Algorithmic Truncation of MiniMax Polynomial Coefficients

Sherif A. Tawfik [*] and Hossam A. H. Fahmy [†]
Electronics and Communications Department, Faculty of Engineering, Cairo University, Giza, Egypt, 12613.

Abstract— **Elementary and high-level functions can be computed in hardware using polynomial approximation techniques. There are many techniques in the literature to calculate the coefficients of such polynomials. Remez algorithm [1] provides the optimal polynomial in the Chebyshev sense that is minimizing the maximum error (minimax approximation).**

This paper presents an algorithm for truncating the coefficients of the minimax polynomials obtained from Remez algorithm using an algorithmic method. A gain of 3 and 4 bits of accuracy over the direct rounding is reported.

Muller [2] addressed the same problem but his algorithm is applicable for the second order polynomials only. This paper presents an algorithm that is applicable for any order.

I. Introduction

The computation of elementary functions in hardware is performed by three steps. The first step is called argument reduction. Argument reduction maps the given argument into another argument that has a reduced range. The second step approximates the function at the reduced argument. The third step is the reconstruction step where we compute the function at the original argument from the approximated value of the function at the reduced argument.

The range reduction and reconstruction steps are related and they are function-dependent. There is no single reduction and reconstruction technique that is applicable to all functions. The modular reduction is a popular technique. It is applicable for example to the exponential and sinusoidal functions.

The hardware algorithms for approximating functions fall into four broad categories: the digit recurrence [3]–[5], the functional recurrence [6]–[8], the polynomial approximation [2], [7], [9]–[16], and the rational approximation [17].

Each of these categories has some design parameters that need to be set in order to achieve the desired precision at the lowest cost. The cost is either a function of the area, the latency, the power consumed, or any combination of the three. The type of the cost function determines the category that we should employ.

In this paper we are concerned with the polynomial approximation implemented in the parallel architecture. Remez algorithm [1] provides the best polynomial approximation. To minimize the hardware, we aim at truncating the coefficients of the approximating polynomial using an algorithmic method instead of rounding. We model the problem in the form of a mixed integer linear programming problem and solve it using the branch and bound algorithm.

The rest of the paper is organized as follows. Section II presents a review of Remez algorithm. Section III presents our truncation algorithm. Section IV presents the parallel architecture and the usefulness of our truncation algorithm in regard to this architecture. Section V presents the results and comparison. Finally conclusions are given in section VI.

[*] Currently at ECE department, University of Wisconsin-Madison, USA.
[†] Currently at the School of Computer Engineering, Nanyang Technological University, Singapore.

II. Review of Remez Algorithm

Remez algorithm is an iterative algorithm. It aims to find the coefficients of the approximating polynomial such that the maximum error is minimized.

We denote the function that we need to approximate by $F(X)$ where the argument $X \in [a, b]$. We denote the approximating polynomial of order n by $P_n(X)$ such that $P_n(X) = c_0 + c_1X + c_2X^2 + \ldots + c_nX^n$. We define the error function by $e(x) = F(X) - P_n(X)$. The algorithm has two main steps:

The first step is to select $(n + 2)$ points in the given interval. We denote these points by $X_0, X_1, \ldots, X_{n+1}$ such that $a \le X_0 < X_1 < \ldots < X_{n+1} \le b$. Then we force the error function to take the same magnitude at these points with alternating sign.

The second step is to exchange some or all of these $(n+2)$ points by other points in the interval such that we approach the required optimal polynomial. There are two exchange techniques. The first technique exchanges a single point every iteration while the second technique exchanges all points every iteration.

These two steps are repeated several times until the difference between the new and old point(s) becomes less than a given threshold. A detailed description of these two steps follows:

First step: we force the error at the $(n + 2)$ points to be equal in magnitude, denoted by E and to have alternating sign.

$$\begin{aligned} e(X_m) = F(X_m) - P_n(X_m) &= (-1)^m E \\ F(X_m) - (c_0 + c_1X_m + \ldots + c_nX_m^n) &= (-1)^m E \\ m = 0, 1, \ldots, n+1 & \qquad (1) \end{aligned}$$

Equation 1 is a system of $(n+2)$ linear equations in the $(n+2)$ unknowns: $\{c_0, c_1, \ldots, c_n, E\}$. These $(n+2)$ equations are independent [1] therefore there is one unique solution which can be obtained using any algorithm from linear algebra. By solving this system of equations we determine the coefficients of the polynomial and the magnitude of the error at the $(n+2)$ points.

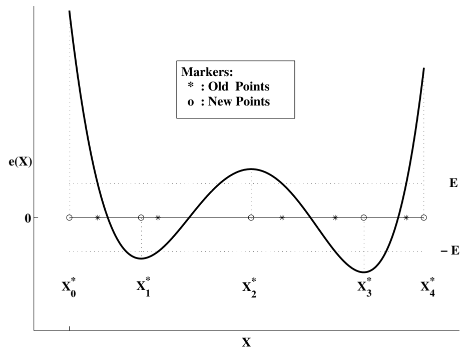

Fig. 1. Illustration of the second step of Remez algorithm for a third order polynomial.

Second step: since the error function $e(X)$ has an alternating sign at the the $(n+2)$ points that we used in the first step therefore the error function crosses the x-axis $(n+1)$ times. We calculate the points at which $e(X)=0$ using a suitable root-finding algorithm and denote these points by $Z_0, Z_1, \ldots, Z_n$.

We then divide the given interval $[a,b]$ into the $(n+2)$ sub-intervals $[a,Z_0],[Z_0,Z_1],[Z_1,Z_2],\ldots,[Z_n,b]$. In each of these sub-intervals we calculate the value of X at which $e(X)$ takes its maximum or minimum value and denote these points by $X_0^*, X_1^*, \ldots X_{n+1}^*$ respectively.

We can calculate the maximum or the minimum of $e(X)$ in a given sub-interval by calculating the root of $\frac{\partial e(X)}{\partial X}$ in the given sub-interval if such root exists. If it doesn't exist we calculate $e(X)$ at the end-points of the sub-interval and choose the one that gives a bigger absolute value for $e(X)$.

We define

$$k = \max_m |e(X_m^*)| \quad m = 0,1,\ldots,n+1 \tag{2}$$

The single exchange technique exchanges X_k with X_k^*. The multiple exchange technique exchanges $\{X_m\}$ with $\{X_m^*\}$ for all $m=0,1,\ldots,n+1$. Figure 1 illustrates the second step graphically for a third order polynomial approximation in the interval $[0,1]$. To start the algorithm the first set of points are chosen arbitrarily.

The algorithm is proved to converge to the optimal polynomial at which the error function reaches its absolute maximum value $(n+2)$ times with alternating sign [1].

III. The Truncation Algorithm

In real hardware, numbers have a finite precision. The area of hardware multipliers decreases if we further constrain the precision of the coefficients in the optimal polynomial. However, such constraints affect the accuracy of the approximation. Our goal is to find an algorithm to constrain the coefficients while having the least effect on the accuracy.

First we run Remez algorithm until we reach the optimal polynomial then we repeat the last iteration with two modifications. The first modification is in the first step of the Remez algorithm. We formulate the first step as a Mixed-Integer Linear Programming (MILP) problem and add precision-constraints to all the coefficients except c_0 and then solve it using the branch and bound algorithm. The second modification is in the second step of the Remez algorithm. After we compute the locations of the local extrema $\{X_n^*\}$ we determine the maximum and minimum values for the error function $e(X)$. We then add the average of these two values to c_0 in order to make the error exactly centered around the origin. The reason why we don't constrain the precision of c_0 is because it is not an operand to a multiplier as we see in the following section. Note that we can determine the maximum error of the approximation numerically from the second step of Remez algorithm.

We formulate the first step (Equation 1) of Remez algorithm as a Linear Programming (LP) problem by subtracting a non-negative variable s_m from the variable E for each equation and setting the objective function of the LP problem to minimize the maximum of the $\{s_m\}$ variables. Since the $\{s_m\}$ variables are non-negative therefore minimizing their maximum implies that we minimize them all. We transform the problem into a standard form by introducing a variable s and letting $s = max(s_m)$ therefore $s \geq s_m$ for all m. Equation 1 is reformulated as **minimize** s **subject to:**

$$\begin{aligned} F(X_m) &- (c_0 + c_1X_m + \ldots + c_nX_m^n) \\ &= (-1)^m(E - s_m) \qquad (3) \\ s_m &\geq 0 \qquad (4) \\ s &\geq 0 \qquad (5) \\ s_m &\leq s \qquad (6) \\ m &= 0,1,\cdots,n+1 \end{aligned}$$

The optimal solution of this problem occurs at $s_m = 0$ for all m and this gives the same solution that we get from solving equation 1 as a linear system of equations using any method from linear algebra.

Note that without the $\{s_m\}$ variables the system has a unique solution and any new constraint will render the problem infeasible. The above LP formulation enables us to add precision-constraints on the coefficients and still get a feasible solution. After adding the precision constraints the LP problem becomes an MILP problem as **minimize** s **subject to:**

$$\begin{aligned} F(X_m) &- (c_0 + c_1X_m + \ldots + c_nX_m^n) \\ &= (-1)^m(E - s_m) \qquad (7) \\ s_m &\geq 0 \qquad (8) \\ s &\geq 0 \qquad (9) \\ s_m &\leq s \qquad (10) \\ m &= 0,1,\cdots,n+1 \\ & c_1, c_2, \ldots, c_n \text{ have J fractional bits} \qquad (11) \end{aligned}$$

In this mixed-integer programming problem, we put precision constraints on all the coefficients except c_0. The solution to this MILP problem gives unequal magnitudes to the error at the $(n+2)$ points. However the differences between the magnitude of the error at the $(n+2)$ points are minimized.

IV. The Proposed Architecture

Polynomial approximation may require a high order for a given precision. Higher order polynomials implies longer computation time for the same hardware or larger hardware for the same computation time. One way to decrease the order of the approximating polynomial is to divide the given interval into a number of smaller sub-intervals and compute the polynomial coefficients for every sub-interval. This solution comes at a price. We now need a table to store the coefficients of those polynomials. However this solution has an advantage in that it gives the designer more freedom to trade-off the size of the table with the order of the approximating polynomials and hence the latency of the algorithm or the area of the other hardware units.

The easiest way for dividing the given interval into smaller sub-intervals is to divide it into a power of 2 equal sub-intervals. This method simplifies the hardware implementation. What remains to fully describe the general architecture is to describe how we select the coefficients of the polynomial of the sub-interval in which a given argument lies and how to evaluate that polynomial from the selected coefficients.

Without loss of generality we assume that the argument X lies in the interval $[0,1[$ and has the binary representation $[0.x_1x_2\dots x_g\dots x_\beta]$ where $x_g \in \{0,1\}$. We divide the argument X into two parts $X_{hi} = 0.x_1x_2\dots x_\alpha$ and $X_{lo} = 0.00\dots 0x_{\alpha+1}\dots x_\beta$ such that $X = X_{hi} + X_{lo}$.

If we divide the interval into equal 2^α sub-intervals we can determine the sub-interval in which a given argument lies by $\lfloor X \times 2^\alpha \rfloor$ and this is equivalent to using the bits of X_{hi} without the leading 0 as an index to the coefficients' table. Each entry in that table should hold the coefficients of the approximating polynomial corresponding to its sub-interval.

We modify Remez algorithm such that the approximating polynomials are functions in X_{lo} instead of X and the dependence on X_{hi} is accounted for in the coefficients. This modification is justified because X_{hi} is constant in all the points of a sub-interval. Thus the approximating polynomial has the form $P_n(X) = c_0X_{lo} + c_1X_{lo}^2 + \cdots + c_nX_{lo}^n$.

The approximating polynomial can be evaluated in hardware using truncated powering units [18] and truncated multipliers [19] and final addition. The use of truncated multipliers and powering units leads to a significant decrease in the area and the delay of these units. Figure 2 illustrates the proposed architecture. It is a generalization of the one given by Muller [2].

In this architecture, we use specialized powering units to compute the powers of X_{lo}. At the same time we read the coefficients from a table by using the bits of X_{hi} as the index. We then multiply the powers of X_{lo} by the coefficients using parallel multipliers. Finally we sum the results using a multi-operand adder.

We note that c_0 is not an operand to any multiplier while each of the remaining coefficients is an operand to a parallel

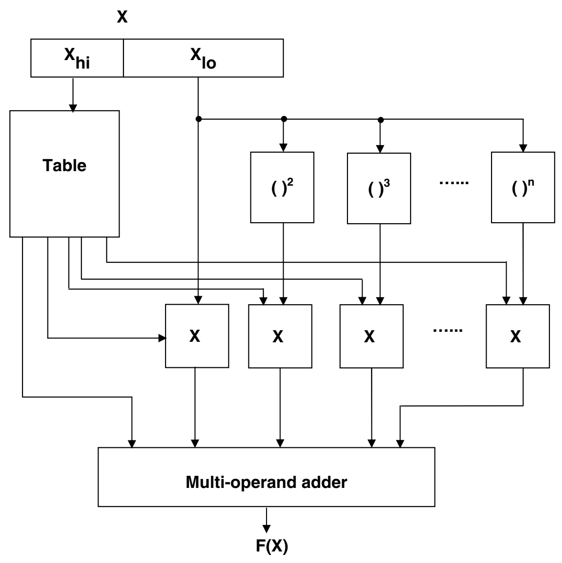

Fig. 2. The proposed architecture.

multiplier hence by reducing the precision of the all the coefficients except c_0 we reduce the size of the employed multipliers. Moreover the powers of X_{lo} are small numbers since $X_{lo} < 2^{-\alpha}$ therefore constraining the precision of the higher order coefficients has less impact on the final accuracy than constraining c_0. Therefore we don't constrain the precision of c_0 beyond the working precision.

It is to be noted that some coefficients may have leading zeros or ones depending on their sign. There is no need to store those leading zeros or ones since they can be obtained by sign extension. The architecture can be easily pipelined into two or three stages by using registers.

In the proposed architecture, there is a trade off between the size of the coefficients table on the one hand and the size of the powering units and multipliers and their number and the size of the multi-operand adder on the other hand. After the functions are specified careful analysis should be carried out to determine the optimal table size, polynomial order and the coefficients width in bits such that the required precision is achieved at the smallest area.

V. Results and Comparison

In this section we present some results of our algorithm and compare them with the results of Muller's algorithm for second order polynomials and with the direct rounding for the second and higher order polynomials.

Table I lists the error of the three different truncation techniques namely Muller's algorithm, direct rounding and our truncation algorithm for different functions, polynomials order and coefficients' precision in bits.

We can deduce from this table that the results of our algorithm is close to that of Muller's algorithm. Our algorithm is slightly better for some cases and slightly worse in others. Muller's algorithm is applicable only for second order polynomials hence we compare our algorithm with the direct rounding for the case of higher order polynomials. It is clear from table I that our algorithm outperforms the direct rounding. The gain in precision of the final result reaches more than 4 bits in some cases.

It is useful here to quickly compare the second order and third order polynomials for single precision approximation (the two entries before the last for each function in table I). Second order polynomials use three coefficients while the third order requires four coefficients. Hence, the table size of the third order is one third $(\frac{16\times 4}{64\times 3})$ that of the second order. This reduction comes at the cost of adding a cubing unit, a multiplier, and an additional addend in the final adder.

If we need to implement more than one function with the powering units and multipliers being shared then the third order polynomials approximation may be more attractive from the area perspective. On the other hand if we need to implement one function then the second order polynomials approximation may be better from the area perspective.

Once the functions that need to be approximated are specified, a designer should carry a thorough comparison between the various orders of polynomial approximations.

VI. Conclusions

Our algorithm outperforms direct rounding in all cases. It is applicable not just to the second order but also to higher order polynomials. Using our algorithm, designers have more options to trade-off area versus speed.

References

[1] L. Veidinger, "On the numerical determination of the best approximations in the Chebychev sense," *Numerische Mathematik*, vol. 2, pp. 99–105, 1960.

[2] J.-M. Muller, "Partially rounded small-order approximation for accurate, hardware-oriented, table-based methods," in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain*, June 2003.

[3] J. E. Volder, "The CORDIC trignometric computing technique," *IRE Transactions on electronic Computers*, pp. 330–334, Sept. 1959.

[4] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[5] S. K. Jean-Claude Bajard and J.-M. Muller, "BKM: A new hardware algorithm for complex elementary functions," *IEEE Transactions on Computers*, vol. 43, pp. 955–963, Aug. 1994.

[6] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, pp. 702–706, Aug. 1970.

[7] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 2–9, July 1995.

[8] A. A. Liddicoat and M. J. Flynn, "High-performance floating point divide," in *Proceedings of Euromicro Symposium on Digital System Design , Warsaw, Poland*, pp. 354–361, Sept. 2001.

[9] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic, Grenoble, France*, pp. 232–236, June 1991.

[10] D. D. Sarma and D. W. Matula, "Faithful interpolation in reciprocal tables," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 82–91, July 1997.

[11] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 126–131, 1997.

[12] D. D. Sarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 17–28, July 1995.

TABLE I

Results for different functions, polynomial order (n), number of sub-intervals 2^{α} and coefficients precision in bits (J)

n	2^{α}	**J**	**error**		
			Muller	**Rounding**	**Our work**
2	8	6	$2^{-12.4}$	$2^{-10.1}$	$2^{-13.1}$
2	16	6	$2^{-13.58}$	$2^{-10.9}$	$2^{-13.94}$
2	16	9	$2^{-16.4}$	$2^{-13.9}$	$2^{-16.67}$
2	32	9	$2^{-17.62}$	$2^{-14.96}$	2^{-18}
2	64	14	$2^{-23.16}$	$2^{-20.9}$	$2^{-23.33}$
3	16	14	-	2^{-19}	2^{-23}
4	8	15	-	$2^{-19.48}$	$2^{-24.39}$
Exponential Function EXP(X)				Interval: $[0, 1]$	
2	8	6	$2^{-12.13}$	$2^{-9.87}$	$2^{-12.89}$
2	32	10	$2^{-18.68}$	$2^{-15.98}$	$2^{-18.88}$
2	64	14	$2^{-23.48}$	$2^{-20.98}$	$2^{-23.69}$
3	16	14	-	2^{-19}	$2^{-23.1}$
4	8	15	-	2^{-19}	2^{-24}
Logarithmic Function Ln(X)				Interval: $[1, 2]$	
2	8	5	$2^{-11.15}$	2^{-9}	$2^{-11.86}$
2	16	7	$2^{-14.66}$	$2^{-12.34}$	$2^{-15.3}$
2	64	14	$2^{-23.53}$	2^{-21}	$2^{-23.17}$
3	16	14	-	$2^{-18.9}$	2^{-23}
4	8	14	-	2^{-18}	2^{-23}
Sinusoidal Function sin(X)				Interval: $[0, 1]$	
2	8	6	$2^{-12.65}$	2^{-10}	2^{-13}
2	16	10	$2^{-17.48}$	2^{-15}	$2^{-17.8}$
2	64	14	$2^{-23.5}$	$2^{-20.98}$	$2^{-23.2}$
3	16	15	-	$2^{-19.94}$	$2^{-23.8}$
4	8	15	-	$2^{-19.13}$	2^{-24}
Square-root reciprocal				Interval: $[1, 2]$	

[13] M. J. Schulte and J. E. Stine, "Symmetric bipartite tables for accurate function approximation," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 175–183, July 1997.

[14] J.-M. Muller, "A few results on table-based methods," *Research Report*, vol. 5, Oct. 1998.

[15] M. J. Schulte and J. E. Stine, "The symmetric table addition for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, 1999.

[16] A. T. Florent de Dinechin, "Some improvements on multipartite table methods," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA*, pp. 128–135, June 2001.

[17] O. Mencer, *Rational Arithmetic Units in Computer Systems*. Ph.D thesis, Stanford University, Stanford, CA, USA, Jan. 2000.

[18] A. A. Liddicoat and M. J. Flynn, "Parallel square and cube computations," in *Proceedings of the 34th Asilomar Conference on Signals, Systems, and Computers, California, USA*, Oct. 2000.

[19] M. J. Schulte and J. Earl E. Swartzlander, "Truncated multiplication with correction constant," *in VLSI Signal Processing VI, IEEE Workshop on VLSI Signal Processing, Eindhoven, Netherlands*, pp. 388–396, 1993.