Computation of elementary functions on the IBM RISC System/6000 processor

by P. W. Markstein

The additional speed and precision of the IBM RISC System/6000* floating-point unit have motivated reexamination of algorithms to perform division, square root, and the elementary functions. New results are obtained which avoid the necessity of doing special testing to get the last bit rounded correctly in accordance with all of the IEEE rounding modes in the case of division and square root. For the elementary function library, a technique is described for always getting the last bit rounded correctly in the selected IEEE rounding mode.

Introduction

The extra precision and high speed of the IBM RISC System/6000* (RS/6000) floating-point instruction set has significantly changed the balance between fixed- and floating-point arithmetic, and performing conditional branches. Re-examining the elementary functions in light of these new capabilities has led to interesting new results concerning those functions.

This paper discusses results concerning division, square root, and the evaluation of the common elementary functions (e.g., sin, exp). Division and square root are primitive IEEE operations [1]; the methods chosen to implement them for the RS/6000 processor required that proofs of their correctness be established. Finally, as a consequence of implementing the common elementary functions using the IBM Haifa Research Group's accurate table method [2, 3], it became apparent that with some modification these routines could always produce the correctly rounded values. While these results could always have been obtained without the RS/6000, it was that machine's new floating-point capabilities that inspired the investigations leading to the results which follow.

The additional precision of the accumulate (floating-point multiply and add) instruction [4] permits certain applications of the Newton–Raphson iteration to converge to correctly rounded results. In developing proofs of the convergence of these iterative techniques, it has also become clear why attempts to carry out such algorithms without the precision of the RS/6000 accumulate instruction were doomed to failure, or required an impractical amount of computation.

In machines having a relatively slow floating-point instruction set, the objective in designing routines for the elementary functions was to minimize the use of floating-point arithmetic. With the RS/6000 processor, there is no need to try to circumvent most floating-point operations. Polynomials of degree n can be evaluated by Horner's

* RISC System/6000 is a trademark of International Business Machines Corporation.

method using just n accumulate instructions. In the case of division and square root, we show how additional floating-point computation avoids the need for a conditional branch, which would be more expensive in these cases.

Division

The RS/6000 divide instruction is implemented by microcode which computes the quotient by iteratively refining guesses of the quotient and the reciprocal of the divisor using the accumulate instruction. The decision to use a microcode approach for floating-point division was motivated by the need to save space on the floating-point chip.

Historically, using a Newton–Raphson approach for division required that special corrective action be taken at the end of the algorithm in order to get the last bit rounded correctly, as required by the IEEE standard. Furthermore, in an IEEE implementation, an indicator must be correctly set to describe whether or not the division was exact. Without corrective action, previous applications of the Newton–Raphson method to division seemed to round some results incorrectly [5]. However, conditional branches at the end of the division process could cause the floating-point pipeline to be drained at the cost of several additional cycles.

If P and D are floating-point numbers, how can P/D be computed with the last bit correctly rounded according to each of the four defined IEEE rounding modes? The entire difficulty lies in computing the bits of the mantissa of P/D. P and D will be written in a nonstandard floating-point form to reduce the problem to one of integer division. Let N be the number of bits represented in the mantissa of a floating-point word. For the RS/6000 architecture, $N = 53$. Select integers B, S, d, and s such that

$$D = B \times 2^d, \qquad 2^{N-1} \le |B| < 2^N,$$

$$\frac{P}{D} = S \times 2^s, \qquad 2^{N-1} \le |S| < 2^N. \tag{1}$$

This choice of the integers B, S, d, and s determines integers A and p such that $P = A \times 2^p$, and $A \equiv 0 (\mathrm{mod}\ 2^{N-1})$, in which the leading N bits of A are the bits of the mantissa of P. Since the exponent of the quotient is simply computed by $s = p - d$, it remains only to determine the integer S which best approximates A/B according to the IEEE rounding mode in which the computer is currently running.

The initial approximation Y to $1/B$ is derived from a table. The Newton–Raphson iteration [6] refines Y to a better approximation Y' by the iteration

$$E = 1 - BY,$$

$$Y' = Y + EY. \tag{2}$$

In (2), B is an integer as determined in (1), and E and Y are floating-point quantities having N-bit mantissas. When Y is correct to within one ulp (unit in the last place), E is computed exactly if the full, double-length product BY can be preserved, or if the hardware is capable of an "accumulate" instruction $(x + yz)$ and allows all the bits of the product yz to participate in the sum, as does the RS/6000 floating-point unit. If the original approximation was not exactly $1/B$, then Y' generated by (2) underestimates $1/B$ before rounding, since

$$\frac{1}{B} = Y(1 + E + E^2 + \cdots).$$

An initial approximation Q to A/B is computed as AY, where Y is an approximation to $1/B$. Q is refined to a better approximation Q' by computing a residual

$$R = A - BQ \tag{3}$$

and then computing

$$Q' = Q + RY. \tag{4}$$

As above, assume that (3) and (4) are evaluated with RS/6000 accumulate instructions. If Q approximates A/B to better than one ulp, the computation of R in (3) is exact.

When (2) is used, Y' will have twice as many bits of $1/B$ correct as Y. The number of correct bits in Q' will be approximately the number of correct bits in Q plus the number of correct bits in Y.

If all intermediate computations of Equations (2), (3), and (4) are carried out in round-to-nearest mode, and the final evaluation of (4) is carried out in the desired rounding mode, then it is not difficult to get the correctly rounded result, except for the case of round to nearest.

The issue in round-toward-zero (truncation), round-to-infinity (ceiling), or round-to-minus-infinity (floor) is whether the exact result is a representable floating-point number, or falls to the left or right of a representable number. If the next-to-last application of (4) produces more than N good bits before rounding, and if the question is a representable floating-point number, then that application of (4) will produce that representable number [because (4) was performed in round-to-nearest mode]. In any case, the next computation of the residual (3) will determine to which side of the current approximation of the quotient the true quotient lies, and the final application of (4) with the machine running in the desired rounding mode will select either the current approximation or its neighbor in the direction of RY, depending on the rounding mode.

The difficulty when the rounding mode is round-to-nearest stems from the fact that the last application of (4) can yield an approximation good to $2N$ bits before rounding, but on the opposite side of the midpoint

between two representable floating-point numbers from the true quotient. The reader can easily verify that this can happen by trying to compute 1/15 using only 4-bit floating-point arithmetic, and taking as the first guess to 1/15, 1/16, that is, $(1000_2 \times 2^{-7})$. Whereas the correctly rounded-to-nearest result is 9/128, that is, $(1001_2 \times 2^{-7})$, the algorithm never budges from the guess 1/16 (see **Figure 1**).

The remainder of this section shows that when Equations (2), (3), and (4) are applied sufficiently often, Q' will become the correctly rounded-to-nearest approximation to A/B, provided that the initial guess to $1/B$ is properly chosen, and that all evaluations of (2), (3), and (4) are performed with RS/6000 accumulate instructions (or their equivalent). (The failure shown in Figure 1 to converge to the correct round-to-nearest result stems from what will be shown to be an improperly chosen initial approximation to 1/15.)

$Y = 0.00010000, Q = 1(Y) = 0.0001000$

$E = 1 - 15\,Y = 0.00010000$

$Y' = 0.0001000 + 0.00000001000$

$= 0.00010001000$, rounds to 0.0001000

$R = 1 - 15\,Q = 0.0001000$

$Q' = 0.0001000 + (0.0001000)\,(0.0001000)$

$= 0.00010001000$, rounds to 0.0001000

Figure 1

Failure to converge to round-to-nearest result. (All fractions shown in binary radix.)

Lemma 1
The residual R cannot equal $B/2$ in radix-2 arithmetic.

Proof If the residual $R = B/2$, then the true quotient q would be exactly representable in $N + 1$ bits, and $A = Bq$ exactly. The product of an $N + 1$-bit number by an integer will always require at least $N + 1$ significant bits. But the dividend A was represented in just N significant bits.

Lemma 1 shows that a quotient cannot lie exactly between two representable numbers. This is very helpful, because it removes the necessity of devising a means of simulating IEEE rounding in the event that a result were to lie exactly between two representable numbers. It is also desirable to influence the algorithm to avoid attempting to add a correction which lies exactly between two representable numbers. Figure 1 also illustrates the danger of using such a correction, since the rounding during the Newton–Raphson algorithm does not necessarily give the desired quotient.

Lemma 2
There is no solution in integers to the diophantine equation

$$2^N X = (2^N - 1)Y + 2^{N-1} - 1$$

satisfying

$$2^{N-1} \le X < 2^N \quad \text{and} \quad 2^{N-1} \le Y < 2^N.$$

Proof It is easy to verify that the diophantine equation is solved by

$$X \equiv 2^{N-1} - 1 (\text{mod } 2^N - 1)$$

and

$$Y \equiv 2^{N-1} - 1 (\text{mod } 2^N),$$

and consequently no solutions match the required constraints on X and Y.

The next proposition is the cornerstone of the division algorithm, because it shows just how close an approximation to a reciprocal must be to guarantee that (3) and (4) yield a correctly rounded quotient.

Proposition 1
If Q is correct to within one ulp, and $Y = 1/B$ correctly rounded, then one application of (3) and (4) yields $Q' = A/B$ correctly rounded.

Proof If $Q = A/B$ to within one ulp, the computation of R by (3) results in a value whose absolute value is less than B. When the adjustment RY to Q is computed, then when $R < B/2$, RY must be less than 1/2, and when $R > B/2$, RY must be greater than 1/2 for correct rounding to occur. (By Lemma 1, the residual $B/2$ cannot occur.)

If $|R| > B/2$, we can write $R = (B + M)/2$, where $M \equiv B (\text{mod } 2)$ and $0 < M < B$. The requirement that $RY > 1/2$ becomes

$$\frac{B + M}{2} Y > \frac{1}{2},$$

or

$$Y > \frac{1}{B + M}.$$

The most stringent condition is when $M = 1$ for odd B, when the following must hold:

$$Y > \frac{1}{B + 1}. \tag{5}$$

Similarly, if $|R| < B/2$, write $R = (B - M)/2$, where $M \equiv B \pmod 2$, $0 < M < B$. The requirement that $RY < 1/2$ leads to

$$\frac{1}{B-1} > Y. \tag{6}$$

Together, (5) and (6) show that if A/B is computed to within one ulp, one additional application of (3) and (4) will lead to a correctly rounded quotient, provided that

$$\frac{1}{B+1} < Y < \frac{1}{B-1}. \tag{7}$$

Notice that

$$\frac{1}{B} - \frac{1}{B+1} = \frac{1}{B(B+1)} > \frac{1}{2^{2N}},$$

with the scaling assumed in (1) for B, whereas $Y = 1/B$ correctly rounded means

$$\left| Y - \frac{1}{B} \right| < \frac{1}{2^{2N}}.$$

Thus, $Y = 1/B$ correctly rounded satisfies (7).

Proposition 1 will be used to show that (2) converges to the correctly rounded reciprocal except for a divisor whose mantissa consists of all 1 bits.

Proposition 2
If Y approximates $1/B$ to within one ulp, one application of (2) will give $Y' = 1/B$ correctly rounded, except when $B = 2^N - 1$.

Proof If Y satisfies (7), then Proposition 1 shows that one iteration of (3) and (4) will yield the correctly rounded quotient. But an application of (3) and (4) is equivalent to (2) when $A = 1$. What are the consequences when Y approximates $1/B$ to within one ulp, but is not correctly rounded? According to the proof of Proposition 1, only if the residual given by (3) takes on the value

$$\left(\frac{B+1}{2}\right) 2^{-2N+1}$$

might an application of (3) and (4) not lead to the correctly rounded result. [Since we are dividing an N-bit integer into 1.0 to get an N-significant-bit fraction, notice that the first $N - 1$ bits to the right of the binary point are zeros, followed by the N significant bits. The remainder $(B + 1)/2$ would have been right-aligned $2N - 1$ bits to the right of the binary point.] Had the quotient Y been truncated instead of rounded up, then

$$Y = \frac{1}{B} - \frac{(B+1)2^{-2N}}{B} = \frac{1 - (B+1)2^{-2N}}{B}.$$

Examine $Y - \dfrac{1}{B+1}$ to determine whether Y satisfies (7):

$$Y - \frac{1}{B+1} = \frac{B + 1 - (B+1)^2 2^{-2N} - B}{B(B+1)}$$

$$= \frac{1 - (B+1)^2 \times 2^{-2N}}{B(B+1)}. \tag{8}$$

For $B < 2^N - 1$, (8) shows that $Y - 1/(B+1) > 0$, and thus satisfies (7). For $B = 2^N - 1$, $Y = 1/(B+1)$, and so (7) is not satisfied.

The reader can easily confirm by hand calculation that (2) does not converge to $1/(2^N - 1)$ correctly rounded to N bits when $B = 2^N - 1$. (Figure 1 illustrates this phenomenon with $N = 4$.)

For divisors having mantissas consisting of all 1 bits, the next proposition shows that (3) and (4) converge to the correctly rounded result if the quotient is approached from above.

Proposition 3
If $B = 2^N - 1$, Y is within one ulp of $1/B$, Q is within one ulp of A/B, and $|Q|$ is at least as large as $|A/B|$ correctly rounded, then one application of (3) and (4) will lead to the correct quotient.

Proof If Y is within one ulp of $1/B$, but is not equal to $1/B$ correctly rounded, then Y will not satisfy (7), but it will satisfy

$$\frac{1}{B+2} < Y < \frac{1}{B-1}. \tag{9}$$

By virtue of the discussion in the proof of Proposition 1, such a Y will produce the correctly rounded quotient provided Q is within one ulp of A/B and the (magnitude of the) residual is not equal to

$$\frac{B+1}{2} = 2^{N-1}.$$

First, if Q is already equal to A/B correctly rounded, the residual R must be less than $B/2$, so that by the proof of Proposition 1, an application of (3) and (4) will return Q correctly rounded.

If Q exceeds A/B correctly rounded, (3) and (4) will still yield the correctly rounded quotient, provided that the residual is not equal to 2^{N-1} in magnitude. Since Q is too large, R will be negative. But a residual of -2^{N-1} is congruent to $2^{N-1} - 1 \pmod{2^N - 1}$, and Lemma 2 shows that it cannot arise.

Using the above propositions, the conditions for computing a correctly rounded quotient without any conditional branching to adjust the last bit will be given.

The computations in (2), (3), and (4) must each be done with rounding occurring only on the additions or subtractions. Regardless of the rounding mode with which the quotient is to be computed, all applications of

(2), (3), and (4) are to be done in round-to-nearest mode, except for the final application of (4), which should be performed in the desired rounding mode.

It is necessary to repeat (2) sufficiently often that on the next-to-last iteration there is more than N-bit accuracy before rounding. While this result may not be rounded correctly, the last iteration will produce the correctly rounded reciprocal, except for divisors whose mantissas consist of all 1 bits, by Proposition 2. The number of iterations depends on the precision of the initial "guess" to the reciprocal of B.

The first approximation of the quotient A/B must be made in such a manner that it overestimates A/B in those cases where the mantissa of B consists of all 1 bits. Whenever (3) and (4) are applied, the value Y must have resulted from an application of (2). Equations (3) and (4) must first be applied sufficiently often that the value of Q' has more than N-bit accuracy before rounding. After rounding, it will be within one ulp of the correctly rounded result. In particular, if the quotient is exact, it will have been produced by this time. If the approximate reciprocal is generated according to the previous paragraph, then by Propositions 1 and 3, final application of (4) will result in the correctly rounded quotient. As a by-product, the last application of (4), which should be executed in the desired rounding mode, will correctly characterize the result as to whether or not it is exact.

Square root

For the RS/6000 processor, the primitive IEEE operation, square root, is implemented in software. Two algorithms that can be used are power series evaluation and Newton–Raphson iteration. With either algorithm, the last bit must be rounded correctly according to whichever of the four IEEE rounding modes is in effect when the routine is entered. Tuckerman rounding provides the precise criterion for determining whether an approximation of a square root is correctly rounded-to-nearest, or must be changed (up or down) by one ulp [7].

If g is a guess to $\sqrt{x}$, Tuckerman rounding states that g is correctly rounded-to-nearest if and only if

$$g(g-u) < x \leq g(g+u), \tag{10}$$

where u = one ulp. This inequality stems from the requirement that for g to equal $\sqrt{x}$ correctly rounded-to-nearest,

$$(g-u/2)^2 < x < (g+u/2)^2.$$

Examining the right-hand inequality above shows that

$$x < \left(g+\frac{u}{2}\right)^2 = g(g+u) + \frac{u^2}{4},$$

or

$$x \leq g(g+u),$$

since there are no numbers z representable in floating-point in the interval

$$g(g+u) < z \leq g(g+u) + \frac{u^2}{4}.$$

(The reader can convince himself of this by rescaling the floating-point numbers to force one ulp to equal 1.) Thus, if

$$x \leq g(g+u),$$

g must be closer than $g+u$ to $\sqrt{x}$. A similar argument shows that if

$$g(g-u) < x,$$

g must be closer than $g-u$ to $\sqrt{x}$.

After g is computed with sufficient accuracy to be in error by no more than one ulp (either by taking a sufficient number of terms in a series expansion or by using a sufficient number of iterations of the Newton–Raphson iteration), Tuckerman rounding can be applied to resolve whether g requires adjustment by one ulp. On the RS/6000 processor, the floating-point compares would be followed immediately by conditional branches. Because of the high degree of pipelining in the floating-point unit, the processor would have to delay as much as 15 cycles before returning the final result. Can the conditional branches needed for Tuckerman rounding be avoided?

Analysis of the Newton–Raphson method shows that only two mantissas cause the method to converge to the incorrectly rounded value. As in the case of division, it is only the treatment of the mantissa of the result that is of interest. So, to force the analysis into number-theoretic terms, the number x whose square root is sought is assumed to be an integer in the range

$$2^{2N-2} \leq x < 2^{2N}, \tag{11}$$

so that its square root lies in the range

$$2^{N-1} \leq \sqrt{x} < 2^N. \tag{12}$$

Of course, since x must be representable with only N significant bits, it follows that in order to satisfy (11),

$$x \equiv 0 (\text{mod } 2^{N-1}) \text{ when } x < 2^{2N-1} \tag{13}$$

and

$$x \equiv 0 (\text{mod } 2^N) \text{ when } x \geq 2^{2N-1} \tag{14}$$

The most difficult cases for the round-to-nearest mode occur when $x = g(g \pm u)$, since the correct square root is almost $g \pm u/2$. Because of the scaling chosen for this analysis, the guesses g are always integers, and $u = 1$. The most difficult square root problems occur when

$$g(g \pm 1) \equiv 0 (\text{mod } 2^{N-1}),$$

$$g(g \pm 1) \equiv 0 (\text{mod } 2^N). \tag{15}$$

The only g satisfying (14) in such a way that $x = g(g \pm 1)$ with x satisfying (11) are

$$\begin{aligned} g &= 2^N, & x &= 2^{2N} - 2^N, \\ g &= 2^N - 1, & x &= 2^{2N} - 2^N, \\ g &= 2^{N-1}, & x &= 2^{2N-2} + 2^{N-1}, \\ g &= 2^{N-1} + 1, & x &= 2^{2N-2} + 2^{N-1}. \end{aligned} \tag{16}$$

The discussion which follows shows that for all x satisfying (11) except for the two specific values cited in (15), the Newton–Raphson algorithm must converge correctly to the round-to-nearest square root. The variant of the Newton–Raphson iteration that will be examined is

$$g' = g + \frac{x - g^2}{2g}. \tag{17}$$

This form of the Newton–Raphson iteration is avoided in most computer implementations because $x - g^2$ suffers loss of accuracy due to cancellation which gets worse as g^2 gets closer to x. Instead, $g' = 0.5(g + x/g)$ is favored. But in the RS/6000 floating-point unit, with its accumulate instruction, the mantissa of $x - g^2$ always contains N significant bits. For all x satisfying (11) except those shown in (16), $x - g^2 \neq g$, so the correction to g will be other than exactly 1/2. If y, the reciprocal of $2g$, were available, correctly rounded (or one ulp too large or too small), the analysis of Proposition 1 shows that

$$g' = g + (x - g^2)y \tag{18}$$

will round correctly to $g + u$ or $g - u$ whenever $|x - g^2| > g$, and resolve to g whenever $|x - g^2| < g$. For the case where $x - g^2 = g$, Tuckerman rounding shows that g is the correctly rounded-to-nearest result, and if $x - g^2 = -g$, $g - u$ is the correctly rounded result. Since the Newton–Raphson iteration converges to $\sqrt{x}$ from above, it is sufficient to ensure that when $|x - g^2| = g$, (18) will yield $g - u$. This will be achieved if y overestimates $1/2g$ by at least 1/2 ulp. However, to still guarantee convergence for all other cases, y cannot overestimate $1/2g$ by more than one ulp.

Equation (18) can be evaluated with two accumulate instructions. The first computes $x - g^2$, and the second the resultant value of g', the improved guess with twice as many good bits as g (before rounding), if y were available. How can y be computed cheaply when division is a very expensive operation? The initial value of g is derived from a table based on the leading n bits of the mantissa of x. (For the RS/6000 application, n was taken to be 8.) A parallel table yields an approximation y of $1/2g$, also good to n bits. With these starting values, the $2n$-bit approximation g' is computed. Since g and g' agree to n bits, y is an n-bit approximation of $1/2g'$. Interleaved with the next computation of g' is the refinement of y to the reciprocal of $2g'$. The cost of repeated evaluations of (18) would be four cycles if an oracle provided y (since each accumulate instruction is dependent on the previous one.) By interleaving the refinement of the reciprocal with the computation of (18), one iteration takes six cycles. The final touch is to perturb the reciprocal computation to always overestimate the reciprocal of $2g$. This is easily accomplished by using $1 + ulp$ instead of 1 in the evaluation of (2). This never creates an error of more than one ulp in $1/2g$. If $|x - g^2| = g$, the correction (18) yields $g - u$. [If this slightly incorrect value for y is applied when $x - g^2 = g$, (18) will incorrectly produce $g + u$, but the values given in (16) are the only ones that can cause this to happen when their correctly rounded square roots are obtained. But by that time no further iterations are performed if the initial guesses to those square roots are sufficiently poor!]

The code fragment presented in **Figure 2** shows the refinement of the initial guess g (and the guess y for $1/2g$) to the correctly rounded $\sqrt{x}$, assuming that the mantissas have 53 bits and the initial guesses have eight correct significant bits. Each line of code produces one RS/6000 instruction with the PL.8 compiler. The instructions which lose one cycle by depending on the previous operation are marked in the comment field with "!!."

Elementary function library

In the previous two sections, we have shown how the IBM RISC System/6000 architecture is exploited to perform division and square root as required by the IEEE floating-point standard. The standard is silent, however, on the precision with which other elementary mathematical functions should be computed. Perhaps no precision requirements were specified because it was deemed unreasonable or impossible to achieve last-bit accuracy. At additional execution cost, of the order of 20%, the standard mathematical library can be written so as to always achieve the correctly rounded result.

In almost all cases, the process of computing an elementary function consists of three major components: reducing the argument to a small domain, evaluating the function (or a related one) for the reduced argument, and combining the function just evaluated with data developed during the argument-reduction process. Recent papers [2, 3, 8] have stressed the importance of controlling errors during the process of argument reduction. Here again the accumulate instruction proves to be very useful. Argument reduction often involves computation of

$$x' = x - nc,$$

where x is the original argument and n is an integer chosen so that x' will be less than $c/2$ in magnitude. Because c is usually known to high precision, in practice there are two (or more) floating-point numbers, c_1 and c_2 (and perhaps more), so that $c = c_1 + c_2$, but c_2 is less than 1/2 ulp of c_1. The above reduction is then carried out as follows:

$t = x - nc_1$ (Note: This is exact if $n = x/c1$ is correctly rounded),

$x' = t - nc_2$.

The above scheme can be extended to use even longer approximations of c in the event that x' is sufficiently small (implying that there are fewer than N significant bits in x').

Gal and Bachelis [3] propose computing an elementary function $f(x)$ as follows (after preliminary argument reduction of the sort shown above has been performed): Split the domain of the argument-reduced function into 256 approximately equal intervals. Instead of choosing equidistant boundary points, choose the domain boundaries x_i so that $f(x_i)$ when evaluated to high precision has 11 or more 0s or 1s beginning at significant bit 54 of its mantissa. In this way, a long floating-point IEEE word with 53 bits of mantissa behaves as though it had 64 bits of precision. Now compute $f(x)$ as follows:

$$f(x) = t_i + g(x - x_i), \tag{19}$$

where i is chosen so that $|x_i - x|$ is minimized. In (19) t_i is the function value $f(x_i)$ represented as a long IEEE floating-point word extracted from a table indexed on i, and $g(x - x_i)$ is evaluated by economized power series [6]. Since $g(x_i) = 0$, g itself can be written as

$$g(x - x_i) = (x - x_i)h(x - x_i), \tag{20}$$

where h is a polynomial. After $h(x - x_i)$ is computed, (19) can be evaluated with one accumulate instruction:

$$f(x) = t_i + (x - x_i)h(x - x_i). \tag{21}$$

In (21), if h has been evaluated with sufficient accuracy and if the product is less than 1/2048 of the magnitude of t_i, the accumulate instruction will produce the correctly rounded result in 1023/1024 of the cases. Since polynomial evaluation is fast on the RS/6000 processor, this technique (or minor variations) leads to subroutines for the elementary functions that are both fast and accurate.

What is needed to obtain the correct result in the remaining 1/1024 of the cases? First, the elementary functions exp, sin, cos, tan, and their inverses are all transcendental. While floating-point arguments are termed "real," they are in fact limited to rational values, since their mantissas are finite in length. The above-

```
d = x - g * g;          /*assume machine in round-to-nearest mode */
y2 = y + y;
g = g + y * d;          /*16-bit approximation to g               */
e = almost_half - y * g; /* ! ! Newton-Raphson iteration for      */
d = x - g * g;          /*reciprocal is interleaved with          */
y = y + e * y2;         /*Newton-Raphson iteration for the        */
g = g + y * d;          /* ! ! sqrt.  32-bit approx to g.         */
y2 = y + y;
e = almost_half - y * g;
d = x - g * g;          /*almost_half = 0.5 + 2**(-53)            */
y = y + e * y2;
g = g + y * d;          /* ! ! 64-bit approximation before rounding */
y2 = y + y;
e = almost_half - y * g; /* Caution! The approximation was         */
d = x - g * g;          /* good to 64 bits before rounding, but    */
y = y + e * y2;         /* it may have rounded incorrectly.        */
setflm(fmode);          /* setflm is a PL.8 built-in function to   */
                        /* restore the original floating-point status */
                        /* and the original user-defined rounding  */
                        /* mode. One more iteration gets the last  */
g = g + y * d;          /* bit right by the arguments in this      */
                        /* paper, and avoids Tuckerman rounding.   */
                        /* Furthermore, the last computation of g  */
                        /* completely characterizes the result and */
                        /* correctly sets all the status bits.     */
```

Figure 2

Square root by Newton–Raphson iteration.

mentioned transcendental functions are all known to yield transcendental numbers for all rational arguments (except, for each of these functions, one specific rational argument, for which a rational result is produced, e.g., $e^0 = 1$). Therefore, evaluating a transcendental function to sufficient precision will always lead to the correct determination of the low-order bit of its long floating-point representation.

When is it necessary to resort to more than the machine's natural longest word length to resolve the last bit of precision? Suppose a formula such as (21) is used to compute the transcendental function. The error introduced by rounding in the computation of (21) will be called the discriminant, and it can easily be

computed as

$$discriminant = [t_i - f(x)] + (x - x_i)h(x - x_i). \quad (22)$$

If the discriminant is not too close to the critical value for the rounding mode (1/2 ulp for round-to-nearest, and 0 for the other rounding modes), the evaluation of (21) has yielded the correctly rounded result. Because the correct value of $f(x_i)$ differs from t_i by less than 1/2048 ulp, and the polynomial h has been chosen to be correct to within 1/2048 ulp of f, only if $|\ discriminant\ |$ differs from the critical value by less than 1/1024 ulp is the correct rounding of (21) in doubt.

If the discriminant is too close to the critical value, the transcendental term must be computed to higher precision. This will occur about once in a thousand evaluations. If the higher-precision calculation can be contained within 1000 cycles, the average increase in computing a transcendental will be increased by about 15 cycles' calculation of the discriminant and its comparison to the critical value, and on the average by one cycle for the high-precision evaluation. To put this cost into perspective, e^x and $\ln x$ can be computed in 50 to 70 cycles by the accurate table method.

To ensure that the high-precision calculation can be kept to within 1000 cycles, a double-length computation is coded very carefully. Effectively, the number of correct bits in the discriminant has been increased by more than 40. So, except for possibly once in a trillion times, the double-length computation will resolve the question of the last bit. Otherwise, a longer detailed calculation is required, but the remote possibility of its occurrence allows great latitude in its coding.

The double-length computation offers an interesting choice, which is discussed here only for the round-to-nearest case. The obvious method is simply to compute the transcendental to double length by brute force. In this situation, the accurate table t is no longer sufficiently precise, so that other means are required to obtain the required precision.

Alternatively, the inverse function can be used to resolve the last bit. If $f(x)$ is being evaluated with g being the inverse function over the range in question, then to resolve whether $f(x)$ should be y or $y + u$ (where y is a representable floating-point number and u is one ulp), $g(y + u/2)$ can be evaluated. If f is an increasing function in the neighborhood of x, then $g(y + u/2) > x$ implies that $f(x)$ rounds to y, and $g(y + u/2) < x$ implies that $f(x)$ rounds to $y + u$. Again, $y + u/2$ is rational, so $g(y + u/2)$ is transcendental and cannot equal x. The analysis of $g(y + u/2)$ is exactly the approach that leads to the derivation of Tuckerman rounding in the case of the square root. The notion of computing $g(y + u/2)$ to determine the rounding for $f(x)$ can be considered an extension of Tuckerman rounding.

Whether the double-length computation should be in terms of the given function or its inverse depends on the relative rates of increase of these functions. As an example, consider the exponential function. If

$$y = e^x,$$

$$\ln y = x,$$

$$\frac{dy}{y} = dx = x\left(\frac{dx}{x}\right).$$

In this form, the relative error in y is expressed as a function of the relative error in x. If dx/x is one ulp, dy/y is x ulps. Now consider the same analysis for the inverse function, the natural logarithm:

$$x = \ln y,$$

$$\ln x = \ln \ln y,$$

$$\frac{dx}{x} = \frac{1}{\ln y}\left(\frac{dy}{y}\right).$$

Reducing the range of arguments for the exponential function leaves new arguments whose absolute value is bounded by 1/2 ln 2. Thus, over the domain in which exp is evaluated, the relative error in exp will be less than 0.4 times the relative error in x. On the other hand, the natural logarithm would be evaluated in the domain $\sqrt{1/2} \le y \le \sqrt{2}$. The closer y gets to 1, the larger the relative error in its logarithm. The conclusion is that the logarithm will be much sharper than the exponential in determining the correct rounding. Even though the series for the logarithm converges more slowly than the exponential, the logarithm reduces the probability of having to resort to more than double-length calculation.

As an illustration, consider the evaluation of $e^{2^{-53}}$. The series expansion gives

$$e^{2^{-53}} = 1 + 2^{-53} + 2^{-107} + \cdots.$$

It would take a triple-length computation to have the 2^{-107} term enter the calculation to indicate that the value should be rounded to $1 + 2^{-52}$. On the other hand, one could compute

$$\ln(1 + 2^{-53}) = 2^{-53} - 2^{-107} + \cdots.$$

Clearly, a double-length computation (single-length almost sufficed) shows that $\ln(1 + 2^{-53}) < 2^{-53}$, and therefore $e^{2^{-53}}$ should be rounded up to $1 + 2^{-52}$.

There are interesting open questions related to this approach to computing the elementary functions. If a rational argument can be expressed in N significant bits, will the discriminant, when computed to N-bit accuracy, always differ from the critical value by more than 2^{-N} ulps? For the exponential, the above example showed that N-bit discriminants are not sufficient over the

domain of interest. Are N-bit discriminants sufficient to correctly determine the rounding for the logarithm function for all arguments between $\sqrt{1/2}$ and $\sqrt{2}$ that are representable as floating-point numbers? If that question could be answered in the affirmative, computation greater than double length would be unnecessary for the exponential and logarithm functions.

Conclusions

The architecture of the IBM RISC System/6000 processor has motivated reexamination of mathematical algorithms, including those used to compute the commonly used elementary functions. The precision of the RS/6000 accumulate instruction allows low-order bits of a product to participate in a sum. Thus, cancellation of high-order bits can become an advantage in certain cases. The form of the Newton–Raphson iteration used for the square root is a good example. The closer a guess g comes to $\sqrt{x}$, the more precise is the computation of $x - g^2$. When g is within an ulp of $\sqrt{x}$, then $x - g^2$ is exact on the RS/6000 processor, whereas it exhibits its worst cancellation characteristics on standard architectures. Knowing that such cancellation leaves a full word of useful data has led to new insights in the numerical analysis of these algorithms, including the opportunity to apply number theory. The resultant algorithms have better characteristics, both in accuracy and the exploitation of pipelined architecture.

The code fragments presented here are all in higher-level language, reflecting the fact that the actual math library for the RS/6000 processor is coded in PL.8. The code available from the RISC System/6000 compiler family is of high quality, potentially eliminating the need to resort to assembly language. Since the programs changed during the development process, it was a relief not to have to struggle with the problems of register allocation and scheduling, knowing that the compiler would faithfully solve these problems in a tailor-made fashion for every new instance of the code.

Acknowledgments

The author has received help and advice from many people. John Cocke suggested the division problem, and was available for frequent discussions about all phases of this work. Professor W. Kahan provided the author with several monographs from the University of California at Berkeley on the analysis of floating-point computations. Reference [5] shows how number theory can be employed for floating-point arithmetic analysis. Ramesh Agarwal also offered advice on the division problem. Peter Oden enriched the PL.8 language to allow the floating-point rounding modes to be controlled from within a PL.8 program; this modification made it possible to do all the coding without resorting to assembly language. Joel Boney and Sharon Lamb consulted frequently on implementation issues, and the author is indebted to them for their extensive testing of the elementary function library.

References

1. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754*, American National Standards Institute, Washington, DC, 1988.
2. S. Gal, "Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance," *Lecture Notes in Computer Science No. 235, Accurate Scientific Computations*, Springer-Verlag, New York, 1986.
3. S. Gal and B. Bachelis, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *IBM Technical Report 88.223*, IBM Israel, Technion City, Haifa, Israel, 1988.
4. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. Develop.* **34,** 59–70 (1990, this issue).
5. W. Kahan, *Checking Whether Floating-Point Division Is Correctly Rounded*, monograph, Dept. of Computer Science, University of California, Berkeley, 1987.
6. F. Hildebrand, *Introduction to Numerical Analysis*, McGraw-Hill Book Company, Inc., New York, 1956.
7. *Elementary Math Library, Programming RPQ P81005, Program No. 5799-BTB, Program Reference and Operations Manual*, a) First Edition (Release 1, EML1), January 1984, Order No. ST40-2230-00; b) Second Edition (Release 2, EML2), August 1984, Order No. SH20-2230-1; both are available through IBM branch offices.
8. Ramesh C. Agarwal, James W. Cooley, Fred G. Gustavson, James B. Shearer, Gordon Slishman, and Bryant Tuckerman, "New Scalar and Vector Elementary Functions for the IBM System/370," *IBM J. Res. Develop.* **30,** 126–144 (1986).

Received February 28, 1989; accepted for publication February 6, 1990

Peter W. Markstein *IBM Research Division, 11400 Burnet Road, Austin, Texas 78758.* Dr. Markstein received his S.B. and S.M. degrees in mathematics from the Massachusetts Institute of Technology in 1958 and 1959, respectively, and his Ph.D. in computer science from New York University in 1975. From 1959 to 1970, he was a member of the Computer Systems Department at the IBM Thomas J. Watson Research Center, where he developed an early multiprogrammed operating system for the STRETCH computer. Dr. Markstein received an IBM Outstanding Contribution Award in 1966 for developing a 7090-to-System/360 code-to-code translator. He also served as manager of the Thomas J. Watson Computing Center. As a member of the Computer Science Department, he participated in a study of operating system security in the early 1970s. He has been a member of the 801 project from its inception in 1975. Dr. Markstein's research interests focus on compiler optimization, for which he received an IBM Outstanding Contribution Award in 1985. He has also contributed to the architecture and design of RISC arithmetic units, and has recently developed a library of elementary function routines for the RISC System/6000 processor. He is currently on assignment to the Advanced Workstations Division, Austin, Texas, in the Hardware Architecture Department. Dr. Markstein holds several patents in compiler optimization and RISC technology.