A METHOD FOR APPLYING SCIENTIFIC SUBROUTINE PACKAGE IN MICROPROCESSOR

CONTENTS PAGE

A METHOD FOR APPLYING SCIENTIFIC SUBROUTINE PACKAGE IN MICROPROCESSORS

INTRODUCTION

The scientific subroutine package is one of the most important parts of the software for the scientific industry. By now, most big computers have scientific packages, but applying such a software package in microprocessors requires consideration of the microprocessor's facilities, such as limited main memory, slow execution time, and only a few small registers. In any scientific package, the trigonometric functions are the ones more widely used.

This paper discusses a method for implementing several trigonometric function programs in a scientific package in microprocessors. These programs will contain routines for computing sin, cos, tan, and cot of any angle within the range of (-360^o, $+360^o$).

The paper will also include the discussion of several approaches to computing trigonometric functions used in different computer systems.

FLOATING POINT IN MICROPROCESSORS

Most of the microprocessors have 8 bit registers which are inadequate for scientific applications. Sixteen or even 32 bit fixed point calculations should be used for greater accuracy. However, these techniques are still inherently inadequate for calculations performed over a wide range of numbers. If one could dynamically slide the radix point, the number range would be dramatically increased. This is made possible by the use of floating point representation. By using floating point formats, we increase the range of numbers and obtain better accuracy.[2]

There are many ways to represent floating point numbers, but three formats are more common. The first one is hexadecimal form with binary representation; the second one is in binary form. The third format uses a binary coded decimal (BCD) representation.

One format using a base of 16 is used in the IBM/360, as shown in Figure 1. This format consists of a sign

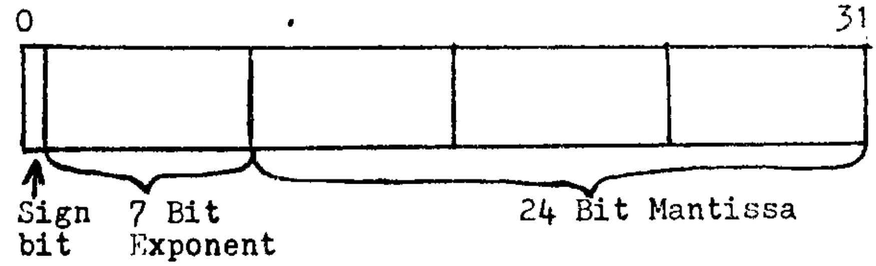


Figure 1: IBM/360 Floating Point Format.

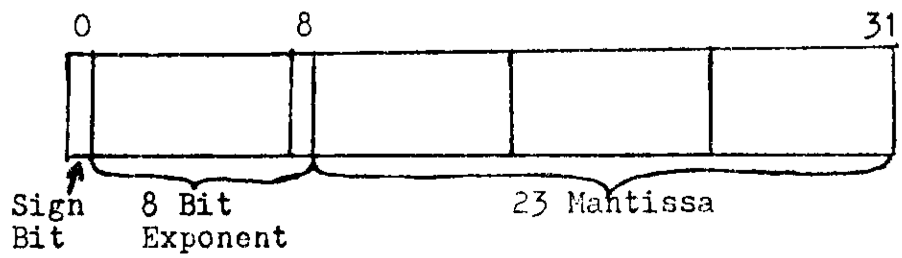


Figure 2: A Binary Floating Point Format Used by Digital Equipment Corporation and Hewlett Packard.

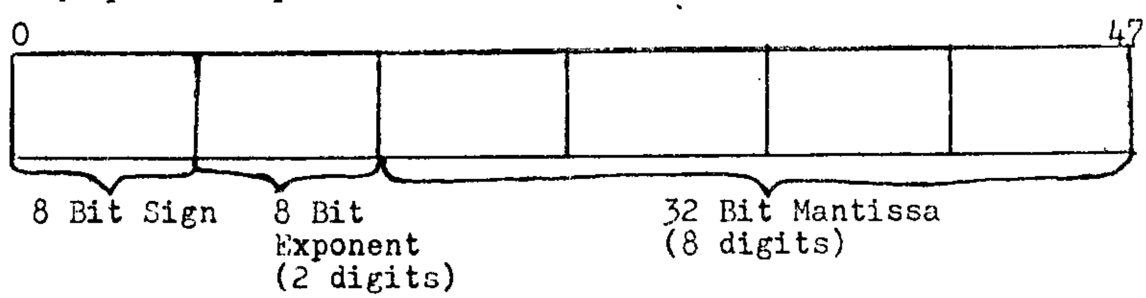


Figure 3: A BCD Floating Point Format.

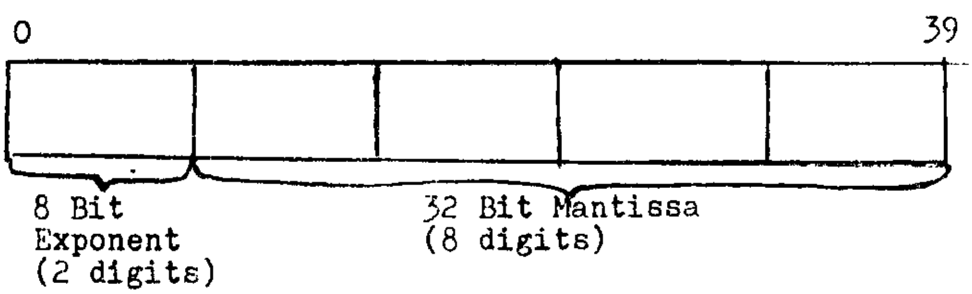


Figure 4: Applied Floating Format.

bit for fraction, a 7 bit exponent, and 24 bit mantissa. The radix point is to the left of the most significant digit. The exponent sign is inherent. The sign bit is zero if the number is positive, otherwise it is 1.

Another format, as shown in Figure 2, is a binary format, which is used by Digital Equipment Corporation (DEC) and Hewlett-Packard in their BASIC interpreters. In this format, the most significant bit is always one, a normalized number, unless the entire number is zero. The sign of a number is shown in the sign bit. The exponent is stored in 8 bits and represents a power of two in excess-128 notation. The range of this format is from 2^{+126} to 2^{-128}, or approximately 10^{+38} to 10^{-38}, with 7 decimal digit accuracy.1

There are several kinds of BCD (binary coded decimal) floating point formats currently in use. The range of the mantissa in this format can be as few as four digits to as many as sixteen digits of accuracy, and the exponent can be 10^{-99} to 10^{+99} or even 10^{+127} to 10^{-127}. 6 The popular format, as shown in Figure 3, has 8 bits (2 digits) for the exponent, and an 8 digit mantissa with the decimal point assumed to be to the left of the most significant digit. The sign of the number is represented by a whole byte, 00 for positive and FF for negative. The exponent can be represented in the form of excess-128 notation, similar to DEC.

In floating point computations, the use of guard digits is well established.[2] A Guard bit or byte is used in floating point registers to maintain accuracy in performing the calculation. The guard byte is a <u>8</u> bit extention to the least significant byte of the fraction, mantissa. By using guard digits, significance will not be lost when round off occurs.

<u>Applied Floating Point Format</u>

Each of the above floating point formats has its own particular advantages and disadvantages. Which is best depends upon the requirements of the particular application: speed, small memory size, variable mantissa length, ease of interfacing to other software routines, etc.

The floating point format used in this application, as shown in Figure 4, has the following BCD format: The

The exponent is two digit decimal in excess-50 notation, e.g. a zero exponent (10^0) is 50, and the maximum exponent (10^{49}) is 99. The mantissa is normalized. The radix point is assumed to be to the left of the fraction. Thus the most significant digit is not zero. However, the mantissa is represented by the 8 digit number, 32 bits. Therefore, the number has 8 digit accuracy. Whenever the number is negative, it will be represented by the ten's complement of the original number. If the most significant digit of the mantissa is o through 4, then the number is positive; otherwise, the number is negative.

Numbers from $+0.49999999 \times 10^{49}$ to $-0.49999999 \times 10^{-50}$ can be represented by this format. A "guard byte" is used in the floating point registers to maintain accuracy in performing the calculations.

Since the applied floating point form is a kind of BCD format, it has the same advantage which BCD does:[6] it is easy to compute. The applied format is the shortest length for BCD format, because the sign of the represented number is inherent and there is no need to represent the sign in one byte. However, it is easy to convert numbers from ASCII to BCD and vice-versa.[6]

This format has some disadvantages, but the commercial computer industry has adapted to it. One of the disadvantages is that the execution times for this format are significantly slower than the binary floating points.[1]

TRIGONOMETRIC FUNCTIONS

There are several kinds of methods for computing trigonometric functions, such as Table searching, Taylor's expansion, and Chebychev polynomials.

In regard to the microcomputer facilities (such as small size of memory, 8 bit accumulator, and relatively slow Computation speed), most of these methods are inefficient.

In order to use the table searching method, a relatively large amount of the main memory is devoted to the table of the required angles.[7] As a matter of fact, several program routines should be used for searching the table. Since in microprocessors we are dealing with a small size main memory, applying this method for computing trigonomtric functions would not be efficient.

From Taylor's expansion, we know that, for example, the value of Sin(x) can be computed as

$$\mathrm{Sin}(x) \cong X - \frac{x^3}{3!} + \frac{x^5}{5!} \;\ldots\ldots\; + (-1)^{m+1} \frac{x^{2m-1}}{(2m-1)!} + \ldots$$

This approximation for Sin has the error as remainder

$$R = E < \left| \frac{x^{2n+1}}{(2n+1)!} \right|$$

In the remainder term, as the range of x increases, it is necessary to include more and more terms in the series in order to obtain any desired accuracy. For getting a relatively small error bound for the approximating formula, at least ten terms of the equation should be used. Computing these terms in microprocessors causes relatively large round-off errors and also requires a large amount of main memory. However, the execution time for computation would be slow.[7]

The IBM system library uses Chebychev polynomials for computing the value of trigonometric functions. In this algorithm, the main part of computation is finding the value of Sin ($\pi/4 \cdot r_1$) or Cos ($\pi/4 \cdot r_1$) where r_1 is within the range of $0 \leq r_1 \leq 1$.; for computing Sin ($\pi/4 \cdot r_1$) the following polynomial is used:

$$\sin(\pi/4 \cdot r_1) = r_1 (a_0 + a_1 r_1^2 + a_2 r_1^3 + a_3 r_1^6)$$

The coefficients were obtained by the roots of the Chebychev polynomials of degree 4. The relative error in this method is less than 2^{-28} for the range of $-\pi/2 \leq x \leq +\pi/4$

If Cos ($\pi/4 \cdot r_1$) is needed, it is computed by a polynomial of the following form:

$$\cos(\pi/4 \cdot r_1) = 1 + b_1 r_1^2 + b_2 r_1^4 + b_3 r_1^6$$

Coefficients were obtained by a variation of the minimax approximation, which provides a partial rounding for short precision computation. The error of this approximation is less than 2^{-24}.

However, as the value of x increases, the relative error would increase too, and no consistent relative error control can be maintained outside the principal range $-\pi/2 \leq x \leq +\pi/4$.

As we see, applying the Chebychev polynomial for computing the trignometric functions in microprocessors has almost the same disadvantages that the Tayor expansion does.[8]

Trigonometric Functions on the M6800

For implementing any method to compute the trigonometric functions on the M6800, three facts should be considered. The method should be accurate, easy to execute and efficient in its use of main memory.

One of the methods for computing the trigonometric functions which is recomended by several text books is approximating polynomails in the terms of finite differences. During the research, it was found out this method would be more efficient, more accurate, and faster than the others.[3,9]

Although this method uses the approximating formula for computing, the execution time is not very long, moreover, the execution time for microprocessors is not as important as occupied main memory. In regard to the other techniques for approximating polynomials, the finite difference algorithm has less arithmetic computation, so the execution time would be faster.

DESCRIPTION OF THE APPROXIMATING FORMULA

The problem is to obtain a function g(x), which approximates a given function, f(x) at a certain number of specified points. One of the important requirements for g(x) is simplicity of evaluation; a polynomial will certainly fit this requirement.

The approximating polynomial, in terms of finite differences, Gregory-Newton Formula, is one of the formulations which is not essentially difficult for digital computers.[3]

The applied algorithm for finding trigonometric functions is the third degree polynomial given by:

$$P_3\ (x) = Y_0 + \mu \Delta Y_0 + \Delta^2 Y \frac{(\mu-1)}{2!} + \Delta^3 Y \frac{(\mu-1)(\mu-2)}{3!}$$

Where X is a point in the interval (x_0, x_3). x_0, x_1, x_2 and x_3 are 4 points of the function; $\mu = (x-x_0)/h$, and

$h = x_1-x_0 = x_2-x_1 = x_3-x_2$.

ΔY_0, $\Delta^2 Y$ and $\Delta^3 Y$ are defined in the following table.

X	Y	ΔY_0	$\Delta^2 Y$	$\Delta^3 Y$
X_0	Y_0			
		Y_1-Y_0		
X_1	Y_1		$\Delta Y_1 - \Delta Y_0$	
		Y_2-Y_1		$\Delta^2 Y_1 - \Delta^2 Y_0$
X_2	Y_2		$\Delta Y_2 - \Delta Y_1$	
X_3	Y_3	Y_3-Y_2		

For the best accuracy, the four nearest points to any given value, x, should be used in the above formula.

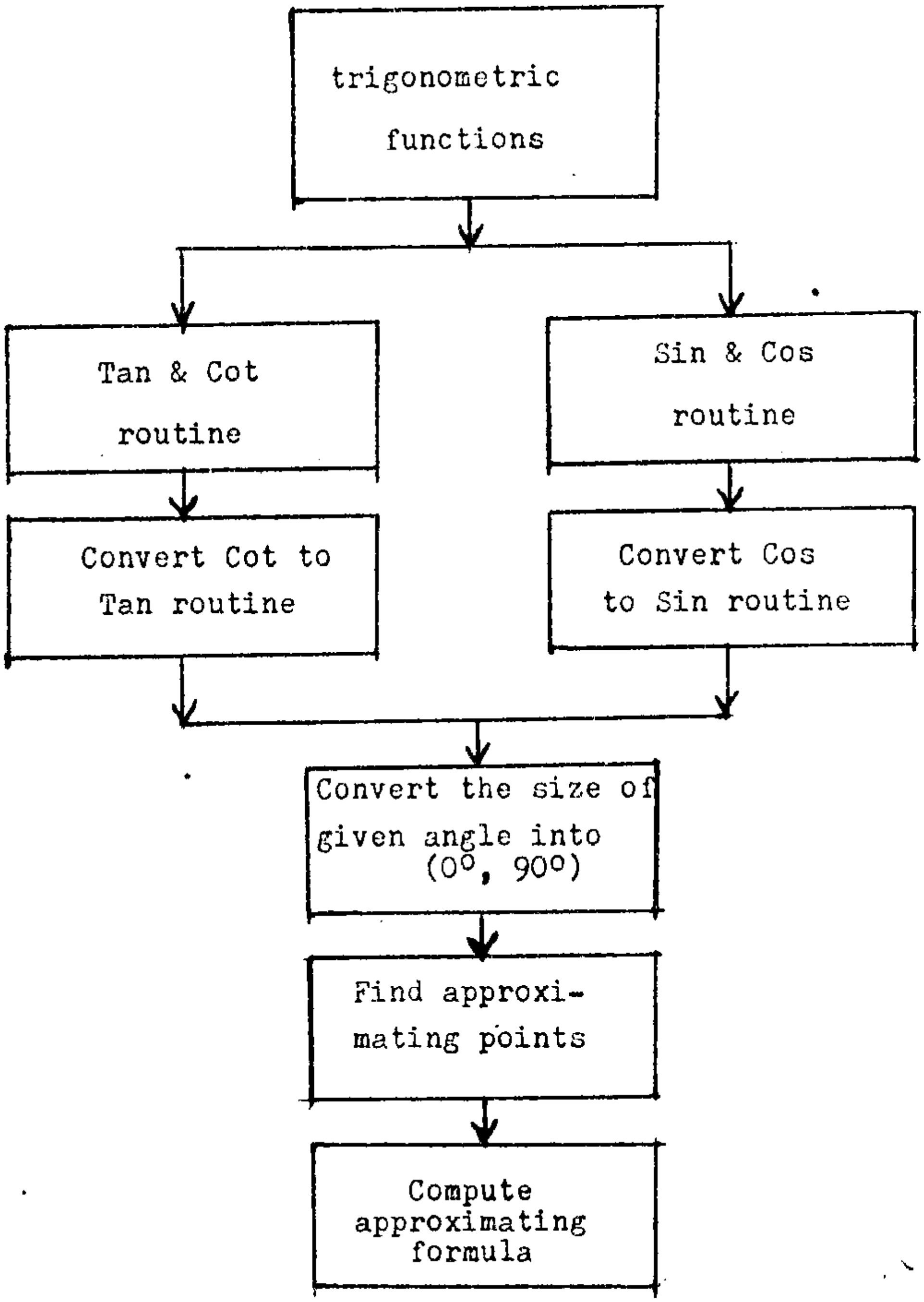


Figure 5: The System Design

Algorithm Description

For implemeting the trigonometric functions on the microprocessor, following procedure is required:

1. Convert the given angle from the range of $(-360^o, +360^o)$ into $(0^o, 90^o)$ by using the trigonometric relationship.

2. Find the four angles which are the nearest points to the given angle in the reserved table.

3. Build the approximating polynomial used the four obtained points in the table and compute the approximated value of the required trigonometric function.

ERROR ANALYSIS

There are three kinds of errors which arise in the computation of the trigonometric functions.

1. Errors due to shortened initial data

2. Truncation errors in the computational procedures

3. Errors due to the use of pseudo-arithmetic operations.

For the applied algorithm, each kinds of error should be considered and the bound for each should be defined.

Errors Due to Shortened Initial Data

These kinds of errors arise from shortening data by the need to use no more than a certain number of digits, say d digits, to represent any given number. It is assumed that if a number such as $a = \sum_i b_i 10^{n-i}$ is to be represented by a number such as $\hat{a}$ in the computer with d digits of precision, then the error $\alpha = a-\hat{a}$ will be such that $|\alpha| \leqslant 0.5 \times 10^{n-d+1}$, that is, the error will be less than or equal to $\frac{1}{2}$ in the last digit position to be retained. Further, if a number, a, is to be represented by a number $\hat{a}$ which is to have m decimal places of accuracy, then $\alpha = a-\hat{a}$ is such that $|\alpha| \leqslant 0.5 \times 10^{-m}$.

The applied floating point is represented in a way that there is at least 7 decimal places of accuracy. So, the errors due to the shortened initial data for the input data will be $|\alpha| \leqslant 0.5 \times 10^{-7}$

Truncation Error in the Computational Procedure

Since digital computing devices can perform only the fundamental arithmetical operations of addition, subtraction, multiplication, and division, the only mathematical quantity which can be calculated by their use is a rational fraction. It is fortunately the case that most functions commonly emountered can be approximated by rational fractions. However, the fact that these are only approximations should be emphasized. Thus, for example, in the applied formula which is used for computing the trigonometrical functions we have

$$Y = Y_0 + \mu^{[1]} \Delta Y_0 + \mu^{[2]} \frac{\Delta^2 Y_0}{2!} + \dots\dots + \mu^{[n]} \frac{\Delta^n Y_0}{n!}$$

where n is chosen large enough to produce an error term which is acceptable. However, regardless of the size of n, the error is generally present.[3] This error is called truncation error which is actually the remainder term of the equation.

In general, the remainder term of the applied approximation formula, Gregory-Newton Formula, is R(x) which can be computed as follows

$$R(x) = \frac{f(p)^{[n+1]}}{(n+1)!} h^{n+1} \mu^{[n+1]},$$ where p is a point in the range of the function.[3] Since the third degree of the polynomial will be used for approximation, we have

$R(x) = f^{[4]}(p)/4! \cdot h^4 \cdot \mu^4$; the interval of two points, h, assumed to be o.1; for Sin and Cos function, $f(p) \leqslant 1$ for any value of p because all orders of the derivatives of the function are Sin or Cos, and the upper bound on the magmitudes of all derivatives will be less than or equal one. Since in the range of $(0^o, 90^o)$ the value Tan and Cot are not bounded, the relative error for R(x) should be computed instead of absolute error. Since the value of $\mu = (x-x_0)/h$, for any value of x the value of μ will be $o \leqslant \mu \leqslant 1$, so the remainder is

$$R(x) \leqslant \frac{1}{24} \times h^4(\mu)(\mu-1)(\mu-2)(\mu-3)$$

$$R(x) \leqslant \frac{1}{24} \times 10^{-4} \times o.5(-o.5)(-1.5)(-2.5)$$

$$R(x) \leqslant \frac{1}{24} \times 10^{-7} \times o.837 \leqslant o.35 \times 10^{-6}$$

ERRORS DUE TO THE USE OF PSEUDO ARITHMETIC OPERATIONS

A pseudo-arithmetic operation is some operation which produces the same result as a corresponding arithmetic operation to within a certain unavoidable error.

The pseudo-operations of concern with digital computers are the counterparts of +, −, X, ÷ of usual arithmetic in which every result must be a number with at most d digits in it, for some d. Hence, some loss of accuracy will occur.[5]

Each pseudo-arithmetic operation which is performed on two numbers with d digits may result in more than d digits. In this situation, some digit must be discarded.

Since in the applied floating point representation there is at least 7 decimal place of accuracy, the maximum error due to the pseudo arithmetic operation for each performance is less than $\underline{10^{-7}}$.

The input to the approximating formula uses at least 7 decimal place of accuracy, so the round-off error in f(x) is of the oreder of 0.5×10^{-7}. In general, the maximum round-off error in computation will be

$f(x) = Y_0$	ΔY	$\Delta^2 Y$	$\Delta^3 Y$
0.5×10^{-7}	10^{-7}	2×10^{-7}	4×10^{-7}

Since the approximating formula is

$$P_3(x) = Y_0 + \Delta Y \cdot \mu + \Delta^2 Y \frac{\mu(\mu-1)}{2!} + \Delta^3 Y \mu(\mu-1)(\mu-2),$$

the maximum error added to the remainder error, $R_4(x)$, due to round-off error is

Error = $\text{Error}_a + \text{Error}_b + \text{Error}_c + \text{Error}_d$ where

$\text{Error}_a = \text{Error}(Y_0) = \underline{0.5 \times 10^{-7}}$

$\text{Error}_b = \text{Error}(\Delta Y \cdot \mu) = \Delta Y \times 2 \times 10^{-7} + \mu \times 10^{-7} \leqslant 2 \times 10^{-7} + 10^{-7} = \underline{3 \times 10^{-7}}$

$\text{Error}_c = \text{Error}\left[\Delta^2 Y \mu(\mu-1)\right] = \Delta^2 Y \left[0.5 \times 3 \times 10^{-7} + 0.5 \times 2 \times 10^{-7}\right] + 2 \times 10^{-7} \times 0.5 \times 0.5 \leqslant 2.5 \times 10^{-7} + 0.5 \times 10^{-7} = \underline{3 \times 10^{-7}}$

$\text{Error}_d = \text{Error}\left\{\Delta^3 Y \left[\mu(\mu-1)\right]\left[\mu-2\right]\right\} \leqslant \left[2.5 \times 10^{-7} \times 3 \times 3 \times 10^{-7} \times 0.25\right] \times \Delta^3 Y + 4 \times 10^{-7} \times 3 \times .25 = 7.5 \times 10^{-7} + .75 \times 10^{-7} + 3 \times 10^{-7} < \underline{10.75 \times 10^{-7}}$

So the total error is

$$\text{Error}_{Max} = \left|\text{Error}_a\right| + \left|\text{Error}_b\right| + \left|\text{Error}_c/2!\right| + \left|\text{Error}_d/3!\right| \leqslant 0.5 \times 10^{-7} + 3 \times 10^{-7} + 3/2 \times 10^{-7} + 10.75/(3 \times 2) \times 10^{-7} \leqslant 0.50 \times 10^{-7} + 3 \times 10^{-7} + 1.5 \times 10^{-7} + 1.76 \times 10^{-7} < 6.76 \times 10^{-7}$$

So maximum round-off error is less than 0.675×10^{-6}.

CONCLUSION

Since microprocessors are relatively small in main memory and slow in execution time, implementing any scientific routine in these kinds of machines should have the following characteristics: it should be easy to compute, it should be accurate and it should occupy the least amount of main memory.

The proposed formula for implementing floating point computation and the approximating algorithm for computing the trigonometric functions on the M6800 microprocessor have the above characteristics.

DOCUMENTATION
USER'S INFORMATION

INPUT

The input to the program is in the A and B registers. The value of the input angle is in the range of (-360^{0} , $+360^{0}$.) in the BCD, binary code decimal, form. The left half byte of the A register represents the sign of the angle. When the angle is in the positive range, zero should be stored into the half byte sign, otherwise nine is stored. The right half byte of the A register and the whole byte of the B register hold the value of the angle in the degree. The right half byte of the A register is zero if the absolute value of the angle is not more than 99.

e.g. The angle of -60^{0} for input is stored in the A and B registers as

A re.	B re.
90	60

and 170^{0} is stored as

A re.	B re.
01	70

OUTPUT

The output of the program is in the floating point form. Each number is represented in five contiguous locations as a block. The address of the starting location is in the x register. The first four locations hold the fraction part of the result number, and the fifth location of the block has the exponent of the number. The resulting number is represented in the BCD, binary code decimal, form. The fraction and the exponent are in the ten's complement form if they are negative n numbers. When the most significant digit of the number is zero through four the number is positive, otherwise the number is negative and is in the ten's complement form. The decimal point of the fraction part is assumed to be to the right of the least significant digit of the number.

The Major Functions of the Programs

There are seven major subroutines used in the algorithm(see figure 7). The major function of each one is as follows :

1. The Main Program

This program receives the given angle and converts it from the range of (-360° , $+360^{\circ}$) into (0° , $+90^{\circ}$). It also finds the required trigonometric function. Finally, it calls the search subroutine.

2. The Search Subroutine

After obtaining the angle from the main program, this subroutine will find the four angles which are the nearest points to the given angle in the reserved table. This subroutine then calls the polynomial subroutine and transfers the five angular values to the called subroutine.

3. The Polynomial subroutine

By calling the delta subroutine, this subroutine will compute the approximated value of the required trigonometric function and transfers the value to the main program.

4. The Delta Subroutine

By getting the approximating points, this subroutine computes the value of the coefficients. For computing these coefficients, the subroutine needs to call the addition, subtraction and the multiply subroutines. This subroutine also calls the Muo subroutine.

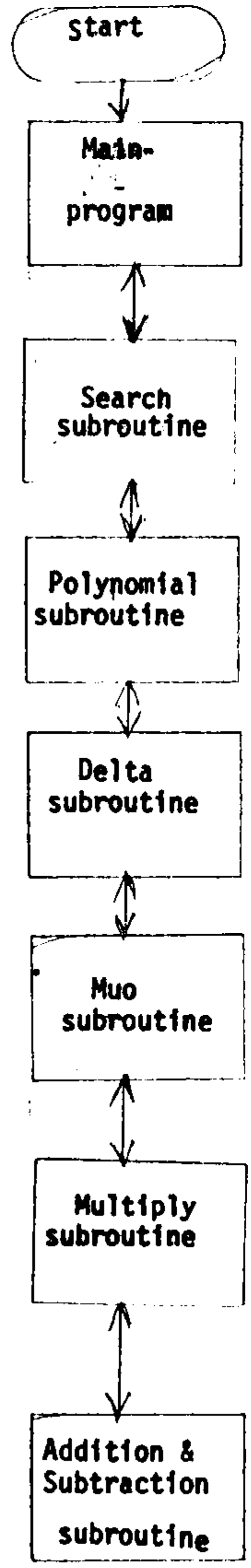


Figure 7: System Flowchart

5. The Muo Subroutine

This subroutine computes the value of μ for a given value of an angle, and it transfers the value to the polynomial subroutine for use in the approximating polynomial.

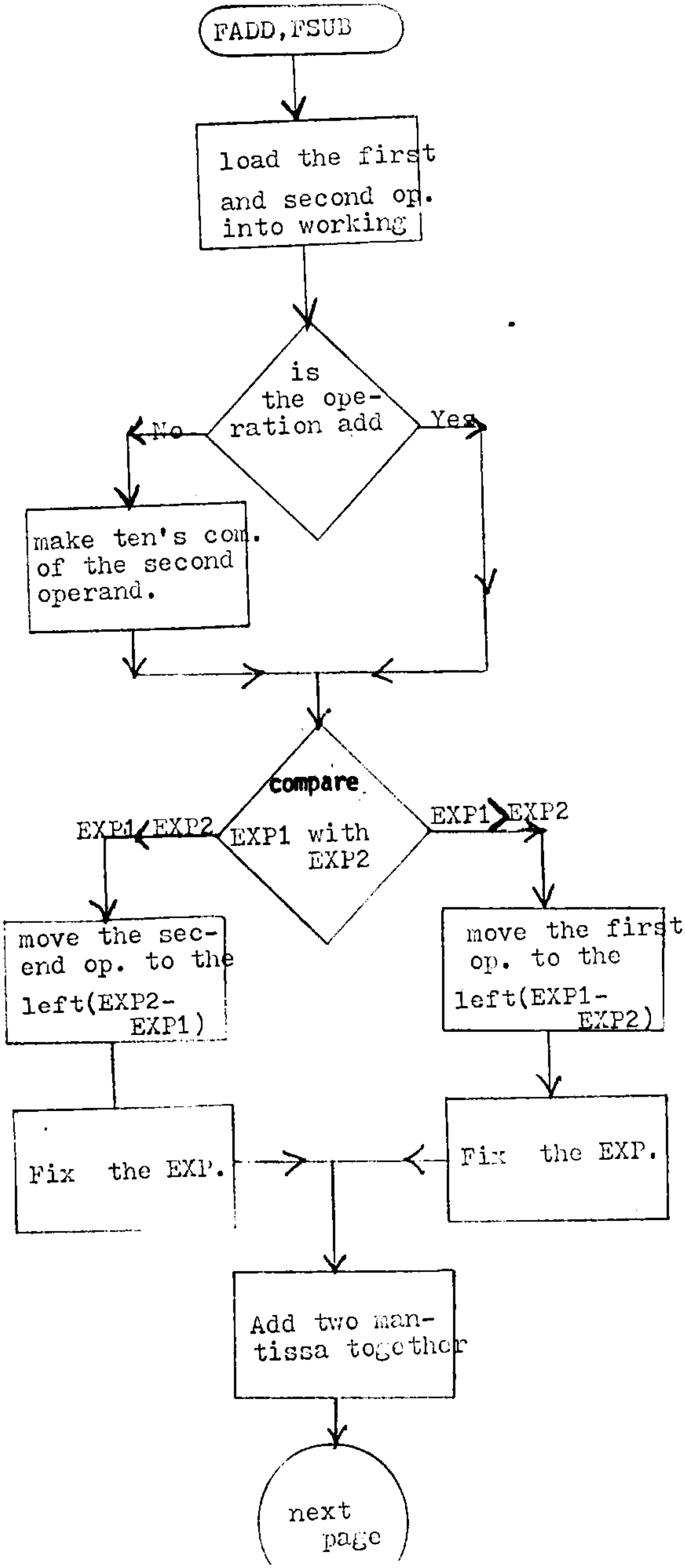
6. The Multiply Addition and Subtraction

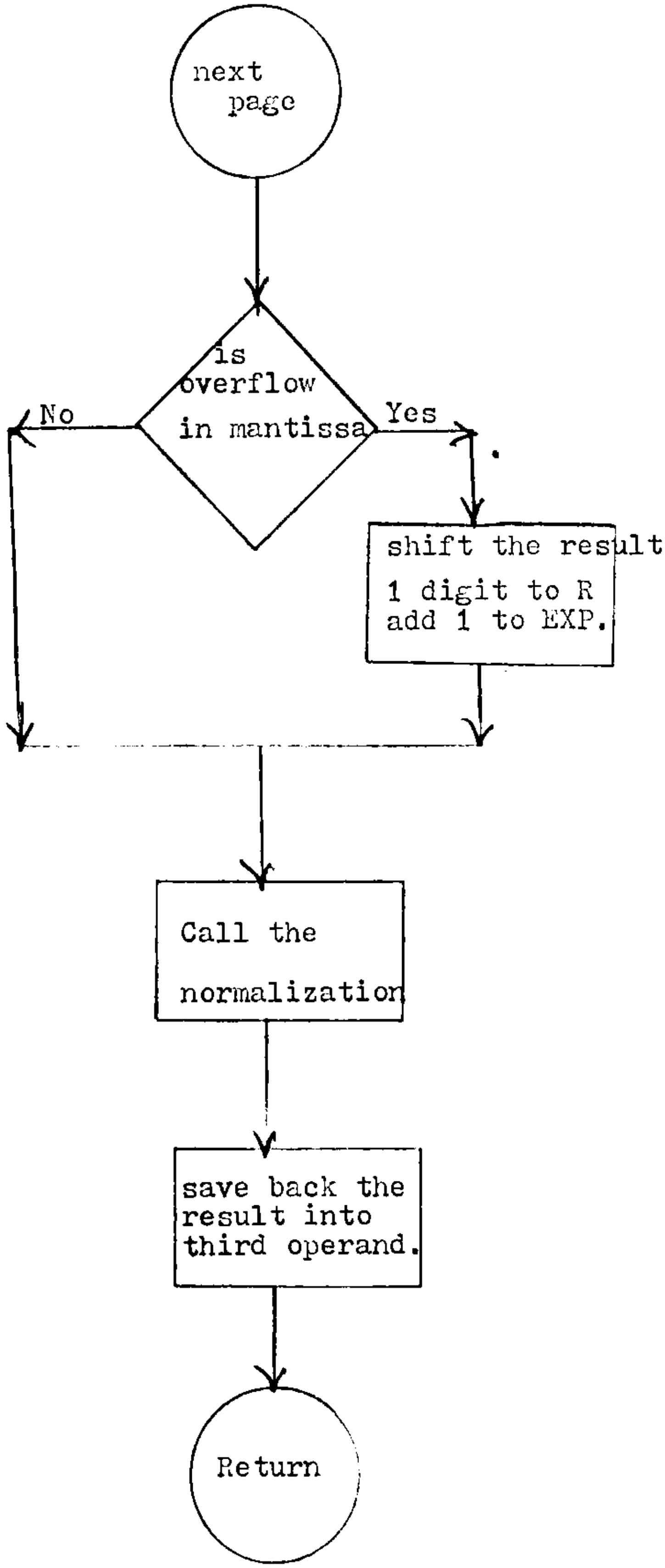
This subroutine gets two floating point numbers and adds them together. It then returns the result to the calling program.

7. The Multiply Subroutine

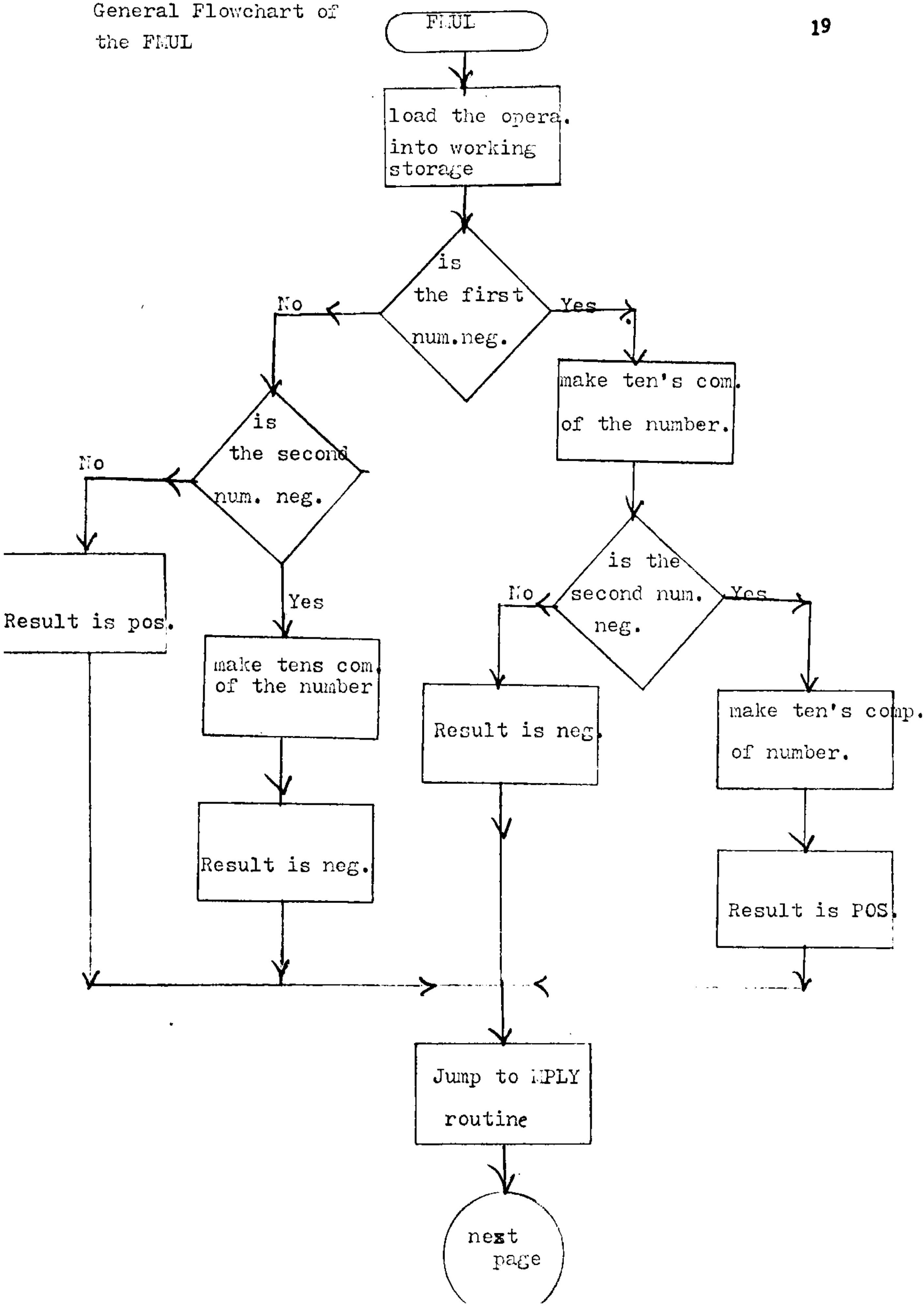
This subroutine gets two floating point numbers and multiplies them and then returns the result to the calling program.

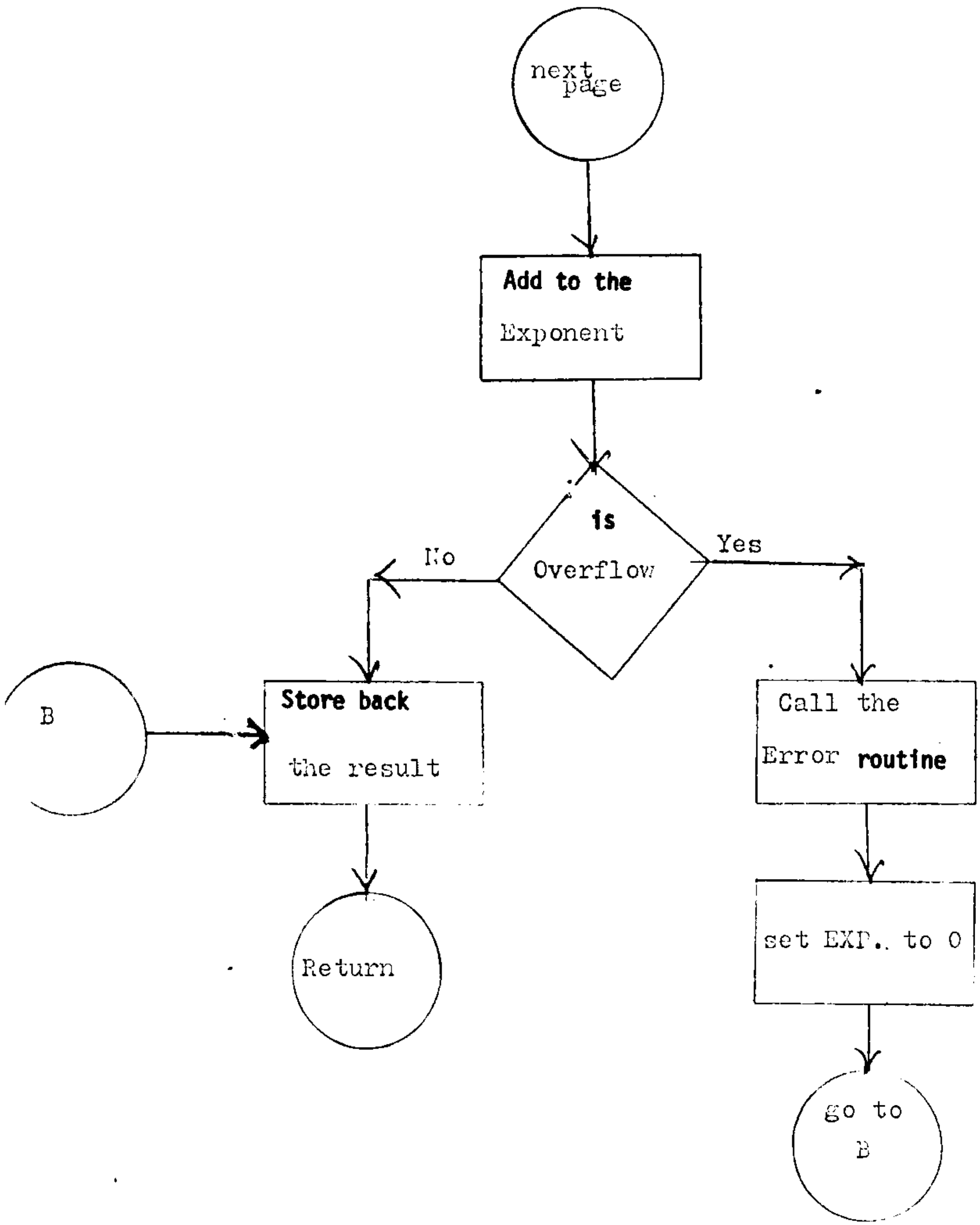
General Flowchart of the FADD,and,FSUB subroutine



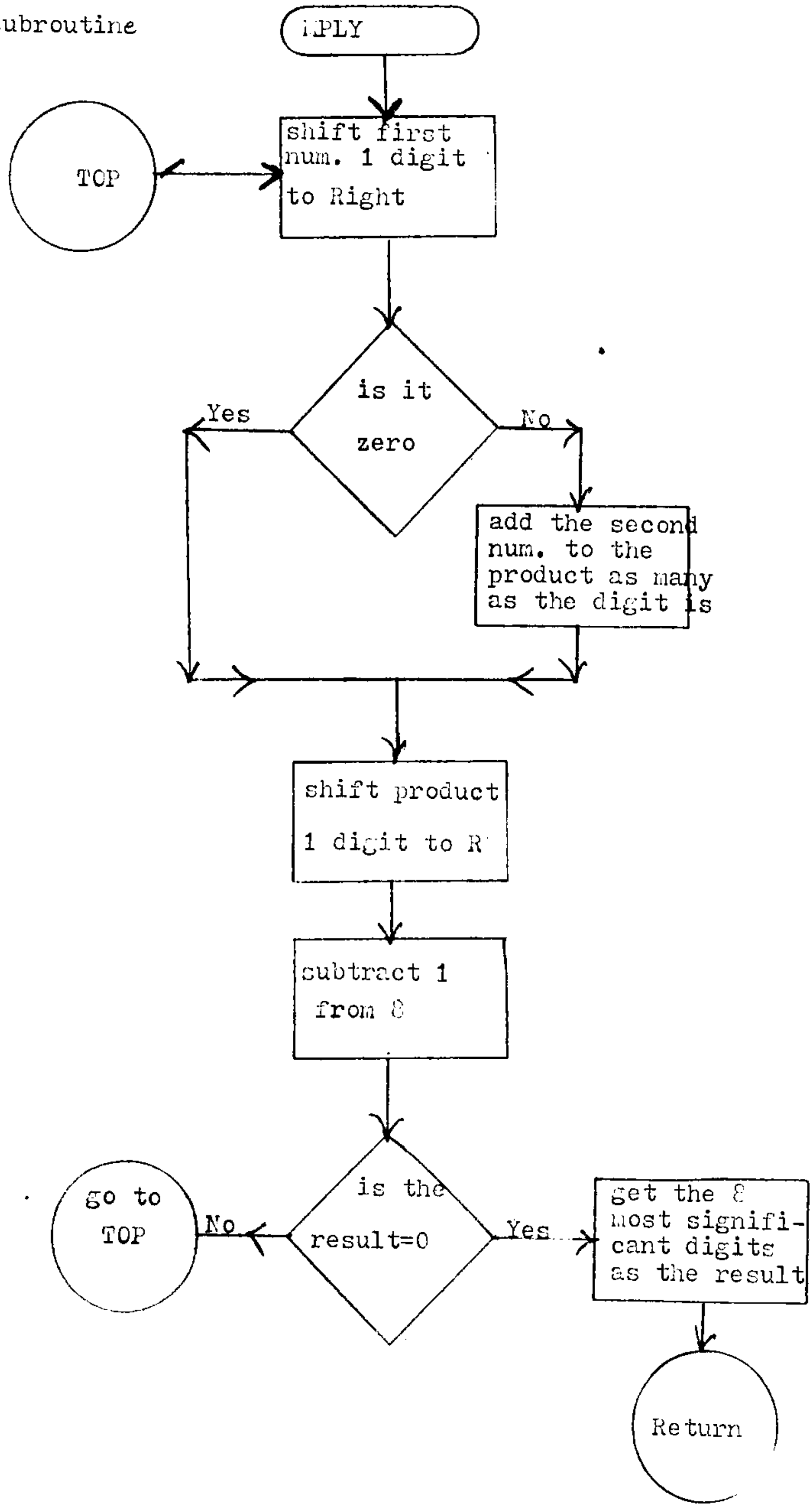

next page
is overflow in mantissa
No
Yes
shift the result 1 digit to R add 1 to EXP.
Call the normalization
save back the result into third operand.
Return

General Flowchart of
the FMUL

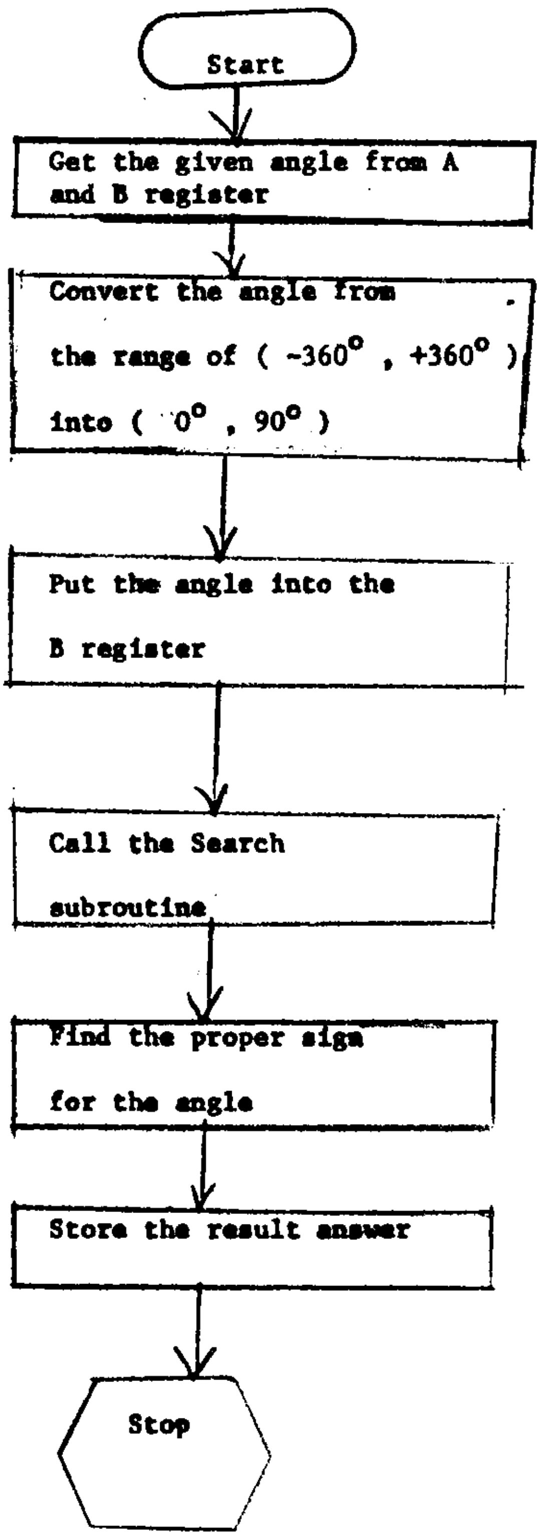



next page
Add to the Exponent
is Overflow
No
Yes
B
Store back the result
Call the Error routine
set EXP.. to 0
Return
go to B

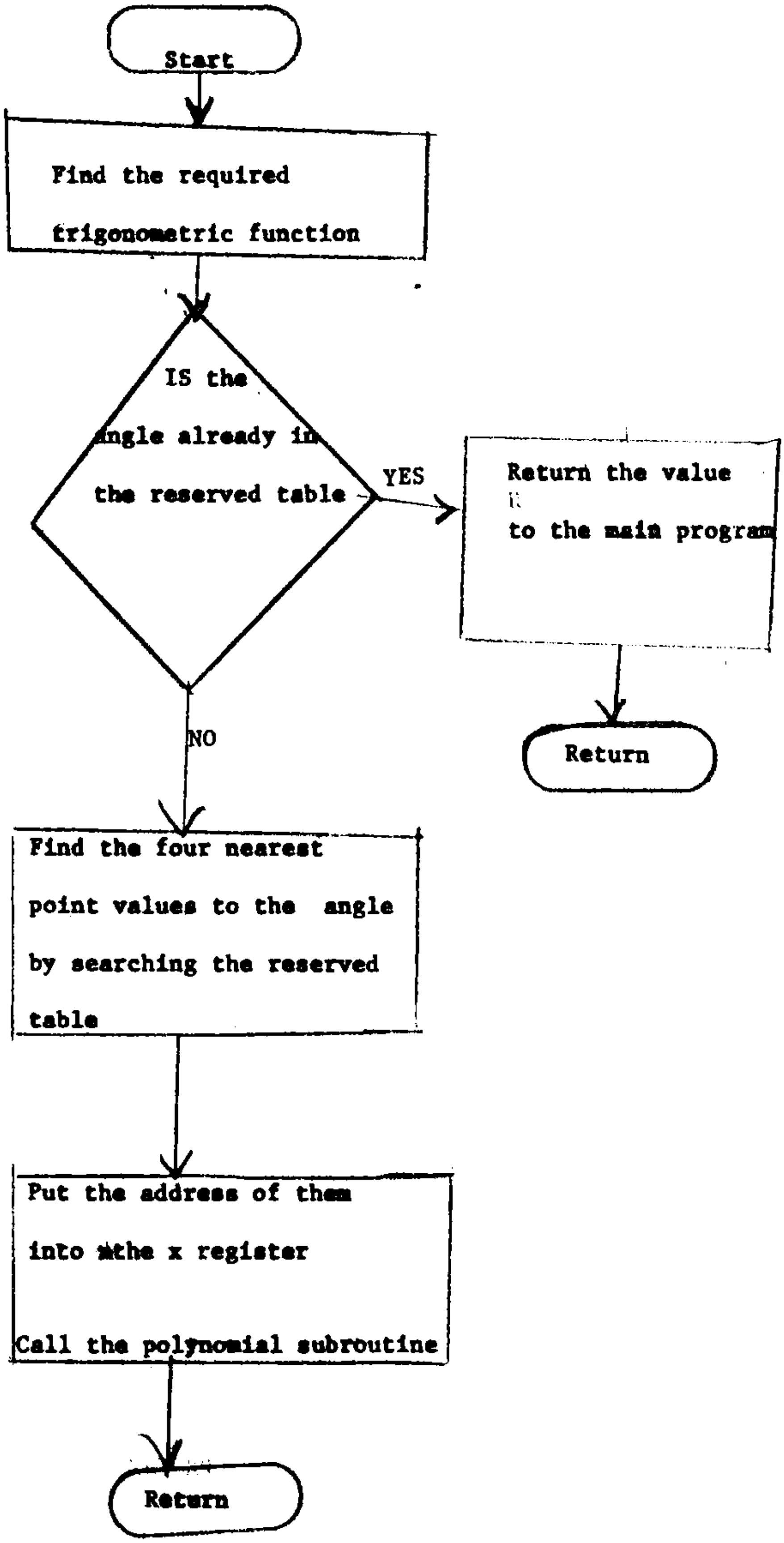
Flowchart of the MPLY subroutine



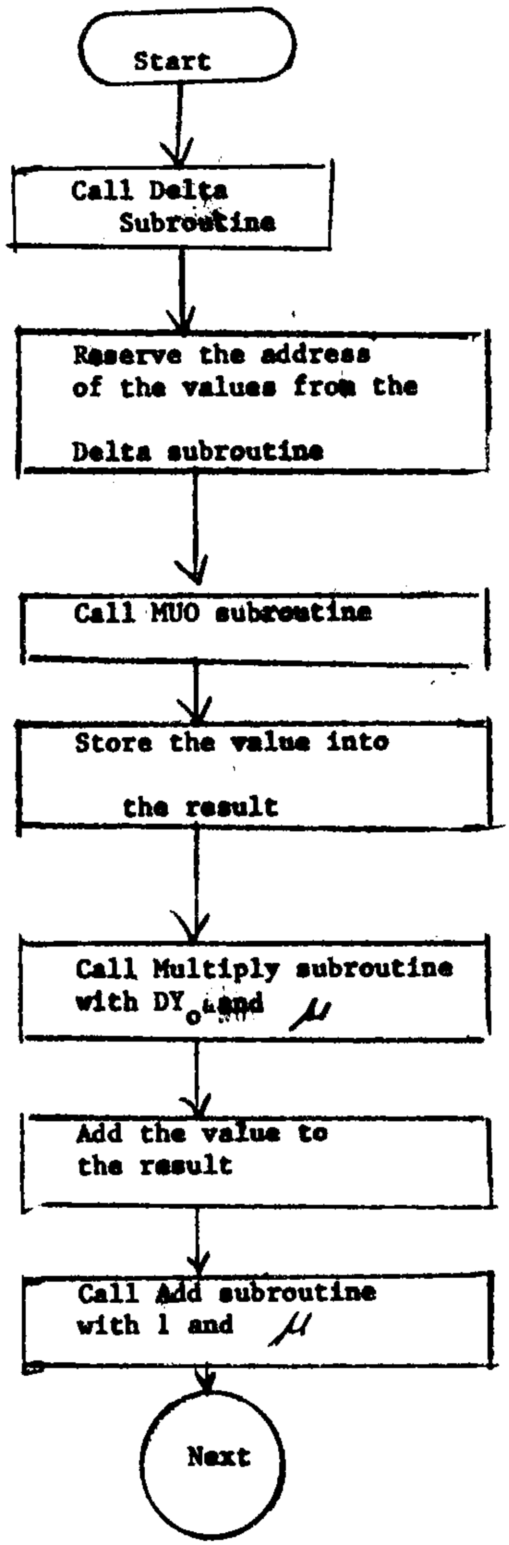
Flowchart of the Main Program

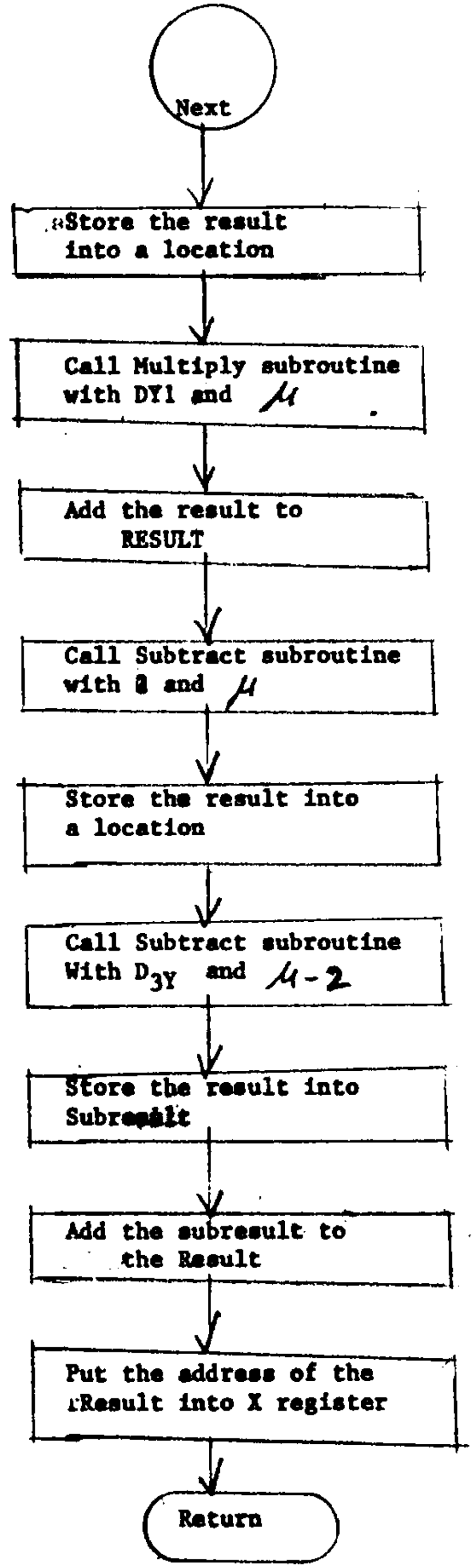


Flowchart of the Search subroutine

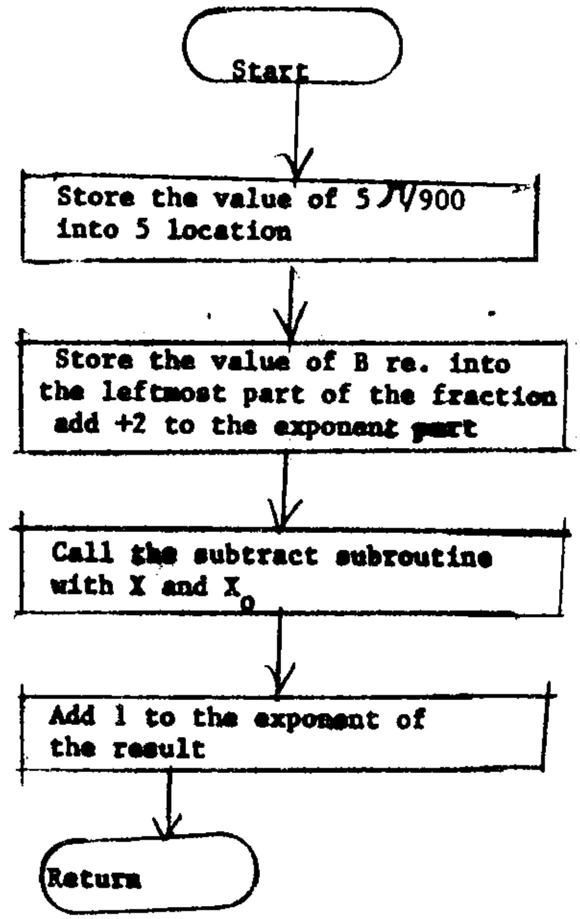


Flowchart of the Polynomial Subroutine

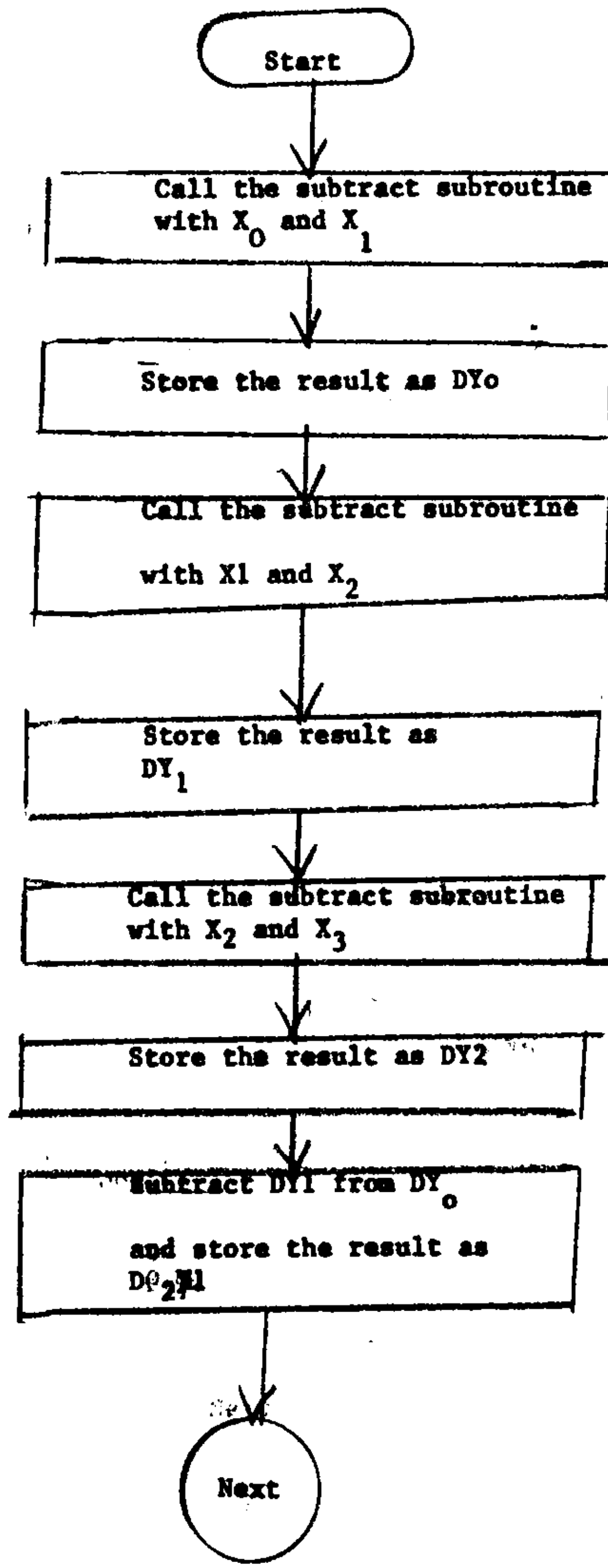


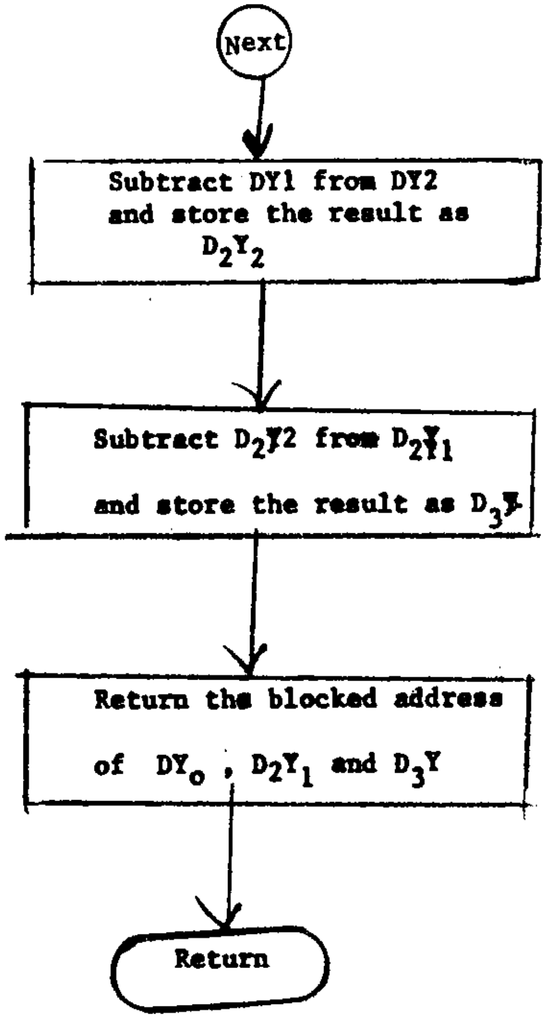

Next
Store the result into a location
Call Multiply subroutine with DY1 and μ
Add the result to RESULT
Call Subtract subroutine with 2 and μ
Store the result into a location
Call Subtract subroutine With D_{3Y} and μ-2
Store the result into Subresult
Add the subresult to the Result
Put the address of the Result into X register
Return

Flowchart of the MUO Subroutine



Flowchart of the Delta Subroutine




Next
Subtract DY1 from DY2
and store the result as
D2Y2
Subtract D2Y2 from D2Y1
and store the result as D3Y
Return the blocked address
of DYo , D2Y1 and D3Y
Return

BIBLIOGRAPHY

1. Linker, Sheldon, "What's in a Floating Point Package?", Byte, May, 1977.

2. Sterbenz, Pat H., Floating Point Computation, Englewood Cliffs, N. J.

3. Weeg, Reed, Introduction to Numerical Analysis, U. S. A. 1966.

4. Osborne, A., An Introduction to Microcomputers, Vol. II, California 1976.

5. Morton, B. R., Numerical Approximation, New York, 1964.

6. Hashizume, Burt, "Floating Point Arithmetic", Byte, November 1977.

7. Taylor L. Booth, Computing, U. S. A. 1974.

8. IBM System/360 OS, FORTRAN IV Library mathematical and service subprogram

9. **Hayes, J. G., Numerical Approximation to Functions and Data, New York, 1970.**

10. Carnahan, B., Applied Numerical Methods, U. S. A. 1969.