CGTM 18
John R. Ehrman
August 1967

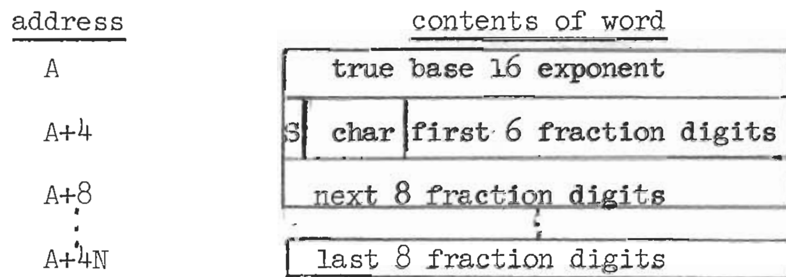
A Multiple-Precision Floating-Point Arithmetic Package for System/360

It frequently occurs that calculations performed to high precision cannot be easily handled using the hardware facilities provided with the computer. We are accustomed to the presence of double-precision floating-point instructions in present-day computers, which helps to relieve the problem of insufficient precision in the single-precision instructions. To aid the programmer whose application requires still greater precision, a set of routines has been prepared which will perform a variety of operations on floating-point numbers of dynamically variable precision. The routines have been written with the System/360 Fortran programmer in mind, but the routines are also callable from any other language which observes the standard OS/360 linkage and parameter-passing conventions.

The routines are described in detail in the following text. Users of the routines who have any comments or suggestions regarding their use are encouraged to make their desires known; such suggestions will help in clarifying any difficulties and in making additions to the set of routines.

Number Representation in Memory

Each multiple precision (abbreviated MP) floating-point number is represented in memory in the following way: a single fullword integer contains a base 16 exponent (the address of this word is taken to be the address of the MP number); the next fullword in memory contains the sign of the number in the leftmost bit, and the first 6 hexadecimal digits of the fraction part (or mantissa); succeeding fullwords contain 8 hexadecimal digits each of the fraction. The exponent is stored in true two's complement integer form; that is, it contains no bias as does the seven-bit characteristic of the usual System/360 floating-point representation. If a typical MP number with N fraction words is assumed to be stored beginning at an address A (or, in short, "at A"), then its representation in memory is as shown in the following diagram.



The bits in the position occupied by the characteristic of a standard System/360 floating-point number (namely the 7 low-order bits of the first byte of the word at A+4) do not participate in any of the operations performed on the MP numbers. However, a user may wish to use the result of a calculation performed in multiple precision for some further operations with short or long floating-point operands; to facilitate this usage the rightmost 7 bits of the exponent are inserted (with a bias of 64 added) in the proper position of the word at A+4. Thus, so long as the exponent of the MP number satisfies the inequality $-64 \leq \text{exponent} < 64$, the short (or long) floating-point number contained in A+4 (or A+4 and A+8) is the correct unrounded representation of the leading six (or fourteen) hexadecimal digits of the MP number.

All MP numbers are assumed to be either true zero (which includes sign, exponent, and fraction) or normalized such that the leading hexadecimal digit (the 4 high order bits of the byte at A+5) is not zero. If this condition is not satisfied the MP routines cannot be expected to work properly. All results generated by these routines from properly normalized operands will be properly normalized.

Because the exponent is a full word in length, the range of possible values of MP numbers is greatly extended over that available for standard System/360 floating-point numbers. Except for zero, a MP number will lie approximately in the range

$$10^{-4.4 \times 10^9} \sim 16^{-(2^{31}+1)} < |\text{MP number}| < 16^{(2^{31}-1)} \sim 10^{4.4 \times 10^9}$$

This should be sufficient to satisfy most needs.

Operations on Multiple-Precision Numbers

Some of the routines which manipulate or perform arithmetic on MP numbers require an area of memory to be set aside for the use of these routines. In particular, operands will be taken from and placed into this area for certain operations, so that it may be regarded as an "accumulator area". This area will simply be called the AC in the following descriptions; it should be noted that the assignment of the location of the AC is done by the user of these routines, so that he has control over its location and access to its contents.

Before describing the individual routines that make up the MP floating-point package, several notational devices and design considerations will be discussed.

1. The notation C(X) or C(AC) will be taken to mean "the multiple-precision number stored beginning at "X" or the "the MP number in the AC". This of course refers to a block of words.

2. The notation s(X) or s(AC) refers to the sign of the MP number stored at X or in the AC.

3. All the routines leave results in core memory, rather than in one or another of the general or floating-point registers. This means that the user need not be concerned with declaring any modes

for the routines used. In particular, some of the routines also store a fullword integer as an indication of the condition of the result of the calculation; this is discussed in the description of each routine.

4. Though it is possible to generate results which (due for example to exponent overflow) cannot properly be represented in MP form, an attempt is always made to compute a useful result and then to indicate to the user that the error condition has occurred. The user can then take any corrective action desired, and will often be able to make use of the computed result.

5. No attempt is made to round computed results, since it is a trivial matter to increase the precision of a MP number by simply adding further words to the mantissa. In general, errors are on the order of 1 or 2 bits in the least significant hexadecimal digit. In all arithmetic operations performed in the AC (this excludes only the conversion routines to and from MP form) an additional fraction word is automatically carried throughout the computation. This "guard word" serves to maintain accuracy by assuring, for example, that a normalizing left shift after a multiplication does not introduce truncation errors. These routines may therefore be used with confidence that spurious errors will not appear in the course of the computation.

6. In the descriptions of the individual routines, only the Fortran calling sequence will be given, since the statements necessary to call from a machine-language program are very similar.

7. Whenever the letter N is used, it refers to the quantity specified by the first parameter in the call to MPASET (q.v.).

A summary of the routines and their arguments, actions, and results is given in Table I, following the descriptions of the individual routines. Information regarding the length of each routine and its execution time is given in Table II. Some of the execution times are estimates; see reference 3. A number of examples of the use of these routines is given after the descriptions. In particular, example B illustrates a method which may be used to input MP quantities; no input routine is provided.

Details of methods used are given in the Appendix, in the program listings, and in reference 4.

CALLING SEQUENCES:

Initialization Routine

Before any other routines in the multiple-precision package are called, it is necessary to provide two pieces of information: (1) The length, or precision, to which computations are to be performed, and (2) the location of the area of memory to be used for the AC. To provide this information, the following call is used:

```
CALL MPASET(N,ACLØC,S)
```

where N is an integer quantity which specifies the number of fraction words to be carried in all MP operations; this is roughly equivalent to specifying a precision of 9N - 2 deciaml digits. (The exact number of decimal digits equivalent to this number of fraction words is 9.64N - 2.41 .) Since one word is required for the exponent, N+1 words will be required to hold a MP number. N must satisfy $3 \leq N \leq 10000$. The lower bound is 3 because shorter length arithmetic can be performed by the floating-point hardware. The upper bound of 10000 is arbitrary and is simply used to catch incorrect calls to MPASET; it may be changed by reassembling MPASET. ACLØC is the location of the first full word of a block of N+3 full words to be used for the AC, which is addressed at ACLØC. ACLØC must be the address of a byte on a fullword boundary. S is the name of a fullword switch which is set to zero if no errors were detected in the parameters N and ACLØC, and is set to the integer value +4 otherwise. This quantity should be tested by the user on return from MPASET; no other error indication is given.

Conversion Routines

A. Machine Form to Multiple-Precision Form

There are three routines which convert quantities from standard System/360 representation to MP form, and they are called in the following way:

```
CALL MPAITM(I,X)
```

converts the fullword integer quantity I to MP form and stores it beginning at X.

CALL MPAETM(E,X)

converts the short floating-point quantity E to MP form and stores it beginning at X.

CALL MPADTM(D,X)

converts the long floating-point quantity D to MP form and stores it beginning at X.

Note that because any quantity representable in standard System/360 form can be represented properly in MP form, no error conditions are detected by these routines. If, however, the floating-point quantities E or D are unnormalized, their MP representations will be incorrect, and the operation of the routines performing arithmetic on such quantities is not generally known. Note also that there is no restriction on X, so that it may be the same as the location specified for the AC.

B. Multiple-Precision Form to Machine Form

There are three routines which convert quantities from MP form to standard System/360 representation.

CALL MPAMTE(X,E,S)

causes the MP quantity at X to be converted to short floating-point form and stored at E;

CALL MPAMTD(X,D,S)

causes the MP quantity at X to be converted to long floating-point form and stored at D. In both routines, the fullword quantity S indicates whether the conversion was performed successfully. If the MP number can be represented to the desired accuracy by the exponent range provided for System/360 floating-point numbers (namely -64 to +63) then S will be set to zero. If the exponent of the MP number exceeds this range, then the correct characteristic (i.e. exponent+64) modulo 128 is used for the result characteristic and S is set nonzero: if the exponent is greater than +63, S is the

integer +4 and if the exponent is smaller than -64, S is set to -4. In any case, some result will be stored at E or D.

CALL MPAMTI(X,I,S)

causes the quantity at X to be converted to a fullword integer and stored at the fullword location I. If the conversion is successfully performed, S is set to zero. If the conversion cannot be performed, S will be set to +4 and the contents of I are unpredictable.

C. Multiple-Precision Form to Decimal Character String

This routine is used to convert MP numbers to a form suitable for printing.

CALL MPACVD(IE,DS,ND)

causes the MP number in the AC to be converted from hexadecimal to a decimal representation in the form of a string of EBCDIC digits. An iterative method is employed which first converts the number to its decimal floating-point value and then places the decimal (i.e. base 10) integer exponent of the result in the fullword location IE, and a string of digits representing the normalized decimal fraction, preceded by a sign and a decimal point, beginning at the location DS. Exactly ND digits will be placed in the string, so that ND+2 EBCDIC characters are output in all and occupy storage positions DS through DS+ND+1. If the MP number is zero, the exponent placed in IE and all the digits will be zero. Otherwise, the first digit following the decimal point will be nonzero.

Because an iterative method of conversion is used, MP numbers with exponents differing from zero by large amounts may require considerable time to convert. If the user knows, for example, that a number to be converted is on the order of magnitude of $10^{1000000}$, he is advised to precompute that constant, divide the MP number by it, and then add 1000000 to the decimal exponent in IE to give an arithmetically correct result.

No conversion will be attempted if ND is less than zero or greater than 2^{17}, or if the exponent exceeds 2^{29} in magnitude. The contents of the AC are destroyed by the conversion process.

Integer-Part Routine

To replace a MP number by the same number with its fractional part removed, the following call is used:

```
CALL MPAIPT(X)
```

causes the integer part of the MP number at X to replace the MP number at X. If the MP number is less than one in magnitude, its integer part is zero; if its magnitude is greater than the number of significant digits carried, it is unchanged.

Note that the fractional part of a number may be found by subtracting its integer part from it.

Addition and Subtraction Routines

These routines assume that one MP operand is already present in the AC, and leave the result of the operation in the AC.

```
CALL MPAADD(X,S)
```

causes the MP quantity at X to be added to the MP quantity in the AC.

```
CALL MPASUB(X,S)
```

causes the MP quantity at X to be subtracted from the MP quantity in the AC.

```
CALL MPAADM(X,S)
```

causes the magnitude of the MP quantity at X to be added to the MP quantity in the AC.

```
CALL MPASBM(X,S)
```

causes the magnitude of the MP quantity at X to be subtracted from the MP quantity in the AC.

In each case, the fullword integer quantity S will be set to

indicate whether the computation was performed correctly. If the result in the AC is correct, S will be set to zero. If there was an exponent overflow or underflow, the correct result with exponent modulo 2^{32} is left in the AC, and S is set to the integer +4; if the exponent is then +, the exponent underflowed, and if the exponent is -, the exponent overflowed. If any internal error checks fail during the addition, S is set to +8 and the computation of the sum is abandoned. Hence the user must be aware of the possibility of exponent spill when his operands approach the ends of the allowable exponent range.

A single guard word of 8 hexadecimal digits is carried throughout.

Note that an unnormalized MP quantity cannot be normalized by adding zero to it or by adding it to itself.

Note that the address of the quantity X may be the same as the address of the AC; however, to double a number use MPAIMP, and to clear the AC use MPAZAC.

Integer Multiply and Divide Routines

These routines are particularly useful when it is desired to multiply or divide a number in MP form by a number in fullword integer form, because they require only one pass over the MP number to be operated on rather than the many required by the multiplication and division routines MPAMPY and MPADIV.

```
CALL MPAIMP(I,S)
```

causes the MP quantity in the AC to be multiplied by the integer quantity represented by the fullword at I. The product is left in the AC.

```
CALL MPAIDV(I,S)
```

causes the MP quantity in the AC to be divided by the integer quantity represented by the fullword at I. The quotient is left in the AC.

If the result can be computed correctly, the fullword integer quantity S is set to zero. If an attempt is made to divide by zero, or if an internal check fails during the computation, S is set to the

the integer value +8 and control is returned to the user. If an exponent overflow or underflow occurs, the correct result is left in the AC with exponent modulo 2^{32} and S is set to +4. Because multiplication or division by an integer can change the exponent by at most 8, overflows and underflows are easily anticipated; as in the case of the addition routines, if S is +4 an overflow is indicated when the resultant exponent is negative and an underflow when the resultant exponent is positive.

Multiplication Routine

To form the product of two MP numbers, the following call is used:

```
CALL MPAMPY(X,S)
```

causes the product of the MP number in the AC and the MP number at X to be left in the AC. If the product was computed correctly, the fullword integer quantity S will be set to zero. If an exponent overflow or underflow occurs, S will be set to the fullword integer value +4; if an arithmetic error is detected during multiplication, S will be set to +8. Note that exponent spills are easily anticipated since the exponent of the product is almost always the sum of the exponents of the operands, and the direction of the spill can be found by examining the sign of the resultant exponent, as described above. X may be the same as the address of the AC.

A single guard word of 8 hexadecimal digits is carried throughout the computation, so that the error in the product is expected to be at most one in the least significant digit.

Division Routine

To form the quotient of two MP numbers, the following call is used:

```
CALL MPADIV(X,S)
```

causes the MP number in the AC to be divided by the MP number at X,

and the quotient to be left in the AC. If the quotient is correctly computed, the fullword integer quantity S will be set to zero. If there is an exponent overflow or underflow, S is set to +4; if a fatal error occurs during the computation, or a division by zero is attempted, S is set to +8. Note that exponent spills are easily anticipated because the exponent of the quotient is almost always the difference of the exponents of the operands, and that the direction of the spill may be found by testing the sign of the resultant exponent.

The error in the quotient is expected to be at most one in the least significant digit. The address of X should not be the same as the address of the AC.

Comparison Routine

To compare two MP numbers, the following call is used:

CALL MPACAS(X,S)

causes the fullword integer quantity S to be set to the sign of the difference C(AC)-C(X). If the MP numbers in the AC and at X are equal, S will be set to zero. Otherwise S will be set the integer value +4 if C(AC) > C(X), or -4 of C(AC) < C(X). Note that this routine is considerably faster than an actual subtraction, since the hardware compare instructions are used.

Sign Manipulation Routines

These routines allow the user to manipulate the signs of MP numbers.

CALL MPACHS

causes the sign of the MP number in the AC to be inverted.

CALL MPASSP

causes the sign of the MP number to be set +.

CALL MPASSM

causes the sign of the MP number in the AC to be set -.

CALL MPASAS(X)

causes the sign of the MP number in the AC to be set to agree with the sign of the number at X.

CALL MPASSS(X)

causes the sign of the MP number at X to be set to agree with the sign of the MP number in the AC.

Note, however, that a zero quantity always has a + sign which will not be changed by these routines. This is consistent with standard System/360 usage, in which a zero quantity is always assumed to be positive. The test for zero is usually made by testing the most significant fraction digit, so that an unnormalized operand will usually be treated as a zero.

Number Transmission Routines

In each of the following routines, zero quantities will have a + sign even though a sign change may be implied in the subroutine call.

A. To AC

The following four routines cause MP numbers to be transmitted to the AC.

CALL MPACLA(X)	causes $C(X) \rightarrow C(AC)$
CALL MPACLS(X)	causes $-C(X) \rightarrow C(AC)$
CALL MPACAM(X)	causes $\lvert C(X) \rvert \rightarrow C(AC)$
CALL MPACSM(X)	causes $-\lvert C(X) \rvert \rightarrow C(AC)$

In each case the MP number originally at X remains unchanged.

B. From AC

The following four routines cause MP numbers to be transmitted from the AC.

CALL MPASTØ(X)	causes C(AC) → C(X)
CALL MPASTN(X)	causes -C(AC) → C(X)
CALL MPASTM(X)	causes |C(AC)| → C(X)
CALL MPASNM(X)	causes -|C(AC)| → C(X)

In each case the contents of the AC is unchanged.

C. Elsewhere

The following four routines cause the MP number at X to be transmitted to the block of N+1 words beginning at Y.

CALL MPAMØV(X,Y)	causes C(X) → C(Y)
CALL MPAMVN(X,Y)	causes -C(X) → C(Y)
CALL MPAMVM(X,Y)	causes |C(X)| → C(Y)
CALL MPAMNM(X,Y)	causes -|C(X)| → C(Y)

In each case the MP number originally at X is unchanged.

D. Zeroing

To clear the AC to zero, use

```
CALL MPAZAC
```

To clear to zero the block of N+1 words beginning at the fullword X, use

```
CALL MPASTZ(X)
```

In each case, no other MP quantities are affected.

Hexadecimal Print Routine

Occasionally it is useful to be able to examine the actual bits of a number in MP form. To print such numbers,

```
CALL MPAPRH(X, N,IU)
```

causes the MP number at X consisting of an exponent and N fraction words to be formatted and transmitted to Fortran logical unit IU. The format is such that the exponent is printed as a decimal integer, and the fraction words are printed in hexadecimal, 10 words to a line. This routine requires the presence of the Fortran I/O routine IBCØM.

Utility Routines

Two utility routines are used by the other routines in the MP arithmetic package: These are MPASHR# and MPASHL# , which are used for shifting MP numbers in the AC right and left respectively. Since these routines are used only indirectly, their calling sequences are not of general interest, but are given in the program listing. The user should realize however that their presence is required and that space for them will be needed in the program as a whole.

Appendix

This appendix describes certain details of the methods used in performing the MP computations, and some considerations regarding the use of the routines.

* Instruction Set

All of the routines were coded using only the standard instruction set. Thus, instructions of the floating-point and decimal features are not used, so that these routines will run on all models of System/360.

* Condition Code

No "condition code" is set to indicate the result of a computation as in System/360, since all quantities are left in areas of the memory known to and available to the calling programs. Thus operands may be tested directly during the intermediate steps of a calculation, which is often not the case in higher level languages.

* Return Code

Many of the routines indicate the status or condition of a computed result by storing a fullword integer at a specified place in the calling program. This integer is always a multiple of 4, so that this integer quantity may be used as an index in a branching switch by machine-language calling programs. The integer is occasionally allowed to take on negative values; this facilitates its use in Fortran IF statements in a natural and consistent way.

* Program Re-usability and Re-entrancy

All the MP routines are serially re-useable; many are also re-entrant. To use these routines for multiprogramming purposes, however, a slight change will be needed to MPASET, which contains the quantities N and ACLØC. This is because most of the MP routines refer to N and ACLØC via the external symbol MPANBR# , and the current values of the two quantities would have to be saved and restored in some manner.

The necessary modifications to the other routines are very simple, and involve generally only the necessity of obtaining a save area from the supervisor.

* Division

The division method used is essentially that of Pope and Stein (reference 1) with several modifications to account for the fact that fractional rather than integer quantities are being manipulated. Care is taken to scale all operands so as to obtain maximum accuracy and efficiency. No remainder term is computed, since it would essentially require arithmetic of length 2N words; if such quantities are needed, the programmer can easily arrange to increase the precision of the appropriate operands during the computation.

* Conversion to Decimal

A short internal table is kept of fullword fixed-point fractions which represent ratios of certain powers of 10 and 2. Thus, for example, 10/16 is a fraction representing $2^{-4}*10^{+1}$ when numbers with positive exponents are being converted, and representing $2^{+4}*10^{-1}$ when numbers with negative exponents are being converted. These single-length fractions are used because each iteration in the conversion is proportional only to N rather than N^2, which would be the case if a conversion constant were to be computed by the routine. The largest fraction stored represents 16^{11}, so that the reduction process requires roughly (E/11)+1 cycles, where E is the magnitude of the base 16 exponent. Decimal digits are generated 9 at a time, so that roughly (ND/9)+1 additional conversion cycles are needed, where ND is the number of decimal digits requested. Hence any pre-scaling performed by the programmer can effectively reduce the time required for conversion. Each cycle will require about the same time as a single call to MPATMP.

* Multiplication

Even though the product is kept only to N fraction words, a full N+1 words are computed prior to postnormalization. It is on this basis that the maximum error in the final digit will be roughly $N*2^{-20}$ before

truncation, and hence will differ by at most 1 in the final digit from the correctly rounded result.

* Addition and Subtraction

As in the other computations, a single guard word is kept during the intermediate steps. This means that if severe cancellation occurs when a difference is taken that the final result will contain at least one significant hexadecimal digit. If such a situation is anticipated, however, it is a simple matter to increase the precision of the calculation for as many steps as may be necessary. Shifting of the smaller operand is done during the addition, so that no actual shifting in memory is required.

* Interrelationships

Each routine operates independently of the others: the only routine called by any of the other routines in the package are the shift routines MPASHL# and MPASHR# . Thus extra space will not be required in memory for uncalled routines.

Examples

A. Read an integer from a data card, compute its reciprocal, and print the result in hexadecimal, using 25 fraction words.

```
      DIMENSIØN I(28)              I  will be used for the AC
      CALL MPASET(25,I(1),S)       this sets N and AC location
      IF(S)9,1,9                   check error code
 1    READ (5,2) J
 2    FØRMAT(I15)
      CALL MPAITM(1,I(1))          puts MP value of 1 in AC
      CALL MPAIDV(J,S)             divide by integer J
      IF(S)9,3,9                   check error code
 3    CALL MPAPRH(I(1),25,6)       prints MP number on unit 6
      GØ TØ 8
 9    WRITE(6,4)
 4    FØRMAT(6HOERRØR)
 8    STØP                         to read more cards, GØ TØ 1 instead
      END
```

B. Compute and store at TWENY3 the square root of 23, accurate to at least 50 decimal places. Then read three integer quantities A,B, and C each of up to 50 digits in length, and compute and print $(A + B\sqrt{23})/C$.

Since the number of decimal digits carried is roughly 9N-2, six fraction words will suffice. It is assumed that the integers A,B, and C are punched without sign on separate cards, right-justified in the first 54 columns of each data card, and that a non-zero number in columns 55-63 means the preceding integer is negative.

```
    INTEGER Q(7,3)TWENY3(7),X(10),T(7),CH(13)
    DØUBLE PRECISIØN TW3
    TW3 = DSQRT(23.0D0)          get double precision √23
    CALL MPASET(6,X(1),Z)
    IF(Z)99,1,99
 1  CALL MPADTM(TW3,TWENY3(1))
    DØ 3 K = 1,2                 two iterations are enough for √23
    CALL MPAITM(23,X(1))
    CALL MPADIV(TWENY3(1),Z)
    IF(Z)99,2,99
 2  CALL MPAADD(TWENY3(1),Z)
    IF(Z)99,10,99
10  CALL MPAIDV(2,Z)
 3  CALL MPASTØ(TWENY3(1))
 4  READ(5,5) Q                  read 3 data cards
 5  FØRMAT(7I9)
    DØ 8 J = 1,3
    CALL MPAZAC
    DØ 6 K = 1,6
    CALL MPAITM(Q(K,J),T(1))     forget about checking Z for brevity's sake
    CALL MPAIMP(10**9,Z)
 6  CALL MPAADD(T(1),Z)
    IF(Q(7,J))7,8,7
 7  CALL MPACHS
 8  CALL MPASTØ(Q(1,J))          the integers A,B, and C are in MP form
    CALL MPACLA(TWENY3(1))       get √23      in the columns of array Q
    CALL MPAMPY(Q(1,2),Z)        *B
    CALL MPAADD(Q(1,1),Z)        +A
    CALL MPADIV(Q(1,3),Z)        /C
    CALL MPACVD(E,CH(1),50)      convert AC
    WRITE (6,9) E,CH             print result
 9  FØRMAT(I12,2X,13A4)
    GØ TØ 4
99  WRITE (6,98)
98  FØRMAT(5H1PUNT)
    STØP
    END
```

C. Compute and print the square root of 10 to 5000 decimal places. In this case it is advantageous to change the length to which the calculation is performed as it is done. A slightly different square root algorithm will be used which allows an integer rather than a MP division to be used:

$$X_{n+1} = \frac{X_n}{2}\left(3 - \frac{X^2{}_n}{J}\right) \qquad \text{(derived from } f(X) = \frac{J}{X^2} - 1\text{)}$$

```
    DIMENSIØN AC(600),CH(2000),THREE(600),T(600),N(9)
    DØUBLE PRECISIØN X
    N(1) = 5    )
    N(2) = 10   |
    N(3) = 20   |
    N(4) = 40   |
    N(5) = 80   }   these are the lengths to which each step will be
    N(6) = 160  |                                         calculated
    N(7) = 320  |
    N(8) = 555  |
    N(9) = 555  )
    CALL MPASET(555,AC(1),Z)    again, Z will be ignored for simplicity's sake
    CALL MPAITM(3,THREE(1))
    X = DSQRT(1.OD1)
    CALL MPADTM(X,T(1))
    DØ 2 K = 1,9
    CALL MPASET(N(K),AC(1),Z)
    CALL MPACLA(T(1))
    CALL MPAMPY(T(1),Z)
    CALL MPAIDV(10,Z)
    CALL MPACHS
    CALL MPAADD(THREE(1),Z)
    CALL MPAMPY(T(1),Z)
    CALL MPAIDV(2,Z)
 2  CALL MPASTØ(T(1))
    CALL MPACVD(Z,CH(1),5002)
    WRITE(6,1)Z,CH(N),N=1,1251)
 1  FØRMAT('1SQRT(10)'//4X,'E',I3,5X,26A4/(17X,25A4))
    STØP
    END
```

Table I

Call	Mnemonic	Action
CALL MPASET(N,AC,S)	SET SETUP	Initialize MP routines, S = 0 if OK
CALL MPAITM(I,X)	ITM Integer to Multiple	Convert integer I to MP at X
CALL MPAETM(E,X)	ETM Short Float to Multiple	Convert Short Floating E to MP at X
CALL MPADTM(D,X)	DTM Long Float to Multiple	Convert Long Floating D to MP at X
CALL MPAMTE(X,E,S)	MTE Multiple to Short Float	Convert MP at X to Short Float at E; S shows error
CALL MPAMTD(X,D,S)	MTD Multiple to Long Float	Convert MP at X to Long Float at D; S shows error
CALL MPAMTI(X,I,S)	MTI Multiple to Integer	Convert MP at X to integer at I; S = 0 if OK
CALL MPAIPT(X)	IPT Integer Part	Leaves Integer Part of C(X) at X
CALL MPAADD(X,S)	ADD Add	$C(X) + C(AC) \rightarrow C(AC)$
CALL MPASUB(X,S)	SUB Subtract	$-C(X) + C(AC) \rightarrow C(AC)$ S = 0 indicates OK
CALL MPAADM(X,S)	ADM Add Magnitude	$\lvert -C(X)\rvert + C(AC) \rightarrow C(AC)$
CALL MPASBM(X,S)	SBM Subtract Magnitude	$-\lvert C(X)\rvert + C(AC) \rightarrow C(AC)$
CALL MPACVD(IE,DS,ND)	CVD Convert to Decimal	Decimal exponent → IE, ND digits in string at DS
CALL MPAIMP(I,S)	IMP Integer Multiply	$I*C(AC) \rightarrow C(AC)$ S = 0 indicates OK
CALL MPAIDV(I,S)	IDV Integer Divide	$C(AC)/I \rightarrow C(AC)$
CALL MPAMPY(X,S)	MPY Multiply	$C(X)*C(AC) \rightarrow C(AC)$, S = 0 indicates OK
CALL MPADIV(X,S)	DIV Divide	$C(AC)/C(X) \rightarrow C(AC)$, S = 0 indicates OK
CALL MPACAS(X,S)	CAS Compare AC to Storage	Sign $(C(AC)-C(X)) \rightarrow S$
CALL MPACHS	CHS Change Sign	$-C(AC) \rightarrow C(AC)$
CALL MPASSP	SSP Set Sign Plus	$\lvert C(AC)\rvert \rightarrow C(AC)$
CALL MPASSM	SSM Set Sign Minus	$-\lvert C(AC)\rvert \rightarrow C(AC)$
CALL MPASAS(X)	SAS Set AC Sign	$s(X) \rightarrow s(AC)$
CALL MPASSS(X)	SSS Set Storage Sign	$s(AC) \rightarrow s(X)$
CALL MPAMØV(X,Y)	MØV Move	$C(X) \rightarrow C(Y)$
CALL MPAMVN(X,Y)	MVN Move Negative	$-C(X) \rightarrow C(Y)$
CALL MPAMVM(X,Y)	MVM Move Magnitude	$\lvert C(X)\rvert \rightarrow C(Y)$
CALL MPAMNM(X,Y)	MNM Move Negative Magnitude	$-\lvert C(X)\rvert \rightarrow C(Y)$
CALL MPACLA(X)	CLA Clear Add	$C(X) \rightarrow C(AC)$ C(X) unchanged
CALL MPACLS(X)	CLS Clear Subtract	$-C(X) \rightarrow C(AC)$
CALL MPACAM(X)	CAM Clear Add Magnitude	$C(X) \rightarrow C(AC)$
CALL MPACSM(X)	CSM Clear Subtract Magnitude	$- C(X) \rightarrow C(AC)$
CALL MPASTØ(X)	STØ Store	$C(AC) \rightarrow C(X)$
CALL MPASTN(X)	STN Store Negative	$-C(AC) \rightarrow C(X)$
CALL MPASTM(X)	STM Store Magnitude	$\lvert C(AC)\rvert \rightarrow C(X)$ C(AC) unchanged
CALL MPASNM(X)	SNM Store Negative Magnitude	$-\lvert C(AC)\rvert \rightarrow C(X)$
CALL MPASTZ(X)	STZ Store Zero	Zero → C(X)
CALL MPAZAC(X)	ZAC Zero AC	Zero → C(AC)
CALL MPAPRH(X,N,IU)	PRH Print in Hex	Print N-precision MP at X on unit IU in hex

Table II

Routine		Length(bytes)		Timing: A + BN + CN^2 microseconds					
Name	Deck Name*	Hex	Dec.	Model 50 A	Model 50 B	Model 50 C	Model 75 A	Model 75 B	Model 75 C
MPASET	MPA0	60	96	110			22		
MPAITM	MPA1	F0	240	308	13		54	1.4	
MPAETM				160	10		32	1.7	
MPADTM				160	10		32	1.7	
MPAMTE	MPA2	B2	178	173			37		
MPAMTD				178			35		
MPAMTI	MPA3	88	136	124			28		
MPAIPT	MPA5	A4	164	158	11		28	2.3	
MPAADD etc.	MPAA	4A6	1190	426	34		87	4.4	
MPAIMP	MPAI	22C	556	464	68		88	10.3	
MPAIDV				681	103		131	18.4	
MPAMPY	MPAM	224	548	526	198	37	97	31	5.2
MPADIV	MPAQ	2AC	684	497	262	66	96	40	10
MPACHS etc.	MPAS	C0	192	95			20		
MPACAS	MPAT	C0	192	170			37		
MPAMØV	MPAZ	1CC	460	168	30		36	.69	
MPASTZ				183	45		44	.79	
MPAZAC etc.				163	70		36	1.73	
MPASHR#	MPA4	EC	236	included in times for routines which call them					
MPASHL#									
MPACVD	MPAC	350	848	see text					
MPAPRH	MPAP	138	312	I/O dependent					

* used only for sequencing information

REFERENCES:

1. Pope and Stein, Communications of the ACM, December 1960, Volume 3, p. 652.

2. IBM System/360 Principles of Operation, Form No. S360-01, Form A22-6821.

3. IBM System/360 Model 50 Functional Characteristics, File No. S360-01, Form A22-6898.

   IBM System/360 Model 65 Functional Characteristics, File No. S360-01, Form A22-6884.

   IBM System/360 Model 75 Functional Characteristics, File No. S360-01, Form A22-6889.

4. Logical Arithmetic on Computers with Two's Complement Binary Arithmetic, SLAC-PUB-302, April, 1967.