Parameterized Function Evaluation for FPGAs

Oskar Mencer*, Nicolas Boullis**, Wayne Luk*** and Henry Styles***

*Computing Sciences Research Center, Lucent, Bell Labs, Murray Hill, NJ 07974, USA
**Ecole Normale Superieure de Lyon, 69364 Lyon, France
***Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Abstract

This paper presents parameterized module-generators for pipelined function evaluation using lookup tables, adders, shifters and multipliers. We discuss trade-offs involved between (1) full-lookup tables, (2) bipartite (lookup-add) units, (3) lookup-multiply units, and (4) shift-and-add based CORDIC units. For lookup-multiply units we provide equations estimating approximation errors and rounding errors which are used to parameterize the hardware units. The resources and performance of the resulting design can be estimated given the input parameters. The method is implemented as part of the PAM-Blox module generation environment. An example shows that the table-multiply unit produces competitive designs with data widths up to 20 bits when compared with shift-and-add based CORDIC units. Additionally, the table-multiply method can be used for larger data widths when evaluating functions not supported by CORDIC.

1 Introduction

The evaluation of differentiable functions can often be the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\log(x)$ or $\sqrt{x}$, and compound functions such as $(1 - \sin^2(x))^{1/2}$ or $\tan^2(x)+1$. Hardware implementation of elementary functions are studied by Wong and Goto [16]. Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. Such customization can take place at run time by reconfiguring the FPGA, so that different functions or precisions can be introduced according to run-time conditions.

This paper presents parameterizable designs for evaluating a differentiable function using lookup tables, adders and multipliers, which can be implemented very efficiently using FPGAs and other stream-based architectures [4], [11], [7], [5]. We describe architectures based on full-lookup units, lookup-add units and lookup-multiply units, and compare their size and performance for different input data widths. These architectures are often highly pipelined for high throughput. Such parallelism provides an edge over general-purpose microprocessors.

Module generators are implemented in the PAM-Blox environment [8], so that instances of particular architectures can be generated rapidly and automatically from a parameterized description. The key problem addressed in this paper is the automatic generation of parameterized function evaluation units. The challenge is to find the required intermediate precisions given an input and output precision requirement. For example, if we require a 15-bit sine function from a 13 bit input, the module-generators have to create the function evaluation unit while optimizing speed and area of the unit.

Preliminary results suggest that there is a tradeoff between the various table lookup methods and shift-and-add based (e.g. CORDIC) units based on the number of required bits[2], [9], [14]. This paper sheds some light into the details of this tradeoff, and the applicability of the various methods to a module generation environment, which is the heart of computing with FPGAs.

2 Function Evaluation Methods

Our objective is to provide efficient circuits for differentiable function evaluation. In general, function evaluation consists of three stages. The first stage, range reduction (see for example Muller [10], Chapter 8), reduces the argument x to a small interval $[a, b]$, resulting in a new argument $\overline{x}$. The second stage evaluates the function $F(\overline{x})$. The third stage extrapolates $F(x)$ from $F(\overline{x})$. In this paper, we are only concerned with the second stage: evaluating a function $F(x)$ where x is in a small evaluation interval $[a, b]$. For any given function, area and timing restrictions, there exist many different architectures with different evaluation intervals [a,b].

For reconfigurable datapaths, we are mainly concerned with high throughput architectures. We therefore limit ourselves to fully-pipelined FPGA implementations of function evaluation units. In the following, we describe three architectures for evaluating a given function based on lookup tables.

2.1 Three Lookup-Table based Units

The first architecture, a full-lookup unit, consists of a single lookup table. While a full-lookup table is straightforward to implement, its size and latency grows very rapidly with the required precision or range.

The second architecture, a lookup-add unit (Figure 1), is based on bipartite tables involving an addition of the results of two parallel lookups. The use of symmetric tables further reduces the required memory while improving the error bound [12]. However,

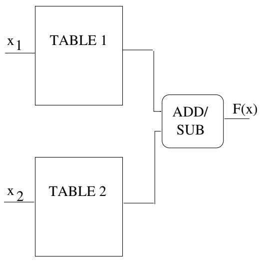

Fig. 1. Bipartite (lookup-add) tables computing the function $F(x)$. x_1 and x_2 are substrings of the original binary input x.

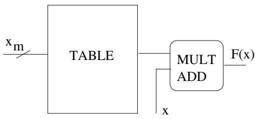

Fig. 2. A lookup-multiply unit evaluating $F(x)$. x_m are m bits of the binary input x.

in contrast to the full lookup, we now have to find the precision required for each of the tables and the precision for the final addition. Current state-of-the-art uses simulation to find the appropriate internal bitwidths.

The third, and most promising architecture is a lookup-multiply unit (Figure 2) based on affine polynomial approximation of a differentiable function. The coefficients for a polynomial approximation can be computed to minimize the average error of the approximation over the desired interval. The m most-significant bits of the input x are used to lookup the intermediate value in the lookup table, and the product of this value and x is then produced.

2.2 Shift-and-Add based CORDIC units

CORDIC units are one of a family of shift-and-add based function evaluation methods. The basic CORDIC unit computes up to two functions simultaneously using only constant shifts and additions. Details on the CORDIC architecture, it's advantages and limitations can be found in the literature (e.g. [2], [9], [14]). In summary, CORDICs converge approximately at around 1 bit per shift-add. Thus, a 10-bit CORDIC unit requires about 10 stages of shifts and adds. CORDICs lend themselves naturally to high throughput pipelining. One of the disadvantages is that CORDICs are limited to a relatively small set of elementary functions.

3 The Lookup-Multiply Method

3.1 Parameterizing the Lookup-Multiply Unit

This section contains an exact mathematical description of the relationships between the external and internal precisions for a lookup-multiply unit. The availability of such a description is the main advantage of the table-multiply method over bipartite (lookup-add) methods and shift-and-add based CORDICs.

To construct a module generator for a lookup-multiply unit, we need to compute all internal parameters (precisions) given a particular function, and the required precision/range of the output, denoted by the width l of the result. Internal parameters consist of the width and height of the lookup tables and the number of bits for the multiplication and addition.

Assume we require a unit to compute a function $F(x)$. We first compute a continuous piecewise affine approximation of the function $F(x)$, as shown in Figure 3. This

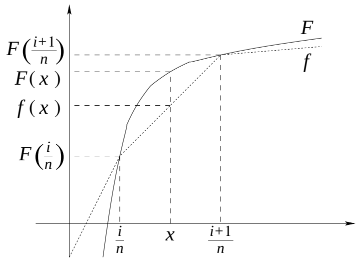

Fig. 3. $f(x)$ is the iecewise linear approximation of $F(x)$.

approximating function is called f. We compute the function F on an interval $[a, b)$. Then, we cut this interval into $n = 2^m$ intervals $[i/n, (i+1)/n)$ ($i \in [0, n]$). The value, m, is also the number of most-significant bits that are used to lookup the intermediate value in the lookup table. All the values are computed with k bits, and the result is rounded to l bits ($l < k$). $\widetilde{f}$ is the function in hardware which approximates f, and it may contain rounding errors.

To provide a method for estimating the bitwidths required for a given precision, we have to deal with two sources of error: approximation error and rounding error.

$$\mathtt{Error}_{\mathtt{total}} = \mathtt{Error}_{\mathtt{approx}} + \mathtt{Error}_{\mathtt{round}} \tag{1}$$

The first error $E_{\mathtt{approx}}$ comes from the approximation of the function f:

$$\mathtt{Error}_{\mathtt{approx}} = |\mathtt{F(x)} - \mathtt{f(x)}| \leq (2^{-(\mathtt{2m+3})}) \cdot \max_{\mathtt{z} \in [0,1]} (|\mathtt{F}''(\mathtt{z})|) \tag{2}$$

The approximation error $\mathtt{Error}_{\mathtt{approx}}$ is a function of the number of intervals stored in the lookup table for f and the maximum deviation of $F(x)$ from its linearization $f(x)$. This maximal deviation is controlled by the second derivative $F''(x)$.

The second error $\mathtt{Error}_{\mathtt{round}}$ comes from rounding errors inside the evaluation unit:

$$\mathtt{Error}_{\mathtt{round}} = \left|\mathtt{f(x)} - \widetilde{\mathtt{f}}(\mathtt{x})\right| < 0.75 \cdot 2^{-(\mathtt{k}-1)} + 2^{-(\mathtt{l}+1)} \tag{3}$$

In this case, $\mathtt{Error}_{\mathtt{round}}$ is a function of the internal precision k and the final precision l. Given a piecewise differentiable function $F(x)$ and a target precision l, we can easily find values for m and k to guarantee the precision of the final result up to one unit in the last place.

More details on the derivation of the equations above can be found in a report available from the authors [3]. Similar error analysis techniques can be found elsewhere [13]. Using these equations, lookup table area can be minimized with respect to valid values of m and k for a target precision l.

For F to be approximated to l bit range and domain accuracy, ($l < k$) must hold. Under this condition there is no benefit in replacing the lookup table with a bipartite table, as its minimal area is bounded by the target function output width, k. A full lookup table is the most efficient approximator for functions with a small number of input bits since the full-lookup grows with $2^{nb.of input bits}$. Our experience shows that there is no benefit in cascading bipartite and lookup multiply function approximation methods.

3.2 Resources and Performance of Lookup-Multiply Units

The two equations (2,3) shown above, provide a way of determining bitwidths for the multiply-lookup unit to meet accuracy requirements. This section suggests a generic layout for the hardware implementation, in order to develop parametric estimates of the required resources and the achievable performance.

We start by deriving an estimate for the resources. $\widetilde{f}(x)$ is encoded with l bits, whereas all the other numbers are encoded with k fractional bits. Let p be the number of bits of the input. Generally, we have $p > m$ and $p \simeq l$.

The goal is to place the unit into a rectangle. Our design contains two lookup tables, so that the two lookups for $F((i+1)/n)$ and $F(i/n)$ (see Figure 2) can be carried out in parallel. One lookup table (LUT1) contains 2^m numbers of $k+1$ bits, while the other lookup table (LUT2) has 2^m numbers of k bits. There are also $(k+1)$-bit by $(p-m)$-bit multiplier, one $(k+1)$-bit adder, and an l-bit rounding unit.

If $m \geq 4$, then the two lookup tables fit into rectangles:

$$\mathtt{size}_{\mathtt{LUT1}} = \frac{2^{\mathtt{m}-3} + 1 - 2 \cdot (\mathtt{m} \bmod 2)}{3} \times (\mathtt{k}+1) \tag{4}$$

and

$$\mathtt{size}_{\mathtt{LUT2}} = \frac{2^{\mathtt{m}-3} + 1 - 2 \cdot (\mathtt{m} \bmod 2)}{3} \times \mathtt{k} \tag{5}$$

If $m < 4$, the rectangles are $\mathtt{size}_{\mathtt{LUT1}} = 1 \times (\mathtt{k}+1)$ and $\mathtt{size}_{\mathtt{LUT2}} = 1 \times \mathtt{k}$.

The multiplier fits into a $(p-m) \times (k+1)$ rectangle, the adder in a $1 \times (k+1)$ rectangle, and the rounding unit in a $1 \times l$ rectangle. Hence the whole lookup-multiply design fits into a rectangle

$$\left(2 \cdot \mathtt{size}_{\mathtt{total}} = \frac{2^{\mathtt{m}-3} + 1 - 2 \cdot (\mathtt{m} \bmod 2)}{3} + (\mathtt{p}-\mathtt{m}) + 2\right) \times (k+1)$$

as shown in Figure 4, assuming that $m \geq 4$.

We now consider the performance of our design. The delay of both lookup tables is $1 + \max(0, \lceil (m-5)/6 \rceil)$ cycles. Both lookups can be performed in parallel. The delay of the multiplier is $1 + \lfloor (p - m + \lceil (8(k-2))/35 \rceil / 5 \rfloor$ cycles. The delay of the adder is 1 cycle; as is the delay of the rounding unit. Hence, the total delay of the fully pipelined approximator is:

$$\mathtt{delay}_{\mathtt{total}} = \max\left(0, \left\lceil \frac{\mathtt{m}-5}{6} \right\rceil\right) + \left\lfloor \frac{\mathtt{p} - \mathtt{m} + \lceil 8(\mathtt{k}-2)/35 \rceil}{5} \right\rfloor + 4$$

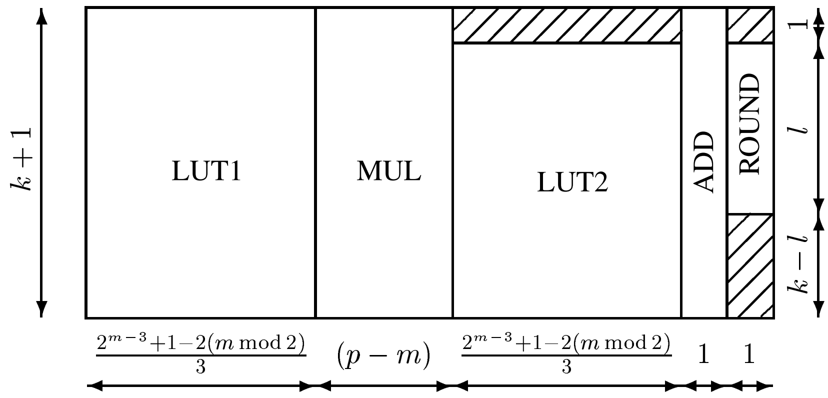

Fig. 4. Proposed layout of the lookup-multiply unit.

3.3 Example: $\ln(x)$

Consider the hardware unit for computing the natural logarithm which is a representative elementary function. We compute $\ln(x)$ in the interval $[1, 2)$: $F(x) = \ln(1 + x)$. This function fulfills all the requirements outlined above. We know $F'(x) = 1/(1 + x)$ and $F''(x) = -1/(1 + x)^2$. Hence for equation 2 with $F(x) = \ln(x)$:

$$\max_{x \in [0,1]} |F''(x)| = 1$$

.

Assuming that the input and the output are both q bits wide, we have $l = q$ and $p = q - 1$. We now have to choose m and k.

Because of the term 2^m in the space needed by the design, we choose the smallest possible m. Thus, we choose $m = \lfloor l/2 \rfloor$ so that

$$2^{-(2m+3)} \cdot \max_{x \in [0,1]} |F''(x)| < 2^{-(l+1)}$$

Consider the case that l is even, such that $m = l/2$. We choose $k = l + 2$ such that

$$\left(1 - \frac{1}{4}\right) 2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+3)}$$

If l is odd, then $m = (l - 1)/2$. We choose $k = l + 3$ such that

$$\left(1 - \frac{1}{4}\right) 2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+2)}$$

Moreover, since the logarithm is an increasing function, we can adopt a rounding method so that all the values involved in the computation are positive. This means that there is no need to store or compute sign bits.

Hence the design for a lookup-multiply $\ln(x)$ unit fits into a rectangle:

$$\mathtt{size}_{\mathtt{ln(x)}} = \left(2 \cdot \frac{2^{\mathtt{m}-3} + 1 - 2(\mathtt{m} \bmod 2)}{3} + (\mathtt{p} - \mathtt{m}) + 2\right) \times \mathtt{k}$$

4 Implementing the Module Generators in PAM-Blox

The function evaluation units are implemented as classes in C++ within PAM-Blox[8]. The input parameters set the precision of the input and output values. Inheritance is used throughout the implementation to optimize code efficiency. The top class evaluates polynomial functions. The first subclass inherits all the functionality of the top class and specializes the evaluation to, for instance, piecewise polynomial functions. The subclass of the piecewise polynomial evaluation class evaluates a given function using the lookup-multiply unit. Hence, we can reuse some parts of the code to build, for example, high degree polynomial approximations in the future.

The example in Figure 5 shows how to utilize the lookup-multiply class to build a hardware unit for the logarithm function. The code example shows the `Lookup_Multiply_Log` class, which generates $\ln(x)$ evaluation units that can guarantee a particular output precision given input and output precisions. The value of `INTERNAL_WIDTH`, which corresponds to the precision inside the unit, follows from equation 2 and equation 3.

```
// hardware unit for the logarithm function

const int INTERNAL_WIDTH = OUTPUT_WIDTH+2+(OUTPUT_WIDTH%2);

template<int INPUT_WIDTH, int OUTPUT_WIDTH>
class Lookup_Multiply_Log:
  public Lookup_Multiply<INPUT_WIDTH,
                         OUTPUT_WIDTH,
                         INTERNAL_WIDTH,
                         OUTPUT_WIDTH/2> {
  public:

  // constructor
  Lookup_Multiply_Log(WireVector<Bool, INPUT_WIDTH> &input_in,
                              Bool *clock=NULL,
                              const char *name=NULL):
         Lookup_Multiply<INPUT_WIDTH,
                          OUTPUT_WIDTH,
                          INTERNAL_WIDTH,
                          OUTPUT_WIDTH/2>(input_in, clock, name){}


  double Eval_Function(double x){
    return log(1.0+x);
  }

};
```

Fig. 5. The lookup-multiply method described in PAM-Blox.

5 Results

This section compares the results of implementing a given function using the full-lookup method, the lookup-add method, the lookup-multiply method, and a shift-and-add based CORDIC design.

In a full-lookup unit, the precision/range (l) determines width (l) and height (2^l) of the lookup table. The procedure for implementing lookup-add units can be found elsewhere [12], while that for implementing lookup-multiply units has been outlined in Section 3.1.

In contrast, parallel, pipelined CORDIC designs require two parameters: the number of bits per iteration stage, and the number of stages. Determining the minimal precision of each stage inside CORDIC units is complex. We can, however, give an upper bound on the required bits of precision inside the CORDIC unit: $(l + \ln(\textit{\# of shift-and-add stages}))$. The total number of required stages depends on the approximated function, the internal scaling method [10] [1], the precision of each stage, and of course the precision required at the output. In general, CORDIC units converge at the approximate rate of one bit per stage, which we use for our estimates. A more precise determination of the needed number of stages requires extensive simulations for each function, internal precision, scaling method, and for each output precision.

Figure 6 shows the results for the area of a function evaluation unit, in our case $F(x) = \ln(x)$ for Xilinx XC4000 FPGAs. Xilinx XC4000 FPGAs consist of a 2D array of Configurable Logic Blocks (CLBs). Each CLB contains two 4-input lookup tables and two flip-flops. In lookup-based function evaluation, the proportion of CLBs used as lookup table ROM grows as precision is increased. If implemented on a modern FPGA architecture such as Xilinx Virtex devices, CLB count could be significantly reduced by storing lookup table entries in available block RAMs.

Table 1 shows the latency of the fully-pipelined units in number of clock cycles. We assume that the cycle times for the different units are similar, because they are all fully-pipelined, possibly including a carry chain, between any two registers.

data width	full-lookup	lookup-add	lookup-multiply	CORDIC
4	1	2	4	5
8	2	2	5	9
12	2	3	6	13
16	3	3	7	17
20	4	4	7	21
24	4	4	9	25

Table 1. Speed comparison, in number of clock cycles, for designs with varying data width.

To summarize, for data width up to around 10 bits, a full-lookup unit provides the

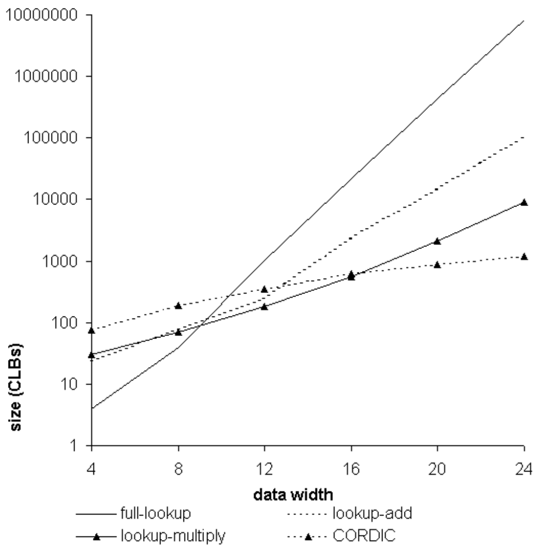

Fig. 6. Size comparison, in width $\times$ height $=$ CLBs, of designs with varying data width. Note that values for bipartite tables are estimated based on computational resources per CLB.

best trade-off in size and performance. For data width between 10 and 12 bits, the lookup-add design is appropriate. For data width between 12 and 20 bits, the lookup-multiply design should be considered. For data width more than 20 bits, CORDIC provides efficient solutions within its limitations.

The area results show that for data widths up to about 20 bits, a lookup-multiply strategy results in similar or smaller area than a shift-and-add CORDIC unit. However, while the lookup-multiply unit can be automatically designed to compute any differentiable function, the CORDIC unit is limited to a small set of elementary functions and has a less regular architecture across all possible function evaluations than a lookup-multiply unit.

6 Conclusion

The paper presents an approach to parameterize pipelined designs for differentiable function evaluation using lookup tables, adders, shifters and multipliers. This approach can be used to develop efficient implementations of function evaluators based on lookup tables, the size and performance of which can be estimated parametrically.

Our approach is implemented in C++ as part of the PAM-Blox module generation environment. We demonstrate that, depending on the data width, different lookup-based implementations for function evaluation should be used to improve efficiency. Examples confirm that the lookup-multiply approach produces competitive designs for data widths up to 20 bits when compared with shift-and-add based CORDIC units, without suffering from the limitations of CORDIC. Additionally, the table multiply method is applicable to larger data widths when evaluating functions not supported by CORDIC.

Current and future work includes assessing the effectiveness of the lookup-multiply approach for various differentiable functions, relating our tools to other pipeline synthesis techniques [15], retargeting our module generators to cover the latest FPGAs, and extending the development framework to support run-time reconfigurable designs.

Acknowledgements

We thank Mike Flynn and Martin Morf for support and encouragement during this work, and Lorenz Huelsbergen for discussions and helpful suggestions. The support of UK Engineering and Physical Sciences Research Council (Grant number GRN/66599), Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

1. H.M. Ahmed, *Signal Processing Algorithms and Architectures*, PhD Thesis, E.E. Department, Stanford University, June 1982.
2. R. Andraka, “A Survey of CORDIC Algorithms for FPGAs,” *Proc. ACM/SIGDA Int. symp. Field Programmable Gate Arrays*, ACM Press, pp. 191–200, 1998.
3. N. Boullis, *Designing Arithmetic Units for Adaptive Computing with PAM-Blox,* MIM Internship Report, ENS-Lyon, France, Sept. 2000.
4. C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky and S.G. Berg, “Mapping Applications to the RaPiD Configurable Architecture,” *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, pp. 106–115, 1997.
5. R. Laufer, R.R. Taylor and H. Schmit, “PCI-PipeRench and SWORD API: A System for Stream-based Reconfigurable Computing,” *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, pp. 200–208, 1999.
6. O. Mencer, *Rational Arithmetic Units in Computer Systems,* PhD Thesis (with M.J. Flynn), E.E. Dept., Stanford University, Jan. 2000.
7. O. Mencer, H. Huebert, M. Morf and M.J. Flynn, “StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox”, *Field-Programmable Logic and Applications*, LNCS 1896, Springer, pp. 595–604, 2000.
8. O. Mencer, M. Morf and M.J. Flynn, “PAM-Blox: High Performance FPGA Design for Adaptive Computing,” *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, pp. 167–174, 1998.
9. O. Mencer, L. Séméria, M. Morf and J.M. Delosme, “Application of Reconfigurable CORDIC Architectures,” *Journal of VLSI Signal Processing*, Vol. 24, No. 2–3, pp. 211-221, March 2000.
10. J.M. Muller, *Elementary Functions, Algorithms and Implementation,* Birkhaeuser, Boston, 1997.
11. S. Rixner et al., “A Bandwidth-Efficient Architecture for Media Processing,” *Proc. ACM/IEEE Int’l Symposium on Microarchitecture,* IEEE Computer Society Press, pp. 3–13, 1998.
12. M.J. Schulte and J.E. Stine, “Approximating Elementary Functions with Symmetric Bipartite Tables”, *IEEE Trans. Comput.*, Vol. 48, No. 8, pp. 842–847, August 1999.

13. P.T.P. Tang, "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic*, IEEE Press, pp. 232–236, 1991.
14. J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. on Electronic Computers*, Vol. EC-8, No. 3, Sept. 1959.
15. M. Weinhardt and W. Luk, "Pipeline Vectorization," *IEEE Trans. Comput. Aided Design*, Vol. 20, No. 2, pp. 234–248, February 2001.
16. W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Trans. Comput.*, Vol. 43, pp. 278-294, March 1994.