Journal of VLSI Signal Processing 36, 17–25, 2004

Parameterized High Throughput Function Evaluation for FPGAs

OSKAR MENCER
MAXELER Technologies, Florham Park, NJ 07932, USA

WAYNE LUK
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Received January 28, 2002; Revised September 19, 2002; Accepted October 15, 2002

Abstract. This paper presents parameterized module-generators for pipelined function evaluation using lookup tables, adders, shifters, multipliers, and dividers. We discuss trade-offs involved between (1) full-lookup tables, (2) bipartite (lookup-add) units, (3) lookup-multiply units, (4) shift-and-add based CORDIC units, and (5) rational approximation. Our treatment mainly focuses on explaining method (3), and briefly covers the background of the other methods. For lookup-multiply units, we provide equations for 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. A selection of the compared methods are implemented as part of the current PAM-Blox module generation environment. An example shows that the lookup-multiply unit produces competitive designs with data widths up to 20 bits when compared with shift-and-add based CORDIC units. Additionally, the lookup-multiply method or rational approximation can produce efficient designs for larger data widths when evaluating functions not supported by CORDIC.

Keywords: function approximation, FPGAs, lookup table, CORDIC, rational approximation

1. Introduction

The evaluation of differentiable functions can often be the performance bottleneck of compute-bound applications. Examples of these functions include elementary functions such as $\sin(x)$, $\log(x)$ or $\sqrt{x}$, and compound functions such as $(1 - \sin^2(x))^{1/2}$ or $\tan^2(x) + 1$.

An appealing way to overcome such performance bottlenecks is to use a reconfigurable computing system, which contains Field-Programmable Gate Arrays (FPGAs) to accelerate an application running on the microprocessor. In such systems, a function can be evaluated either in software on the processor, or in hardware on the FPGA. However, prior work [1] on computing elementary functions is focused on Application Specific Integrated Circuits (ASICs). 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 and compares parameterizable designs for evaluating differentiable functions using lookup tables, adders, multipliers, and dividers, which can be implemented very efficiently using FPGAs and other stream-based architectures [2–5]. We describe architectures based on full-lookup units, lookup-add units [6], lookup-multiply units [7], CORDIC units [8], and rational approximations, and compare their size and performance on FPGAs for different input data widths. We consider FPGA implementations which offer a high degree of parallelism and pipelining while allowing the user to trade off area for reduced latency. For highly pipelinable applications, such parallelism provides an edge over

general-purpose microprocessors and can lead to significant speedups.

The function evaluators are implemented as module generators within the PAM-Blox environment [9], 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 for FPGAs. One of the challenges of automatic generation of parametrizable function evaluation units 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 which may contain temporary values that require different precision than inputs or outputs. These intermediate precisions have to be selected to optimize 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 [8, 10, 11]. 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 [12], Chapter 8), reduces the argument $\overline{x}$ to a small interval $[a, b]$, resulting in a new argument x; for instance

$$x = \bar{x} - kC \quad \text{(for trigonometric functions)} \quad (1)$$

$$x = \bar{x}/C^k \quad \text{(for the logarithm function)} \quad (2)$$

where C is a function specific constant and k is the appropriate integer to reduce the particular x to the range $[a, b]$. The second stage evaluates the function $F(x)$. The third stage extrapolates $F(\overline{x})$ from $F(x)$.

In this paper, we are mainly concerned with the second stage: evaluating a function $F(x)$ where x is in a small, natural evaluation interval $[a, b]$. For example, for $\sin(x)$ the evaluation interval could be $[0, \pi/2]$. For any given function, area restrictions, and timing restrictions, there exist many different architectures with different evaluation intervals $[a, b]$. Given a specific function and interval it is possible to pick the architecture which best meets the area and timing requirements of the application.

For reconfigurable datapaths, we are concerned with high throughput architectures. We therefore limit the scope of this paper to bit-parallel and pipelined FPGA implementations of function evaluation units. In the following, we describe three architectures for evaluating a given function based on lookup tables, the CORDIC method for evaluating functions using only shift and add primitives, and rational approximation.

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 (Fig. 1), is based on bipartite tables involving an addition of the results of two parallel lookups. The use of addition in conjunction with symmetric tables further reduces the required memory for the lookup tables while improving the error bounds. However, in contrast to the full lookup method, we now have to find the precision required for each of the tables and the precision for the final addition. Current techniques use simulation to find the appropriate internal bitwidths. Recently a version of this method, based on multi-partite tables [19, Muller (2001)], has been implemented in the JBits FPGA development tool.

The third architecture, which is the most promising, is a lookup-multiply unit (Fig. 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

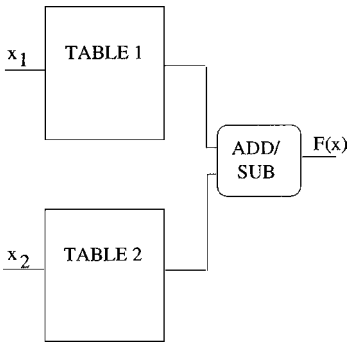

Figure 1. Bipartite (lookup-add) tables computing the function $F(x)$. x_1 and x_2 are substrings of the original binary input x.

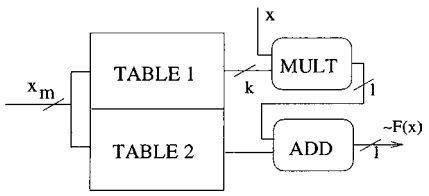

Figure 2. A lookup-multiply unit evaluating $F(x)$. x_m are m bits of the binary input x.

approximation over the desired interval. The m most-significant bits of the input x are used to lookup the intermediate value stored in a lookup table, whose output feeds a multiply-add unit.

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 of the CORDIC architecture can be found in the literature [8, 10, 11].

The following three equations are the core of a coordinate rotation unit using the CORDIC algorithm.

$$x_i = x_{i-1} - sign(z_{i-1}) \cdot y_{i-1} \cdot 2^{M_i} \quad (3)$$

$$y_i = y_{i-1} + sign(z_{i-1}) \cdot x_{i-1} \cdot 2^{M_i} \quad (4)$$

$$z_i = z_{i-1} - sign(z_{i-1}) \cdot f^{-1}(2^{M_i}) \quad (5)$$

Each of the iterations (shift-add) adds approximately one bit of precision to the final results. Thus, a 10-bit CORDIC unit requires about 10 stages of shifts and adds. Unfortunately it is only approximately one bit per iteration. The actual convergence varies with the function, stage, and precision.

CORDICs lend themselves naturally to high throughput pipelining. One of the disadvantages, however, is that CORDICs are limited to a relatively small set of elementary functions. Another disadvantage is that due to the nature of the CORDIC computations, it is difficult to compute the necessary precision requirements for each of the parameter combinations on the fly. Figure 3 show the general structure of a fully parallel and pipelined CORDIC unit and its FPGA layout.

2.3. Rational Approximation

Rational approximation offers efficient evaluation of analytic functions represented by the ratio of two polynomials.

$$f(x) \sim \frac{((a_n x + a_{n-1})x + \cdots a_1)x + a_0}{((b_m x + b_{m-1})x + \cdots b_1)x + b_0} = \frac{c_n x^n + c_{n-1}x^{n-1} \cdots c_1 x + c_0}{d_m x^m + d_{m-1}x^{m-1} \cdots d_1 x + d_0} \quad (6)$$

In general, rational approximations are the most efficient method to evaluate functions on a microprocessor. Typical polynomial sizes for floating-point precision are smaller than ten coefficients. Hardware implementations of rational approximation are studied in [14].

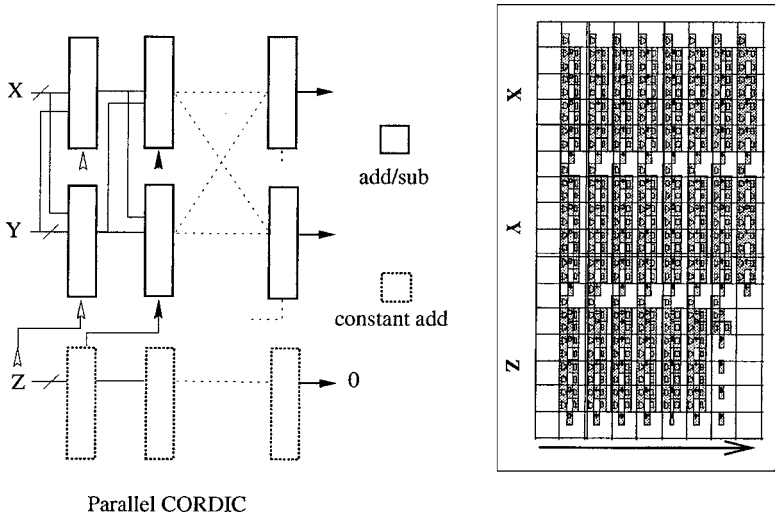

Figure 3. Block diagram and FPGA implementation of a fully parallel and pipelined CORDIC unit.

The results section of this paper shows how an FPGA implementation of rational approximation scales compared to the other methods.

3. Implementing a Lookup-Multiply Generator

3.1. Parameterizing the Lookup-Multiply Unit

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

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.

Given that 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 Fig. 4. Let us call this approximating function f. We approximate the function F on an interval $[a, b)$. In order to get the approximation for F, we cut this interval into $n = 2^m$ sub-intervals $[i/n, (i+1)/n)$, where $i \in [0, n]$. The integer 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 ($l < k$). $\tilde{f}$ 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.

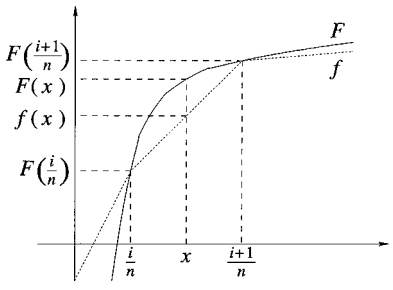

Figure 4. $f(x)$ is the piecewise linear approximation of $F(x)$.

$$\mathtt{Error}_{\mathtt{total}} = \mathtt{Error}_{\mathtt{approx}} \pm \mathtt{Error}_{\mathtt{round}} \quad (7)$$

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

$$\mathtt{Error}_{\mathtt{approx}} = |F(x) - f(x)| \leq \left(2^{-(2m+3)}\right) \cdot \max_{z\in[0,1]}(|F''(z)|) \quad (8)$$

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:

$$\mathtt{Error}_{\mathtt{round}} = |f(x) - \tilde{f}(x)| < 0.75 \cdot 2^{-(k-1)} + 2^{-(l+1)} \quad (9)$$

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 [7]. Similar error analysis techniques can be found elsewhere [15]. 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, the condition ($l < k$) 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 (B) since the full lookup table grows with 2^B. 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

Equations (8) and (9) shown above provide a way of determining bitwidths for the multiply-lookup unit while

meeting 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. The estimates are based on Xilinx XC4000 FPGAs.

Xilinx XC4000 FPGAs consist of a two-dimensional matrix of Configurable Logic Blocks (CLBs). Each CLB contains two 4-input lookup tables (4-LUTs) and two flip-flops. These 4-input lookup tables and flip-flops form the basic building blocks which can be used to design a sequential circuit on the FPGA. In order to achieve maximal throughput, we utilize as many of the flip-flops as possible to implement pipelining for every 4-input lookup table, generating fully pipelined designs.

We start by deriving an estimate for the resources. $\tilde{f}(x)$ 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 which can then be composed easily with other hardware components. Our design contains two lookup tables, so that the two lookups for $F((i+1)/n)$ and $F(i/n)$ can be carried out in parallel (Fig. 2). One lookup table (LUT1) contains 2^m numbers of $k+1$ bits each, while the other lookup table (LUT2) has 2^m numbers of k bits. There are also one $(k+1)$-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 of CLBs:

$$\texttt{size}_{\texttt{LUT1}} = \frac{2^{m-3} + 1 - 2 \cdot (m \bmod 2)}{3} \times (k+1) \tag{10}$$

and

$$\texttt{size}_{\texttt{LUT2}} = \frac{2^{m-3} + 1 - 2 \cdot (m \bmod 2)}{3} \times k \tag{11}$$

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

$$\texttt{size}_{\texttt{total}} = \left(2 \cdot \frac{2^{m-3} + 1 - 2 \cdot (m \bmod 2)}{3} + (p-m) + 2\right) \times (k+1) \tag{12}$$

as shown in Fig. 5, assuming that $m \geq 4$.

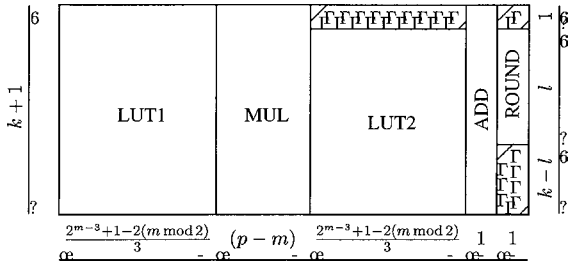

Figure 5. An example of a possible layout for the lookup-multiply unit.

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:

$$\texttt{delay}_{\texttt{total}} = \max\left(0, \left\lceil \frac{m-5}{6} \right\rceil\right) + \left\lfloor \frac{p - m + \lceil 8(k-2)/35 \rceil}{5} \right\rfloor + 4 \tag{13}$$

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 Eq. (8) with $F(x) = \ln(x)$:

$$\max_{x \in [0,1]} |F''(x)| = 1. \tag{14}$$

Assuming that the input and the output are both q bits wide, we have the target precision $l = q$ and $p = q - 1$. We now have to choose height (2^m) and width (k) of the lookup table.

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

$$2^{-(2m+3)} \cdot \max_{x \in [0,1]} |F''(x)| < 2^{-(l+1)} \tag{15}$$

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

$$\left(1 - \frac{1}{4}\right)2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+3)} \qquad (16)$$

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

$$\left(1 - \frac{1}{4}\right)2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+2)} \qquad (17)$$

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 containing the following number of CLBs:

$$\texttt{size}_{\texttt{ln(x)}} = \left(2 \cdot \frac{2^{m-3} + 1 - 2(m \bmod 2)}{3} + (p - m) + 2\right) \times k \qquad (18)$$

4. Implementing the Module Generators in PAM-Blox

The function evaluation units are implemented as classes in C++ within PAM-Blox [9]. 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 Fig. 6 shows how to utilize the lookup-multiply class to build a hardware unit for the logarithm function. The code example shows the original `Lookup_MultiplyLog` class, which generates $\ln(x)$ evaluation units that can guarantee a particular output precision given input and output precisions. The value of `INTERNALWIDTH`, which corresponds to the precision inside the unit, follows from Eqs. (8) and (9). A more current version of the module generation environment, PAM-Blox II, does not require any

```
// 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);
  }

};
```

Figure 6. The lookup-multiply method described as a PAM-Blox module generator. For details on how this code generates a hardware unit, see [9].

template parameters (values in $< \cdots >$), but instead infers the bitwidths directly from the state of the input Nets (which are C++ objects); the details are beyond the scope of this paper.

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, the shift-and-add based CORDIC method, and the rational approximation method.

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 [6], 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(*# of shift-and-add stages*)). The total number of required stages depends on the approximated function, the internal scaling method [12, 16], 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, for each internal precision, for each scaling method, and for each desired output precision.

Rational approximations vary in area, throughput and latency depending on the particular implementation of multiplication and especially division. For high throughput, pipelined array division is most suitable, and the results below are therefore based on such units. The particular rational approximation used for the results below is modeled after a Unix C library implementation of the logarithm function.

Figure 7 shows the results for the area of a function evaluation unit, in our case $F(x) = \ln(x)$ for Xilinx XC4000 FPGAs, as described earlier. In lookup-based function evaluation, the proportion of CLBs used as lookup table ROMs grows as precision is increased. If implemented on more recent FPGA architectures such as Xilinx Virtex devices, the CLB count could be significantly reduced by storing lookup table entries in available block RAMs. In case of Virtex II devices, dedicated block multipliers further improve the efficiency of the lookup-multiply method.

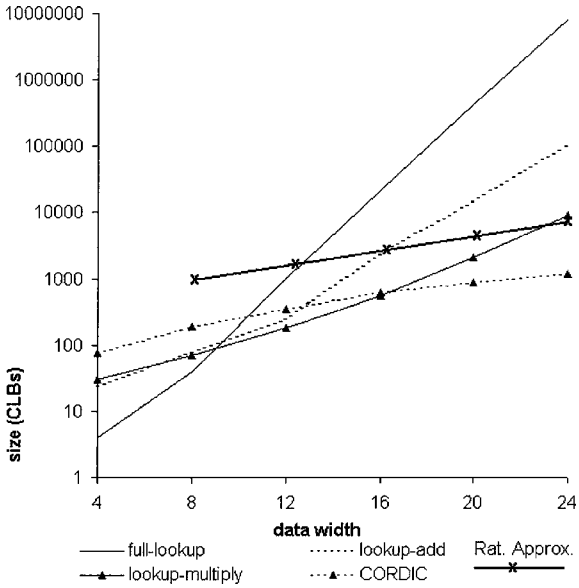

Figure 7. Size comparison, in number of CLBs (= width × height), of the function evaluation stage with varying data width. Note that values for bipartite tables (lookup-add) are estimated based on computational resources per CLB.

Table 1 shows the latency of the fully pipelined units in number of clock cycles. Latency is of interest to applications with a critical loop dependency that results in a bottleneck of the computation. 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.

To summarize our results, for data widths up to around 10 bits, a full-lookup unit provides the best trade-off in size and performance. For data widths between 10 and 12 bits, the lookup-add design is

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

Data width	Full-lookup	Lookup-add	Lookup-multiply	CORDIC	Rat. approx.
4	1	2	4	5	21
8	2	2	5	9	29
12	2	3	6	13	37
16	3	3	7	17	45
20	4	4	7	21	53
24	4	4	9	25	61

appropriate. For data widths between 12 and 20 bits, the lookup-multiply design should be considered. For data widths of more than 20 bits, CORDIC provides efficient solutions within its limitations, and rational approximations evaluate any function but incur a large latency penalty due to division.

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. Moreover, 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. Such regularity can be a key asset in simplifying the writing of general module generators, and can make the resulting designs more scalable.

6. Conclusion

We present an approach to parameterize pipelined designs for differentiable function evaluation using lookup tables, adders, shifters, multipliers, and dividers. This approach provides an efficient way to develop implementations of function evaluators for FPGAs based on lookup tables, the size and performance of which can be estimated parametrically.

Our approach is implemented in C++ as part of the current PAM-Blox module generation environment for Xilinx FPGAs. We demonstrate that, depending on the data width, different implementations of function evaluation should be used to maximize 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 lookup-multiply method is applicable to designs with larger data widths when evaluating functions not supported by CORDIC, and not suitable for rational approximation.

The lookup-multiply method and rational approximation can be further optimized on current state-of-the-art FPGAs such as the Xilinx Virtex II devices, which include dedicated fast multipliers and large chunks of block RAM on the FPGA chip.

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 [17], retargeting our module generators to cover the latest FPGAs, and extending the development framework to support run-time reconfigurable designs. We expect the results in this paper to hold also for other FPGA families with micro-architectures similar to the Xilinx 4000 devices.

Further optimization opportunities lie within the range reduction and extrapolation stages of function evaluation. Such optimization could be achieved by adapting the approximation intervals and the range reduction method to the particular requirements of the application.

Acknowledgments

Nicolas Boullis did most of the work addressing the lookup-multiply issues presented in the paper. Henry Styles helped in obtaining the presented results. We thank Mike Flynn and Martin Morf for support during the initial phase of this project, and Lorenz Huelsbergen for discussions. The support of Lucent, UK Engineering and Physical Sciences Research Council (Grant number GR/R55931 and GRN/66599), Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

1. W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Trans. Comput.*, vol. 43, 1994, pp. 278–294.
2. C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky, and S.G. Berg, "Mapping Applications to the RaPiD Configurable Architecture," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1997, pp. 106–115.
3. R. Laufer, R.R. Taylor, and H. Schmit, "PCI-PipeRench and SWORD API: A System for Stream-based Reconfigurable Computing," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1999, pp. 200–208.
4. 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, 2000, pp. 595–604.
5. S. Rixner et al., "A Bandwidth-Efficient Architecture for Media Processing," in *Proc. ACM/IEEE Int'l Symposium on Microarchitecture*, IEEE Computer Society Press, 1998, pp. 3–13.
6. M.J. Schulte and J.E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Comput.*, vol. 48, no. 8, 1999, pp. 842–847.
7. N. Boullis, *Designing Arithmetic Units for Adaptive Computing with PAM-Blox*, MIM Internship Report, ENS-Lyon, France, Sept. 2000.
8. J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. on Electronic Computers*, vol. EC-8, no. 3, 1959.
9. O. Mencer, M. Morf, and M.J. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," in *Proc. IEEE*

Symp. on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1998, pp. 167–174.
10. R. Andraka, "A Survey of CORDIC Algorithms for FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, ACM Press, 1998, pp. 191–200.
11. O. Mencer, L. Séméria, M. Morf, and J.M. Delosme, "Application of Reconfigurable CORDIC Architectures," *Journal of VLSI Signal Processing*, vol. 24, nos. 2/3, 2000, pp. 211–221.
12. J.M. Muller, *Elementary Functions, Algorithms and Implementation.* Birkhaeuser, Boston, 1997.
13. F. de Dinechin and A. Tisserand, *Some Improvements on Multipartite Table Methods*, Tech. Rep. RR-4059, INRIA, Nov. 2000.
14. I. Koren and O. Zinaty, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Trans. Comput.*, vol. 39, no. 8, 1990.
15. P.T.P. Tang, "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," in *Proc. 10th IEEE Symp. Computer Arithmetic*, IEEE Press, 1991, pp. 232–236.
16. H.M. Ahmed, *Signal Processing Algorithms and Architectures*, PhD Thesis, E.E. Department, Stanford University, June 1982.
17. M. Weinhardt and W. Luk, "Pipeline Vectorization," *IEEE Trans. Comput. Aided Design*, vol. 20, no. 2, 2001, pp. 234–248.
18. O. Mencer, *Rational Arithmetic Units in Computer Systems*. PhD Thesis (with M.J. Flynn), E.E. Dept., Stanford University, Jan. 2000.
19. F. de Dinechin and J. Detrey, *Multipartite Tables in JBits for the Evaluation of Functions on FPGAs*, Tech. Rep. RR-4305, INRIA, Nov. 2001.

Oskar Mencer founded MAXELER Technologies in 2003 after three years as member of Technical Staff in the Computing Sciences Research Center at Bell Labs. He received his PhD and M.S. in Electrical Engineering from Stanford University in 2000 and 1997 respectively, and a B.S. degree in Computer Engineering from the Technion/Israel in 1994.

His research interests span computer architecture, computer arithmetic, VLSI microarchitecture, VLSI CAD, and reconfigurable (custom) computing. More specifically, he is interested in exploring application specific representations of computation at the algorithm level, the architecture level, and the arithmetic level.
mencer@research.bell-labs.com

Wayne Luk is a member of academic staff in Department of Computing, Imperial College, University of London. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communication. His current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays. He received his M.A., M.Sc. and D. Phil. in engineering and computing science from University of Oxford.