High-Performance Hardware for Function Generation

Jun Cao, Belle W. Y. Wei*
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. The paper presents a new interpolator architecture for generating elementary functions based on an optimal trade-off between the use of memory modules and computational circuits. The architecture uses one third less memory than alternative schemes while incurring no time penalty and minimal additional circuit. The pipelined design has a throughput of generating one functional value per clock cycle, and a latency of two clock cycles.

I. Introduction

Elementary functions such as trigonometric functions, square-root, and exponential are essential to many DSP applications. These functions are often implemented in software routines [2][3] 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.

In this paper we propose a new design for a hardware function generator based on an optimal trade-off between memory and computational circuits. The generator uses a second-degree interpolating polynomial passing through Chebyshev nodes. Different from existing designs [5][6][9][10], our scheme stores a combination of the polynomial's coefficients and function values for fast function interpolation. Our design implements function generation for cosine, sine, reciprocal, square root, and exponential functions. With this scheme, three steps [11][12] are involved in finding $Y \doteq f(X)$ 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.

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

In Section II, we present range reduction and reconstruction steps for the functions of our interest. We then discuss current interpolation methods as well as our proposed method in Section III. Section IV presents the architecture and implementation of our scheme. Its performance and hardware requirements are compared with those of existing schemes in Section V. The last section summarizes our results.

II. Range Reduction and Reconstruction

In calculating $Y \doteq f(X)$, 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 step in order to compute Y. Presented below are the range reduction and reconstruction algorithms for the five functions implemented by our function generator.

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), *sin*(X):

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

After reduction, calculation of *cos*(X), *sin*(X) will be calculating *cos*(x), *sin*(x) if n is even, and calculating *sin*(x), *cos*(x) if n is odd. No reconstruction step is required.

Exponential (e^X):

$$e^X = 2^{\frac{X}{\ln 2}} = 2^{\frac{1.M}{\ln 2} \cdot 2^E} = 2^{E'+x} = 2^x \cdot 2^{E'}$$

where E' is an integer and $x \in [0, 1)$

Reciprocal ($1/X$):

$$x = 0.1M \rightarrow x \in [0.5, 1), E' = 255 - E$$

$$\frac{1}{X} = \frac{1}{x} \cdot 2^{E'}$$

Square Root:

if E is even: $x = 0.01M, E' = \frac{(E+2)}{2}$

if E is odd: $x = 0.1M, E' = \frac{(E+1)}{2}$

$\rightarrow x \in [0.25, 1)$

$$\sqrt{X} = \sqrt{x} \cdot 2^{E'}$$

III. Function Interpolator

A. Interpolation Methods

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 space, 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 be either 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 less of the look-up table memory at 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.

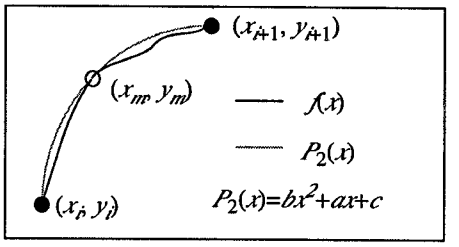

Figure 1: Second-Degree Interpolation

The second-degree approximation polynomial discussed above is not optimal with respect to minimizing the maximum approximation error [1][8]. Instead, the optimal interpolating polynomial uses Chebyshev nodes whose values (t_j) on [-1,1] are:

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

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

$$w_j = t_j \frac{b-a}{2} + \frac{a+b}{2}$$

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

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

$$x_{i,0} = \frac{x_{i+1}+x_i}{2}$$

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

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

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 $P(z)$ that passes through (z_{-1}, g_{-1}), (z_0, g_0) and (z_1, g_1) can be formed by:

$$P(z) = \frac{(z-z_0)(z-z_1)}{(z_{-1}-z_0)(z_{-1}-z_1)}g_{-1} + \frac{(z-z_{-1})(z-z_1)}{(z_0-z_{-1})(z_0-z_1)}g_0 + \frac{(z-z_{-1})(z-z_0)}{(z_1-z_{-1})(z_1-z_0)}g_1$$

Coefficients for each order of z 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, which includes the Newton-Gregory Forward, Newton-Gregory Backward, Gauss Forward, Gauss Backward, Bessel and Stirling methods [1]. The Stirling algorithm computes a second-degree polynomial $P(z)$ passing through (z_{-1}, g_{-1}), (z_0, g_0) and (z_1, g_1) as follows:

$$P(z) = \frac{s^2}{2}(g_1 - 2g_0 + g_{-1}) + \frac{s}{2}(g_1 - g_{-1}) + g_0 \quad (3)$$
$$= \frac{s^2}{2}b + \frac{s}{2}a + c = \frac{s}{2}(a + sb) + c$$

where

$$s = \frac{(z-z_0)}{k} \text{ and } k = z_1 - z_0 = z_0 - z_{-1}$$

EQ. (3) shows that the polynomial's coefficients are a weighted sum of existing function values and can be easily calculated. Note that the Stirling algorithm is applicable only to evenly spaced points, e.g. $k = z_1 - z_0 = z_0 - z_{-1}$. In the next section, we will discuss how we can use this algorithm to interpolate using Chebyshev nodes.

B. A Hybrid Method

The simplicity of the Stirling algorithm 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 for each interpolation subinterval. It has the advantages of both *stored function values* and *stored coefficients* methods.

The Stirling method specified in EQ. (3) shows that the coefficient b is a weighted sum of three function values g_1, g_0, and g_{-1}. Coefficients a and c can be computed from b, g_1, and g_{-1}. For subinterval $[x_i, x_{i+1}]$, g_{-1} becomes the function value at x_i, g_1 becomes the function value at x_{i+1}, b becomes b_i, and g_0 corresponds to the function value at the subinterval's midpoint. Our hybrid method stores g_{-1}, g_1, and b. 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 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 (4) and EQ(5) shown below.

$$a = g_1 - g_{-1} \quad (4)$$

$$c = \frac{g_1 + g_{-1} - b}{2} \quad (5)$$

Our design is able to interpolate based on evenly spaced points as well as Chebyshev nodes. However, in case of the interpolating polynomial using Chebyshev nodes, the function values stored are not the true function values, but values extrapolated from the polynomial. For instance, given subinterval $[x_i, x_{i+1}]$, its Chebyshev nodes can be found based on EQ. (2). The nodes are then used to generate an interpolating polynomial, $C(x)$. The function values that get stored are $C(x_i)$ and $C(x_{i+1})$ instead of $f(x_i)$ and $f(x_{i+1})$.

IV. 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 255 subintervals where the b-ROM has 255 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 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 $a_i = f_{i+1} - f_i$ and c_i as shown in EQ (5). 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

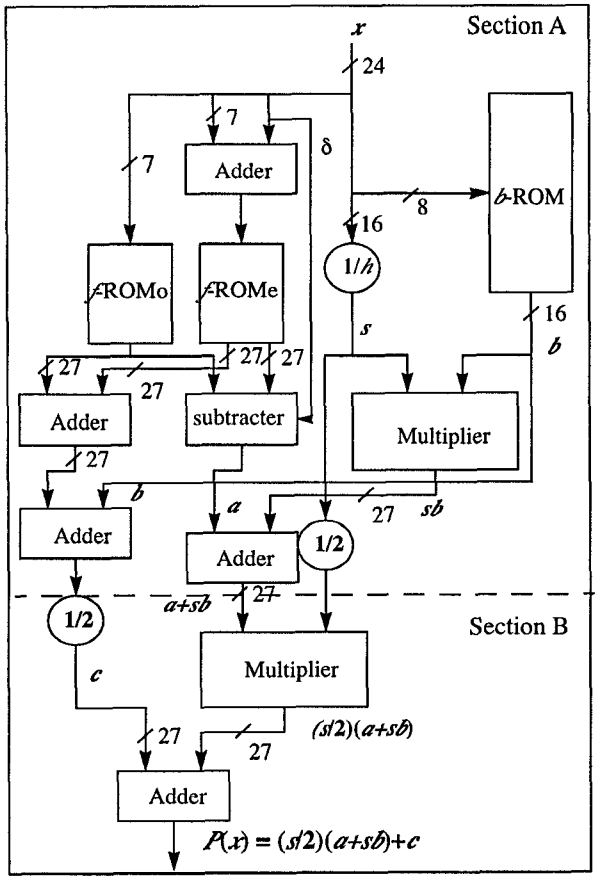

Figure 2: Architecture of the Hybrid Method

to retrieve f_i and f_{i+1} respectively. The a_i value is obtained by subtracting fROMo's output from fROMe's, an operand swap with respect to the even-i case. In summary, fROMo is addressed by i's leading 7 bits, and fROMe is addressed by the sum of i's leading 7 bits and its 8th bit, δ, which is 0 for an even i and 1 for an odd i. The δ 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 structures used by alternative implementations [6].

As shown in Figure 2, our hybrid method saves 1/3 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 data widths of various components are determined by the accuracy required by the IEEE floating point format. The actual memory size used depends on the specific function, and is tabulated in Table 1.

Table 1: Memory Requirement

functions	No. of Entries	fWidth	b-Width	Total Memory (KBytes)
cosine/ sine	202	27	12	0.96
exp	128	27	14	0.64
sqrt	96	27	12	0.46
recip.	256	27	16	1.44

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. Based on LSI Logic's 1.0 micron technology [7], the architecture can sustain a throughput of 55 ns per function calculation with a latency of 110 ns. Another variation of the architecture is to fold Section B into Section A [4] for hardware economy at the expense of longer latency.

V. Comparison with Existing Schemes

Our proposed scheme compares favorably with published results proposed by Schulte [10], Jain [5], and Lewis [6]. 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 [5]. Lewis uses the *stored function values* method to generate logarithmic numbers used in a logarithmic number system unit [6]. 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 2. 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 2: Comparison with Existing Architectures

	Schulte	Jain	Lewis	Hybrid
Function	x^{-1}, $\sqrt{x}$, $\log 2\,(x)$	x^{-1}, $\sqrt{x}$, cos/sine *arc*tan	$\log 2\,(x)$	x^{-1}, $\sqrt{x}$, cos/sine e^x
Table Size (entries) per Function	832 × 3	256 × 3	229 × 10	208 × 2 (cos & sin) 128 × 2(exp) 96 × 2(sqrt) 256 × 2(recip)
Accuracy	exactly rounded	$<2^{-24}$	$<2^{-26}$	$<2^{-24}$
Computational Units	multi. (2) sqr. ckt (1) adder (1)**	multi. (2)* adder (2)	multi. (2) adder (4)	multi. (2) adder (6)

* Uses a secondary look-up ROM to evaluate square function
** Uses a multi-operand adder
*** Uses interleaved ROM

VI.Conclusion

We have developed a high-performance hardware for function generation. Different from existing designs, our hardware uses minimal memory to store a combination of coefficients and function values for function interpolation while incurring no speed penalty. Furthermore, our architecture uses simple memory structure and addressing scheme to quickly retrieve function values in parallel. The result is a competitive design for general-purpose function generation.

VII.References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] LSI Logic, *1.0 Micron Cell-Based Products Data Book*, LSI Logic Corporation, Milpitas, California, 1991.

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

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

[10] 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.

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

[12] 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.