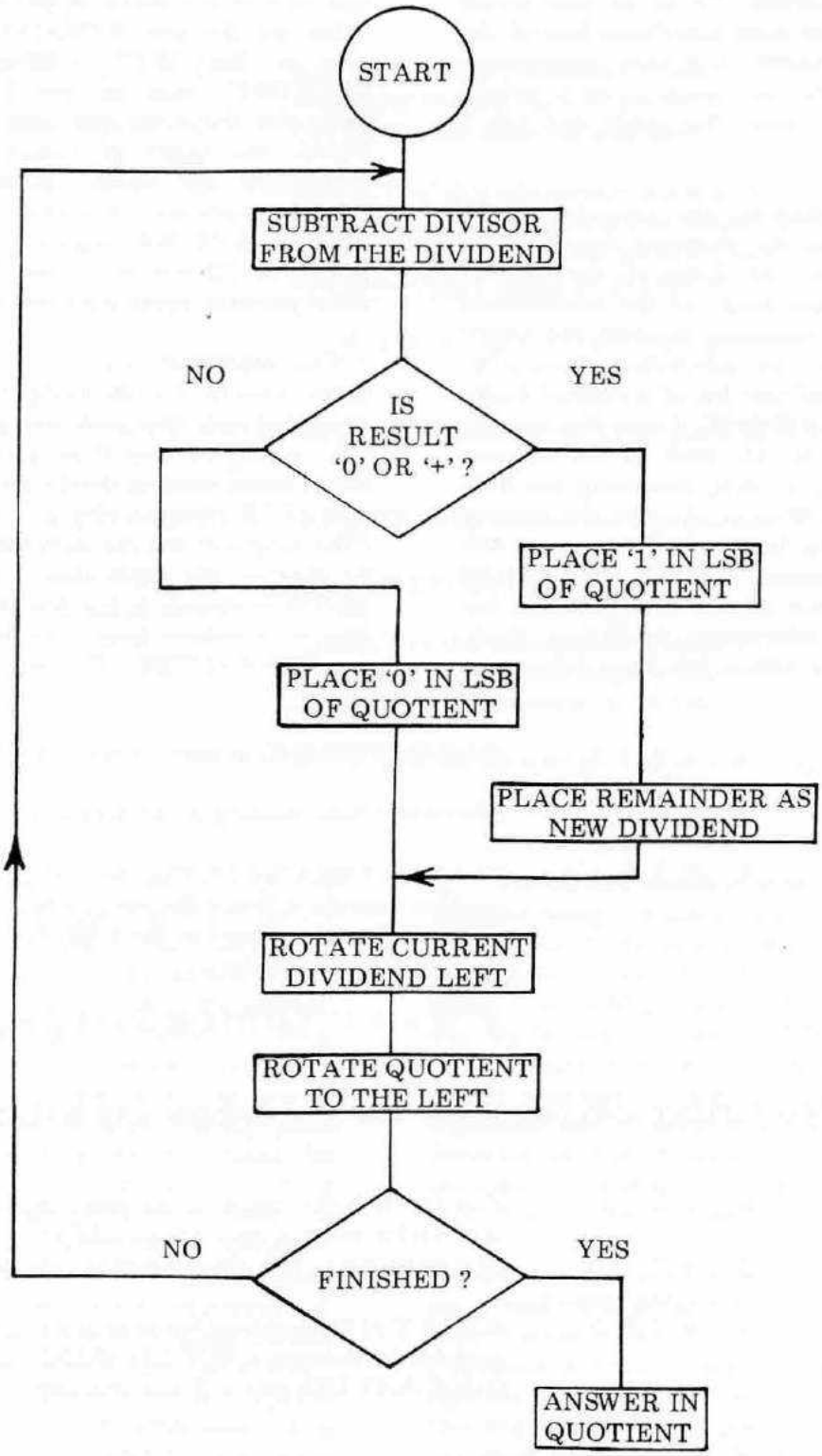

```
FPMULT,  CAL CKSIGN   Setup routine and check sign of numbers
ADDEXP,  LLI 137      Set pointer to FPOP exponent
         LAM          Fetch FPOP exponent into accumulator
         LLI 127      Set pointer to FPACC exponent
         ADM          Add FPACC exponent to FPOP exponent
         ADI 001      Add one for algorithm compensation
         LMA          Store result in FPACC exponent
```

```
SETMCT,  LLI 102      Set bit counter storage pointer
         LMI 027      Set bit counter to 23 decimal (27 octal)
MULTIP,  LLI 126      Basic multiply algorithm, set pntr to MSW of FPACC
         LBI 003      Set precision counter
         CAL ROTATR   Rotate multiplier RIGHT into carry flag
         CTC ADOPPP   If carry equals one, add multiplicand to partial-product
         LLI 146      Set pointer to partial-product MSW
         LBI 006      Set precision counter
         CAL ROTATR   Shift partial-product RIGHT
         LLI 102      Set pointer to bit counter
         CAL CNTDWN   Decrement value in bit counter
         JFZ MULTIP   If bit counter not zero, repeat algorithm
         LLI 146      Set pointer to partial-product MSW
         LBI 006      Set precision counter, now rotate partial-product
         CAL ROTATR   Once more to make room for possible rounding
         LLI 143      Set pointer to access 24'th bit in partial-product
         LAM          Fetch 24'th bit
         RAL          Position it to MSB position
         LAA          NOP inserted to correct algorithm
         NDA          Set flags after rotate operation
         CTS MROUND   If 24'th bit is a '1' then do rounding process
         LLI 123      Now set pointer to FPACC
         CAL SWITCH   Save FPACC pointer
         LHD          Ensure that H is '000'
         LLI 143      Set pointer to partial-product
         LBI 004      Set precision counter
EXMLDV,  CAL MOVEIT   Move answer from partial-product into FPACC
         LBI 000      Set B for standard normalization
         CAL FPNORM   Normalize the answer
         LLI 101      Set pointer to SIGN indicator
         LAM          Fetch SIGN indicator
         NDA          Set flags after load operation
         RFZ          If SIGN has value, result is positive, exit subroutine
         LLI 124      But if SIGN is zero, set FPACC LSW pointer
         LBI 003      And set precision counter
         CAL COMPLM   And negate the answer
         RET          Before exiting the FPMULT subroutine
CKSIGN,  CAL CLRWRK   Clear working locations for multiplication
         LLI 101      Set pointer to SIGN storage
         LMI 001      Place the initial value of '1' into SIGN storage
         LLI 126      Set pointer to MSW of FPACC
         LAM          Fetch MSW of FPACC
         NDA          Set flags after load operation
         JTS NEGFPA   If number is minus, need to do two's complement
OPSGNT,  LLI 136      Set pointer to MSW of FPOP
         LAM          Fetch MSW of FPOP
         NDA          Set flags after load operation
         RFS          If number is positive, return to calling routine
         LLI 101      If number is minus, set pointer to SIGN storage
         CAL CNTDWN   Decrement value of SIGN indicator
         LLI 134      Set pointer to LSW of FPOP
```

```
        LBI 003     Set precision counter
        CAL COMPLM  Perform two's complement of number in FPOP
        RET         Go back to calling routine
NEGFPA, LLI 101     Set pointer to SIGN storage
        CAL CNTDWN  Decrement value of SIGN indicator
        LLI 124     Set pointer to LSW of FPACC
        LBI 003     Set precision counter
        CAL COMPLM  Negate the value in the FPACC
        JMP OPSGNT  Go check sign of FPOP
CLRWRK, LLI 140     Clear partial-products work area (140 - 147)
        LBI 010     Set pointer and counter
        XRA         Set accumulator to zero
CLRNEX, LMA         Deposit accumulator contents into memory
        DCB         Decrement counter
        JTZ CLROPL  When done go to next area
        INL         Else continue clearing partial-product working area
        JMP CLRNEX  By stuffing zeroes in next memory location
CLROPL, LBI 004     Clear additional room for multiplicand
        LLI 130     At 130 to 133, first set counter and pointer
CLRNX1, LMA         Put '000' in memory
        DCB         Decrement counter
        RTZ         Return to calling program when done
        INL         Else advance pointer
        JMP CLRNX1  And continue clearing operation
ADOPPP, LEI 141     Pointer to LSW of partial-product
        LDH         On PAGE 00 in D & E pointer
        LLI 131     Pointer to LSW of multiplicand
        LBI 006     Set precision counter
        CAL ADDER   Perform addition
        RET         Exit subroutine
MROUND, LBI 003     Set precision counter
        LAI 100     Add '1' to 23'rd bit of partial-product
        ADM         Here
CROUND, LMA         Restore to memory
        INL         Advance pointer
        LAI 000     Clear ACC without disturbing CARRY FLAG
        ACM         And propogate rounding
        DCB         In partial-product
        JFZ CROUND  Finished when counter equals zero
        LMA         Restore last word of partial-product
        RET         Exit subroutine
```

FLOATING POINT DIVISION

In a manner that is sort of the reverse of multiplication (which uses ADDITION and ROTATE operations) one can perform division using an algorithm that utilizes SUBTRACTION and ROTATE operations. An algorithm will be presented directly in the form used in floating point operations because in this case it is simpler than describing it for numbers that are not in floating point form. The alert reader should have little difficulty observing that the algorithm


START
SUBTRACT DIVISOR FROM THE DIVIDEND
NO
IS RESULT '0' OR '+' ?
YES
PLACE '1' IN LSB OF QUOTIENT
PLACE '0' IN LSB OF QUOTIENT
PLACE REMAINDER AS NEW DIVIDEND
ROTATE CURRENT DIVIDEND LEFT
ROTATE QUOTIENT TO THE LEFT
NO
FINISHED ?
YES
ANSWER IN QUOTIENT

could be used for numbers that are not in floating point format. To do so, one would have to align the most significant bits of the divisor and dividend, and take appropriate action to handle the location of a binary point in cases where the result was not a pure integer.

In rambling English, the algorithm could be stated as follows. Subtract the value of the divisor from the value of the original dividend. Test the result of the subtraction. If the result is negative, meaning the entire divisor could not be subtracted, place a '0' in the least significant bit of a register designated as the QUOTIENT. Leave the current dividend alone. If the result of the subtraction is positive, or zero, indicating the dividend was larger than or equal to the divisor, place a '1' in the least significant bit of the QUOTIENT register, then set the dividend equal to the value of the REMAINDER (or result) of the subtraction operation. Next, once appropriate action has been taken as a result of the subtraction operation, rotate the contents of the dividend (whether its original value or the new REMAINDER) one position to the LEFT. Similarly, rotate the QUOTIENT once to the LEFT to allow room for the next least significant bit. Now repeat the entire procedure until one has performed the above operations as many times as there are bit positions in the register used to hold the original dividend! (That would be 23 decimal times for the floating point package being discussed.)

The algorithm may be visualized a little more clearly by studying the flow chart presented on the previous page. Additionally, a step-by-step illustration of the algorithm being used to divide the binary equivalent of 15 (decimal) by 5 is presented next. (The length of the registers have been reduced to shorten the illustration.) Remember, the algorithm shown is for the MANTISSA portion of numbers once they have been stored in NORMALIZED floating point format!

```
0 . 1 1 1 1     Original DIVIDEND at start of routine.

0 . 1 0 1 0     DIVISOR (Note floating point format.)
---------------
0 . 0 1 0 1     This is the REMAINDER from the sub-
                traction operation. Since the result was
                POSITIVE a '1' is placed in the LSB of
                the QUOTIENT register.

                0 . 0 0 0 1   QUOTIENT after 1'st loop.
```

NOW BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

```
0 . 1 0 1 0     New DIVIDEND (which is the previous RE-
                MAINDER rotated once to the LEFT).
0 . 1 0 1 0     DIVISOR (Does not change during routine).
---------------
0 . 0 0 0 0     RESULT of this subtraction is zero and thus
                qualifies to become a NEW DIVIDEND. Also,
                QUOTIENT LSB gets a '1' for this case!

                0 . 0 0 1 1   QUOTIENT after 2'nd loop.
```

AGAIN BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

```
0 . 0 0 0 0     New DIVIDEND (which is the last
                remainder rotated once to the left).
0 . 1 0 1 0     DIVISOR (still same old number).
---------------
1 . 0 1 1 0     RESULT of this subtraction is a minus
                number (note that the SIGN bit chang-
                ed). Thus, old DIVIDEND stays in place
                and QUOTIENT gets a '0' in LSB!

                0 . 0 1 1 0  QUOTIENT after 3'rd loop.
```

NOW BOTH QUOTIENT, AND IN THIS CASE, THE OLD DIVIDEND, ARE ROTATED LEFT

```
0 . 0 0 0 0     Old DIVIDEND rotated once to the left.

0 . 1 0 1 0     Same old DIVISOR.
---------------
1 . 0 1 1 0     RESULT of this subtraction is again a
                minus. Old DIVIDEND stays in place.
                QUOTIENT gets another '0' in LSB.

                0 . 1 1 0 0  QUOTIENT after 4'th loop.
```

Since there were just four bits in the multiplicand register, the algorithm would be completed at the end of the fourth loop in the illustration above. The answer would be that shown in the quotient. Remember, that since floating point format is being used, there would be binary exponents involved. Similar to the way one would handle exponents in decimal floating point notation, one subtracts the exponents for the two numbers (DIVISOR exponent from the DIVIDEND exponent) to obtain the exponent value for a division operation. In the above example, the multiplicand would have had the binary exponent '4' (decimal) to represent the normalized storing of 15 and the divisor would have had a binary exponent of '3.' The above algorithm requires a compensation factor of +1 after subtracting the exponents (can the reader think of ways in which this could be avoided?) in order to have the correct floating point result. In the example being discussed here, (4 - 3) + 1 = 2, and indeed if the answer shown was moved two places to the left (of the implied binary point) one could quickly verify that the result was the binary equivalent of 3 decimal! The reader might want to try using other small valued numbers to test the validity of the algorithm and to develop a thorough understanding of the process. A good case to examine is one where the result is non-ending such as when the number '1' is divided by '3.'

Just as in the multiplication routine, there are several other parameters that must be considered when developing the division routine. For instance, there is again the matter of the signs of the numbers. The algorithm requires that the numbers be in positive format. Again one must keep track of the signs of the original numbers and convert any negative ones to

positive values for the routine. If the signs of the two numbers involved are identical, the result must be a positive value. If they are different then the program must negate the answer obtained from the actual division process. And, because some calculations could result in a non-ending series for an answer, some rounding capability must be included in the routine. Then, there is a special case in division that one must check for and take appropriate action upon finding. That is the case of an attempted divide by zero! In such a situation, the program should branch off to notify the operator of an error condition. The floating point routine shown next considers these matters as the reader may observe.

```
FPDIV,  CAL CKSIGN    Setup registers and check sign of numbers
        LLI 126       Set pointer to MSW of FPACC (DIVISOR)
        LAI 000       Clear accumulator
        CPM           See if MSW of FPACC is zero
        JFZ SUBEXP    If find anything proceed to divide
        DCL           Decrement pointer
        CPM           See if NSW of DIVISOR is zero
        JFZ SUBEXP    If find anything proceed to divide
        DCL           Decrement pointer
        CPM           See if LSW of DIVISOR is zero
        JTZ DERROR    If DIVISOR equals zero, have error condition!
SUBEXP, LLI 137       Set pointer to DIVIDEND (FPOP) exponent
        LAM           Fetch DIVIDEND exponent
        LLI 127       Set pointer to DIVISOR (FPACC) exponent
        SUM           Subtract DIVISOR exp from DIVIDEND exp
        ADI 001       Compensate for division algorithm
        LMA           Store exponent result in FPACC exponent
SETDCT, LLI 102       Set pointer to bit counter storage
        LMI 027       Set it to 27 octal (23 decimal)
DIVIDE, CAL SETSUB    Main division subroutine, subtract DIVIS from DIVID
        JTS NOGO      If result is negative then put '0' in QUOTIENT
        LEI 134       If '+' or '0' then move REMAINDER into DIVIDEND
        LLI 131       Set pointers
        LBI 003       And precision counter
        CAL MOVEIT    And move REMAINDER into DIVIDEND
        LAI 001       Put a '1' into accumulator
        RAR           And move it into the CARRY BIT
        JMP QUOROT    Proceed to ROTATE it into the QUOTIENT
NOGO,   LAI 000       When RESULT is NEGATIVE, put '0' into ACC
        RAR           And move it into CARRY BIT
QUOROT, LLI 144       Set pointer to LSW of QUOTIENT
        LBI 003       Set precision counter
        CAL ROTL      Move CARRY BIT into LSB of QUOTIENT
        LLI 134       Set pointer to DIVIDEND LSW
        LBI 003       Set precision counter
        CAL ROTATL    Rotate DIVIDEND left
        LLI 102       Set pointer to bits counter
        CAL CNTDWN    Decrement bits counter
        JFZ DIVIDE    If not finished then continue algorithm
        CAL SETSUB    Do one more divide for rounding operations
```

```
        JFS DVEXIT      If 24'th bit equal zero then no rounding
        LLI 144         When 24'th bit is '1' set pntr to QUOTIENT LSW
        LAM             Fetch LSW of QUOTIENT
        ADI 001         Add '1' to 23'rd bit
        LMA             Restore LSW
        LAI 000         Clear accumulator while saving CARRY FLAG
        INL             Advance pointer to NSW of QUOTIENT
        ACM             Add with carry
        LMA             Restore NSW
        LAI 000         Clear accumulator while saving CARRY FLAG
        INL             Advance pointer to MSW of QUOTIENT
        ACM             Add with carry
        LMA             Restore MSW
        JFS DVEXIT      If MSB of MSW is zero prepare to exit
        LBI 003         Otherwise set precision counter
        CAL ROTATR      Move QUOTIENT to the RIGHT to clear SIGN BIT
        LLI 127         Set pointer to FPACC exponent
        LBM             Fetch exponent
        INL             Increment it for ROTATE RIGHT operation above
        LMB             Restore exponent
DVEXIT, LLI 144         Set pointers to transfer
        LEI 124         QUOTIENT to FPACC
        LBI 003         Set precision counter
        JMP EXMLDV      Exit through FPMULT routine at EXMLDV
SETSUB, LLI 131         Set pointer to LSW of working register
        CAL SWITCH      Save pointer
        LHD             Set H = '0' for sure
        LLI 124         Set pointer to LSW of FPACC
        LBI 003         Set precision counter
        CAL MOVEIT      Move FPACC value to working register
        LEI 131         Reset pointer to working register LSW (DIVISOR)
        LLI 134         Set pointer to LSW of FPOP (DIVIDEND)
        LBI 003         Set precision counter
        CAL SUBBER      Subtract DIVISOR from DIVIDEND
        LAM             Get MSW of RESULT from subtraction operations
        NDA             And set flags after load operation
        RET             Before returning to calling routine

DERROR, CAL DERMSG      **User defined ERROR routine for handling
        JMP USERDF      Attempted divide by zero, exit as directed**
```

The five fundamental floating point subroutines, FPNORM, FPADD, FPSUB, FPMULT and FPDIV when assembled into object code will fit within three pages of memory in an '8008' system. Additionally, the routines as presented in this chapter use some space on PAGE 00 for storing data and counters. Needless to say, the programs as developed for discussion could be modified to use other memory locations with little difficulty. For reference purposes, the locations used on PAGE 00 by the fundamental floating point routines just presented are listed on the next page.

100	SIGN indicator
101	SIGNS indicator (multiply & divide)
102	Bits counter
123	FPACC extension
124	FPACC least significant word (LSW)
125	FPACC next significant word (NSW)
126	FPACC most significant word (MSW)
127	FPACC exponent
130 - 133	Working area
134	FPOP least significant word
135	FPOP next significant word
136	FPOP most significant word
137	FPOP exponent
140 - 147	Working area

The fundamental floating point routines which have been presented and discussed are extremely powerful routines which should be of considerable value to anyone desiring to manipulate mathematical data in an '8008' or similar system. The routines in the form presented for illustrative purposes are capable of handling binary numbers that are the decimal equivalent of six to seven digits raised to approximately the plus or minus 38'th power of ten! The routines may be used to solve a wide variety of mathematical formulas by simply calling the appropriate subroutines after loading the FPOP and FPACC registers with the values that are to be manipulated (when they are in normalized floating point format). Furthermore, the basic routines illustrated can become the fundamental routines in more sophisticated programs. Such programs might be developed to calculate functions such as SINES and COSINES using numerical techniques such as expansion series formulas.

The interested programmer should have little difficulty in modifying the routines illustrated to upgrade their capability to provide more significant digits (by increasing the length of the mantissa). Or, to extend the exponents capability by providing double or even triple-precision registers for the exponent. For many applications, however, the user will be well satisfied with the capability provided by the routines as they have been presented for educational purposes.

The floating point routines which have been presented are used to manipulate numbers once they are in binary format. In some applications, such as when formulas are being solved by a computer to control the operation of a machine, or applications where there is little or no need to communicate with humans, the above routines coupled with I/O routines and whatever operating programs are dictated by the application, would be sufficient for handling the mathematical operations. However, in probably the majority of applications, at some time or other it will be desirable for humans to communicate with the computer. Or, for the computer to at least present information to humans. It seems that the vast majority of people prefer to manipulate mathematical data using decimal notation. Most people would not want to change their ways by working in floating point binary notation! So, most programmers would find it beneficial to have some conversion routines that would convert numbers from decimal floating point notation to binary floating point notation as well as the reverse. The next section of this chapter is

devoted to discussing and developing routines that accomplish such a worthwile objective.

CONVERTING FLOATING POINT DECIMAL TO FLOATING POINT BINARY

Most people using a digital computer for handling mathematical functions would like to input data in the form:

1234.567

OR

1.234 E+3

Using an input device such as a keyboard or electronic typing machine. In order to accept data in such format one needs to develop a program that will first convert the information from the decimal mantissa and exponent form to the binary equivalent. The process is fairly straightforward conceptually.

First, one needs to develop a method for breaking down the mantissa portion into a DECIMAL NORMALIZED format. This may be done quite readily because:

1234.567 = 1234567.0 E-3

AND

1.234 E+3 = 1234.0 E+0

Thus, to effectively normalize a decimal number one has to simply keep track of where the decimal point is placed by the operator in the mantissa. Then one needs to compensate for that factor by removing the decimal point (making the mantissa an integer value) and changing the exponent value to compensate for the removal of the decimal point!

Next, one needs to convert the mantissa portion of the number from decimal to binary. That conversion process can actually be accomplished as each decimal number is inputted by the operator using the algorithm described below.

DECIMAL TO BINARY CONVERSION

Each time a digit is received in decimal form, immediately convert it to its binary equivalent. In many cases this consists of simply MASKING OFF extra bits to leave a value in BCD format. Next, in order to compensate for the powers of ten denoted by the positional weight of decimal numbers, multiply any previous number(s) that are already stored in binary form by multiplying them by ten (decimal). Then add in the binary equivalent of the number that has just been received.

The algorithm can be illustrated by considering the following example. An operator enters the decimal number 63 by first entering the number '6' and then '3' from an input device such as an ASCII encoded keyboard:

00 000 000 Input register initially cleared.

Operator initially types in the character for a '6.' This is immeidately converted to 1 1 0 as its binary equivalent. Since it is the first character received it is not necessary to multiply the present value of the storage register by ten. The binary value 1 1 0 can simply be placed in the INPUT register giving:

00 000 110 Input register after 1'st number.

The operator then enters the character for a '3.' Once again this is immediately converted to 0 1 1 as its binary equivalent. But, before this new digit is added to the binary storage register, the contents of the register must be multiplied by ten to account for the positional value of the previous digit. A simple way to multiply a binary register by ten is to perform the following steps:

00 000 110	Input register initially contains '6'
00 001 100	Rotate left = multiply by 2
00 011 000	Rotate left = multiply by 4
00 011 110	Add in original value = times 5
00 111 100	Rotate left = multiply by 10

With the previous value of '6' now multiplied by ten to represent 60 (decimal) in the binary register, the new value of '3' can now be added in to yield:

00 111 111 Binary equivalent of 63 (decimal)

The above algorithm is repeated each time an additional decimal character is received to maintain the binary equivalent. The algorithm is valid for multiple-precision storage of numbers.

Finally, it is necessary to convert the decimal exponent value (which again is immediately converted to a binary number as it is received from the input device) to represent a binary number raised to an equivalent value. Conversion at this point may be accomplished by first converting the binary representation of the mantissa to its normalized format (using the special capability of the FPNORM routine). Then multiplying the normalized floating point binary number by 10 (decimal) for each unit of a positive decimal exponent. This can be accomplished by using the FPMULT routine previously described!

The decimal to binary input program to be presented next handles the above considerations plus several other functions. The routine will allow an operator to specify the sign of the decimal mantissa and exponent and takes appropriate action to negate numbers designated as being minus in value. It also allows for erasure of the current input string by typing a special character. The routine assumes that characters are received from an input device that uses ASCII code and that an output device using ASCII code is used to ECHO (repeat back) information as it is received from the input. Neither the actual input or output subroutines are shown in the sample program that follows. (Information on typical I/O routines will be presented in another chapter.) The program also assumes that certain locations on PAGE 00 will be used for storage of numbers received and for maintaining counters and indicators. A listing

of the locations used will be provided later. The program calls on other routines previously detailed in this chapter such as FPNORM and FPMULT.

```
DINPUT,  LHI 000        Set pointer to INPUT
         LLI 150        Storage registers
         XRA            Clear accumulator
         LBI 010        Set a counter
CLRNX2,  LMA            And clear memory locations 150 - 157
         INL            By depositing zeroes and advancing pointer
         DCB            And decrementing loop counter
         JFZ CLRNX2     Until finished
         LLI 103        Set pointers to counter/indicator storage
         LBI 004        Set a counter
CLRNX3,  LMA            And clear memory locations 103 - 106
         INL            In a similar fashion by depositing zeroes
         DCB            And decrementing loop counter
         JFZ CLRNX3     Until finished
         CAL INPUT      Now bring in a character from I/O device
         CPI 253        Test to see if it is a '+' sign
         JTZ SECHO      If yes, go to ECHO and continue
         CPI 255        If not '+' see if '-' sign
         JFZ NOTPLM     If not '+' or '-' test for valid character
         LLI 103        If minus, set pointer to INPUT SIGN
         LMA            And make it non-zero by depositing character
SECHO,   CAL ECHO       Output character in ACC as ECHO to operator
NINPUT,  CAL INPUT      Fetch a new character from I/O device
NOTPLM,  CPI 377        See if character is code for RUBOUT
         JTZ ERASE      If yes, prepare to start over
         CPI 256        If not, see if character is a period (.)
         JTZ PERIOD     If '.' process as decimal point
         CPI 305        If not, see if character is 'E' for exponent
         JTZ FNDEXP     If 'E' process as exponent indicator
         CPI 260        If not, see if character is a valid number
         JTS ENDINP     If none of above, terminate input string
         CPI 272        Still checking for valid number
         JFS ENDINP     If not, terminate input string
         LLI 156        Have a number, set pntr to MSW of INPUT register
         LBA            Save character in register B
         LAI 370        Form a mask and check to see if input
         NDM            Registers can accept larger number
         JFZ NINPUT     If not, ignore present input
         LAB            If O.K., restore character to accumulator
         CAL ECHO       And ECHO number back to operator
         LLI 105        Set pointer to digit counter
         LCM            Fetch digit counter
         INC            Increment its value
         LMC            And restore it to storage
         CAL DECBIN     Perform decimal to binary conversion
         JMP NINPUT     Get next character for mantissa
PERIOD,  LBA            Subroutine to process '.' - save in B
```

```
        LLI 106     Set pointer to '.' storage indicator
        LAM         Fetch contents
        NDA         Set flags after load operation
        JFZ ENDINP  If '.' already present, end input string
        LLI 105     Otherwise set pointer to digit counter
        LMA         And reset digit counter to zero
        INL         Advance pointer back to '.' storage
        LMB         And put a '.' there
        LAB         Restore '.' to accumulator
        CAL ECHO    And echo it back to operator
        JMP NINPUT  Get next character in number string
ERASE,  LAI 274     Put ASCII code for < in accumulator
        CAL ECHO    Display it
        LAI 240     Put ASCII code for SPACE in ACC
        CAL ECHO    And leave a couple of spaces
        CAL ECHO    Before going back to
        JMP DINPUT  Start the input string over
FNDEXP, CAL ECHO    Subroutine to process exponent, echo 'E'
        CAL INPUT   Get next part of exponent
        CPI 253     Test for a '+' sign
        JTZ EXECHO  If yes, proceed to echo it
        CPI 255     If not, test for a '-' sign
        JFZ NOEXPS  If not, see if a valid character
        LLI 104     If have '-' then set pointer to EXPONENT SIGN
        LMA         Set EXPONENT SIGN minus indicator
EXECHO, CAL ECHO    Echo character back to operator
EXPINP, CAL INPUT   Get next character for exponent portion
NOEXPS, CPI 377     See if code for RUBOUT
        JTZ ERASE   If yes, prepare to re-enter entire string
        CPI 260     Otherwise check for valid decimal number
        JTS ENDINP  If not, end input string
        CPI 272     Still testing for valid number
        JFS ENDINP  If not, end input string
        NDI 017     Have valid number, form mask and strip ASCII
        LBA         Character to pure BCD, save in register B
        LLI 157     Set pointer to input exponent storage location
        LAI 003     Set accumulator = '3'
        CPM         See if 1'st exponent number was greater than three
        JTS EXPINP  If yes, ignore input (limits exponent to less than 40)
        LCM         If O.K., save previous exponent value in register C
        LAM         And also place it in accumulator
        NDA         Clear the carry bit
        RAL         Multiply times ten algorithm: 1'st multiply by two
        RAL         Multiply by two again
        ADC         Add in original value
        RAL         Multiply by two once more
        ADB         Add in new number to complete the decimal to
        LMA         Binary conversion for exponent and restore to memory
        LAI 260     Restore ASCII code by adding 260
        ADB         To BCD value of the number
        JMP EXECHO  And echo number, then look for next input
```

```
ENDINP,  LLI 103      Set pointer to mantissa SIGN indicator
         LAM          Fetch SIGN indicator
         NDA          Set flags after load operation
         JTZ FININP   If nothing in indicator, number is positive
         LLI 154      Set pointer to LSW of input mantissa
         LBI 003      Set precision
         CAL COMPLM   Perform two's complement to negate number
FININP,  LLI 153      Set pointer to input storage LSW-1
         XRA          Clear accumulator
         LDA          Clear register D
         LMA          Clear input storage location LSW-1
         LEI 123      Set pointer to FPACC LSW-1
         LBI 004      Set precision counter
         CAL MOVEIT   Move input + LSW-1 to FPACC + LSW-1
         LBI 027      Set special FPNORM mode by setting bit count
         CAL FPNORM   In register B and then call normalization routine
         LLI 104      Set pointer to EXPONENT SIGN indicator
         LAM          Fetch EXPONENT SIGN indicator to ACC
         NDA          Set flags after load operation
         LLI 157      Set pointer to decimal exponent storage
         JTZ POSEXP   If exponent positive, jump ahead
         LAM          If exponent negative, fetch it into accumulator
         XRI 377      And perform two's
         ADI 001      complement
         LMA          Then restore to storage location
POSEXP,  LLI 106      Set pointer to period indicator
         LAM          Fetch contents to accumulator
         NDA          Set flags after load operation
         JTZ EXPOK    If nothing, no decimal point involved
         LLI 105      If have decimal point, set pointer to digit
         XRA          Counter then clear accumulator
         SUM          Subtract digit counter from '0' to give negative
EXPOK,   LLI 157      Set pointer to decimal exponent storage
         ADM          Add in compensation for decimal point
         LMA          Restore compensated value to storage
         JTS MINEXP   If compensated value minus, jump ahead
         RTZ          If compensated value zero, finished!
EXPFIX,  CAL FPX10    Compensated decimal exponent is positive, multiply
         JFZ EXPFIX   FPACC by 10, loop until decimal exponent is zero
         RET          Exit with converted value in FPACC
FPX10,   LEI 134      Multiply FPACC by 10 subroutine, set pointer to
         LDH          FPOP LSW, then set D = zero for sure
         LLI 124      Set pointer to FPACC LSW
         LBI 004      Set precision counter
         CAL MOVEIT   Move FPACC to FPOP (including exponents)
         LLI 127      Set pointer to FPACC exponent
         LMI 004      Place FP form of 10 (decimal) in FPACC
         DCL          Place FP form of 10 (decimal) in FPACC
         LMI 120      Place FP form of 10 (decimal) in FPACC
         DCL          Place FP form of 10 (decimal) in FPACC
         XRA          Place FP form of 10 (decimal) in FPACC
```

```
        LMA         Place FP form of 10 (decimal) in FPACC
        DCL         Place FP form of 10 (decimal) in FPACC
        LMA         Place FP form of 10 (decimal) in FPACC
        CAL FPMULT  Now multiply original binary number (in FPOP) by ten
        LLI 157     Set pointer to decimal exponent storage
        CAL CNTDWN  Decrement decimal exponent value
        RET         Return to calling program
MINEXP, CAL FPD10   Compensated decimal exponent is minus, multiply
        JFZ MINEXP  FPACC by 0.1, loop until decimal exponent is zero
        RET         Exit with converted value in FPACC
 FPD10, LEI 134     Multiply FPACC by 0.1 routine, pointer to FPOP LSW
        LDH         Set D = '0' for sure
        LLI 124     Set pointer to FPACC
        LBI 004     Set precision counter
        CAL MOVEIT  Move FPACC to FPOP (including exponent)
        LLI 127     Set pointer to FPACC exponent
        LMI 375     Place FP form of 0.1 (decimal) in FPACC
        DCL         Place FP form of 0.1 (decimal) in FPACC
        LMI 146     Place FP form of 0.1 (decimal) in FPACC
        DCL         Place FP form of 0.1 (decimal) in FPACC
        LMI 146     Place FP form of 0.1 (decimal) in FPACC
        DCL         Place FP form of 0.1 (decimal) in FPACC
        LMI 147     Place FP form of 0.1 (decimal) in FPACC
        CAL FPMULT  Now multiply original binary number (in FPOP) by 0.1
        LLI 157     Set pointer to decimal exponent storage
        LBM         Fetch value
        INB         Increment it
        LMB         Restore it to memory
        RET         Return to calling program
DECBIN, LLI 153     Decimal to binary conversion, set pntr to temp storage
        LAB         Restore character to accumulator
        NDI 017     Mask off ASCII bits to leave pure BCD number
        LMA         Place current BCD number in temporary storage
        LEI 150     Set pointer to working area LSW
        LLI 154     Set another pointer to LSB of input registers
        LDH         Set D = '0' for sure
        LBI 003     Set precision counter
        CAL MOVEIT  Move original value to working area
        LLI 154     Set pointer to LSW of INPUT storage
        LBI 003     Set precision counter
        CAL ROTATL  Rotate LEFT (X 2)              (Total = X 2)
        LLI 154     Set pointer to LSW again
        LBI 003     Set precision counter
        CAL ROTATL  Rotate LEFT (X 2)              (Total = X 4)
        LEI 154     Set pointer to LSW of rotated value
        LLI 150     And another to LSW of original value
        LBI 003     Set precision counter
        CAL ADDER   Add original to rotated     (Total now = X 5)
        LLI 154     Set pointer to LSW again
        LBI 003     Set precision counter
        CAL ROTATL  Rotate LEFT (X 2)          (Total now = X 10)
```

```
        LLI 152     Set pointer to clear working area
        XRA         Clear accumulator
        LMA         Deposit in MSW of working area
        DCL         Decrement pointer to MSW
        LMA         Put zero there too
        LLI 153     Set pointer to current digit storage
        LAM         Fetch latest BCD number
        LLI 150     Set pointer to LSW of working area
        LMA         Deposit latest BCD number in LSW
        LEI 154     Seup pointer
        LBI 003     Set precision counter
        CAL ADDER   Add in latest number to complete DECBIN conversion
        RET         Return to calling program
```

CONVERTING FLOATING POINT BINARY TO FLOATING POINT DECIMAL

The following program will convert binary numbers stored in floating point format to decimal floating point format and display them on an output device such as an electronic printer (using ASCII code) in the following format:

+0.1234567 E+07

The routine operates essentially in the reverse manner to the input routine just described. First the binary floating point number is converted to a regularly formatted binary number. Then the number is converted to a decimal number using a multiply by ten algorithm. Since the reader should now be quite adept at following the operation of a program from the commented source listing, the floating point binary to floating point decimal conversion routine will be presented without further discussion. Remember that the routine illustrated assumes an ASCII encoded output device is being utilized. In addition, several subroutines used by the previously illustrated DINPUT program are called by the routine.

```
FPOUT,  LLI 157     Set pointer to decimal exponent storage
        LMI 000     Clear decimal exponent storage location
        LLI 126     Set pointer to MSW FPACC MANTISSA
        LAM         Fetch MSW FPACC MANTISSA to accumulator
        NDA         Set flags after load operation
        JTS OUTNEG  If MSB = 1 have a negative number
        LAI 253     Otherwise number is positive, set ASCII code for '+'
        JMP AHEAD1  Go to display '+' sign
OUTNEG, LLI 124     Have a negative number, set pntr to LSW FPACC
        LBI 003     Set precision counter
        CAL COMPLM  Perform two's complement on FPACC
        LAI 255     Set ASCII code for '-' sign
AHEAD1, CAL ECHO    Display sign of MANTISSA
        LAI 260     Set ASCII code for '0'
        CAL ECHO    Display '0'
        LAI 256     Set ASCII code for '.'
        CAL ECHO    Display '.'
        LLI 127     Set pointer to FPACC exponent
        LAI 377     Put '-1' in accumulator
```

```
            ADM         Effectively subtract one from exponent
            LMA         Restore compensated exponent
DECEXT,     JFS DECEXD  If compen exp is zero or positive, multip MANT by 0.1
            LAI 004     If compensated exponent is negative
            ADM         Add '4' (decimal) to exponent value
            JFS DECOUT  If exponent now zero or positive, output MANTISSA
            CAL FPX10   Otherwise, multiply MANTISSA by 10
DECREP,     LLI 127     Set pointer to FPACC exponent
            LAM         Get exponent after multiplication routine
            NDA         Set flags after load operation
            JMP DECEXT  Repeat above test for zero or positive condition
DECEXD,     CAL FPD10   Multiply FPACC by 0.1
            JMP DECREP  Check status of FPACC exponent after multiplication
DECOUT,     LEI 164     Set pointer to LSW of OUTPUT registers
            LDH         Make D = zero for sure
            LLI 124     Set pointers to LSW of FPACC
            LBI 003     Set precision counter
            CAL MOVEIT  Move FPACC to OUTPUT registers
            LLI 167     Set pointer to MSW+1 of OUTPUT register
            LMI 000     And clear that location
            LLI 164     Now set pointer to LSW of OUTPUT register
            LBI 003     Set precision counter, perform one
            CAL ROTATL  Rotate operation to compensate for space of sign bit
            CAL OUTX10  Multiply OUTPUT register by 10, overflow into MSW+1
COMPEN,     LLI 127     Set pointer to FPACC exponent
            LBM         Compensate for any remainder in binary
            INB         Exponent by performing a ROTATE RIGHT on
            LMB         OUTPUT registers until binary exponent becomes zero
            JTZ OUTDIG  Go to output digits when compensation done
            LLI 167     Binary exponent compensation rotate loop
            LBI 004     Set pointer to OUTPUT MSW+1 and set counter
            CAL ROTATR  Perform compensating ROTATE RIGHT operation
            JMP COMPEN  Repeat loop until binary exponent equals zero
OUTDIG,     LLI 107     Set pointer to output digit counter
            LMI 007     Set digit counter to '7' to initialize
            LLI 167     Set pointer to MSD in OUTPUT register MSW+1
            LAM         Fetch BCD form of digit to be displayed
            NDA         Set flags after load operation
            JTZ ZERODG  See if 1'st digit is a '0'
OUTDGS,     LLI 167     If not, set pointer to MSW+1 (BCD code)
            LAI 260     Form ASCII number code by adding 260 (octal)
            ADM         To the BCD code
            CAL ECHO    And display the ASCII encoded decimal number
DECRDG,     LLI 107     Set pointer to output digit counter
            CAL CNTDWN  Decrement value of output digit counter
            JTZ EXPOUT  When it is = '0' go do exponent output routine
            CAL OUTX10  Otherwise multiply OUTPUT register by 10
            JMP OUTDGS  And output next decimal digit
ZERODG,     LLI 157     If 1'st digit, then set pointer to MSW
            CAL CNTDWN  Decrement value to compensate for skipping display
            LLI 166     Of first digit, then set pointer to MSW
```

```
         LAM       Of output registers, fetch contents
         NDA       Set flags after load operations
         JFZ DECRDG  Check to see if entire mantissa is '0'
         DCL       Check to see if entire mantissa is '0'
         LAM       Check to see if entire mantissa is '0'
         NDA       Check to see if entire mantissa is '0'
         JFZ DECRDG  Check to see if entire mantissa is '0'
         DCL       Check to see if entire mantissa is '0'
         LAM       Check to see if entire mantissa is '0'
         NDA       Check to see if entire mantissa is '0'
         JFZ DECRDG  Check to see if entire mantissa is '0'
         LLI 157   If entire mantissa is zero, set pointer to
         LMA       Decimal exponent storage and set it to '0'
         JMP DECRDG  Before proceeding to finish display
OUTX10,  LLI 167   Multiply output registers by 10 to push out
         LMI 000   BCD code of MSD, first clear output MSW+1
         LLI 164   Set pointer to LSW of output registers
         LDH       Make sure D equals zero
         LEI 160   Set another pointer to working area
         LBI 004   Set precision counter
         CAL MOVEIT  Move original value to working area
         LLI 164   Set pointer to original value LSW
         LBI 004   Set precision counter
         CAL ROTATL  Start multiply by 10 routine      (Total = X 2)
         LLI 164   Reset pointer
         LBI 004   And counter
         CAL ROTATL  Multiply by two again             (Total = X 4)
         LLI 160   Set pointer to LSW of original value
         LEI 164   And another to LSW of rotated value
         LBI 004   Set precision counter
         CAL ADDER  Add original value to rotated      (Total = X 5)
         LLI 164   Reset pointer
         LBI 004   And counter
         CAL ROTATL  Multiply by two once more         (Total = X 10)
         RET       Finished multiplying output registers by ten
EXPOUT,  LAI 305   Set ASCII code for letter E
         CAL ECHO  Display E for Exponent
         LLI 157   Set pointer to decimal exponent storage location
         LAM       Fetch decimal exponent to accumulator
         NDA       Set flags after load operation
         JTS EXOUTN  If MSB equals one, value is negative
         LAI 253   If value is positive, set ASCII code for '+' sign
         JMP AHEAD2  Go to display the sign
EXOUTN,  XRI 377   For negative exponent, perform two's comp
         ADI 001   In standard manner
         LMA       And restore to storage location
         LAI 255   Set ASCII code for '-' sign
AHEAD2,  CAL ECHO  Display sign of the exponent
         LBI 000   Clear register B for use as a counter
         LAM       Fetch decimal exponent value
SUB12,   SUI 012   Subtract 10 (decimal)
```

```
         JTS TOMUCH   Look for negative result
         LMA          Restore positive result, maintain count of how
         INB          Many times 10 (decimal) can be subtracted
         JMP SUB12    to obtain most significant digit of exponent
TOMUCH,  LAI 260      Form ASCII character for MSD of exponent by
         ADB          Adding 260 to count in register B
         CAL ECHO     And display most significant digit of exponent
         LAM          Fetch remainder from decimal exponent storage
         ADI 260      And form ASCII character for LSD of exponent
         CAL ECHO     Display least significant digit of exponent
         RET          Exit FPOUT routine
```

Once one has a decimal to binary INPUT routine, and binary to decimal OUTPUT routine to work with the fundamental floating point routines, it is a relatively simple matter to tie them all together. By doing so, one may form an OPERATING PACKAGE that will allow an operator to specify numerical values in decimal floating point notation, indicate whether addition, subtraction, multiplication, or division was desired, and then obtain an answer from the computer. An illustrative operating program that utilizes all the demonstration routines presented in this section is shown below. The program will allow an operator to make entries and receive results in the format illustrated here:

+33.0E+3 X -4 = -0.1320000E+6

```
FPCONT,  CAL CRLF2    Display a few Cr's & LF's for I/O device
         CAL DINPUT   Let operator enter a FP decimal number
         CAL SPACES   Display a few spaces after number
         LLI 124      Set pointer to LSW of FPACC
         LDH          Set D = 0 for sure
         LEI 170      Set pointer to temp number storage area
         LBI 004      Set precision counter
         CAL MOVEIT   Move FPACC to temporary storage area
NVALID,  CAL INPUT    Fetch OPERATOR from input device
         LBI 000      Clear register B
         CPI 253      Test for '+' sign
         JTZ OPERA1   Go setup for '+' sign
         CPI 255      If not '+' then test for '-' sign
         JTZ OPERA2   Go set up for '-' sign
         CPI 330      If not above, test for X (multiply) sign
         JTZ OPERA3   Go set up for X sign
         CPI 257      If not above, test for / (divide) sign
         JTZ OPERA4   Go set up for / sign
         CPI 377      If none of above, test for RUBOUT
         JFZ NVALID   If none of above then ignore current input
         JMP FPCONT   If ROBOUT then start a new input sequence
OPERA1,  DCB          Setup register B based on above tests
         DCB          Setup register B based on above tests
OPERA2,  DCB          Setup register B based on above tests
         DCB          Setup register B based on above tests
OPERA3,  DCB          Setup register B based on above tests
```

```
                DCB         Setup register B based on above tests
OPERA4,         LCA         Save OPERATOR character in register C
                LAI ***     *** = Next to last location in LOOK-UP table
                ADB         Modify  ***  by contents of register B
                LLI 110     Set pointer to LOOK-UP table address storage
                LMA         Place LOOK-UP address in storage location
                LAC         Restore OPERATOR character to ACC
                CAL ECHO    Display the OPERATOR sign
                CAL SPACES  Display a few spaces after OPERATOR sign
                CAL DINPUT  Let operator enter 2'nd FP decimal number
                CAL SPACES  Provide a few spaces after 2'nd number
                LAI 275     Place ASCII code for = in accumulator
                CAL ECHO    Display '=' sign
                CAL SPACES  Display a few spaces after the '=' sign
                LLI 170     Set pointer to temporary number storage area
                LDH         Set D = 000 for sure
                LEI 134     Set another pointer to LSW of FPOP
                LBI 004     Set precision counter
                CAL MOVEIT  Move 1'st number inputted to FPOP
                LLI 110     Set pointer to LOOK-UP table address storage
                LLM         Bring in LOW order address of LOOK-UP table
                LHI XXX     XXX = PAGE this routine located on!
                LEM         Bring in an address stored in LOOK-UP table
                INL         Residing on this PAGE (XXX) at LOCATIONS
                LDM         '*** + B' and '*** + B + 1' and place it
                LLI Z+1     In registers D and E then change pointer to address
                LME         Part of instruction labeled RESULT below
                INL         And transfer the LOOK-UP table contents so that it
                LMD         Becomes the address portion of  the instruction
                LHI 000     Labeled RESULT, then restore
                LDH         registers H and D to zero
                JMP RESULT  Now JUMP to command labeled RESULT
CRLF2,          LAI 215     Subroutine to provide CR & LF's
                CAL ECHO    Place ASCII code for CR in ACC then display
                LAI 212     Place ASCII code for LF in ACC
                CAL ECHO    Then display
                LAI 215     Do it again, first setup code for CR in ACC
                CAL ECHO    Display it
                LAI 212     Setup code for LF
                CAL ECHO    Display it
                RET         Return to calling routine
SPACES,         LAI 240     Setup code for SPACE in accumulator
                CAL ECHO    Display a SPACE
                LAI 240     Do it again, place code for SPACE in ACC
                CAL ECHO    Display a SPACE
                RET         Return to calling routine
* Z *  RESULT,  CAL DUMMY   CALL the subroutine indicated by current address here
                CAL FPOUT   Display results of the calculation
                JMP FPCONT  Go back and wait for next problem input!
LOOK-UP TABLE   AAA         LOW address for start of FPADD subroutine
                BBB         PAGE address for start of FPADD subroutine
```

CCC	LOW address for start of FPSUB subroutine
DDD	PAGE address for start of FPSUB subroutine
EEE	LOW address for start of FPMULT subroutine
FFF	PAGE address for start of FPMULT subroutine
GGG	LOW address for start of FPDIV subroutine
HHH	PAGE address for start of FPDIV subroutine

The three subroutines, FPINP, FPOUT, and FPCONT as presented would require about three pages of memory for storage. However, as will be discussed in the next chapter, the subroutines could be modified to fit into considerably less memory. The demonstration routines used certain locations on PAGE 00 for storage of transient data and these are listed below for reference. Naturally, the routines could be easily altered to use other temporary storage locations.

LOCATIONS	USAGE
103	Input MANTISSA sign storage
104	Input EXPONENT sign storage
105	Input DIGIT COUNTER
107	Output DIGIT COUNTER
110	Temporary storage for control OPERATOR
150 - 153	Input working area
154 - 156	Input storage registers (for DECBIN conv)
157	Input EXPONENT (decimal equivalent)
160 - 163	Output working area
164 - 167	Output storage registers (for BINDEC conv)
170 - 173	Temporary number storage

ASSEMBLED LISTING OF THE DESCRIBED FLOATING POINT PROGRAM

The following is an assembled listing of the floating point package just described in this chapter as it would appear for an '8008' system. The order in which the major routines appear in the following assembled version is different than the order in which they were presented for explanation. The routines were presented for explanation in a manner related to the increasing complexities of the various portions of the package. The assembled version is arranged more along the logical lines of order of usage. As a guide to the assembled version which is presented next, a memory map shown below gives the starting and ending addresses of the major routines. It may be noted, however, that while the order of the routines have been changed in the assembled version, all of the actual instructions in the routines themselves have been left unchanged. (The assembled version has the comments portion of the listing deleted in order to save space.)

FLOATING POINT PROGRAM MEMORY MAP

ROUTINE	STARTING ADDRESS	ENDING ADDRESS
SCRATCH PAD AREA	PG 00 LOC 100	PG 00 LOC 177
FPCONT	PG 01 LOC 000	PG 01 LOC 243
FPOUT	PG 01 LOC 244	PG 02 LOC 263
DINPUT	PG 02 LOC 264	PG 04 LOC 107
FPNORM	PG 04 LOC 110	PG 04 LOC 237
FPADD	PG 04 LOC 240	PG 05 LOC 114
FSUB	PG 05 LOC 115	PG 05 LOC 126
FPMULT	PG 05 LOC 127	PG 06 LOC 021
FPDIV	PG 06 LOC 022	PG 06 LOC 254
UTILITY ROUTINES	PG 06 LOC 255	PG 07 LOC 004

The assembled version assumes that user defined routines for INPUT and OUTPUT to an I/O device, as well as user defined routines for displaying an attempted divide by zero operation as well as re-directing program operation after such an error, will reside at the locations indicated below.

ROUTINE	STARTING ADDRESS	DEFINITION
DERMSG	PG 07 LOC 100	Attempted divide by zero error message
USERDF	PG 07 LOC 160	Direct program flow after above error
INPUT	PG 07 LOC 200	ASCII input routine
ECHO	PG 07 LOC 300	ASCII Output routine

ASSEMBLED LISTING OF THE FLOATING POINT PROGRAM FOR AN '8 0 0 8' SYSTEM

ADDRESS	MACHINE CODE		MNEMONICS
001 000	106 163 001	FPCONT,	CAL CRLF2
001 003	106 264 002		CAL DINPUT
001 006	106 210 001		CAL SPACES
001 011	066 124		LLI 124
001 013	335		LDH
001 014	046 170		LEI 170
001 016	016 004		LBI 004
001 020	106 076 005		CAL MOVEIT
001 023	106 200 007	NVALID,	CAL INPUT
001 026	016 000		LBI 000
001 030	074 253		CPI 253

```
001 032          150 064 001                JTZ OPERA1
001 035          074 255                    CPI 255
001 037          150 066 001                JTZ OPERA2
001 042          074 330                    CPI 330
001 044          150 070 001                JTZ OPERA3
001 047          074 257                    CPI 257
001 051          150 072 001                JTZ OPERA4
001 054          074 377                    CPI 377
001 056          110 023 001                JFZ NVALID
001 061          104 000 001                JMP FPCONT
001 064          011                OPERA1, DCB
001 065          011                        DCB
001 066          011                OPERA2, DCB
001 067          011                        DCB
001 070          011                OPERA3, DCB
001 071          011                        DCB
001 072          320                OPERA4, LCA
001 073          006 242                    LAI 242
001 075          201                        ADB
001 076          066 110                    LLI 110
001 100          370                        LMA
001 101          302                        LAC
001 102          106 300 007                CAL ECHO
001 105          106 210 001                CAL SPACES
001 110          106 264 002                CAL DINPUT
001 113          106 210 001                CAL SPACES
001 116          006 275                    LAI 275
001 120          106 300 007                CAL ECHO
001 123          106 210 001                CAL SPACES
001 126          066 170                    LLI 170
001 130          335                        LDH
001 131          046 134                    LEI 134
001 133          016 004                    LBI 004
001 135          106 076 005                CAL MOVEIT
001 140          066 110                    LLI 110
001 142          367                        LLM
001 143          056 001                    LHI 001
001 145          347                        LEM
001 146          060                        INL
001 147          337                        LDM
001 150          066 224                    LLI 224
001 152          374                        LME
001 153          060                        INL
001 154          373                        LMD
001 155          056 000                    LHI 000
001 157          335                        LDH
001 160          104 223 001                JMP RESULT
001 163          006 215            CRLF2,  LAI 215
001 165          106 300 007                CAL ECHO
001 170          006 212                    LAI 212
001 172          106 300 007                CAL ECHO
```

```
001 175          006 215                LAI 215
001 177          106 300 007            CAL ECHO
001 202          006 212                LAI 212
001 204          106 300 007            CAL ECHO
001 207          007                    RET
001 210          006 240        SPACES, LAI 240
001 212          106 300 007            CAL ECHO
001 215          006 240                LAI 240
001 217          106 300 007            CAL ECHO
001 222          007                    RET
001 223          106 000 000    RESULT, CAL DUMMY
001 226          106 244 001            CAL FPOUT
001 231          104 000 001            JMP FPCONT
001 234          240                    240
001 235          004                    004
001 236          115                    115
001 237          005                    005
001 240          127                    127
001 241          005                    005
001 242          022                    022
001 243          006                    006

001 244          066 157        FPOUT,  LLI 157
001 246          076 000                LMI 000
001 250          066 126                LLI 126
001 252          307                    LAM
001 253          240                    NDA
001 254          160 264 001            JTS OUTNEG
001 257          006 253                LAI 253
001 261          104 275 001            JMP AHEAD1
001 264          066 124        OUTNEG, LLI 124
001 266          016 003                LBI 003
001 270          106 311 006            CAL COMPLM
001 273          006 255                LAI 255
001 275          106 300 007    AHEAD1, CAL ECHO
001 300          006 260                LAI 260
001 302          106 300 007            CAL ECHO
001 305          006 256                LAI 256
001 307          106 300 007            CAL ECHO
001 312          066 127                LLI 127
001 314          006 377                LAI 377
001 316          207                    ADM
001 317          370                    LMA
001 320          120 343 001    DECEXT, JFS DECEXD
001 323          006 004                LAI 004
001 325          207                    ADM
001 326          120 351 001            JFS DECOUT
001 331          106 300 003            CAL FPX10
001 334          066 127        DECREP, LLI 127
001 336          307                    LAM
001 337          240                    NDA
```

```
001 340          104 320 001                 JMP DECEXT
001 343          106 346 003     DECEXD,     CAL FPD10
001 346          104 334 001                 JMP DECREP
001 351          046 164         DECOUT,     LEI 164
001 353          335                         LDH
001 354          066 124                     LLI 124
001 356          016 003                     LBI 003
001 360          106 076 005                 CAL MOVEIT
001 363          066 167                     LLI 167
001 365          076 000                     LMI 000
001 367          066 164                     LLI 164
001 371          016 003                     LBI 003
001 373          106 340 006                 CAL ROTATL
001 376          106 122 002                 CAL OUTX10
002 001          066 127         COMPEN,     LLI 127
002 003          317                         LBM
002 004          010                         INB
002 005          371                         LMB
002 006          150 023 002                 JTZ OUTDIG
002 011          066 167                     LLI 167
002 013          016 004                     LBI 004
002 015          106 352 006                 CAL ROTATR
002 020          104 001 002                 JMP COMPEN
002 023          066 107         OUTDIG,     LLI 107
002 025          076 007                     LMI 007
002 027          066 167                     LLI 167
002 031          307                         LAM
002 032          240                         NDA
002 033          150 064 002                 JTZ ZERODG
002 036          066 167         OUTDGS,     LLI 167
002 040          006 260                     LAI 260
002 042          207                         ADM
002 043          106 300 007                 CAL ECHO
002 046          066 107         DECRDG,     LLI 107
002 050          106 305 006                 CAL CNTDWN
002 053          150 177 002                 JTZ EXPOUT
002 056          106 122 002                 CAL OUTX10
002 061          104 036 002                 JMP OUTDGS
002 064          066 157         ZERODG,     LLI 157
002 066          106 305 006                 CAL CNTDWN
002 071          066 166                     LLI 166
002 073          307                         LAM
002 074          240                         NDA
002 075          110 046 002                 JFZ DECRDG
002 100          061                         DCL
002 101          307                         LAM
002 102          240                         NDA
002 103          110 046 002                 JFZ DECRDG
002 106          061                         DCL
002 107          307                         LAM
002 110          240                         NDA
```

```
002 111          110 046 002                  JFZ DECRDG
002 114          066 157                      LLI 157
002 116          370                          LMA
002 117          104 046 002                  JMP DECRDG

002 122          066 167              OUTX10, LLI 167
002 124          076 000                      LMI 000
002 126          066 164                      LLI 164
002 130          335                          LDH
002 131          046 160                      LEI 160
002 133          016 004                      LBI 004
002 135          106 076 005                  CAL MOVEIT
002 140          066 164                      LLI 164
002 142          016 004                      LBI 004
002 144          106 340 006                  CAL ROTATL
002 147          066 164                      LLI 164
002 151          016 004                      LBI 004
002 153          106 340 006                  CAL ROTATL
002 156          066 160                      LLI 160
002 160          046 164                      LEI 164
002 162          016 004                      LBI 004
002 164          106 255 006                  CAL ADDER
002 167          066 164                      LLI 164
002 171          016 004                      LBI 004
002 173          106 340 006                  CAL ROTATL
002 176          007                          RET
002 177          006 305              EXPOUT, LAI 305
002 201          106 300 007                  CAL ECHO
002 204          066 157                      LLI 157
002 206          307                          LAM
002 207          240                          NDA
002 210          160 220 002                  JTS EXOUTN
002 213          006 253                      LAI 253
002 215          104 227 002                  JMP AHEAD2
002 220          054 377              EXOUTN, XRI 377
002 222          004 001                      ADI 001
002 224          370                          LMA
002 225          006 255                      LAI 255
002 227          106 300 007          AHEAD2, CAL ECHO
002 232          016 000                      LBI 000
002 234          307                          LAM
002 235          024 012              SUB12,  SUI 012
002 237          160 247 002                  JTS TOMUCH
002 242          370                          LMA
002 243          010                          INB
002 244          104 235 002                  JMP SUB12
002 247          006 260              TOMUCH, LAI 260
002 251          201                          ADB
002 252          106 300 007                  CAL ECHO
002 255          307                          LAM
002 256          004 260                      ADI 260
```

```
002 260          106 300 007                CAL ECHO
002 263          007                        RET

002 264          056 000            DINPUT, LHI 000
002 266          066 150                    LLI 150
002 270          250                        XRA
002 271          016 010                    LBI 010
002 273          370                CLRNX2, LMA
002 274          060                        INL
002 275          011                        DCB
002 276          110 273 002                JFZ CLRNX2
002 301          066 103                    LLI 103
002 303          016 004                    LBI 004
002 305          370                CLRNX3, LMA
002 306          060                        INL
002 307          011                        DCB
002 310          110 305 002                JFZ CLRNX3
002 313          106 200 007                CAL INPUT
002 316          074 253                    CPI 253
002 320          150 333 002                JTZ SECHO
002 323          074 255                    CPI 255
002 325          110 341 002                JFZ NOTPLM
002 330          066 103                    LLI 103
002 332          370                        LMA
002 333          106 300 007        SECHO,  CAL ECHO
002 336          106 200 007        NINPUT, CAL INPUT
002 341          074 377            NOTPLM, CPI 377
002 343          150 046 003                JTZ ERASE
002 346          074 256                    CPI 256
002 350          150 022 003                JTZ PERIOD
002 353          074 305                    CPI 305
002 355          150 066 003                JTZ FNDEXP
002 360          074 260                    CPI 260
002 362          160 170 003                JTS ENDINP
002 365          074 272                    CPI 272
002 367          120 170 003                JFS ENDINP
002 372          066 156                    LLI 156
002 374          310                        LBA
002 375          006 370                    LAI 370
002 377          247                        NDM
003 000          110 336 002                JFZ NINPUT
003 003          301                        LAB
003 004          106 300 007                CAL ECHO
003 007          066 105                    LLI 105
003 011          327                        LCM
003 012          020                        INC
003 013          372                        LMC
003 014          106 006 004                CAL DECBIN
003 017          104 336 002                JMP NINPUT
003 022          310                PERIOD, LBA
003 023          066 106                    LLI 106
```

```
003 025          307                        LAM
003 026          240                        NDA
003 027          110 170 003                JFZ ENDINP
003 032          066 105                    LLI 105
003 034          370                        LMA
003 035          060                        INL
003 036          371                        LMB
003 037          301                        LAB
003 040          106 300 007                CAL ECHO
003 043          104 336 002                JMP NINPUT
003 046          006 274          ERASE,    LAI 274
003 050          106 300 007                CAL ECHO
003 053          006 240                    LAI 240
003 055          106 300 007                CAL ECHO
003 060          106 300 007                CAL ECHO
003 063          104 264 002                JMP DINPUT
003 066          106 300 007      FNDEXP,   CAL ECHO
003 071          106 200 007                CAL INPUT
003 074          074 253                    CPI 253
003 076          150 111 003                JTZ EXECHO
003 101          074 255                    CPI 255
003 103          110 117 003                JFZ NOEXPS
003 106          066 104                    LLI 104
003 110          370                        LMA
003 111          106 300 007      EXECHO,   CAL ECHO
003 114          106 200 007      EXPINP,   CAL INPUT
003 117          074 377          NOEXPS,   CPI 377
003 121          150 046 003                JTZ ERASE
003 124          074 260                    CPI 260
003 126          160 170 003                JTS ENDINP
003 131          074 272                    CPI 272
003 133          120 170 003                JFS ENDINP
003 136          044 017                    NDI 017
003 140          310                        LBA
003 141          066 157                    LLI 157
003 143          006 003                    LAI 003
003 145          277                        CPM
003 146          160 114 003                JTS EXPINP
003 151          327                        LCM
003 152          307                        LAM
003 153          240                        NDA
003 154          022                        RAL
003 155          022                        RAL
003 156          202                        ADC
003 157          022                        RAL
003 160          201                        ADB
003 161          370                        LMA
003 162          006 260                    LAI 260
003 164          201                        ADB
003 165          104 111 003                JMP EXECHO
003 170          066 103          ENDINP,   LLI 103
```

```
003 172          307                    LAM
003 173          240                    NDA
003 174          150 206 003            JTZ FININP
003 177          066 154                LLI 154
003 201          016 003                LBI 003
003 203          106 311 006            CAL COMPLM
003 206          066 153        FININP, LLI 153
003 210          250                    XRA
003 211          330                    LDA
003 212          370                    LMA
003 213          046 123                LEI 123
003 215          016 004                LBI 004
003 217          106 076 005            CAL MOVEIT
003 222          016 027                LBI 027
003 224          106 110 004            CAL FPNORM
003 227          066 104                LLI 104
003 231          307                    LAM
003 232          240                    NDA
003 233          066 157                LLI 157
003 235          150 246 003            JTZ POSEXP
003 240          307                    LAM
003 241          054 377                XRI 377
003 243          004 001                ADI 001
003 245          370                    LMA
003 246          066 106        POSEXP, LLI 106
003 250          307                    LAM
003 251          240                    NDA
003 252          150 261 003            JTZ EXPOK
003 255          066 105                LLI 105
003 257          250                    XRA
003 260          227                    SUM
003 261          066 157        EXPOK,  LLI 157
003 263          207                    ADM
003 264          370                    LMA
003 265          160 337 003            JTS MINEXP
003 270          053                    RTZ
003 271          106 300 003    EXPFIX, CAL FPX10
003 274          110 271 003            JFZ EXPFIX
003 277          007                    RET
003 300          046 134        FPX10,  LEI 134
003 302          335                    LDH
003 303          066 124                LLI 124
003 305          016 004                LBI 004
003 307          106 076 005            CAL MOVEIT
003 312          066 127                LLI 127
003 314          076 004                LMI 004
003 316          061                    DCL
003 317          076 120                LMI 120
003 321          061                    DCL
003 322          250                    XRA
003 323          370                    LMA
```

```
003 324          061                        DCL
003 325          370                        LMA
003 326          106 127 005                CAL FPMULT
003 331          066 157                    LLI 157
003 333          106 305 006                CAL CNTDWN
003 336          007                        RET
003 337          106 346 003    MINEXP,     CAL FPD10
003 342          110 337 003                JFZ MINEXP
003 345          007                        RET
003 346          046 134        FPD10,      LEI 134
003 350          335                        LDH
003 351          066 124                    LLI 124
003 353          016 004                    LBI 004
003 355          106 076 005                CAL MOVEIT
003 360          066 127                    LLI 127
003 362          076 375                    LMI 375
003 364          061                        DCL
003 365          076 146                    LMI 146
003 367          061                        DCL
003 370          076 146                    LMI 146
003 372          061                        DCL
003 373          076 147                    LMI 147
003 375          106 127 005                CAL FPMULT
004 000          006 157                    LLI 157
004 002          317                        LBM
004 003          010                        INB
004 004          371                        LMB
004 005          007                        RET

004 006          066 153        DECBIN,     LLI 153
004 010          301                        LAB
004 011          044 017                    NDI 017
004 013          370                        LMA
004 014          046 150                    LEI 150
004 016          066 154                    LLI 154
004 020          335                        LDH
004 021          016 003                    LBI 003
004 023          106 076 005                CAL MOVEIT
004 026          066 154                    LLI 154
004 030          016 003                    LBI 003
004 032          106 340 006                CAL ROTATL
004 035          066 154                    LLI 154
004 037          016 003                    LBI 003
004 041          106 340 006                CAL ROTATL
004 044          046 154                    LEI 154
004 046          066 150                    LLI 150
004 050          016 003                    LBI 003
004 052          106 255 006                CAL ADDER
004 055          066 154                    LLI 154
004 057          016 003                    LBI 003
004 061          106 340 006                CAL ROTATL
```

```
004 064          066 152                    LLI 152
004 066          250                        XRA
004 067          370                        LMA
004 070          061                        DCL
004 071          370                        LMA
004 072          066 153                    LLI 153
004 074          307                        LAM
004 075          066 150                    LLI 150
004 077          370                        LMA
004 100          046 154                    LEI 154
004 102          016 003                    LBI 003
004 104          106 255 006                CAL ADDER
004 107          007                        RET

004 110          301              FPNORM,   LAB
004 111          240                        NDA
004 112          150 120 004                JTZ NOEXCO
004 115          066 127                    LLI 127
004 117          371                        LMB
004 120          066 126          NOEXCO,   LLI 126
004 122          307                        LAM
004 123          066 100                    LLI 100
004 125          240                        NDA
004 126          160 136 004                JTS ACCMIN
004 131          250                        XRA
004 132          370                        LMA
004 133          104 146 004                JMP ACZERT
004 136          370              ACCMIN,   LMA
004 137          016 004                    LBI 004
004 141          066 123                    LLI 123
004 143          106 311 006                CAL COMPLM
004 146          066 126          ACZERT,   LLI 126
004 150          016 004                    LBI 004
004 152          307              LOOK0,    LAM
004 153          240                        NDA
004 154          110 171 004                JFZ ACNONZ
004 157          061                        DCL
004 160          011                        DCB
004 161          110 152 004                JFZ LOOK0
004 164          066 127                    LLI 127
004 166          250                        XRA
004 167          370                        LMA
004 170          007                        RET
004 171          066 123          ACNONZ,   LLI 123
004 173          016 004                    LBI 004
004 175          106 340 006                CAL ROTATL
004 200          307                        LAM
004 201          240                        NDA
004 202          160 214 004                JTS ACCSET
004 205          060                        INL
004 206          106 305 006                CAL CNTDWN
```

```
004 211          104 171 004                 JMP ACNONZ
004 214          066 126             ACCSET, LLI 126
004 216          016 003                     LBI 003
004 220          106 352 006                 CAL ROTATR
004 223          006 100                     LLI 100
004 225          307                         LAM
004 226          240                         NDA
004 227          023                         RFS
004 230          066 124                     LLI 124
004 232          016 003                     LBI 003
004 234          106 311 006                 CAL COMPLM
004 237          007                         RET

004 240          066 126             FPADD,  LLI 126
004 242          016 003                     LBI 003
004 244          307                 CKZACC, LAM
004 245          240                         NDA
004 246          110 275 004                 JFZ NONZAC
004 251          011                         DCB
004 252          150 261 004                 JTZ MOVOP
004 255          061                         DCL
004 256          104 244 004                 JMP CKZACC
004 261          106 276 006         MOVOP,  CAL SWITCH
004 264          353                         LHD
004 265          066 134                     LLI 134
004 267          016 004                     LBI 004
004 271          106 076 005                 CAL MOVEIT
004 274          007                         RET
004 275          066 136             NONZAC, LLI 136
004 277          016 003                     LBI 003
004 301          307                 CKZOP,  LAM
004 302          240                         NDA
004 303          110 314 004                 JFZ CKEQEX
004 306          011                         DCB
004 307          053                         RTZ
004 310          061                         DCL
004 311          104 301 004                 JMP CKZOP
004 314          066 127             CKEQEX, LLI 127
004 316          307                         LAM
004 317          066 137                     LLI 137
004 321          277                         CPM
004 322          150 016 005                 JTZ SHACOP
004 325          054 377                     XRI 377
004 327          004 001                     ADI 001
004 331          207                         ADM
004 332          120 341 004                 JFS SKPNEG
004 335          054 377                     XRI 377
004 337          004 001                     ADI 001
004 341          074 030             SKPNEG, CPI 030
004 343          160 360 004                 JTS LINEUP
004 346          307                         LAM
```

```
004 347          066 127                    LLI 127
004 351          227                        SUM
004 352          063                        RTS
004 353          066 124                    LLI 124
004 355          104 261 004                JMP MOVOP
004 360          307              LINEUP,   LAM
004 361          066 127                    LLI 127
004 363          227                        SUM
004 364          160 004 005                JTS SHIFTO
004 367          320                        LCA
004 370          066 127          MORACC,   LLI 127
004 372          106 046 005                CAL SHLOOP
004 375          021                        DCC
004 376          110 370 004                JFZ MORACC
005 001          104 016 005                JMP SHACOP
005 004          320              SHIFTO,   LCA
005 005          066 137          MOROP,    LLI 137
005 007          106 046 005                CAL SHLOOP
005 012          020                        INC
005 013          110 005 005                JFZ MOROP
005 016          066 123          SHACOP,   LLI 123
005 020          076 000                    LMI 000
005 022          066 127                    LLI 127
005 024          106 052 005                CAL SHLOOP
005 027          066 137                    LLI 137
005 031          106 052 005                CAL SHLOOP
005 034          335                        LDH
005 035          046 123                    LEI 123
005 037          016 004                    LBI 004
005 041          106 255 006                CAL ADDER
005 044          016 000                    LBI 000
005 046          106 110 004                CAL FPNORM
005 051          007                        RET
005 052          317              SHLOOP,   LBM
005 053          010                        INB
005 054          371                        LMB
005 055          061                        DCL
005 056          016 004                    LBI 004
005 060          307              FSHIFT,   LAM
005 061          240                        NDA
005 062          160 071 005                JTS BRING1
005 065          106 352 006                CAL ROTATR
005 070          007                        RET
005 071          022              BRING1,   RAL
005 072          106 353 006                CAL ROTR
005 075          007                        RET
005 076          307              MOVEIT,   LAM
005 077          060                        INL
005 100          106 276 006                CAL SWITCH
005 103          370                        LMA
005 104          060                        INL
```

```
005 105          106 276 006                CAL SWITCH
005 110          011                        DCB
005 111          053                        RTZ
005 112          104 076 005                JMP MOVEIT

005 115          066 124            FSUB,   LLI 124
005 117          016 003                    LBI 003
005 121          106 311 006                CAL COMPLM
005 124          104 240 004                JMP FPADD

005 127          106 257 005        FPMULT, CAL CKSIGN
005 132          066 137            ADDEXP, LLI 137
005 134          307                        LAM
005 135          066 127                    LLI 127
005 137          207                        ADM
005 140          004 001                    ADI 001
005 142          370                        LMA
005 143          066 102            SETMCT, LLI 102
005 145          076 027                    LMI 027
005 147          066 126            MULTIP, LLI 126
005 151          016 003                    LBI 003
005 153          106 352 006                CAL ROTATR
005 156          142 367 005                CTC ADOPPP
005 161          066 146                    LLI 146
005 163          016 006                    LBI 006
005 165          106 352 006                CAL ROTATR
005 170          066 102                    LLI 102
005 172          106 305 006                CAL CNTDWN
005 175          110 147 005                JFZ MULTIP
005 200          066 146                    LLI 146
005 202          016 006                    LBI 006
005 204          106 352 006                CAL ROTATR
005 207          066 143                    LLI 143
005 211          307                        LAM
005 212          022                        RAL
005 213          300                        LAA
005 214          240                        NDA
005 215          162 002 006                CTS MROUND
005 220          066 123                    LLI 123
005 222          106 276 006                CAL SWITCH
005 225          353                        LHD
005 226          066 143                    LLI 143
005 230          016 004                    LBI 004

005 232          106 076 005        EXMLDV, CAL MOVEIT
005 235          016 000                    LBI 000
005 237          106 110 004                CAL FPNORM
005 242          066 101                    LLI 101
005 244          307                        LAM
005 245          240                        NDA
005 246          013                        RFZ
```

```
005 247          066 124                    LLI 124
005 251          016 003                    LBI 003
005 253          106 311 006                CAL COMPLM
005 256          007                        RET
005 257          106 336 005    CKSIGN,     CAL CLRWRK
005 262          066 101                    LLI 101
005 264          076 001                    LMI 001
005 266          066 126                    LLI 126
005 270          307                        LAM
005 271          240                        NDA
005 272          160 317 005                JTS NEGFPA
005 275          066 136        OPSGNT,     LLI 136
005 277          307                        LAM
005 300          240                        NDA
005 301          023                        RFS
005 302          066 101                    LLI 101
005 304          106 305 006                CAL CNTDWN
005 307          066 134                    LLI 134
005 311          016 003                    LBI 003
005 313          106 311 006                CAL COMPLM
006 316          007                        RET

005 317          066 101        NEGFPA,     LLI 101
005 321          106 305 006                CAL CNTDWN
005 324          066 124                    LLI 124
005 326          016 003                    LBI 003
005 330          106 311 006                CAL COMPLM
005 333          104 275 005                JMP OPSGNT
005 336          066 140        CLRWRK,     LLI 140
005 340          016 010                    LBI 010
005 342          250                        XRA
005 343          370            CLRNEX,     LMA
005 344          011                        DCB
005 345          150 354 005                JTZ CLROPL
005 350          060                        INL
005 351          104 343 005                JMP CLRNEX
005 354          016 004        CLROPL,     LBI 004
005 356          066 130                    LLI 130
005 360          370            CLRNX1,     LMA
005 361          011                        DCB
005 362          053                        RTZ
005 363          060                        INL
005 364          104 360 005                JMP CLRNX1
005 367          046 141        ADOPPP,     LEI 141
005 371          335                        LDH
005 372          066 131                    LLI 131
005 374          016 006                    LBI 006
005 376          106 255 006                CAL ADDER
006 001          007                        RET
006 002          016 003        MROUND,     LBI 003
006 004          006 100                    LAI 100
```

```
006 006          207                    ADM
006 007          370            CROUND, LMA
006 010          060                    INL
006 011          006 000                LAI 000
006 013          217                    ACM
006 014          011                    DCB
006 015          110 007 006            JFZ CROUND
006 020          370                    LMA
006 021          007                    RET

006 022          106 257 005    FPDIV,  CAL CKSIGN
006 025          066 126                LLI 126
006 027          006 000                LAI 000
006 031          277                    CPM
006 032          110 047 006            JFZ SUBEXP
006 035          061                    DCL
006 036          277                    CPM
006 037          110 047 006            JFZ SUBEXP
006 042          061                    DCL
006 043          277                    CPM
006 044          150 247 006            JTZ DERROR
006 047          066 137        SUBEXP, LLI 137
006 051          307                    LAM
006 052          066 127                LLI 127
006 054          227                    SUM
006 055          004 001                ADI 001
006 057          370                    LMA
006 060          066 102        SETDCT, LLI 102
006 062          076 027                LMI 027
006 064          106 216 006    DIVIDE, CAL SETSUB
006 067          160 111 006            JTS NOGO
006 072          046 134                LEI 134
006 074          066 131                LLI 131
006 076          016 003                LBI 003
006 100          106 076 005            CAL MOVEIT
006 103          006 001                LAI 001
006 105          032                    RAR
006 106          104 114 006            JMP QUOROT
006 111          006 000        NOGO,   LAI 000
006 113          032                    RAR
006 114          066 144        QUOROT, LLI 144
006 116          016 003                LBI 003
006 120          106 341 006            CAL ROTL
006 123          066 134                LLI 134
006 125          016 003                LBI 003
006 127          106 340 006            CAL ROTATL
006 132          066 102                LLI 102
006 134          106 305 006            CAL CNTDWN
006 137          110 064 006            JFZ DIVIDE
006 142          106 216 006            CAL SETSUB
006 145          120 205 006            JFS DVEXIT
```

```
006 150          066 144                    LLI 144
006 152          307                        LAM
006 153          004 001                    ADI 001
006 155          370                        LMA
006 156          006 000                    LAI 000
006 160          060                        INL
006 161          217                        ACM
006 162          370                        LMA
006 163          006 000                    LAI 000
006 165          060                        INL
006 166          217                        ACM
006 167          370                        LMA
006 170          120 205 006                JFS DVEXIT
006 173          016 003                    LBI 003
006 175          106 352 006                CAL ROTATR
006 200          066 127                    LLI 127
006 202          317                        LBM
006 203          060                        INL
006 204          371                        LMB
006 205          066 144          DVEXIT,   LLI 144
006 207          046 124                    LEI 124
006 211          016 003                    LBI 003
006 213          104 232 005                JMP EXMLDV
006 216          066 131          SETSUB,   LLI 131
006 220          106 276 006                CAL SWITCH
006 223          353                        LHD
006 224          066 124                    LLI 124
006 226          016 003                    LBI 003
006 230          106 076 005                CAL MOVEIT
007 233          046 131                    LEI 131
006 235          066 134                    LLI 134
006 237          016 003                    LBI 003
006 241          106 364 006                CAL SUBBER
006 244          307                        LAM
006 245          240                        NDA
006 246          007                        RET
006 247          106 100 007      DERROR,   CAL DERMSG
006 252          104 160 007                JMP USERDF

006 255          240              ADDER,    NDA
006 256          307              ADDMOR,   LAM
006 257          106 276 006                CAL SWITCH
006 262          217                        ACM
006 263          370                        LMA
006 264          011                        DCB
006 265          053                        RTZ
006 266          060                        INL
006 267          106 276 006                CAL SWITCH
006 272          060                        INL
006 273          104 256 006                JMP ADDMOR
```

```
006 276          325            SWITCH,  LCH
006 277          353                     LHD
006 300          332                     LDC
006 301          326                     LCL
006 302          364                     LLE
006 303          342                     LEC
006 304          007                     RET

006 305          327            CNTDWN,  LCM
006 306          021                     DCC
006 307          372                     LMC
006 310          007                     RET

006 311          307            COMPLM,  LAM
006 312          054 377                 XRI 377
006 314          004 001                 ADI 001
006 316          370            MORCOM,  LMA
006 317          032                     RAR
006 320          330                     LDA
006 321          011                     DCB
006 322          053                     RTZ
006 323          060                     INL
006 324          307                     LAM
006 325          054 377                 XRI 377
006 327          340                     LEA
006 330          303                     LAD
006 331          022                     RAL
006 332          006 000                 LAI 000
006 334          214                     ACE
006 335          104 316 006             JMP MORCOM

006 340          240            ROTATL,  NDA
006 341          307            ROTL,    LAM
006 342          022                     RAL
006 343          370                     LMA
006 344          011                     DCB
006 345          053                     RTZ
006 346          060                     INL
006 347          104 341 006             JMP ROTL

006 352          240            ROTATR,  NDA
006 353          307            ROTR,    LAM
006 354          032                     RAR
006 355          370                     LMA
006 356          011                     DCB
006 357          053                     RTZ
006 360          061                     DCL
006 361          104 353 006             JMP ROTR

006 364          240            SUBBER,  NDA
006 365          307            SUBTRA,  LAM
```

```
006 366          106 276 006          CAL SWITCH
006 371          237                  SBM
006 372          370                  LMA
006 373          011                  DCB
006 374          053                  RTZ
006 375          060                  INL
006 376          106 276 006          CAL SWITCH
007 001          060                  INL
007 002          104 365 006          JMP SUBTRA

007 100                               DERMSG,

007 160                               USERDF,

007 200                               INPUT,

007 300                               ECHO,
```