Table-based polynomials for fast hardware function evaluation

Jérémie Detrey, Florent de Dinechin

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Table-based polynomials for fast hardware function evaluation

Jérémie Detrey and
Florent de Dinechin

November 2004

Research Report N° RR2004-52

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

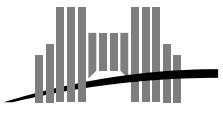

Table-based polynomials for fast hardware function evaluation

Jérémie Detrey and
Florent de Dinechin

November 2004

Abstract

Many general table-based methods for the evaluation in hardware of elementary functions have been published. The bipartite and multipartite methods implement a first-order approximation of the function using only table lookups and additions. Recently, a single-multiplier second-order method of similar inspiration has also been published. This paper presents a general framework extending such methods to approximations of arbitrary order, using adders, small multipliers, and very small ad-hoc powering units. We obtain implementations that are both smaller and faster than all previously published approaches.
This paper also deals with the FPGA implementation of such methods. Previous work have consistently shown that the more complex methods were also faster: The reduction of the table size meant a reduction of its lookup time, which compensated for the addition and multiplication time. A second contribution is therefore to finally create a tradeoff between space and time among table-based methods.

Keywords: Function evaluation, polynomial approximation, table-based method, hardware operators, FPGA

Résumé

De nombreuses méthodes générales à base de tables pour l'évaluation matérielle de fonctions élémentaires ont été publiées. Les mathodes bipartite et multipartite implémentent une approximation du premier ordre de la fonction en utilisant uniquement des accès à des tables et des additions. Récemment, une méthode du second ordre n'utilisant qu'un multiplieur, inspirée des précédentes, a aussi été publiée. Cet article présente un cadre général pour étendre de telles méthodes à des approximations d'ordres supérieurs, utilisant des additionneurs, de petits multiplieurs, et de très petites unités dédiées d'élévation à une puissance donnée. Nous obtenons des implémentations qui sont tant plus petites que plus rapides que toutes les méthodes précédemment publiées.
Cet article présente aussi l'implémentation de telles méthodes sur FPGA. Des travaux antérieurs ont montré à plusieurs reprises que des méthodes plus complexes sont aussi plus rapides : la réduction de la taille des tables entraîne une réduction du temps d'accès, qui contre-balance le coût des additions et des multiplications. Cette contribution vise donc à trouver où se situe le compromis entre surface et latence pour les méthodes à base de tables.

Mots-clés: Évaluation de fonctions, approximation polynomiale, méthode à base de tables, opérateurs matériels, FPGA

1 Introduction

Many applications require the evaluation in hardware of a numerical function: Trigonometric functions for DSP algorithms, inverse and inverse square root for providing seed values to the Newton-Raphson algorithms for division and square root [1], exponential and logarithm for some scientific computing applications or for the logarithm number system (LNS) [2], etc. When a compound function (such as $\log_2(1+2^x)$ for instance) is needed, it is often more efficient to implement it as one operator instead of a combination of successive operators for each function (here an exponential, an addition and a logarithm).

Specific methods exist for implementing most of the elementary functions. For example, the CORDIC algorithm and its derivatives implement trigonometric and exp/log functions. With some work, this is probably also true of most useful compound function (see for example the litterature about LNS arithmetic for methods dedicated to evaluating $\log_2(1+2^x)$). However these specific methods usually have their constraints. For instance, the CORDIC derivative leads to small but slow operators. Besides they may require a lot of non-reusable work to get a functional implementation.

An alternative is to use a general implementation method which may be tailored easily to any function. The simplest of these methods is, of course, to tabulate all the values that the function takes in the needed discrete range. The drawback is then the hardware cost, as the size of the table increases exponentially with the size (in bits) of its input argument.

Table-and-addition methods [3, 4, 5, 6] use a first-order Tailor approximation to the function. The product terms are themselves tabulated, leading to an architecture composed of table lookups and additions, and therefore very fast. Recently, we proposed a method allowing a second order approximation using only one small multiplier [7]. Here "small" means that its input and output sizes are much smaller than the input and output precision of the function to be evaluated. Both methods can be applied to any function, elementary or compound, that fullfills basic continuity requirements. This means that they lend themselves to the implementation of automatic operator generators: We have programs that take an arbitrary function with an input and output precision, and compute the optimal implementation of this function as a hardware operator, according to some predefined optimality criterion. The size and speed of the operator depends on the input and output precision, but also on the function. These generators output circuits descriptions in the VHDL language, suitable for synthesis.

These methods may target Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs). The metrics of ASICs and FPGAs are very different: Tables may be implemented in various ways in both world, the same holds for arithmetic operators, and the relative size and speed of arithmetic and tables are different. For practical reasons we mostly studied FPGA implementation. In both case, an operator generator will try and synthesize a few candidates to optimality before giving a definitive result.

With the FPGA metrics, an interesting result of previous work on table-based methods was that a multipartite implementation was a win-win situation compared to a simple table: Although the former has one table lookup and several additions on the critical path, it is faster than the latter which has only a table lookup. The reason is simply that the tables are so much reduced in the multipartite implementation that the lookups are much faster, which compensates the addition time. More surprising was the fact that going to second-order approximation and to architectures with multipliers on the critical path was again a win-win move [7]. This is a motivation to study higher-order methods in this paper.

This paper first presents in Section 2 a general framework for the hardware implementation of arbitrary polynomials. Polynomiers using the Horner evaluation order have been studied, but their iterative nature leads to implementations with long latency. The approach studied here is to use a developed form of the polynomial, where each monomial is evaluated in parallel. Each monomial may then be implemented by multipliers and powering units, or table-based methods, or a combination of both. The philosophy is here to carry out a careful error computation, not only to guarantee faithful correct rounding of the result, but also to build blocks which are never more accurate than strictly needed, as exposed in Section 3. The architectures obtained are depicted in Section 4, and their speed and area is studied in Section 5 and compared to results obtained using other methods.

2 Presentation of the method

2.1 Function evaluation

The problem of function evaluation can be expressed as follows: We are given a function f defined on a finite input interval $\mathcal{I} \subset \mathbb{R}$, along with two positive integers w_I and w_O which specify the length in bits of the input and output words respectively, and we want to build a hardware circuit which will compute an approximation $\tilde{f}$ of the function f on the interval $\mathcal{I}$.

Without loss of generality, we can take $\mathcal{I} = [0; 1[$ and scale f such as $f(\mathcal{I}) = [0; 1[$. Therefore, we will write the input word X as $X = .x_1 x_2 \dots x_{w_I}$, and similarly the output word $Y = \tilde{f}(X) = .y_1 y_2 \dots y_{w_O}$. We also want our evaluation operator to guarantee the accuracy of the result. As rounding to nearest is impossible because of the table-maker's dilemma [1], we choose to ensure faithful rounding:

$$\epsilon = \max_X |f(X) - \tilde{f}(X)| < 2^{-w_O}.$$

2.2 Piecewise polynomial approximation

The method we present here is based on a piecewise polynomial approximation: The input interval $\mathcal{I} = [0; 1[$ is regularly split in several sub-intervals $\mathcal{I}_i = [i \cdot 2^{-\alpha}; (i+1) \cdot 2^{-\alpha}[$. These sub-intervals are addressed by the α most significant bits of X, and we approximate f on each of them by a degree n polynomial P_i. Each polynomial P_i is computed using by a minimax scheme[1], and therefore minimizes the maximum error entailed by this approximation.

Note that the input interval partition that we perform here is uniform, as all the sub-intervals have the same size. Lee *et al.* have developed a method for hierarchical segmentation of this interval in [8] which could be worth trying to apply to our work.

As the sub-intervals are addressed by α bits from X, we can split the input word in two sub-words $A = .a_1 a_2 \dots a_\alpha$ and $B = .b_1 b_2 \dots b_\beta$ of length α and $\beta = w_I - \alpha$ respectively, such as shown in Fig. 1. This gives:

$$X = A + B \cdot 2^{-\alpha}.$$

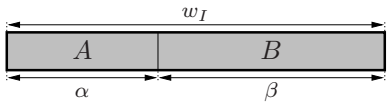

Figure 1: Splitting of the input word X into A and B.

Thus, to compute $\tilde{f}(X)$, we need to evaluate the polynomial $P_A(B \cdot 2^{-\alpha})$, which we will write $P(A, B \cdot 2^{-\alpha})$ to simplify the notations. We can then expand the polynomial to obtain:

$$\begin{aligned} P(A, B \cdot 2^{-\alpha}) = K_n(A) \cdot B^n \cdot 2^{-n\alpha} + \dots \\ + K_1(A) \cdot B \cdot 2^{-\alpha} + K_0(A). \end{aligned} \tag{1}$$

The main idea of the method presented in this article is to evaluate separately each of the terms (or monomials) of the form $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$ for k ranging from 0 to n. A final summation of all the terms then effectively computes the approximated function $\tilde{f}(X)$.

2.3 Computing the terms

There are several methods to evaluate a term $T_k(A, B)$, and we chose to implement two of them in our work, as described in the following paragraphs.

2.3.1 Simple ROM

The first and the simplest method is to extensively compute all the possible values for the term and tabulate these values in a table addressed by A and B, or only by A for $T_0(A, B) = K_0(A)$.

2.3.2 Power-and-multiply

This second method consists in first computing B^k by using a powering unit, and then multiplying the result by $K_k(A)$.

Yet several implementation choices remain. The powering unit can either be a simple table addressed by B where all the possible values of B^k are stored, or a specialized *ad-hoc* unit which first generates then adds all the partial products required to compute B^k (such as in [9] for $k = 2$).

Moreover, the product of B^k by $K_k(A)$ can be spread on several multipliers by splitting the word $B^k = .p_1p_2 \dots p_{k\beta}$, which is of length $k\beta$, in m_k sub-words $S_{k,j}$, as in the multipartite method [6]. Spreading the product will allow us to optimize separately each multiplier as detailed in Section 3.2.

We then obtain:

$$B^k = S_{k,1} + S_{k,2} \cdot 2^{-\rho_{k,2}} + \ldots + S_{k,m} \cdot 2^{-\rho_{k,m_k}},$$

where, for j ranging from 1 to m_k:

$$S_{k,j} = .p_{\rho_{k,j}+1} p_{\rho_{k,j}+2} \cdots p_{\rho_{k,j}+\sigma_{k,j}}$$

is the sub-word of B^k starting at bit $\rho_{k,j}$ and of length $\sigma_{k,j}$. As it is a partition, we also have the natural conditions on the $\rho_{k,j}$'s and the $\sigma_{k,j}$'s:

$$\begin{aligned} \rho_{k,1} &= 0, \\ \rho_{k,j+1} &= \rho_{k,j} + \sigma_{k,j}, \text{ for } 1 \le j < m_k, \\ \sum_{j=1}^{m_k} \sigma_{k,j} &= \rho_{k,m_k} + \sigma_{k,m_k} = k\beta. \end{aligned}$$

This partition is shown in Fig. 2.

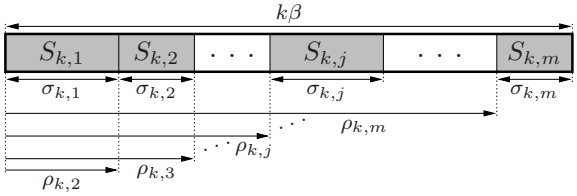

Figure 2: Splitting of the word B^k into $S_{k,1}$, $S_{k,2}$, ..., S_{k,m_k}.

We can therefore rewrite the k-th term as :

$$\begin{aligned} T_k(A,B) &= K_k(A) \cdot B^k \cdot 2^{-k\alpha} \\ &= \left(\sum_{j=1}^{m_k} K_k(A) \cdot S_{k,j} \cdot 2^{-\rho_{k,j}} \right) \cdot 2^{-k\alpha}. \end{aligned} \tag{2}$$

Finally, another choice raised by this method is, for each product $Q_{k,j}(A, S_{k,j}) = K_k(A) \cdot S_{k,j}$, whether to use a table addressed by A and a multiplier, or a single but larger table addressed by A and S_j. We will consider that the m_k^M first products will be implemented with multipliers, whereas the $m_k^T = m_k - m_k^M$ last ones will have their values tabulated.

2.4 Exploiting symmetry

By a simple change of variable in Eq. 1, we can easily obtain a new expression for the polynomial approximation $P(A, B \cdot 2^{-\alpha})$, such that all the terms are symmetric with respect to the middle of the sub-interval $\mathcal{I}(A)$:

$$\begin{aligned} P(A, B \cdot 2^{-\alpha}) = {} & K'_n(A) \cdot (B - \Delta)^n \cdot 2^{-n\alpha} + \ldots \\ & + K'_1(A) \cdot (B - \Delta) \cdot 2^{-\alpha} + K'_0(A), \end{aligned}$$

where $\Delta = \frac{1}{2}(1 - 2^{-\beta})$.

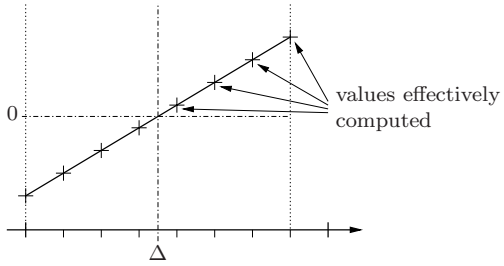

Figure 3: Example of segment symmetry.

This transformation allows us to use a trick from Schulte and Stine [4] to compute the terms only on one half of the sub-interval and deduce the values for the other half by symmetry at the expense of a few XOR gates, as depicted in Fig. 3.

Remark : To avoid overloading too much the notations, we will continue to write the terms $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$, even when symmetry is implied.

3 Decreasing accuracy

In this method so far, the only error is entailed by the initial polynomial approximation of the function. However, we can see from Eq. 1 that, because of the different power-of-two factors, the terms do not have the same weight in the final addition and thus some of them are computed with too much accuracy when compared to others.

In order to simplify the tables, and consequently gain in area and latency for our operator, we can therefore decrease the accuracy of those terms that are relatively too accurate.

3.1 Terms as simple ROMs

When considering a term $T_k(A, B)$ implemented as a table addressed by A and B, the idea is to decrease the size of the address word:

- Decreasing the size of A by using only its α_k most significant bits to address the table means that we will use a same value of the coefficient K_k for $2^{\alpha-\alpha_k}$ consecutive intervals.
- Decreasing the size of B by using only its β_k most significant bits to address the table means that less values will be computed for each interval.

We can therefore refine the splitting of the input word as in Fig. 4 and use only $A_k = .a_1a_2 \dots a_{\alpha_k}$ and $B_k = .b_1b_2 \dots b_{\beta_k}$ to address the table of term $T_k(A_k, B_k)$.

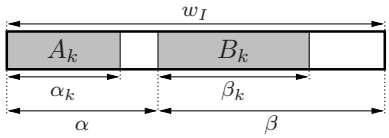

Figure 4: Splitting of the input word X into A_k and B_k.

3.2 Terms as power-and-multiply units

In this case, the first idea is also to decrease the size of A and B. But here, as the product is spread over m_k multiplications, we have m_k^M tables addressed by A, and m_k^T tables addressed by A and $S_{k,j}$. Once again, according to Eq. 2, those tables have different relative accuracies due to the $2^{-\rho_{k,j}}$ factors. We can therefore address them with sub-words of A of different sizes: The table used by $Q_{k,j}$ will be addressed with only the $\alpha_{k,j}$ most significant bits of A.

Yet, from Eq. 2 we can see that the relative weight of $Q_{k,j}$ decreases as j increases. This gives the following constraint on the $\alpha_{k,j}$'s:

$$\forall j, j' \in [1; m], \text{ if } j < j' \text{ then } \alpha_{k,j} \geq \alpha_{k,j'}.$$

Moreover, we can also use $B_k = .b_1 b_2 \dots b_{\beta_k}$ instead of B, as shown Fig. 5. Thus, the length of B_k^k will be only $k\beta_k$, which also implies smaller $S_{k,j}$'s. In fact, we can be even more general and suppose that the powering unit will generate only the λ_k most significant bits of B_k^k (with $\lambda_k \leq k\beta_k$).

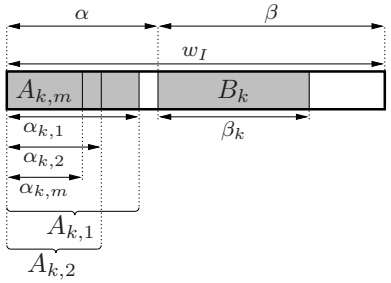

Figure 5: Splitting of the input word X into $A_{k,j}$'s and B_k.

This way, the $S_{k,j}$'s are much smaller, and consequently so are the product (both multiplier-based and table-based) units.

3.3 A few words about the *ad-hoc* powering units

If generating only λ_k bits of B_k^k is not a problem for the table-based powering units, for the *ad-hoc* powering units, on the other hand, will entail a larger error if only the partial products of weight greater than λ_k are computed then added.

To solve this problem without having the unit to compute all the $k\beta_k$ bits of B_k^k, we introduce another parameter μ_k which specifies the minimal weight of the partial products considered internally by the operator, before truncating the result to λ_k bits. This parameter is illustrated in Fig. 6, where the partial products are represented by bullets and sorted according to their weight.

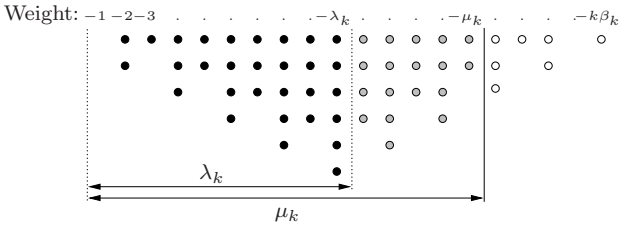

Figure 6: Use of the λ_k and μ_k parameters for *ad-hoc* powering units (here for $k = 2$ and $\beta_k = 10$).

4 Architecture

4.1 Overview

The overall architecture of the operators designed by the proposed method is given in Fig. 7. This architecture is quite straightforward, as it is directly derived from Eq. 1.

Still, a few points have to be detailed. First, it is obvious that the order 0 term T_0 does not depend on B, and therefore will be implemented as a simple ROM.

Concerning the term T_n of degree n, one can notice that the accuracy required for this term is very low, due to the $2^{-n\alpha}$ factor. We can then decrease α_n and β_n to only a few bits, and therefore implement also this term with a simple ROM. The same argument sometimes hold for lower order terms such as T_{n-1}.

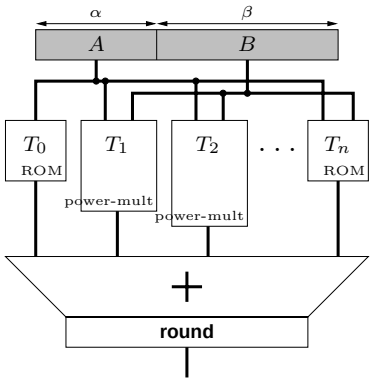

Figure 7: Overall architecture.

On the other hand, the other terms need to be computed with a larger accuracy, and will usually be implemented with slower but smaller power-and-multiply units.

4.2 Term as a simple ROM

The architecture for evaluating a term T_k using a simple ROM is also quite straightforward, as shown in Fig. 8.

The most significant bit b_1 of the input word B_k selects if B_k is in the first or the second half of the sub-interval $\mathcal{I}(A)$. A row of XOR gates is used to compute the symmetric of B_k if it falls on the wrong half. The table lookup is addressed by the α_k bits of A_k, and the $\beta_k - 1$ bits $B'_k = b'_2 b'_3 \dots b'_{\beta_k}$ from the XOR gates.

If k is odd, a last row of XOR gates controled by b_1 computes if necessary the opposite of the value given by the ROM. If k is even, we do not need these XOR gates.

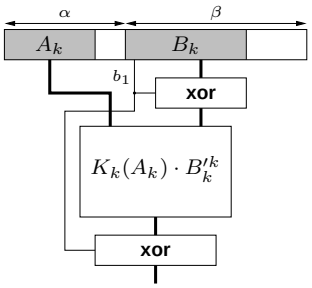

Figure 8: Architecture of the term T_k implemented as a simple table.

4.3 Term as a power-and-multiply unit

The architecture of a power-and-multiply unit is given in Fig. 9.

As for table-based terms, the most significant bit b_1 of B_k controls a row of XOR gates used to take the symmetric value of B_k if it is in the wrong half of the sub-interval. The resulting $\beta_k - 1$ bits $B'_k = b'_2 b'_3 \dots b'_{\beta_k}$ are then given to the powering unit, which outputs the λ_k most significant bits of B'^k_k. This word is split in m_k sub-words.

Each of these words $S_{k,j}$ is then multiplied by $K_k(A_{k,j})$, either using a normal multiplier or a lookup table. In both cases, we again exploit the symmetry of the product, and use some rows of XOR gates. It is to be noted that the last row of XORs is controled by both b_i and $p_{\rho_{k,j}}$ when k is odd.

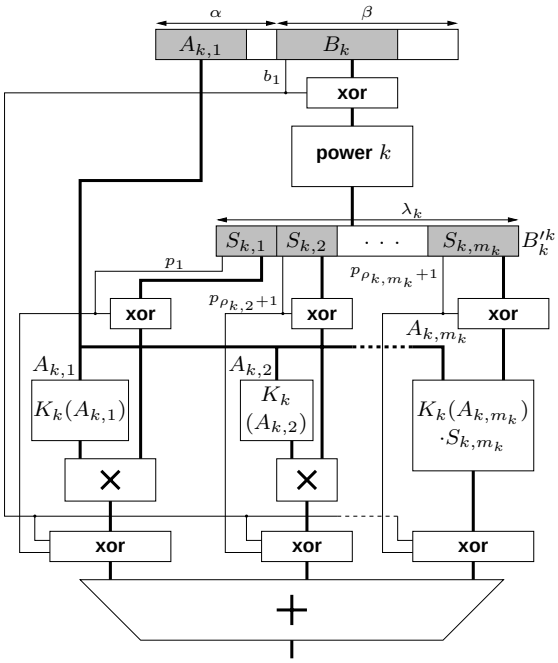

Figure 9: Architecture of the term T_k implemented as a power-and-multiply unit.

4.4 Table-based powering unit

The architecture of a table-based powering unit is perfectly straightforward, as it is only a lookup table, addressed by the $\beta_k - 1$ bits of B'_k, which contains the λ_k most significant bits of B'^k_k.

4.5 *Ad-hoc* powering unit

The architecture of an *ad-hoc* powering unit is also very simple, as shown in Fig. 10. The first part of the operator generates all the partial products that are required to compute the μ_k most significant bits of B'^k_k. Then, these partial products are added, and finally the result is truncated to λ_k bits.

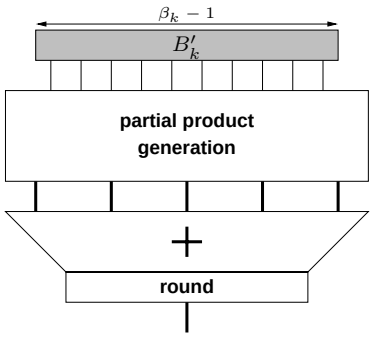

Figure 10: Architecture of an *ad-hoc* powering unit.

5 Error analysis

In this section we briefly describe how to keep track of all the errors entailed by the presented method, and therefore how to guarantee faithful rounding for the final result. In fact, this method can easily be adapted to any error bound $\epsilon_{\max} > 2^{-w_O-1}$, but in this paper we only consider the case of $\epsilon_{\max} = 2^{-w_O}$.

For simplicity's sake we do not detail all the equations involved in this error analysis, but we have extensively tested it.

5.1 Polynomial approximation: ϵ_{poly}

The polynomial approximation of the function on each sub-interval obviously yields an error bounded by ϵ_{poly}. The Remez algorithm [1] that we use to compute the minimax polynomials gives us the value of ϵ_{poly} for each sub-interval $\mathcal{I}(A)$.

5.2 Decreasing accuracy: ϵ_{method}

5.2.1 Reducing table input size (ϵ_{tab})

Reducing the number of bits used to address a table is in fact using a constant value for several entries of the table.

For instance, considering a term $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$ implemented as a ROM, decreasing the word length of A to α_k bits means that a same value of the coefficient K_k will be used for $2^{\alpha-\alpha_k}$ consecutive sub-intervals. To minimize the error, we can use for this value $K_k(A_k)$ the average of the extremal values of $K_k(A)$ for the sub-intervals. The aforementioned error is thus the half of the distance between those extremal values.

Similarly, reducing B to β_k bits means that a constant value of B^k will be used for $2^{\beta-\beta_k}$ successive values of B. Taking the average of the extremal values of B^k also yields the minimum error.
Remark: It is important to note that, though the symmetry trick allows us to use only $\beta_k - 1$ bits of B to address the table, it entails abolutely no additional error.

The same argument holds for all the tables used in our method. We can therefore quantify the total error entailed by these approximations, as each term implemented as a ROM, each table-based powering unit or multiplier and each coefficient table yields an error. The sum of these errors is noted ϵ_{tab}.

5.2.2 *Ad-hoc* powering units (ϵ_{pow})

Using only the β_k most significant bits of B when computing B^k produces a quantifiable error, as $B_k = B - .b_{\beta_k+1}b_{\beta_k+2}\dots b_\beta \cdot 2^{-\beta_k}$ and $0 \leq .b_{\beta_k+1}b_{\beta_k+2}\dots b_\beta < 1$. To center this error around 0, we add an implicit half-ulp (unit in the last place) to B_k before computing B_k^k.

Moreover, the error made when reducing the number of partial products taken into account in the computation of B_k^k can also be bounded in advance, as we already know the number and the weight of the partial products that are ignored. We can then compute the sum s of those partial products, as the error will be in the interval $[0; s]$. Adding $s/2$ to the sum of the partial products computed by the unit will center the error around 0 as much as possible.

The errors yielded by each *ad-hoc* powering unit can be suitably scaled and added to obtain the error term ϵ_{pow}.

We finally note $\epsilon_{\text{method}} = \epsilon_{\text{tab}} + \epsilon_{\text{pow}}$.

5.3 Rounding considerations: ϵ_{rt}, ϵ_{rf}

The tables cannot be filled with results rounded to the target precision w_O: Each table would entail a maximum error of 2^{-w_O-1}, exceeding the total error budget $\epsilon_max = 2^{-w_O}$. This argument also applies to multipliers, whose result cannot be rounded to the target precision. We therefore need to extend the internal precision of our operator by g guard bits.

The values stored in the tables will then be rounded to the precision w_O+g, thus yielding a maximum rounding error of 2^{-w_O-g-1}. Similarly, the result of the multipliers will be rounded to $w_O + g$ bits, by truncating and adding a half-ulp, to ensure here also a maximum error of 2^{-w_O-g-1}.

The sum of all these errors is noted ϵ_{rt}.

The final summation is also performed on w_O+g bits, and is then truncated to the target precision w_O. A trick by Das Sarma and Matula [3] allows us to bound this rounding error by $\epsilon_{\text{rf}} = 2^{-w_O-1}(1 - 2^{-g})$.

5.4 Putting it all together

Summing all the errors described previously, we have the following contraint to ensure faithful rounding:

$$\epsilon = \epsilon_{\text{poly}} + \epsilon_{\text{method}} + \epsilon_{\text{rt}} + \epsilon_{\text{rf}} < \epsilon_{\text{max}}$$

We can then expand the values of ϵ_{rt} and ϵ_{rf} to obtain an inequation that we can solve to find the smallest number of required guard bits g.

In fact, as we have added the maximum errors for each term, the total error may be overestimated, and a smaller g could be enough. We therefore apply a simple trial-and-error method to find the smallest acceptable g.

6 Results

This section presents synthesis results obtained for the presented method. We have successfully implemented order 2, 3 and 4 approximations for the function $\log_2(1+x)$, and we compare them for various precisions with the SMSO method in terms of estimated area and delay of the operators in Fig. 11 and 12. SMSO has already been proven to be always better than the best available multipartite methods [7] and than the order 2 method from [9]. We have only the logarithm function on these graphs because other functions give completely similar results.

Those estimations were obtained using the Xilinx design suite ISE 5.2 along with the Xilinx synthesizer XST, for a Virtex-II XC2V-1000-4 FPGA. However, we chose not to implement multipliers using the Virtex-II 18×18 multipliers, to allow a more accurate comparison with other works.

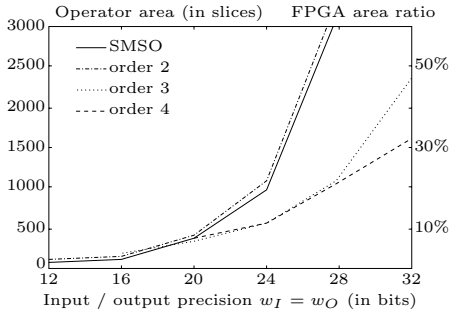

Figure 11: Operator area for the $\log_2(1+x)$ function.

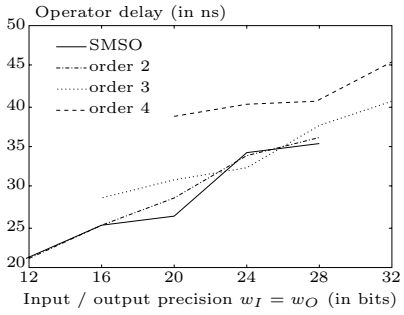

Figure 12: Operator delay for the $\log_2(1+x)$ function.

7 Conclusion and future work

This article presents a general method, implemented in a functional tool, to build fast (combinatorial) implementations of arbitrary functions. The method leads to faster and smaller implementations than

those previously published. As a rule of thumb, a second order approximation is optimal for precisions up to 16 bits and leads to operators which consume only a few percent of even the smallest FPGAs. For 24-bit precision, an order 3 approximation is optimal (order 4 is no smaller, but slower). For 32 bits, a precision out of reach of previous methods, we have a tradeoff between order 3 and order 4, one being 30% smaller, the other being faster. Besides, all these architectures are very regular and easy to pipeline.

The implementation space finally exhibits tradeoffs on FPGAs, where previous methods always lead to win-win situation (smaller area and smaller delay).

Future work

We now have to study the application of this method to ASIC synthesis, where the metrics are very different. Since the architectures involve the sum of many terms, the intermediate results should probably be expressed in carry-save notation, with only one fast adder in the architecture. Therefore, there is some work to do on the VHDL backend. We also should study the metrics (the relative cost of implementing a table as ROM or logic, the relative cost of a squarer unit, etc), and probably placement considerations.

Another problem raised by this work is the explosion of the number of parameters, which excludes an exhaustive enumeration of the parameter space as in the multipartite case [6]. Current heuristics for fixing the parameters are based on trial and error, and should be improved. Experience may then show that the approach is uselessly complex, and that some parameters should remain fixed to obvious values.

Moreover, even though we have considered error bounds as constant on the interval $\mathcal{I}$ in section 5, the very same scheme can be applied when considering the bounds on ϵ_{method} as piecewise polynomials. This gives a much finer approximation of the method error bound, as shown in Fig. 13. The idea here is to gradually decrease the accuracy of the tables when ϵ_{method} is small compared to its extremal values (e.g. in the middle of the sub-intervals in Fig. 13) by using a non-constant number of input bits to address the table. This strategy will be interesting when synthesizing the tables as logic, as logic minimization will apply. We have already implemented this error analysis but we do not take advantage of it yet.

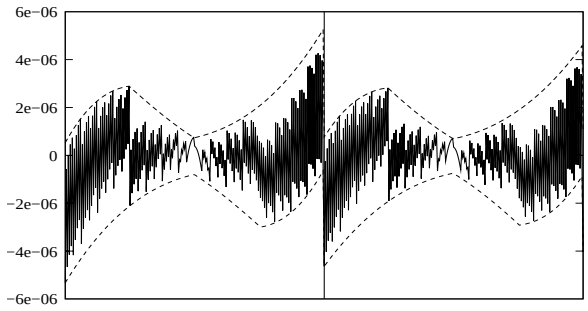

Figure 13: Bounding the method error ϵ_{method} with piecewise polynomials (the dashed lines).

Another question which remains open is the interest of the Horner evaluation method when targetting hardware. In the litterature concerning the precisions considered, we are only aware of very naive approaches [8]. To get a fair comparison, the Horner approach should be studied with an effort on the error analysis similar to that described in this paper.

Acknowledgements

The authors would like to thank Arnaud Tisserand for many interesting discussions on this topic, and also for administrating the CAD tool server on which all the synthesis presented in this paper were performed.

References

[1] J.-M. Muller, *Elementary Functions, Algorithms and Implementation.* Boston: Birkhauser, 1997.

[2] J. Detrey and F. de Dinechin, "A VHDL library of LNS operators," in *37th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, USA, Oct. 2003.

[3] D. Das Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *12th IEEE Symposium on Computer Arithmetic*, S. Knowles and W. McAllister, Eds. Bath, UK: IEEE Computer Society Press, 1995, pp. 17–28.

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

[5] J.-M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[6] F. de Dinechin and A. Tisserand, "Some improvements on multipartite table methods," in *15th IEEE Symposium on Computer Arithmetic*, N. Burgess and L. Ciminiera, Eds., Vail, Colorado, June 2001, pp. 128–135, updated version of LIP research report 2000-38.

[7] J. Detrey and F. de Dinechin, "Second order function approximation using a single multiplication on FPGAs," in *14th Intl Conference on Field-Programmable Logic and Applications*. Antwerp, Belgium: LNCS 3203, Aug. 2004, pp. 221–230.

[8] D.-U. Lee, W. Luk, J. Villasenor, and P. Cheung, "Hierarchical segmentation schemes for function evaluation," in *IEEE Conference on Field-Programmable Technology*, Tokyo, dec 2003.

[9] J. A. Piñeiro, J. D. Bruguera, and J.-M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *15th IEEE Symposium on Computer Arithmetic*, N. Burgess and L. Ciminiera, Eds., Vail, Colorado, June 2001, pp. 40–47.