Roundings in Floating-Point Arithmetic

J. MICHAEL YOHE

Abstract—In this paper we discuss directed roundings and indicate how hardware might be designed to produce proper upward directed, downward directed, and certain commonly used symmetric roundings. Algorithms for the four binary arithmetic operations and for rounding are presented, together with proofs of their correctness; appropriate formulas for *a priori* error analysis of these algorithms are presented. Some of the basic applications of directed roundings are surveyed.

Index Terms—Data format, directed rounding, guard digits, ideal floating-point hardware, rounding.

I. Introduction

THERE has been a great deal of work done in the areas of rounding, floating-point arithmetic, and approximation of real numbers by computer representable numbers; it might seem as though we are beating a dead horse. However, to our knowledge, no manufacturer yet builds a computer that performs these functions ideally—at least, not by our definition.

In this paper, we sketch our definition of "ideal" floating-point hardware, present algorithms for software simulation of the four binary arithmetic operations and proofs of their correctness, present the appropriate formulas for *a priori* error analysis based on this design, and survey some of the basic applications of this arithmetic. Much of the material in this paper is treated in greater detail in [3], [6]-[8]. A thorough discussion of floating-point arithmetic, including some of the ideas presented here, can be found in Knuth [1]; however, he does not deal with directed roundings, which we feel are essential to proper operation of a computer.

Throughout this paper, we will assume that our computer operates in the base β number system. A floating-point number is a pair (e, f), where e is an m-digit signed integer exponent (power of β) and f is a p-digit signed fraction. Since the size and particular representation of e have no bearing on accuracy apart from limiting the size of the largest and smallest machine numbers, we will not concern ourselves with the details of exponent representation here. In subsequent remarks in this section that deal with number representations, it is to be understood that the statements are true only within the range permitted by the exponent size.

The floating-point number (e, f) represents the number $\beta^e \times f$, where

$$f = \pm \sum_{i=1}^{p} a_i \beta^{-i}, \qquad a_i = 0, 1, \cdots, (\beta - 1). \tag{1.1}$$

Manuscript received January 31, 1973. This work was sponsored by the U.S. Army under Contract DA-31-124-ARO-D-462.

The author is with the Mathematics Research Center, University of Wisconsin, Madison, Wis.

It is clear from (1.1) that any number that can be expressed as a p-digit base β fraction times an m-digit power of β is a machine representable number, and no other numbers are representable. Since m and p are fixed, only finitely many different real numbers are representable. Any nonrepresentable number R must be approximated by a machine number; if m_1, m_2 are the two consecutive machine numbers such that $m_1 < R < m_2$, and R is approximated by either m_1 or m_2, then it is clear that the approximation is subject to an error of the order of β^{-p}. This is sometimes referred to as the basic machine precision.

A floating-point number is said to be *normalized* if $a_1 \neq 0$; we will assume that all floating-point numbers are normalized, since maximum accuracy is maintained by use of normalized numbers. In this terminology, zero is a special case; we will assume that it is expressed by a zero fraction and any desired (but specific) exponent (this is usually taken to be the smallest allowable exponent); we will admit zero as an exception to the rule that all numbers must be normalized.

II. Roundings and Directed Roundings

An axiomatic approach to computational rounding has been given by Kulisch in [2]. For the sake of completeness, we sketch some of the points of his theory here. We do not state the theory in full generality; those interested in further information along these lines should consult [2].

Let $\boldsymbol{R}$ be the real number system and let $\boldsymbol{M}$ be the set of machine representable numbers. A mapping $\square : \boldsymbol{R} \to \boldsymbol{M}$ is said to be a *rounding* if, for all $a, b \in \boldsymbol{R}$ we have

$$\square a \leqslant \square b \quad \text{whenever} \quad a \leqslant b.$$

A rounding is called *optimal* if for all $a \in \boldsymbol{M}$, $\square a = a$. In practice, this must be true for any reasonable representation of a, which must certainly include any representation the computer might manufacture during an intermediate stage of an arithmetic operation. The definition of optimal rounding implies that if $a \in \boldsymbol{R}$ and m_1, m_2 are consecutive members of $\boldsymbol{M}$ with $m_1 < a < m_2$, and if $\square : \boldsymbol{R} \to \boldsymbol{M}$ is an optimal rounding, then either $\square a = m_1$ or $\square a = m_2$.

A rounding is *downward directed* (*upward directed*) if, for all $a \in \boldsymbol{R}$, we have $\square a \leqslant a$ ($\square a \geqslant a$). A rounding is *symmetric* if $\square a = -\square(-a)$. If $\square$ is a rounding, a, b are machine numbers, and $*$ is an arithmetic operation, then by $a \circledast b$ we will mean $\square(a * b)$.

By [2, theorem 1] optimal directed roundings are unique. We denote the optimal upward directed rounding by Δ, and the optimal downward directed rounding by ∇.

There are three symmetric roundings that are of interest: inward directed rounding, or rounding toward zero, which is often called truncation and which we denote by T; outward

directed rounding, or rounding away from zero, which we call augmentation and denote by A; and the symmetric rounding that takes each real number to the closest machine number (rounding to the next machine number whose magnitude is larger if there is a tie), which we denote by $\bigcirc$.

III. Floating-Point Hardware Design

By our definition, proper hardware design would enable the computer to perform any of the roundings Δ, ∇, and $\bigcirc$ at the user's option. The rounding $\bigcirc$ is most frequently used, since it produces maximum accuracy. However, the roundings Δ and ∇ are used in implementation of interval arithmetic, for example, and in certain other situations; we will discuss applications briefly in Section VII. The following is a brief sketch of the theory presented in [6].

What information does a computer need in order to round a real number properly? It clearly needs the first p digits of the appropriate base β fraction. Moreover, in order to be able to round to the nearest machine number (by our definition of such rounding) it needs the $p+1$st digit of the fraction. Finally, in order to obtain a correct upward or downward directed rounding, it needs an indicator to tell us whether there are any nonzero digits in the remainder of the fraction.

The result of an arithmetic operation combining two machine numbers is not, in general, a machine number; we must usually approximate the answer. In order to assure ourselves that our computer has all of the above information at the conclusion of an arithmetic operation, we must design it to preserve even more information during the execution of the operation. We will illustrate this by means of a floating-point decimal representation which uses a three-digit fraction and sign-magnitude representation for negative numbers. We will confine our discussion to addition, since subtraction is essentially the same, and multiplication and division have less stringent requirements for additional information.

First, we clearly must have a signed fraction in the arithmetic register; we denote the sign by S (it may be a single binary digit) and the fraction by FFF. Next, since the addition of two three-digit numbers can yield a four-digit number, the arithmetic register sometimes needs an overflow digit to contain the sum of two numbers. This is denoted by O in Fig. 1. These parts of the arithmetic register would be necessary even if we were doing only integer arithmetic.

We will also need two guard digits at the right-hand end of the register to preserve information that is shifted out of the right-hand end of the three-digit fraction. These guard digits are appended to the three fractional digits to form a five-digit fraction; all five digits participate fully in the addition. The initial value of the guard digits is, of course, zero (in $(\beta-1)$'s complement arithmetic, a zero digit is expressed as zero if the number is positive and as $(\beta-1)$ if the number is negative). The need for one guard digit is self-evident; the need for two is illustrated by the following problem:

$$0.100 \times 10^{0}$$
$$-0.995 \times 10^{-2}.$$

This would be computed as follows in sign-magnitude arithmetic:

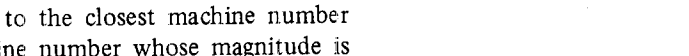

Fig. 1. The structure of the accumulator.

$$\begin{array}{r} 0.10000 \\ -0.00995 \\ \hline 0.09005. \end{array}$$

Normalization now yields 0.90050×10^{-1}; the former second guard digit is now the first guard digit, and is necessary for proper rounding. It will be shown in Section V that two guard digits are always sufficient to preserve maximal accuracy.

The two guard digits are denoted by GG in Fig. 1.

The final item of information needed is an indicator to show whether there are any nonzero digits beyond the second guard digit. This indicator can be a single binary digit, and is denoted by I in Fig. 1.

Although we have avoided any mention of exponent overflow and underflow conditions, proper hardware design should include proper handling of out-of-range numbers. This includes an interrupt upon occurrence of the error condition, a complete set of indicators to tell the user exactly what went wrong, and a result that is at least indicative of the problem if not meaningful. The algorithms Section IV are designed with these considerations in mind.

One further word about hardware is appropriate: if the machine operates in the base $\beta \neq 10$, then the hardware ought to provide facilities for conversion between base β and base 10. If the hardware is designed to do arithmetic operations and rounding in the manner described here, accurate conversions—at least from base 10 to base β—should also be relatively easy to incorporate. This is discussed in detail in [8], and we will explore it no further here.

IV. Algorithms for Rounded Floating-Point Arithmetic

The algorithms presented in this section are designed to simulate the floating-point arithmetic described in Section III. These algorithms are distillations of the algorithms appearing in [6]; those algorithms were intended primarily as microflow algorithms to suggest how hardware might be designed to produce the above roundings, whereas these algorithms are designed primarily for software implementation of the principles outlined in Section III.

The algorithms presented in this section postulate the following machine features and capabilities.

1) There is a double-length register AX that can be shifted left or right as a single register; however, either half (A or X) of the register may be loaded, stored, tested, and shifted separately. When considered as a double-length register, A is assumed to contain the most significant portion of the number. The full double-length register is not of vital importance to any of the algorithms; in fact, the X register is primarily a substitute for the indicator I of Section III. What *is* important is the ability to determine whether the result contains nonzero digits beyond the second guard digit. Thus, if the double-length register is not available, it will be necessary to find another means of sensing and preserving this information.

2) There is a single-length auxiliary register or temporary storage location called U that can be manipulated as necessary, and an indicator or storage cell called S that may be used to keep track of the sign of the result during calculation.

3) There are separate arithmetic registers for carrying out exponent arithmetic, and these registers have sufficient capacity that overflow will not occur during operations on valid exponents.

4) The computer is equipped with the following fixed-point arithmetic operations.

a) Single-length fixed-point addition that adds the contents of U to the contents of A, leaving the result in A.

b) A fixed-point fractional multiply instruction that multiplies the contents of A by the contents of U and produces a double-length product in the AX register. The radix point is assumed to lie immediately to the left of the high-order digit of A (and U).

c) A fixed-point fractional divide instruction that divides the contents of AX by U, leaving the quotient in A and the remainder in X.

If the computer is not equipped with fractional multiply and divide instructions, but is equipped with integer instructions that produce a double-length product or require a double-length dividend, respectively, these instructions may be used, provided necessary modifications to the algorithms are made. The reason for choosing the fractional instructions is that, on binary machines, the sign bit of the low-order word may or may not be regarded as a significant digit in the double-length number, according to hardware design. Choice of the fractional operations relieves us from having to clutter up the algorithms with the two cases. In the algorithms that follow, we will always assume this fractional format in describing fixed-point numbers.

5) The arithmetic mode is not important; however, in either sign-magnitude or β's-complement arithmetic, clearing a portion of a word to zero means replacing each affected digit by zero, while in $(\beta - 1)$'s-complement arithmetic each affected digit is to be replaced by zero if the sign of the number is positive, and by $(\beta - 1)$ if the sign of the number is negative.

6) The exponent field of a floating-point number is located in the high-order portion of the machine word (exclusive of sign), and occupies at least three base β digits. The fractional part of a floating-point number occupies the sign position and and p low-order base β digits; i.e., when regarded as a fixed-point number, the fractional part of a normalized floating-point number always lies in the range $\beta^{-(m+1)} \leqslant |f| \leqslant \beta^{-m} - \beta^{-(m+p)}$.

We will denote the largest positive floating-point number by max, and the smallest normalized positive floating-point number by min.

The location of the exponent is restricted only in order to allow for precise specification of shifting in the arithmetic algorithms. The size of the exponent is specified in order to allow room in a single register for the two guard digits and the overflow digit, which are essential to the algorithm. We need two base β digits for guard digits, and we need one digit for overflow. The latter need not be a base β digit; a binary digit will suffice for this purpose. (On a machine such as System/360, the single-length accumulator will not be sufficient to perform the algorithms, and double-length accumulators will have to be used for the calculations. This will lead to special problems in representing the information that these algorithms assume to be contained in the X register; see 1) above.)

In addition to these machine features, we assume in these algorithms that any exponent bias present will be removed during the unpacking of floating-point numbers, and replaced during the packing process. No assumptions are made about the exponent system used; in particular, it is not assumed that a floating-point zero is represented by a word that is identically zero. (Perhaps this results in slightly more complicated algorithms than would otherwise be necessary.) We will designate the exponent used to represent zero by e_0; we designate the maximum possible exponent by $e_{\max}$, and the minimum possible exponent by $e_{\min}$.

The algorithms assume that the operands are normalized floating-point numbers that do not represent $\pm\infty$ and are present in the A and U registers in packed floating-point format. The result will be placed in the A register in packed floating-point format. In addition, there are four error indicators or fault flags that may be set by the floating-point operations given by these algorithms: exponent underflow, exponent overflow, infinity, and divide by zero. These flags will be set if the corresponding fault has occurred during the calculation of the result.

The rounding procedure is the most important part of the algorithm; apart from it, the algorithms are essentially those presented by Knuth in [1]. The rounding algorithm will provide all five rounding options mentioned in Section II, namely: T, A, Δ, ∇, and $\bigcirc$.

In each of these cases, the action of the rounding algorithm is obvious if the result lies between two nonzero normalized floating-point machine numbers. In particular, if the result is an exact normalized machine number all of these options will produce that machine number.

In the case of underflow, the rounding algorithm will set the exponent underflow indicator; the result will be zero, +min, or -min depending on the rounding option chosen.

In the case of overflow, the rounding algorithm will set the exponent overflow indicator. If the rounding option implies rounding toward zero the result will be $\pm$max. If the rounding option implies rounding away from zero, the infinity indicator is set and the result is replaced by the particular bit configuration used to represent "infinity." There may not be such a bit configuration; in this case, $\pm$max may be substituted as being indicative of the result. We designate this configuration by (e_∞, f_∞), or simply ∞.

In Table I, we show the machine number produced by each of the five roundings for any real number R, together with the indicators that are to be set, if any. In this table, m_1 and m_2 represent consecutive positive normalized floating-point machine numbers with $m_1 < m_2$, and m represents any number representable on the machine as a normalized floating-point number or zero.

In Table II, we show the implications of Table I in terms of rounding toward zero or rounding away from zero.

The algorithms are presented in what follows. During the

TABLE I
EFFECT OF ROUNDING

Case	R Range of Values	Rounding Option: Δ Value	Δ Ind[a]	∇ Value	∇ Ind	T Value	T Ind	A Value	A Ind	$\bigcirc$ Value	$\bigcirc$ Ind
1	$R = m$	m		m		m		m		m	
2	$\max + \beta^{e_{\max}-p} \leqslant R$	∞	o,i	max	o	max	o	∞	o,i	∞	o,i
3	$\max + \frac{1}{2}\beta^{e_{\max}-(p+1)} \leqslant R < \max + \beta^{e_{\max}-p}$	∞	o,i	max		max		∞	o,i	∞	o,i
4	$\max < R < \max + \frac{1}{2}\beta^{e_{\max}-(p+1)}$	∞	o,i	max		max		∞	o,i	max	
5	$\frac{1}{2}(m_1 + m_2) \leqslant R < m_2$	m_2		m_1		m_1		m_2		m_2	
6	$m_1 < R < \frac{1}{2}(m + m_2)$	m_2		m_1		m_1		m_2		m_1	
7	$\min - \frac{1}{2}\beta^{e_{\min}-(p+1)} \leqslant R < \min$	min		0	u	0	u	min		min	
8	$\frac{1}{2}\min \leqslant R < \min - \frac{1}{2}\beta^{e_{\min}-(p+1)}$	min	u	0	u	0	u	min	u	min	u
9	$0 < R < \frac{1}{2}\min$	min	u	0	u	0	u	min	u	0	u
10	$-\frac{1}{2}\min < R < 0$	0	u	$-\min$	u	0	u	$-\min$	u	0	u
11	$-\min + \frac{1}{2}\beta^{e_{\min}-(p+1)} < R \leqslant -\frac{1}{2}\min$	0	u	$-\min$	u	0	u	$-\min$	u	$-\min$	u
12	$-\min < R \leqslant -\min + \frac{1}{2}\beta^{e_{\min}-(p+1)}$	0	u	$-\min$		0	u	$-\min$		$-\min$	
13	$-\frac{1}{2}(m_1 + m_2) < R < -m_1$	$-m_1$		$-m_2$		$-m_1$		$-m_2$		$-m_1$	
14	$-m_2 < R \leqslant -\frac{1}{2}(m_1 + m_2)$	$-m_1$		$-m_2$		$-m_1$		$-m_2$		$-m_2$	
15	$-\max - \frac{1}{2}\beta^{e_{\max}-(p+1)} < R < -\max$	$-\max$		$-\infty$	o,i	$-\max$		$-\infty$	o,i	$-\max$	
16	$-\max - \beta^{e_{\max}-p} < R \leqslant -\max - \frac{1}{2}\beta^{e_{\max}-(p+1)}$	$-\max$		$-\infty$	o,i	$-\max$		$-\infty$	o,i	$-\infty$	o,i
17	$R \leqslant -\max - \beta^{e_{\max}-p}$	$-\max$	o	$-\infty$	o,i	$-\max$	o	$-\infty$	o,i	$-\infty$	o,i

[a] i represents the infinity indicator; o represents the overflow indicator; and u represents the underflow indicator.

TABLE II
ROUNDING IMPLICATIONS

Rounding Option	Sign of Number +	Sign of Number −
T	round toward zero	round toward zero
A	round away from zero	round away from zero
Δ	round away from zero	round toward zero
∇	round toward zero	round away from zero
$\bigcirc$; $\lvert\text{result}\rvert \geqslant \min$ and p + 1st digit $< \beta/2$	round toward zero	
$\bigcirc$; $\lvert\text{result}\rvert \geqslant \min$ and p + 1st digit $\geqslant \beta/2$	round away from zero	
$\bigcirc$; $\lvert\text{result}\rvert < \frac{1}{2}\min$	round toward zero	
$\bigcirc$; $\frac{1}{2}\min \leqslant \lvert\text{result}\rvert < \min$	round away from zero	
$\bigcirc$; exponent overflow	round away from zero	

course of the actual arithmetic computations, the overflow digit, the p digits of the fraction, and the two guard digits described in Section III are all carried in the low-order $p + 3$ digits of the A register, while the X register functions as the indicator I of Section III. After normalization but before rounding, the guard digits are shifted into the X register to allow for easier determination of an exact result; from there on, the two high-order digits of the X register function as the guard digits, and the remainder of X functions as the indicator I.

Algorithm 1 (Addition)

Step 1: (Unpack.) Remove the exponents of the operands and clear the corresponding position to zero. Clear all indicators. Clear X to +0.

Step 2: (Prepare for addition.) If $e_A > e_U$, interchange e_A with e_U and A with U. Shift both A and U left circularly two places.

Step 3: (Make A positive.) If $A < 0$, change signs of A, U, and S.

Step 4: (Test for zero operand.) If $U = 0$, go to Step 9. If $A \neq 0$, go to Step 5. Otherwise, set $A \leftarrow U$, $e_A \leftarrow e_U$, and, if $A < 0$, change the signs of A and S. Go to Step 9.

Step 5: (Compute shift necessary to align fractions.) Set $\sigma = \min(p + 2, e_U - e_A)$.

Step 6: (Align fractions.) Shift AX right σ places. Set $e_A \leftarrow e_U$.

Step 7: (Add.) Algebraically add U to A using fixed-point arithmetic.

Step 8: (Prepare for normalization.) If $A = 0$, clear A to

+0, set $e_A \leftarrow e_0$, and go to Step 9. If $A > 0$, go to Step 9. Otherwise, change signs of A and S, and, if $X \neq 0$, subtract $\beta^{-(m+p)}$ from A.

Step 9: (Normalize, round, end sign correction, and return.) Execute Algorithm 5.

Algorithm 2 (Subtraction)

Step 1: (Change sign.) Change the sign of U.

Step 2: (Add.) Execute Algorithm 1.

Algorithm 3 (Multiplication)

Step 1: (Unpack.) Remove the exponents of the operands and clear the corresponding positions to zero. Clear all indicators. Clear X to +0.

Step 2: (Test for zero operand.) If $A = 0$, go to Step 5. If $U = 0$, clear A to +0, set $e_A \leftarrow e_0$, and go to Step 5.

Step 3: (Prepare to multiply.) If $A < 0$, change the signs of A and S. If $U < 0$, change the signs of U and S. Shift U left circularly m places. Shift A left circularly two places.

Step 4: (Multiply.) Execute the fractional multiply instruction to multiply A by U. Set $e_A \leftarrow e_A + e_U$.

Step 5: (Normalize, round, end sign correction, pack, and return.) Execute Algorithm 5.

Algorithm 4 (Division)

Step 1: (Unpack.) Remove the exponents of the operands and clear the corresponding positions to zero. Clear all indicators. Clear X to +0.

Step 2: (Test for zero operand.) If $U = 0$, set the zero divisor indicator, set $S \leftarrow$ (sign of A), set $e_A \leftarrow e_{\max} + 1$, set $A \leftarrow \beta^{-(m-1)}$, and go to Step 5. If $A = 0$, go to Step 5.

Step 3: (Prepare to divide.) If $A < 0$, change the signs of A and S. If $U < 0$, change the signs of U and S. Shift AX right $m - 2$ places.

Step 4: (Divide.) Execute the fractional divide instruction to divide AX by U. Set $e_A \leftarrow e_A - e_U$.

Step 5: (Normalize, round, end sign correction, pack, and return.) Execute Algorithm 5.

Algorithm 5 (Normalization, Rounding, End Sign Correction, and Packing)

Step 1: (Test for zero result.) If $A = 0$, go to Step 11.

Step 2: (Right normalization.) If $A \geqslant \beta^{-(m-2)}$, shift AX right one place, set $e_A \leftarrow e_A + 1$, and go to Step 4.

Step 3: (Left normalization.) If $A < \beta^{-(m-1)}$, shift A left one place, set $e_A \leftarrow e_A - 1$, and repeat Step 3.

Step 4: (Align fraction.) Shift AX right two places.

Step 5: (Test for exact result.) If $X = 0$, go to Step 9.

Step 6: (Test for truncation.) If the rounding option is O and either $\beta = 2$, $e_A = e_{\min}^{-2}$, and $A = \beta^{-m} - \beta^{-(m+p)}$, or $\beta > 2$, $e_A = e_{\min} - 1$, and $A = \frac{1}{2}\beta^{-m} - \beta^{-(m+p)}$, go to Step 13. Otherwise, if rounding option implies rounding toward zero (see Table II) go to Step 9.

Step 7: (Round away from zero.) Set $A \leftarrow A + \beta^{-(m+p)}$.

Step 8: (Test for rounding overflow.) If $A < \beta^{-m}$ go to Step 9. Otherwise, shift AX right one place and set $e_A \leftarrow e_A + 1$.

Step 9: (Test for bounds faults.) If $e_A < e_{\min}$, go to Step 13. If $e_A > e_{\max}$, go to Step 14.

Step 10: (End sign correction.) If $S < 0$, change the sign of A.

Step 11: (Pack.) Pack the result into the desired format.

Step 12: (Return.) Operation is complete; return to calling program.

Step 13: (Underflow fault.) Set the exponent underflow indicator. If rounding option implies rounding toward zero, set $A \leftarrow 0$ and $e_A \leftarrow e_0$ and go to Step 10. Otherwise, set $e_A \leftarrow e_{\min}$, $A \leftarrow \beta^{-(m+1)}$ and go to Step 10.

Step 14: (Overflow fault.) Set the exponent overflow indicator. If rounding option implies rounding away from zero, set $e_A \leftarrow e_\infty$ and $f_A \leftarrow f_\infty$, set the infinity indicator, and go to Step 10. Otherwise, set $e_A \leftarrow e_{\max}$, $A \leftarrow \beta^{-m} - \beta^{-(m+p)}$ and go to Step 10.

V. Proofs of the Correctness of the Arithmetic Algorithms

In this section, we prove that the algorithms given in the previous section indeed produce the results claimed for them. Since the primary purpose of these algorithms is to produce results of known accuracy, and since the primary reason for having results of known accuracy is to be able to obtain rigorous bounds on the results of computations, it is reasonable to regard such proofs as a necessary part of the algorithms themselves. Indeed, developing rigorous error analyses on the basis of unproven algorithms would be akin to building a skyscraper on a sand foundation.

We first prove (Lemma 2) that the normalization and rounding algorithm produces a correctly rounded packed floating-point representation of a real number R under certain hypotheses; we then show for each of the arithmetic algorithms that the result produced prior to the execution of Algorithm 5 satisfies the hypotheses of Lemma 2. Thus the algorithms are seen to be correct.

Lemma 1: Suppose $R = \beta^E \times F$ is a normalized real number and * is one of the five rounding options shown in Table I. Then applying * to R is equivalent to rounding R as shown in Table II. Moreover, if $\hat{R} = \beta^{\hat{E}} \times \hat{F}$ is the approximation to R obtained by rounding F to a p-digit fraction as specified by *, renormalizing and adjusting the exponent if necessary, then we have the following.

1) Underflow is indicated in Table I if and only if $\hat{E} < e_{\min}$.
2) Overflow is indicated in Table I if and only if $\hat{E} > e_{\max}$.
3) Infinity is indicated in Table I if and only if either:
   a) the rounding option specifies a bound, and there is no machine representable real number that will serve as a bound of the type indicated, or
   b) the rounding option is O, and $\hat{E} > e_{\max}$, so that R is an indeterminate distance from max; i.e., no *a priori* bound can be given.

Proof: For any of the rounding options *, $*(R) = R$ if and only if R is a machine representable number; hence the first line of Table I is correct, for all roundings.

The value column shown under T in Table I can easily be verified by inspection to be equivalent to rounding toward zero.

The rounding A is defined to be rounding away from zero. If an attempt is made to round a number whose magnitude is larger than max away from zero, there will be no machine representable number greater than R, and the infinity indicator will be set. On the other hand, if R is any number less than or equal to max, then max is at least as far away from zero as R; consequently, there is a machine representable number that will serve as a bound of the type indicated, and the infinity indicator will not be set. Thus the value column and the presence of infinity indicators under the A column are verified.

It is shown in [2, theorem 1] that the rounding Δ is equivalent to the rounding T for negative numbers, and the rounding A for positive numbers; similarly, the rounding ∇ is equivalent to the rounding A for negative numbers and to the rounding T for positive numbers. Consequently, the value columns and the presence of infinity indicators are verified for these columns also.

For the rounding $\bigcirc$, lines 5 through 14 of the value column are easily verified by inspection from the definition of the rounding given in Section II. Lines 4 and 15 follow from the fact that $\max + \frac{1}{2}\beta^{e_{\max}-(p+1)}$ is exactly halfway between max and the real number whose exponent is $e_{\max} + 1$ and whose fraction is β^{-1}. This fact also serves to verify the infinity indications in lines 2, 3, 16, and 17 of this column.

The only points that must still be verified are the overflow and underflow indications. Clearly, whenever we have an infinity indication we have exponent overflow, since any number greater than max, when rounded away from zero, rounds to a number whose exponent is necessarily greater than $e_{\max}$. The only other overflow case to consider is when a number R whose magnitude is greater than or equal to $\max + \beta^{e_{\max}-p}$ is rounded toward zero. But the exponent of such a number is greater than $e_{\max}$, and rounding the fraction toward zero cannot decrease the exponent. Consequently, the exponent of the number $\hat{R}$ is still out of range, and the overflow indication is valid.

We must now prove that underflow is indicated if and only if $E < e_{\min}$. For the rounding T if $|R| < \min$ then $\hat{E} < e_{\min}$ since rounding toward zero cannot increase the exponent. On the other hand, if $|R| > \min$, then min is a machine number lying between R and zero, hence $\hat{E} \geq e_{\min}$ and no underflow condition exists. This verifies the underflow indicators in the T column.

For the rounding A, we observe that any number in the range shown on line 12, when rounded away from zero, will round to min; thus there is no exponent underflow in this case. However, if R is any number in the range shown on lines 10 and 11, we will still have $\hat{E} < e_{\min}$. Conversely, we have $\hat{E} < e_{\min}$ if R is in the range shown on these two lines. The cases for lines 7 through 9 are proved by symmetry, and the underflow indicators in the Δ and ∇ columns follow from the relationships between those roundings and the T and A roundings.

The proof of the underflow indicators for the $\bigcirc$ rounding follows from the above discussion, and the observation that the exponent resulting from rounding a number toward zero is no greater than the exponent resulting from rounding that number away from zero. The lemma is proved.

Lemma 2: Let $R = \pm\beta^{E+(m-2)} \times F$ be a real number, where F is zero or a positive fixed-point number less than $\beta^{-(m-3)}$. Suppose that Algorithm 5 is presented with the following information in the locations indicated.

a) E is contained in e_A.
b) The high-order $m + p$ digits of F are contained in the A register; moreover, either the contents of A are an exact representation of F or $F \geq \beta^{-m}$.
c) There is a nonzero number in the high-order p digits of the X register if and only if the contents of the A register are not an exact representation of F; i.e., if we have $A < F < A + \beta^{-(m+p)}$.
d) The sign of R is the same as the sign of the indicator S.

Suppose further that Algorithm 5 is given one of the five rounding options shown in Table I. Then Algorithm 5 will produce the correct packed floating-point approximation to R as shown in Table I together with the error indications shown there.

Proof: If $A = 0$, then zero is the correct result; the hypotheses of the lemma imply that e_A will contain the exponent used to represent zero; Step 1 takes us directly to the packing step and at Step 12 we correctly return zero as the result of the algorithm. Note that, since A is identically zero, no adjustment is necessary to reposition the radix point.

If $A \neq 0$, then it may be necessary to normalize the number; i.e., to multiply the fraction by a "fudge factor" to insure that $\beta^{-(m-1)} \leq F < \beta^{-(m-2)}$. We will do this by shifting; of course, we must make the appropriate adjustment to the exponent. A right shift of one position divides the fraction by β, consequently, we must add 1 to the exponent. Similarly, we must subtract 1 from the exponent for each position of left shift of the fraction.

If $F \geq \beta^{-(m-2)}$, then the hypotheses of the Lemma insure that $F < \beta^{-(m-3)}$; i.e., that one place of right shift will be sufficient to normalize F. The normalization and exponent adjustment are correctly performed in Step 2.

If $F < \beta^{-(m-1)}$, then the hypotheses of the lemma guarantee that either the normalization can be accomplished by one position of left shift, or that $X = 0$. In either case, left shifting by the number of positions required to normalize the number yields a normalized fraction that is correct to $p + 1$ digits; this is done in Step 3.

We now shift AX right two places to align the fraction as required by the floating-point format. Now, regardless of the normalization steps executed, we have a p-digit fraction, normalized, in A, the $p + 1$st digit of the fraction in the high-order position of X, and a nonzero value in X if and only if A is not an exact representation of the fraction $F \times \beta^{-2}$. For if right normalization occurred, the total right shift is at most three places, and consequently no information was lost. (Note that X cannot be -0; the only possibility of this would be on a binary machine with 1's complement arithmetic; but in this case the sign bit is not counted in the $m + p$ bits we are using, so the low-order bit of X would still be zero.) If left normalization occurred, either the net right shift of the X register is $+1$, or X was zero to begin with; in either case, none of the information in X was lost. In addition, the contents of the two guard digits (after normalization) now appear in X, and the claim is verified.

If the result is exact, Step 5 takes us directly to the test for bounds faults. If the result is not exact, we perform the tests in Table II for choice of rounding; by Lemma 1, these choices will yield a correct rounding.

If the rounding option chosen implies rounding toward zero, then the contents of the A register are already a correctly rounded fraction and we proceed to Step 9. If the rounding option implies rounding away from zero, then the fraction must be incremented by 1 in the low-order digit. This is done in Step 7.

If incrementing the fraction results in an unnormalized fraction, then it must be renormalized. This is done in Step 8; the exponent is increased by 1.

At this point, we have an approximation to R consisting of a properly rounded p-digit fraction, normalized and positioned in the low-order p digits of A; and an exponent in e_A. The pair (e_A, f_A) is thus a correct approximation to R if the exponent e_A is in the proper range. In that event, we apply the proper sign correction to f_A, pack the result into floating-point format, and return it; no error indicators have been set. (The divide by zero indicator is the only one not under the control of Algorithm 5; but if it is set, then the division algorithm has also insured that exponent overflow will occur.)

Step 13 is reached only if the exponent is too small; i.e., if exponent underflow has occurred. It can be reached either from Step 6 or from Step 9.

If Step 13 is reached from Step 6, then the rounding option is "nearest machine number." In this case only, no rounding correction has yet been applied. If Step 13 was reached from Step 9, and the rounding option implies rounding away from zero, a rounding correction has already been applied and the result failed to be a representable machine number, but the result of rounding the rounded number will be the same as it would have been for the original result. (The only case where double rounding can produce an erroneous result is the case of "nearest machine number" where the original number was less than $\frac{1}{2}$ min but the rounded number is equal to $\frac{1}{2}$ min; this case was specifically tested for in Step 6 and rounding was inhibited.)

At Step 13, the exponent underflow indicator is set; if rounding toward zero was implied, a zero result is produced; if rounding away from zero was implied, min is produced as the result. In either case, we exit to Step 10, having produced the result indicated in Table I.

If we reach Step 14, overflow has occurred. The exponent overflow indicator is set, and, in addition, if the rounding option required rounding away from zero, the infinity indicator is set. The maximum positive machine representable number is substituted for the result in the case that rounding toward zero was indicated; the representation of infinity is substituted for the result in case rounding away from zero was indicated.

In any case, the correct exponent and fraction pair is produced, and the indicators are set properly. We then exit to Step 10, where we apply the appropriate sign correction, pack the result, and return it.

This completes the proof of the lemma.

Lemma 3: If more than one position of left shift is required to normalize the result of Step 8 of Algorithm 1, then at most one position of right shift was required in Step 6 to equalize the exponents, and therefore in this case the result of the addition is exactly expressed in the A register.

Proof: See [1, p. 194, exercise 4.2.1.2]. The proof is also given explicitly in [6].

Theorem 1: The results of Algorithm 1 (addition) satisfy the hypotheses of Lemma 2 with respect to the real number $(e_A, f_A) + (e_U, f_U)$.

Proof: It will be convenient to divide the proof into several cases.

Case 1: One of the operands is zero. Suppose first that $(e_U, f_U) = 0$. Then the exact result is (e_A, f_A). After Step 1, the indicators will be cleared and X will be cleared to +0. Step 2 will shift A and U left two positions, thus placing the radix point as required for Lemma 2. If $e_A > e_U$ (which is probably the usual state of affairs), no interchange takes place. In Step 3, A is made positive and S is made negative if A was negative to begin with, and Step 4 sends us directly to Step 9 with all hypotheses of Lemma 2 satisfied. If $e_A < e_U$, then interchange takes place at Step 2; Step 3 has no effect, and interchange again takes place at Step 4. The sign of A is now tested; A is made positive and S is correctly set to the sign of the result. We then proceed to Step 9, again with all hypotheses of Lemma 2 satisfied.

Suppose now that $(e_U, f_U) \neq 0$; then $(e_A, f_A) = 0$. The argument for this case is analogous to the above argument; consequently, we again reach Step 9 with all hypotheses of Lemma 2 satisfied.

Case 2: Both numbers are positive and nonzero. Step 2 places the number with the smaller exponent in A and its exponent in e_A; the number with the larger exponent is placed in U and its exponent in e_U. Both numbers are shifted left two places to position the radix point as required by Lemma 2. Since both numbers are assumed to be positive, Step 3 has no effect; since both are assumed to be nonzero, Step 4 sends us to Step 5.

In Step 5, σ is calculated. If $(e_U - e_A) \leqslant p + 2$, then in Step 6, we shift AX right $(e_U - e_A)$ places, which simply divides A by β^σ, creating an unnormalized fraction with the same exponent as e_U. Now e_A is set to e_U, which is the correct exponent of the result. Note that at most p digits of f_A are shifted into X, so X satisfies the hypotheses of Lemma 2 with regard to positioning of nonzero information; moreover, there will be nonzero digit in the high-order p positions of X if and only if A does not exactly represent the original number with its new exponent.

Addition now takes place; the result is an exact representation of the sum, with the AX register being the fraction. Moreover, since both operands were assumed to have been normalized, we have $\beta^{-(m-1)} \leqslant A < \beta^{-(m-3)}$; consequently, the hypotheses of Lemma 2 are satisfied. Step 8 has no effect in this case, so we enter Step 9 with all hypotheses of Lemma 2 satisfied.

If, at Step 5, $(e_U - e_A) > p + 2$, then all p digits of A are shifted into X; note again that X satisfies the condition of Lemma 2 with regard to positioning of nonzero information. Addition now is equivalent to placing U into A; since both operands were assumed to have been normalized, the contents of A satisfy hypothesis b) of Lemma 2. Moreover, in this

case we are assured that $X \neq 0$ and that A is not an exact representation of the sum. Again, Step 8 has no effect, and we enter Step 9 with all hypotheses of Lemma 2 satisfied.

Case 3: Both operands are negative. Then at Step 3, both are made positive and S is set negative, reflecting that the result is negative. Algorithm 1 proceeds with the magnitudes of the operands and, by the above argument, produces results that satisfy the hypotheses of Lemma 2. Consequently, the theorem is proved for this case.

Case 4: The operands are of opposite signs. Let us suppose first that (e_A, f_A) is positive at Step 3. (The previous discussion extends easily to establish the validity of Step 2 in this case.)

Steps 5 and 6 remain valid; again, X contains a nonzero value in the high-order p digits if and only if f_A contains nonzero digits beyond the second guard digit, and this value indicates that, when the contents of the A register are regarded as a fractional number N, we have $N < f_A < N + \beta^{-(m+p)}$.

In Step 7, we perform the addition. Since f_U is negative, the sum may be positive, zero, or negative. Since the operands were assumed to have been normalized, the sum will be zero if and only if $(e_U, f_U) = -(e_A, f_A)$ (proof of this statement is an easy exercise). Thus in this case we clear A to 0 (in case addition cannot produce -0, this step is unnecessary), set e_A to e_0, and proceed to Step 9 with all hypotheses of Lemma 2 satisfied.

If the sum is positive, then, again because the operands were normalized, we must have $e_A = e_U$; thus the result is exact (and X is zero); we may proceed with normalization since all hypotheses of Lemma 2 are again satisfied.

If the sum is negative, we must change the sign of A since the normalization procedure requires a positive fraction. We record this by also changing the sign of S. If X is zero, the result in the A register exactly represents $f_A + f_U$, and we may proceed directly to the rounding and normalization. By Lemma 3, the result in A will be exact if $A < \beta^{-m}$; consequently, in this case the hypotheses of Lemma 2 are satisfied. If, however, X is nonzero, then the original result represented $N + f_U$, and we have $N + f_U < f_A + f_U < N + f_U + \beta^{-(m+p)}$. The presence of nonzero digits in X indicates that we must make a positive correction to the negative number $N + f_U$; and thus it indicates that we must make a negative correction to the positive number we now have in A. Since the rounding algorithm always regards the presence of nonzero digits in X as indicating a positive correction, we must make an adjustment to the contents of A. Specifically, the presence of nonzero digits in X indicates that the result F of the addition satisfies the inequality

$$f_A < F < f_A + \beta^{-(m+p)}$$

or

$$-f_A - \beta^{-(m+p)} < -F < -f_A.$$

Thus, when we change signs, we must subtract $\beta^{-(m+p)}$ from our result in order for the contents of AX to satisfy the hypotheses of Lemma 2.

In the case where (e_A, f_A) is negative at Step 3, we appeal to the previous argument for the case where both numbers are negative to conclude that S is set properly, and then to the above argument to conclude that the algorithm yields a valid result in this case. This completes the proof of the theorem.

Theorem 2: The results of Algorithm 2 (subtraction) satisfy the hypotheses of Lemma 2 with respect to the real number $(e_A, f_A) - (e_U, f_U)$.

Proof: Since subtraction is accomplished by changing the sign of the subtrahend and adding, the theorem follows from Step 1 of Algorithm 2 and Theorem 1.

Theorem 3: The results of Algorithm 3 (multiplication) satisfy the hypotheses of Lemma 2 with respect to the real number $(e_A, f_A) \times (e_U, f_U)$.

Proof: If either operand is zero, the product is zero. In Step 2, we test for this condition and produce a zero result if the condition is satisfied.

In Step 3, suppose first that both factors are positive. We align the fractions so the product will have its true radix point positioned between the $(p+2)$nd and $(p+3)$rd digits to the left of the right-hand end of the A register. To do this, we position the radix point of the multiplicand in that position, and the radix point of the multiplier immediately to the left of the high-order digit of the U register.

In Step 4, we multiply; the double-length product will in fact have at most $2p$ significant digits, so in particular the three low-order digits of X will still be zero. Since we are using the fractional multiply instruction, we are multiplying a fraction in the range $\beta^{-(m-1)} \leqslant f < \beta^{-(m-2)}$ by a fraction in the range $\beta^{-1} \leqslant f' < 1$; the result, then, will be a fraction in the range $\beta^{-m} \leqslant f \times f' < \beta^{-(m-2)}$. Thus either the fraction is already normalized, or one place of left shift will be required to normalize it. The exponent of the product is the sum of the exponents of the factors.

Thus all hypotheses of Lemma 2 are satisfied and the theorem is proved in this case. If both numbers are not positive, Step 3 will set S negative if the numbers are of opposite signs, and positive if they are both of the same sign. Thus S is the sign of the expected result. The multiplication then proceeds with the magnitudes of the numbers, and the hypotheses of Lemma 2 are again satisfied. This completes the proof of the theorem.

Theorem 4: The results of Algorithm 4 (division) satisfy the hypotheses of Lemma 2 with respect to the real number $(e_A, f_A)/(e_U, f_U)$ provided $f_U \neq 0$. If $f_U = 0$, Algorithm 4 sets the zero division indicator and produces an out-of-range result.

Proof: If the divisor is zero, the algorithm sets the zero divisor indicator and assures that Algorithm 5 will sense an overflow condition so that the result produced will be infinity if rounding away from zero is specified. In case the divisor is nonzero but the dividend is zero, Step 2 simply goes to Step 5, since the dividend is the result.

If both divisor and dividend are nonzero, we proceed to Step 3. Assume first that both are positive. X is cleared to zero, and the AX register is shifted right $m-2$ places so that the result of the division will be in the correct range.

In Step 4, division takes place. Since the dividend is a fraction in the range $\beta^{-(2m-1)} \leqslant f < \beta^{-(2m-2)}$ and the divisor is a fraction in the range $\beta^{-(m+1)} \leqslant f' < \beta^{-m}$, the quotient is a fraction in the range $\beta^{-(m-1)} < f/f' < \beta^{-(m-3)}$. Thus in particular we have at least $p+2$ significant digits in the quotient, and

the fraction is either normalized or it will become normalized with one position of right shift. The exponent is set to the exponent of the dividend minus the exponent of the divisor, and the remainder is placed in X. Note that the remainder can have no more than p significant digits, and since it is placed in the high-order positions of X, the three low-order digits of X are zero. Thus the hypotheses of Lemma 2 are satisfied. The arguments for the other cases of the division algorithm are analogous to the corresponding arguments for the multiplication algorithm; we will not repeat them. This completes the proof of the theorem.

VI. *A Priori* Error Analysis

The standard reference work on *a priori* error estimates for floating-point arithmetic is Wilkinson [5]. For multiplication and division, Wilkinson's error estimates are as natural as can be expected; however, in the case of addition (and subtraction) the realities of computer design make it necessary to produce a rather unnatural and somewhat intractable error estimate in order to reflect the true situation. We will see that the floating-point arithmetic and rounding algorithms presented here yield a more natural and more tractable error bound for addition than can be hoped for if the computer produces less than optimal accuracy.

Throughout this section, we assume that overflow does not occur during arithmetic operations. Overflow is almost invariably fatal to the computation; underflow can often be tolerated, and calculation can proceed with a zero replacing the underflowed quantity. (All the same, underflow should at least optionally trigger an error indicator or interrupt so that it can be detected; the absence of automatic error detection on underflow can lead to failure to recognize invalid computational results!) Replacing an undersized result with zero complicates the error analysis slightly; this was considered by Schoenfeld in [4] and we present his modifications here.

If $*$ is any of the four arithmetic operations in the real field, then we will denote the machine approximation to $*$ by $*_M$. The constants μ, δ, and α that appear in the following formulas are determined by the hardware design (see [4]), but are essentially of the order of β^{-p}. The constant ι is determined such that if computer underflow occurs on the operation $X *_M Y$ then we always have $|X * Y| \leqslant \iota$. The smallest normalized floating-point machine number is an upper bound on ι. In each formula, θ and θ_0 are constants in the range $-1 \leqslant \theta, \theta_0 \leqslant 1$ that depend on the operation and the operands; to keep notation uncluttered, we will not reflect this dependency.

Schoenfeld's modification of Wilkinson's *a priori* formulas is as follows:

$$x \times_M y = (x \times y)(1 + \theta\mu) + \theta_0\iota \tag{6.1}$$

$$x \div_M y = (x \div y)(1 + \theta\delta) + \theta_0\iota \tag{6.2}$$

$$x +_M y = x(1 + \theta\alpha) + y(1 + \theta'\alpha) + \theta_0\iota. \tag{6.3}$$

These formulas imply that

$$|x \times_M y - x \times y| \leqslant |x \times y|\mu + \iota \tag{6.4}$$

$$|x \div_M y - x \div y| \leqslant |x \div y|\delta + \iota \tag{6.5}$$

$$|x +_M y - x + y| \leqslant \alpha(|x| + |y|) + \iota. \tag{6.6}$$

Typical values for the constants μ, δ, and α are $\frac{1}{2}\beta^{1-p}$, $(\beta - \frac{1}{2})\beta^{-p}$, and $2\beta^{-p}$, depending on the design of the hardware.

If we are using the floating-point algorithms presented here, however, we can regard our machine M as being three machines, known as $\bigcirc$, $\triangle$, $\triangledown$, which perform the roundings $\bigcirc$, $\triangle$, and $\triangledown$ (described in Section II), respectively. For each of these machines, we have $x *_{\square} y = x \boxplus y$ whenever $*$ is one of the four arithmetic operations. Moreover, we can replace (6.3) with the formula

$$x \boxplus y = (x + y)(1 + \theta\alpha) + \theta_0\iota, \tag{6.3'}$$

which implies that

$$|x \boxplus y - x + y| \leqslant |x + y|\alpha + \iota. \tag{6.6'}$$

These estimates are clearly more natural and perhaps more aesthetically pleasing than (6.3) and (6.6). Moreover, in the case of rounding to the nearest machine number, we have $\mu = \delta = \alpha = \frac{1}{2}\beta^{-p}$, while in the cases of upper and lower bounds, $\mu = \delta = \alpha = \beta^{-p}$.

In case underflow does not occur, the ι term may be dropped. Thus, if underflow does not occur, the following formulas are valid:

$$x \circledast y = (x * y)(1 + \theta\beta^{-p}), \qquad -\tfrac{1}{2} \leqslant \theta \leqslant \tfrac{1}{2} \tag{6.7}$$

$$x \mathbin{\bigcirc\!\!\!\!\!\!\triangle} y = x * y + \theta|x * y|\beta^{-p}, \qquad 0 \leqslant \theta < 1 \tag{6.8}$$

$$x \mathbin{\bigcirc\!\!\!\!\!\!\triangledown} y = x * y - \theta|x * y|\beta^{-p}, \qquad 0 \leqslant \theta < 1. \tag{6.9}$$

These formulas imply that

$$|x * y|(1 - \tfrac{1}{2}\beta^{-p}) \leqslant x \circledast y \leqslant |x * y|(1 + \tfrac{1}{2}\beta^{-p}) \tag{6.10}$$

$$x * y \leqslant x \mathbin{\bigcirc\!\!\!\!\!\!\triangle} y < x * y + |x * y|\beta^{-p} \tag{6.11}$$

$$x * y - |x * y|\beta^{-p} < x \mathbin{\bigcirc\!\!\!\!\!\!\triangledown} y \leqslant x * y. \tag{6.12}$$

Moreover, it should be noted that whenever the result of any operation is a machine representable number, then the result of the operation is exact. (This, unfortunately, is not always the case with present-day hardware.)

As an extreme example of the improvement in error bounds, let us consider the case of a machine with a ten-digit fraction, operating in the decimal number system; let us assume that $\alpha = \frac{1}{2}10^{-10}$. Then, if we have the following addition:

$$\begin{array}{l} 0.1000000000 \times 10^{1} \\ \underline{-0.9999999999 \times 10^{0}} \\ 0.0000000001 \times 10^{0} = 0.1000000000 \times 10^{-9} \end{array}$$

(in which the result is exact), the error bound obtained from (6.6) would tell us that the result is, assuming $\iota < 10^{-30}$, $0.1000000000 \times 10^{-9} \pm 0.99999999996 \times 10^{-10}$, or, essentially, that we have no significance left. However, (6.6') says that the result is $0.1000000000 \times 10^{-9} \pm 0.1000000001 \times 10^{-19}$, which is a far more optimistic bound on this particular addition! (Actually, in this particular case, the ι term does not apply, but we do not know that *a priori*.)

It can be argued that these summands are probably inaccurate in the last decimal place, so that the bound given in

(6.6) is more realistic than the bound (6.6′). Perhaps this is the case; however, that decision should be left to appropriate error analysis on the summands. The important fact here is that, using Algorithm 1, the result of the above problem is computed *exactly*, and consequently the less uncertainty the bound reflects, the better it is.

The necessity for such a pessimistic bound as that in (6.6) can be seen from looking at the preceding example as it might be computed by a computer that truncated before adding, as some present-day computers do

$$\begin{array}{l} 0.1000000000 \times 10^{1} \\ \underline{-0.09999999999 \times 10^{1}} \\ 0.0000000001 \times 10^{1} = 0.1000000000 \times 10^{-8}. \end{array}$$

Here, of course, we would have $\alpha \sim 0.100000000 \times 10^{-9}$ since the machine truncates before adding, and consequently (6.6) yields an estimate of $0.1000000000 \times 10^{-9} \pm 0.19999999999 \times 10^{-8}$; this is not unduly pessimistic. Of course, (6.6′) does not apply to hardware of this design.

VII. Applications

We will mention a few of the applications of the roundings presented here. The rounding $\bigcirc$ has applications in almost every computation using floating-point arithmetic. It is this rounding that we expect to get, and (usually erroneously) assume we do get, from a piece of equipment costing several million dollars.

The roundings Δ and ∇, while not provided with any production computer we know of, also have important applications.

Perhaps the most obvious application of these roundings is in the implementation of interval arithmetic. Hardware designed to produce these roundings would render the programming of an interval arithmetic package nearly trivial, and would enable interval operations to be executed in one tenth to one fifth of the time normally required to execute them with simulated floating-point arithmetic (simulation is usually necessary if we are to be able to produce the tightest possible bounds). The formula for addition of two intervals, for example, is

$$[a, b] + [c, d] = [a + c, b + d].$$

If we assume that a, b, c, and d are machine representable numbers and denote the computer approximation to $[a, b] + [c, d]$ by $[a, b] \oplxor [c, d]$, then the above formula translates as follows:

$$[a, b] \oplxor [c, d] = [a \nabla c, b \Delta d].$$

Evaluation of this formula on a computer equipped with directed rounding takes just twice as long as evaluating the sum of two floating-point numbers.

An interval arithmetic package for the Univac 1108, using simulated floating-point arithmetic as described in Section IV, is detailed in [3].

Another consequence of directed roundings is that upper and lower bounds for sequences of machine operations are much more easily and accurately computed, both *a priori* and during computation, than is possible with conventional rounding. This enables one to combine *a priori* analysis with computational considerations to produce rigorous bounds for relative error in evaluation of mathematical functions. In [7], it is proved that on a binary computer with optimal directed rounding, the square root of a machine representable number can be calculated exactly if it is machine representable, and bracketed by two consecutive machine numbers if it is not machine representable; this is accomplished without using interval arithmetic. Similar, although not as tight, bounds are obtained for the cube root.

References

[1] D. E. Knuth, *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*. Reading, Mass.: Addison-Wesley, 1969.

[2] U. Kulisch, "An axiomatic approach to rounded computations," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 1020, Nov. 1969.

[3] T. D. Ladner and J. M. Yohe, "An interval arithmetic package for the Univac 1108," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 1055, May 1970.

[4] L. Schoenfeld, "Floating point error estimates," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 721, ch. 9, Aug. 1967.

[5] J. H. Wilkinson, *Rounding Errors in Algebraic Processes, National Physical Laboratory Notes on Applied Science*, no. 32. London: HMSO, 1963.

[6] J. M. Yohe, "Best possible floating point arithmetic," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 1054, Mar. 1970.

[7] ——, "Rigorous bounds on computed approximations to square roots and cube roots," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 1088, Sept. 1970.

[8] ——, "Accurate conversion between number bases," Math. Res. Cen., Univ. Wisconsin, Madison, Tech. Summary Rep. 1109, Oct. 1970.

J. Michael Yohe was born in Delaware, Ohio, on June 8, 1936. He received the B.A. degree from DePauw University, Greencastle, Ind., in 1957, the M.S. degree from the University of Wisconsin, Madison, in 1962, and the Ph.D. degree in mathematics from the same university in 1967.

From 1967 to 1968 he was an Assistant Professor of Mathematics at the Mathematics Research Center, University of Wisconsin, and from 1968 to 1969 he was an Assistant Professor of Mathematics at Pennsylvania State University, University Park. From 1969 to 1971 he was a Project Associate at the Mathematics Research Center, University of Wisconsin, and since 1971 he has been an Assistant Director at the Mathematics Research Center. From 1971 to 1972 he was a Lecturer at the University of Wisconsin. His interests lie in the areas of computer systems programming, computer arithmetic, and topology of three-dimensional manifolds with emphasis on link theory.

Dr. Yohe is a member of the American Mathematical Society, the Mathematical Association of America, and the Association for Computing Machinery.