Microprogrammed significance arithmetic: A perspective and feasibility study*

by C. V. RAMAMOORTHY and M. TSUCHIYA

The University of Texas at Austin
Austin, Texas

INTRODUCTION

This study is an attempt to evaluate the feasibility of microprogrammed routines for monitoring significant digits in the numerical result of digital computers in real time. The first part is tutorial and, in the second part, microprograms for two methods of significance arithmetic are designed and evaluated.

The digital computer is a finite device that has a limit in representing numerical values; thus, in general, it maintains the most significant part of true values for computation. Therefore, errors are nearly always involved in numerical computations on a digital computer due to the finiteness of representation, truncation and round-off. Unfortunately, whether they are in the original data or generated during computation process, errors are “hidden” in the computed results. One cannot readily determine how many digits of the computed result represents the true value unless rigorous error analyses such as those by Wilkinson1 are performed.

Significance arithmetic was suggested as a means for monitoring the significant digits in the numerical result of digital computers. If an algorithm for significance arithmetic is effectively implemented, it will provide an indication of the number of significant digits that the result retains. It will also be a powerful tool for debugging and testing numerical method for computers by tracing numerical significance at every stage of computation. Thus a study of error propagation of computational methods will be made easier. It will also pinpoint the computational stage where a large loss of significance occurs so that the computational method could be modified analytically to prevent the loss of significance and to improve the accuracy of the computed result.

SIGNIFICANCE ARITHMETICS

A number of studies have been made in the area of significance arithmetic in an effort to trace the significant digits accurately and to preserve the given number of significant digits of operands. Gray and Harrison2 used a normalized arithmetic with an index of significance in which part of a word is used as an index to indicate explicitly the number of significant digits of the numerical value. In this method, arithmetic operations are performed in the same manner as any normalized arithmetic except an extra computation, although simple, is necessary. Metropolis and Ashenhurst3 used unnormalized arithmetic in which numerical values are stored in an unnormalized form preserving only the significant digits. Unnormalized arithmetic is relatively simpler than other methods, but since operands in this method are not normalized, their leading zeros* must be compared with each other as well as with those of the result to adjust and preserve significance. Moore4 proposed interval arithmetic that used two values, upper and lower bounds, to indicate the range of true value. Error bounds are computed in this method to indicate significance instead of the number of significant digits. The range for true value, i.e., the difference of the two bounds, tends to become large with a loss of significance as the number of arithmetic operations increases. Wilkinson1 provides excellent error analyses for digital computation. His analyses, however, take a “pessimistic” approach and do not offer sharp bounds.

In this study, the first two methods are considered: namely, the normalized arithmetic with an index of significance and the unnormalized arithmetic. They are provided with simple algorithms for determining

* This work was supported by National Science Foundation Grant GJ-492 and in part by the Joint Services Electronics Program under contract AF-44620-71-C-0091.

* In Reference 3, the term “leading digit” is used since, in negative numbers, leading zeros may be represented by leading ones. In this paper, the term “leading zeros” is used because leading ones in negative number, in fact, represent “leading zeros.”
the number of significant digits explicitly at each stage of computation.

The microprogrammed implementation of these algorithms are shown to be quite simple. Most importantly the computational overhead is also proved to be quite small.

DESCRIPTION OF METHODS

When the floating-point arithmetic is implemented in a computer, a normalized method is generally employed. The normalized floating-point arithmetic, however, does not provide a facility to indicate explicitly the number of significant digits in the result. It generates results with no indication as to how many of their digits might represent the true value. Since the error analysis in general is not easy, the significant digits in the result are not easily determined. Thus the idea of significant digit arithmetic or significance arithmetic is introduced.

In the significance arithmetic, the number of significant digits in the results is computed at every stage of computation and, in some cases, an adjustment is made to preserve the original significance. To realize the significance arithmetic, a number of unorthodox arithmetic methods have been proposed and experimented. Among these two methods, namely the unnormalized arithmetic by Ashenhurst and the normalized arithmetic with an index of significance by Gray and Harrison, were chosen for this study. In this section, the schemes used in the two methods are briefly described and illustrated with examples. The numbers in the example are represented as follows:

- \((\text{exponent}, \text{mantissa})\) for unnormalized number;
- \((\text{exponent, index, mantissa})\) for normalized number with index.

Significant digits of the operands are indicated by boldface types so that if an operand is 1.732051, for example, the first five digits 1.7320 are significant. In the example a double-length arithmetic register is assumed.

Addition

\[193.764 + 8453.12 = 8472.50\]

(a) Index method

\[
(3, 0.193764) + (5, 0.845312) = (5, 0.847250)
\]

(b) Unnormalized method

\[
(5, 0.001937) + (6, 0.084725) = (6, 0.084725)
\]

Result in double-length register

Subtraction

\[193.764 - 98.123 = 95.641\]

(a) Index method

\[
(3, 0.193764) - (2, 0.981230) = (2, 0.956410)
\]

(b) Unnormalized method

\[
(5, 0.001937) - (3, 0.098123) = (5, 0.000956)
\]

Result in double-length register

Multiplication

\[193.764 \times 34.1526 = 6617.54\]

(a) Index method

\[
(3, 0.193764) \times (2, 0.341526) = (4, 0.661754)
\]

(b) Unnormalized method

\[
(5, 0.001937) \times (3, 0.034153) = (6, 0.006619)
\]

Result in double-length register; exponents added; smaller index used.

From the collection of the Computer History Museum (www.computerhistory.org)
Division

8453.12 + 350.916 = 24.0887

(a) Index method

(4, 0.845312) + (3, 0.350916) = (2, 0.240887)

Smaller index is used for quotient.

Quotient normalized and rounded, exponent adjusted, index unchanged assuming an extra digit is not good.

(Pessimistic Approach)

(b) Unnormalized method

(5, 0.084531) + (5, 0.003509) = (4, 0.002409)

Difference of exponents and quotient.

Significance of quotient is adjusted to be the same as that of the divisor (that contains less significant digits).

As intuitively seen in the examples, both methods preserve and indicate the same amount of significance. It can be shown that both methods are in fact equivalent methods that use different number representations. Both methods use a carry out of the highest digit and a number of normalizing shifts (or equivalently a number of leading zeros in case of unnormalized arithmetic) as criteria for determining the number of significant digits for the result. Since they use the same criteria for determining an amount of significance, it may be concluded that they yield the same result.

A case for fully significant operands

A special case is where both operands have full significance, i.e., the value of operands happens to be precisely representable within the limit of the exponent-and-mantissa representation. If this is the case, the significance computation changes. Under the significance computation rule, a large gain in the number of significance is possible. Given two operands \( A = (e_A, s_A, m_A) \) and \( B = (e_B, s_B, m_B) \) where \( e_A \) and \( e_B \) are exponents, \( s_A \) and \( s_B \) significance, \( m_A \) and \( m_B \) mantissas of \( A \) and \( B \), respectively, the modified rule is as follows:

- addition/subtraction: \( s_A + |e_A - e_B| \); 
- multiplication/division: \( s_A + s_B \).

Clearly, in order to keep the result with increased significance a double-length mantissa is necessary. If it is not available, the result is rounded and its significance is reduced.

Unfortunately the increased significance in a double-length mantissa is possibly reduced to single-length significance by rounding, etc., in a subsequent computation. If a result with increased significance represented by a double-length mantissa is used for a computation with an operand with a single-length mantissa, the result will have the significance at most that of the latter operand, i.e., the significance is reduced to the single-length. Thus the implementation of the rule does not necessarily guarantee a result with higher significance.

Another problem associated with the special case is the detection of operands that represent a value exactly. Since the rule applies only when both operands have full significance, i.e., when they represent exact values, operands must be checked for their full significance at every arithmetic operation if the rule is to be applied. The test for full significance can cost a few steps and, if the test is performed on every operand used in arithmetic operations, it can be expensive in terms of computation time. Moreover, the probability of computing two operands with full significance is conceivably very low mainly due to rounding and truncation operated on intermediate results. The inclusion of the algorithm for the special case, therefore, will reduce computational efficiency of significance arithmetic and may not be practical unless a larger number of significance is highly desirable.

MICROPROGRAMMED SIGNIFICANCE ARITHMETIC: AN IMPLEMENTATION

The following is a brief description of microprograms for arithmetics in the unnormalized method and the normalized method with an index of significance. In this paper microprograms for addition and multiplications are considered and described. Division may be considered a multiplication in "reversed" order; its flow diagrams are presented. Many results observed in multiplication also apply to division.

The following assumptions are made in designing microprograms for significance arithmetic:

1. A double-length arithmetic register that performs addition/subtraction in two clock cycles (the register for higher order part of operand will be called the upper accumulator, and the other, the lower accumulator);
2. One's complement addition;
3. One's complement representation for negative numbers.
In the subsequent discussion, numbers in parentheses refer to the box bearing the same number in the flow diagram. Two operands are referred to variables "A" and "B" for convenience. The mantissas of the two operands are denoted by \( m_A \) and \( m_B \), and the exponents \( e_A \) and \( e_B \), respectively.

**Normalized addition/subtraction with an index of significance**

Addition and subtraction are carried out in a conventional manner using ones complement (Figure 1). The exponents of the two operands are compared first (1). If the difference of the two exponents is too large, i.e., if it is greater than the number of bits in mantissa, then the smaller operand is too small to have any effect as far as the mantissa and significance of the sum are concerned (2). Otherwise addition/subtraction with an index of significance commences.
by one (4). After binary-point alignment, the indicies of the two operands are compared and the smaller is kept as the basic index for the result (5). The larger of the two exponents is preserved as the basic exponent for the result (6). After the adder is cleared, the two operands are transferred to the adder for addition (7), and an overflow condition is checked (8). If there has been an overflow, the result is shifted one bit to the right while increasing the index of significance and exponent by one (9, 10, 11). If there has been no overflow, the result is normalized. To normalize, the highest mantissa bit is tested against the sign bit (12); if they are equal, the mantissa is shifted one bit to the left (13); if they are not equal, the result has been normalized. For every normalizing shift, both the index of significance and exponent are decreased by one (14, 15). The normalized result is rounded for the final result (16). The rounding is accomplished by adding one to the highest-order bit of the lower accumulator.

Where appropriate registers and data buses among them are provided in the computer architecture, more concurrency in microprogram execution is realized. If micro-operations can be executed simultaneously without conflict, they can be coded into a microinstruction for concurrent executions (Figure 2). With an adequate computer organization, the microprogram in Figure 1 can be reorganized to take advantage of parallel data paths for a faster execution. Figure 2 shows the microprogram for normalized addition with an index of significance in parallel execution. Numbers in the node of Figure 2 refer to the micro-operations bearing the same number in Figure 1. Nodes 1a, 8a, and 12a are pseudo-nodes used as a convention for the parallel task graph representation of microprograms and cost no time in execution. The “exclusive-OR” sign ⊕ between two edges out of a node indicates that either one of the edges is traversed exclusively of the other. Note that steps 3 and 4 of the microprogram in Figure 2 do not interfere with each other and can be executed concurrently rather than sequentially. Similarly steps 9, 10, and 11 as well as steps 13 and 14 can be executed in parallel.

Unnormalized addition/subtraction

The unnormalized addition procedure, nearly identical to the normalized addition, does not require normalizing the final result. Note that the normalizing loop consisting of steps 12, 13, and 14 in Figure 2 is deleted in Figure 3 for unnormalized addition. A major difference is that unnormalized addition does not require computation of significance indices since the unnormalized mantissa represents only significant digits.

Another difference is the test for an insignificant operand (2). The number of mantissa bits is compared with the difference of two exponents in the normalized method; in the unnormalized method, it is compared with the sum of the difference of two exponents and the number of leading digits of the smaller operand. If the sum exceeds the number of bits in mantissa, the smaller operand is too small to have any effect on the result.

Possible parallelism in the unnormalized procedure is limited to steps 7 and 8 for adjustment of overflow (Figure 4). These steps are executed infrequently because unnormalized operands usually do not cause overflow. Although the unnormalized method lacks concurrent micro-operations, its microprogram is simpler and faster than the normalized method which requires the normalizing procedure.
Normalized multiplication with an index of significance

Normalized multiplication with an index of significance uses a conventional method: the lowest multiplier bit is examined as the multiplier is right shifted and the multiplicand is accumulated to form the product (Figure 5). The index of significance is set to the smaller of the two indices.

The exponent of two operands are added first to form the basic exponent for the product (1). The index of
significance for the product is the smaller of the two (2). (See algorithm in the previous section.) The adder is cleared before the multiplication procedure commences.

The multiplier is scanned bit by bit from right to left for multiplication. If the lowest-order bit of multiplier is one, the multiplicand is added to the partial product formed in the adder; if zero, no action takes place (4, 5). The multiplier and the partial product are shifted one bit to the right for a multiplication at the next higher magnitude (6, 7). A counter counts the number of multiplier bits that have been scanned (8, 9). The result is checked for normalization (10). If a normalizing shift (at most one left shift) is necessary (11), the exponent and index are both decreased by one (12, 13). The result is rounded (14).

Clearly; steps 1, 2, and 3 are potentially parallel in the microprogram (Figure 6). Among these operations, the second that determines the smaller index requires the longest time. Similarly steps 6, 7, and 8 can be executed simultaneously. This saving is significant because these steps are loop embedded; a reduction of one step can save many cycles. For a parallel execution of steps 11, 12, and 13, at least two decrementing counters are necessary for decreasing exponent and index of significance by one.

Unnormalized multiplication

The procedure used for unnormalized multiplication is essentially the conventional multiplication procedure except the magnitude of mantissa is compared and leading zeros of the larger mantissa are counted to initialize multiplication. It takes $m$ steps to multiply two $m$-bit operands. An improvement to the method suggested by Metropolis and Ashenhurst by “automatically” adjusting the leading zeros in the result to that of the smaller mantissa after the multiplication procedure is completed. Where there is a carry out of the highest order significant bit (not necessarily the highest-order bit of register because of the unnormalized representation) significance is gained by one bit.

The leading zeros of the larger mantissa are counted and saved in a counter which is used later as a procedure counter for counting the number of loops executed for the multiplication proper (1). The sum of exponents is reduced by the number of leading zeros of the larger mantissa plus one (2). This subtraction enables the “automatic” adjustment of leading zeros. The subtraction by one from the sum of exponents compensates the result that is not shifted one bit to the right at the last multiplication proper loop.

Multiplication is performed in a conventional manner using shift and add operations (4-9). Note that since
the procedure counter initially contains the number of leading zeros the multiplication loop is executed for the number of bits that represent a value in mantissa (6, 7). After the last multiplication loop is executed, the result contains either the same number of bits or one bit more to represent a value than the operand with the smaller mantissa. This is a major improvement over the original method in which a significance gain in multiplication is suppressed to preserve the number of leading zeros.

An overflow condition is checked after completing multiplication in case both operands happen to be normalized and a carry out of the highest order bit has occurred (10). If there is an overflow, the result is shifted one bit to the right and the corresponding exponent is increased by one (11, 12). A probability of the occurrence of overflow is presumably quite low, however, and these steps will be seldom executed. After an overflow condition is checked, the result is rounded to form the unnormalized product (15).

Parallellism in unnormalized multiplication may be realized in several ways (Figure 8). Steps 4 and 5 in Figure 7 are executed concurrently, or more specifically, they are constrained by step 4 execution time, and step 5 is performed at a last phase before step 4 is completed without incurring extra time. Steps 7 and 8 can be also executed concurrently when step 7 is necessary. Steps 10 and 11 can be performed concurrently if two shift registers are provided. Similarly, steps 13 and 14 can be concurrently executed. The execution time reduction by concurrency in steps 7 and 8 and steps 10 and 11 is significant because these loop steps are executed as many times as there are significant digits. A small reduction of micro-operation steps in the loop, therefore, can be a large saving in overall execution time.

EVALUATION AND COMPARISON OF METHODS

The microprogrammed algorithms for significance arithmetic are simulated on a step-by-step basis. They are compared and evaluated with regard to speed,
simplicity of microprogram implementation, and parallel processability. The speed of microprogram is measured in terms of the number of steps required to complete the task. The simplicity of microprogram is a capability of being represented compactly in a microprogram memory. The parallel processability is a feature that takes advantage of parallel computer structure for faster computation. The mathematical aspect of the algorithms is discussed elsewhere.\textsuperscript{2,6}

**Addition/subtraction**

The procedure for addition/subtraction is similar in both methods. This is readily observed in the microprogram in Figures 1 and 3. The microprogram for the unnormalized arithmetic, however, is simpler because it does not handle the index of significance. It is also faster especially when the microprograms are executed sequentially one micro-operation at a time for the same operands. This is mainly because the unnormalized arithmetic algorithm does not require the final normalization procedure. Normalization which involves shifting of mantissa and adjusting exponent (and index, in case of the index method) can slow down the procedure if there are many digits to be normalized.

Another reason that the unnormalized method is faster is that it seldom generates mantissa overflow because of the unnormalized representation. A carry out of the highest bit of the larger operand (not necessarily the highest bit of mantissa) is accommodated in the leading zero space. This implies that the steps 7 and 8 of Figure 3 for a corrective shift and adjustment of exponent for overflow will be rarely executed. The occurrence of mantissa overflow depends on the operands: more specifically, it depends on the number of significant digits kept and the sign of operands. Hence in general the frequency of the occurrence of mantissa overflow is unpredictable. The comparative advantage of the unnormalized method over the index method for mantissa overflow, therefore, cannot be easily measured or determined.

Figure 9—Normalized division with an index of significance

Figure 10—Unnormalized division
The normalized arithmetic with an index of significance is relatively slower than the unnormalized method when the addition/subtraction microprograms are executed sequentially because it must compute the index of significance. The additional steps for index computation can be costly especially when there is a large amount of change in significance as a result of the computation. Consider a subtraction of two close values, for instance, where a loss of a large number of significant digits occurs. Clearly the mantissa of the computed result needs a large number of shifts for normalization, and for every normalizing step, the index of significance must be reduced by one to account for the loss of significant bit before it is readjusted by a number of significant bits recovered by the normalization.

The same argument holds for parallel execution of microprograms except for the last instance. If micro-operations of a microprogram are arranged and executed to take full advantage of multiple data paths for the maximum concurrency of micro-operations, the index and exponent adjustments can be performed concurrently provided that two additional arithmetic registers that accommodate the index and the exponent are available. The parallelism in microprogram will, therefore, reduce the disadvantage of the index method over the unnormalized method by eliminating the sequential execution of index and exponent adjustments to improve execution speed.

**Multiplication**

Assuming that the multiplication procedure is carried out sequentially in the Interdata 4 on two 60-bit operands, the microprogram for the index method takes about 467 cycles while that for the unnormalized method requires about 491 cycles. The multiplication procedure used, in both microprograms is essentially the same: register shift and addition of multiplicand as required. It takes m steps for the m-bit mantissa and each of m steps in turn consists of several micro-operations: a test of bits in mantissa against the sign bit, an addition of multiplicand into a partial product in the adder, a right shift of the partial product and of the multiplier, and an increment of the procedure counter by one. An obvious reason that the unnormalized method requires more time is that the unnormalized mantissa representation has more bits than the normalized number representation with an index. If the mantissa of the unnormalized representation is the same size as the normalized representation with an index, the multiplication procedure will take about 453 cycles. This reduction of cycles is due to fewer micro-operations executed for scaling leading digits of the two mantissas.

While both methods appear to take approximately the same number of steps and hence the same amount of time, the unnormalized method can be faster if the number of significant bits of one of the operands is small. If the number of significant bits for one of the operands is 34 instead of 44, for instance, then the total number of cycles required for the unnormalized method is 441 compared to 491. This reduction results from an increase in the number of leading bits which is due to the decrease in the number of significant bits. The number of leading zeros is counted and used as the procedure counter so that the multiplication proper is performed only on the significant bits. In other words, if k is the number of significant bits then only k multiplication steps are necessary. It is evident from the microprogram of Figure 5 that the larger is the number of leading bits (or the smaller is the number of significant bits), the smaller becomes the number of cycles required because the leading-zero scaling requires fewer cycles than a multiplication step.

The discussion so far is based on the sequential execution of microprograms. What would happen, then, if the microprogram can be executed in parallel or, more specifically, if the horizontal microinstruction format is available so that more than one micro-operation can be initiated at the same time? Assuming that a sufficient number of micro-operations (in our case, four or five would be sufficient) can be specified in a microinstruction, the microprograms can possibly be rearranged to take advantage of parallelism in execution. The index method in the rearranged microprogram takes 316 cycles which is the reduction of cycles by 32 percent over the sequential microprogram. The unnormalized method in the rearranged microprogram requires 326 cycles which is a reduction of 34 percent over the sequential microprogram. The reduction rate by the parallel microprogram for both methods is nearly the same. Speed up in the method originally suggested by Ashenhurst is hindered by the extra steps for leading digits adjustment of the final result. The improved method that adjusts leading digits "automatically" eliminated the extra steps providing more reduction and better execution time.

In summary, both normalized multiplication with an index of significance and unnormalized multiplication equally benefit from parallel data paths in a computer architecture. The former can perform separate computations on exponent, index, and mantissa concurrently, and the latter can perform computations on exponent and mantissa.

When sequentially executed, unnormalized multiplication can be executed faster, especially where the number of significant digits is small since leading-zero counting requires less time than multiplying significant
digits. This peculiarity is nullified by the parallel execution of multiplication loop, however, so that the significance of operands does not affect execution time.

SOFTWARE VS. MICROPROGRAM

A Fortran program for a significance arithmetic algorithm has been implemented by Bright, Colhoun, and Mallory. It monitors computer arithmetic upon request by a program in execution and provides a number of significant digits retained in the computed result at every stage of computation. The initial significance value may be either specified by the user or determined by the program at compilation or execution time. Their program is sizable (approximately 1400 Fortran statements). Being a large interpretable program, it also takes considerable computing time. A computation that requires a few seconds in normal mode would take several minutes for completion in significance mode. They concede that, for a very large computation, the use of the program would be "inordinately costly."

Microprogramming appears to be a solution to the problems associated with the software implementation of significance arithmetic algorithms. It offers a compact representation of algorithms for a smaller memory space requirement. It controls internal data flow at gate-level so that redundant micro-operations for machine code execution may be eliminated for a higher performance of algorithms.

Compactness of microprogram also implies enhanced speed in execution. Micro-level redundancy that exists in machine code program is eliminated to reduce the number of micro-level steps. An elimination of one redundant microinstruction can be a saving of many clock cycles in program execution if the microinstruction is in a loop, and can yield a significant saving in overall computation cost.

Microprogramming enables fineness of control of internal data flows. In particular, where the horizontal microinstruction and appropriate data paths are available, algorithms may be implemented to take full advantage of existing parallel data paths, which is not possible in software. Although the microprogram may become somewhat complex in this case, drastic reduction in execution times accrue for frequently used routines.

To prove these points, algorithms for normalized arithmetic with an index of significance and unnormalized arithmetic were programmed both in the COMPASS assembly language for the CDC 6600 computer and the microprogramming language for the Interdata 4 computer. Although the assembly language provided relatively sophisticated operations such as floating-point arithmetic, floating-point to fixed point format conversion, and various conditional operations on many registers, the program for addition with an index of significance required more than 50 instructions and unnormalized addition approximately 30 instructions. In order for the fairness of evaluations the following modifications were assumed for the Interdata 4 computer:

1. the machine operates on 60-bit operands;
2. eight arithmetic registers;
3. eight index registers;
4. multiple-bit shift available so that any number of bits may be shifted in three cycles.

A microprogrammed emulator is defined for each machine code operation using the microprogramming language of the Interdata 4 computer. The microinstruction of Interdata 4 uses a vertical instruction format similar to a machine code and consists of primitive and simple operations such as register-to-register transfers, simple tests for a register condition, one-bit shift, etc. Most instructions are executed in one clock cycle of 400 nanoseconds; some in two clock cycles.

The assembly language programs for the two algorithms were translated into equivalent microprograms via a set of microprogrammed emulators. The emulator-translated microprogram for addition with an index of significance consisted of 113 microinstructions, and unnormalized addition 70 microinstructions. They were simulated on two operands A and B with the following characteristics:

1. \(|A| > |B|\);
2. index of significance of A is greater than that of B;
3. difference of the exponents of A and B is five;
4. result requires one-bit left shift for normalization.

The execution times of the addition with an index of significance and unnormalized addition on these operands are 119 and 69 cycles, respectively. The index method requires nearly twice as much time as the unnormalized method for bookkeeping operations such as extracting exponent, index and mantissa to perform computation, shifting, testing, etc., separately on each part.

The same algorithms were hand-coded directly in the Interdata 4 microprogramming code for a comparison with the software programs. In these directly-coded microprograms, much redundancy introduced in the emulator-translated microprograms was eliminated. The index method was coded in 78 microinstructions and the unnormalized method in 51 microinstructions. For the
same operands, the microprogram for addition with an index of significance takes 68 cycles whereas that for unnormalized addition uses 43 cycles. Comparing execution times of microprogrammed algorithm with that of software programmed algorithm, reduction realized by hand-coded microprograms is remarkable. Microprogramming the algorithms, about 43 percent of execution time is saved for the index method and about 37 percent is saved for the unnormalized method. In other words, the speed of performing significance arithmetic can be increased by approximately 40 percent by directly microprogramming.

Note that the execution of the Interdata microprograms is strictly sequential. If parallel data paths at gate-level are available, a better execution time reduction rate may be expected. Assuming that the microprograms coded for the Interdata 4 can be executed in "ideal" parallel, i.e., any possible parallelism in the microprograms may be realized, the index method for the two operands would take 35 cycles and the unnormalized method 27 cycles. Reductions realized in this case are nearly 70 percent and 61 percent, respectively, over software programmed algorithms, and approximately 49 percent and 37 percent over sequential microprograms. It may be anticipated that a further comparison of the parallel microprogram with a higher level language program for these algorithms would yield more than 80 percent reduction in execution time.

The estimated execution times of the other arithmetics is shown in Table I. Since execution time varies depending upon particular operands, an amount of significance, etc., times shown are approximate. The estimated execution times of normalized arithmetic with an index of significance are based on 44-bit mantissas and those of unnormalized arithmetic are based on 48-bit mantissas.

To illustrate the consequence of execution time reduction in significance arithmetic realizable by microprogramming, the overhead execution times for $n \times n$ matrix computation are estimated. Although the execution time for arithmetic operations varies according to the value of operands, an assumption is made that the execution times computed in the simulation and shown in Table I are about an average.

For an $n \times n$ matrix addition, there are necessary. When normalized addition with an index of significance is used it would take $119 \times n^2$ cycles for a software program and $68 \times n^2$ cycles for a sequential microprogram. For $n=10$, this would mean 11,700 cycles for a software program and 6,800 cycles for a microprogram. Furthermore, it is conceivable that the difference becomes much larger if a large number of bits must be shifted for normalizing. Unnormalized addition would take approximately $69 \times n^2$ cycles in a software program and $43 \times n^2$ cycles in a microprogram. For $n=10$, the software program for unnormalized matrix addition takes approximately 6,900 cycles and the microprogram 4,300 cycles.

An $n \times n$ matrix multiplication requires $n^3$ multiplications and $n^2(n-1)$ additions. This would mean that multiplying two $n \times n$ matrices using normalized arithmetic with an index of significance takes

$633 \times n^3 + 119 \times n^2(n-1)$ cycles in software

and

$467 \times n^3 + 68 \times n^2(n-1)$ cycles in microprogram.

Using unnormalized arithmetic, the same matrix multiplication would take

$655 \times n^3 + 69 \times n^2(n-1)$ cycles in software

and

$491 \times n^3 + 43 \times n^2(n-1)$ cycles in microprogram.

From these execution time estimates, it is readily observed that a small amount of reduction at microprogram level may be significant in the overall efficiency of computation.

**REQUIREMENTS FOR COMPUTER ORGANIZATION**

There are a number of desirable features in computer organization for an effective significance arithmetic. Some of them may coincide with the numerical analysts' wishes in the design of computers for numerical computations such as a large word-size, multiple-length arithmetic registers, etc. The requirements for computer organization here, however, are considered from the standpoint of microprogramming.
The first requirement in computer organization for significance arithmetic is a double-length register to preserve significance. Wilkinson shows that if only a single-length arithmetic register is available rounding errors propagates to widen error bounds and to reduce significance. Besides preserving significance, a double-length register may be used for double-precision arithmetic for greater accuracy.

A small arithmetic register large enough for computing exponent and index of significance is desirable for parallel execution of microprograms. With this register available, mantissa and exponent (and index of significance) can be computed simultaneously. For a faster normalized arithmetic with an index of significance, two such arithmetic registers may be necessary: one for exponent computation, the other for index computation. They are particularly useful in normalization in the index method. Normalization calls for left shifts of mantissa and, for every shift, reduction of exponent and index values. The two registers are used for the exponent and index computations concurrently with the mantissa shift operation to reduce overall execution time.

Two or more registers capable of shifting one bit at a time are necessary. They have many uses in both normalized and unnormalized arithmetics. They may be used in multiplication for a multiplier and partial product, in division for a dividend (and partial remainder) and a partial quotient, and in normalization. At least one of them should be able to shift its content in a group of bits at a time. Such register would reduce a time requirement in binary-point alignment for addition/subtraction in which a shift of many bits is often necessary.

Three or more mask registers for extracting any part of the register content are desirable. If they can be set to extract any part of the register content, the separation of exponent, index, and mantissa for separate computation is done faster.

A high-speed, alterable microprogram memory is desirable. It permits flexible specifications of control. When a new algorithm for significance arithmetic becomes available, it may be microprogrammed to replace the old microprogrammed algorithm without hardware rewiring. Thus, the alterable microprogram memory will prevent obsolescence of a machine with a small cost for remicroprogramming control specifications.

The microinstruction format should be horizontal. The horizontal microinstruction format permits specifying simultaneous activities of multiple data paths; it permits elemental control for efficient executions. Microprogram coding scheme must be simple, however, so that basic controls can be specified easily. If the microprogram operation code is encoded in the case of Honeywell H4200, microprogramming becomes somewhat restricted in terms of specifying data paths. Therefore, a more general microprogramming method and supporting tools such as microprogram interpreter, simulator, etc., are desirable for easier microprogramming.

The machine code instruction format should have a provision for specifying the number of significant digits. Basically the following machine code instructions are desirable for significance arithmetic: the input instruction for specifying a number of significant digits of input data; the significance arithmetic operation code separate from the regular arithmetic instruction; the output instruction to output the number of significant digits. The input instruction is a pseudo code that simply defines a constant and its significant digits. The significance arithmetic operation code may be the regular arithmetic instruction with its tag field set to indicate the significant arithmetic mode. The output instruction to extract and output the number of significant digits may be used to print out the number of significant digits. It may be also used to pass the number of significant digits to a routine that monitors significance and determines the next action such as a use of multiple-precision arithmetic, etc.

A possible format for a number representation in a 60-bit word for significance arithmetic with an index is shown in Figure 11a. It consists of two sign bits, one for exponent, the other for mantissa, eight bits for exponent, six bits for an index of significance, and 44 bits for mantissa. A format for the unnormalized number representation in a 60-bit word is shown in Figure 11b. It consists of a sign bit for exponent, a sign bit for
TABLE II—Comparisons of the Two Significance Arithmetic Methods

<table>
<thead>
<tr>
<th></th>
<th>memory requirement</th>
<th>execution speed</th>
<th>microprogram simplicity</th>
<th>parallelism in procedure</th>
<th>range of representable numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index method</td>
<td>larger</td>
<td>nearly the same</td>
<td>slightly complex</td>
<td>more parallelism</td>
<td>smaller</td>
</tr>
<tr>
<td>Unnormalized method</td>
<td>smaller</td>
<td>faster add/sub</td>
<td>simpler</td>
<td>less parallelism</td>
<td>larger</td>
</tr>
</tbody>
</table>

mantissa, ten bits for exponent, and 48 bits for mantissa. Clearly, for the same word size, the unnormalized format has a potential capacity for maintaining a larger number of significant digit (because of a larger mantissa) and representing a wider range of numbers (because of a larger field for exponent).

None of the requirements discussed so far is extreme or special-purpose oriented. Most of the desirable features are in fact provided by the existing, general-purpose computer except the alterable microprogram memory. A general speculation, however, is that the alterable, high-speed microprogram memory will be available at a reasonable cost in the near future with the advancement of memory technology. It will be used more widely as better microprogramming techniques are developed. In conclusion, the results suggest that the existing microprogrammable computers are reasonably well suited for microprogram implementing significance arithmetic.

CONCLUSION

Microprograms have been designed and shown in flow diagrams for the two significance arithmetic methods: normalized arithmetic with an index of significance and unnormalized arithmetic. The two methods are compared in terms of speed, simplicity in microprograms, and effects in computer organization.

Software implementation and microprogram implementation of significance arithmetic are also compared to justify the microprogrammed implementation. A software implementation of significance arithmetic appears to require a formidable amount of overhead computation.

The results of this study suggest many advantages for the microprogrammed implementation of significance arithmetic. Whereas the unnormalized method is considerably faster in addition than the index method, the index method is more suitable to a computer with parallel processable characteristics. The microprogram using computational parallelism for the index method is slightly simpler than that of the unnormalized method. Although the difference in execution time between the two methods is small, it can be significant over a long period of time especially when arithmetic functions are used more frequently in the computation.

Where memory requirements are of concern, the choice seems apparent. If the sizes of exponent and mantissa are fixed, the index method requires additional bits for the index of significance. If the mantissa consists of m bits, it requires \( \log_2 m \) additional bits for the index of every operand. For an n×n matrix, this would mean that at least \( \log_2 m \times n^2 \) additional bits are necessary.

If on the other hand the exponent, index, and mantissa are packed into a fixed word size, the size of exponent and/or mantissa fields becomes smaller for the index number representation. The index method in this case would suffer in the range of representable numbers due to smaller exponent and mantissa fields. A smaller size for mantissa would also mean less accuracy in representing numbers.

The observations and comparisons discussed above are summarized in Table II. The entry comments in the table are for the two methods relative to each other.

In conclusion, our studies have shown beyond doubt that significance arithmetic can be implemented in microprograms for a performance that costs little overhead over comparable regular floating-point arithmetic.

REFERENCES

1 J H WILKINSON
   Rounding errors in algebraic processes
   Prentice-Hall Englewood Cliffs N J 1963
2 H L GRAY C HARRISON JR
   Normalized floating-point arithmetic with an index of significance
   Proc of the Eastern Joint Computer Conference pp 244-248
3 N METROPOLIS R L ASHENHURST
   Significant digit computer arithmetic
   IRE Trans on Elec Computers Dec 1958 pp 265-267

* The notation \( \lceil x \rceil \) denotes the smallest integer greater than or equal to x.
4 R E MOORE
The automatic analysis and control of error in digital computation based on the use of interval numbers
Error in Digital Computation Vol 1 ed by L B Rall John Wiley & Sons Inc 1964

5 H S BRIGHT B A COLHOUN F B MALLORY
A software system for tracing numerical significance during computer program execution
AFIPS Proc of SJCC 1971 pp 387-392

6 C V RAMAMOORTHY M TSUCHIYA
A study of user-microprogrammable computers
AFIPS Proc of SJCC 1970 pp 165-181

7 R L ASHENHURST
Techniques for automatic error monitoring and control
Error in Digital Computation Vol 1 ed by L B Rall John Wiley & Sons Inc 1964