Exact Rounding of Certain Elementary Functions

Michael Schulte and Earl Swartzlander
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Abstract

An algorithm is described which produces exactly rounded results for the functions of reciprocal, square root, 2^x, and $\log_2(x)$. Hardware designs based on this algorithm are presented for floating point numbers with 16 and 24 bit significands. These designs use a polynomial approximation in which coefficients are originally selected based on the Chebyshev series approximation and are then adjusted to ensure exactly rounded results for all inputs. To reduce the number of terms in the approximation, the input interval is divided into subintervals of equal size and different coefficients are used for each subinterval. For floating point numbers with 16 bit significands, the exactly rounded value of the function can be computed in 51 ns on a 20 mm^2 chip. For floating point numbers with 24 bit significands, the functions can be computed in 80 ns on a 98 mm^2 chip.

1 Introduction

In general, the exact value of an elementary function (e.g., reciprocal, exponential, square root, logarithm, etc.) cannot be represented as a fixed-sized floating point number. Therefore, the computed result of the function is rounded to a finite number of bits. To minimize the difference between the infinitely precise value of the function and the computed result, the computed result should be exactly rounded. Exact rounding as defined in [1] requires the rounded result to be identical to the result obtained if the infinitely precise value of the function is rounded to the nearest floating point number.

In the IEEE 754 Standard [2], round to nearest even is the default rounding mode for addition, subtraction, multiplication, division, square root, remainder and conversion between integer and floating point formats. Round to nearest even is a method of exact rounding which requires that if the infinitely precise result falls exactly halfway between the two nearest floating point numbers, the result should be rounded such that its least significant bit is zero. The IEEE 754 Standard, however, does not require exact for the elementary functions. This is largely due to a problem known as the Table Maker's Dilemma [1] [3]. This dilemma and a solution to it are discussed in Section 4.

Evaluating elementary functions with a method that produces exactly rounded results has several advantages. In addition to minimizing the error between the computed result and the exact value of the function, exact rounding also preserves several desirable properties of the functions such as symmetry and monotonicity [4]. Another advantage is that machines which have the same floating point format and ensure exact rounding will always produce the same results. This improves software portability and allows the correctness of floating point algorithms to be verified for a standardized system. Other advantages of having an industry standard for elementary functions are discussed in [5].

Because of the advantages offered, much research has been performed to develop software routines which produce exactly rounded results for the elementary functions. In [6], software routines are described which always produce exactly rounded results for single precision exponential, and single and double precision square root and complex absolute value in the IBM System 370 floating point format. However, these routines require between 50 and 70 machine cycles to execute on a general purpose computer. For the other elementary functions, exact rounding is achieved for between 95.0 and 99.9 percent of the inputs. Routines which are expected to produce exactly rounded results for elementary functions in the IEEE double precision format are described in [7]. For the first iteration, the result is computed using double precision arithmetic. If the result of this routine is not guaranteed to be exactly rounded, it is recomputed using a higher precision routine which may be orders of magnitude slower than the original routine. With this approach hundreds of cycles may be required to compute results which are not exactly rounded after the first iteration.

To speed up computation of the elementary functions, several techniques have been developed for approximating the elementary functions with special purpose hardware. These techniques include the CORDIC algorithm [8], Newton-Raphson iteration [9], rational approximations [10], and polynomial approximations [11]. However, no known implementation of these techniques produces results which are guaranteed to be exactly rounded. Instead, most hardware approaches only guarantee that the computed results will differ from the exact answer by less than a specified amount, which is often less than one 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. The algorithm described in this paper also offers a speed advantage over existing methods, because most of the computation is performed in parallel and division is not required.

This paper presents hardware designs which produce exactly rounded results for the functions of reciprocal, square root, 2^x and $\log_2(x)$ for floating point numbers with 16 and 24 bit significands. Section 2 contains a discussion of polynomial approximations with an emphasis on the Chebyshev series approximation. Section 3 examines range reduction, which limits the range of the input operand. In section 4, the difficulty of obtaining exactly rounded results is discussed and a method is presented for determining the accuracy needed to guarantee exactly rounded results. Section 5 presents an algorithm by which the coefficients of the polynomials are adjusted to guarantee exactly rounded results. In Section 6, hardware designs which produce exactly rounded results for all inputs are given.

The algorithm presented in this paper has two limitations. First, it cannot be used for all functions. For elementary functions such as sine and cosine, range reduction techniques may introduce errors which cause the final results to not be exactly rounded. Second, the algorithm may not be feasible for floating point numbers with large significands (e.g., double precision), because the algorithm for adjusting the coefficients requires an exhaustive simulation of all values on the input interval.

2 Polynomial Approximations

The algorithm discussed in this paper uses polynomial approximations to compute the value of a function f(x). Polynomial approximations have the form

$$q_{n-1}(x) = a_0 + a_1\cdot x + \dots + a_{n-1}\cdot x^{n-1} = \sum_{i=0}^{n-1} a_i\cdot x^i \quad (1)$$

where $q_{n-1}(x)$ is a polynomial of degree n - 1, and a_i is the coefficient of the ith term.

Typically, the function is approximated on a input interval $[x_{min}, x_{max})$ and range reduction is needed for values outside this interval. To reduce the number of terms in the approximation, the input interval is divided into a set of equally sized subintervals. This is done in hardware 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 interval [0,1), then

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

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

$$p_m(x) = \sum_{i=0}^{n-1} a_i(x_m)\cdot x_l^i \quad (3)$$

where $p_m(x)$ is the approximating polynomial of degree n-1 for subinterval m. The a_is 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.

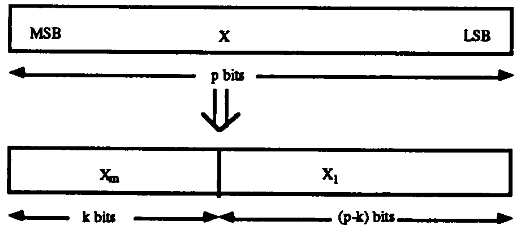


Figure 1: Dividing the Input Value.

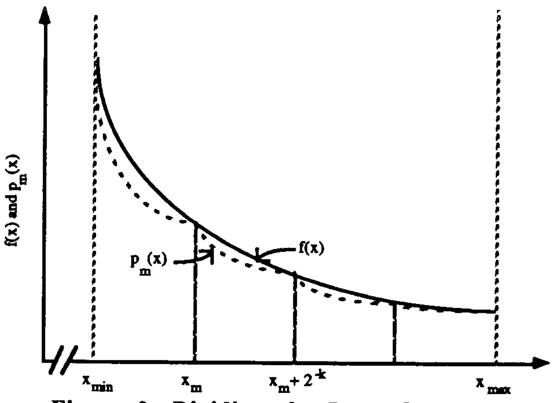
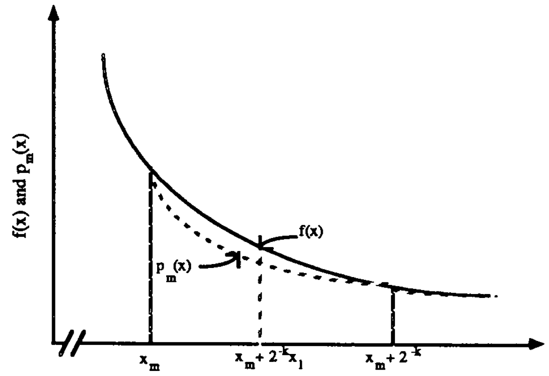


Figure 2: Dividing the Input Interval.

For normalized IEEE floating point numbers [2], the input interval is often [1,2). In this case, x_m consists of 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

$$x = 1 + x_m + 2^{-k}\cdot x_l \quad (4)$$



.Figure 3: A Single Subinterval

An approximation to the minimax polynomial, the Chebyshev series approximation, is originally used to select the coefficients for each of the subintervals. On the subinterval $[x_m, x_m + 2^{-k})$, the Chebyshev series approximation $p_m(x)$ is computed as follows [12]:

1. The Chebyshev nodes t_i on $[-1,1)$ are computed using the formula
$$t_i = \cos\left(\frac{(2 \cdot i+1) \cdot \pi}{2 \cdot n}\right) \qquad (0 \le i < n) \qquad (5)$$
2. The Chebyshev nodes are transformed from $[-1,1)$ to $[x_m, x_m + 2^{-k})$ through the equation
$$x_i = x_m + (t_i + 1) \cdot 2^{-k-1} \qquad (0 \le i < n) \qquad (6)$$
3. The Lagrange polynomial which interpolates the Chebyshev nodes on $[x_m, x_m + 2^{-k})$ is computed as
$$p_m(x) = \sum_{i=0}^{n-1} y_i \cdot L_i(x) \qquad (7)$$
where
$$y_i = f(x_i) \qquad (8)$$
$$L_i(x) = \frac{\prod_{k=0;k\ne i}^{n-1} (x - x_k)}{\prod_{k=0;k\ne i}^{n-1} (x_i - x_k)} \qquad (9)$$

The coefficients for each power of x can be factored to express $p_m(x)$ in the form given in equation (3). The maximum error between f(x) and $p_m(x)$ is
$$E_n(x) \le \frac{2^{-n(k+2)+1} \cdot f^n(\xi)}{n!} \qquad (10)$$
where $f^n(\xi)$ is the nth derivative of $f(\xi)$ and
$$x_m \le \xi < x_m + 2^{-k}$$
As can be seen from equation 10, increasing the number of bits in x_m by one decreases the maximum error by a factor of 2^{-n}. Unfortunately, this doubles the number of coefficients. In comparison, increasing the number of terms by one decreases the maximum error by a factor of approximately $2^{-(k+2)}$. However, this increases the number of multiplies and adds which are needed.

3 Range Reduction

Before computing the value of a function, the input argument is first transformed so that it falls within a given input interval. The function is then computed for the transformed input, followed by a second transformation which compensates for the original transformation and normalizes the result. The input and output transformation are commonly referred to as range reduction. For floating point numbers, the approximation is usually computed based on the value of the significand and range reduction is performed to compensate for the value of the exponent.

Although range reduction is discussed in other papers, e.g. [13], it is discussed here to illustrate that for the functions of reciprocal, square root, 2^x and $\log_2(x)$, the range reduction transformations 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 Figure 4. In these formulas, it is assumed that the numbers are in the IEEE floating point format for normalized numbers, which take the form
$$x = (-1)^{S_x} \cdot M_x \cdot 2^{E_x} \qquad (1 \le M_x < 2)$$
For convenience, 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 inputs
M_y' and E_y'	The pre-transformed outputs
M_y and E_y	The transformed outputs
S_x and S_y	The sign bit of the input and output.

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 exactly rounded results on the input interval guarantees exactly rounded results for all inputs. For the output transformation of $\log_2(x)$, if E_x is non-zero, 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. If E_x is equal to zero, leading zeros may appear in $\log_2(M_x')$ which leads to a loss of precision. Since
$$1 \le \frac{\log_2(M_x')}{M_x' - 1} < 2$$
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 adjusted.

reciprocal

$$\frac{1}{M_x \cdot 2^{E_x}} = \frac{1}{M_x} \cdot 2^{-E_x}$$

(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

if (E_x even) $\sqrt{M_x \cdot 2^{E_x}} = \sqrt{M_x} \cdot 2^{E_x/2}$

if (E_x odd) $\sqrt{M_x \cdot 2^{E_x}} = \sqrt{2 \cdot M_x} \cdot 2^{(E_x - 1)/2}$

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

$\log_2(x)$

if ($E_x \neq 0$) $\log_2(M_x \cdot 2^{E_x}) = \log_2(M_x) + E_x$

if ($E_x = 0$) $\log_2(M_x \cdot 2^{E_x}) = \frac{\log_2(M_x)}{1 - M_x} \cdot (1 - M_x)$

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

2^x

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

$Z_x = M_x \cdot 2^{E_x}$ $E_x' = \lfloor Z_x \rfloor$ $M_x' = Z_x - E_x'$

(1)	$S_y = 0$	
(2)	$M_x' = Z_x - E_x'$	$E_x' = \lfloor Z_x \rfloor$
(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 4: Range Reduction Equations

4 Determining the Necessary Accuracy

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 is $.001_2$ 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 determined whether f(x) is $.00100...01XX..._2$ or $.00011...10XX..._2$, where X represents either 0 or 1. Due to this problem [1] and [3] claim that it is not practical to require that the results of elementary functions are exactly rounded. This section describes a method by which the necessary accuracy to guarantee exactly rounded results can be determined analytically for a specified 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:

$$\text{IF } \lvert p(x) - f(x) \rvert < 0.5 \cdot \text{ulp} - \lvert [f(x)]_p - f(x) \rvert \text{ THEN}$$
$$\lvert p(x) - [f(x)]_p \rvert < 0.5 \cdot \text{ulp} \quad \text{AND}$$
$$[p(x)]_p = [f(x)]_p \qquad (12)$$

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

$$Y(x) = 0.5 \cdot \text{ulp} - \lvert [f(x)]_p - f(x) \rvert \qquad (13)$$

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 5 illustrates this requirement.

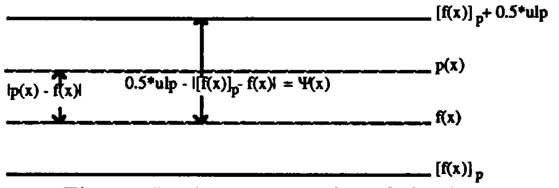


Figure 5: Exact Rounding Criterion.

Based on the previous discussion, the accuracy in the pre-rounded result which will guarantee exact rounding is determined by finding the minimum value of Y(x) for all values on the input inverval. The minimum 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 1. The number of accurate bits required is

$$\text{Bits} = \lceil -\log_2(Y(x)_{min}) \rceil$$

Table 1: Necessary Accuracy

	16 bit significand		24 bit significand	
Function	$Y(x)_{min}$	Bits	$Y(x)_{min}$	Bits
reciprocal	$2.33 \cdot 10^{-10}$	32	$3.56 \cdot 10^{-15}$	48
sqr. 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

5 Adjusting the Coefficients

If the coefficients of the Chebyshev series approximation are used to compute the elementary functions, either the size of the table look-up or the number of terms in the approximation must be very large. Figure 6 contains an algorithm by which the values of the coefficients can be adjusted to achieve exactly rounded results.

```
(1)  Set the best coefficients to the coefficients of the
     Chebyshev approximation;
(2)  for i = 1 to number of coefficients do
(3)    for j = 1 to number of subintervals do
(4)      compute the number of incorrect results for this
         subinterval using the best coefficients;
(5)      for k = 1 to number of iterations do
(6)        for sign = -1 to 1 step 2 do
(7)          modify the value of the ith coefficient on
             the jth subinterval by
             $a[i][j] = a[i][j] + sign \cdot k \cdot 2^{-p_i}$;
(8)          compute the number of incorrect results
             using the best coefficients and the modified
             ith coefficient ;
(9)          if (the number of incorrect results is reduced
             and none of the approximations have more
             than one ulp of absolute error) then
(10)           the modified coefficient becomes the best
               coefficient for this subinterval;
(11)           remember the number of incorrect results;
(12)           if (the number of incorrect results is zero)
               then
(13)             exit this subinterval (exit j);
(14)       end k
(15)     end j
(16)     if (the total number of incorrect results is zero)
         then
(17)       exit modifying coefficients (exit i);
(18)   end i
```

Figure 6: Adjusting the Coefficients.

To determine the number of iterations for adjusting the coefficients, it is assumed that the difference between the exact answer and the pre-rounded result is less than 2^{-q} and the rounded result has an ulp of 2^{-p}. The ulp of the ith coefficient is 2^{-p_i}, where $p \leq q \leq p_i$.

1. For a given $a_1, a_2, \ldots, a_{n-1}$, the maximum number of iterations needed to select the optimal value of a_0 is $2^{p_0 - q}$.
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 $2^{p_i - p}$.

The algorithm for adjusting the coefficients executes on a 50 MFLOPS processor in under one hour for numbers with 24 bit significands.

6 Hardware Designs

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

Because the terms in the approximation are independent, they can be generated in parallel and then summed using a multi-operand adder. Figure 7 shows a block diagram for a polynomial approximation of degree n.

To reduce the complexity of the elementary function generator, special purpose multipliers and multi-operand adders are designed which take advantage of the characteristics of the polynomial approximations. Since the x_l^is are guaranteed to be positive, each term is computed using a n by m 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 8. To avoid sign extension, the sign bit of each of the partial products is complemented and a one is added to the nth column. This is similar to the method of sign extension presented in [14]. The multi-operand adder computes the sum of 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 9, where W, X, Y and Z are the four terms being 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. 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. Since the sign of the result is computed based on the sign of the input and the function being evaluated, the sign of a_0 is not stored and carry into this position is ignored.

One of the disadvantages of the design shown in Figure 7 is that a carry-propagate adder is needed for each multiplication and the final summation. Instead of performing the multiplications with separate multipliers and then summing the terms with a multi-operand adder, the multiplications and summation can be merged using

Dadda [15] or Wallace [16] trees. With this approach, only a single carry propagate adder is needed and the delay and area are reduced. Implementations for merged arithmetic and a discussion of its advantages are given in [17].

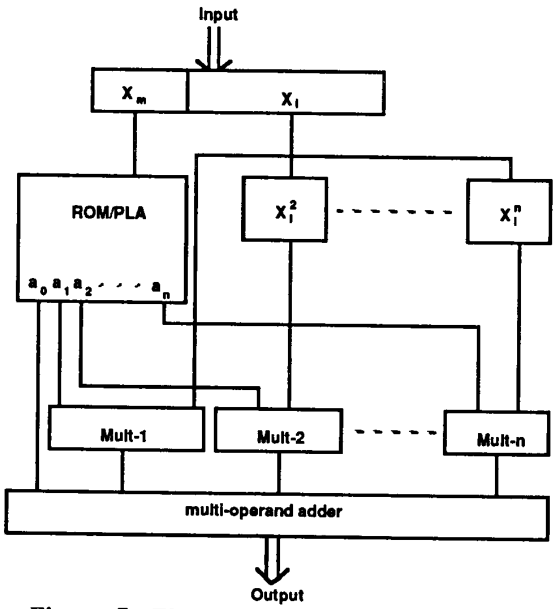
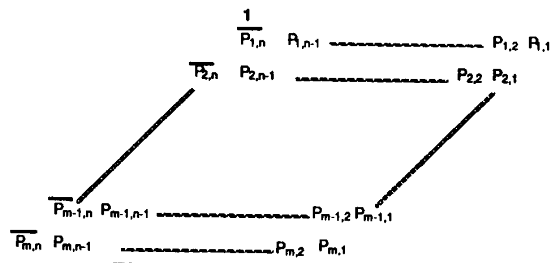
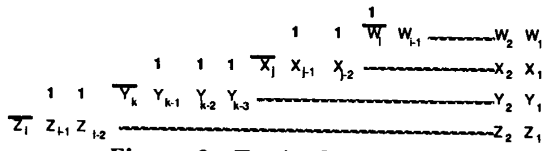


Figure 7: Elementary Function Generator.



1
$\overline{P_{1,n}}$ $P_{1,n-1}$ $P_{1,2}$ $P_{1,1}$
$\overline{P_{2,n}}$ $P_{2,n-1}$ $P_{2,2}$ $P_{2,1}$
$\overline{P_{m-1,n}}$ $P_{m-1,n-1}$ $P_{m-1,2}$ $P_{m-1,1}$
$\overline{P_{m,n}}$ $P_{m,n-1}$ $P_{m,2}$ $P_{m,1}$

Figure 8: n by m Multiplier.



1
1 1 $\overline{W_l}$ W_{l-1} W_2 W_1
1 1 1 $\overline{X_j}$ X_{j-1} X_{j-2} X_2 X_1
1 1 $\overline{Y_k}$ Y_{k-1} Y_{k-2} Y_{k-3} Y_2 Y_1
$\overline{Z_l}$ Z_{l-1} Z_{l-2} Z_2 Z_1

Figure 9: Two's Complement Multi-operand Adder.

The hardware requirements to obtain exactly rounded results for the four functions were determined through computer simulation. The simulation first determines the coefficients of the Chebyshev series approximation for each subinterval. It then simulates the computation of functions for all values on the input interval and adjusts the coefficients using the algorithm presented in Section 5 to guarantee that all results are exactly rounded. The exactly rounded value of the function is determined by rounding the IEEE double precision value of the function using round to nearest even. 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 2. For the multipliers, the number of bits in the multiplicand and multiplier are given. 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 input is given. Table 3 shows the lengths of the coefficients and the memory requirements for each design.

Area and delay estimates are shown in Tables 4 and 5. The area in mm^2 and delay in ns is given for each component. These estimates are based on data from a 1.0 micron CMOS standard cell library [19]. The estimates for the multipliers, squaring circuits and multi-operand adders assume a Wallace reduction followed by carry look-ahead addition, with four bit blocks. The areas of each component are estimated by calculating the total size of the macrocells (e.g., AND gates, full adders, half adders, etc.) which make up the component. An area overhead of 50% is assumed for the internal wiring of each component. The estimated area and delay for the entire chip are also given. The overhead due to global wiring and power and ground signals is estimated as 50% of the total area, and an additional 1.0 mm is added to the height and width of the chip for I/O pads. The components which contribute to the critical path are marked by a '*'. To determine the delay of the entire chip, an overhead of 25% is assumed for delay due to global routing. The values given do not take into account the time and area needed to perform range reduction.

The powers of x_l are generated with either special purpose squaring circuits or by a table look-up operation. Implementations for squaring circuits are given in [18]. For numbers with 24 bit significands, x_l^3 are obtained either through a table look-up or by multiplying x_l by x_l^2. The estimates for these two designs are label cubic(24)-1 and cubic(24)-2, respectively.

The values shown in Tables 4 and 5 illustrate the tradeoffs that can be made in terms of delay and area by varying the degree of the polynomials. If the product of delay and area is the criterion used to select the optimum design, then the quad(16) approximation is the best design for numbers with 16 bit significands, while the cubic(24)-2 approximation is the best design for numbers with 24 bit significands. The linear(24) approximation cannot be practically implemented, because of its huge memory requirements.

Table 2: Hardware Requirements for Exactly Rounded Results.

Approximation	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 3: Memory Requirements for Exactly Rounded Results.

	Coefficient Lengths				Table Size		
Approximation	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 K

Table 4: Estimated Area (mm^2).

Approximation	Mult1	Mult2	Mult3	Square	Cube	Adder	ROM	Total	Chip
linear(16)	1.0					0.6	13.8	15.4	34
quad(16)	1.9	1.6		0.4		0.8	3.1	7.8	20
cubic(16)	2.8	3.5	1.9	0.8	1.4	1.0	2.1	13.5	30
linear(24)	1.8					0.9	2191	2194	3407
quad(24)	4.6	3.8		0.9		1.3	62.7	73.3	132
cubic(24)-1	6.6	9.5	3.3	1.9	19.3	1.3	10.8	52.7	98
cubic(24)-2	6.6	9.5	3.3	1.9	2.8	1.3	10.8	36.2	70

Table 5: Estimated Delay (ns).

Approximation	Mult1	Mult2	Mult3	Square	Cube	Adder	ROM	Total	Chip
linear(16)	17*						13*	30	38
quad(16)	19*	16		13		14*	8*	41	51
cubic(16)	22	23*	16	8*	8	15*	6	46	58
linear(24)	19*					15*	150*	184	230
quad(24)	24*	23		14		17*	14*	55	69
cubic(24)-1	24	27*	23	19*	14	18*	11	64	80
cubic(24)-2	24	27	23*	19*	22*	18*	11	82	103

Table 6: Maximum Error and Minimum Bits of Accuracy.

	reciprocal		square root		$\log_2(x)$		2^x	
Approximation	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	$8.52 \cdot 10^{-8}$	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	$3.45 \cdot 10^{-12}$	39	$3.83 \cdot 10^{-12}$	38	$2.40 \cdot 10^{-12}$	39
cubic(24)	$1.55E \cdot 10^{-12}$	40	$9.45 \cdot 10^{-13}$	40	$9.62 \cdot 10^{-13}$	40	$9.15 \cdot 10^{-13}$	40

Table 6 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 1 demonstrates that our algorithm produces exactly rounded results using much less accuracy than is required by the analysis developed in Section 4. For example, the analysis of Section 4 shows that for reciprocal, 48 bits of accuracy will 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.

7 Conclusion

An algorithm has been presented which produces exactly rounded results for the functions of reciprocal, square root, 2^x and $\log_2(x)$. Because the coefficients are adjusted based on the error in the original approximation, exactly rounded results can be obtained using much less hardware than would be required if a more conventional method of polynomial approximations had been employed. Area and delay estimates illustrate the feasibility of obtaining exactly rounded results with special purpose hardware.

Acknowledgments

The authors wish to thank Srilatha Manne who developed the area and delay estimates. They are also grateful to Prof. V. K. Jain who introduced them to a method for elementary function generation which led to some of the ideas presented in this paper.

References

[1] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, pp. 5-48, 1991.

[2] "IEEE Standard 754 for Binary Floating Point Arithmetic," ANSI/IEEE Standard No. 754, American National Standards Institute, Washington DC, 1988.

[3] David Hough, "Elementary Functions Based on IEEE Arithmetic," *Mini/Micro West Conference Record*, Electronic Conventions, Inc., Los Angeles, pp. 1-4, 1983.

[4] Shmuel Gal and Boris Bachelus, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *ACM Transactions on Mathematical Software*, Vol. 17, pp. 26-45, 1991.

[5] C.M. Black, R.P. Burton, and T.H. Miller, "The Need for an Industry Standard of Accuracy for Elementary Function Programs," *ACM Transactions on Mathematical Software*, Vol. 1, pp. 361-366, 1984.

[6] Ramesh C. Agarwal, James W. Cooley, Fred G. Gustavson, James B. Shearer, Gordon Slishman, and Bryant Tuckerman, "New Scalar and Vector Elementary Functions for the IBM System/370," *IBM Journal of Research and Development*, Vol. 30, 126-144, 1986.

[7] Abraham Ziv, "Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit," *ACM Transactions on Mathematical Software*, Vol.. 17, pp. 410-423, 1991.

[8] J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, Vol. EC-8, pp. 330-334, 1959.

[9] M.J. Flynn, "On Division by Functional Iteration," *IEEE Transactions on Computers*, C-19, pp. 702-706.

[10] I. Koren and O. Zinaty, "Evaluation of Elementary Functions in a Numerical Co-Processor Based on Rational Approximations, " *IEEE Transactions on Computers*, Vol. 39, pp. 1030-1037, 1990.

[11] P.M. Farmwald, "High Bandwidth Evaluation of Elementary Functions," *Proceedings of the 5th Symposium on Computer Arithmetic*, pp. 139-142, 1981.

[12] J.H. Mathews, *Numerical Methods for Computer Science, Engineering and Mathematics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[13] J.S. Walther, "A Unified Algorithm for Elementary Functions," *Spring Joint Computer Conference*, pp. 379-385, 1971.

[14] C.R. Baugh and B.A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers* Vol. C-22, pp. 1045-1047, 1973.

[15] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, 349-356, May 1965.

[16] C.S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, Vol. EC-13, pp.14-17, 1964.

[17] E. E. Swartzlander, Jr., "Merged Arithmetic," *IEEE Transactions on Computers*," Vol. C-29, pp. 946-950, 1980.

[18] T. Jayarshee and D. Basu, "On Binary Multiplication Using the Quarter Square Algorithm," *Spring Joint Computer Conference*, pp. 957- 960, 1974.

[19] *LSI Logic 1.0 Micron Cell-Based Products Databook*, LSI Logic Corporation, Milpitas, California, 1991.