Interval Computations
No 4, 1993

Parallel Hardware Designs for Correctly Rounded Elementary Functions

Michael J. Schulte and Earl E. Swartzlander, Jr.

This paper presents an algorithm for evaluating the functions of reciprocal, square root, 2^x, and $\log_2(x)$ with special purpose hardware. For these functions, the algorithm produces correctly rounded results, according to a specified rounding mode. This algorithm can be used to implement directed rounding which is essential for interval arithmetic, or exact rounding which minimizes the maximum error of the result. Hardware designs based on this algorithm are discussed. These designs use a polynomial approximation in which the coefficients are originally selected based on the Chebyshev series approximation and are then adjusted to ensure correctly rounded results for all inputs. The terms in the approximation are generated in parallel and are then summed using a high-speed, 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. Range reduction techniques that maintain correct rounding are presented. Area and delay comparisons are made based on the degree of the polynomial and the accuracy of the final result. For single-precision floating point numbers, the correctly rounded value of the function can be computed in approximately 103 ns on a 70 mm^2 chip.

Проектирование параллельного аппаратного обеспечения для вычисления корректно округленных элементарных функций

М. Й. Шульте, Е. Е. Шварцландер, мл.

Представлен алгоритм для вычисления функций $\frac{1}{x}$, $\sqrt{x}$, 2^x и $\log_2(x)$ с помощью специализированного аппаратного обеспечения. Алгоритм обеспечивает корректное округление этих функций в соответствии с заданным

режимом округления и может применяться для реализации как направленного округления в интервальной арифметике, так и точного округления с целью минимизации погрешности результата. Обсуждаются способы аппаратной реализации этого алгоритма, использующие приближение многочленами, коэффициенты которых сначала выбираются на основе приближения ряда Чебышева, а затем корректируются для обеспечения корректного округления результатов при любых входных данных. Члены ряда вычисляются параллельно и затем суммируются высокоскоростным многооперандным сумматором. Для уменьшения числа членов входной интервал делится на подынтервалы одинакового размера, в каждом из которых применяются различные коэффициенты. Представлена методика понижения ранга, обеспечивающая корректное округление. Приводятся сравнительные данные о времени вычислений и требуемой площади кристалла для различных степеней многочлена и точности результата. Для чисел с плавающей точкой одинарной точности корректно округленное значение функции вычисляется приблизительно за 103 *нс* на кристалле площадью 70 $мм^2$.

1 Introduction

The rapid and accurate evaluation of the elementary functions (e.g., reciprocal, square root, exponential, logarithm, etc.) is important for a number of scientific applications. Computation of these functions is often performed by software routines which employ various techniques including polynomial approximation, rational expressions, and continued fraction expansion [1]. The disadvantage of most software routines is that they do not guarantee last bit accuracy and are often too slow for numerically intensive applications. To overcome the speed disadvantage of software routines, several algorithms have been developed for approximating the elementary functions with special purpose hardware. These algorithms include the CORDIC algorithm [2], Newton-Raphson iteration [3], rational approximations [4], and polynomial approximations [5]. While hardware algorithms typically have a speed advantage over software routines, they often produce even less accurate results. In addition, their speed advantage is limited because they are usually implemented iteratively and a large number of iterations may be required.

The IEEE 754 standard [9] requires correct rounding for addition, subtraction, multiplication, division, square root, remainder, and conversion between integer and floating point formats. Correct rounding requires the rounded result to be identical to the result obtained if the infinitely precise value of the function is rounded according to a specified rounding mode.

The IEEE 754 standard specifies four rounding modes: round to nearest, round toward $+\infty$, round toward $-\infty$, and round toward zero. Round to nearest, also known as exact rounding, is the default rounding mode. The other three rounding modes, the directed rounding modes, are often used for interval arithmetic [7, 8]. The IEEE 754 standard, however, does not require correct rounding for the elementary functions. This is largely due to a problem known as the Table Maker's Dilemma [10, 11]. This problem and a solution to it are discussed in Section 6.

Interval arithmetic provides a systematic technique for keeping track of rounding errors and errors that occur due uncertainly in initial values. As discussed in [6], providing hardware platforms which support directed rounding improves the speed and efficiency of interval arithmetic. Furthermore, depending on the instruction set of the computer, it can be difficult, or impossible, to implement correct rounding of the elementary functions in software. In most cases, several extra bits of precision are required for internal computation. Software routines can also be orders of magnitude slower than hardware implementations. As a result, most existing mathematical libraries do not provide directed rounding for the elementary functions. Because there is a lack of hardware and software support for directed rounding of the elementary functions, it is often difficult to implement interval arithmetic for applications which use these functions.

To illustrate the use of directed rounding of the elementary functions, suppose the result of a function $f(x)$ is required for the interval $X = [a, b]$ and $f(x)$ is monotonically increasing on $[a, b]$. If we denote round toward $+\infty$ as Δ and round toward $-\infty$ as ∇, then the resultant interval is $Y = [\nabla f(a), \Delta f(b)]$. Without directed rounding modes, optimal bounds on the resultant interval cannot be established.

In addition to the benefits offered for the interval arithmetic, evaluating the elementary functions with an algorithm that produces correctly rounded result has several other advantages. Correct rounding limits the maximum error to one unit in the last place (ulp) for the directed rounding modes and half an ulp for round to nearest. 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. Correct rounding also ensures that machines which have the same floating point format will 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 sys-

tem. Correct rounding with round to nearest (i.e., exact rounding) is useful in elementary function evaluation because it minimizes the error between the rounded result and the exact value of the function. Exact rounding also preserves several desirable properties of the functions such as symmetry and monotonicity [12]. Other advantages of having an specified standard for the elementary functions are given in [13].

Because of the advantages offered, much research has been performed to develop software routines which produce correctly rounded results for the elementary functions. In [12], software routines are described that use accurate range reduction techniques, followed by an iterative polynomial approximation to compute the elementary functions for floating point numbers in the IEEE double precision format. Although most of these routines achieve correct rounding for over 99.8 percent of the elementary functions, they require more than 70 machine cycles to execute on a general purpose computer. Routines which are expected to produce correctly rounded results for elementary functions in the IEEE double precision format are described in [14]. For the first iteration, the result is computed using double precision arithmetic. If the pre-rounded result of this routine does not meet a specified error criterion, the result is recomputed using a higher precision routine which may be orders of magnitude slower than the original routine. This is repeated using slower and more precise routines at each iteration until a correctly rounded result is guaranteed. The goal is that the average time to compute the elementary functions will be relatively low since most input values will require only a single iteration. This approach, however, is not practical for real time computations, since hundreds of cycles may be required to compute results which are not correctly rounded after the first iteration. For numerically intensive applications, hardware support for elementary function generation is often required.

This paper presents a parallel hardware algorithm for computing the functions of reciprocal, square root, 2^x and $\log_2(x)$. Because this algorithm performs the computation in parallel and division is not required, it is faster than existing hardware and software algorithms. In addition, the results produced by this algorithm are correctly rounded, making it useful for interval arithmetic. In Section 2, polynomial approximations are discussed with an emphasis on the Chebyshev series approximation. Section 3 presents a novel algorithm by which the coefficients of the polynomials are adjusted to guarantee correctly rounded results. Section 4 presents range reduction

techniques which maintain correct rounding. In Section 5, hardware designs which produce correctly rounded results for floating point numbers with 16 and 24 bit significands are given. Section 5 also examines the reduction in delay and area when the results are allowed to have last bit errors. In Section 6, the difficulty of obtaining correctly rounded results is discussed, and the reduction in delay and area obtained by adjusting the coefficients is examined.

2 Polynomial approximations

The algorithm discussed in this paper uses polynomial approximations to compute the elementary functions. Polynomial approximations have the form

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

where $f(x)$ is the function to be approximated, $q_{n-1}(x)$ is a polynomial of degree $n-1$, and a_i is the coefficient of the i-th term. The function is approximated on a specified input interval $[x_{\min}, x_{\max})$ and range 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 equally sized 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 Figure 1.

If x is on the input interval $[0, 1)$, then

$$x = x_m + x_l \cdot 2^{-k} \qquad (2)$$

where $0 \le x_m < 1$ and $0 \le x_l < 1$. Equation (1) then becomes

$$p_m(x) = a_0(x_m) + a_1(x_m) \cdot x_l + \cdots + a_{n-1}(x_m) \cdot x_l^{n-1} = \sum_{i=0}^{n-1} a_i(x_m) \cdot x_l^i \qquad (3)$$

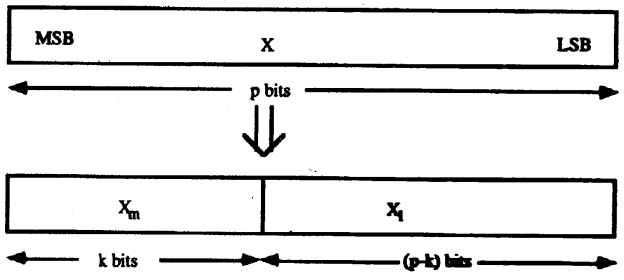

Figure 1: Dividing the input value

where $p_m(x)$ is the approximating polynomial of degree $n-1$ for subinterval m. The a_i's are obtained by a table look-up based 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. Figure 2 illustrates the effect of dividing the input interval into subintervals. Figure 3 shows the approximation for a single subinterval.

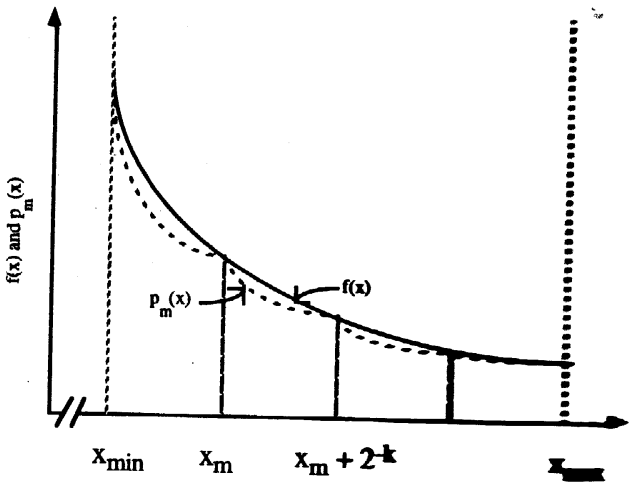

Figure 2: Dividing the input interval

For formalized IEEE floating point numbers [9], the input interval is often $[1, 2)$. For these numbers, x_m consists on the $(k-1)$ most significant bits of x, excluding the most significant bit which is always 1. Numbers of this form are specified by the equation

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

Originally, an approximation to the minimax polynomial, the Chebyshev series approximation, is used to select the coefficients for each of the subin-

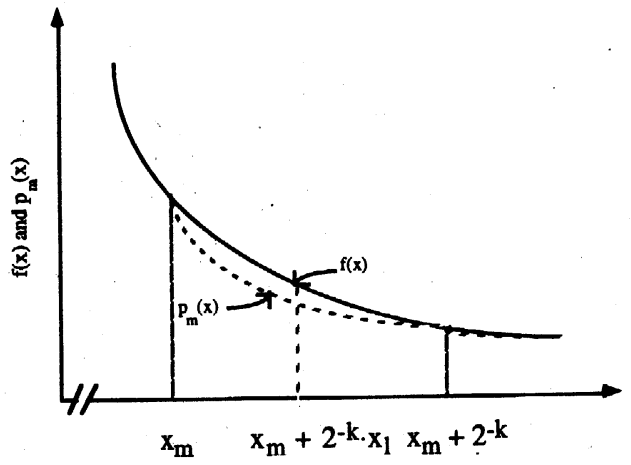

Figure 3: A single subinterval

tervals. The coefficients are then adjusted to obtain correctly rounded results for all values on each subinterval. The Chebyshev series approximation $p_m(x)$ is computed on the subinterval $[x_m, x_m + 2^{-k})$ by the following algorithm [15]:

1. The Chebyshev nodes on $[-1, 1)$ are computed using the formula

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

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

2. The Chebyshev nodes are transformed from $[-1, 1)$ to $[x_m, x_m + 2^{-k})$ through the equation

$$x_i = x_m + (t_i + 1) \cdot 2^{-k-1} \qquad (0 \leq i < n) \tag{6}$$

   where x_i is the i-th Chebyshev node on $[x_m, x_m + 2^{-k})$.

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

$$p_m(x) = y_0 \cdot L_0(x) + y_1 \cdot L_1(x) + \cdots + y_{n-1} \cdot L_{n-1}(x) \tag{7}$$

   where

$$L_i(x) = \frac{(x - x_0) \times \cdots \times (x - x_{i-1}) \times (x - x_{i+1}) \times \cdots \times (x - x_{n-1})}{(x_i - x_0) \times \cdots \times (x_i - x_{i-1}) \times (x_i - x_{i+1}) \times \cdots \times (x_i - x_{n-1})} \tag{8}$$

and

$$y_i = f(x_i). \tag{9}$$

4. $p_m(x)$ is expressed in the form given in equation (3) by combining terms in $p_m(x)$ with equal powers of x_l.

5. The coefficients of $p_m(x)$ are rounded to finite precision using round-to-nearest-even.

The maximum error between a function and its Chebyshev series approximation on an interval $[a, b)$ is

$$E_n(x) \leq \left(\frac{b-a}{4}\right)^n \cdot \frac{2 \cdot f^n(x)}{n!}, \qquad a \leq x < b. \tag{10}$$

Since the input interval is divided into subintervals of size 2^{-k}, the maximum error is

$$E_n(x) \leq \frac{2^{-n(k+2)+1} \cdot f^n(x)}{n!}, \qquad x_m \leq x < x_m + 2^{-k}. \tag{11}$$

This compares favorably with the Taylor series approximation which has a maximum error of

$$E_n(x) \leq \frac{2^{-n \cdot k} \cdot f^n(x)}{n!}, \qquad x_m \leq x < x_m + 2^{-k}. \tag{12}$$

For the Chebyshev series approximation, increasing the number of bits in x_m by one decreases the maximum error by a factor of 2^{-n}, but this doubles the number of coefficients. In comparison, increasing the number of terms by one decreases the maximum error by a factor of approximately $2^{-(k+2)}$, but this increases the required number of multiplies and adds and the width of the table for storing the coefficients.

3 An algorithm for adjusting the coefficients

Polynomial approximations provide a high-speed method for computing the elementary functions. However, to obtain correctly rounded results either the size of the table look-up or the number of terms in the approximation

must be very large. This section describes an algorithm by which the coefficients are adjusted to obtain correctly rounded results. This leads to a significant reduction in the amount of hardware and the overall delay for computing the elementary functions. This algorithm may not be applicable to large word-length numbers (e.g. double-extended precision), because it requires an exhaustive test of all the values on the input interval. The algorithm fro adjusting the coefficients is shown in Figure 4. Array variables are shown in bold face type.

```
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 incorrect approximations [j] using the best coefficients;
  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^(−p_i);
      compute the number of incorrect approximations [j]
      using the best coefficients and a[i][j];
      if (the number of incorrect approximations [j] is reduced) then
        a[i][j] becomes the best coefficient for this subinterval;
        remember the number of incorrect approximations;
      if (the number of incorrect approximations [i][j] is zero) then
        exit this subinterval (exit j);
    end sign
  end k
 end j
 if (the number of incorrect approximations on all subintervals is zero) then
  exit modifying coefficients (exit i);
end i
```

Figure 4: Adjusting the coefficients

To determine the required numbers of iterations, the following two observations are made. It is assumed that the difference between the exact value of the function and the pre-rounded result is less than 2^{-q}. The rounded result has an ulp of 2^{-p}, and the ulp of the coefficient of the i-th term is 2^{-p_i}, where $p \leq q < p_i$.

1. For a given $a_1, a_2, \ldots, a_n$, the maximum number of iterations needed to select the optimal value of a_0 is 2^{p_0-q}. This is verified by observing that adjusting a_0 by 2^{-q} increases the value of every approximation by an amount 2^{-q}. Adjusting a_0 by 2^{-q} requires 2^{p_0-q} iterations. After

this adjustment, every approximation on the subinterval will be above the corresponding value of the function, since the maximum error was originally less than 2^{-q}. Further adjustment of the coefficients will not reduce the number of incorrect approximations. A similar argument holds if a_0 is adjusted by -2^{-q}.

2. For a given $a_0, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1}$, the maximum number of iterations needed to select the optimal value of a_i is $2^{p_i - p}$. Adjusting a_i by 2^{-p_i} changes the value of each approximation on the subinterval by an amount $2^{-p_i} \cdot x_l^i$. After 2^{p_i-p} iterations, the approximation at x_l is adjusted by $2^{-p} \cdot x_l^i$. Since it is require that $|p(x) - f(x)| \leq 2^{-p}$, more than 2^{p_i-p} iterations will lead to results which are not correctly rounded for some value of x_l.

4 Range reduction transformations

Before performing the polynomial approximation, it is necessary to transform the original input value so that it falls within a specified input interval. The value of the function is then computed for the transformed input. This is followed by a second transformation which compensates for the original transformation and normalized the result. The input and output transformation are commonly referred to as range reduction. The range reduction transformations presented in this section maintain correct rounding. Thus, if the results computed over the input interval are correctly rounded, all results will be correctly rounded.

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

$$x = (-1)^{S_x} \cdot M_x \cdot 2^{E_x} \qquad (1 \leq M_x < 2). \tag{13}$$

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

M_x and E_x	The original significand and exponent
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
M_y and E_y	The final output values
S_x and S_y	The sign bits of the input and output values

reciprocal: $\dfrac{1}{M_x \cdot 2^{E_x}} = \dfrac{1}{M_x} \cdot 2^{-E_x}$

(1)	$S_y = S_x$	
(2)	${M_x}' = M_x$	${E_x}' = E_x$
(3)	${M_y}' = \frac{1}{{M_x}'}$	${E_y}' = -{E_x}'$
(4a) if $({M_y}' = 1)$	$M_y = {M_y}'$	$E_y = {E_y}'$
(4b) else $({M_y}' < 1)$	$M_y = 2 \cdot {M_y}'$	$E_y = {E_y}' - 1$

square root: $\sqrt{M_x \cdot 2^{E_x}} = \begin{cases} \sqrt{M_x} \cdot 2^{E_x/2} & \text{if } E_x \bmod 2 = 0 \\ \sqrt{2 \cdot M_x} \cdot 2^{(E_x-1)/2} & \text{if } E_x \bmod 2 = 1 \end{cases}$

(1a) if $(S_x = 1)$	ERROR	
(1b) else $(S_x = 0)$	$S_y = 0$	
(2a) if $(E_x \bmod 2 = 0)$	${M_x}' = M_x$	${E_x}' = E_x$
(2b) else $(E_x \bmod 2 = 1)$	${M_x}' = 2 \cdot M_x$	${E_x}' = E_x - 1$
(3)	${M_y}' = \sqrt{{M_x}'}$	${E_y}' = \frac{{E_x}'}{2}$
(4)	$M_y = {M_y}'$	$E_y = {E_y}'$

log2(x): $\log_2(M_x \cdot 2^{E_x}) = \begin{cases} \log_2(M_x) + E_x & \text{if } E_x \neq 0 \\ \frac{\log_2(M_x)}{1-M_x} \cdot (1 - M_x) & \text{if } E_x = 0 \end{cases}$

(1) if $(S_x = 1$ or $x = 0)$	ERROR	
if $(E_x \geq 0)$	$S_y = 0$	
else $(E_x < 0)$	$S_y = 1$	
(2)	${M_x}' = M_x$	${E_x}' = E_x$
(3a) if $({E_x}' = 0)$	${M_y}' = \frac{\log_2({M_x}')}{{M_x}'-1}$	${E_y}' = 0$
(4a)	$\Delta = \lfloor {M_y}' \cdot (\log_2({M_x}' - 1) \rfloor$	
	${M_y} = {M_y}' \cdot ({M_x}' - 1)2^{-\Delta}$	$E_y = \Delta$
(3b) else $({E_x}' \neq 0)$	${M_y}' = \log_2({M_x}')$	${E_y}' = {E_x}'$
(4b)	$\Delta = \lfloor \log_2(\lvert {E_y}' \rvert) \rfloor$	
	$M_y = \lvert {M_y}' + {E_y}' \rvert \cdot 2^{-\Delta}$	$E_y = \Delta$

2x:

$$2^{M_x \cdot 2^{E_x}} = 2^{{M_x}'} \cdot 2^{{E_x}'},$$

where ${M_x}' = M_x \cdot 2^{E_x} - \lfloor M_x \cdot 2^{E_x} \rfloor$ and ${E_x}' = \lfloor M_x \cdot 2^{E_x} \rfloor$.

(1)	$S_y = 0$	
(2)	${M_x}' = M_x \cdot 2^{E_x} - \lfloor M_x \cdot 2^{E_x} \rfloor$	${E_x}' = \lfloor M_x \cdot 2^{E_x} \rfloor$
(3a) if $(S_x = 0)$	${M_y}' = 2^{{M_x}'}$	${E_y}' = {E_x}'$
(4a)	$M_y = {M_y}'$	$E_y = {E_y}'$
(3b) else $(S_x = 1)$	${M_y}' = 2^{-{M_x}'}$	${E_y}' = -{E_x}'$
(4b) if $({M_y}' = 0.5)$	$M_y = 2 \cdot {M_y}'$	$E_y = {E_y}' - 1$
(4c) else $({M_y}' < 0.5)$	$M_y = 4 \cdot {M_y}'$	$E_y = {E_y}' - 2$

Figure 5: Range reduction for the elementary functions

For the range reduction formulas, multiplication by 2^b corresponds to a b bit left shift, and division by 2^b corresponds to a b bit right shift. Since the input and output transformations for inverse, reciprocal and 2^x do not modify the bit values of the significand, ensuring correctly rounded results on the input interval guarantees correctly rounded results for all inputs. For the output transformation of $\log_2(x)$, if E_x is nonzero, it is necessary to add ${E_y}'$ to the result and then normalize by a right shift of $\lfloor \log_2(|{E_y}'|) \rfloor$ bits. Correct rounding is maintained if the normalized result is rounded using the specified rounding mode. If E_x is equal to zero, leading zeros may appear in $\log_2({M_x}')$ which leads to a loss of precision. Since $1 \leq \frac{\log_2({M_x}')}{{M_x}'-1} < 2$ 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 adjusted to account for the normalization of $({M_x}'-1)$.

5 Hardware designs

This section presents hardware designs for elementary function evaluation, along with their area and delay estimates. Our hardware designs for evaluating the elementary functions are similar to the designs discussed in [5] and [20]. Our designs, however, make use of specialized multipliers, multioperand adders, and squaring circuits which are designed for elementary function approximation. In addition, the size of the table-lookup and the size of the arithmetic units have been tailored to minimize the hardware requirements, while still guaranteeing correct rounding of the elementary functions. The hardware designs presented in [5] and [20] do not guarantee correct rounding.

Once the input transformation has been performed, a polynomial approximation is 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. They are then summed together with a high-speed,

multi-operand adder. Figure 6 shows a block diagram for a polynomial approximation of degree n.

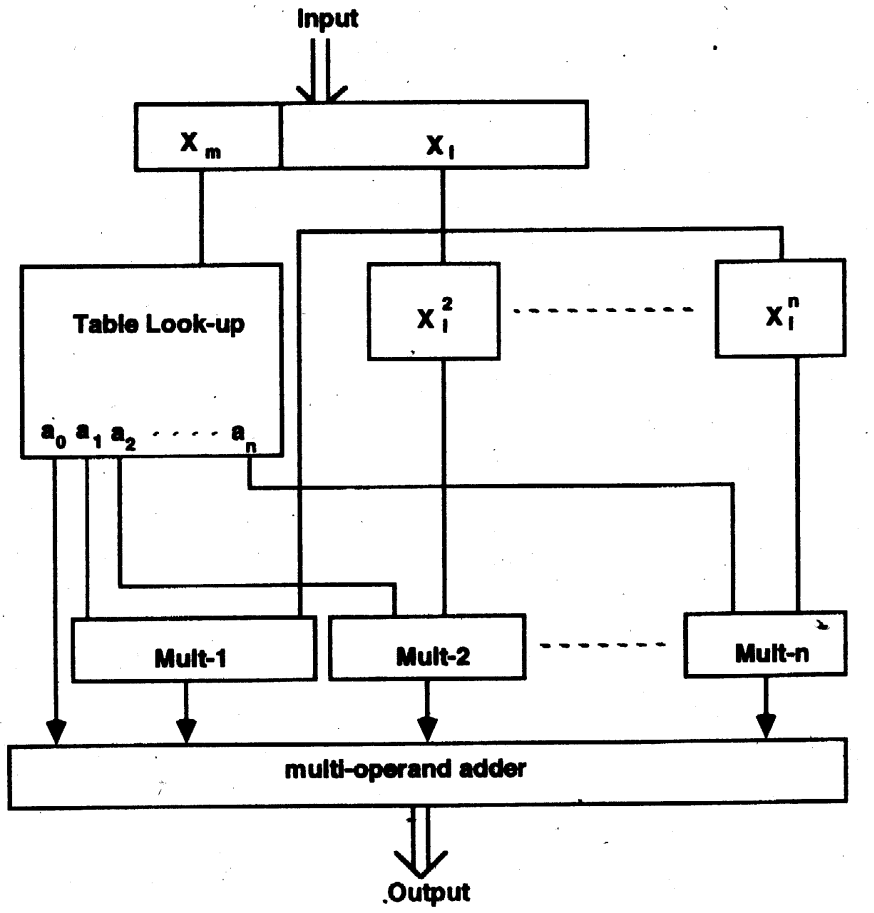

Figure 6: Block diagram of an elementary function generator

To reduce the hardware complexity and area requirements of the elementary function generator, special purpose parallel multipliers and a high-speed, multi-operand adder are designed which take advantage of the characteristics of the polynomial approximations. Since the x_l^i's are guaranteed to be positive and the a_i's can be either positive or negative, each term is computed with an n-bit by m-bit multiplier in which the multiplicand is a two's complement number and the multiplier is always positive. The partial products for this multiplier are shown in Figure 7. To avoid sign extension, the sign bit of each partial products is complemented and a one is added to the N-th column. This is similar to the method of sign extension presented in [16]. As developed in [17] by Wallace, 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 together with a carry look-ahead adder to form the product. This method of multiplication yields delays that are proportional to the logarithm of the size of the input values.

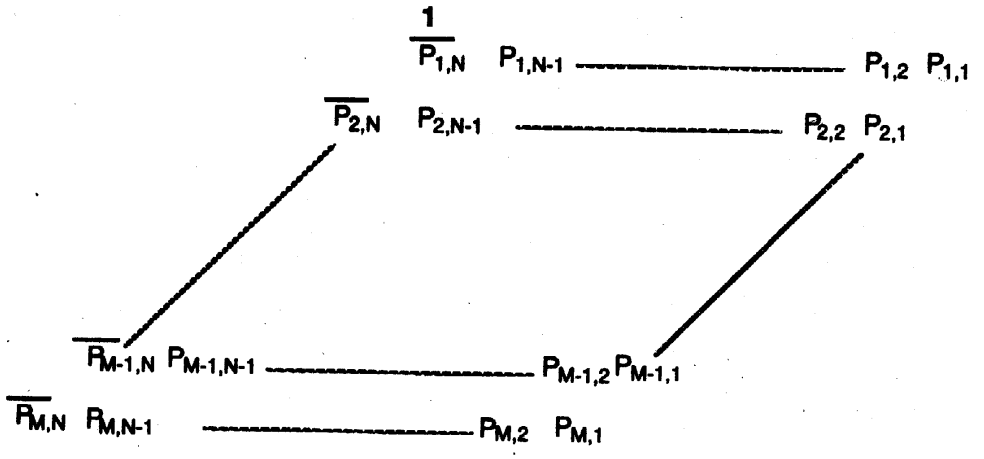

Figure 7: Partial products of a parallel N-bit by M-bit multiplier

The multi-operand adder sums together 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 Figure 8, where W, X, Y, and Z are the four terms to be added. For a cubic approximation 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. A parallel reduction process, followed by carry look-ahead addition, is employed to compute the result. Instead of using extra hardware to add the ones during the computation, they are added to the coefficient a_0 when its value is originally determined.

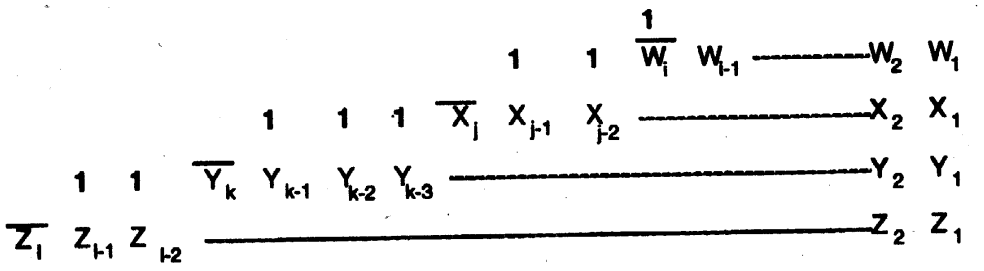

Figure 8: Two's complement multi-operand adder

An alternative to the design shown in Figure 6 is to merge the multiplications and their summation. With this approach, only a single carry look-ahead adder is needed and the overall delay and area are reduced. Implementations of merged arithmetic and a discussion of its advantages are given in [18].

The amount of hardware required to obtain exactly rounded results for the four functions was determined through computer simulation. The simulation first determines the coefficients of the Chebyshev series approximation for each subinterval. 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 3. For the hardware requirements shown in this section, the correctly rounded value of the function is determined by rounding the IEEE double precision value of the function using round to nearest even. Similar results are expected for the other rounding modes. The simulation was performed for numbers with 16 and 24 bit significands using linear, quadratic and cubic approximations.

The hardware requirements for each design are given in Table 1. For the multipliers, 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 inputs is given. The lengths of the coefficients and the memory requirements are shown in Table 2.

Approx.	Mult1	Mult2	Mult3	Square	Cube	Adders
linear(16)	$15 \times 5(16)$					24, 16
quad(16)	$19 \times 8(21)$	$12 \times 10(14)$		8(10)		24, 21, 14
cubic(16)	$22 \times 10(23)$	$17 \times 16(18)$	$12 \times 12(13)$	10(16)	10(12)	25, 23, 18, 13
linear(24)	$21 \times 6(22)$					36, 22
quad(24)	$31 \times 12(33)$	$19 \times 16(21)$		12(16)		40, 33, 21
cubic(24)	$35 \times 15(37)$	$27 \times 24(29)$	$18 \times 14(20)$	15(24)	14(14)	41, 37, 29, 20

Table 1: Hardware requirements for correctly rounded results

	Coefficient lengths				Table size		
Approx.	a_0	a_1	a_2	a_3	Words	Bits/word	Total bits
linear(16)	24	15			6656	39	253.5 K
quad(16)	24	19	12		832	55	44.7 K
cubic(16)	25	22	17	12	208	76	15.4 K
linear(24)	36	21			851,968	57	46.3 M
quad(24)	40	31	19		17,408	90	1.49 M
cubic(24)	41	35	27	18	1,920	121	226.9 M

Table 2: Memory requirements for correctly rounded results

Delay and area estimates for implementing the four functions are shown in Figures 9 and 10. These estimates are based on data from a 1.0 micron CMOS standard cell library [19], and do not take into account the delay and area needed to perform range reduction. The estimates for obtaining correctly rounded results are in black. The cubic-1 estimates are for an elementary function generator in which x_l^3 is obtained by a table look-up on x_l. For the estimates corresponding to cubic-2, x_l^3 is computed by multiplying x_l by x_l^2. The cubic-1 design uses more area, but has shorter delay time than the cubic-2 design. As the number of terms in the approximation increases, the area required for the table look-up decreases, while the area needed for the multipliers and the multi-operand adder increases.

For the numbers with 16 bit significands, the quadratic approximation requires the lowest area, while 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 due to its huge 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 Figure 11. In comparison, a 24 by 24-bit multiply in the same technology has a delay of 34 ns, requires an area of 16 mm^2 and has an delay-area product of 544 $ns \cdot mm^2$.

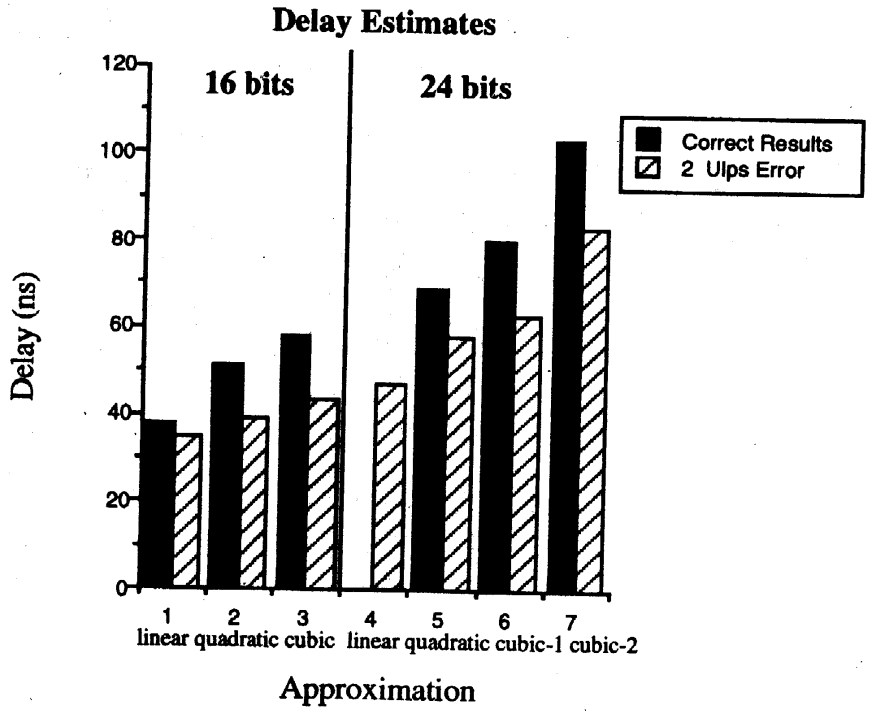

Figure 9: Delay estimates

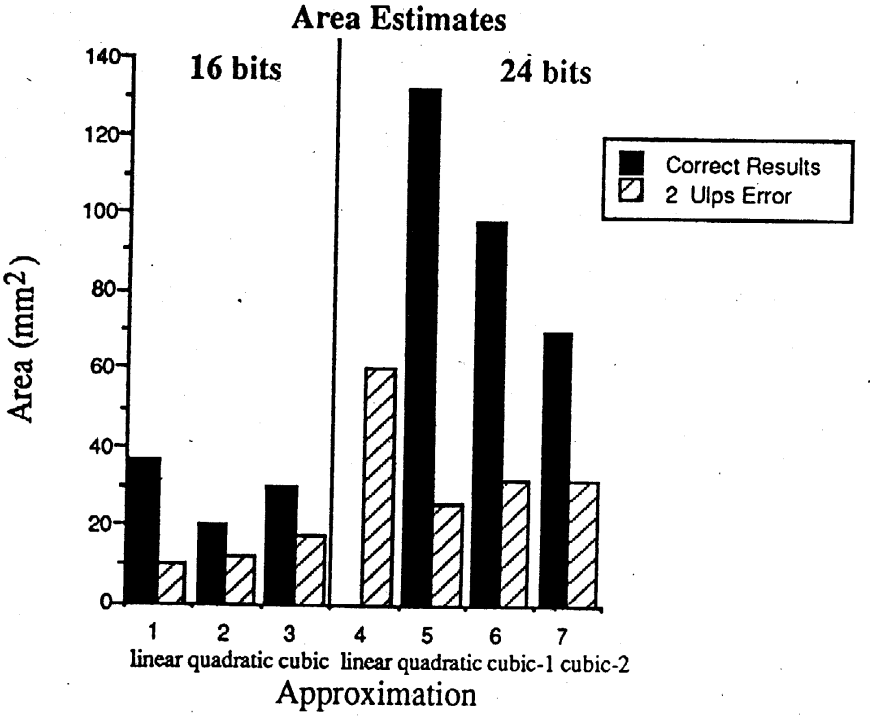

Figure 10: Area estimates

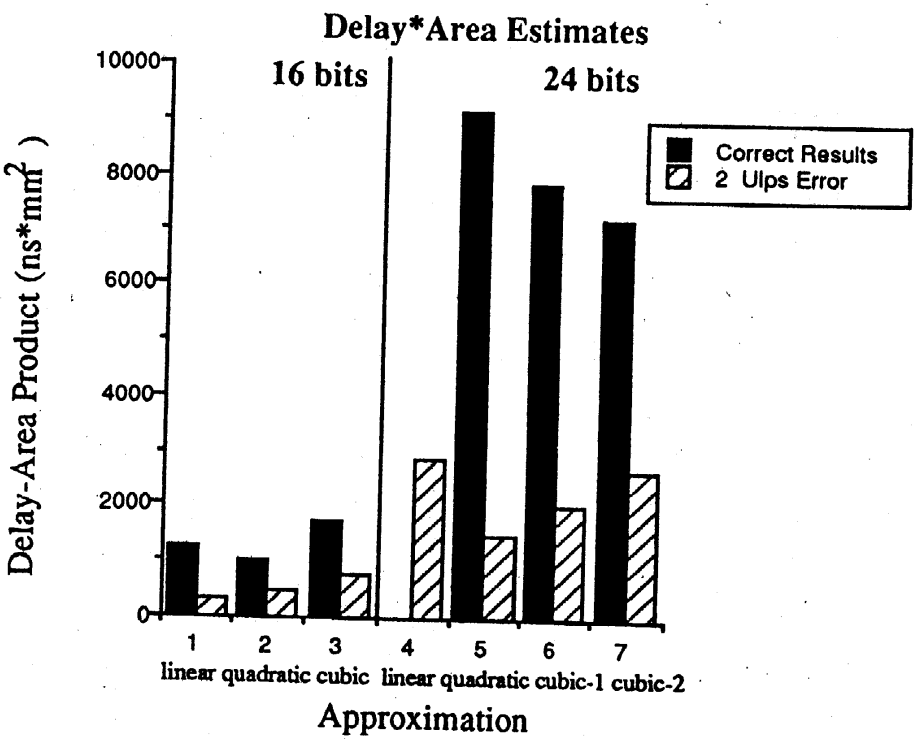

Figure 11: Delay $*$ area estimates

Correct rounding produces approximations with a maximum error of one ulp for the directed rounding modes and half an ulp for round to nearest. If the maximum error is allowed to be two ulps, the delay is reduced by 5 to 30 percent and the area is reduced by 33 to 37 percent. These estimates correspond to the shaded bars in Figure 9, 10, and 11. For approximations which have a maximum error of two ulps, the best designs are obtained with linear and quadratic approximations for numbers with 16 and 24 bit significands, respectively.

6 Sufficient accuracy for exactly rounded results

This section presents a method for determining the accuracy that is sufficient to guarantee exactly rounded results. The results presented here are used to show the advantages that are realized by adjusting the coefficients, instead of using a traditional Chebyshev series approximation.

For most elementary functions there is no known theoretical method to determine in advance the accuracy of the pre-rounded result which is required to guarantee that the final answer will be exactly rounded. This problem is known as the Table Maker's Dilemma. For example, suppose the value of a function $f(x)$ when computed to 4 bits is $.0010_2$. It cannot be determined whether the 3 bit exactly rounded result should be $.001_2$ or $.010_2$. If $f(x)$ computed to 5 bits is $.00100_2$, 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 determine whether $f(x)$ is $.00100\ldots001XXX$ or $.000111\ldots111XXX$. Due to this problem [1] and [3] claim that it is not practical to require that the results of elementary functions are exactly rounded. As described below, however, the accuracy which is sufficient to guarantee exact rounding can be determined analytically for a given floating point format.

To ensure exact rounding for the elementary functions, it is sufficient to guarantee the following: (1) the pre-rounded result is less than 0.5 ulps from the exactly rounded value of the function and (2) the pre-rounded result is rounded using round to nearest. If $f(x)$ is the exact value of the function and $p(x)$ is the value of the pre-rounded result, the following statements holds:

$$\begin{aligned}&\text{IF } |p(x) - f(x)| < 0.5 \cdot \text{ulp} - |[f(x)]_p - f(x)| \text{ THEN}\\ &\quad |p(x) - [f(x)]_p| < 0.5 \cdot \text{ulp}\\ &\text{AND}\\ &\quad [p(x)]_p = [f(x)]_p\end{aligned} \tag{14}$$

where $[x]_p$ is the value of x rounded to p bits using round to nearest. Thus, if the distance between the pre-rounded result and the exact value of the function is less than $Y(x)$, where

$$Y(x) = 0.5 \cdot \text{ulp} - |[f(x)]_p - f(x)| \tag{15}$$

exact rounding is guaranteed. This is equivalent to requiring that $f(x)$ is closer to $p(x)$ than it is to the midpoint of the two nearest floating point numbers. Figure 12 illustrates this requirement.

Based on the previous discussion, the accuracy in the pre-rounded result which will guarantee exact rounding can be determined by finding the minimum value of $Y(x)$ for all numbers on the input interval. The minimum

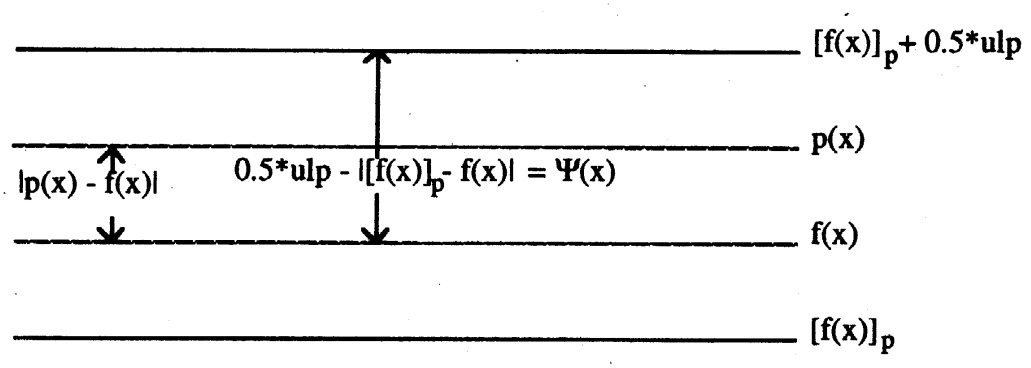

Figure 12: Exactly rounded answers

value of $Y(x)$ for floating point numbers with 16 and 24 bit significands, along with the required number of accurate bits in the normalized, pre-rounded result is shown in Table 3. The number of accurate bits required is $\lceil -\log_2(Y(x)_{\min}) \rceil$.

	16 bit significand		24 bit significand	
Function	$Y(x)_{\min}$	Accurate bits	$Y(x)_{\min}$	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
$2^x (x \geq 0)$	$2.37 \cdot 10^{-9}$	29	$6.21 \cdot 10^{-15}$	48

Table 3: Required accuracy for 16 and 24 bit numbers

It is important to note that the condition given in (14) is sufficient, but not necessary to ensure exactly rounded results. If the distance between the pre-rounded result and the rounded value of the function is less than the distance between the exact value of the function and the exactly rounded value of the function, then exactly rounded results will be obtained even if (14) does not hold.

Table 4 shows the maximum error and the minimum number of accurate bits in the pre-rounded result for each of the designs. Comparing these values to those given in Table 3, shows that our algorithm requires much less accuracy in the pre-rounded result. This is because it uses knowledge about the exactly rounded result to adjust the coefficients. For example, Table 3 shows that 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.

	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

Table 4: Maximum error and minimum number of accurate bits

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 have not been adjusted. For numbers with 16 bit significands, the design for a quadratic approximation has a delay of 65 ns and an area of 39 mm^2. Compared the design in which the coefficients are adjusted, this design has an increase in delay of 27 percent and an increase in area of 95 percent. 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 percent and an increase in area of 136 percent, compared to the cubic-2 approximation with adjusted coefficients.

7 Conclusion

A parallel algorithm has been presented which produces correctly rounded results for the functions of reciprocal, square root, 2^x, and $\log_2(x)$. It is useful for interval arithmetic, because it allows directed rounding of the elementary functions. 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, correctly rounded results are obtained with much less hardware than designs which use traditional Chebyshev series approximations. Allowing the results to have a maximum error of 2 ulps decreases the area and delay,

and is suitable for applications in which stringent control of the error is not required.

Acknowledgements

The authors would like to thank Dr. David Matula who provided useful comments on the content of the paper. They are also grateful to Dr. V. K. Jain who introduced them to a method for elementary function generator which led to some of the ideas presented in this paper.

References

[1] Cody, W. and Waite, W. *Software manual for the elementary functions.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[2] Volder, J. E. *The CORDIC trigonometric computing technique.* IRE Transactions on Electronic Computers **EC–8** (1959), pp. 330–334.

[3] Flynn, M. J. *On division by functional iteration.* IEEE Transactions on Computers **C–19**, pp. 702–706.

[4] Koren, I. and Zinaty, O. *Evaluation of elementary functions in a numerical co-processor based on rational approximations.* IEEE Transactions on Computers **39** (1990), pp. 1030–1037.

[5] Farmwald, P. M. *High bandwidth evaluation of elementary functions.* In: "Proceedings of the 5th Symposium on Computer Arithmetic", 1981, pp. 139–142.

[6] Kahan, W. M. *Interval arithmetic options in the proposed IEEE floating point arithmetic standard.* Interval Mathematics (1980), pp. 99–129.

[7] Moore, R. E. *Interval analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[8] Kulisch, U and Miranker, W. L. *Computer arithmetic in theory and practice.* Academic Press, New York, 1981.

[9] *IEEE Standard 754 for binary floating point arithmetic.* ANSI/IEEE Standard No 754, American National Standards Institute, Washington DC, 1988.

[10] Goldberg, D. *What every computer scientist should know about floating-point arithmetic.* ACM Computing Surveys **23** (1991), pp. 5–48.

[11] Hough, D. *Elementary functions based on IEEE arithmetic.* Mini/Micro West Conference Record, Electronic Conventions, Inc., Los Angeles, 1983, pp. 1–4.

[12] Gal, S. and Bachelus, B. *An accurate elementary mathematical library for the IEEE floating point standard.* ACM Transactions on Mathematical Software **17** (1991), pp. 26–45.

[13] Black, C. M., Burton, R. P., and Miller, T. H. *The need for an industry standard of accuracy for elementary function programs.* ACM Transactions on Mathematical Software **1** (1984), pp. 361–366.

[14] Ziv, A. *Fast evaluation of elementary mathematical functions with correctly rounded last bit.* ACM Transactions on Mathematical Software **17** (1991), pp. 410–423.

[15] Mathews, J. N. *Numerical methods for computer science, engineering and mathematics.* Prentice-Hall, Englewood Cliffs, N. J., 1987.

[16] Baugh, C. R. and Wooley, B. A. *A two's complement parallel array multiplication algorithm.* IEEE Transactions on Computers **C–22** (1973), pp. 1045–1047.

[17] Wallace, C. S. *A suggestion for a fast multiplier.* IEEE Transactions on Electronic Computers **EC–13** (1964), pp. 14–17.

[18] Swartzlander, E. E., Jr. *Merged arithmetic.* IEEE Transactions on Computers **C–29** (1980), pp. 946–950.

[19] *LSI logic 1.0 micron cell-based products databook.* LSI Logic Corporation, Milpitas, California, 1991.

[20] Jain, V. K. *Arithmetic analysis of a new reciprocal cell.* In: "1992 International Conference on Computer Design: VLSI in Computers and Processors", 1992, pp. 106–109.

Department of Electrical
and Computer Engineering
University of Texas at Austin
Austin, TX 78712