NICOLET 1080 SERIES

# FLOATING POINT PACKAGE—1972

Description and Instructions

for

NIC-80/S-7209

Programming Manual—Volume II

Nicolet Instrument Corporation
5225 Verona Road
Madison, Wisconsin

August 1972

# TABLE OF CONTENTS

# FLOATING POINT PACKAGE -- 1972

## I. DESCRIPTION OF THE FLOATING POINT PACKAGE

### A. Introduction

The NIC Floating Point Package (FPP) is a collection of subroutines which free the user from the need to program complex arithmetic operations. Each of the routines operates on a number in a floating point format, similar to that of scientific notation. Floating Point-1972 occupies locations 6000-7577 and consists of two parts: the Basic Arithmetic section, and the Extended Functions section. These routines assume the presence of the hardware multiply-divide circuitry now standard on all NIC-1080 computers.

### B. Reasons for Using the Floating Point Routines

The NIC-1080 computer stores all information in 20-bit memory words, in which one can represent unsigned integers from 0 to $2^{20}-1$. This is equivalent to $0 - 1,048,575_{10}$ or $0 - 3777777_8$. If one chooses to operate on signed numbers, the range drops to $-2^{19}$ to $2^{19}-1$, or $-524,288$ to $+524,287_{10}$. Note that these large numbers contain nearly six significant figures. However, if one is handling small integers such as 20 or 6 or 1, the number of significant figures drops off rapidly. Further, it is not possible to represent fractional numbers successfully within the limits of 20 bits without reducing the number of significant figures even more drastically.

One solution to this problem is to represent each number in two 20-bit words, or double precision, allowing one word to represent the integer part and the other word to represent the fractional part. However, the same problem arises in this format concerning very large and very small numbers. There is no effective way to represent numbers such as $10^{15}$, or $10^{-12}$ and still maintain the same significance.

The Floating Point package overcomes these problems by utilizing an internal computer representation similar to scientific notation. In scientific notation, one represents all numbers in the form $n.nnnn \times 10^{nnn}$. By convention, all numbers are reduced to lie between 1 and 10, and are multiplied by an appropriate power of ten.

Similarly, the internal representation or floating point format requires that the number lie between 0.5 and 1.0 and that the exponent be adjusted appropriately. It is customary, although not altogether accurate, to refer to the number as the mantissa, and the power to which the base is raised to as the exponent. Since the 1080 is a binary machine, the exponent and mantissa are both binary (base 2) numbers.

C. Conversion to Binary Representation

Let us consider a simple example of this conversion procedure. The decimal number 5 is represented in binary as 101. This is scaled right to lie in the requisite range as follows:

$$101 \times 2^0$$
$$10.1 \times 2^1$$
$$1.01 \times 2^2$$
$$.101 \times 2^3$$

The last item in this list, $.101 \times 2^3$, is the representation used by the Floating Point package. Since this is binary notation, the period to the left of the mantissa is called the binary point.

Just as the decimal fraction .213 means

$$2 \times 10^{-1}$$
$$+1 \times 10^{-2}$$
$$+3 \times 10^{-3}$$

the binary fraction .101 means

$$1 \times 2^{-1}$$
$$+0 \times 2^{-2}$$
$$+1 \times 2^{-3}$$

Two more examples of conversions are given below.

$$50_{10} = 62_8 = 110\ 010_2$$

Shifting right, this equals $.110010 \times 2^6$

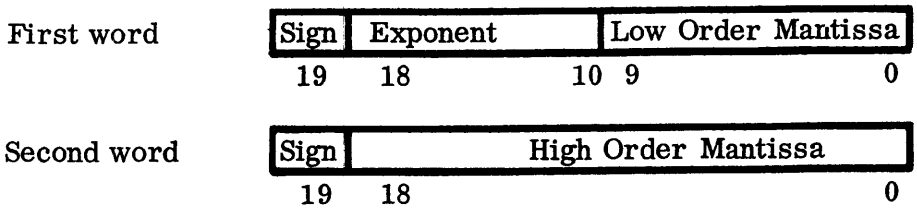
The representation of fractions in octal and binary is somewhat harder to grasp, but since the program takes care of all such conversions internally, it is not usually necessary to become too familiar with this technique. If we wished to represent 0.75 as a binary number, we need only recognize that 0.75 = 0.50 + 0.25, and that this is equivalent to $2^{-1} + 2^{-2}$. Thus, $0.75 = 0.11 \times 2^0$.

D. Internal Representation of Binary Fractions

Each floating point number is taken to occupy two consecutive locations. These two 20-bit words are divided so that the exponent occupies 10 bits and the mantissa 30 bits. This division allows us to represent numbers having signed exponents in the range $\pm 2^9$ or $\pm 512$. This is equivalent to roughly $10^{-150}$ to $10^{+150}$. The 30-bit mantissa is used to represent signed numbers in the range $\pm 2^{29}$. This range is greater

than ± 500,000,000 and thus implies an accuracy of better than eight decimal digits.

The two computer memory words are divided so that the sign of the exponent and mantissa are each represented by the sign bit (bit 19) for rapid access during calculations. Thus, the left hand ten bits of the first word constitute the exponent, with bit 19 the sign, and the second word represents the most significant 19 bits (plus sign) of the mantissa. The right hand ten bits of the first word, then, represent the least significant 10 bits of the mantissa. This is illustrated below.




A negative mantissa or exponent is represented by its two's complement. Thus, if the first word begins with octal digit 2 or 3 (bit 19 = 1) the exponent is negative and if the second word begins with a 2 or 3, the mantissa is negative.

In summary, the principal advantages of using the floating point routines are

(1) All numbers are represented to the same number of significant figures.
(2) A much larger range of magnitudes can be represented.
(3) The programmer need not keep track of a binary point.
(4) Simplified programming of mathematical functions.

It should be pointed out that there are a few disadvantages to the use of these routines, the most important being:

(1) The execution time is much slower than similar integer routines.
(2) An entire page of memory is required for the subroutines, so that less space is available for programming.

## II. PROGRAMMING USING THE FLOATING POINT PACKAGE

### A. Pseudo-Registers

The FPP utilizes two pseudo-registers which behave as if they were actually hardware registers: the Floating Accumulator (FAC) and the Floating Argument (FAR). These are actually a set of memory locations, but their loading and operation is handled entirely by software internal to the FPP.

Just as all numerical operations appear to occur in the hardware accumulator (AC) all floating point operations appear to occur in the FAC. Whenever an operation requires two numbers, such as floating point addition, one number is loaded into each of these registers prior to calling the subroutine to perform the addition. The result is always held in the FAC. In all basic arithmetic operations, the FAR is destroyed by the computation.

Locations 7572 and 7573 constitute the FAC and have been given the names FACE and FACM, representing FAC-exponent and FAC-mantissa. During calculations internal to the FPP, the FAC is expanded into three locations: FACE, FACM and FACML, where FACML is location 7574. The FAR occupies 7575, 7576 and 7577. At the end of internal calculation, the calculated result is rounded to 30 significant bits and re-packed into the two-word format.

### B. Offpage Subroutine Calls

Since the FPP resides on page 6000, all calls to these subroutines must be in the form of an indirect call. It is convenient to give the subroutine pointers the same names as the actual subroutines as a simple way of remembering the function of the subroutine. Each memory page which refers to the FPP must have its own set of pointers, however, and in the event that more than one page is assembled at a time, different names must be given to the pointers to avoid the DL (duplicate label) error message.

The convention that only one page is being assembled at a time will be adopted in this manual in order to simplify the examples. Thus, to call the FPP addition routine, one simply calls

```
        JMS @ FADD      /PERFORM FPP ADDITION
          .
          .
          .
FADD,   7214            /POINTER TO ADDITION SUBROUTINE
```

This causes a JMS to the subroutine located at address $7214_8$, and the subsequent addition of the FAC and FAR. The subroutine returns to the calling program with the result of the addition left in the FAC. The FAR is destroyed.

C. Loading the FAC and FAR

Since all operations take place in the FAC and FAR, it is necessary that the programmer load these registers before calling the arithmetic routines. Subroutines to load these registers are part of the FPP. To load the FAC, one calls the subroutine GETAC with the address of the first of the two words to be transferred in the location following the call. The subroutine GETAC looks at this location and takes its contents as the address of the first word to be moved into the FAC. This is illustrated by the following example, in which the floating point constant FPNUM is loaded into the floating accumulator.

```
Address  Contents          Mnemonic
2000     3000003           JMS @ GETAC    /LOAD THE FAC
2001        2004           FPNUM          /ADDRESS OF THE FIRST WORD
2002        5220           STOP           /HALT THE PROCESSOR
2003        7036  GETAC,   7036           /POINTER TO GETAC SUBROUTINE
2004        xxxx  FPNUM,   xxxx           /FP CONSTANT
2005        xxxx           xxxx
```

The above routine transfers the contents of locations 2004 and 2005 into the FAC and then halts. The first instruction, JMS @ GETAC jumps to the subroutine pointed to by GETAC, location 2003, causing an effective JMS to 7036. The subroutine at 7036 examines the location following the JMS call, location 2001, and finds the number 2004. It takes this number as the address of the first word to be moved to the FAC. It then increments this address internally and transfers the contents of 2005 to the second word of the FAC. Following this second transfer it exits to the second location following the subroutine call, location 2002, where the computer halts. At this point locations 7572 and 7573 (the actual FAC addresses) contain the same numbers as 2004 and 2005. Neither the actual number nor the pointer to it are changed by this operation. The previous contents of the FAC are destroyed.

A similar subroutine, GETAR, loads the floating argument in an exactly analogous way. A third routine, FACFAR, transfers the contents of the FAC to the FAR directly. FACFAR requires no calling addresses since its action is entirely internal to the FPP. There is no analogous routine to transfer from the FAR to the FAC, since the FAR is generally destroyed by the arithmetic operation, leaving valid information only in the FAC.

Depositing of floating point numbers in memory following such calculations is accomplished in an entirely analogous manner, with the address of the first word given in the location following the call to PUTAC. Thus one can store the contents of the FAC in the two word location TEMP by simply calling:

```
        JMS @ PUTAC     /CALL THE PUTAC ROUTINE
        TEMP            /ADDRESS OF LOCATION 1 OF TEMP
        STOP
TEMP,   ø               /ACTUAL LOCATION TEMP
        ø
PUTAC,  7050
```

It should be emphasized at this point that the surest way to programming disaster is to neglect to specify <u>two</u> locations for each floating point constant to be used by the program.

A simple routine for adding the contents of floating point numbers ANUM and BNUM is given below. The result is stored in ANUM following the operation.

```
        JMS @ GETAC     /LOAD THE FAC WITH ANUM
        ANUM            /ADDRESS OF ANUM
        JMS @ GETAR     /LOAD THE FAR WITH BNUM
        BNUM            /ADDRESS OF BNUM
        JMS @ FADD      /PERFORM FP ADDITION
        JMS @ PUTAC     /STORE IN ANUM
        ANUM
        STOP            /AND STOP
GETAC,  7036            /POINTER TO GETAC
GETAR,  7024            /POINTER TO GETAR
ANUM,   xxxx            /ACTUAL FP NUMBER ANUM
        xxxx
BNUM,   xxxx            /ACTUAL FP NUMBER BNUM
        xxxx
PUTAC,  7050
```

An additional temporary storage register TEM is available for programmer use. The subroutines FACTEM and TEMFAC move FAC to TEM and vice-versa. This register must be used with care, since the floating point input routine, FLIP, and all of the extended functions utilize TEM. The basic arithmetic routines, however, do not utilize TEM.

All of the data moving routines are summarized below:

```
GETAC    7036
GETAR    7024
FACFAR   7002
PUTAC    7050
FACTEM   7010
TEMFAC   7016
```

## D. Basic Arithmetic Routines

The following subroutines accomplish basic arithmetic functions:

| Subroutine Name | Function | Subroutine Location |
|---|---|---|
| FADD | FAC + FAR → FAC | 7214 |
| FSUB | FAC - FAR → FAC | 7255 |
| FMULT | FAC × FAR → FAC | 7350 |
| FDIV | FAC / FAR → FAC | 7413 |
| FNEG | - FAC ——→ FAC | 7263 |

In each case, the result is contained in the FAC. Negation is accomplished by taking the two's complement of the mantissa. Addition and subtraction are accomplished by shifting the FAR or FAC right until the exponents are aligned and then adding, or negating and adding. Multiplication is accomplished by multiplying the mantissas together and adding the exponents, and division by dividing the mantissas and subtracting the exponents. A complete discussion of the algorithms used is given in Part V.

The FAR is destroyed by all basic arithmetic functions except negation. It should be noted that the FAR is subtracted from the FAC during subtraction and that the FAR is the divisor during division. If the exponent becomes too large during addition or subtraction or if division by zero is attempted, the error flag is set. Exponent overflow or underflow does not, however, cause a "wrap-around" which would allow $10^{-150}$ to ever become $10^{+150}$ or vice-versa.

## E. The Error Flag

Location 7556, called ERRF, is a general purpose error flag. It is set to zero on the FPP binary tape, and is set to one by various error conditions. The error flag is never cleared, once set by any routine. It is thus up to the user to zero it at the beginning of routines and check it at the end of routines. Since ERRF is never re-zeroed, it is only necessary to check its state occasionally, such as at the end of each major section of your program rather than after each individual step. The error flag is set by such conditions as division by zero, exponent overflow, square root or logarithm of a negative number, and number out of range.

## F. Floating Point Input and Output

In order to allow the programmer to utilize the FPP most efficiently, a pair of subroutines have been designed to accept data from and output data to the Teletype. The Floating Point Input routine FLIP will accept data in virtually any format and the output routine FLOP prints data in scientific notation, with a variable number of digits, and with a character counter to allow column justification.

## 1. FLIP

This routine is called by the call JMS @ FLIP, where FLIP actually points to the input routine, located at 6712. The Teletype is then active, and will accept and echo all characters typed, until an illegal character is entered. Exit then occurs, with the terminating character in the AC, and the converted number stored in the FAC. A carriage return-line feed is not typed by FLIP and must be provided externally. The FAR is destroyed.

Since the Teletype does not have superscript capability, exponents are represented by typing E followed by the desired power of ten. Thus, 5E1 represents 50. Should an exponent larger than 150 be entered, the error flag ERRF is set = 1 upon exit.

Virtually any format is legal for input, except that spaces may not be embedded in the number. A space is detected as an illegal character and causes immediate exit from FLIP. Thus, 5.03E-6 is legal input, but 5.03 E-6 is not. Other examples of legal input include:

```
50
050
50.0
0.005E5
500E-1
+50E+0
etc.
```

Other conditions that cause FLIP to exit include:

(a) a sign anywhere except immediately before the mantissa or exponent,
(b) two decimal points in the mantissa or one in the exponent,
(c) a second E.

The above conditions are not interpreted as errors, however, and the numbers typed before they occur are converted and stored in the FAC. However, the terminating character is always in the AC upon exit from FLIP and can be examined by the calling program. One might require, for example, that the terminating character be a Return.

A mistake during entry of data to FLIP can be corrected by typing a Rubout. The typing of a Rubout causes FLIP to echo with a backslash (\) and zeroes the FAC. The entire number can then be re-entered. FLIP automatically types a backslash and zeroes the FAC if more than 11 significant figures were entered. The value can then be re-entered.

Following an FP input, the valid data flag VFLAG (6760) can be checked for validity of the input. VFLAG is always set to ø on entry to FLIP and to 1 if valid data was typed. This enables one to wait for a correct input before going on to the next program step. For instance if upon entrance to FLIP, only Q were typed, there would be an immediate exit from FLIP with 321 (ASCII Q) in the AC and VFLAG = ø.

VFLAG can be used in combination with the terminating character to determine the nature of the data entered. For example, one might wish to read in a pre-punched paper tape containing floating point numbers with a variable number of terminating characters between them, such as spaces, carriage returns, line feeds, etc. The following program will read in 10 numbers from paper tape, store them, and then halt. All illegal characters will be ignored.

```
        MEMA PNTSET     /SET BEGINNING OF STORAGE LIST INTO POINTER
        ACCM DPNT
        MEMA (12        /SET COUNTER TO 10 (BASE 8)
        ACCM COUNT
FAGN,   JMS @ FLIP      /ENTER FLOATING INPUT ROUTINE
        MMOZ @ VFLAG    /LEGAL INPUT?
        JMP FAGN        /NO, RE-ENTER SUBROUTINE
        JMS @ PUTAC     /YES, STORE THE RESULT
DPNT,   ø               /IN LOCATIONS POINTED TO BY THIS POINTER
        MPOM DPNT       /INCREMENT POINTER TWICE
        MPOM DPNT       /FOR NEXT FP NUMBER
        MMOMZ COUNT     /10 DONE YET?
        JMP FAGN        /NO, GET ANOTHER
        STOP            /YES, HALT PROCESSOR
PNTSET, 100000
COUNT,  ø
FLIP,   6712
VFLAG,  6760
PUTAC,  7050
```

2. FLOP

The floating point output routine FLOP types out the value of the FAC in scientific notation on the Teletype. If the FAC contained

```
7572/0002000
7573/1000000
```

calling JMS @ FLOP

```
        .
        .
        .
FLOP,  6543
```

would produce on the Teletype: 1.00000E0. Both the FAC and FAR are destroyed by FLOP. FLOP does not type a carriage return or line feed.

The number of significant figures typed out by FLOP is controlled by the contents of location 6544. In the form the tape is provided, FLOP types out six significant figures, and location 6544 contains 70006, equivalent to MNGA (6. To change the number of figures to 3, for example, this location would be changed to 70003, or MNGA (3.

Since the total number of characters typed by FLOP will vary with the sign of the exponent and the size of the exponent, a character counter has been included in the print routine. Each time any part of the FPP prints a character using the internal subroutine PCHAR, the counter CARCNT (6775) is incremented. It is up to the user to set and check this counter. In designing programs using this feature, it should be kept in mind that the maximum number of characters which could be produced by FLOP with six significant figures is $13_{10}$: -n.nnnnnE-nnn.

The following example causes each output from FLOP to be exactly 14 characters wide. The number of significant figures is set to 4.

```
PFLOP,  ∅               /SUBROUTINE ENTRY WITH FAC LOADED
        MEMA (16        /14 BASE TEN
        ANGM @ CARCNT   /SET CHARACTER COUNTER TO -14
        JMS @ FLOP
PAGN,   MEMZ @ CARCNT   /IS CHARACTER COUNTER = ∅?
        ZERZ            /NO, SKIP EXIT INSTRUCTION
        JMP @ PFLOP     /YES, EXIT FROM SUBROUTINE
        MEMA (240       /PRINT SPACES TO FILL TO 14
        JMS @ PCHAR     /PRINT ROUTINE IN FPP INCLUDES CARCNT
                        /INCREMENT
        JMP PAGN        /LOOP UNTIL 14 CHARACTERS TYPED
FLOP,   6543
CARCNT, 6775
PCHAR,  6767
        *6544
        MNGA (4         /SETS 4 SIGNIFICANT FIGURE OUTPUT
```

### 3. FIXOP

Data printed in the form n.nnnE-nn are sometimes confusing to read for particular applications. While it should be recognized that the only way one can maintain the same number of significant figures in numbers throughout the entire numerical range handled by FPP-1972, is in scientific notation, it is entirely possible for numbers whose dynamic range is well known, that a fixed decimal point output may be desirable.

FPP-1972 allows output in a format where the decimal point is always located in the same place regardless of the size of the number using the routine FIXOP. The format of this output is specified by the two locations NUMD and PREDIG. NUMD is the total number of digits to be printed and PREDIG is the number of digits to be printed before the decimal point. To print out ten digits, four before the decimal point and six after it, one sets PREDIG to 4 and NUMD to $12_8$. These numbers remain set in these locations and this format will be used for all further calls to FIXOP, until these numbers are specifically changed. The following code would accomplish this:

```
        MEMA (4
        ACCM @ PREDIG    /SET # BEFORE DECIMAL POINT
        MEMA (12
        ACCM @ NUMD      /SET TOTAL NUMBER OF DIGITS
        JMS @ FIXOP      /PRINT NUMBER IN FLOATING ACCUMULATOR
        .
        .
        .
PREDIG, 7554
NUMD,   7553
FIXOP,  7524
```

The fixed point output format suppresses leading zeroes in numbers smaller than the total space left of the decimal point, and fills with zeroes to the right to the end of the format space. Numbers whose value is smaller than that allowed by the format are printed as zero, and numbers whose value is larger are printed as XXXX.XX. If NUMD is set equal to PREDIG, so that all the digits are to the left of the decimal point, the decimal point itself is not printed.

## G. Conversion to Floating Point Format

The subroutine FLOAT converts a fixed point integer in the FAC to floating point format, leaving the converted result in the FAC. FLOAT operates on signed integers or signed fractions with a fixed binary point. It considers the two locations FACM and FACML (7573 and 7574) of the expanded FAC to be a 40-bit number with the binary point located between the two words. The contents of the floating exponent word FACE (7572) are unimportant on entry. On exit, the result is left in the FAC in standard floating point format.

Thus, to float a standard 20-bit integer, such as might be found in signal averaged data, one must be sure to zero FACML before calling FLOAT. The following subroutine accomplishes this flotation, assuming the integer is in the AC on entry:

```
FLOTIT,  ø              /SUBROUTINE TO FLOAT 20-BIT INTEGERS
         ACCM @ FACM    /AC CONTAINED INTEGER ON ENTRY, STORE IN FACM
         ZERM @ FACML   /ZERO LOW ORDER FAC
         JMS @ FLOAT    /PERFORM THE FLOAT
         JMP @ FLOTIT   /AND EXIT FROM THE SUBROUTINE
FACM,    7573
FACML,   7574
FLOAT,   7534
```

## H. Conversion of Floating Point Numbers to Fixed Point

The subroutine FIX converts the floating point number found in FAC to a fixed point number whose binary point lies between FACM and FACML. Since converting to integer format requires that the exponent be decremented until it reaches zero, FACE will be zero upon exit if the FIX was successful. If the FP number was too large to FIX, FACE will be non-zero. If the number was too small to FIX, FACM-FACML will be all zeroes if the sign was positive and all ones if the sign was negative.

## I. Exercises

1. Without consulting the FPP-1972 listing, write a routine that operates in the same fashion as GETAC.

2. Convert the following floating point numbers to integers:

| | | |
|---|---|---|
| 0002000 | 0000000 | 0002000 |
| 1000000 | 1400000 | 1400000 |

3. For values X, M and B stored in memory, write a program to calculate

$$Y = MX + B$$

and store the result Y in memory.

4. Write a program to calculate and print out Y in the equation:

$$Y = AX^2 + BX + C$$

for X entered at the Teletype and A, B and C stored in memory.

5. Rewrite problem 4 so that A, B, C and X are entered at the Teletype. The program should type A=, B=, C= and X= and allow values to be entered. The calculation should be performed and the resulting Y printed out.

## III. THE EXTENDED FUNCTIONS

### A. Summary of the Extended Functions

The Floating Point package may be logically divided into two sections: the basic arithmetic section, and the extended functions. While the extended functions utilize the basic arithmetic section, the basic arithmetic section stands by itself. In fact, if additional program storage space is needed, and the extended functions are not used by that program, one can overwrite the extended functions section, from 6000 - 6464.

The complete list of extended functions is given below:

| Subroutine Name | Function | Location |
|---|---|---|
| FSIN | $\sin(FAC) \longrightarrow FAC$ | 6000 |
| FCOS | $\cos(FAC) \longrightarrow FAC$ | 6114 |
| FARCTN | $\arctan(FAC) \rightarrow FAC$ | 6122 |
| FRIP | $1/FAC \longrightarrow FAC$ | 6170 |
| FSQRT | $(FAC)^{1/2} \longrightarrow FAC$ | 6176 |
| FLOG | $\log(FAC) \longrightarrow FAC$ | 6311 |
| FLN | $\ln(FAC) \longrightarrow FAC$ | 6317 |
| FSQAR | $(FAC)^2 \longrightarrow FAC$ | 6333 |
| FEXP | $10^{FAC} \longrightarrow FAC$ | 6337 |
| FEXPN | $e^{FAC} \longrightarrow FAC$ | 6345 |

In each case, the result of the calculation is placed in the FAC. If the calculation is not possible, the error flag ERRF is set, and the result is meaningless.

### B. Square, Square Root and Reciprocal Functions

FSQAR, FSQRT and FRIP all maintain 30 bits of accuracy. If the squaring of a number causes exponential overflow, the error flag will be set. If FAC is negative, the error flag will be set, and the square root is taken of the absolute value of the FAC. Any attempt to take the reciprocal of zero will also set the error flag. In this last case the FAC will be meaningless.

### C. Sine, Cosine and Arctangent

FSIN, FCOS and FARCTN all maintain at least 26-bit accuracy. The decrease in accuracy is a result of the successive approximation methods employed. There are no error conditions.

The argument presented to FSIN and FCOS must be in units of $\pi/2$ radians. This is a convenient unit to work with since four such units make a circle. One represents an angle such as 45° by 0.5, for example. Similarly, FARCTN produces a result in units of $\pi/2$ radians.

## D. FLOG, FLOGN, FEXP and FEXPN

FLOG, FLOGN, FEXP and FEXPN all maintain at least 26 bits of accuracy. An attempt to exponentiate too large a number will cause the error flag to be set. This number is about 150 for FEXP and about 350 for FEXPN. Any attempt to compute the logarithm of a negative number or of zero will cause the error flag to be set. No operation is performed on FAC in that case.

## E. Extended Functions Programming Example

The extreme ease with which the extended functions can be used to perform complex calculations is shown by the following example which calculates $\exp(1/x^2)$.

```
JMS @ GETAC      /GET X FROM MEMORY
X
JMS @ FSQAR      /X**2
JMS @ FRIP       /1/X**2
JMS @ FEXPN      /EXP(1/X**2)
STOP
```

## F. Exercises

1. Write a program to calculate the sine of a number typed on the Teletype in degrees. The result should be printed on the same line.

2. Write a program to evaluate

$$Y = \frac{-B \pm (B^2 - 4AC)^{1/2}}{2A}$$

where A, B and C are entered at the Teletype. If an illegal operation should occur, the program should discover it and print a ?.

3. Write a program to calculate and display a semicircle on the oscilloscope. As each point is calculated, it should be displayed. When the calculation is complete, the entire semicircle should be displayed.

## IV. ADVANCED PROGRAMMING CONCEPTS

### A. Testing the Terminating Character of FLIP

The first illegal character encountered by the floating point input routine causes an immediate exit, with that ASCII character remaining in the AC. This feature can be used to determine how the input is to be converted. In the following example, FLIP is used to accept numbers assumed to be in the units of $\pi/2$ radians. The terminating character is then either S or C, which implies that the program is to compute the sine or cosine of the entered number, and print it on the Teletype.

```
START,   JMS @ FLIP      /ENTER FLOATING INPUT ROUTINE, GET ARG IN
                         /PI/2 UNITS
         A-MZ (323       /WAS TERMINATING CHARACTER "S"?
         JMP CTEST       /NO, TEST FOR C
         JMS @ FSIN      /YES, CONVERT TO SINE
OUTPUT,  MEMA (275       /PRINT EQUALS SIGN
         JMS @ PCHAR     /USING FFP PRINT ROUTINE
         JMS @ FLOP      /PRINT CONVERTED SINE OR COSINE
         JMS CRLF        /PRINT CARRIAGE RETURN-LINE FEED; ROUTINE
                         /NOT SHOWN
         JMP START       /AND GET NEW INPUT VALUE
CTEST,   A-MZ (303       /WAS CHARACTER "C"?
         STOP            /NO, ERROR, HALT PROCESSOR
         JMS @ FCOS      /YES, CONVERT TO COSINE
         JMP OUTPUT      /AND PRINT VALUE ON TTY
FLIP,    6712            /POINTERS TO FLOATING POINT SUBROUTINES
FLOP,    6543
FSIN,    6000
FCOS,    6114
PCHAR,   6767
```

### B. Reading and Printing Characters

The subroutines RCHAR and PCHAR read and print characters on the Teletype. RCHAR reads a character from the Teletype and then calls PCHAR to print it. It is therefore not necessary, in general, to write one's own read and print subroutines. Should the user decide to write similar routines for other memory pages, it is necessary that they have the same timing structure as those in the FPP. RCHAR and PCHAR are both structured in the sense: wait for the flag and skip, jump back, then read or print:

```
T1,  TTYRF      P1,  TTYPF
     JMP T1          JMP P1
     RDTTY           PRTTY
```

It is possible, of course, to write routines in the order:

```
     PRTTY
T2,  TTYPF
     JMP T2
```

so that the program waits for the flag to go up before continuing. This second method can not be used with the FPP, since it would cause timing errors between the user's subroutine and the FPP subroutine.

### C. Multiplication and Division by Two

In fairly long calculations, it becomes apparent that Floating Point calculations are significantly slower than integer calculations. It is therefore desirable to avoid the slower method whenever a faster one is available. Multiplication by 2 can be relatively time consuming if carried out in floating point, for example, but is easily accomplished in fixed point. Since the exponent of a floating point number is a power of two, simply incrementing the exponent by 1 will accomplish this multiplication. However, the exponent occupies the left hand ten bits of an FP word, and therefore the addition must be done to the exponent alone, by adding $2000_8$ to the first word. The following sequence of code multiplies the FAC by 2:

```
MPOA (1777       /GET THE CONSTANT 2000 INTO THE AC
A+MM @ FACE      /AND ADD INTO THE EXPONENT
```

Similarly, division by 2 can be accomplished by subtracting $2000_8$ from FACE:

```
MPOA (1777
M-AM @ FACE      /SUBTRACT 2000 FROM FACE
```

### D. Testing the FAC for Zero

After any operation, one can test the FAC to see if it has become zero by examining FACM. While the exponent may still have some non-zero value, the mantissa will be zero if and only if the FP number is zero. One can therefore test for a zero input from FLIP by the following code:

```
FLOOP,  JMS @ FLIP       /FLOATING INPUT ROUTINE
        MEMZ @ FACM      /ZERO INPUT?
        ZERZ             /NO, INPUT OK
        JMP FLOOP        /YES, GET NEW INPUT
```

It should be emphasized however, that it is poor programming practice, just as in high-level languages, to assume that any two floating point numbers will ever become exactly equal. If one wishes to find out whether a number has reached a value of 1.9, he cannot assume that subtraction of 1.9 from that number will produce exactly zero. The actual result of such a subtraction may well be $10^{-9}$ or so, but will be finite and non-zero. This is simply because the internal representation of some numbers is not exactly the same in base 2 as in base 10. It also could be because a calculated number may be somewhat different than a floated integer. The usual procedure in this case is to subtract the two numbers and determine whether their difference is less than some tolerance, such as $10^{-6}$.

### E. Skipping on Positive or Negative FAC

Since the sign bit of FAC is readily available in bit 19 of FACM, it is quite possible to perform a simple calculation and then allow the program to branch depending on the sign of the result. If this procedure is to be carried out a number of times in a program, it is advantageous to use a branching subroutine like that shown below. The subroutine is entered with FAC and FAR loaded with the two numbers to be compared. It will produce a skip if the result after subtraction is positive.

```
SKIP+,  ø                /ENTER WITH FAC AND FAR LOADED
        JMS @ FSUB       /PERFORM THE SUBTRACTION
        MEMA @ FACM      /TEST SIGN OF MANTISSA
        EXCT AC19        /IS THE SIGN NEGATIVE?
        JMP @ SKIP+      /YES, EXIT WITHOUT SKIPPING
        MPOM SKIP+       /NO, INCREMENT EXIT POINTER
        JMP @ SKIP+      /AND EXIT WITH SKIP OF NEXT INSTRUCTION
```

### F. Determining of Floating Point Constants

The following constants have been converted into packed two word floating point format for general use:

| Constant | Octal Value | Decimal Value |
|---|---|---|
| π | 0005526<br>1444176 | 3.1415926 |
| π/2 | 0003526<br>1444176 | 1.5707963 |
| e | 0005212<br>1267702 | 2.7182818 |
| 10.0 | 0010000<br>1200000 | |
| 1.0 | 0002000<br>1000000 | |

It is very easy to generate constants in floating point format by using the subroutine call instructions contained in Nicobug II. If you wish to find out the floating point equivalent of 355.29, for example, you need only call the floating input routine from Nicobug II and type in the number 355.29 followed by a Return. Then examination of the FAC, locations 7572 and 7573 will show the answer in floating point format. This is illustrated below:

| | |
|---|---|
| 6712S355.29 | The command 6712S calls the subroutine FLIP at 6712. The constant 355.29 is entered and a Return typed. |
| 7572/0023727<br>7573/1306450 | The contents of the FAC, 7572 and 7573, are examined using Nicobug II. |

Conversely, to determine the decimal value of any floating point number, simply enter it in the FAC and call the floating output routine using Nicobug II. Below we determine the value of Floating Point Constant 0024702 - 1246775:

| | |
|---|---|
| 7572/0000000 24702 | FACE opened and 24702 entered; Line Feed deposits this number and opens 7573 |
| 7573/0000000 1246775 | The value of FACM is changed to 1246775, and Return typed. |
| 6543S 6.78995E2 | The subroutine FLOP at 6543 is called. The contents of the FAC in decimal is $6.78995 \times 10^2$ or 678.995. |

## G. Roundoff and Overflow

There has been a great deal of discussion among programmers about roundoff problems. The magnitude of the problem is illustrated by the following experiment. Ask several computers in several languages to add pairs of numbers, like .1 and 1.9, and take the integer part. The answers will undoubtedly differ somewhat from machine to machine.

This problem arises partly because computers are binary machines. A number that is simple to represent in decimal, such as 0.1, is impossible to represent exactly in binary. The internal representation of 0.1 may be high (or low) by an amount not greater than one part in one billion in the case of the FPP. The most accurate decimal representation of the internal representation may well be .099999999. Knowing this makes it easy to see how the integer part of (1.9 + 0.1) can be one.

The appearance of a number like .0999999 is something of a surprise when one expects 0.1. The FPP solves this problem of aesthetics by adding approximately one part per billion to a number before printing it. The effect on the sixth digit of the

mantissa is almost always invisible. The feature can be removed or the amount of roundoff changed by varying the roundoff constant shown in the listing.

As mentioned earlier, all arithmetic operations produce 40-bit mantissas which are truncated to 30 bits. Before truncation these 10 bits are examined. If the most significant bit is a one, the 30 bit final mantissa is incremented. If this were not done, all arithmetic operations would produce answers systematically too small by an amount averaging .5 parts per billion.

## H. Exercises

1. Using Nicobug II, calculate the floating point values of the following numbers:

$8.6 \times 10^{-3}$ 50002 91

2. Write a program to calculate the integer value of Y in Y = 2X+1 where X is entered at the Teletype using FLIP. If overflow occurs a ? should be typed. Otherwise, the calculated value should be printed out. If an illegal input to FLIP is detected, a ? should be printed.

3. Write a subroutine to skip on a negative result after subtracting B from A. The program should be called with A in the FAC and the address of B in the location following the call:

```
JMS SKIPM       /A IN FAC
B               /B FOLLOWS CALL
....            /RETURN HERE IF RESULT +
....            /RETURN HERE IF RESULT -
```

TABLE I

POINTERS TO FLOATING POINT, 1972 NIC-80/S-7209

| | | |
|---|---|---|
| FADD | fac + far → fac | 7214 |
| FSUB | fac - far → fac | 7255 |
| FNEG | - fac → fac | 7263 |
| FMULT | fac × far → fac | 7350 |
| FDIV | fac / far → fac | 7413 |
| FIXOP | fixed point output | 7524 |
| FLOP | floating output | 6543 |
| FLIP | floating input | 6712 |
| PCHAR | prints character | 6767 |
| RCHAR | reads & prints char. | 6761 |
| GETAC | x → fac | 7036 |
| GETAR | x → far | 7024 |
| FACFAR | fac → far | 7002 |
| PUTAC | fac → x | 7050 |
| FACTEM | fac → tem | 7010 |
| TEMFAC | tem → fac | 7016 |
| FLOAT | floats facm-facml | 7466 |
| FIX | fixes fac | 7473 |
| FSIN | sin(fac) → fac | 6000 |
| FCOS | cos(fac) → fac | 6114 |
| FARCTN | arctan(fac) → fac | 6122 |
| FRIP | 1/fac → fac | 6170 |
| FSQRT | $fac^{1/2}$ → fac | 6176 |
| FLOG | log(fac) → fac | 6311 |
| FLOGN | ln(fac) → fac | 6317 |
| FEXP | $10^{fac}$ → fac | 6337 |
| FEXPN | $e^{fac}$ → fac | 6345 |
| FSQAR | $fac^2$ → fac | 6333 |

| | | |
|---|---|---|
| NUMD | = | 7553 |
| PREDIG | = | 7554 |
| ERRF | = | 7556 |
| CARCNT | = | 6775 |
| VFLAG | = | 6760 |
| FACE | = | 7572 |
| FACM | = | 7573 |
| FARE | = | 7575 |
| FARM | = | 7576 |

TABLE II

FLOATING POINT PACKAGE SUMMARY

| Operation | Mnemonic | Units | Accuracy | Speed* | Error Conditions | Registers Destroyed | |
|---|---|---|---|---|---|---|---|
| Multiplication & Division | FMULT, FDIV | | 30 bits | .8 ms | Exponent overflow, Zero division | FAR | |
| Addition & Subtraction | FADD, FSUB | | 30 bits | .8 ms | None | FAR | |
| Square | FSQAR | | 30 bits | .8 ms | Exponent overflow | FAR | |
| Square root | FSQRT | | 30 bits | 6 ms | Negative argument | FAR | TEM |
| Reciprocal | FRIP | | 30 bits | .8 ms | Zero argument | FAR | |
| Sine | FSIN | $\pi/2$ radians (input) | 26 bits | 12 ms | None | FAR | TEM |
| Cosine | FCOS | $\pi/2$ radians (input) | 26 bits | 12 ms | None | FAR | TEM |
| Arctangent | FARCTN | $\pi/2$ radians (output) | 26 bits | 31 ms | None | FAR | TEM |
| Logarithm, base ten | FLOG | | 26 bits | 12 ms | Negative or zero argument | FAR | TEM |
| Logarithm, base e | FLOGN | | 26 bits | 12 ms | Negative or zero argument | FAR | TEM |
| Exponentiate, base ten | FEXP | | 26 bits | 20 ms | Zero argument<br>Argument > 150 | FAR | TEM |
| Exponentiate, base e | FEXPN | | 26 bits | 20 ms | Zero argument<br>Argument > 350 | FAR | TEM |

*Highly data dependent; as rough guide only

## V. ALGORITHMS USED IN THE EXTENDED FUNCTIONS

### A. Introduction

This is a description of the algorithms used to compute the extended functions. Most of the functions are computed by methods described by Cecil Hastings in his book Approximations for Digital Computers (Princeton University Press, 1955).

### B. Square Root

First a guess is made by dividing the exponent of the argument by 2. Then the guess is refined by setting it equal to:

$$\text{New Guess} = \left(\frac{\text{Old Guess} + \text{Argument}}{2^* \text{ Old Guess}}\right)$$

Then the process is repeated 5 times.

### C. Sine

First the argument is "rotated" into the first quadrant by adding or subtracting ones. The following identities are used:

$$\text{sine } (-x) = -\sin(x)$$
$$\text{sine } (1+x) = \text{sine } (1-x)$$

Then the following Taylor series polynomial is evaluated:

$$\sin x = \sum_{i=0}^{n} C_{2i+1} x^{2i+1}$$

where n = 4. The values for C are

$$C_1 = .157080 \times 10^1$$
$$C_3 = -.645964 \times 10^0$$
$$C_5 = .796897 \times 10^{-1}$$
$$C_7 = -.467377 \times 10^{-2}$$
$$C_9 = .151484 \times 10^{-3}$$

### D. Cosine

The cosine function is evaluated with the sine subroutine with the aid of the identity:

$$\cos(x) = \sin (1+x)$$

E. Arctangent

If the argument is $\leq 1$, then

$$\text{arctangent } x = \sum_{i=0}^{n} C_{2i+1} x^{2i+1}$$

Otherwise:

$$\text{arctangent } x = 1 - \sum_{i=0}^{n} C_{2i+1} \left(\frac{1}{x}\right)^{2i+1}$$

where n = 7.

The values of C are

$C_1 = 0.636619347$
$C_3 = -0.212184453$
$C_5 = 0.126983591$
$C_7 = -0.08854474$
$C_9 = 0.061382906$
$C_{11} = -0.035503338$
$C_{13} = 0.013917289$
$C_{15} = -0.002580893$

F. Logarithm Base ten and Base e

The logarithm to the base 2 is computed and the final result is determined from the fact that

$$\log_e x = (\log_2 x)(\log_e 2) \text{ for base e}$$

$$\log_{10} x = (\log_2 x)(\log_{10} 2) \text{ for base ten}$$

The log to base 2 is calculated as follows:

(1) If the argument is $\leq 0$, the error flag is set and the logarithm subroutine exits.
(2) If the argument is $< 1$, the end result is negated.
(3) If the argument is $\geq 1$, the reciprocal of the argument is taken.
(4) The original exponent is saved and the exponent of FAC is set to 1. Thus $1 < \text{FAC} \leq 2$.
(5) Z is computed from the equation

$$Z = \frac{x-\sqrt{2}}{x+\sqrt{2}}$$

(6) Then

$$\text{Log}_2 x = -1/2 + \sum_{i=0}^{n} C_{2i+1} Z^{2i+1}$$

is computed for n = 2. The values of C are:

$C_1 = .288539 \times 10^1$
$C_3 = .961471 \times 10^0$
$C_5 = .598979 \times 10^0$

(7) The exponent is retrieved, converted to a floating number, and added to FAC.
(8) FAC is negated if necessary and multiplied by the proper constant.

G. Exponentiation, Base ten and Base e

FAC is multiplied by a constant so that the internal base 2 exponentiation subroutine can be used.

$e^x = 2^{x \log_2 e}$ for base e

$10^x = 2^{x \log_2 10}$ for base ten

If FAC is negative the absolute value is taken and the final answer is the reciprocal.

Then FAC is separated into a fractional part and an integer part by subtracting ones. The fractional part, F, is evaluated.

$$2^F \times 1 + \frac{2F}{A - F + BF^2 - \frac{C}{D+F2}}$$

where the constants are:

A = +9.95459578
B = +0.03465735903
C = +617.97226053
D = +87.417497202

Now the integer part is converted into a fixed point number and added to the exponent of FAC.

Section VI, Listing of Basic Arithmetic and Section VII, Listing of Extended Functions are included as a part of the 1080 Instruction Manual and are available upon request.

# HARDWARE ARITHMETIC TEST - INSTRUCTIONS FOR USE
# NUS-204

Load the binary tape and begin executing at location zero. If all is well, the following messages should be typed:

TRANSFER OK
BIT-INVERT OK
MULTIPLY OK
DIVIDE OK

If an error occurs, one of the following messages will be typed:

1) TRANSFER FAILED
SOFTWARE XXXXXXX HARDWARE XXXXXXX
This signifies that either a TACMQ or a TMQAC failed in its operation. The Software # is what the program tried to load into the MQ and the Hardware # is what came out.

2) BIT-INVERT FAILED
SOFTWARE XXXXXXX HARDWARE XXXXXXX UNINVERTED XXXXXXX
This signifies an error in the bit interchanges; 19-Ø, 18-1, etc. The Software number is the simulated software result, the Hardware number is the hardware result and the Uninverted number is the initial operand.

3) NO SKIP
The contents of memory location following a MULT or DIVIDE instruction should be executed. This error message signifies failure of the computer to skip execution of these memory locations.

4) MULTIPLY FAILED
SOFTWARE XXXXXXX * XXXXXXX = XXXXXXX, XXXXXXX
HARDWARE XXXXXXX * XXXXXXX = XXXXXXX, XXXXXXX
This signifies a failure in the operation of the MULT instruction. If the second Software operand differs from the Hardware operand it means the hardware failed to restore it to memory. If the Hardware product is greater than the Software product by one, then the hardware failed to clear the accumulator prior to multiplication. All other differences signify other hardware failures.

5) DIVIDE FAILED
SOFTWARE XXXXXXX, XXXXXXX / XXXXXXX = XXXXXXX R XXXXXXX
HARDWARE XXXXXXX, XXXXXXX / XXXXXXX = XXXXXXX R XXXXXXX
This signifies a hardware failure in the DIVIDE instruction.

# ASTROTEST - INSTRUCTIONS FOR USE
NUS-248


## Abstract

Astrotest is a simple, minimum length program for testing, and diagnosing faults in core memory.


## Loading

Astrotest starts at location $\emptyset$ and is only $150_8$ instructions long. The block of memory to be tested is determined by two memory locations which must be accessed through the 291.

| | |
|---|---|
| Location 46 | size of block |
| Location 47 | first address of block |

## Theory of Operation

Each location in the block of memory to be tested is set to a known state. Then the block of memory is read out and a comparison is made to determine if there was an error. The states are: 0, 3777777, 1, 2, 10, 20, 40, 100, 200, 400, 1000, 2000, 4000, 10,000, 20,000, 40,000, 100,000, and 200,000 followed by the compliment of these states. The entire sequence is repeated 40 times after which the program restarts itself.

If an error is detected, astrotest will print the address, the state the location should be set to, and the erred state. It does not restart after an error, but rather it continues from where the error was found.


Submitted By: Jack Kisslinger
Astrodigit