Hardware Designs for Exactly Rounded Elementary Functions

Michael J. Schulte, *Member, IEEE*, and Earl E. Swartzlander, Jr., *Fellow, IEEE*

***Abstract*—This paper presents hardware designs that produce exactly rounded results for the functions of reciprocal, square-root, 2^x, and $\log_2(x)$. These designs use polynomial approximation in which the terms in the approximation are generated in parallel, and then summed by using a multi-operand adder. To reduce the number of terms in the approximation, the input interval is partitioned into subintervals of equal size, and different coefficients are used for each subinterval. The coefficients used in the approximation are initially determined based on the Chebyshev series approximation. They are then adjusted to obtain exactly rounded results for all inputs. Hardware designs are presented, and delay and area comparisons are made based on the degree of the approximating polynomial and the accuracy of the final result. For single-precision floating point numbers, a design that produces exactly rounded results for all four functions has an estimated delay of 80 ns and a total chip area of 98 mm^2 in a 1.0-micron CMOS technology. Allowing the results to have a maximum error of one unit in the last place reduces the computational delay by 5% to 30% and the area requirements by 33% to 77%.**

***Index Terms*—Computer arithmetic, elementary functions, exact rounding, polynomial approximations, multi-operand adder, parallel multiplier, argument reduction, special-purpose hardware.**

I. Introduction

THE rapid and accurate evaluation of elementary functions (e.g., reciprocal, square-root, exponential, logarithm, etc.) is important for scientific applications. The computation of these functions is often performed by software routines that employ various techniques including polynomial approximations [1], rational expressions [2], and continued fraction expansion [3]. Although software routines can provide extremely accurate results, they are often too slow for numerically intensive applications.

To overcome the speed disadvantage of software routines, several algorithms have been developed for evaluating the elementary functions with special-purpose hardware. These algorithms include the CORDIC algorithm [4], Newton–Raphson iteration [5], rational approximations [6], digit-by-digit algorithms [7], and polynomial approximations [8]–[10]. Although hardware algorithms typically have a speed advantage over software routines, they often produce less accurate results. In addition, their speed advantage is limited, because they are usually implemented iteratively or sequentially, and thus require a large number of machine cycles.

Manuscript received October 15, 1993; revised March 5, 1994. This paper is a revised version of a paper that appeared in the *Proc. 10th Symp. on Comput. Arithmetic*, Windsor, ON, Canada, June 1993, pp. 138–145.

The authors are with the Department of Electrical and Computer Engineering, University of Texas at Austin, TX 78712 USA; e-mail: e.swartzlander@compmail.com.

IEEE Log Number 9403090.

The IEEE 754 Standard [11] requires exact rounding for addition, subtraction, multiplication, division, square-root, remainder, and conversion between integer and floating point formats. Exact rounding, as defined in [12], requires the rounded result to be identical to the result obtained if the infinitely precise value of the function is rounded by using round-to-nearest-even. This minimizes the maximum error of the result. The IEEE 754 Standard, however, does not require exact rounding for the elementary functions. This is largely due to a problem known as the Table Maker's Dilemma, which occurs because there is no known method for determining the accuracy required to guarantee exactly rounded results for the elementary functions [12], [13].

Evaluating the elementary functions with an algorithm that produces exactly rounded results has several advantages. Exact rounding limits the maximum error to half a unit in the last place (ulp). If x is a positive normalized floating point number, then the ulp of x is the difference between x and the next larger floating point number. Exact rounding also preserves several desirable properties of the functions, such as symmetry and monotonicity [14]. Furthermore, exact rounding ensures that machines that have the same floating point format will always produce the same results for a given computation. This improves software portability and allows the correctness of floating point algorithms to be verified for a standardized system. Other advantages of having a specified standard for the elementary functions are given in [15].

Because of the advantages offered, much research has been performed to develop software routines that produce exactly rounded results for the elementary functions. In [14], software routines are described that use accurate argument reduction techniques, followed by polynomial approximations to compute the elementary functions for floating point numbers in the IEEE double-precision format. Although most of these routines achieve exact rounding for over 99.8% percent of the input values, they require more than 70 machine cycles to execute on a general purpose computer.

Routines that are expected to produce exactly rounded results for elementary functions in the IEEE double-precision format are described in [16]. For the first iteration, the result is computed by using double-precision arithmetic. If the prerounded result of this routine does not meet a specified error criterion, the result is recomputed by using a higher-precision routine that is orders of magnitude slower than the original one. This is repeated, using progressively slower

and more precise routines at each iteration, until an exactly rounded result is guaranteed. The goal is that the average time required to compute the elementary functions will be relatively low, because most input values will require only a single iteration. This approach is obviously not practical for real time computations, because hundreds of cycles are required to compute results that are not exactly rounded after the first iteration.

This paper, which is an expansion of [17], presents special-purpose hardware designs that produce an exactly rounded result for the functions of reciprocal, square-root, 2^x, and $\log_2(x)$. Because these designs perform the computation in parallel and because division is not required, they are faster than existing hardware and software methods for function evaluation. In Section II, polynomial approximations are discussed, with an emphasis on the Chebyshev series approximation. Section III presents an algorithm by which the coefficients of the polynomials are adjusted to guarantee exactly rounded results. Section IV gives argument reduction techniques that maintain exact rounding. In Section V, special-purpose hardware for evaluating the elementary functions is discussed. Section VI examines area and delay estimates of hardware designs that produce exactly rounded results for floating point numbers with 16- and 24-bit significands. Section VI also examines the reduction in delay and area when the results are allowed to have a maximum error of one ulp. In Section VII, the difficulty of obtaining exactly rounded results is discussed, and the reduction in delay and area obtained by adjusting the coefficients is analyzed. Section VIII presents our conclusions.

The algorithm presented in this paper has two limitations. First, it cannot be used for all functions. For trigonometric and inverse trigonometric functions, argument reduction can introduce errors that cause the final results not to be exactly rounded. Second, the algorithm is not feasible for numbers with large significands (e.g., double-precision), because the method for adjusting the coefficients requires an exhaustive simulation of all values on a specified input interval.

II. Polynomial Approximations

The hardware designs discussed in this paper use polynomial approximations to compute the elementary functions. Polynomial approximations have the following form:

$$f(x) \approx a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{n-1} \cdot x^{n-1} = \sum_{i=0}^{n-1} a_i \cdot x^i, \tag{1}$$

where $f(x)$ is the function to be approximated, n is the number of terms in the polynomial approximation, and a_i is the coefficient of the ith term. The function is approximated on a specified input interval $[x_{\min}, x_{\max})$, and argument reduction is employed for values outside this interval.

The accuracy of the approximation is dependent on the number of terms in the approximation, the size of the interval on which the approximation is performed, and the method for selecting the coefficients. To reduce the number of terms, the input interval is divided into a set of equal-size subintervals, and different coefficients are used for each subinterval. This is done by separating the p-bit input value into two parts: a k-bit most significant part x_m and a $(p-k)$-bit least significant part x_l, as shown in Fig. 1.

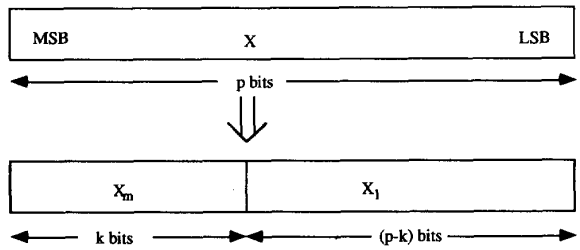


Fig. 1. Dividing the input value.

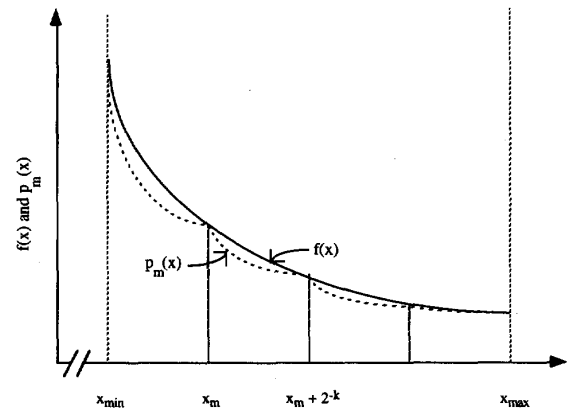


Fig. 2. Dividing the input interval.

If x is on the input interval [0, 1), then we have the following condition:

$$x = x_m + x_l \cdot 2^{-k}, \tag{2}$$

where $0 \leq x_m < 1$ and $0 \leq x_l < 1$. Equation (1) then becomes as follows:

$$\begin{aligned} p_m(x) &= a_0(x_m) + a_1(x_m) \cdot x_l + \cdots + a_{n-1}(x_m) \cdot x_l^{n-1} \\ &= \sum_{i=0}^{n-1} a_i(x_m) \cdot x_l^i, \end{aligned} \tag{3}$$

where $p_m(x)$ is the approximating polynomial of degree $n-1$ for subinterval m. Fig. 2 illustrates the effect of dividing the input interval into subintervals. The coefficients for the approximation are obtained by a table lookup on x_m. The value of x_m determines the subinterval on which the approximation occurs, and the value of x_l specifies the point on the subinterval at which the approximation is made. Fig. 3 shows the approximation for a single subinterval.

For normalized IEEE floating point numbers [11], the input interval is often [1, 2). For these numbers, xm consists of the k most significant bits of x, excluding the most significant bit, which is always 1. Numbers of this form are specified by the following equation:

$$x = 1 + x_m + 2^{-k} \cdot x_l. \tag{4}$$

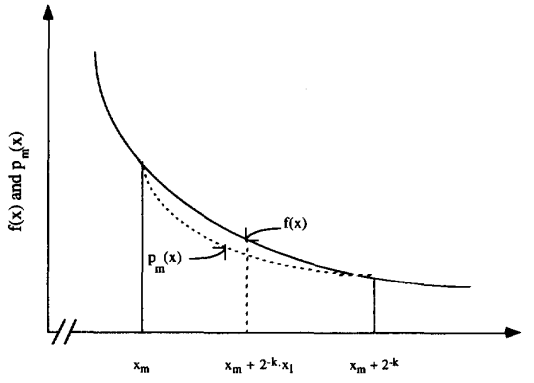


Fig. 3. The approximation on a single subinterval.

For these numbers, p corresponds to the number of bits in the significand, including the hidden one.

Initially, the Chebyshev series approximation [18] (an approximation to the minimax polynomial) is used to select the coefficients for each of the subintervals. The coefficients are then adjusted to obtain exactly rounded results for all values on each subinterval, as described in Section III. The Chebyshev series approximation $p_m(x)$ of degree $n-1$ is computed on the subinterval $[x_m, x_m + 2^{-k})$ by the following algorithm.

1) The Chebyshev nodes on $[-1, 1)$ are computed by using the following formula:

$$t_i = \cos\left(\frac{(2\cdot i+1)\cdot\pi}{2\cdot n}\right) \qquad (0 \le i < n), \tag{5}$$

where t_i is the ith Chebyshev node on $[-1, 1)$.

2) The Chebyshev nodes are transformed from $[-1, 1)$ to $[a, b)$ through the following equation:

$$x_i = \frac{t_i\cdot(b-a)+(b+a)}{2} \qquad (0 \le i < n). \tag{6}$$

For subinterval m on $[x_m, x_m + 2^{-k})$, this becomes as follows:

$$x_i = x_m + (t_i+1)\cdot 2^{-k-1} \qquad (0 \le i < n). \tag{7}$$

3) The Lagrange polynomial $p_m(x)$ is formed, which interpolates the Chebyshev nodes on $[x_m, x_m + 2^{-k})$ as follows:

$$p_m(x) = y_0\cdot L_0(x) + y_1\cdot L_1(x) + \cdots + y_{n-1}\cdot L_{n-1}(x), \tag{8}$$

where the conditions given in (9) and (10) (see the bottom of this page) exist.

4) $p_m(x)$ is expressed in the form given in (3) by combining terms in $p_m(x)$ that have equal powers of x.
5) The coefficients of $p_m(x)$ are rounded to a specified precision using round-to-nearest-even.

As given in [18], the maximum error between a function and its Chebyshev series approximation on an interval $[a, b)$ is as follows:

$$E_n(x) = \left(\frac{b-a}{4}\right)^n \cdot \frac{2\cdot|f^n(\xi)|}{n!} \qquad a \le \xi < b, \tag{11}$$

where ξ is the point on $[a, b)$ where the nth derivative of $f(x)$ has its maximum value. Since the input interval is divided into subintervals of size 2^{-k}, the maximum approximation error is as follows:

$$E_n(x) = \frac{2^{-n(k+2)+1}\cdot|f^n(\xi)|}{n!} \qquad x_m \le \xi < x_m + 2^{-k}. \tag{12}$$

This compares favorably with a Taylor series approximation [18] that has a maximum approximation error of:

$$E_n(x) = \frac{2_{-nk}\cdot|f^n(\xi)|}{n!} \qquad x_m \le \xi < x_m + 2^{-k}. \tag{13}$$

For the Chebyshev series approximation, increasing the number of bits in x_m by one decreases the maximum approximation error by a factor of 2^{-n}, but doubles the number of coefficients. Alternatively, increasing the number of terms by one decreases the maximum approximation error by a factor of approximately $2^{-(k+2)}$, but increases the number of multiplications and additions, as well as the number of coefficients. To limit the approximation error to 2^{-q} with a polynomial of degree $n-1$, k is selected as follows:

$$k = \left\lceil \frac{q - 2n + 1 + \log_2(|f^n(\xi)|) - \log_2(n!)}{n} \right\rceil. \tag{14}$$

Thus, the number of bits in the table lookup is determined by the function being approximated, by the degree of the approximation, and by the required accuracy of the result.

III. An Algorithm for Adjusting the Coefficients

Polynomial approximations provide a high-speed method for computing the elementary functions. However, to obtain exactly rounded results, either the size of the table lookup or the number of terms in the approximation must be large. This section describes an algorithm by which the coefficients are adjusted to obtain exactly rounded results. This algorithm is not applicable to large word-length numbers (e.g., double-precision), because it requires an exhaustive test of all the values on the input interval. Adjusting the coefficients leads to a significant reduction in the size of the table lookup, because

$$Li(x) = \frac{(x-x_0)\cdot\cdots\cdot(x-x_{i-1})\cdot(x-x_{i+1})\cdot\cdots\cdot(x-x_{n-1})}{(x_i-x_0)\cdot\cdots\cdot(x_i-x_{i-1})\cdot\cdots\cdot(x_i-x_{i+1})\cdot\cdots\cdot(x_i-x_{n-1})} \tag{9}$$

$$y_i = f(x_i) \tag{10}$$

```
set the best coefficients to the coefficients of the Chebyshev approximation;
for (i = 1 to number of coefficients ) do
    for (j = 1 to number of subintervals) do
        compute the number of inexact approximations[j] using the best
        coefficients[j];
        for (k = 1 to number of iterations[i]) do
            for sign = -1 to 1 step 2 do
                modify coefficient i on subinterval j by
                    a[i][j] = a[i][j] + sign*k*2^-Pi;
                compute the number of inexact approximations[j] using the
                best coefficients[j] and a[i][j];
                if (the number of inexact approximations[j] is reduced) then
                    a[i][j] becomes the best coefficient for this subinterval;
                    remember the number of inexact approximations[j];
                if (the number of inexact approximations[j] is zero) then
                    exit this subinterval (exit loop j);
            end loop sign;
        end loop k;
    end loop j;
    if (the number of inexact approximations on all subintervals is zero) then
        exit algorithm (exit loop i);
end loop i;
```

Fig. 4. Adjusting the coefficients.

fewer coefficients need to be stored to obtain exactly rounded results. The algorithm for adjusting the coefficients is shown in Fig. 4. Array variables appear in boldface type.

This algorithm requires that the function be evaluated for all the values on the input interval. Thus, for numbers with p-bit significands, at least 2^p function evaluations are required. If exactly rounded results are not obtained on a given subinterval, then the coefficients for that subinterval are modified, and another attempt is made to obtain exactly rounded results. Since most subintervals produce exactly rounded results after a small number of coefficient adjustments (typically each coefficient is adjusted by at most 2 ulps in either direction), the total number of function evaluations is $O(2^p)$. The algorithm for coefficient selection and adjustment executes in under 1 hr for numbers with 24-bit significands on a 50-MFLOPS processor.

IV. Argument Reduction

Before performing the polynomial approximation, it is necessary to transform the original input value so that it falls within a specified input interval. After the function is computed for the transformed input, a second transformation occurs that compensates for the original transformation and normalizes the result. The input and output transformations are commonly referred to as argument reduction. The argument reduction transformations presented here for reciprocal, square-root, $\log_2(x)$, and 2^x maintain exact rounding. Thus, if the results computed over the input interval are exactly rounded, all results will be exactly rounded.

The steps needed to compute each of the elementary functions are shown in Fig. 5. In this figure, it is assumed that the numbers are in the IEEE floating point format [11] for normalized numbers, which have the following form:

$$x = (-1)^{S_x} \cdot M_x \cdot 2^{E_x} \qquad (1 \le M_x < 2). \tag{15}$$

The exponent is assumed to have no bias. The following notation is used.

S_x, M_x, and E_x	The sign bit, significand, and exponent of the input
M'_x and E'_x	The transformed values before the function is computed
M'_y and E'_y	The results of the function before the output transformation
S_y, M_y, and E_y	The sign bit, significand, and exponent of the final result

In Fig. 5, multiplication by 2^{Δ} corresponds to a Δ-bit left shift, and multiplication by $2^{-\Delta}$ corresponds to a Δ-bit right shift. Since the input and output transformations for reciprocal, square-root, and 2^x do not modify the bit values of the significand, ensuring exactly rounded results for all values on the input interval guarantees exactly rounded results for all inputs. For the output transformation of $\log_2(x)$ when E_x is nonzero, it is necessary to add E'_y to M'_y and then normalize the result. Exact rounding is maintained if the normalized result is rounded to the nearest floating point number. If E_x is zero, leading zeros may appear in $\log_2(M'_x)$, which leads to a loss of precision. Since $1 \le \frac{\log_2(M'_x)}{M'_x - 1} < 2$ computing this value, instead of $\log_2(M'_x)$, eliminates the leading zeros. In the next cycle, this result is multiplied by the normalized value of $(M'_x - 1)$, and the exponent is set to account for the normalization of $(M'_x - 1)$.

V. Special Purpose Hardware Designs

This section presents special-purpose hardware designs for elementary function evaluation. Other parallel hardware designs for evaluating the elementary functions by polynomial approximations are presented in [19]–[22]. Our designs, however, make use of specialized multipliers, multi-operand adders, and squaring circuits that are designed for elementary function approximation. In addition, the size of the table lookup and the size of the arithmetic units are tailored to minimize the hardware requirements while guaranteeing exact rounding of the elementary functions. The hardware designs presented in [19]–[22] do not guarantee exact rounding, but allow last-bit errors to occur.

Polynomial approximations are computed on the input interval in three steps.

1) Obtain the coefficients $a_i(x_m)$ and the powers x_l^i.
2) Compute the terms $a_i(x_m) \cdot x_l^i$.
3) Sum together the terms from step 2.

The terms in the approximation are independent of one another, and are generated in parallel by using high-speed multipliers. They are then summed with a two's complement, multi-operand adder. Fig. 6 shows a general block diagram for a polynomial approximation of degree n.

To reduce the hardware complexity of the elementary function generator, special-purpose multipliers, squaring circuits, and multi-operand adders are employed that take advantage of the characteristics of polynomial approximations. Since x_l^i is always positive and a_i can be positive or negative, each term is computed with a specialized multiplier in which the N-bit multiplicand (a_i) is a two's complement number, and the M bit multiplier (x_l^i) is always positive. The partial products for

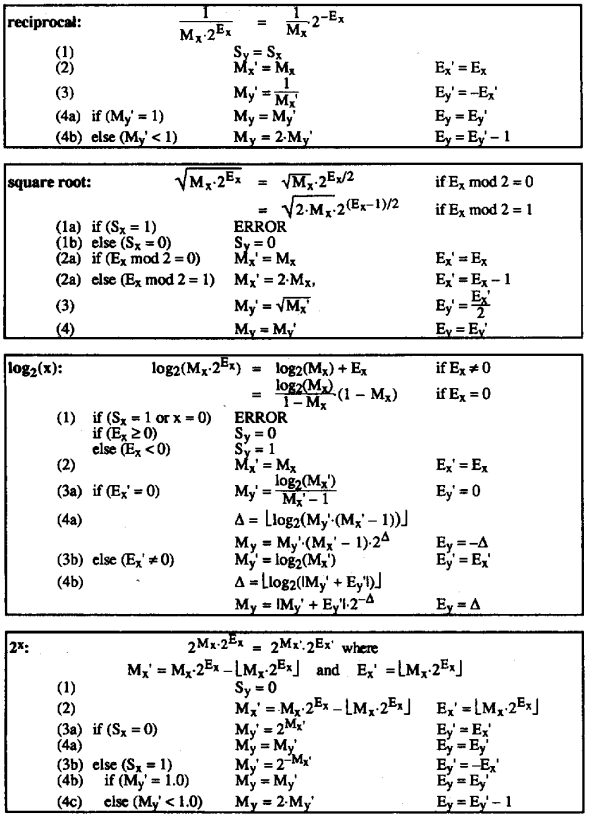


Fig. 5. Argument reduction transformations for the elementary functions.

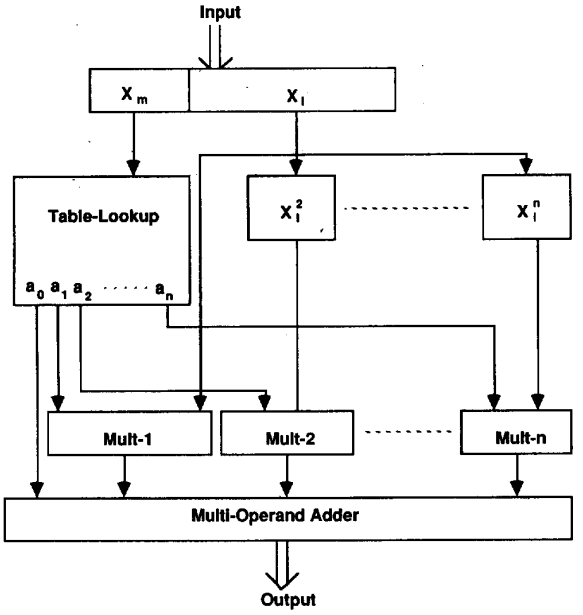


Fig. 6. Block diagram of an elementary function generator.

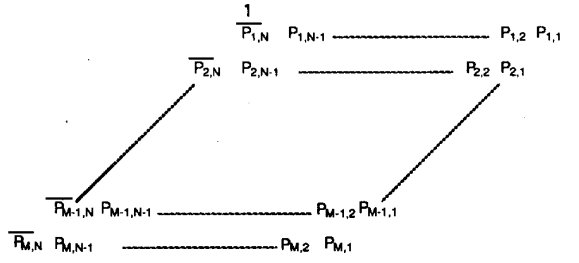


Fig. 7. Partial products of a parallel N-bit-by-M-bit multiplier.

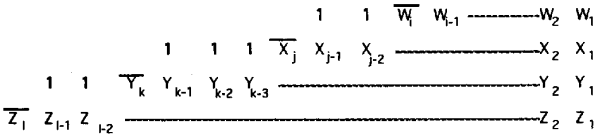


Fig. 8. Two's complement multi-operand adder.

this multiplier are shown in Fig. 7. To avoid sign extension, the sign bit of each partial products is complemented and a 1 is added to the Nth column. This is similar to the method of sign extension presented in [23]. As developed in [24], pseudo adders are applied in parallel to reduce the partial products to two numbers whose sum is equal to the product of the two inputs. The resulting two numbers are then added with a carry look-ahead adder [25] to form the product.

The multioperand adder sums two's complement numbers. The high-order terms in the approximation will have leading ones or zeros. Sign extension of these terms is performed as shown in Fig. 8, where W, X, Y, and Z are the four terms to be added. For cubic approximations W, X, Y, and Z correspond to $a_3 \cdot x_l^3, a_2 \cdot x_l^2, a_1 \cdot x_l$, and a_0, respectively. The most significant bit of each addend is complemented, and ones are added to the appropriate columns. A parallel reduction process [24], followed by carry look-ahead addition, is employed to compute the result. Instead of adding the 1's during the computation, they are added to the coefficient a_0 before it is stored in the lookup table.

For polynomial approximations of degree 2 or greater, the value of x_l^2 is generated with either special-purpose hardware or by a table lookup operation. Special purpose squaring circuits take advantage of the symmetry obtained when a number is multiplied by itself. A 6-by-6 squaring matrix is shown in Fig. 9, where $S_{i,j}$ is the AND of the ith and jth bits of the number being squared. Since the squaring matrix is symmetric, $S_{i,j} = S_{j,i}$ and $S_{i,j} + S_{j,i} = 2S_{i,j}$. The value $2S_{i,j}$ is obtained by shifting $S_{i,j}$ left one position. This reduces the squaring matrix to the sum of the upper half of the squaring matrix shifted left by one position and the antidiagonal. AND gates are not needed to compute the partial product bits on the antidiagonal, because (S_i AND S_i) $= S_i$. The simplified squaring matrix is shown in Fig. 10. In [26], methods are given for further reducing the hardware complexity, and computation time of squaring circuits.

An alternative to the design shown in Fig. 6 is to merge the multiplications and their summation. Implementations of merged arithmetic and a discussion of its advantages are given in [27]. With this approach the multiplications and additions

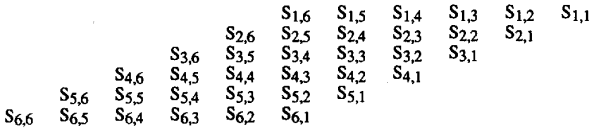


Fig. 9. 6-by-6 squaring matrix.

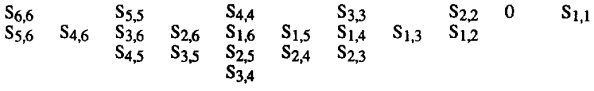


Fig. 10. Simplified 6-by-6 squaring matrix.

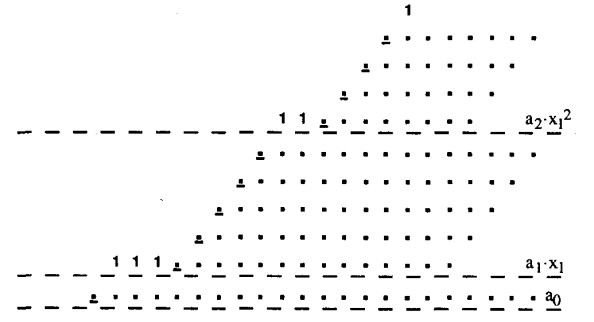


Fig. 11. Dot diagram of a 3-term, merged arithmetic, carry-save adder tree.

are merged into a single carry-save adder tree. A dot diagram [28] of a carry-save adder tree for a three-term approximation is shown in Fig. 11. Each dot corresponds to a partial product bit of $a_i x_l^i$ or a bit in a_0. Dots with lines under them correspond to complemented bits. The technique for sign extension is similar to that used for the multi-operand adder. As before, the ones required for sign extension are added to a_0 before it is stored in the lookup table. Since only a single carry look-ahead adder is used for the entire tree, the overall delay and area are potentially reduced.

VI. Delay and Area Estimates

The hardware required to obtain exactly rounded results for the functions of reciprocal, square-root, 2^x, and $\log_2(x)$ was determined through computer simulation. The simulation first determines the coefficients of the Chebyshev series approximation for each subinterval for a given hardware specification. It then simulates the computation of each function for all values on the input interval and adjusts the coefficients, using the algorithm presented in Section III. The exactly rounded value of the function is determined by rounding the IEEE double-precision value of the function using round-to-nearest-even. Initially, the length of each coefficient (p_i), including leading zeros, is selected to be 1.5 times the length of the significand (p), and the number of bits in the table lookup (k) is selected according to (14) with $q = 1.5 \cdot p$. If exactly rounded results are not produced, the hardware specification is adjusted, and the algorithm is run again. The simulation was performed for numbers with 16- and 24-bit significands, using linear, quadratic, and cubic approximations.

TABLE I
Hardware Requirements for Exactly Rounded Results

Approx.	Mult1	Mult2	Mult3	Square	Cube	Adders
linear(16)	15x5(16)					24, 16
quad(16)	19x8(21)	12x10(14)		8(10)		24 ,21 ,14
cubic(16)	22x10(23)	17x16(18)	12x12(13)	10(16)	10(12)	25, 23, 18, 13
linear(24)	21x6(22)					36, 22
quad(24)	31x12(33)	19x16(21)		12(16)		40, 33, 21
cubic(24)	35x15(37)	27x24(29)	18x14(20)	15(24)	14(14)	41, 37, 29, 20

TABLE II
Memory Requirements for Exactly Rounded Results

	Coefficient Lengths				Table Size		
Approx.	a_0	a_1	a_2	a_3	Words	Bits/Word	Total Bits
linear(16)	24	15			4 K	39	156 K
quad(16)	24	19	12		512	55	27.5 K
cubic(16)	25	22	17	12	128	76	9.5 K
linear(24)	36	21			512 K	57	28.5 M
quad(24)	40	31	19		16 K	90	1.44 M
cubic(24)	41	35	27	18	1 K	121	121 K

The hardware requirements for designs that compute all four functions are shown in Table I. For each multiplier circuit, the number of bits in the multiplicand and multiplier are given, and the number of bits in the rounded product is shown in parenthesis. The number of input bits and output bits is given for the Square and Cube circuits. For the multi-operand adder, the number of bits in each of the addends is listed. The lengths of the coefficients (excluding leading zeros) and the memory requirements are shown in Table II.

Delay and area estimates for hardware units that compute all four functions are shown in Figs. 12 and 13, respectively. These estimates are based on data from a 1.0-micron CMOS standard cell library [29], and do not take into account the delay and area needed to perform argument reduction. The estimates for obtaining exactly rounded results are in black. In the cubic-1 design, x_l^3 is obtained by a table lookup on x_l. In the cubic-2 design, x_l^3 is computed by multiplying x_l by x_l^2. The cubic-1 design uses more area, but has less delay than the cubic-2 design.

As the number of terms in the approximation increases, the area required for the table lookup decreases, and the area needed for the multipliers and the multi-operand adder increases. For numbers with 16-bit significands, the quadratic approximation requires the lowest area, whereas the linear approximation has the lowest delay. For numbers with 24-bit significands, the cubic-2 approximation requires the lowest area, and the quadratic approximation has the lowest delay. The linear approximation cannot be practically implemented for numbers with 24-bit significands, because of its high memory requirements. If the delay-area product is used as the design criterion, then the quadratic and cubic-2 approximations are the best designs for numbers with 16- and 24-bit significands, respectively. The delay-area product for each design is shown in Fig. 14. In comparison, a 24-by-24-bit multiplier in the same technology has a delay of 34 ns, an area of 16 mm^2, and a delay-area product of 544 ns·mm^2.

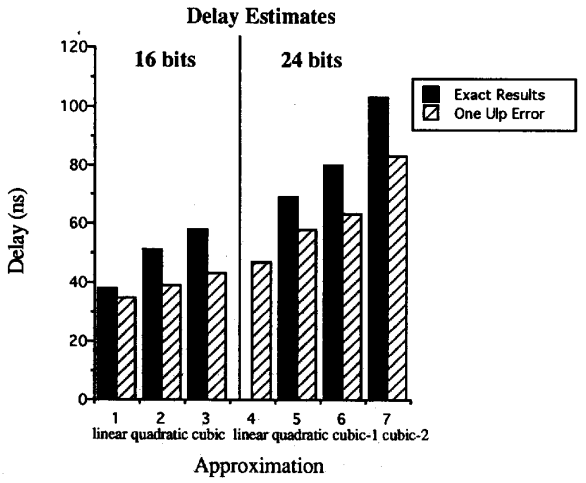


Fig. 12. Delay estimates.

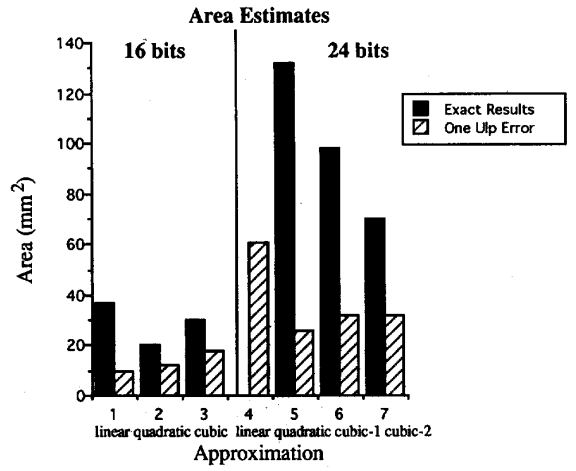


Fig. 13. Area estimates.

Exact rounding produces approximations with a maximum error of half an ulp. If results are allowed to have a maximum error of one ulp, the delay is reduced by 5% to 30%, and the area is reduced by 33% to 77%. These estimates correspond to the shaded bars in Figs. 12–14. For approximations that have a maximum error of one ulp, the best designs are obtained with linear and quadratic approximations for numbers with 16- and 24-bit significands, respectively.

To illustrate the trade-offs in the designs in more detail, block diagrams of hardware units for elementary function generation are shown in Figs. 15–18. The coefficients are selected based on the most significant part of the input, x_m, and the function to be evaluated, $f(x)$. The "Sign-Logic" determines the sign of the output based on the sign of the input and the function being evaluated. It generates an error signal when a negative input is given for the functions of logarithm and square-root. The linear approximation shown in Fig. 17 uses a multiply-and-add unit instead of a separate multiplier and adder to compute the value $a_0 + a_1 \cdot x_l$. Since a_0 is added

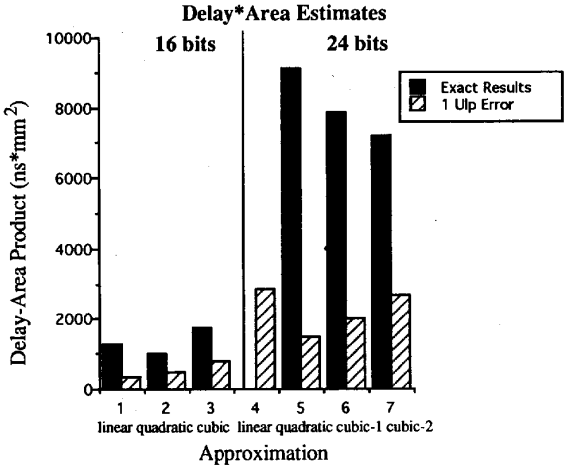


Fig. 14. Delay*area estimates.

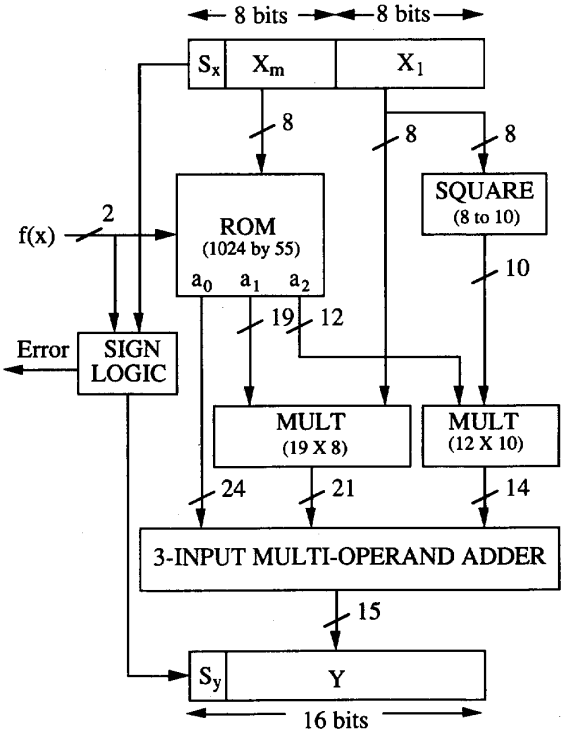


Fig. 15. Quadratic approximation: 16 bits (exact).

along with the partial products of $a_1 \cdot x_l$, the total area and delay of the unit are reduced.

VII. Sufficient Accuracy for Exactly Rounded Results

This section presents a method for determining the accuracy of the approximation that is sufficient to guarantee exactly rounded results. This method requires the function to be evaluated for all values on the input interval, and thus is not applicable for numbers with large word lengths (e.g.,

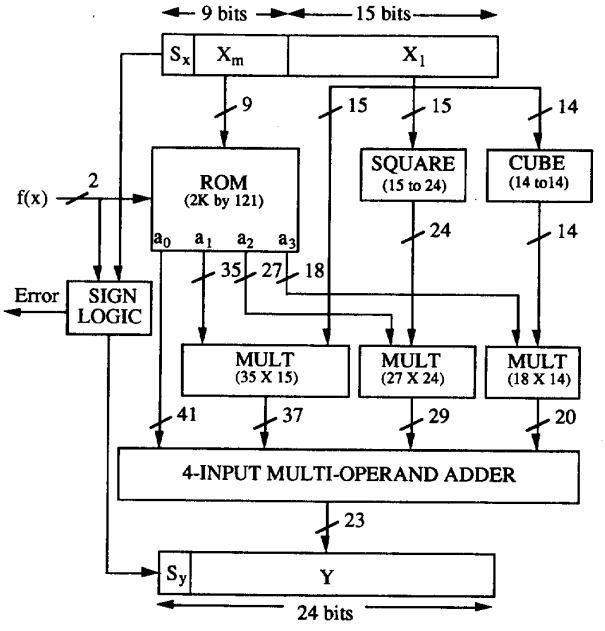


Fig. 16. Cubic approximation: 24 bits (exact).

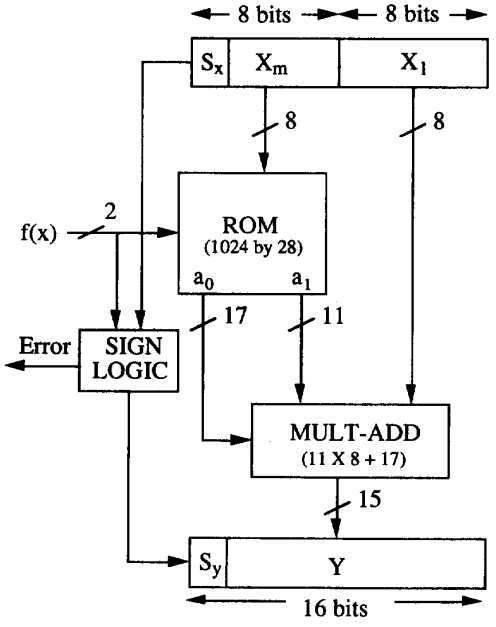


Fig. 17. Linear approximation: 16 bits (1 ulp error).

double-precision). The results presented here are used to show the reduction in area and delay realized by adjusting the coefficients of the approximation.

For most elementary functions, there is no known theoretical method to determine in advance the accuracy of the prerounded result that is sufficient to guarantee that the final result will be exactly rounded. This problem is known as the Table Maker's Dilemma [12], [13]. For example, suppose the

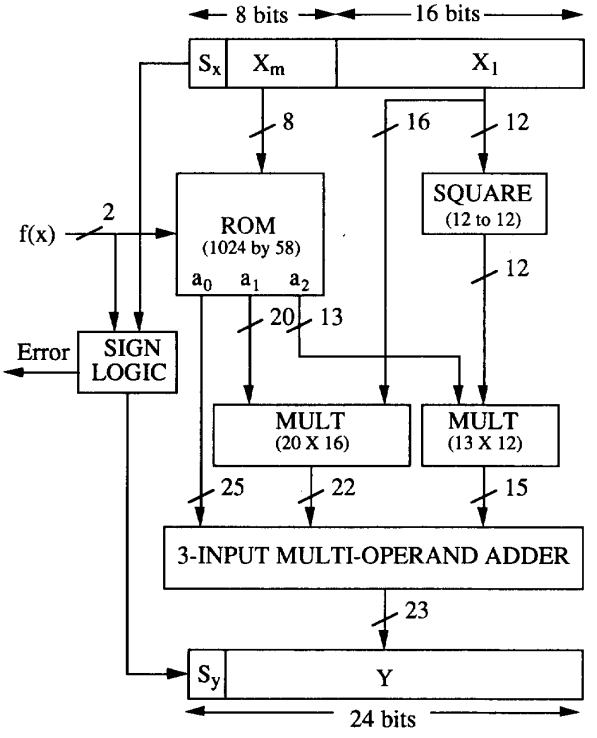


Fig. 18. Quadratic approximation: 24 bits (1 ulp error).

value of a function $f(x)$ when computed to 4 bits is 0.0001_2. It cannot be determined whether the 3-bit exactly rounded result should be 0.001_2 or 0.000_2. If $f(x)$ computed to 5 bits is 0.00010_2, the 3-bit exactly rounded result still cannot be determined. For transcendental functions, an arbitrary number of accurate bits may need to be computed before it can be determined whether $f(x)$ is $0.000100\cdots001XXX$ or $0.000011\cdots111XXX$. Because of this problem, [12] and [13] claim that it is not practical to require that the results of elementary functions are exactly rounded. As described below, however, the accuracy that is sufficient to guarantee exact rounding can be determined via simulation for a given floating point format.

To ensure exact rounding for the elementary functions, it is sufficient to guarantee the following:

1) The prerounded result is less than 0.5 ulp from the exactly rounded value of the function.
2) The prerounded result is rounded by using round-to-nearest.

Fig. 19 illustrates this requirement. If $f(x)$ is the exact value of the function and $p(x)$ is the value of the prerounded result, the following statement holds:

$$\text{IF } |p(x) - f(x)| < 0.5\cdot\text{ulp} - |[f(x)]_p - f(x)| \text{ THEN}$$
$$|p(x) - [f(x)]_p| < 0.5\cdot\text{ulp}$$
$$\text{AND}$$
$$[p(x)]p = [f(x)]p, \tag{16}$$

where $[x]_p$ is the value of x rounded to p bits, using round-

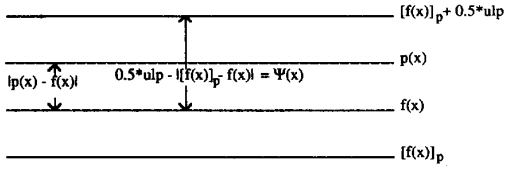


Fig. 19. Exactly rounded results.

TABLE III
SUFFICIENT ACCURACY FOR 16 AND 24-BIT NUMBERS

	16 bit significand		24 bit significand	
Function	$\Psi(x)_{min}$	Accurate Bits	$\Psi(x)_{min}$	Accurate Bits
reciprocal	$2.33 \cdot 10^{-10}$	32	$3.56 \cdot 10^{-15}$	48
square root	$2.33 \cdot 10^{-10}$	32	$3.56 \cdot 10^{-15}$	48
$\log_2(x)$	$3.69 \cdot 10^{-11}$	35	$6.11 \cdot 10^{-16}$	51
2^x	$2.37 \cdot 10^{-9}$	29	$6.21 \cdot 10^{-15}$	48

to-nearest. Therefore, if the distance between the prerounded result and the exact value of the function is less than $\Psi(x)$, where the following holds:

$$\Psi(x) = 0.5 \cdot \text{ulp} - |[f(x)]_p - f(x)|, \tag{17}$$

exact rounding is guaranteed. $\Psi(x)$ corresponds to the distance between the exact value of the function and the nearest rounding boundary.

It is important to note that the condition given in (16) is sufficient, but not necessary, to ensure exactly rounded results. If the error in the prerounded result is greater than $\Psi(x)$, an exactly rounded result is produced if the prerounded result and the exact value of the function are on the same side of the rounding boundary. It is also not possible for $\Psi(x)$ to have a value of zero, which corresponds to the true value of the function being exactly halfway between two consecutive floating point numbers. This is proven in [1] for reciprocal, and a similar argument holds for square-root. Similarly, the function $y = \log_2(x)$ is transcendental and has only rational results when x is an integer power of 2. In this case, y is equal to the exponent of x, which cannot be exactly halfway between two consecutive floating point numbers. The function $y = 2^x$ is also transcendental and has only rational results when x is an integer. In this case, the significand of y is $1.00\cdots000$ (which is represented exactly), and the exponent of y is equal to x.

Based on the previous discussion, the accuracy in the prerounded result that will guarantee exact rounding can be determined by finding the minimum value of $\Psi(x)$ for all numbers on the input interval. The minimum value of $\Psi(x)$ for floating point numbers with 16- and 24-bit significands, along with the number of accurate bits in the normalized, prerounded result is shown in Table III. The following is the number of accurate bits necessary to assure exact rounding:

$$\text{Accurate Bits} = -\lceil \log_2(\Psi(x)_{\min}) \rceil. \tag{18}$$

Table IV shows the maximum error and the minimum number of accurate bits in the prerounded result for each of the designs. Comparing these values to those given in Table III shows that our algorithm requires much less accuracy in the prerounded result. For example, Table III shows that

TABLE IV
MAXIMUM ERROR AND MINIMUM NUMBER OF ACCURATE BITS

	reciprocal		square root		$\log_2(x)$		2^x	
Approx.	Max Error	Bits	Max Error	Bits	Max Error	Bits	Max Error	Bits
linear(16)	$3.12 \cdot 10^{-7}$	22	$1.32 \cdot 10^{-7}$	23	$1.64 \cdot 10^{-7}$	23	$1.32 \cdot 10^{-7}$	23
quad(16)	$1.23 \cdot 10^{-7}$	23	$9.14 \cdot 10^{-8}$	24	$7.75 \cdot 10^{-8}$	24	$8.52 \cdot 10^{-9}$	24
cubic(16)	$4.36 \cdot 10^{-8}$	25	$2.58 \cdot 10^{-8}$	26	$2.36 \cdot 10^{-8}$	26	$3.22 \cdot 10^{-8}$	25
linear(24)	$2.90 \cdot 10^{-11}$	35	$2.01 \cdot 10^{-11}$	36	$1.90 \cdot 10^{-11}$	36	$1.88 \cdot 10^{-11}$	36
quad(24)	$2.94 \cdot 10^{-12}$	39	$3.45 \cdot 10^{-11}$	39	$3.83 \cdot 10^{-12}$	38	$2.40 \cdot 10^{-12}$	39
cubic(24)	$1.55 \cdot 10^{-12}$	40	$9.45 \cdot 10^{-13}$	40	$9.62 \cdot 10^{-13}$	40	$9.15 \cdot 10^{-13}$	40

for reciprocal, 48 bits of accuracy are sufficient to guarantee exactly rounded results for floating point numbers with 24-bit significands. However, our algorithm requires only 35, 39, and 40 bits of accuracy for the linear, quadratic, and cubic designs, respectively. This is because the algorithm uses knowledge about the exactly rounded result to adjust the coefficients.

Estimations were made to determine the overall delay and area required to produce exactly rounded results with a Chebyshev series approximation in which the coefficients are not adjusted. For numbers with 16-bit significands, the design for a quadratic approximation has a delay of approximately 65 ns and an area of 39 mm^2. Compared to the quadratic design in which the coefficients are adjusted, this design has an increase in delay of 27% and an increase in area of 95%. For numbers with 24-bit significands, the design for a cubic approximation has a delay of 128 ns and an area of 165 mm^2. This is an increase in delay of 24% and an increase in area of 136%, compared to the cubic-2 approximation with adjusted coefficients.

VIII. CONCLUSION

A parallel algorithm has been presented that produces exactly rounded results for the functions of reciprocal, square-root, 2^x, and $\log_2(x)$. Area and performance estimates illustrate the feasibility of obtaining exactly rounded results with special-purpose hardware. By adjusting the coefficients based on the error in the original approximation, exactly rounded results are obtained with much less hardware than designs that use traditional polynomial approximations. Allowing the results to have a maximum error of one ulp decreases the area and delay, and is suitable for applications in which stringent error control is not required. Compared to previous methods, our algorithm has a shorter execution time, yet greater hardware complexity. By varying the number of terms in the approximation and the size of the table lookups, trade-offs can be made in terms of the area, delay, and accuracy of the approximation. Specially designed multipliers, squaring circuits, and multi-operand adders reduce the hardware complexity and improve performance.

ACKNOWLEDGMENT

The authors would like to thank D. Matula and U. Kulisch and his students for providing useful comments on the content of the paper. We are grateful to V. K. Jain, who introduced us to a method for elementary function generation that led to

many of the ideas presented in this paper. Comments from the anonymous referees were very useful in revising the paper.

REFERENCES

[1] P.W. Markstein, "Computation of elementary functions on the IBM RISC System/6000 processor," *IBM J. Research Dev.*, vol. 34, pp. 111–119, 1990.

[2] W. Cody and W. Waite, *Software Manual for the Elementary Functions.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

[3] C.T. Fike, *Computer Evaluation of Mathematical Functions.* Englewood Cliffs, NJ: Prentice-Hall, 1968.

[4] J.E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.*, vol. EC-8, pp. 330–334, 1959.

[5] M.J. Flynn, "On division by functional iteration," *IEEE Trans. Comput.*, vol. C-19, pp. 702–706, 1970.

[6] I. Koren and O. Zinaty, "Evaluation of elementary functions in a numerical co-processor based on rational approximations," *IEEE Trans. Comput.*, vol. 39, pp. 1030–1037, 1990.

[7] M.D. Ercegovac, "Radix-16 evaluation of certain elementary functions," *IEEE Trans. Comput.*, vol. C-22, pp. 561–566, 1973.

[8] A.S. Noetzel, "an interpolating memory unit for function evaluation: analysis and design," *IEEE Trans. Comput.*, vol. 38, pp. 377–384, 1989.

[9] G.H. Garcia and W.J. Kubitz, "Minimum mean running time function generation using read only memory," *IEEE Trans. Comput.*, vol. C-32, pp. 147–156, 1983.

[10] P.T.P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," *Proc. 10th Symp. Comput. Arithmetic*, 1991, pp. 232–236.

[11] American National Standards Institute, "IEEE Standard 754 for binary floating point arithmetic," *ANSI/IEEE Standard No. 754*, Washington DC, 1985.

[12] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surv.*, vol. 23, pp. 5–48, 1991.

[13] D. Hough, "Elementary functions based on IEEE arithmetic," *Mini/Micro West Conf. Rec.*, 1983, pp. 1–4.

[14] S. Gal and B. Bachelus, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Trans. Mathematical Software*, vol. 17, pp. 26–45, 1991.

[15] C.M. Black, R.P. Burton, and T.H. Miller, "The need for an industry standard of accuracy for elementary function programs," *ACM Trans. Mathematical Software*, vol. 1, pp. 361–366, 1984.

[16] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Trans. Mathematical Software*, vol. 17, pp. 410–423, 1991.

[17] M.J. Schulte and E.E. Swartzlander, "Exact rounding of certain elementary functions," *Proc. 11th Symp. Comput. Arithmetic*, 1993, pp. 138–145.

[18] J.H. Mathews, *Numerical Methods for Computer Science, Engineering and Mathematics.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

[19] P.M. Farmwald, "High bandwidth evaluation of elementary functions," *Proc. 5th Symp. Comput. Arithmetic*, 1981, pp. 139–142.

[20] V.K. Jain, "Arithmetic analysis of a new reciprocal cell," *1992 Int. Conf. Comput. Design: VLSI in Comput. and Processors*, 1992, pp. 106–109.

[21] V.K. Jain, S.A. Wadekar, and L. Lin, "Universal nonlinear component and its application to WSI," *IEEE Trans. Components, Hybrids, and Manufacturing Technol.*, vol. 16, pp. 656–664, 1993.

[22] V.K. Jain, G.E. Perez, and J.M. Wills, "DSP coprocessor cell for systolic arrays," *VLSI Signal Processing*, vol. VI, pp. 480–488, 1993.

[23] C.R. Baugh and B.A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, pp. 1045–1047, 1973.

[24] C.S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14–17, 1964.

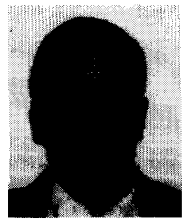
[25] R.P. Brent and H.Y. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, pp. 260–264, 1982.

[26] A. Weinberger and J.L. Smith, "A logic for high-speed addition," *Nat. Bureau of Standards Circular*, no. 591, pp. 3–12, 1958.

[27] E.E. Swartzlander, Jr., "Merged arithmetic," *IEEE Trans. Comput.*, vol. C-29, pp. 946–950, 1980.

[28] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

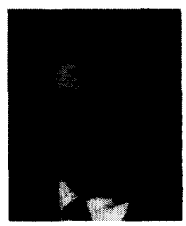
[29] LSI Logic Corp., *LSI Logic 1.0 Micron Cell-Based Products Databook.* Milpitas, CA: 1991.



Michael J. Schulte (S'92–M'93) received the B.S. degree in electrical engineering from the University of Wisconsin, Madison, in 1991, and the M.S. degree in electrical engineering from the University of Texas at Austin in 1993.

He is currently pursuing the Ph.D. degree in electrical engineering at the University of Texas at Austin, with support from a TRW graduate fellowship. Previously, he received support from the Microelectronic and Computer Development Fellowship and the Basdall Gardner Memorial Fellowship. His research interests include computer arithmetic, computer architecture, and the design of special-purpose processors.

Mr. Schulte is a member of the University Bible Fellowship, Mission Possible, and Friends of Jesus.



Earl E. Swartzlander, Jr. (S'64–M'72–SM'79–F'88) received the B.S. degree in electrical engineering from Purdue University, West Lafayette, IN, in 1967, the M.S. degree in electrical engineering from the University of Colorado in 1969, and the Ph.D. degree from the University of Southern California, Los Angeles, CA, in 1972.

In 1990, he became a Professor of Electrical and Computer Engineering at the University of Texas at Austin, where he holds the Schlumberger Centennial Chair in Engineering. Previously he was with TRW for 15 years, where (among other assignments) he managed the Independent Research and Development program for TRW Defense Systems Group from 1987 to 1990, and managed the Digital Processing Laboratory in TRW Electronic Systems Group from 1985 to 1987. His research interests are in application-specific computing and the interaction between computer architecture and technology.

He is currently Treasurer of the IEEE Solid-State Circuits Council, a member of the IEEE Signal Processing Society ADCOM, and a former member of the IEEE Computer Society Board of Governors (1987 to 1991). Currently, he is Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS, the Hardware Area Editor for *ACM Computer Reviews*, and the founding Editor-in-Chief of the *Journal of VLSI Signal Processing*. He was an Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (1989 to 1990), an Editor of the IEEE TRANSACTIONS ON COMPUTERS (1982 to 1986), and an Associate Editor of the IEEE JOURNAL OF SOLID-STATE CIRCUITS (1984 to 1988). He was the General Chair of the first three IEEE Real-Time Systems Symposia (1980-1982), the 5th IEEE International Conference on Distributed Computing Systems (1985), and the first IEEE International Conference on Wafer Scale Integration (1989). He was Co-General Chair of the 1990 Applications Specific Array Processors Conference and the 3rd International Conference on Parallel and Distributed Systems. He was Co-Program Chair of the 9th Symposium on Computer Arithmetic, and the General Chair of the 11th Symposium on Computer Arithmetic. He obtained his doctorate in computer design with the support of a Howard Hughes Doctoral Fellowship. He has written the book, *VLSI Signal Processing Systems* (Kluwer Academic Publishers 1986), and has edited five books, including two volumes of reprints on computer arithmetic (IEEE Computer Society Press 1990). He has written approximately 100 papers in the fields of computer arithmetic, signal processing, and very large scale integration (VLSI) implementation. He belongs to Eta Kappa Nu, Sigma Tau, and Omicron Delta Kappa honorary fraternities, and is a registered Professional Engineer in the USA states of Alabama, California, Colorado, and Texas. He is a Knight of the Imperial Russian Order of Saint John of Jerusalem Ecumenical Foundation (Knights of Malta) and is an Outstanding Electrical Engineer and a Distinguished Engineering Alumnus of Purdue University.