Computer Algorithms for the Manipulation of Multi-precise Integers

A thesis presented to the

Department of Computer Science

University of Manitoba

In partial fulfilment of the requirements

for the degree of

Master of Science

By John C. Gabel

August 1972✓

Acknowledgements

I would like to thank Mr. Lee James for his guidance and encouragement throughout the course of my work. I am also grateful to Dr. C. R. Zarnke for assisting me with the interpretation of programs which he developed to manipulate multi-precise integers, and for suggesting algorithms to be implemented in the programs which I developed.

TABLE OF CONTENTS

ABSTRACT

This paper is an investigation into the manipulation of fixed-point (integer) values which are too large to be stored using the standard word length provided on a computer. Important aspects of two program packages which operate on multi-precise integers of arbitrary length are examined. A study is made of the algorithms used, and of the machine resource requirements of each program package. From this study, conclusions are drawn regarding the merits and limitations of each program package.

CHAPTER 1

Introduction

The Concept of a Multi-precise Integer

Integer arithmetic is used extensively in many problems especially in number theory. Frequently the integers being dealt with become very large and manipulation of these integers, even with the aid of a computer, is difficult.

Computers use a fixed word length for representing integer values. They perform operations with integers by using machine instructions designed to operate on words of this fixed length. Although this word length varies from computer to computer, in every case the fixed word length restricts the user who wishes to carry out computations involving integers too large to be represented by a single word.

This problem can sometimes be overcome by using floating-point values. However, even though numbers of greater magnitude can be represented in this way, there is usually a loss of precision. Furthermore, the maximum magnitude of floating-point values may be intolerably low.

To maintain maximum precision and to represent large integer values, two steps must be undertaken. First, a method of representing large integers by using more than one word for each integer must be developed. Integers of this type are called multi-precise integers. Secondly, a set of algorithms for operating on integers must be decided upon and a set of routines must be written which use these algorithms to manipulate multi-precise integers. These routines will perform basic operations such as addition, multiplication, and input/output of multi-precise integers, as well as performing other useful operations; such as raising a multi-precise integer to a power or finding its square root.

The method of representing multi-precise integers and the routines written to manipulate integers in this representation should require as little computer memory as possible and should execute as quickly as possible. In addition to this, the routines should be easy to use and should be portable to other computer systems. Certainly, trade-offs will be necessary in attempting to satisfy these conditions and they must be made as the result of a decision on the desired attributes of the routines.

Goals of This Study

This paper examines two program packages, each of which implements algorithms for doing operations on integers of arbitrary magnitude and sign. The first program package was developed at the University of California (Berkeley) by D. H. Lehmer and P. Weinberger. The name given to this set of routines is MPARITH and this name will be used throughout this paper to refer to this program package. The second set of routines has been given the name MPACK. The basic routines in this package were developed by C. R. Zarnke of the University of Manitoba to manipulate positive integers only. These basic routines were modified by the author to operate on integers of either sign. In addition to this, several routines were added to this package to have the same routines available in MPACK which are available in MPARITH. A more complete description of the content and characteristics of these two packages is given in Chapter II. A complete explanation of how these routines are used is given in Appendix A.

This study is designed to compare all aspects of the two program packages mentioned above. This comparison will include an examination of the algorithms used, memory requirements, time requirements, and any other features or limitations of interest. Appropriate circumstances for the use of each program package will be discussed.

Throughout this paper, a single asterisk is used to denote multiplication. Thus (A*B) means A times B. Two asterisks are used to denote exponentiation. Thus (A**B) means A raised to the power B.

CHAPTER II

Description of program packages

All routines in each package are designed to be called by programs written in the Fortran language. Provided that the Fortran linkage conventions are adhered to, the routines could be called using other languages. However, since the routines are designed for a Fortran calling program, Fortran will be used when the calling program is being considered.

MPARITH

This package of programs is designed for multi-precise integer arithmetic on any computing system having a Fortran compiler.

Each multi-precise integer X is represented by using a Fortran array and an extra word called the length of X. If the array is denoted by N(I),I=1(1)K, then:

abs(X)=N(1)+2**m*N(2)+2**(2*m)*N(3)+...2**((K-1)*m)*N(K)

while the length of X, denoted by the Fortran variable LX, has the value K or -K, according as X is positive or negative. The length of X=0 is taken to be zero. Each N(I) satisfies:

$0 \le N(I) < 2**m$.

The number m and a base 2**m must be specified for each computer system depending on the word length used by the Fortran compiler on that system. The number m should be chosen as the largest such that the square of every m-bit integer is given exactly by the Fortran compiler. For the CDC 6000 Fortran compiler, m is 24. For the IBM system /360, m is only 15. Furthermore, if one is to use the multi-precise square root subroutine m must be even. In that case m would be 14 for the IBM /360. The specification of m and 2**m is accomplished in the main program by inserting the Fortran statements:

```
COMMON IBASE,NBITS,IROOT
IBASE=2**m
NBITS=m
IROOT=2**(m/2)
```

It should be noted that the least significant bits of a multi-precise integer using this representation are contained in N(1); the least significant bit is the right-most bit in N(1). The most significant bits are in N(K); the most significant bit in the integer is the left-most bit in N(K). The average user of the MPARITH routines would be unconcerned about this arrangement of bits but it is important in an examination of the algorithms used.

The requirement that only m bits be used in each word allows the sum and product of any two words in this form to be represented in a single word. This requirement simplifies the algorithms for the basic operations as will be seen in

Chapter III. The value 2**m which is stored in IBASE is greater by 1 than the maximum integer that can be represented using m bits and this will prove to be useful when the algorithms are examined.

MPARITH consists of twenty routines, four of which are subsidiary to the others and which would not generally be called by a user program. A complete description of how to use the remaining sixteen routines appears in Appendix A. Fourteen of the routines are examined in this paper and the operations carried out by these routines are as follows:

i) addition - adds two multi-precise integers.

ii) subtraction - subtracts one multi-precise integer from another.

iii) multiplication - multiplies two multi-precise integers.

iv) division - performs a division where the dividend and divisor are multi-precise integers.

v) division - performs a division with a multi-precise integer dividend and a fullword integer divisor.

vi) square root - finds the square root of a multi-precise integer and the resulting remainder.

vii) transmit routine - moves one integer into another.

viii) shifting routine - multiplies or divides an integer by 2**K.

ix) input routine - reads a multi-precise integer from data cards as a base 10 number and converts it to internal form.

x) output routine (printer) - prints a multi-precise integer as a base 10 number.

xi) output routine (card punch) - punches a multi-precise integer onto cards.

xii) greatest common divisor - finds the greatest common divisor of any two multi-precise integers.

xiii) power routine - given three multi-precise integers B, E, and M, this routine finds the integer R congruent to B**E (mod M).

xiv) Jacobi symbol - finds the value of the Jacobi symbol $\left(\frac{N}{K}\right)$ where N is a multi-precise integer and K is a fullword integer.

In addition to these routines, there is a linear equation solving routine and a sample mainline program which tests integers for primality. These routines are described in Appendix A.

MPACK

This package of routines is designed specifically for the IBM System/360 series of computers. The System/360 Assembler language is used throughout, except for the routines which perform input and output. The input and output routines are written partly in the Fortran language and partly in Assembler.

Each multi-precise integer X is represented in a Fortran array which can be denoted by N(I),I=1(1)K. Any integer X such that:

$$2^{**}(32^{*}K)-1 < ABS(X) \leq 2^{**}(32^{*}(K+1))-1$$

will require (K+2) words in the array N for its representation. A byte is the smallest addressible unit of a 360 computer and 4 bytes make up a full word. That is, each element of a Fortran array requires four bytes of memory. For a multi-precise integer, the absolute value of N(1) is always four times the number of words of N required for the integer (exclusive of N(1) itself). If the multi-precise integer being represented is positive, location N(1) is positive. That location is negative if the integer being represented is negative. The remaining (K+1) words of N contain the representation of X as follows:

X=2**(32*K)*N(2)+2**(K-1)*N(3)+.....+N(K+2)

where each N(I) is a string of thirty-two binary bits to be treated as an unsigned binary number. If more than (K+2) words have been allotted for X in N, the remaining words beyond word (K+2) are unused. Of course, they may be used if the value of X is increased.

This representation makes use of all thirty-two bit positions in a word to represent binary digits. Although the leftmost bit in each word is normally the sign bit for fixed-point values, this is not the case in this representation. The bits used are right-justified in as many words as necessary starting at the second word of the array. Thus, the least significant bit is the right-most bit in word N(K+2).

MPACK consists of routines which perform the fourteen operations described above for MPARITH. In addition to these routines, there is a comparison routine to test which of two multi-precise integers is algebraically greater. There are also several subsidiary routines. A complete description of the use of these routines appears in Appendix A.

CHAPTER III

Algorithms and Programming Techniques

For the MPARITH and the MPACK routines, the sign of a multi-precise integer is indicated in the value used to record the length of the integer. This length is positive if the integer is positive; it is negative if the integer is negative. The algorithms employed by MPARITH and MPACK routines are actually algorithms for positive integers only. If negative integers are used in an operation, the routines retain this information but perform the operation on the absolute value of the integer. The sign of the result is determined and recorded as part of the length of the result. Therefore, the algorithms described here are for operations on positive integers only.

Throughout the following discussion IBASE and NBITS are the variables described in Chapter II.

Addition and Subtraction

In MPARITH, addition of two multi-precise integers is performed by calling subroutine MADD. This routine retains the signs of the operands but also sets their lengths to be positive. It calls the subsidiary routine INTADD if the two integers originally had like signs or it calls the routine INTSUB if the integers were of opposite sign. When control is returned from either of these routines to MADD the signs are set to their correct values before control is returned to the calling routine. Thus the operation of addition is actually addition or subtraction, depending upon whether the operands are of like or of opposite sign. For a subtraction, the user-called subtraction routine MSUB changes the sign of the second operand and calls MADD. Addition and subtraction are performed using base IBASE. The order in which the words of an integer are stored allows addressing with a simple subscript, which is also a DO-loop parameter.

In MPACK, addition and subtraction routines are set up in the same way as for MPARITH. The user-called routines are MPADDA and MPSUBA. Addition is performed using base 2**32. The logical add instruction and the logical subtract instruction of the IBM /360 computer are used extensively in these routines.

Multiplication

The MPARITH multiplication routine MULT calls the subsidiary routine IMULT. Multiplication is performed using base IBASE. Because of the way IBASE is chosen, the product of any two words in this base can always be represented in a full word. This allows easy separation of the part of the word which is less than IBASE from the part of the word which is to be carried to the next higher order word.

The MPACK multiplication routine MPMULA calls the subsidiary routine MPMUL. Multiplication is performed using base 2**32.

<u>Division</u>

Routines from both program packages perform division using a divide and correct technique similar to that described by Knuth (1).

The MPARITH division routine performs division using base IBASE. First of all it normalizes the dividend and divisor; that is, it performs multiplications by 2 until the highest order bit in the highest order word is 1. It then computes words of the quotient one at a time by performing divisions of two words of the partial remainder (in the first step this is the dividend) by a divisor of base IBASE. Not more than one correction is required to obtain the true quotient word after each such division. The division routine makes use of no other routines while performing a division. Other routines in the package are called to normalize the operands before division, to unnormalize the operands after division, and to check the computed quotient and remainder, and to make a final correction if necessary. The requirement that each word of the operands be of base IBASE allows a dividend of base IBASE**2 to be formed and to be divided by a divisor of base IBASE by performing a simple integer division.

The MPACK division routine uses double precision floating point values as dividend and divisor. First of all, a floating point value is formed from as many significant digits of the divisor as possible. This divisor is used throughout the division to approximate words of the quotient, the digits of which are extracted from floating point results. If necessary a correction is made at the end of each step. Each word of the quotient is either correct when formed or it is too large by 1.

Square Roots

The square root routines use a method similar to that used for the division routines. An approximation to one word of the square root is computed at each step and this approximation may be too large by 1. It is corrected if necessary and the current remainder is computed before the next step is performed.

Input Conversion

On input a conversion routine is required to convert the decimal digits of a multi-precise integer to the correct internal representation.

In MPARITH the integer being read is initially set to zero. Each data card is read as a string of characters (A format in Fortran). To determine the internal representation of the integer on the card, each digit is tested and the multiplication routine is called to multiply the current value of the integer by 10. The addition routine is then called to add the digit to the present value of the integer. Thus, for an integer of j digits, j calls to the multiplication routine and j calls to the addition routine are required.

In MPACK each data card is read as a string of characters. These characters are tested and the digits they represent are determined. A work vector is formed in which each element contains the integer value of eight successive digits from a data card. The digits are right-justified in these words. Consider a work vector W and the multi-precise integer 4371926347108926134.

W(2) is given the value 437.

W(3) is given the value 19263471.

W(4) is given the value 8926134.

W(1) is given the value 12 to indicate that the following 12 bytes of W contain an integer to be converted. This work vector is passed to a conversion routine. This routine forms the internal representation of the integer by setting the integer equal to W(2), multiplying it by 10**8, adding W(3), multiplying by 10**8, and then adding W(4).

This routine for conversion does not use any other routines in the package.

Output Conversion

The output of multi-precise integers follows the format described in Appendix A. The output routine in each package converts the multi-precise integer from internal representation to a string of characters which are printed using A format.

In MPARITH, a divisor 10**j is determined where j is the integer part of (NBITS*0.30103). The decimal digits of the multi-precise integer being converted are determined by successive divisions of the integer by 10**j. The division routine which uses a fullword divisor is called for each of these divisions. Each such division produces a remainder less than 10**j which therefore yields j digits of the result. These j digits are determined by performing j divisions by 10; each such division produces a remainder which is a digit of the result. The character representation of each digit produced is stored in a word of a work vector. Divisions are continued until the multi-precise integer being converted is zero. The original integer is not destroyed since it is copied into a work vector before conversion.

In MPACK the conversion of a multi-precise integer from its internal representation to a string of decimal digits is essentially the opposite of the conversion which is performed for input by MPACK. The integer to be converted is divided repeatedly by 10**8. Each division yields a remainder which is less than 10**8. Eight divisions of this remainder by 10 produce eight decimal digits which are converted to characters in preparation for printing. All divisions performed by this routine are accomplished without the use of any other MPACK routines.

Shifting

The shifting routines multiply and divide multi-precise integers by powers of 2. The result may be stored in the original location or it may be stored in the location reserved for another integer. In each package if the shift routine is called and a zero shift is specified, the integer to be shifted is copied into the location reserved for the result.

The MPARITH shift routine is MLSHFT. If a left shift of K positions is specified (a multiplication by 2**K), two values are determined which are used in shifting. The first value is the number congruent to K (modulo NBITS). It represents the number of positions by which all bits in each word of the integer must be shifted left. This shifting is performed by doubling each word of the integer this number of times, each time recording the carry bit and adding it to the next higher order word. Each resulting word is stored in the vector specified for the result. A second value, the integer part of K/NBITS is the number of zero words required as low order words of the result. These words are stored after the shifting has been performed.

If a right shift of K positions is specified (a division by 2**K), the method used is similar to the left shift above. The number congruent to the absolute value of K (modulo NBITS) is the number of positions by which all bits in each word of the integer must be shifted to the right. Let J represent this value. This shifting is accomplished by shifting the contents of each word to the left by (NBITS-J) positions. This is done by doubling each word (NBITS-J) times to form words of the result. Not all words of the integer are shifted in this way because a number of low order words of the original integer will not be part of the shifted integer. This number is the absolute value of K divided by NBITS. These words are left unchanged.

The MPACK shifting routine is DOSHFT. For a left shift, the shifting of bits of each word is performed by the shift left double logical (SLDL) instruction. In this way, the bits that are shifted out of one word remain in a register and can be used as the low order bits of the next higher order word of the result. After this shifting has been performed, the appropriate number of low order words are zeroed. If the number of words to be zeroed is less than eight, the words of the result are zeroed by performing a store from a register into a word of the result. If this number of words is eight or greater, an MVC instruction is used to zero eight words of the result at a time.

A right shift is similar to the above. Bits which will appear in the result are shifted by the shift right double logical (SRDL) instruction. The correct number of words are dropped entirely from the integer and are left unchanged.

Greatest Common Divisor

The MPARITH greatest common divisor routine uses a modified version of Euclid's algorithm. Suppose that the greatest common divisor of two integers X and Y is being sought. Any integer that divides X and Y will divide X-Y, X-2Y, and X-3Y. The number of divisions required by Euclid's algorithm is reduced by performing subtractions of this type. If any of these subtractions gives a negative result, it is restored to a positive value by adding the last value subtracted and subtractions are then performed with the operands interchanged. If three subtractions are performed of which none produces a negative result the numbers may be of greatly differing magnitudes and a division is performed. The subtractions and divisions are repeated in this way and divisions are not performed until three successive subtractions each yield a positive result. This method is more efficient than the standard form of Euclid's algorithm which uses divisions only.

The MPACK greatest common divisor routine uses the standard form of Euclid's algorithm.

CHAPTER IV

Memory Requirements of MPARITH and MPACK

The memory requirements of each program package can be examined in two separate categories; first the amount of memory required by the routines while they are executing, and second the memory required by a Fortran calling program. The latter will include the memory required for the instructions which make up the calling program and the memory required for the representation of each multi-precise integer in internal form.

Memory requirements of the routines

First of all, consider a basic set of routines from each program package. Suppose that the only routines included in this basic set are: addition, subtraction, multiplication, division (with multi-precise operands), assignment routines, and the input and output routines. All routines written in Fortran were compiled on the /360 computer by the Fortran H compiler using the highest level of optimization available. For the package MPARITH, approximately 11,500 bytes of memory are required for these basic routines; for the same routines from MPACK, approximately 6,200 bytes are required. In addition, there would be the required run-time routines, which are the same for each program package.

If all routines which are common to both packages are considered, then the memory requirement for the MPARITH routines is approximately 20,500 bytes; for the MPACK routines, slightly less than 10,000 bytes.

Memory requirements of the calling program

The amount of memory used by the Fortran statements of a calling program does not differ significantly in the two packages. For any particular operation or sequence of operations, the same number of subroutine calls is required in MPARITH as in MPACK. The parameter lists are generally longer in MPARITH but this does not cause a large difference in the amount of memory space required. There is, however, a significant difference in the amount of memory required to store a multi-precise integer.

For each package, every multi-precise integer is stored in a one-dimensional Fortran array. In each case, a fullword is required to store the length of the multi-precise integer. In addition, the MPARITH package requires that at most m bits of each word of the Fortran array be used where m is chosen such that the square of every m-bit number can be represented in a fullword (see Chapter II). Thus, this set of routines uses approximately half of each word of the array for storing the bits that form the integer. To be precise, for an IBM System/360 computer, the word length is thirty-two bits but m can only be 15. Thus when these routines are used on a /360, with m equal to 15, approximately 4.5 decimal digits may be stored in each word of the array. If the square root routine is used, m must be even. Thus, if m is equal to 14, this figure is decreased to about 4.2 decimal digits per word. Thus, to store a multi-precise integer of 200 decimal digits, with m equal to 15, forty-five words of an array are required plus an additional fullword to contain this length. Negative integers require exactly the same amount of memory as their absolute value, since they merely require that the length word be made negative.

The MPACK routines use all 32 bits of each word. In this way, approximately 9.6 decimal digits are stored in each word of an array which is being used for a multi-precise integer. A multi-precise integer of 200 decimal digits uses twenty-one words plus the additional word for the length. In this case the length will be 84, the number of bytes required in the array. As in MPARITH, the length word is made negative for negative integers and all such integers require the same amount of storage as their absolute value.

In summary, one full word is required to store the length of each multi-precise integer using either representation. Additional words are then required to store the integer itself as described above. Any integer requires at least as many words to be stored using the MPARITH routines as it does using the MPACK routines. It can be seen from the above that to store an integer the MPARITH package requires approximately twice as much memory as MPACK requires.

CHAPTER V

Timing of the Routines

Method Used

Extensive timings of most of the routines in each package were performed. This timing was carried out using an IBM System /360 model 65 computer. The method used is described in this chapter. The results of timings appear in Appendix C.

All Fortran routines used were compiled by the IBM Fortran H compiler, using the highest level of optimization available.

For most routines, integers of 12, 25, 50, 100, 200, 400, and 800 digits were used. For each of these lengths a number of calls were made to the subroutine under consideration. For example, to time the addition routine. Two 12-digit integers were added many times by the addition routine from the package MPARITH and the elapsed time was recorded. The same two integers were then added by the addition routine from the package MPACK and the elapsed time was recorded. These times were printed and the program then repeated these steps for two integers of each of the other sizes, each time performing several additions and recording the CPU time used.

The interval timer, which is a hardware component was used to determine the time elapsed during the subroutine calls. This was accomplished by using two subroutines from the Fortran library. The first is $TRTM which is called using the statement:

```
          CALL $TRTM(T)
```

When called, it places the current value of the interval timer into the variable T. The second routine, $TPTM, is called by the statement:

```
          CALL $TPTM(T)
```

It computes the difference between the current value of the interval timer and the value of T which is passed to the routine. This is the amount of central processing unit (CPU) time that has been used for this job since the call to $TRTM was made.

The number of calls to each subroutine is determined by the characteristics of the interval timer. The resolution of the interval timer is .017 seconds. This means that any reading taken from it can be in error by up to this amount. That is, if a call is made to the routine $TRTM exactly .005 seconds after the timer has been updated and if an operation lasting .01 seconds is performed, then a reading taken from the interval timer by $TPTM at that time will be the same as the initial reading taken by $TRTM. This will indicate that no CPU time elapsed while the operation was being performed. Of course, this can have the opposite effect and an operation which takes less than .017 seconds to perform may be given a time of .017 seconds. In addition to this source of error, there is another related one. Every time a program which is executing is interrupted in order to give the CPU to some other program, a reading is taken from the interval timer in order to keep track of the time used by the interrupted job. Each such reading can be in error by up to .017 seconds. Also, a small amount of CPU time is used by a job each time it relinquishes or regains control of the CPU in this way. Therefore, in order to obtain timings which are as accurate as possible, the number of interrupts must be as low as possible. To achieve this, the programs which performed all timings described here were run when there were no other user programs executing. Even using this method, there are still some interrupts from the system. For example, it updates the console display screen every few seconds. These system interrupts introduce a small additional source of error and can cause a timing to differ by more than the .017 seconds accounted for above.

To determine the amount by which a timing differed from one run to the next, the addition routine was used. The addition routine from each package was called 5,000 times for each of seven different lengths of integers. For each size of integer, two results are obtained, one from the MPARITH addition routine and one from the MPACK addition routine. Thus, for each run of this program, fourteen different timings are produced, and for 5,000 calls these timings are between 1 and 13 seconds. Running this program five times produces 70 readings, five in each of the fourteen classes. In twelve of these fourteen classes, all five timings in the class were within .033 seconds of every other timing in that class. This represents two updates of the interval timer. In the remaining two classes, there was one timing in each set of five which differed from the others by more than this amount. It is reasonable to assume, therefore, that most readings taken from the interval timer under these conditions for a given operation can be reproduced to within .033 seconds. The actual reading taken from the timer may be very slightly higher than the true execution time because of the time used by a program which is interrupted by the system. However, this is probably the best value that can be determined. Thus, provided that each timing is of at least 3.4 seconds duration, the value given for it by $TRTM and $TPTM can be expected to be within 0.033 seconds of this best value. This is within 1 per cent, a tolerable amount of error. As the timings become longer, they appear to still be within 0.033 seconds of one another.

Almost all timings included here were of at least 3.4 seconds duration. The number of calls made to a routine for the purposes of timing was chosen so that the elapsed CPU time for that number of calls would be at least 3.4 seconds. Where possible, for a given operation and a given size of integer, the same number of calls was made to a routine from MPARITH as to the corresponding routine from MPACK. However, in some cases, this would have necessitated using a prohibitive amount of time for a certain number of calls from one routine in order to ensure that a time of 3.4 seconds was produced for the same number of calls to the other routine. In that case, a different number of calls was made to each routine.

Timing of routines in this way was performed for every operation for which it was feasible. In the case of operations which require only one operand, such as finding the square root of an integer, one integer of each of the lengths specified above was used. The CPU time required for some operations is entirely dependent upon the length of the integer operands. However, for other operations, the CPU time may vary drastically even for two integers which have the same number of decimal digits. For example, consider the difference in the amount of CPU time required to find the greatest common divisor of two integers each of 400 decimal digits, using Euclid's algorithm. If the two integers have greatest common divisor equal to 1, it will take much longer than it will take to find the greatest common divisor of two 400 digit integers, one of which is twice the other. Thus, for routines of this type, a comparison of CPU time may be made between routines for a particular set of data; however, no general formula can be established.

Analysis of Timings

There are two ways of looking at the timings obtained. Again, consider the addition routines. First of all, a table can be constructed which gives the amount of time required for the addition of two multi-precise integers of the given number of digits. Such tables are useful because it can be seen at a glance how the routines compare when performing the same operation on the same data. Such a table might look like the following Table 5.1.

# of digits	# of additions	MPARITH	MPACK
12	28000	6.856	3.744
25	24000	6.739	3.594
50	20000	7.421	3.944
100	16000	7.604	4.410
200	12000	8.736	5.175
400	8000	9.818	6.057
800	4000	9.085	5.641

Table 5.1. Time required for the addition of two positive integers.

Tables similar to this appear in Appendix B.

The second way of analyzing the results of timing is more detailed and is useful in attempting to estimate the amount of CPU time required to perform a certain operation. Where possible, formulae which produce this time have been established using tables of data such as Table 5.1 above. A description of how they were determined and of how to use them appears below. The formulae appear in Appendix B.

By using a linear or quadratic equation, the time required for operations can be approximated quite closely if this time depends only on the number of machine words used by each integer operand. For example, the addition of two integers requires a basic amount of time for calling the addition routine and manipulating the signs of the integer operands. This time is independent of the length of the integer operands. The time required to add these integers depends on the number of words in each integer. The formula to compute the time required for the addition of two multi-precise integers each of which is represented using w words is approximated by a linear equation of the form:

```
T(w)=a+b*w
```

The formula for the multiplication of two integers each of which is w words in length is approximated by a quadratic equation of the form:

```
T(w)=a+b*w+c*w**2
```

In each of these equations, a, b, and c are constants which are computed using a least squares approximation on data from tables such as Table 5.1. Even though timer readings were taken for integers of seven different lengths, only five of these are used for the approximation. These are the timings for integers of lengths 12, 25, 100, 200, and 400 digits. Since an integer represented using MPARITH generally requires more words for its representation than the same integer using MPACK, the number of words (w) occupied by a given integer differs. For the integers of 12, 25, 100, 200, and 400 digits, w is 3, 6, 23, 45, and 89 respectively for MPARITH. For MPACK, w takes on values 2, 3, 11, 21, and 42 respectively. This is exclusive of the word used to store the length of the integer.

The formulae presented below and in Appendix B give the time required, in seconds, for one call to a routine which performs an operation on integers requiring a certain number of words for their representation. Assume that this number is represented by w1 for an integer using the MPARITH representation, and that it is represented by w2 for an integer using the MPACK representation. Thus using the figures in Table 1, the formula for the addition of two positive integers of w1 words using the MPARITH addition routine is:

$$T(w1)=0.000212+0.000011*w1 \qquad (1)$$

The corresponding formula for the MPACK addition routine is:

$$T(w2)=0.000103+0.000016*w2 \qquad (2)$$

Care must be exercised when comparing these equations. It appears that for sufficiently large integers, the second equation will yield a larger value than the first. However, this is not true.. Recall that only fifteen bits are used in each word of an MPARITH integer on an IBM/360 computer, and 32 bits are used for an MPACK integer. This means that for a given integer, the value of w1 to be substituted into equation 1 is approximately twice as large as the value of w2 to be used in the second equation. In fact, for the purposes of comparison, it is accurate to say that if w2 is known for a given integer, then a close approximation to w1 is given by

$$w3=32/15*w2$$

For each pair of equations, a third equation can be produced by substituting this value of w1 into equation 1. For the pair of equations above, this new equation is:

$$T(w3)=0.000212+0.000024*w2 \qquad (3)$$

This should be interpreted to mean that if it is known that an integer requires w2 words to be represented under MPACK, the time required for the operation under consideration using MPARITH is given approximately by this equation. This allows a direct comparison between equations 2 and 3 by simply examining the coefficients. However, to compute an estimate of the time required to perform an operation for specific integers using either package, it is best to find the exact value of w for the package and to substitute this into either of equations (1) or (2). For each routine which was timed, three equations similar to those above are presented. These equations appear in Appendix B.

There is one additional source of error in the timings discussed here. In every timing taken, the amount of time used by the Fortran DO-loop and the amount of time used by \$TRTM and \$TPTM is included in the time required by the routines themselves. It is difficult to establish this time exactly since a vacuous DO-loop is eliminated during optimization by the Fortran H compiler. However, the time required to execute such a loop one million times using Fortran G is less than 8 seconds. This means that one pass through the loop will require about .000008 seconds. For the Fortran H compiler, it is certainly less than this. Even though this time cannot be established exactly, its contribution to the overhead time required for a call to a routine is small and it can be ignored.

It will be noted that operations require a different amount of time depending on the signs of the integers involved. Where this time is significant formulae are provided for each possible sign combination.

It can be seen from the formulae in Appendix B that there is no case for which an MPARITH routine is faster than the corresponding MPACK routine. This is due to three factors:

i) Most MPACK routines are written in the /360 Assembler language and make use of the features provided by that machine. The MPARITH routines are written in Fortran.

ii) In general, more machine words are used for the representation of an integer when using MPARITH than when using MPACK. This means that more words must be addressed each time an MPARITH routine is called.

iii) In some cases, an algorithm , used by an MPACK routine is superior to the algorithm used by the corresponding MPARITH routine. One example of this is the conversion routines used to convert integers from decimal to binary digits and from binary to decimal digits for input and output. This is not to say that there are no MPARITH routines which use algorithms or programming techniques superior to those used for MPACK routines, but where MPARITH routines are superior it is not enough to compensate for the difference in execution time caused by i) and ii).

CHAPTER VI

Other Considerations

Ease of Use

The subroutine calls to the MPACK routines are, in general, easier to set up than the calls to corresponding MPARITH routines. This is because one argument in the parameter list is required for each multi-precise integer in a call to an MPACK routine, whereas two arguments are required to specify each multi-precise integer in any call to a routine in MPARITH. Because of this, parameter lists are always shorter for a call to an MPACK routine, even though there are a few routines which require more work vectors in MPACK than in MPARITH. This does not mean that MPACK requires more workspace, since the output routine in each package requires a work vector which is dimensioned to have at least as many words as there are digits in the integer being printed, and this large work vector can satisfy the work vector requirements of all other routines.

The use of a separate word for length with MPARITH is not without merit. In some routines it is advantageous to change the bit pattern in a multi-precise integer without changing its length and this is easier if the length is a separate word. (A use of this can be seen in routine POWER.) In addition to this, it allows the user to test the length of a multi-precise integer by testing a simple variable instead of the first word of an array.

Portability

The portability of any program package between different computer systems is an important factor in an analysis of its overall merits. The package MPARITH is written in Fortran and, with minor modifications, can be used on any computer system for which a Fortran compiler is available. In fact, the routines in MPARITH were originally tested using a CDC computer and, for the purposes of this study, they were adapted quite easily for use on an IBM /360 computer. This is a very important feature of this package, since it means that this multiple precision integer arithmetic package can be used with most computers.

MPACK, on the other hand, is restricted to IBM System /360 computers which have the standard and floating-point instruction sets.

Possible Improvements

After extensive testing, both program packages appear to be error-free in the sense that, given proper data, the routines do what it is claimed they will do. The input and output routines for each package and the Jacobi symbol routines from MPARITH produce some error messages if the user has done something incorrect. However, for other routines, if a user supplies incorrect data such as a length word which has been altered by mistake, or a negative length where only a positive length should be used (such as in the square root routine), the result is not easy to interpret. In the case of an MPARITH routine, execution is terminated with a numbered STOP statement or a, system abend code, depending on the error. The first alternative is not serious since the user merely needs to consult a listing of the routines to determine where the error occurred. With some thought, he can then determine how it was caused. If an abend occurs, there is no immediate indication of exactly where the error occurred and a core dump may be required to determine this. Once the instruction that caused the error is located, the user must determine what he may have done wrong to cause the program to abend at this point. For the MPACK routines, a system abend is the only result of a serious user error. Thus, error diagnostics for both packages are minimal, but they are slightly better in MPARITH than in MPACK. This is one possible improvement that could be made to both sets of routines since there are some cases where, with a little extra effort, an error could be detected and an error message printed, or an error code set.

Of course, in each of the above cases, even though the user may have done something wrong, it may have been minor enough to allow the routines to execute but to produce incorrect results. This cannot possibly be detected by the routines.

An Additional Application of These Routines

In addition to being useful in number theory problems where large integers are manipulated, the routines can also be used to retain high precision in the manipulation of floating-point operands. This has been done in a simulation of the machine language for a CDC 6600 computer using the IBM /360 computer. Floating-point values retain a precision of forty-eight binary bits in the CDC 6600 computer which could be represented in a double precision floating-point word in the /360 computer. However, each exponent is 11 bits which cannot be represented easily in the same word. Therefore, the multi-precise integer routines in MPACK were used to perform operations on the floating-point values. The routines computed the significant digits of a result and the exponent of the result was determined by the calling routine.

CHAPTER VII

Summary and Conclusions

The MPARITH package of routines was written so that it could be used on any computer system which has a Fortran compiler. On the other hand, the MPACK package was written specifically for the IBM System /360 series of computers. Because of this, the MPARITH routines are more widely available and this is an important advantage. For most routines, the algorithms employed are quite similar in each package. However, because of the programming language used, all MPACK routines execute faster and use less computer memory than the corresponding MPARITH routines.

In light of the above, deciding which package to use under given conditions is an easy matter. If a user has access to an IBM System /360 computer, the MPACK routines should be used; otherwise, the MPARITH routines should be used on the computer available.

APPENDIX A

This appendix contains a description of how to use the routines in the program packages MPARITH and MPACK. All routines are designed to be called from a Fortran calling routine.

Instructions for the Use of MPARITH Routines

Of the twenty routines in this package, four are subsidiary to the others and are not called by the user. They are denoted by an asterisk in the list below.

MADD	MLSHFT
MSUB	MPIN
*INTADD	MPOUT
*INTSUB	MPNCH
MULT	*MPOXYZ
*IMULT	MGCD
SDIV	POWER
MDIV	JACOBI
MSQRT	LES
MOVE	PT

The use of these routines is as follows:

Addition Subroutine MADD

```
CALL MADD(LA,LB,LC,A,B,C)
```

This causes the sum of the multi-precise integers A and B of lengths LA, LB, to be computed and stored in the array C and the length of this sum to be stored in the word LC. The declared dimension of C must exceed the larger of LA and LB. Any 2 or 3 of the arguments A, B, or C may coincide.

Subtraction Subroutine MSUB

```
CALL MSUB(LA,LB,LC,A,B,C)
```

This call has the same effect as:

```
CALL MADD(LA,-LB,LC,A,B,C)
```

As in MADD, any 2 or 3 of the arguments may coincide.

Note: In both MADD and MSUB, decimal integers of length 1 may be substituted for A or B. Thus:

```
CALL MADD(1,LB,LB,389,B,B)
```

increments B by 389. Integers of this form must be no greater than 2**m, where m is the number of bits which is used in each word of a multi-precise integer. Furthermore, if -389 is to be used in this way, the first argument in the above call must be -1 and the fourth argument must be 389.

Multiplication Subroutine MULT

```
CALL MULT(LA,LB,LC,A,B,C)
```

This causes the product of A and B to be computed and stored in C. The declared dimension of C must exceed LA + LB. C must be neither A nor B. Decimal integers of length 1 may again be substituted for A or B.

There are two division routines. The simpler one, SDIV, has a divisor of length 1. The other division routine, MDIV, is general.

Single Precision Division Subroutine SDIV

```
CALL SDIV(LN,1,LQ,LR,N,D,Q,R)
```

This causes the multi-precise integer N to be divided by D, which is an integer of no more than m bits. The quotient is stored in Q and the remainder is stored in R. That is, N=D*Q+R, where R has the same sign as Q.

Multi-precise Division Subroutine MDIV

```
CALL MDIV(LN,LD,LQ,LR,N,D,Q,R,WS)
```

This divides N by D. The quotient is a multi-precise integer and it is stored in Q; the multi-precise remainder is stored in R. R is given the same sign as Q. The array WS is a work area large enough to contain a copy of N. N and D are not altered by this routine.

Square Root Subroutine MSQRT

```
CALL MSQRT(LN,LS,LR,N,S,R,WS)
```

This finds S and R such that:

N=S**2+R where $0 \le R < 2*S + 1$

Transmitting Subroutine MOVE

```
CALL MOVE(LM,LN,M,N)
```

This causes a copy of the multi-precise integer M to be placed in the array N.

Shifting Subroutine MLSHFT

```
          CALL MLSHFT(K,LA,LB,A,B)
```

This causes a shifted copy of A to be sent to B. The amount of the shift is K and the direction is left if K is positive and right if K is negative. This is equivalent to multiplying A by 2**K.

Card Decimal Input Routine MPIN

```
          CALL MPIN(LN,N,LWS,WS)
```

This causes a multi-precise integer to be placed in the array N and its length to be placed in LN. This integer has been punched in decimal form according to the following format. The integer may be punched on one or more cards. Column 1 of the first card is the sign column and is either +, -, or blank. The digits in columns 2-71 are read and converted. If column 72 is not blank, columns 1-71 on the next card are read. This continues until a card with a blank in column 72 is read. Except for the first column of card 1, all blanks in columns 1-71 are ignored. Any other non-numeric character in these columns causes an error message to be printed. If a number is punched by mistake in column 1 of card 1, the routine reports this and attempts to read the next card. WS is an array whose length LWS should exceed the maximum possible value of LN.

Line Printer Decimal Output Routine MPOUT

```
CALL MPOUT(LA,A,ND,W,Z)
```

This prints the decimal equivalent of the multi-precise integer A of length LA. Seven words of 10 decimal digits each are printed on each line for as many lines as are required. ND must exceed the number of decimal digits expected in A. Workspace W must be dimensioned of length ND and Z must be at least of length LA.

Punched Card Decimal Output Routine MPNCH

```
CALL MPNCH(LA,A,ND,W,Z)
```

This punches one or more cards with the decimal equivalent of the multi-precise integer A in the format described under MPIN. ND, W, and Z have their previous meanings under MPOUT.

The following routines are useful in number theory problems and they make use of the routines above to carry out computations.

Greatest Common Divisor Subroutine MGCD

```
CALL MGCD(LA,LB,LG,A,B,G,X,Y,Z,W)
```

This computes the greatest common divisor of A and B and stores it in the array G. The length of G is stored in LG. The workspace arrays X,Y,Z,W have lengths exceeding the larger of LA and LB. A and B may be positive or negative.

Modular Subroutine POWER

```
        CALL POWER(LB,LE,LM,LR,B,E,M,R,Q,N,WS)
```

This computes the remainder on division by M of B raised to the E-th power. This result is stored in R and its length in LR. The workspace arrays Q,N,WS must have lengths exceeding the largest of LB, LE, LM. Note that B, E, and M are multi-precise integers of any length. If the exact value of B**E is required, LM may be chosen sufficiently large so that M exceeds B**E. This routine destroys E by shifting it left. If E is needed for the future it must be saved by the calling routine before POWER is called.

Jacobi Symbol Subroutine JACOBI

```
        CALL JACOBI(LN,N,K,J,WS)
```

This subroutine calculates the value of the Jacobi symbol $\left(\frac{N}{K}\right)=\pm 1$ or 0; N is a multi-precise integer and K has length 1. This routine destroys K.

Linear Equation Solver LES

CALL LES(LA,LB,LX,LY,LD,A,B,X,Y,D,X1,m,Y1,Q,R,Z)

This finds the greatest common divisor D of the multi-precise integers A and B and also a pair of multi-precise integers X and Y for which:

AX-BY=D where D is greater than 0.

A and B may have either sign. The last six parameters represent workspace. They are arrays whose lengths should exceed the larger of LA and LB.

Primality Tester PT

This mainline routine was supplied with the package as an illustration. It tests a sequence of multi-precise positive integers M for primality. The data deck structure is as follows:

The first card (or cards) contains the decimal digits of the first M, next follow cards for the odd prime factors of M-1. Then comes a blank card to signal the end of data for this M. The same data now follow for the succeeding M's, ending with a second blank card to signal the end of all data.

Instructions for the Use of MPACK Routines

The routines in this package are very similar in function to the routines described above. Where there are important differences in the way these routines operate, they are pointed out below.

Addition Routine

```
CALL MPADDA(B,C,A,W)
```

The routine MPADDA computes the sum of the multi-precise integers stored in arrays B and C. This sum is then stored in the array A. W is a work vector and it must be at least large enough to contain the integer A. Any 2 or 3 of the arguments A, B, C may coincide.

Subtraction Routine

```
CALL MPSUBA(B,C,A,W)
```

The routine MPSUBA subtracts the integer C from the integer B. The result is stored in A. W is workspace at least as large as A. Any 2 or 3 of A, B, or C may coincide.

Multiplication Routine

```
CALL MPMULA(B,C,A,W)
```

When called as indicated, the routine MPMULA computes the product of B and C and stores this result in A. W is a work vector at least large enough to hold the integer stored in C. Any 2 or 3 of A, B, or C may coincide.

Division Routine (Multi-precise Divisor)

CALL MPDIVA(B,C,A,R,W)

This call to the routine MPDIVA causes the multi-precise integer B to be divided by the multi-precise integer C. The quotient is stored in A and the remainder is stored in R. (i.e.B=A*C+R). W is a work vector at least as large as B. If the dividend is not to be retained, B, R, and W may coincide.

Division Routine (Fullword Divisor)

CALL MPDIVN(B,C,A,R,W)

This call to the routine MPDIVN computes the quotient resulting from the division of B by C. This quotient is stored in A. B is a multi-precise integer and C is a fullword integer of either sign. W is workspace at least large enough to contain B.

Card Input Routine

CALL MPINA(A,M,W)

This call to the routine MPINA causes a multi-precise integer to be read from cards (I/O unit 5) and to be stored in the array A. M is an integer value which represents the maximum number of words of workspace that will be required. This is greater by 1 than the number of decimal digits in the number divided by 8. An error message is issued if this number is too small. Each integer punched on data cards must have the following form:
The first card has a +, -, or blank in column 1. The digits in columns 2-72 of this card are read and converted. If the integer is continued on subsequent data cards, column 72 must be non-blank. The subsequent data cards contain digits in columns 1-71 and column 72 is non-blank if the integer is continued on the following card. Blanks are ignored. Any other non-numeric characters cause an error message to be printed, followed by program termination.

<u>Line Printer Output Routine</u>

```
        CALL MPOUTA(A,M,WA,WB)
```

This call to the routine MPOUTA prints the multi-precise integer A on the line printer (I/O unit 6). M is a fullword integer variable representing the maximum number of digits contained in the integer. The workspace vector WA must be dimensioned at length M at least. WB is a second work vector at least as large as A. Integers are printed with 70 digits per line for as many lines as are required.

<u>Card Punch Output Routine</u>

CALL MPNCHA(A,M,WA,WB)

This routine is similar to MPOUTA above except that the integer is punched on cards. A sign and 70 digits appear on the first card; 71 digits appear on each subsequent card, if required. Continuation characters are punched as required. Thus, cards produced by this routine are acceptable for input to the routine MPINA.

<u>Assignment Routine</u>

CALL MPASNB(A,B)

This call causes the multi-precise integer stored in B to be copied into A.

<u>Assignment Routine (Fullword Operand)</u>

CALL MPASNA(A,M)

This call causes the fullword integer M to be stored in A as a multi-precise integer. M may be of any magnitude and sign provided by the /360 computer.

Comparison Routine

```
J=MPCOMA(A,B)
```

MPCOMA is a function subprogram used to algebraically compare two multi-precise integers. The variable J is assigned one of three possible values as a result of the above statement.

J=-1 means that $A < B$

J=0 means that $A = B$

J=+1 means that $A > B$

Square Root Routine

```
CALL MPSQRT(A,B,R,WA,WB)
```

When called as indicated, MPSQRT finds the square root of A and stores it in B. It also finds the remainder and stores it in R (i.e. A=B**2+R). Work vectors WA and WB must be at least as large as the expected square root. The integer A must be positive when passed to the routine.

Shifting Routine

```
CALL DOSHFT(K,A,B)
```

This call causes the multi-precise integer to be shifted by K positions and the result of this shift to be stored in B. That is, B=2**K*A. K is a fullword integer and it may be positive (resulting in A being multiplied by 2**K), or negative (resulting in a division of A by 2**K).

Greatest Common Divisor

```
CALL MPGCDA(A,B,C,WA,WB,WC)
```

This routine finds the greatest common divisor of A and B and stores the result in C. WA and WB are work vectors at least large enough to contain the larger of A and B. A and B may be of either sign.

Jacobi Symbol Routine

```
CALL MPJAC(A,K,M,W)
```

This subroutine calculates the value of the Jacobi symbol $M=\left(\frac{A}{K}\right)= \pm 1$ or 0. A is a multi-precise integer and K is a fullword integer. W is at least as large as A.

Power Routine

```
CALL POWA(A,B,C,R,WA,WB,WC,WD)
```

This computes the remainder on division by C of A raised to the B-th power. A, B, and C are multi-precise integers. This routine destroys B by shifting it left. WA, WB, WC, and WD are each at least large enough to hold the largest of A, B, and C. If A**B is required, C can be made large enough so that C is greater than A**B.

Appendix B

Formulae and Tables Showing Timing of the Routines

Tables showing the results of timing of most MPARITH and MPACK routines appear in this appendix. Where feasible, equations which give the CPU time required for an operation are presented. These equations are based on the number of machine words used to represent each integer. The tables are self-explanatory. Equations appear in the following form:

MPARITH	$T(w1)=0.000212+0.000011*w1$	(1)
MPACK	$T(w2)=0.000103+0.000016*w2$	(2)
ADJUSTED	$T(w3)=0.000212+0.000024*w2$	(3)

Proper interpretation of these equations is important. It is explained in detail on pages 32 and 33.

Addition

The MPARITH addition routine is MADD; the corresponding MPACK routine is MPADDA.

# of digits	# of calls	MPARITH	MPACK
12	28000	6.856	3.744
25	24000	6.739	3.594
50	20000	7.421	3.944
100	16000	7.604	4.410
200	12000	8.736	5.175
400	8000	9.818	6.057
800	4000	9.085	5.641

Table B.1 Time required for the addition of two positive integers.

Equations

MPARITH	$T(w1)=0.000212+0.000011*w1$	(1)
MPACK	$T(w2)=0.000103+0.000016*w2$	(2)
ADJUSTED	$T(w3)=0.000212+0.000024*w2$	(3)

Addition (continued)

# of digits	# of calls	MPARITH	MPACK
12	28000	7.954	5.408
25	24000	7.738	5.009
50	20000	7.971	5.225
100	16000	8.753	5.475
200	12000	10.267	6.190
400	8000	11.781	7.105
800	4000	10.533	6.473

Table B.2 Time required for the addition of two integers, the first integer positive and the second integer negative.

Equations

MPARITH	T(w1)=0.000237+0.000014*w1
MPACK	T(w2)=0.000155+0.000017*w2
ADJUSTED	T(w3)=0.000237+0.000030*w2

The table of times and the equations for the addition of two integers, the first negative and the second positive, is almost exactly the same as the above.

Addition (continued)

# of digits	# of calls	MPARITH	MPACK
12	28000	7.155	4.044
25	24000	6.956	3.860
50	20000	7.638	4.160
100	16000	7.754	4.576
200	12000	8.869	5.275
400	8000	9.901	6.140
800	4000	9.119	5.674

Table B.3 Time required for the addition of two integers, both negative.

Equations

MPARITH	$T(w1)=0.000222+0.000011*w1$
MPACK	$T(w2)=0.000114+0.000016*w2$
ADJUSTED	$T(w3)=0.000222+0.000024*w2$

Subtraction

The MPARITH subtraction routine is MSUB; the corresponding MPACK routine is MPSUBA.

# of digits	# of calls	MPARITH	MPACK
12	28000	10.100	6.223
25	24000	9.551	5.724
50	20000	9.501	5.824
100	16000	9.984	5.940
200	12000	11.165	6.589
400	8000	12.397	7.355
800	4000	10.816	6.623

Table B.4 Time required for the subtraction of one positive integer from another positive integer.

Equations

MPARITH	T(w1)=0.000313+0.000014*w1
MPAK	T(w2)=0.000184+0.000017*w2
ADJUSTED	T(w3)=0.000313+0.000030*w2

The table of times and the equations for the subtraction of one negative integer from another negative integer are almost exactly the same as the above.

Subtraction (continued)

# of digits	# of calls	MPARITH	MPACK
12	28000	9.036	4.576
25	24000	8.603	4.276
50	20000	8.986	4.509
100	16000	8.886	4.859
200	12000	9.684	5.508
400	8000	10.450	6.290
800	4000	9.402	5.774

Table B.5 Time required for the subtraction of a negative integer from a positive integer.

Equations

MPARITH	T(w1)=0.000290+0.000011*w1
MPACK	T(w2)=0.000132+0.000016*w2
ADJUSTED	T(w3)=0.000290+0.000024*w2

Subtraction (continued)

# of digits	# of calls	MPARITH	MPACK
12	28000	9.302	4.909
25	24000	8.819	4.593
50	20000	9.185	4.759
100	16000	8.986	5.075
200	12000	9.784	5.691
400	8000	10.500	6.390
800	4000	9.452	5.791

Table B.6 Time required for the subtraction of a positive integer from a negative integer.

Equations

MPARITH	T(w1)=0.000299+0.000011*w1
MPACK	T(w2)=0.000145+0.000016*w2
ADJUSTED	T(w3)=0.000299+0.000024*w2

Multiplication

The MPARITH multiplication routine is MULT; the corresponding MPACK routine is MPMULA. The time required does not depend significantly on the sign of the operands involved; therefore, times and equations are presented only for positive operands.

	MPARITH		MPACK	
# of digits	# of calls	time	# of calls	time
12	16384	13.162	102400	23.695
25	4096	8.986	25600	8.104
50	1024	6.839	6400	6.007
100	256	6.273	1600	4.293
200	64	5.791	400	3.444
400	16	5.541	100	3.395
800	4	5.441	25	3.295

Table B.7 Time required for the multiplication of two positive integers.

Equations

MPARITH	T(w1)=0.000157+0.000079*w1+0.000043*(w1**2)
MPACK	T(w2)=0.000156+0.0000094*w2+0.000019*(w2**2)
ADJUSTED	T(w3)=0.000157+0.000170*w2+0.00019*(w2**2)

Division

The MPARITH division routine (using a multi-precise integer dividend and divisor) is MDIV; the corresponding MPACK routine is MPDIVA.

Three tables and three sets of equations are given for these routines; however, all results presented are for positive integer operands. The difference in time required for operands of negative or of mixed sign is not significant. The number of digits refers to the number of digits in the dividend.

	MPARITH		MPACK	
# of digits	# of calls	time	# of calls	time
25	1024	4.809	10240	2.729
50	512	4.193	5120	4.376
100	256	4.343	2560	4.360
200	128	5.691	1280	4.609
400	64	9.318	640	6.240

Table B.8 Time required for the division of an integer by a divisor of one-fifth as many digits.

Equations

MPARITH	T(w1)=0.00529+0.00015*w1+0.000016*(w1**2)
MPACK	T(w2)=0.00030+0.00009*w2+0.000003*(w2**2)
ADJUSTED	T(w3)=0.00529+0.00032*w2+0.000073*(w2**2)

Division (continued)

	MPARITH		MPACK	
# of digits	# of calls	time	# of calls	time
25	1024	4.559	10240	5.408
50	512	4.310	5120	4.193
100	256	4.892	2560	4.143
200	128	6.223	1280	4.909
400	64	11.165	640	7.322

Table B.9 Time required for the division of an integer by a divisor of one-half as many digits.

Equations

```
MPARITH     T(w1)=0.00600+0.00003*w1+0.000021*(w1**2)
MPACK       T(w2)=0.00049+0.000055*w2+0.000005*(w2**2)
ADJUSTED    T(w3)=0.00600+0.00007*w2+0.000095*(w2**2)
```

Division (continued)

	MPARITH		MPACK	
# of digits	# of calls	time	# of calls	time
25	1024	4.509	10240	4.193
50	512	3.544	5120	3.578
100	256	3.444	2560	3.062
200	128	4.077	1280	3.345
400	64	5.891	640	4.992

Table B.10   Time required for the division of an integer by a divisor of four-fifths as many digits.

Equations

```
MPARITH     T(w1)=0.00434+0.00022*w1+0.000009*(w1**2)
MPACK       T(w2)=0.00045+0.000032*w2+0.0000034*(w2**2)
ADJUSTED    T(w3)=0.00434+0.00047*w2+0.000039*(w2**2)
```

Division (short)

The MPARITH division routine which uses a fullword integer divisor is SDIV; the corresponding MPACK routine is MPDIVN. The number of digits refers to the number of digits in the dividend.

# of digits	# of calls	MPARITH	MPACK
12	15000	3.644	3.411
25	15000	5.425	3.711
50	15000	8.320	5.491
100	15000	15.475	7.122
200	15000	28.404	11.215
400	5000	18.138	6.772
800	5000	35.427	12.696

Table B.11 Time required for the division of a multi-precise integer by a fullword integer divisor.

Equations

MPARITH	T(w1)=0.000125+0.000039*w1
MPACK	T(w2)=0.000164+0.000028*w2
ADJUSTED	T(w3)=0.000125+0.000084*w2

Square Root

The MPARITH square root routine is MSQRT; the corresponding MPACK routine is MPSQRT.

# of digits	# of calls	MPARITH	MPACK
12	6400	28.404	5.358
25	3200	20.950	7.338
50	1600	16.690	5.541
100	800	20.251	5.458
200	400	21.599	6.623
400	200	33.995	8.536
800	100	62.766	13.761

Table B.12 Time required to find the square root of an integer.

Equations

```
MPARITH     T(w1)=0.003748+0.000443*w1+0.000016*(w1**2)
MPACK       T(w2)=0.000048+0.000531*w2+0.000012*(w2**2)
ADJUSTED    T(w3)=0.003748+0.000946*w2+0.000073*(w2**2)
```

Move Routine

The MPARITH routine which copies one integer into another is MOVE; the corresponding MPACK routine is MPASNB.

# of digits	# of calls	MPARITH	MPACK
12	60000	4.959	3.794
25	60000	5.608	4.027
50	60000	6.972	4.676
100	60000	9.418	5.807
200	60000	14.360	8.070
400	60000	24.228	12.763
800	60000	44.096	22.148

Table B.13 Time required to set one integer equal to another.

Equations

MPARITH	T(w1)=0.000071+0.0000037*w1
MPACK	T(w2)=0.000056+0.0000037*w2
ADJUSTED	T(w3)=0.000071+0.0000080*w2

Shifting (Left)

The MPARITH shifting routine is MLSHFT; the corresponding MPACK routine is DOSHFT. Shifts of 1,14,....,199 positions were performed on each integer. The number of calls indicates the total number of calls to the shifting routine for a given size of integer (i.e. 24000 calls means that the integer was shifted by 1 position 1500 times, by 14 positions 1500 times, etc.)

# of digits	# of calls	MPARITH	MPACK
12	33600	11.415	6.323
25	28800	14.527	5.541
50	24000	18.653	6.207
100	19200	27.489	6.273
200	14400	37.889	6.390
400	9600	48.273	7.987
800	4800	47.507	5.874

Table B.14 Time required to multiply an integer by a power of 2 by shifting it to the left.

Equations

MPARITH	T(w1)=0.000177+0.000055*w1
MPACK	T(w2)=0.000143+0.000016*w2
ADJUSTED	T(w3)=0.000177+0.000116*w2

Shifting (Right)

Shifts are performed as described on the previous page.

# of digits	# of calls	MPARITH	MPACK
12	33600	9.402	3.877
25	28800	12.497	3.528
50	24000	16.773	4.010
100	19200	25.958	4.676
200	14400	36.658	5.275
400	9600	47.324	6.922
800	4800	46.875	5.5412

Table B.15 Time required to divide an integer by a power of 2 by shifting it to the right.

Equations

MPARITH	$T(w1)=0.000111+0.000054*w1$
MPACK	$T(w2)=0.000075+0.000015*w2$
ADJUSTED	$T(w3)=0.000111+0.000115*w2$

Input

The MPARITH input routine is MPIN; the corresponding MPACK routine is MPINA.

# of digits	# of calls	MPARITH	MPACK
12	1536	19.502	16.274
25	768	18.104	7.971
50	384	20.717	4.060
100	192	30.584	4.293
200	96	49.171	3.641
400	48	87.460	4.360
800	24	164.054	5.857

Table B.16 Time required to input integers from cards and to convert these integers to internal form.

Equations

MPARITH	T(w1)=0.004843+0.001928*w1+0.000208*(w1**2)
MPACK	T(w2)=0.008353+0.000892*w2+0.000025*(w2**2)
ADJUSTED	T(w3)=0.004843+0.004112*w2+0.000946*(w2**2)

Output

The MPARITH line printer output routine is MPOUT; the corresponding MPACK routine is MPOUTA. Tables and formulae for the punched card output routine are almost exactly the same as the following.

# of digits	# of calls	MPARITH	MPACK
12	1536	5.325	4.992
25	768	5.059	4.160
50	384	5.225	3.894
100	192	6.140	3.910
200	96	8.137	4.110
400	48	12.264	5.175
800	24	20.584	7.188

Table B.17 Time required for the conversion and output of integers.

Equations

```
MPARITH     T(w1)=0.000678+0.000846*w1+0.000023*(w1**2)
MPACK       T(w2)=0.000477+0.001502*w2+0.000025*(w2**2)
ADJUSTED    T(w3)=0.000678+0.001805*w2+0.000103*(w2**2)
```

Greatest Common Divisor

The MPARITH greatest common divisor routine is MGCD; the corresponding MPACK routine is MPGCDA. The time required to find the greatest common divisor of two integers is dependent on the value of the integers, not on their lengths. However, the following table shows the amount of CPU time required in the case where two integers with greatest common divisor equal to one are used.

# of digits	# of calls	MPARITH	MPACK
12	512	15.741	3.511
50	128	31.050	6.090
100	64	42.565	7.155
200	32	57.541	9.834
400	8	50.868	7.122

Other Routines

The CPU time required by the routines POWER and JACOBI from MPARITH, and by the corresponding routines POWA and MPJAC from MPACK is highly dependent on the data used. Results of timings are not supplied here but timings were carried out and in each case the MPACK routines were faster. The same algorithms are used by the power routine and by the Jacobi routine in each package. In each case other routines in the package are called. This is why the MPACK power and Jacobi symbol routines are faster than the corresponding MPARITH routines.

Bibliography

1. Knuth, D. E., Seminumerical Algorithms. Reading, Mass., Addison-Wesley, 1969.

2. Ehrman, J. R., Logical Arithmetic on Computers with Two's Complement Binary Arithmetic. Communications of the ACM, XI (July, 1968), 517-520.

3. Grosswald, E., Topics from the Theory of Numbers. Macmillan, New York, 1966.

4. Krishnamurthy, E. V., and Nandi, S. K., On the Normalization requirement of divisor in divide-and-correct methods. Communications of the ACM, X (Dec., 1967), 809-813.

5. Lehmer, D. H., and Weinberger, P., Multiprecise Integer Arithmetic Package, University of California (Berkeley), July 1969.

6. Luke, Y. L., and Sommerville, D. I., Multiple Precision Computations on the IBM-360. Interim Technical Report, Midwest Research Institute, August, 1967.

7. Mifsud, C. J., A Multiple-precision Division

Algorithm. Communications of the ACM, XIII (November, 1970), 666-668.

8. Stein, M. L., and Pope, D. A., Multiple-precision Arithmetic. Communications of the ACM, III (Dec., 1960), 652.

9. Stein, M. L., Divide and correct methods for multiple-precision division. Communications of the ACM, VII (August, 1964), 472-474.