1620 DATA PROCESSING SYSTEM BULLETIN

IBM 1620 SUBROUTINES: PRELIMINARY SPECIFICATIONS

A set of coded instructions arranged in a logical sequence and used to direct the 1620 or any data processing system to perform a desired operation, or series of operations, is called a program or routine. Generally, most programs contain or require one or more short sequences of instructions which are a part or sub-set of the entire program and which are used to solve a particular part of a problem. Such a part of a program or routine is called a subroutine.

Usually a subroutine performs a specific function, may be executed several times during the course of the program in which it is contained (main program), and is common to a number of programs. For example, a subroutine which extracts the square root of a number may be required several times during the execution of a pipe stress analysis program. The same subroutine may be used to extract a square root in a bridge and truss design program.

Obviously, then, an efficient programming procedure is one in which all necessary subroutines are coded only once, retained on file, and incorporated into a program whenever the job performed by the subroutine is required. IBM Applied Programming has developed, for the 1620 Symbolic Programming System, a group of the more frequently required subroutines which have general applicability. These subroutines are separated into two general categories: arithmetic and functional.

The arithmetic subroutines are: 1) Floating Point Add, 2) Floating Point Subtract, 3) Floating Point Multiply, 4) Floating Point Divide, and 5) Divide. The functional subroutines are those which evaluate: 1) Floating Point Square Root, 2) Floating Point Sine, 3) Floating Point Cosine, 4) Floating Point Arctangent, 5) Floating Point Exponential (natural), 6) Floating Point Exponential (base 10), 7) Floating Point Logarithm (natural), 8) Floating Point Logarithm (base 10). All subroutines will be supplied on one subroutine tape in a language acceptable to the 1620 Symbolic Programming System; they may thus be assembled in conjunction with the user's source program. This bulletin describes the preliminary specifications of the thirteen subroutines mentioned above. It also contains a description of "floating point arithmetic," and "linkage instructions."

TABLE OF CONTENTS

LINKAGE

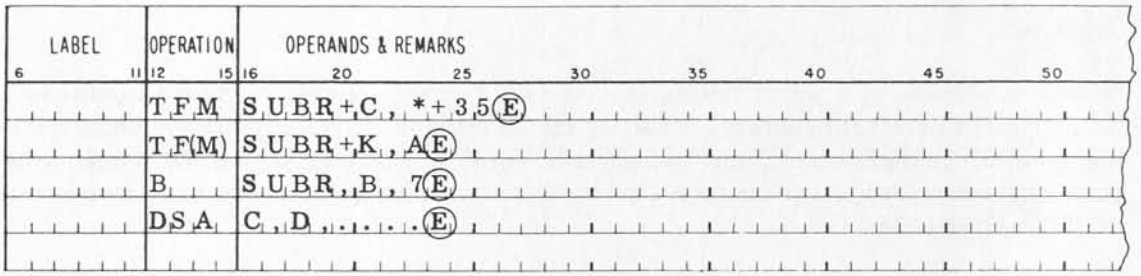
Whenever the use of a subroutine is required by a main program, it must be suitably incorporated into that program. That is, the subroutine must be logically connected to the main program in such a way that the subroutine will be executed at the proper time and, at its completion, will return control to the main program to continue the execution of the main program.

One method by which this connection can be effected is to insert the subroutine directly into the larger program where needed. A subroutine incorporated in this fashion is called an "open subroutine" or "direct insert subroutine."

In most cases, however, this method of connection is not entirely practical or desirable. For example, in a program which requires the evaluation of an Arctangent at ten different places, it would be superfluous to incorporate the same Arctangent subroutine wherever needed. The obvious solution, therefore, is to store the subroutine once, out of the main line sequence of the program and, when required, enter the subroutine by a transfer operation. Provision must also be made to return control to the main program at the completion of the subroutine. Subroutines connected in this fashion are called "closed" or "linked" subroutines. The instructions related to the entry and re-entry function constitute the linkage.

In addition to providing the facilities for connection, linkage instructions invariably contain information required by the subroutine when performing its functions. Thus, if a subroutine is required to work with certain data, the linkage instructions can be used to tell the subroutine where to find this data or they may contain the actual data to be used.

For the 1620 subroutines discussed in this bulletin, all linkages are generated automatically through the use of certain macro-instructions. That is, by placing in the source program, at the point at which a particular subroutine is desired, the macro-instruction related to this subroutine, the programmer will cause the SPS processor to generate, during assembly, the linkage to the desired subroutine. In addition the processor will arrange for the subroutine to be placed in core storage. Thus, when required during the execution of the object program, the subroutine will be transferred to and executed. The data and addresses required by the subroutine and supplied in the macro-instruction are incorporated into the linkage instructions whence they are either transmitted directly to the subroutine or simply made available for use. In this way, the subroutine obtains the information it requires to perform its given task and also obtains the information required to compute a return address to the main program. Control is returned to the main program at the completion of the subroutine by transferring to the return address. The macro-instruction related to each subroutine and the specific linkage each macro generates are discussed in the bulletins, "1620 Symbolic Programming System: Preliminary Specifications," form J28-4201 and "Additional Macro-instructions for the IBM 1620 Symbolic Programming System," form J28-4202. All linkage instructions, however, are based on a general form shown below on symbolic language:



where SUBR is the address of the first instruction to be executed in the desired subroutine, C and K are constants supplied by the SPS processor, and A, B, C, D,..., represent the information specified in the macro-instruction.

Note that in all linkages the first instruction always supplies the subroutine with the location, in the linkage instructions, of the address represented by B; this location is also used by the subroutine to compute the correct return address to the main program. Further note that in all linkages, A is always the second operand of the second linkage instruction, B is always the second operand of the third linkage instruction and any other information required by the subroutine is defined in a DSA statement (fourth statement of linkage). Additional information regarding the nature and use of the information supplied to the subroutine will be given where pertinent in later sections of this bulletin.

FLOATING POINT ARITHMETIC

Scientific and engineering computations frequently involve lengthy and complex calculations in which it is necessary to manipulate numbers that may vary widely in magnitude. To obtain a meaningful answer, problems of this type usually require that as many significant digits as possible be retained during calculation, and that the decimal point always be properly located. When applying such problems to a computer several factors must be taken into consideration, the most important of which is the decimal point location.

Generally speaking, a computer does not recognize the decimal point present in any quantity used during the calculation. Thus a product of 414154 will result regardless of whether the factors are 9.37 x 44.2, 93.7 x .442, or 937 x 4.42, etc. It is the programmer's responsibility to be cognizant of the decimal point location during and after the calculation and arrange the program accordingly. For example, in an addition operation, the decimal point of all numbers in the operation must be lined up to obtain the correct sum. Therefore, the programmer must facilitate this arrangement by shifting the quantities as they are added. It is conceivable though, that when manipulating numbers which vary greatly in magnitude, the resulting quantity could exceed allowable working limits.

The processing of numbers expressed in ordinary form (e.g., 427.93456, 0.0009762, 5382, -623.147, 3.1415927, etc.) can be accomplished on a computer only with extensive analysis to determine the size and range of intermediate and final results. When programmed, this analysis and subsequent number scaling will frequently require a larger percentage of the total time needed to solve the problem than the actual calculation. Furthermore, number scaling requires complete and accurate information regarding the bounds on the magnitude of all numbers that come into the computation (input, intermediate, output). Since it is not always possible to predict the size of all numbers in a given calculation, analysis and number scaling is sometimes impractical.

To alleviate this programming problem, a system must be employed in which information regarding the magnitude of all numbers accompanies the quantities in the calculation. That is, if all numbers are represented in some standard, predetermined format which instructs the computer in an orderly and simple fashion as to the location of the decimal point, and if this representation is acceptable to the routine doing the calculation, then quantities which range from minute fractions having many decimal places to large whole numbers having many integer places may all be handled. The arithmetic system most commonly used in which all numbers are expressed in a format having the above feature is called "floating point arithmetic." Subroutines which handle floating point numbers are called "floating point subroutines."

The notation used in floating point arithmetic is basically an adaptation of the scientific notation widely used today. In scientific work, very large or very small numbers are expressed as some number between one and ten, times a power of ten. Thus 427.93456 is written as 4.2793456×10^{2} and 0.0009762 as 9.762×10^{-4}. In the 1620 floating point arithmetic system the range of numbers is modified to extend between .1 and .99999999. That is, the decimal point of all numbers is placed to the left of the high-order (leftmost) non-zero digit. Hence, all quantities may be thought of as a decimal fraction times a power of ten (e.g., 427.93456 as $.42793456 \times 10^{3}$ and 0.0009762 as $.97620000 \times 10^{-3}$) where the fraction is called the mantissa, and the power of ten, used to indicate the

number of places the decimal point was shifted, the characteristic. In addition to the advantages inherent in scientific notation, the use of floating point numbers during processing eliminates the necessity of analyzing operations to determine the positioning of the decimal point in intermediate and final results since the decimal point is always immediately to the left of the high-order non-zero digit in the mantissa.

In 1620 floating point arithmetic each quantity operated upon is expressed as a ten-digit number consisting of a two-digit characteristic and an eight-digit mantissa. When a floating point operation is performed, the numbers involved are interpreted as

$$\underbrace{XXXXXXXXXX}_{c\quad m}$$

where c is the characteristic and m the mantissa, as explained below, and the original number is $m \times 10^{c-50}$.

The mantissa (m) consists of the rightmost eight digits of the floating point number. The decimal point is always assumed to lie immediately to the left of the high-order mantissa digit. The sign of the original quantity is always associated with the mantissa and is designated by the presence (for negative) or absence (positive) of a flag bit in the units position of the mantissa. A mantissa is called "normal" or "normalized" when its high-order digit is non-zero. In the 1620 floating point subroutines the mantissa of the quantities to be operated upon must always be normalized. Thus, the absolute value of the mantissa ranges from .10000000 to .99999999 or

$$.10000000 \leq m \leq .99999999$$

The results of a 1620 floating point operation will always be normalized. One exception to the "normalized mantissa rule" is floating point zero which is always expressed as .00000000.

The characteristic (c) represents the power of ten used to specify the location of the decimal point in the original number. It stands for the number of places the decimal point was shifted in order to place it to the left of the high-order non-zero digit. The direction of shift could be determined by the sign of the exponent (positive for left, negative for right); however, it is desirable, for the 1620 subroutines, that only one flag bit be used to represent the sign of a field. Since the sign of the floating point number is always associated with the mantissa, it cannot serve the function of indicating the sign of the characteristic. Obviously, this creates a need for expressing the characteristic in some manner which will cause no conflict with positive or negative mantissas. This need is satisfied by algebraically adding fifty to the exponent and using the resulting two digit number as the characteristic. Thus, the characteristic (c) can assume a range of values from 00 to 99 inclusive, or

$$00 \leq c \leq 99$$

Combining the ranges of the mantissa and characteristic, floating point numbers may lie within the range

$$\pm .10000000 \times 10^{-50} \text{ to } \pm .99999999 \times 10^{49}$$

The following four examples demonstrate the conversion of numbers in ordinary form to 1620 floating point notation.

Number	Normalized	1620 Floating Point
123.45678	$.12345678 \times 10^{3}$	5312345678
.00765438	$.76543800 \times 10^{-2}$	4876543800
-.12348693	$-.12348693 \times 10^{0}$	5012348693$\bar{3}$
-.00000070	$-.70000000 \times 10^{-6}$	447000000$\bar{0}$

General Notes on 1620 Floating Point Subroutines

In the 1620 floating point subroutines, numbers of magnitude equal to or greater than 10^{49} create a condition called "characteristic overflow"; those of magnitude less than 10^{-51} create a condition called "characteristic underflow." Should either of these conditions be generated as a result of an arithmetic operation, the programmer will be provided with a choice of two options as follows:

Overflow:

1. Program Halt, or
2. Cause ten nines to be placed in the result field and continue the execution of the subroutine.

Underflow:

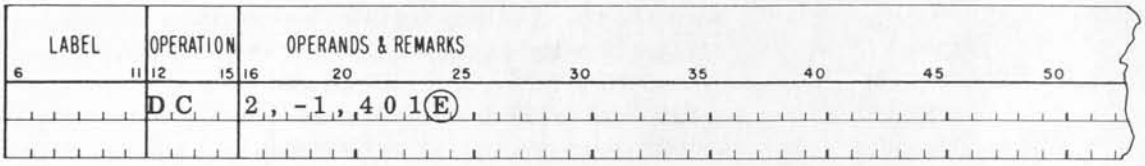
1. Program Halt, or
2. Cause ten zeros to be placed in the result field and continue the execution of the subroutine.

These options will function independently of each other. Thus it is possible to halt on an underflow and place ten nines in the result field on an overflow. The converse is also true.

The detection of an overflow or underflow condition, will cause the subroutine being executed to examine core storage position 00401 to determine the course of action. Options provided the programmer must be represented in position 401 by one of the following characters:

		UNDERFLOW	
		Halt	Store Ten Zeros in Result Field
OVERFLOW	Halt	$\bar{0}$	0
	Store Ten Nines in Result Field	$\bar{1}$	1

The code determining the option may be stored in 401 by using an unlabeled Define Constant (DC) statement as shown in the following example:



Positive codes (0 or 1) need not be preceded by a plus sign.

An overflow and/or underflow condition can arise in only six of the floating point subroutines discussed in this bulletin. They are the four arithmetic subroutines and the two exponential functional subroutines. Special conditions may, however, cause any one of the subroutines to halt. These conditions, and courses of action will be specified where pertinent.

The programmed halts obtained through the options shown above and also those which arise from the special conditions have one thing in common. That is, the programmed halt will contain, in the instruction used to halt the machine, the following information in the P and Q address portions of the instruction:

Positions Q_{10} and Q_{11} - a code which identifies the type of halt.
P address - a return address to the main program.

Thus, through console interrogation of the halt instruction, the operator will know why the routine stopped and is also provided with a means of branching back to the main program to continue execution. The codes which denote the various halts will be made available at a later date.

If, as a result of any floating point operation, a mantissa consisting of all zeros is generated, the computed characteristic will be reduced by eight.

The resulting mantissa of a floating point calculation will always be normalized with proper adjustment to the characteristic. It should be noted that results in the 1620 floating point subroutines are not rounded; therefore, the absolute value of the maximum error will not exceed 10^{-8}. In case of lengthy computation, precision gradually deteriorates. The magnitude of truncation error is dependent upon the individual computation process and cannot be predicted without a knowledge of the process in question. However, truncation error in such cases is usually no greater than the degree of error present in a rounded result.

SPECIFICATIONS OF ARITHMETIC SUBROUTINES

The following four paragraphs contain information of a general nature concerning the 1620 floating point arithmetic subroutines. All information pertaining to the "fixed" point arithmetic subroutine (Divide) is presented in the section devoted to Divide.

In each of the macro-instructions related to the four floating point arithmetic subroutines, two addresses, represented by A and B, must be specified. These addresses (which may be symbolic or actual) must reference the low-order (rightmost) position of the fields containing the floating point data to be added, subtracted, etc. It should be noted that in the context which follows, A and B will be used to reference the entire field (i.e., the quantities to be operated upon) represented by the addresses A and B.

During the execution of the floating point arithmetic subroutines, the overflow, high-positive and equal-zero indicators will be used. The overflow indicator is always reset at the beginning of each floating point arithmetic subroutine. Should its status, prior to the execution of such a subroutine be desired, the indicator must be tested and its condition stored before the linkage instructions are executed. The high-positive and equal-zero indicators will be set according to the mantissa of the result.

All 1620 floating point subroutines (arithmetic and functional) require that the floating point quantities used contain a field-terminating flag bit over the high-order position of the field (i.e., first digit of the characteristic).

Although the specifications for the floating-add and floating-subtract subroutines are shown as two separate entities, they are actually a single routine and are distinguished from each other only by the point at which they are entered. The correct entry point is obtained by the use of the macro pertaining to the particular subroutine desired.

Floating-Add

A. Macro-instruction:

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–50)
	FA	A,,BE

B. Linkage - the symbolic language equivalent of the machine instructions generated as linkage for this subroutine are as follows:

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–50)
	TFM	FA+K,,*+35E
	TFM	FA+C,,DELTAE
	B	FA,,RESISTE

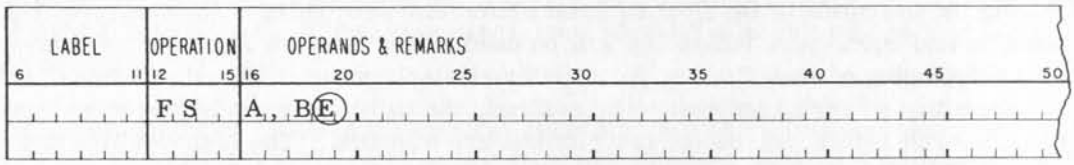
where FA is the symbolic address of the first instruction to be executed in the subroutine and DELTA and RESIST are, respectively, examples of symbolic addresses for A and B.

C. Operation - field B will be added to field A. The floating point sum replaces field A; field B remains unchanged.

D. Estimated average execution time: 9 milliseconds.

Floating-Subtract

A. Macro-instruction:



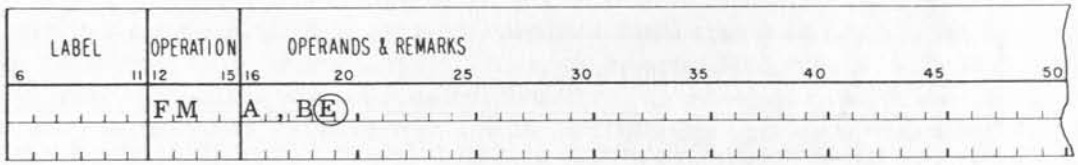
B. Linkage - the instructions generated for linkage as a result of the "FS" macro are identical to those generated for "FA" except that in each instruction an address equivalent to FS will replace the FA equivalent.

C. Operation - field B will be subtracted from field A. The floating point difference will replace field A; field B remains unchanged.

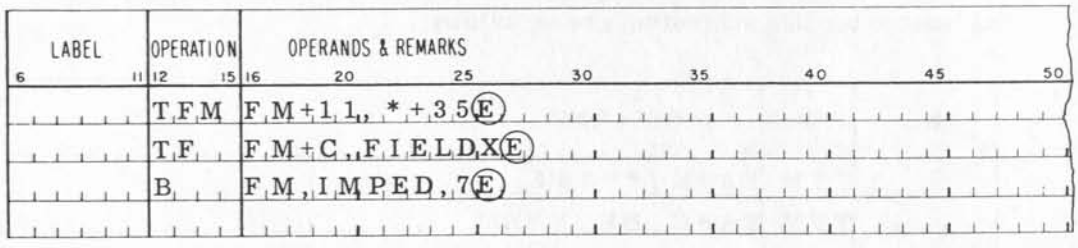
D. Estimated average execution time: 10.5 milliseconds.

Floating-Multiply

A. Macro-instruction:



B. Linkage - the symbolic language equivalent of the machine instructions generated as linkage for this subroutine are as follows:



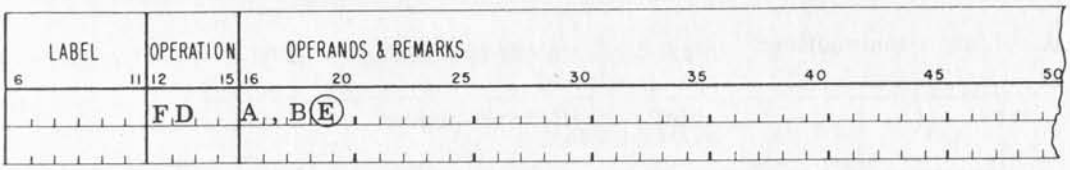
where FM is the symbolic address of the first instruction in the subroutine to be executed and FIELDX and IMPED are, respectively, examples of symbolic addresses for A and B.

C. Operation - field A will be multiplied by field B. The floating point product will be placed in core storage positions 00090 to 00099; fields A and B remain unchanged.

D. Estimated average execution time: 18 milliseconds.

Floating-Divide

A. Macro-instruction:



B. Linkage - the instructions generated for linkage as a result of the "FD" macro are identical to those generated for "FM", except that in each instruction an address equivalent to FD will replace the FM equivalent.

C. Operation - field A will be divided by field B. The floating point quotient will be placed in core storage positions 00090 to 00099; fields A and B remain unchanged.

D. Estimated average execution time: 70 milliseconds.

Divide

In addition to the A and B operands, representing the addresses of dividend and divisor, the macro-instruction for Divide requires two additional operands, which represent (1) a shift factor required by the subroutine, as explained below, and (2) the length of quotient desired.

The Divide subroutine accomplishes division by means of sucessive subtractions. It is necessary to specify to the subroutine the position of the divisor with relation to the dividend when the first subtraction takes place. This is specified in the shift operand which represents the number of positions the rightmost character of the divisor is offset from the rightmost character of the dividend when the divisor is placed beneath the dividend as in a subtraction. If this operand is positive the divisor will be displaced (using the rightmost position of both dividend and divisor as reference points) to the left the number of positions specified by the operand; if negative, displacement is to the right. This operand should be such that no more than nine subtractions will take place before the first overdraw is encountered; otherwise a divide check (overflow) will be indicated. Following are examples of the SHIFT and LQ operands and the resulting field.

Divide	Sucessive Subtractions	SHIFT	LQ	Resulting Field
$273\overline{)3972}$	3972 273	1	3	145
$3972\overline{)273}$	273 3972	-2	3	687

If, in the last example above, the shift operand were specified as -1 rather than -2 the resulting three-digit quotient would be 068 rather than 687. If the shift operand were specified as -3, more than nine subtractions would have taken place before the first overdraw were encountered (divide check). A divide check results in the overflow indicator being turned on and the subroutine returning control to the main program.

Except for the occurrence of a divide check, the overflow indicator will not be disturbed. The high-positive and equal-zero indicators will be set according to the result of the operation provided the subroutine is not terminated by a divide check.

A. Macro-instruction:

LABEL	OPERATION	OPERANDS & REMARKS
	DIV	A,B,SHIFT,LQE

B. Linkage - the symbolic language equivalent of the machine instructions generated as linkage for this subroutine are as follows:

LABEL	OPERATION	OPERANDS & REMARKS
	TFM	DIV+K,*+35E
	TFM	DIV+C,AE
	B	DIV,B,7E
	DSA	X,YE

where DIV is the symbolic address of the first instruction in the subroutine to be executed and X and Y represent constants computed by the SPS processor using the SHIFT and LQ operands of the macro.

C. Operation - field A will be divided by field B. The units digit of the remainder will be placed in core storage position 00099. The remainder will equal the divisor in length; therefore, the units digit of the quotient will be stored at 00099 minus the length of the divisor. Flag bits will be placed in the high-order position of both the quotient and the remainder. Should either the quotient or remainder be negative a flag bit will be placed over their respective units digit. The remainder will take on the sign of the dividend. Fields A and B remain unchanged by the operation.

The following two restrictions must be observed in the Divide subroutine:

1. The LQ operand must be greater than the SHIFT operand.

2. If the length of quotient field (LQ) plus the length of the divisor field (LDVR) is greater than twenty, a number of core storage positions equal to this excess must be cleared to zero. The positions which must be cleared begin at 00079

and extend into the sucessively lower numbered core storage positions. For example, if LQ + LDVR = 25, positions 00075-00079 must be cleared.

D. Estimated core storage requirements: 79 instructions plus the fixed quotient area (00080-00099).

E. Estimated average execution time (in milliseconds):

$$14.71 + .04\,LDVD + .80(LDVD + LDVR - SHIFT) + (8.24 + .68\,LDVR)LQ$$

where:

LDVD = length of the dividend field

LDVR = length of the divisor field

LQ = length of the quotient field.

SPECIFICATIONS OF FLOATING POINT FUNCTIONAL SUBROUTINES

The two addresses, A and B, which must be specified in each of the functional floating point subroutines, must reference the low-order position of (1) the field containing the quantity (argument), in floating point form, to be operated upon (address B) and, (2) the field at which the result of the operation will be stored (address A). Thus, in each functional subroutine the argument found at the field whose address is B will be evaluated and the result stored in the field whose address is A. In the context which follows, B will be used to reference the argument and A the entire result field.

The linkage instructions generated by one of the functional floating point subroutines will be equivalent to the following symbolic instructions:

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–50)
	TFM	SUBR+C,*+3,5(E)
	TFM	SUBR+K,A(E)
	B	SUBR,B,7(E)

where SUBR is the address of the first instruction to be executed in the desired subroutine, C and K are constants supplied by the SPS processor, and A and B are the addresses as specified in the macro-instruction.

At the conclusion of a functional subroutine, the status of the high-positive, equal-zero, and overflow indicators will not necessarily reflect the result of the operation. These indicators will, however, be disturbed during the execution of a functional subroutine Therefore, their status at the conclusion of such a subroutine should not be construed as their status prior to the execution of the subroutine.

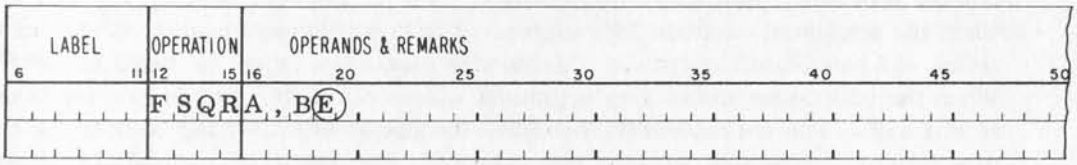
Eight individual functional floating point subroutines are discussed in this section. However, several pairs of subroutine have been combined as single subroutines (i. e., sine and cosine, natural and base 10 exponential, natural and base 10 logarithms). These subroutines have been combined in order that they may share those program steps which are common to both and thus conserve storage requirements. The individual subroutines within each pair are distinguished from each other solely by the point at which they are entered. The correct entry point is obtained through the use of the macro pertaining to the particular subroutine desired.

The functional floating point subroutines discussed in this section evaluate the function of the arguments using the following methods:

Subroutines	Method
Square Root	Odd integer
Sine and Cosine	Based on Hastings' approximation. *
Arctangent	Truncated series
Exponential (natural and base 10)	Hastings' approximation for 10^B. Additional manipulation produces e^B.
Logarithm (natural and base 10)	Truncated series for ln B. Additional manipulation produces $\log_{10} B$.

Square Root

A. Macro-instruction:



B. Operation - the square root of the argument B is extracted and the result, in floating point form, will be stored at A. The argument, which must be in floating point form, will not be disturbed by the operation.

The floating point square root subroutine will accept all numbers within the floating point range which are greater than or equal to zero. If the argument is less than zero, the subroutine will execute a programmed halt. The operator may employ one of the two following options:

1) Using the information found in the halt instruction, branch back to the main routine, or

2) Continue the execution of the subroutine and compute the square root of $|A|$.

C. Estimated average execution time: 175 milliseconds.

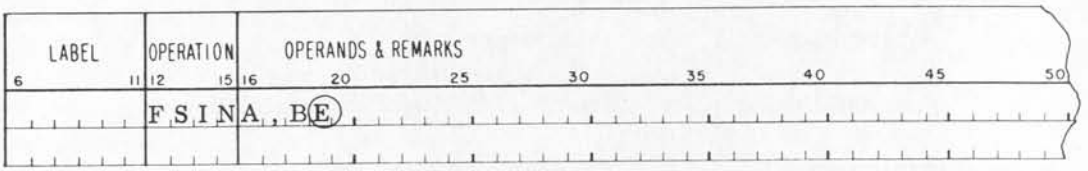
D. Estimated core storage requirements: 47 instructions plus 55 positions for working areas and constants.

---

*Hastings, Cecile Jr., Approximations for Digital Computers, Princeton University Press, Princeton, New Jersey, Copyright 1955 by the Rand Corporation.

Sine

A. Macro-instruction:



B. Operation - the sine of the argument B will be computed and the result, in floating point form, stored at A. The argument must be in radians and in floating point form. The computation will not disturb the original value of the argument.

The floating point sine subroutine will accept all numbers of floating point range up to and including characteristic 58. Should the characteristic exceed 58, the subroutine will execute a programmed halt. A branch back to the main routine can be effected through the console using the information found in the halt instruction.

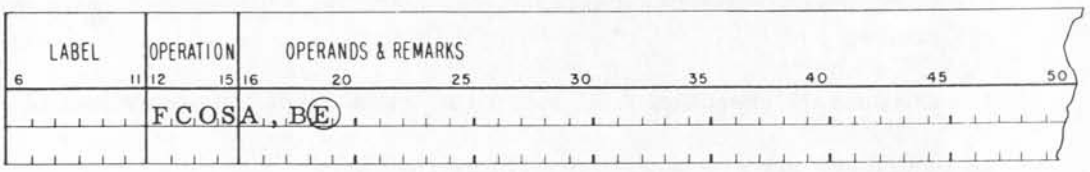
For all arguments with characteristics less than or equal to 50, the subroutine will deliver a result with a maximum error of $\pm 10^{-8}$. Decreased accuracy will result when the argument exceeds 2π radians. For this reason, arguments with characteristics of 51 or 52 may produce results with maximum error as large as $\pm 10^{-6}$. When the subroutine detects an argument whose characteristic is greater than 52, it will halt. The operator will then have the option of branching back to the main program or proceeding with the computation. Any result so computed will contain an error that varies directly with the magnitude of the characteristic.

C. Estimated average execution time: 150 milliseconds.

D. Estimated core storage requirements - see paragraph D of Cosine subroutine.

Cosine

A. Macro-instruction:



B. Operation - the cosine of the argument B will be computed and the result, in floating point form, stored at A. The argument must be in radians and in floating point form. The computation will not disturb the original value of the argument.

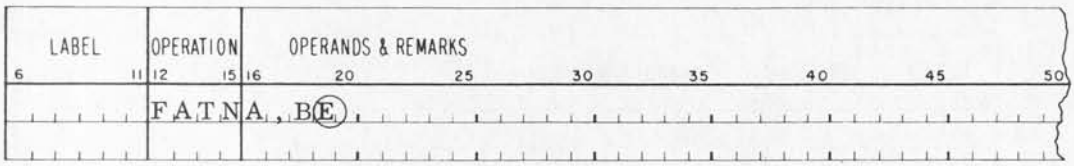
The allowable range of the argument, maximum accuracy, etc., for the Cosine subroutine are the same as those discussed in the Sine subroutine above.

C. Estimated average execution time: 155 milliseconds.

D. Estimated core storage requirements for Sine-Cosine: 71 instructions plus 120 positions for working areas and constants.

Arctangent

A. Macro-instruction:



B. Operation - the floating point value of the arctangent of B will be computed and the result stored at A. The argument must be in floating point form; the result, in radians, will be in floating point form.

The Arctangent subroutine will accept any number within the floating point range.

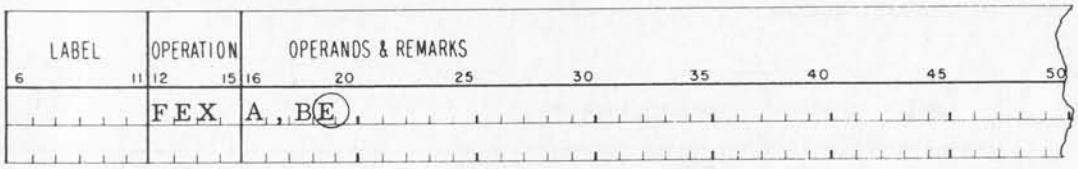
During the evaluation of the arctangent of B, use will be made of the Divide subroutine.

C. Estimated average execution time: 260 milliseconds.

D. Estimated core storage requirements: 91 instructions plus 150 positions for working areas and constants.

Exponential (natural)

A. Macro-instruction:



B. Operation - the antilogarithm of B (value of e^B), where B is in floating point form, will be computed and the result, also in floating point form, stored at A. An input value which exceeds ±112.82666 (i.e., the floating point number, ±5311282666) will result in an exponent overflow and cause the subroutine to examine core storage position 401 to determine the course of action.

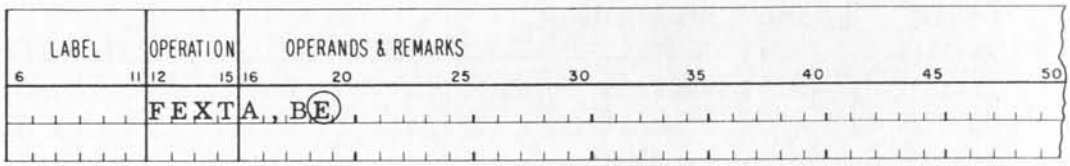
For negative arguments the subroutine will evaluate the function using the absolute value of the argument and then, using the Floating-Divide subroutine, find the reciprocal value.

C. Estimated average execution time: 160 milliseconds plus 70 milliseconds if B is negative.

D. Estimated core storage requirements - see paragraph D of Exponential (base 10) subroutine.

Exponential (base 10)

A. Macro-instruction:



B. Operation - the antilogarithm of B (value of 10^B), where B is in floating point form, will be computed and the result, also in floating point form, stored at A. An input value which exceeds ±48.999996 (i.e., the floating point number ±5248999996) will result in an exponent overflow and cause the subroutine to examine core storage position 401.

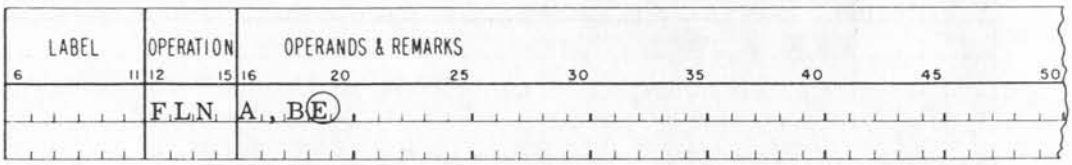
This subroutine handles negative arguments in the same manner as the natural exponential subroutine (above).

C. Estimated average execution time: 145 milliseconds plus 70 milliseconds if B is negative.

D. Estimated core storage requirements for Exponential (natural-base 10): 71 instructions plus 140 positions for working areas and constants.

Logarithm (natural)

A. Macro-instruction:



B. Operation - the floating point value of the ln B will be computed and stored at A. Input arguments must be in floating point form.

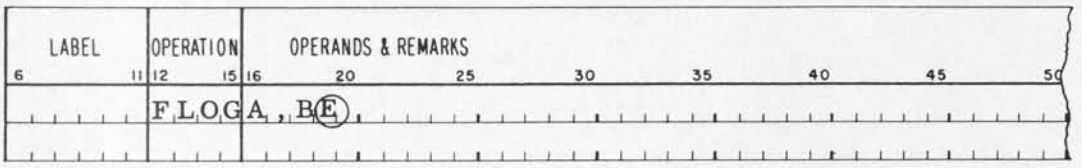
This subroutine will accept all arguments greater than zero within the floating point range. An input argument equal to or less than zero will result in a programmed halt. A branch back to the main program can be effected through the use of the information supplied in the halt instruction.

C. Estimated average execution time: 290 milliseconds.

D. Estimated core storage requirements - see paragraph D of Logarithm (base 10) subroutine.

Logarithm (base 10)

A. Macro-instruction:



B. Operation - the floating point value of the $\log_{10}$ B will be computed and stored at A. Input arguments must be in floating point form.

This subroutine will accept all arguments greater than zero within the floating point range. An input argument equal to or less than zero will result in a programmed halt. A branch back to the main program can be effected through the use of the information supplied in the halt instruction.

Both the natural and base 10 logarithm calculations will make use of the Divide subroutine.

C. Estimated average execution time: 305 milliseconds.

D. Estimated core storage requirements for Logarithm (natural-base 10): 88 instructions plus 160 positions for working areas and constants.

IBM 1620 Publications

The following IBM 1620 Systems literature has been published as of the date of this bulletin.

Form Number	Title
J28-4200-1	IBM 1620 FORTRAN: Preliminary Specifications
J28-4201	1620 Symbolic Programming System: Preliminary Specifications
J28-4202	Additional Macro-Instructions for the IBM 1620 Symbolic Programming System
J28-4203	IBM 1620 Subroutines: Preliminary Specifications

The following IBM 1620 Machine literature has been published as of the date of this bulletin.

Form Number	Title
A22-4500	1620 Machine Reference Manual
G22-4501	1620 Machine Bulletin: General Information

IBM
International Business Machines Corporation
Data Processing Division, 112 East Post Road, White Plains, N.Y.

Printed in U.S.A. J26-4203 2/60