# High-Performance Architectures for Elementary Function Generation

Jun Cao*, Belle W. Y. Wei, Jie Cheng
*Department of Electrical Engineering*
*San Jose State University*
*One Washington Square*
*San Jose, CA 95192-0084*
*(408)924-3881*
*FAX: (408)924-3925*
*bwei@email.sjsu.edu*

**Abstract:**

*High-speed elementary function generation is crucial to the performance of many DSP applications. This paper presents three new architectures for generating elementary functions with IEEE single precision using second-order interpolation. These designs have been developed through a combination of architectural innovations and algorithm developments. They represent a range of trade-off between the use of memory modules and computational circuits. Our most memory intensive architecture uses one third less memory than alternative schemes while incurring no time penalty and minimal additional circuitry.*

## 1. Introduction

Elementary functions such as trigonometric functions, square-root, reciprocal, etc. are essential to many DSP applications. These functions are often implemented in software routines [4][5], which are too slow for numerically intensive or real-time applications. The performance of these applications depends on the design of a hardware function generator. One common hardware scheme employs ROM lookup tables to interpolate functional values using linear or second-order approximation functions. Second-order interpolation produces significant memory savings as shown by Jain [8], whose design reduces the memory size from 129 kb to 22 kb for the square root function at the expense of more complex circuitry. It presents a favorable cost-performance trade-off in comparison with first-order interpolation [8].

This paper presents three different hardware algorithms for second-order interpolation generation elementary functions up to IEEE single precision. It is a continuation of the work presented in [1]. The three architectures represent a range of cost-performance trade-offs in terms of hardware complexity, memory requirements and circuit speed. The first scheme, the Hybrid method, uses a second-degree interpolation polynomial passing through evenly-spaced nodes. Different from existing designs [8][9][12][13], the Hybrid scheme stores a combination of the polynomial's coefficients and function values for fast function interpolation. The Dynamic Range Reduction (DRR) scheme, the second method, stores reduced values of target functions and achieves approximately 7% memory savings compared to the Hybrid scheme. The third scheme uses a composite polynomial generated by combining two neighboring second-degree polynomials. The composite polynomial is proven to have the minimum approximation error and requires at least 30% less memory than the Hybrid method.

Our design implements function generation for *cosine*, *sine*, *reciprocal*, *square root*, and *power of 2* functions. Three steps are involved in finding $Y=f(X)$ using table lookup, where $Y$ and $X$ are in IEEE single-precision floating point format: range reduction, interpolation and reconstruction. It is the interpolation step that is the focus of this paper. Range reduction and reconstruction steps are discussed in [2][6][12][13][14][15].

In this paper, Section II gives an overview of range reduction and reconstruction steps for the functions of interest. Section III presents background information on second-degree polynomial approximation. Sections IV, V and VI present the three hardware algorithms and architectures. Their performance and hardware requirements are compared with those of existing schemes in the last section, Section VII.

## 2. Range Reduction and Reconstruction

In calculating $Y=f(X)$, $X$ is first mapped to $x$ such that $x$ is bounded by $[A, B]$. Then $f(x)$ is interpolated from $f_i=f(x_i)$ with $x_i \in [A, B]$. Lastly, a reconstruction step is used to compensate the range reduction done in the first

*. This work is supported in part by NSF grant MIP-9321143

step in order to compute *Y*.

### A. Number Format

The number format used in our design is the IEEE single-precision floating point format where number *X* is represented by 32 bits with the leading bit as the sign bit. The remaining 31 bits consist of a 23-bit mantissa (*M*) and an 8-bit biased exponent *E*. The value of *X* is given by EQ. (1) where the 1 to the left of the binary point is implied. As a result, the effective precision of the representation is 24 bits.

$$X = \pm 1.M \times 2^{E-127} \tag{1}$$

### B. Reduction and Reconstruction Steps

***cos(X),***

$$\boldsymbol{sin(X)}: x = X - n\frac{\pi}{2},\ n = INT\left(X \bullet \frac{2}{\pi}\right) \rightarrow x \in \left[0, \frac{\pi}{4}\right]$$

After reduction, computing *cos(X)*, *sin(X)* will be equivalent to calculating *cos(x)*, *sin(x)* if *n* is even, and equivalent to calculating *sin(x)*, *cos(x)* if *n* is odd. For negative values of *x*, we utilize the identities: $\cos(-x) = \cos x$ and $\sin(-x) = -\sin x$. No reconstruction step is required.

A loss of precision is introduced in carrying out the reduction step. In particular, consider the case when *X* is close to an integer multiple of $\pi$. Thus, the IEEE requirement of maintaining the accuracy of $2^{-24}$ for the final result cannot be achieved for cosine and sine calculations.

The reduction and reconstructions steps for the other functions are well presented in [13]. Table below contains the resultant range after the reduction step. For these functions, the reduction step does not introduce any loss in precision. Therefore, we can maintain an accuracy of $2^{-24}$ for the results of these functions.

**Table 1: Input Range After Reduction**

| Function | Range |
|---|---|
| $\cos x / \sin x$ | $\left[0, \frac{\pi}{4}\right]$ |
| $2^x$ | $(-2, 1]$ $[1, 2)$ |
| $\sqrt{x}$ | $[1, 4)$ |
| $\frac{1}{x}$ | $[1, 2)$ |

## 3. Function Interpolator

A direct table look-up is the simplest method for calculating any function $y = f(x)$ where the input $x$ can be used as the address to look up $y$. This scheme would use an inordinate amount of memory as the number of table entries is an exponential of the input's data width. If we are to reduce the number of entries, using and storing only $N+1$ evenly spaced points in the functional domain, e.g. $(x_0, y_0)$, $(x_1, y_1)$, ... $(x_N, y_N)$, any entries that are missing from the table must be interpolated by means of interpolating polynomials. For instance, a unique second-degree interpolating polynomial can be defined for a subinterval with two end points, $(x_i, y_i)$ and $(x_{i+1}, y_{i+1})$, and an additional third point. One example for the additional third point is the midpoint of the subinterval, $(x_m, y_m)$ where $x_m=(x_{i+1}+x_i)/2$. Figure 1 illustrates the interpolation method in which the interpolation range is $h=x_{i+1}-x_i$ and the interpolating polynomial is $P_2(x)=bx^2+ax+c$. The function's coefficients, *a*, *b*, and *c*, can either be calculated on-the-fly from tabulated function values (*stored function values*) or precomputed and stored as *stored coefficients*. Using the method of *stored function values*, each interpolation subinterval needs to store three function values. However, since it shares its end points with its neighboring subintervals, it is effectively storing two function values. With the *stored coefficients* method, the subinterval needs to store three coefficients. As a result, the *stored function values* method uses one third fewer look-up table memory entries with the expense of extra hardware and time for calculating the coefficients on-the-fly. The design issue here is how to minimize this extra hardware and computational time for a given approximation polynomial.

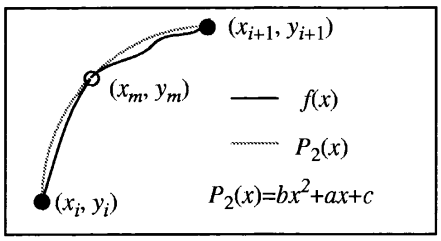


Figure 1: Second-Degree Interpolation

The second-degree approximation polynomial discussed above is not optimal with respect to minimizing the maximum approximation error [1][11]. Instead, the optimal interpolating polynomial uses Chebyshev nodes whose values $(t_j)$ on [-1,1] are:

$$t_j = \cos\left(\frac{(2 \cdot j + 1)\pi}{2N}\right),\ j = 0, 1, 2, \ldots, N-1 \qquad (2)$$

The Chebyshev nodes are then transformed from [-1,1] to $[a, b]$ by the following formula:

$$w_j = t_j \frac{b-a}{2} + \frac{a+b}{2} \qquad (3)$$

The three Chebyshev nodes dividing subinterval $[x_i, x_{i+1}]$, i.e., $N$=3, become:

$$x_{i,-1} = -\frac{\sqrt{3}}{2}\frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2}$$

$$x_{i,0} = \frac{x_{i+1} + x_i}{2} \qquad x_{i,1} = \frac{\sqrt{3}}{2}\frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2} \qquad (4)$$

These three Chebyshev nodes and their corresponding function values uniquely identify a second-degree interpolating polynomial on subinterval $[x_i, x_{i+1}]$.

The maximum approximation error for using the second-degree polynomial generated using Chebyshev nodes is $E_{cheb}$ [11]:

$$E_{cheb}(x) \le \frac{h^3}{24}|f'''(c)| \qquad x_i \le c < x_{i+1} \qquad (5)$$

where $h$ is the subinterval width, $h = x_{i+1} - x_m = x_m - x_i$. In comparison, the approximation error for the polynomial using evenly-spaced points is $E_{even}$[11]:

$$E_{even}(x) \le \frac{h^3}{9\sqrt{3}}|f'''(c)| \qquad x_i \le c < x_{i+1} \qquad (6)$$

For a required accuracy of approximation results, the lower error bound of the Chebyshev series approximation may translate into a bigger subinterval width with resultant fewer subintervals for a given argument domain. However, the actual number of subintervals used also depends on the specific function being approximated and the constraint on the subinterval width, i.e., a negative power of 2 for ease of hardware implementation. Simulation results show that the error margin provided by the Chebyshev series approximation may not be sufficient to reduce the number of subintervals by a factor of 2, and therefore table look-up size which is generally a power of 2.

Given function values, one well-known method for finding the coefficients of the interpolating polynomial is the Lagrange Approximation. According to Lagrange, a second-order approximation polynomial $P(x)$ that passes through $(x_i, y_i)$, $(x_m, y_m)$ and $(x_{i+1}, y_{i+1})$ can be formed by:

$$P(x) = \frac{(x-x_m)(x-x_{i+1})}{(x_i-x_m)(x_i-x_{i+1})}y_i + \frac{(x-x_i)(x-x_{i+1})}{(x_m-x_i)(x_m-x_{i+1})}y_m + \frac{(x-x_i)(x-x_m)}{(x_{i+1}-x_i)(x_{i+1}-x_m)}y_{i+1}$$

Coefficients for each order of $x$ can be calculated by collecting like terms and it is difficult to compute them on-the-fly. An alternative for finding coefficients is to use a family of algorithms for interpolation with equally spaced data points known as the divided difference, which includes the Newton-Gregory Forward, Newton-Gregory Backward, Gauss Forward, Gauss Backward, Bessel and Stirling methods [1]. The Gregory Forward algorithm computes a second-degree polynomial $P(x)$ passing through $(x_i, y_i)$, $(x_m, y_m)$ and $(x_{i+1}, y_{i+1})$ as follows:

$$P(x) = \frac{s^2}{2}(y_{i+1} - 2y_m + y_i) + \frac{s}{2}(y_{i+1} - y_i) + y_m \qquad (7)$$

$$= \frac{s^2}{2}b + \frac{s}{2}a + c = \frac{s}{2}(a + sb) + c$$

where

$$s = \frac{(x-x_m)}{k} \text{ and } k = x_{i+1} - x_m = x_m - x_i$$

EQ. (7) shows that the polynomial's coefficients are a weighted sum of existing function values and can be easily calculated.

## 4. Hybrid Method

### A. Hybrid Algorithm

The simplicity of the divided difference method in computing the polynomial's coefficients leads to the development of a hybrid scheme. The hybrid method stores function values as well as one coefficient, coefficient $b$, for each interpolation subinterval. The coefficient $b$, as shown in EQ. (7), is a weighted sum of three function values $y_i$, $y_m$, and $y_{i+1}$. Storing the coefficient $b$ eliminates the latency associated with its computation, which lies on the critical path of the overall computation. In addition, the cofficient $b$ is a second difference of function values and it has the smallest dynamic range among the three coefficients. It requires the least amount of memory. As a result, the hybrid method has the advantages of both *stored function values* and *stored coefficients* methods.

As specified in EQ. (7), coefficients $a$ and $c$ can be computed from $y_i$, and $y_{i+1}$. For subinterval $[x_i, x_{i+1}]$, $y_i$ is the function value at $x_i$, $y_{i+1}$ is the function value at $x_{i+1}$, $b$ becomes $b_i$, and $y_m$ corresponds to the function value at the subinterval's midpoint. Our hybrid method stores $y_{i+1}$, $y_i$, and $b_i$. Since neighboring subintervals share

function values, we are effectively storing only one function value and one coefficient for each subinterval.

Similar to the *stored function values* method, our hybrid method saves approximately one third of the look-up table memory over the *stored coefficients* method. The advantage it has over the *stored function values* method is that one of the coefficient, $b$, is precomputed. This takes the calculation of $b$ out of the critical path of the overall computation. While the multiplication of $s$ and $b$ takes place, $a$ and $c$ can be calculated using EQ (8) and EQ (9) below.

$$a_i = y_{i+1} - y_i \tag{8}$$

$$c_i = \frac{y_{i+1} + y_i - b}{2} \tag{9}$$

### B. The Architecture and Implementation

Our architecture uses three separate look-up tables: $b$-ROM for $b$ coefficients, $f$-ROMe for even-indexed function values (e.g. $f_0$, $f_2$,... etc.), and $f$-ROMo for odd-indexed function values. This is shown in Figure 2. Consider, as an example, subinterval $[x_i, x_{i+1}]$ as one of 256 subintervals where the $b$-ROM has 256 entries, $f$-ROMe 128 entries, and $f$-ROMo 128 entries. That is, $0 \le i \le 255$ and is represented with 8 bits. The 8-bit $i$, corresponding to the most significant 8 bits of the 23-bit function input $x$, is used to retrieve the $b_i$ coefficient. In case of an even $i$, $\frac{i}{2}$ (represented by $i$'s leading 7 bits) is used to address both $f$-ROMe and $f$-ROMo to retrieve $f_i$ and $f_{i+1}$ respectively in order to compute $a_i$ and $c_i$ as shown in EQ (8) and EQ (9). Namely, the $f$-ROMe's output is subtracted from that of $f$-ROMo in obtaining $a_i$. If $i$ is odd, $\frac{i-1}{2}$ ($i$'s leading 7 bits) and $\frac{i+1}{2}$ ($i$'s leading 7 bits plus one) address $f$-ROMo and $f$-ROMe to retrieve $f_i$ and $f_{i+1}$ respectively. The $a_i$ value is obtained by subtracting $f$-ROMo's output from $f$-ROMe's, an operand swap with respect to the even-$i$ case. In summary, $f$-ROMo is addressed by $i$'s leading 7 bits, and $f$-ROMe is addressed by the sum of $i$'s leading 7 bits and its 8th bit, d, which is 0 for an even $i$ and 1 for an odd $i$. A register (not shown on the diagram) is used to store the boundary function value, i.e. $f_{256}$, when adding $i$'s leading 7 bits and d generates a carry. The d value is also used by the subtractor to select appropriate order of operands. Such memory organization and addressing schemes eliminate the need for complex memory structure used by alternative implementations [9].

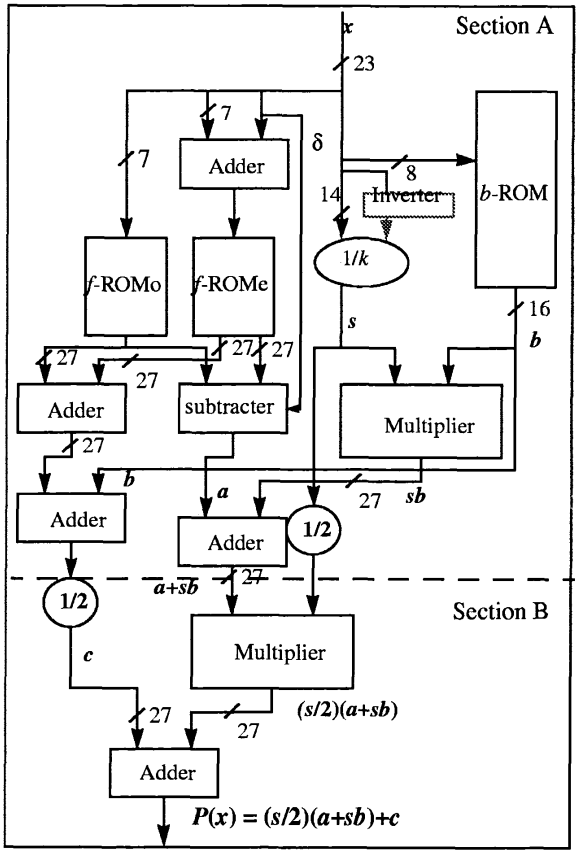


Figure 2: Architecture of the Hybrid Method

As shown in Figure 2, our hybrid method saves 33% of the memory at the expense of two adders and one subtractor, which compute the $a$ and $c$ coefficients on the fly. This trade-off is worthwhile as the additional adders and subtractor are not on the critical path, and they can be shared among multiple functions. The critical path is dictated by the 16-bit multiplier with $s$ and $b$ inputs. Notice that $s$ can be either a positive or negateive number whose sign bit of its 2's complement representation is the inverted most significant 9th bit of a 23-bit $x$ input. In total, the memory used per function is 1.34 KBytes implementing 256 table entries of 27-bit function values and 16-bit $b$ values.

The dotted line in Figure 2 indicates the placement of pipeline registers which divide the whole circuit into two sections. Section A dictates the overall clock frequency, since it has an additional ROM access. Another variation of the architecture is to fold Section B into Section A [7] for hardware economy at the expense of longer latency.

## 5. Dynamic Range Reduction (DRR) Method

### A. DRR Algorithm

In order to further reduce the memory requirements of function generators, another possibility is to reduce the number of bits stored for each function entry, that is, to reduce the dynamic range of the function. The range reduction can be achieved by subtracting a linear reduction function $R(x)$ from the target function. The reduced function can be calculated using the Hybrid architecture described above. In the end, $R(x)$ is added back to the reduced function value to obtain the final results.

The $R(x)$ function can be generated by fitting a linear equation through two Chebyshev nodes given by EQ (2) and EQ (3). For instance, for $\sqrt{x}$ within the interval [1,4), $R(x) = 0.316x + 0.77$. Memory savings thus achieved are at the expense of an additional multiplier and two adders. The extra additions can be absorbed by the final multiplier, but the multiplication can not be eliminated unless $\tilde{R}(x) = (-1)^j 2^k x + l$ is used, where $j \in \{0, 1\}$, the shift amount $k$ is an integer and offset $l$ is found by minimizing $f(x) - \tilde{R}(x)$. Table 2 shows $\tilde{R}(x)$ for all the functions of interest and their corresponding memory savings.

**Table 2: Memory Savings Using the DRR Method**

| Target Function | Reduction Function | Memory Saving |
|---|---|---|
| $\cos x$ | $\tilde{R}(x) = -0.25x + 1$ | 7% |
| $\sin x$ | $\tilde{R}(x) = x$ | 8% |
| $x^{-1}$ | $\tilde{R}(x) = -2x + 2.81$ | 7% |
| $\sqrt{x}$ | $\tilde{R}(x) = 0.25x + 0.77$ | 7% |
| $2^x$ | $\tilde{R}(x) = x + 1.52$ | 7% |

### B. DRR Architecture

The architecture for the Dynamic Range Reduction method needs little extra hardware with respect to the architecture for the Hybrid scheme presented in Section IV. This is shown in Figure 3 in which each function requires two additional memory entries specifying the shift amount $k$ and offset $l$. The shift amount controls a barrel shifter whose output along with $l$ and the computed function value feeds a three-operand adder to produce the final result. If the first order coefficient is negative, i.e. $j = 1$, then the shifter/2's complementer will generate the 2's complement for its input before shifiting it. As mentioned earlier, this three-operand adder can be merged into the final multiplier of the Hybrid architecture. By reducing the dynamic range of the function value, which requires fewer bits to store each function entry, this scheme can save the look-up table size by approximately 7% with minimal speed penalty and additional hardware.

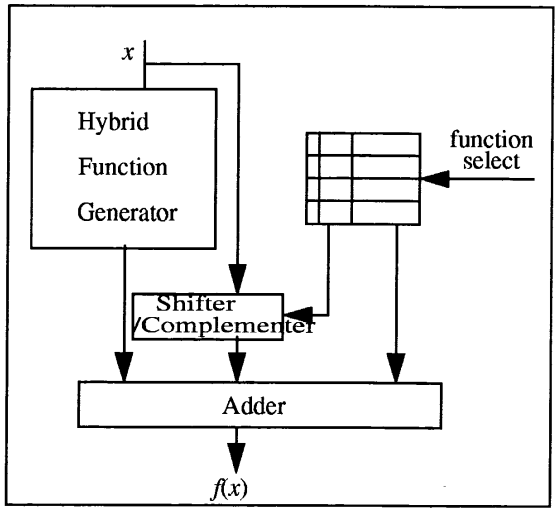


Figure 3: Architecture for Dynamic Range Reduction

## 6. Composite Polynomial

### A. Algorithm

The two schemes discussed previously use two different techniques for reducing look-up table sizes: The Hybrid scheme shares stored function values between adjacent subintervals, and the Dynamic Range Reduction method stores fewer bits for the reduced functions. The third technique is to devise a second-degree approximation polynomial whose lower error bound may result in fewer subintervals for a given argument domain. The third approach can be illustrated by considering the expanded view of Figure 1, as shown in Figure 4.

Points $(x_{i-1}, y_{i-1})$, $(x_i, y_i)$, $(x_{i+1}, y_{i+1})$ and $(x_{i+2}, y_{i+2})$ are evenly spaced on the $x$ axix. Point $(x_m, y_m)$ is the midpoint between $(x_i, y_i)$ and $(x_{i+1}, y_{i+1})$. Polynomial $g_i$ goes through points $(x_{i-1}, y_{i-1})$, $(x_i, y_i)$, and $(x_{i+1}, y_{i+1})$, and polynomial $g_{i+1}$ passes through $(x_i, y_i)$, $(x_{i+1}, y_{i+1})$ and $(x_{i+2}, y_{i+2})$. These two polynomials can be expressed by:

$$g_i = \frac{s^2}{2}(y_{i+1} - 2y_i + y_{i-1}) + \frac{s}{2}(y_{i+1} - y_{i-1}) + y_i \qquad (10)$$

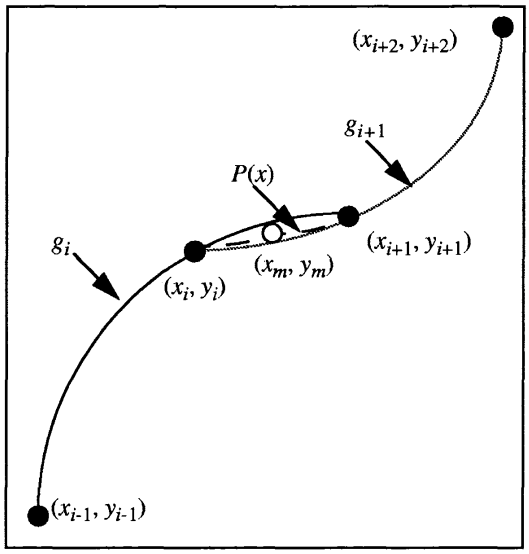


Figure 4: Overlapping Polynomials

$$g_{i+1} = \frac{s(s-2)}{2}(y_{i+2} - 2y_{i+1} + y_i) + \frac{s}{2}(y_{i+2} - y_i) + y_i \qquad (11)$$

where $s = \frac{x - x_i}{h}$, $x_{i-1} \le x \le x_{i+2}$, $h = x_{i+1} - x_i$.

Based on $g_i(x)$ and $g_{i+1}(x)$, a family of parameterized polynomials can be defined for $x_i \le x \le x_{i+1}$:

$$g(x) = \Delta g_i(x) + (1 - \Delta) g_{i+1}(x) \qquad (12)$$

where $0 \le \Delta \le 1$. The second-order polynomials thus defined all go through points $(x_i, y_i)$, and $(x_{i+1}, y_{i+1})$. In particular, for $\Delta = 1$, $g(x) = g_i(x)$, and for $\Delta = 0$, $g(x) = g_{i+1}(x)$. If we let $\Delta = \frac{y_m - g_{i+1}(x_m)}{g_i(x_m) - g_{i+1}(x_m)}$, then the corresponding $g(x)$ passes through $(x_i, y_i)$, $(x_m, y_m)$ and $(x_{i+1}, y_{i+1})$. That is, the $g(x)$ is the second-degree approximation polynomial used in the Hybrid scheme discussed in Section IV. The optimal $\Delta$ parameter can be found for each subinterval by minimizing the approximation error between $g(x)$ and $f(x)$. Such requires storing the optimal $\Delta$ parameter for each subinterval, thus increasing the memory size. An alternative is to choose a fixed $\Delta = \frac{1}{2}$ for all subintervals:

$$\tilde{g}(x) = \frac{1}{2}(g_i(x) + g_{i+1}(x)) \qquad (13)$$

The error bound for $\tilde{g}(x)$ can be derived by considering the following lemma.

*Lemma 1*: Suppose on the interval $[x_{i-1}, x_{i+2}]$, $f(x)$ is continuous, its derivatives up to its third order are continuous, and its second derivative is monotonically increasing or decreasing. Then on its subinterval $[x_i, x_{i+1}]$, $g_i(x) \le f(x) \le g_{i+1}(x)$ or $g_{i+1}(x) \le f(x) \le g_i(x)$. The equality holds only at end points $x_i$ and $x_{i+1}$.

*Proof:* Let $E_i(x)$ and $E_{i+1}(x)$ be the error term for $g_i(x)$ and $g_{i+1}(x)$ respectively, i.e. $E_i(x) = f(x) - g_i(x)$ and $E_{i+1}(x) = f(x) - g_{i+1}(x)$. They have the following formulations [11]:

$$E_i(x) = (x - x_{i-1})(x - x_i)(x - x_{i+1}) f'''(c_1) \qquad (14)$$

$$E_{i+1}(x) = (x - x_i)(x - x_{i+1})(x - x_{i+2}) f'''(c_2) \qquad (15)$$

where $x_{i-1} < c_1 < x_{i+1}$ and $x_i < c_2 < x_{i+2}$.

Consider $x$ in the interval $[x_i, x_{i+1}]$ for EQ. (14) and EQ. (15). These two expressions have opposite signs given the second derivative of $f(x)$ is monotonically increasing or decreasing on $[x_{i-1}, x_{i+2}]$. Both expressions are zero only when $x = x_i$, or $x = x_{i+1}$.

The error terms in EQ. (14) and EQ. (15) have the same bound on their magnitudes :

$$|E_i(x)| \le \frac{h^3}{16}|f'''(c)|$$

$$|E_{i+1}(x)| \le \frac{h^3}{16}|f'''(c)| = \frac{h^3}{16}M \qquad x_{i-1} < c < x_{i+1} \qquad (16)$$

where $h = x_{i+1} - x_i = x_{i+2} - x_{i+1} = x_i - x_{i-1}$. Let $E_{even}$ designate the right-hand side of EQ. (15). Then the error bound for $\tilde{g}(x)$ on $[x_i, x_{i+1}]$ can be formulated as follows, assuming $g_{i+1}(x) \le f(x) \le g_i(x)$ from *Lemma 1*:

$$\begin{aligned} E_{comp} &= |\tilde{g}(x) - f(x)| \\ &= \left|\frac{1}{2}(g_i(x) + g_{i+1}(x)) - f(x)\right| \\ &= \left|\frac{1}{2}(g_i(x) - f(x)) - \frac{1}{2}(f(x) - g_{i+1}(x))\right| \\ &\le \frac{1}{2}E_{even} \\ &\le \frac{h^3}{32}|f'''(c)| = \frac{h^3}{32}M \qquad x_{i-1} < c < x_{i+1} \end{aligned} \qquad (17)$$

EQ. (17) shows that the error bound for the composite polynomial is one half of that of the interpolating polyno-

mial using evenly-spaced nodes. Note that for the error bound presented in EQ. (6), the distance between equally-spaced nodes is $h/2$ whereas it is $h$ for $g_i(x)$ or $g_{i+1}(x)$. Taking into account these differences, the error bounds in both cases (EQ. (6) and EQ. (16) ) are the same. In the case of Chebyshev series approximation, the three Chebyshev nodes in the region of $[x_{i-1}, x_{i+1}]$ have a distance of $\frac{\sqrt{3}}{2}h$ between adjacent nodes, and they produce an approximation polynomial whose error bound on the subinterval $[x_i, x_{i+1}]$ is:

$$E_{cheb} \le \frac{h^3}{24}M \qquad h = x_{i+1} - x_i \tag{18}$$

Comparing EQ. (17) and EQ. (18) shows that the second-degree composite polynomial $\tilde{g}(x)$ has the lower error bound than the polynomial using evenly-spaced nodes or the Chebyshev series approximation on the subinterval [ $x_i, x_{i+1}$]:

$$E_{comp} < E_{cheb} < E_{even} \tag{19}$$

Using the composite polynomial for all functions of interest requires fewer, or the same number of subintervals (entries) as the Hybrid scheme. This is shown in Table 4. However, for each subinterval the composite polynomial requires storing only one function value, whereas the Hybrid scheme requires storing two values: one function value and one coefficient $b$. Thus the composite polynomial uses less memory as shown in Table 3. The cost is a more complex architecture and a slower circuit.

**Table 3: Hybrid Scheme v.s. Composite Polynomial Approach**

| | Hybrid Scheme | | Composite Polynomial | |
|---|---|---|---|---|
| functions | No. of Entries | Total Memory (KBytes) | No. of Entries | Total Memory (KBytes) |
| cosine/ sine | 256 | 1.34 | 128 | 0.43 |
| $2^x$ | 256 | 1.34 | 128 | 0.43 |
| sqrt | 256 | 1.34 | 128 | 0.43 |
| recip. | 256 | 1.34 | 256 | 0.86 |

## B. Composite Polynomial Architecture

Plugging EQ. (10) and EQ. (11) into EQ. (13) results in the following equation:

$$\tilde{g} = \frac{1}{4}[s^2(y_{i-1} - y_i - y_{i+1} + y_{i+2}) \tag{20}$$
$$+ s(-y_{i-1} - 3y_i + 5y_{i+1} - y_{i+2}) + 4y_i]$$

Let $b = y_{i-1} - y_i - y_{i+1} + y_{i+2}$ and $a = -b - 4y_i + 4y_{i+1}$, then EQ. (20) becomes

$$\tilde{g} = \left[\frac{1}{4}s(bs + a) + 4y_i\right] \tag{21}$$

The CPM architecture does not need the coefficient ROM and stores function values into four ROMs: $ROM_0$, $ROM_1$, $ROM_2$, and $ROM_3$. That is, the function value associated with subinterval $i$ gets stored in $ROM_{i \bmod 4}$. Consider a function that requires 256 function values with an 8-bit subinterval index $i$. If we let the leading 6 bits of $i$ be $\hat{x}$ and its trailing 2 bits be $d_1d_0$ , Table 4 lists the addressing function for each ROM and its retrieved value for each of four $d_1d_0$ values.

**Table 4: ROM Addressing for the CPM Architecture**

| | $ROM_0$ | $ROM_1$ | $ROM_2$ | $ROM_3$ |
|---|---|---|---|---|
| Addressing Function | $\hat{x}+\delta_1$ | $\hat{x}+\delta_1 \bullet \delta_0$ | $\hat{x}$ | $\hat{x}-\overline{\delta_1} \bullet \overline{\delta_0}$ |
| $\delta_1\delta_0$=00 | $f_i$ | $f_{i+1}$ | $f_{i+2}$ | $f_{i-1}$ |
| $\delta_1\delta_0$=01 | $f_{i-1}$ | $f_i$ | $f_{i+1}$ | $f_{i+2}$ |
| $\delta_1\delta_0$=10 | $f_{i+2}$ | $f_{i-1}$ | $f_i$ | $f_{i+1}$ |
| $\delta_1\delta_0$=11 | $f_{i+1}$ | $f_{i+2}$ | $f_{i-1}$ | $f_i$ |

Figure shows the block diagram for a CPM architecture. The *Address Function* module implements the addressing functions with a set of incrementers as specifiend in Table 4. The retrieved function values feed an ordering circuit made of multiplexors, which sorts the function values into proper order as described in Table 4. The ordered function values are then forwarded to a four-operand adder/subtracter that generates $b$. The result of the first multiplier, $bs$, is fed into another four-operand adder/subtracter together with the coefficient $b$ and $4y_i$ and $4y_{i+1}$ to compute $bs+a$.

The critical path of the CPM architecture goes through the *Address Function* module, ROM, ordering circuit, two four-operand adders, and two multipliers. The last adder can be absorbed into the second multiplier.

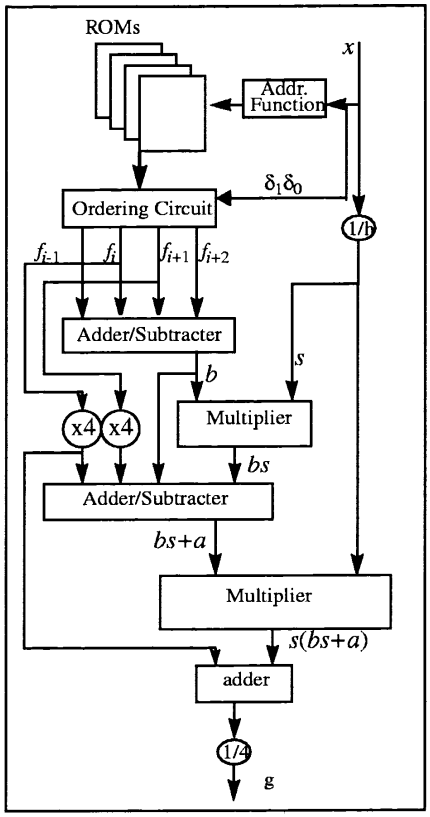


Figure 5: A CPM Architecture

## 7. Summary and Conclusion

Our proposed three architectures for function generation represent a range of trade-offs among memory usage, computational circuits, and speed. The Hybrid scheme uses most memory and least computational circuitry, and the Composite Polynomial Method uses 30% less memory and slightly more circuitry. The speed difference between the two is four carry-save adders. The Dynamic Range Reduction Method represents a midway solution between the two.

The Hybrid scheme compares favorably with published designs using second-order interpolation, which were proposed by Schulte [13], Jain [8], and Lewis [9]. Schulte's architecture implements the *stored coefficients* method and generates exactly rounded results, using second-order Chebyshev polynomials. Jain's scheme uses an interpolating polynomial that passes through the two end points of each subinterval and has its first derivative equal to the function's at the smaller end point. The resulting polynomial has an error bound better than the evenly spaced method but worst than Chebyshev's. By manipulating the polynomial, "Jain's scheme implements the second-order term with a small look-up ROM instead of a square circuit [8]. Lewis uses the *stored function value* method to generate logarithmic numbers used in a logarithmic number system unit [9]. In order to reduce the number of memory accesses in retrieving three function values from the look-up table, Lewis uses an interleaved ROM and a rotator. This approach adds extra delay to the critical path and results in inefficient ROM usage. The hardware requirements of these three existing architectures and our hybrid scheme are summarized in Table (5). As shown in the table, our scheme uses at least one third less memory than alternative methods while using comparable computational units. Schulte's scheme requires much more memory due to its additional accuracy requirement of producing exactly rounded results. His method of producing exactly rounded results can be extended to our hybrid scheme. The critical path of our architecture is comparable to those of Schulte's and Jain's, but better than that of Lewis' primarily due to Lewis' use of complex ROM and a rotator.

**Table 5: Comparison with Existing Architectures**

| | Schulte | Jain | Lewis | Hybrid |
|---|---|---|---|---|
| Function | $2^x$, $\sqrt{x}$<br>$\log 2(x)$ | $2^x$, $\sqrt{x}$<br>$\text{atan}x$, $\cos x/\sin x$ | $\log 2(x)$ | $2^x$, $\sqrt{x}$, $\text{atan}x$<br>$\cos x/\sin x$ |
| Table Size (per function) | $17408 \times 90$ | $256 \times 57$<br>$384 \times 57$ (sq rt) | $299 \times 310$ | $256 \times 43$ |
| Accuracy | Exact Rounding | $2^{-24}$ | $2^{-24}$ | $2^{-24}$ |

In summary, through innovations in architectural design and algorithm developments, we have developed a family of three high-performance hardware for function generation. Different from existing designs, our hardware for Hybrid scheme uses minimal memory to store a combination of coefficients and function values for function interpolation while incurring no speed penalty. It uses simple memory structure and addressing scheme to quickly retrieve function values in parallel. In addition, our development of Dynamic Range Reduction Method and the Composite Polynomial Method further reduces the memory requirements of hardware function generator.

## 8. References


[1] Jun Cao and Belle Wei, "High-Performance Hardware for Function Geneartion," *The 13th Symposium on Computer Arithmetic*, pp. 1984-189.

[2] William J. Cody, Jr., "Software Manual for the Elementary Functions," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

[3] Curtis F. Gerald and Patrick O. Wheatley, *Applied Numerical Analysis*, Fourth Edition, Addison-Wesley Publishing Company, June 1989, pp. 189-203.

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

[5] Shmuel Gal, "Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance," *Accurate Scientific Computations, Lecture Notes in Computer Science*, Springer New York, 1985,pp. 1-16.

[6] John F. Hart, et al, "Computer Approximations," John Wiley & Sons, Inc., New York, 1968.

[7] Xiaoping Huang, Belle W. Y. Wei, Honglu Chen, and Yuhai H. Mao, "High-Performance VLSI Multiplier with a New Redundant Binary Coding," *Journal of VLSI Signal Processing,* Vol. 3, pp. 283 - 291, 1991.

[8] Vijay K. Jain, Subrid A. Wadekar, and Lei Lin, "A Universal Nonlinear Component and Its Application to WSI,"*IEEE Transactions on Components, Hybrids and Manufacturing Technology*, Volume 16, Number 7, November 1993, pp. 656-664.

[9] David M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit,"*IEEE Transactions on Computers*, Volume 43, Number 8, August 1994, pp. 974-982.

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

[11] John H. Mathews, *Numerical Methods for Computer Science, Engineering and Mathematics*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1987, pp. 166-209.

[12] James A. McIntosh and Earl E. Swartzlander, Jr., "High-Speed Cosine Generator," *IEEE Proceeding*, 1995, Pages 273-277.

[13] Michael J. Schulte and Earl E. Swartzlander, Jr., "Hardware Designs for Exactly Rounded Elementary Functions," *IEEE Transactions on Computers*, Volume 43, Number 8, August 1994, Pages 964-972.

[14] Ping Tak Peter Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th Symposium on Computer Arithmetic*, 1991, pp. 232-236.

[15] Ping Tak Peter Tang, "Table-Driven Implementation of the Logarithm Function," *ACM Transactions on Mathematical Software*, Volume 16, Number 4, December 1990, pp. 380-400.