Multipartite Table Methods

Florent de Dinechin, *Member*, *IEEE*, and Arnaud Tisserand, *Member*, *IEEE*

Abstract—A unified view of most previous table-lookup-and-addition methods (bipartite tables, SBTM, STAM, and multipartite methods) is presented. This unified view allows a more accurate computation of the error entailed by these methods, which enables a wider design space exploration, leading to tables smaller than the best previously published ones by up to 50 percent. The synthesis of these multipartite architectures on Virtex FPGAs is also discussed. Compared to other methods involving multipliers, the multipartite approach offers the best speed/area tradeoff for precisions up to 16 bits. A reference implementation is available at www.ens-lyon.fr/LIP/Arenaire/.

Index Terms—Computer arithmetic, elementary function evaluation, hardware operator, table lookup and addition method.

1 Introduction

Table-lookup-and-addition methods, such as the bipartite method, have been the subject of much recent attention [1], [2], [3], [4], [5]. They allow us to compute commonly used functions with low accuracy (up to 20 bits) with significantly lower hardware cost than that of a straightforward table implementation, while being faster than shift-and-add algorithms *à la* CORDIC or polynomial approximations. They are particularly useful in digital signal or image processing. They may also provide initial seed values to iterative methods, such as the Newton-Raphson algorithms for division and square root [6], [7], which are commonly used in the floating-point units of current processors. They also have recently been successfully used to implement addition and subtraction in the logarithm number system [8].

The main contribution of this paper is to unify two complementary approaches to multipartite tables by Stine and Schulte [4] and Muller [5]. Completely defining the implementation space for multipartite tables allows us to provide a methodology for selecting the best implementation that fulfills arbitrary accuracy and cost requirements. This methodology has been implemented and is demonstrated on a few examples. This paper also clarifies some of the cost and accuracy questions which are incompletely formulated in previous papers. This paper is an extended version of an article published in the *Proceedings of the 15th IEEE International Symposium on Computer Arithmetic* [9].

After some notations and definitions in Section 2, Section 3 presents previous work on table-lookup-and-addition methods. Section 4 presents our unified multipartite approach in all the details. Section 5 gives results and compares them to previous work. Section 6 concludes.

• *The authors are with the Arénaire Project (CNRS-ENSL-INRIA-UCBL) LIP, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon, France. E-mail: {Florent.de.Dinechin, Arnaud.Tisserand}@ens-lyon.fr.*

Manuscript received 1 Dec. 2003; revised 16 June 2004; accepted 17 Sept. 2004; published online 18 Jan. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0242-1203.

2 Generalities

2.1 Notations

Throughout this paper, we discuss the implementation of a function with inputs and outputs in fixed-point format. We shall use the following notations:

- We note $f : [a, b[\rightarrow [c, d[$, the function to be evaluated with its domain and range.
- We note w_I and w_O, the required input and output size.

In general, we will identify any word of p bits to the integer in $\{0, \ldots, 2^p - 1\}$ it codes, writing such a word in capital letters. When needed, we will provide explicit functions to map such an integer into the real domain or range of the function. For instance, an input word X will denote an integer in $\{0, \ldots, 2^{w_I} - 1\}$, and we will express the real number $x \in [a, b[$ that it codes by $x = a + (b - a)X/2^{w_I}$. Note that no integer maps to b, the right bound of the input interval, which explains why we define this interval as open in b. Such a mapping should be part of the specification of a hardware function evaluator and several alternatives exist, depending on the function to be evaluated and the needs of the application. Some applications may require that the integer X denotes $x = a + (b - a)(X + 1/2)/2^{w_I}$, some may require that it denotes $x = a + (b - a)X/(2^{w_I} - 1)$. For other applications to floating-point hardware, the input interval may span over two consecutive binades, in which case, we will consider two input intervals with different mappings. The reader should keep in mind that all the following work can be straightforwardly extended to any such mapping between reals and integers. A general presentation would degrade readability without increasing the interest of the paper. Our implementation, however, can accommodate arbitrary styles of discretization of the input and output intervals.

2.2 Errors in Function Evaluation

Usually, three different kinds of error sum up to the total error of an evaluation of $f(x)$:

- The *input discretization* or *quantization* error measures the fact that an input number usually represents a

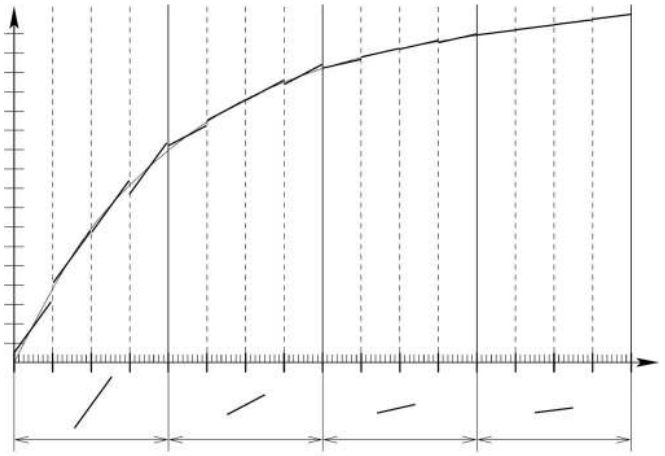

Fig. 1. The bipartite approximation.

small interval of values centered around this number.

- The *approximation* or *method* error measures the difference between the pure mathematical function f and the approximated mathematical function (here, a piecewise affine function) used to evaluate it.
- Finally, the actual computation involves *rounding* errors due to the discrete nature of the final and intermediate values.

In the following, we will ignore the question of input discretization by considering that an input number only represents itself as an exact mathematical number. Again, all the following work could probably be extended to take quantization errors into account.

3 Previous and Related Works

An approximation of a function may be simply stored in a look-up table containing 2^{w_I} values. This approach becomes impractical as soon as w_I exceeds 10-12 bits. In this section, we explore various methods which allow us to approximate functions with much less memory and very little computation.

The present paper improves on the bipartite idea and its successors, which are first presented in detail (Sections 3.1 to 3.3). As our results should be compared to other competitive hardware approximation methods, we then also present these methods (Sections 3.4 to 3.6). We leave out of this survey methods more specifically designed for a particular function, such as the indirect bipartite method for postscaled division [10], many methods developed for evaluating $f(x) = \log_2(1 \pm 2^x)$ for Logarithm Number System (LNS) arithmetic [11], [12], [13], [14], and many others.

3.1 The Bipartite Method

First presented by Das Sarma and Matula [1] in the specific case of the reciprocal function and generalized by Schulte and Stine [3], [4] and Muller [5], this method consists of approximating the function by affine segments, as illustrated in Fig. 1.

The 2^α segments (16 in Fig. 1) are indexed by the α most significant bits of the input word, as depicted in Fig. 2. For each segment, one initial value is tabulated and the other values are interpolated by adding, to this initial value, an offset computed out of the $w_I - \alpha$ least significant bits of the input word.

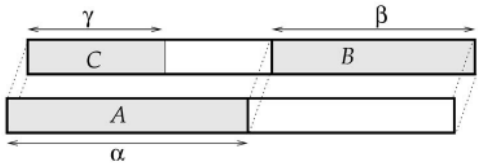

Fig. 2. Bipartite input word decomposition.

The idea behind the bipartite method is to group the 2^α input intervals into 2^γ (with $\gamma < \alpha$) larger intervals (four in Fig. 1) such that the slope of the segments is considered constant on each larger interval. These four constant slopes are figured in Fig. 1 and may be tabulated: Now, there are only 2^γ tables of offsets, each containing 2^β offsets. Altogether, we thus need to store $2^\alpha + 2^{\gamma+\beta}$ values instead of $2^{w_I} = 2^{\alpha+\beta}$.

In all the following, we will call the table that stores the initial points of each segment the *Table of Initial Values (TIV)*. This table will be addressed by a subword A of the input word, made of the α most significant bits. A *Table of Offsets (TO)* will be addressed by the concatenation of two subwords of the input word: C (the γ most significant bits) and B (the β least significant bits). Fig. 2 depicts this decomposition of the input word.

Previous authors [5], [4] have expressed the bipartite idea in terms of a Taylor approximation, which allows a formal error analysis. They find that, for $\gamma \approx \beta \approx \alpha/2$, it is possible to keep the error entailed by this method in "acceptable bounds" (the error obviously depends on the function under consideration). We develop in this paper a more geometrical approach to the error analysis with the purpose of computing the approximation error exactly, where Taylor formulas only give upper bounds.

3.2 Exploiting Symmetry

Schulte and Stine have remarked [3] that it is possible to exploit the symmetry of the segments on each small interval (see Fig. 3, which is a zoom view of Fig. 1) to halve the size of the TO: They store the value of the function in the middle of the small interval in the TIV and the offsets for a half segment in the TO. The offsets for the other half are computed by symmetry. The extra hardware cost (mostly a few XOR gates) is usually more than compensated by the reduction in the TO size (see the SBTM paper, for *Symmetric Bipartite Table Addition Method* [3]).

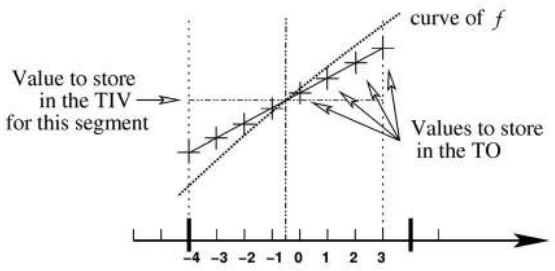

Fig. 3. Segment symmetry.

Note that the initial bipartite paper [1] suggested using an "average curve" approximation instead of a linear one for the TO. This idea wouldn't improve the maximum error, but would bring a small improvement to the average error (a fraction of half an ulp, as Section 4 will show). However, in this case, Fig. 3 is no longer symmetric and the table size reduction discussed here is no longer possible. Therefore, this idea will not be considered further.

3.3 Multipartite Methods

In another paper [4], Stine and Schulte have remarked that the TO can be decomposed into several smaller tables: What the TO computes is a linear function $\mathrm{TO}(CB) = s(C) \times B$, where $s(C)$ is the slope of the segment. The subword B can be decomposed (as seen in Fig. 6) into m subwords, B_i, of sizes β_i for $0 \leq i < m$:

$$B = B_0 + 2^{\beta_0} B_1 + \ldots + 2^{\beta_0+\beta_1+\ldots+\beta_{m-2}} B_{m-1}.$$

Let us define $p_0 = 0$ and $p_i = \sum_{j=0}^{i-1} \beta_j$ for $i > 0$. The function computed by the TO is then:

$$\begin{aligned} \mathrm{TO}(CB) &= s(C) \times \sum_{i=0}^{m-1} 2^{p_i} B_i \\ &= \sum_{i=0}^{m-1} 2^{p_i} s(C) \times B_i \\ &= \sum_{i=0}^{m-1} 2^{p_i} \mathrm{TO}_i(CB_i). \end{aligned} \quad (1)$$

Thus, the TO can be distributed into m smaller tables, $\mathrm{TO}_i(CB_i)$, resulting in much smaller area (symmetry still applies for the m TO_is). This comes at the cost of $m-1$ additions. This improvement thus entails two tradeoffs:

- A cost tradeoff between the cost of the additions and the table size reduction.
- An accuracy tradeoff: Equation (1) is not an approximation, but it will lead to more discretization errors (one per table), which will sum up to a larger global discretization error unless the smaller tables have a bigger output accuracy (and, thus, are bigger). We will formalize this later.

Schulte and Stine have termed this method STAM, for *Symmetric Table and Addition Method*. It can still be improved: Note, in (1) that, for $j > i$, the weight of the LSB of TO_j is $2^{p_j - p_i}$ times the weight of the LSB of TO_i. In other terms, TO_i is more accurate than TO_j. It will be possible, therefore, to build even smaller tables than Schulte and Stine by compensating for the (wasted) higher accuracy of TO_i by a rougher approximation on $s(C)$, obtained by removing some least significant bits from the input C.

A paper from Muller [5], contemporary to that of Stine and Schulte, indeed exploits this idea in a specific case. The *multipartite* method presented there is based on a decomposition of the input word into $2p+1$ subwords $X_1, \ldots, X_{2p+1}$ of identical sizes. An error analysis based on a Taylor formula shows that equivalent accuracies are obtained by a table addressed by X_{2p+1} and a slope determined only by X_1, a table addressed by X_{2p} and a slope determined by X_1X_2, and, in general, a table addressed by X_{2p+2-i} and the i most significant subwords.

Muller claims (although without any numerical support) that the error/cost tradeoffs of this approach are comparable to Schulte and Stine's method. His decomposition, however, is too rigid to be really practical, while his error analysis is based on potentially overestimated error bounds due to the Taylor approximation. Besides, he doesn't address the rounding issue.

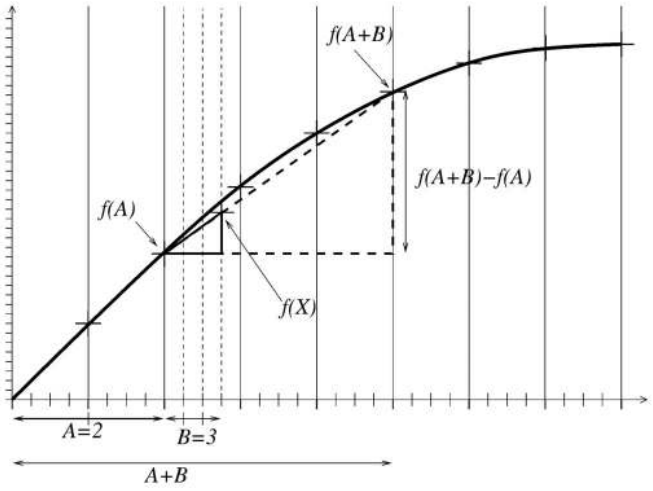

Fig. 4. First order ATA.

3.4 ATA Methods

The *Addition-Table-Addition* methods allow additions before and after the table look-ups. They are termed after Wong and Goto [15]; however, a whole range of such methods is possible and, to our knowledge, unpublished. This section is a survey of these methods.

Let us note $X = A + 2^{-\beta}B = a_{\alpha-1} \ldots a_0 b_{\beta-1} \ldots b_0$, where $\alpha > \beta$.

To compute $f(A + 2^{-\beta}B)$, it is possible to use the first-order Taylor approximation:

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta} B f'(A)$$

with

$$Bf'(A) \approx f(A+B) - f(A).$$

Finally,

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta}(f(A+B) - f(A)).$$

In other terms, this first-order ATA method approximates, in a neighborhood of A, the graph of f with a homotetic reduction of this graph with a ratio of 2^{β}, as pictured in Fig. 4.

Evaluating $f(x)$ thus involves

- one α-bit addition to compute $a_{\alpha-1} \ldots a_0 + b_{\beta-1} \ldots b_0$,
- two lookups in the same table,
  - $f(a_{\alpha-1} \ldots a_0)$ and
  - $f(a_{\alpha-1} \ldots a_0 + b_{\beta-1} \ldots b_0)$,
- one subtraction to compute the difference between the previous two lookups on less than α bits (the size

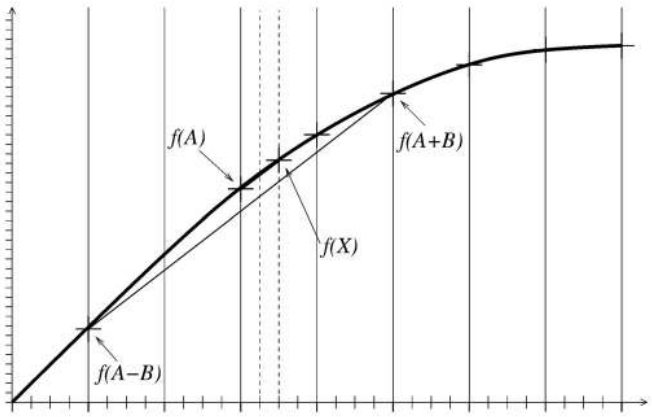

Fig. 5. Second order ATA.

of this subtraction depends on f and β and may be computed exactly),
- one shift by β bits to perform the division by 2^β, and
- one final addition on w_O bits.

Both table lookups can be performed in parallel in a dual-port table or in parallel using two tables or in sequence/pipeline (one read before the addition and one read after). This leads to a range of architectural tradeoffs.

This method can be extended to the second order by using a *central difference formula* to compute a much better approximation of the derivative (the error is a third-order term) as depicted in Fig. 5.

The formula used is now

$$Bf'(A) \approx \frac{f(A+B) - f(A-B)}{2}$$

and the algorithm consists of the following steps:

- Compute (in parallel) $A + B$ and $A - B$;
- Read in a table $f(A)$, $f(A + B)$, and $f(A - B)$;
- Compute

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta-1}(f(A+B) - f(A-B)).$$

We now need three lookups in the same table and seven additions. Here again, a range of space/time tradeoffs is possible.

The original method by Wong and Goto [15] is actually more sophisticated: As in the STAM method, they split B into two subwords of equal sizes, $B = B_1 + 2^{\beta/2}B_2$, and distribute $Bf'(A)$ using two centered differences, which reduces table sizes. Besides they add a table which contains second and third-order corrections, indexed by the most-significant half-word of A and the most-significant half-word of B. For 24 bits of precision, their architecture therefore consists of six tables with 12 or 13 bits of inputs and a total of nine additions.

Another option would be to remark that, with these three table lookups, it is also possible to use a second-order Taylor formula:

$$f(A + 2^{-\beta}B) \approx f(A) \; + \; 2^{-\beta}Bf'(A) \; + \; \frac{(2^{-\beta}B)^2}{2} f''(A).$$

Indeed, we may approximate the term $f''(A)$ by

$$\begin{aligned} f''(A) &\approx \frac{f'(A + B/2) - f'(A - B/2)}{B} \\ &\approx \frac{\frac{f(A+B)-f(A)}{B} - \frac{f(A)-f(A-B)}{B}}{B} \\ &\approx \frac{f(A+B) - 2f(A) + f(A-B)}{B^2}. \end{aligned}$$

And, finally,

$$\begin{aligned} f(A + 2^{-\beta}B) \approx\; & f(A) \\ & + 2^{-\beta-1}(f(A+B) - f(A-B)) \\ & + 2^{-2\beta-1}(f(A+B) - 2f(A) + f(A-B)). \end{aligned}$$

However, the larger number of look-ups and arithmetic operations entails more rounding errors, which actually consume the extra accuracy obtained thanks to this formula.

Finally, the ATA methods can be improved using symmetry, just like multipartite methods.

These methods have been studied by the authors and found to perform better than the original bipartite approaches, but worse than the generalized multipartite approach which is the subject of this paper. This is also true of the initial ATA architecture by Wong and Goto [15], as will be exposed in Section 5.1.

3.5 Partial Product Arrays

This method is due to Hassler and Takagi [2]. The idea is to approximate the function with a polynomial of arbitrary degree (they use a Taylor approximation). Writing X and all the constant coefficients as sums of weighted bits (as in $X = \sum x_i 2^{-i}$), they distribute all the multiplications within the polynomial, thus rewriting the polynomial as the sum of a huge set of weighted products of some of the x_i. A second approximation then consists of neglecting as many of these terms as possible in order to be able to partition the remaining ones into several tables.

This idea is very powerful because the implementation space is very wide. However, for the same reason, it needs to rely on heuristics to explore this space. The heuristic chosen by Hassler and Takagi in [2] leads to architectures which are less compact than their multipartite counterpart [4] (and are interestingly similar). The reason is probably that the multipartite method exploits the higher-level property of continuity of the function, which is lost in the set of partial products.

3.6 Methods Involving Multipliers

The previous two methods involve higher order approximation of the function, but the architecture involves only adders and tables. If this constraint is relaxed, a huge range of approximations becomes possible. The general scheme is to approximate the function with one or several polynomials and trade table size for multiplications. Papers relevant to this work include (but this list is far from exhaustive):

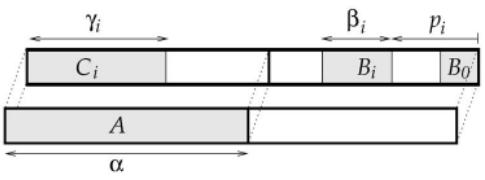

Fig. 6. Multipartite input word decomposition.

- an architecture by Defour et al. [16] involving only two small multipliers (small meaning that their area and delay are comparable to a few adders as one of the inputs is only $w_I/5$ bits wide);
- an architecture by Piñeiro et al. [17] using a squarer unit and a multiplication tree;
- several implementations of addition and subtraction in the logarithm number system with (among others) approximation of order zero [11], order one [12], and order two [13], [14]. As already mentioned, the function to be evaluated is $f(x) = \log_2(1 \pm 2^x)$ and lends itself to specific tricks, like replacing multiplications with additions in the $\log$ domain.

These methods will be quantitatively compared to the multipartite approach in Section 5.5.

3.7 Conclusion: Architectural Consideration

A common feature of all the methods presented in this section is that they lead to architectures where the result is the output of an adder tree. This adder tree lends itself to a range of area/time tradeoffs which depends on the architectural target and also on the time/area constraints of the application.

However, as initially noted by Das Sarma and Matula, there are many applications where the last stage of the adder tree (which is the most costly as it involves the carry propagation) is not needed: Instead, the result may be provided in redundant form to the operator that consumes it. It is the case when a table-and-addition architecture provides the seed to a Newton-Raphson iteration, for instance: The result can be recoded (using Booth or modified Booth algorithm) without carry propagation to be input to a multiplier.

This remark shows that the cost of the adder tree depends not only on the target, but also on the application. For these reasons, the sequel focuses on minimizing the table size.

4 The Unified Multipartite Method

4.1 A General Input-Word Decomposition

Investigating what is common to Schulte and Stine's STAM and Muller's multipartite methods leads us to define a decomposition into subwords that generalize both (see Fig. 6):

- The input word is split into two subwords, A and B, of respective sizes α and β, with $\alpha + \beta = w_I$.
- The most significant subword A addresses the TIV.
- The least significant subword B will be used to address $m \geq 1$ TOs.

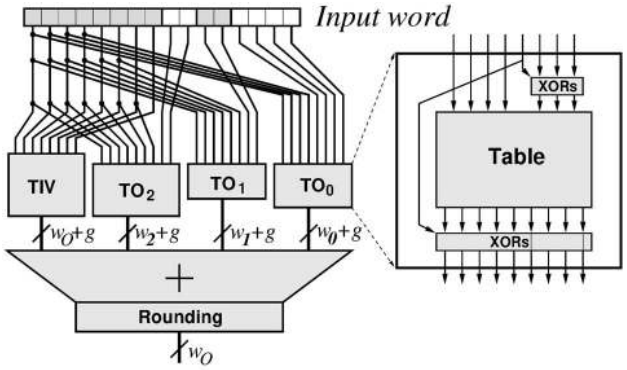

Fig. 7. Multipartite architecture.

  - B will in turn be decomposed into m subwords $B_0, \ldots, B_{m-1}$, the least significant being B_0.
  - A subword B_i starts at position p_i and consists of β_i bits. We have $p_0 = 0$ and $p_{i+1} = p_i + \beta_i$.
  - The subword B_i is used to address the TO_i, along with a subword C_i of length γ_i of A.
- Finally, to simplify notations, we will denote $\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}\}$ such a decomposition.

The maximum approximation error entailed by TO_i will be a function of (γ_i, p_i, β_i) which we will be able to compute exactly in Section 4.3. The TOs implementation will exploit their symmetry, just as in the STAM method.

The reader may check that the bipartite decomposition is a special case of our multipartite decomposition with $m = 1$, $\alpha = 2w_I/3$, $\gamma = w_I/3$, $\beta = \beta_0 = w_I/3$. Similarly, Stine and Schulte's STAM [4] is a multipartite decomposition where all the C_is are equal and Muller's multipartite approach [5] is a specific case of our decomposition where the γ_is are multiples of some integer.

Fig. 7 shows a general multipartite implementation, using symmetry. It should be clear that general decompositions are more promising than Stine and Schulte's in that they allow us to reduce the accuracy of the slopes involved in the TOs (and, thus, their size). They are also more promising than Muller's as they are more flexible (for example, the size of the input word need not be a multiple of some $2p + 1$). Our methodology will also be slightly more accurate than both in computing the slopes and in the error analysis. Section 5 will quantify these improvements.

4.2 An Algorithm for Choosing a Decomposition

Having defined the space of all the possible multipartite decompositions, we define in this section an efficient methodology to explore this space. The purpose of such an exploration is to select the best decomposition (in terms of speed or area) that fulfills the accuracy requirement known as *faithful rounding*: The returned result should be one of the two fixed-point numbers closest to the mathematical value. In other words, the total error should be smaller than the value ϵ_{total} of one unit in the last place (ulp):

$$\epsilon_{\text{total}} = (d - c)2^{-w_O}. \tag{2}$$

This error will be the sum of an approximation error, which depends only on the decomposition, and the various rounding errors.

Unfortunately, the tables cannot be filled with results rounded to the target precision: Each table would entail a maximum rounding error of $0.5\epsilon_{\text{total}}$, meaning that the total error budget of ϵ_{total} is unfeasible as soon as there is more than one table. The tables will therefore be filled with a precision greater than the target precision by g bits (guard bits). Thus, rounding errors in filling one table are now

$$\epsilon_{\text{rnd\_table}} = 2^{-g-1}\epsilon_{\text{total}} \tag{3}$$

and can be made as small as desired by increasing g. The sum of these errors will be smaller than

$$\epsilon_{\text{rnd\_m\_tables}} = (m+1)\epsilon_{\text{rnd\_table}}, \tag{4}$$

where $(m+1)$ is the number of tables.

However, the final summation is now also performed on g more bits than the target precision. Rounding the final sum to the target precision now entails a rounding error up to $\epsilon_{\text{rnd\_final}} = 0.5\epsilon_{\text{total}}$. A trick due to Das Sarma and Matula [1] allows us to improve it to

$$\epsilon_{\text{rnd\_final}} = 0.5\epsilon_{\text{total}}(1 - 2^{-g}). \tag{5}$$

This trick will be presented in Section 4.6.2.

This error budget suggests the following algorithm:

1. Choose the number of tables m. A larger m means smaller tables, but more additions.
2. Enumerate the decompositions
$$\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}\}.$$
3. For each decomposition $\mathcal{D}$,
   a. Compute the bounds $\epsilon_i^{\mathcal{D}}$ on the approximation errors entailed by each TO_i (see Section 4.3) and sum them to get $\epsilon_{\text{approx}}^{\mathcal{D}} = \sum_{i=0}^{m-1} \epsilon_i^{\mathcal{D}}$. Keep only those decompositions for which this error is smaller than the error budget.
   b. As the two other error terms $\epsilon_{\text{rnd\_final}}$ and $\epsilon_{\text{rnd\_m\_tables}}$ depend on g, compute the smallest g allowing to match the total error budget. This will be detailed in Section 4.4.
   c. Knowing g allows precise evaluation of the size of the implementation of $\mathcal{D}$, as will be detailed in Section 4.5.
4. Synthesize the few best candidates to evaluate their speed and area accurately (with target constraints).

Enumerating the decompositions is an exponential task. Fortunately, there are two simple tricks which are enough to cut the enumeration down to less than a minute for 24-bit operands (the maximum size for which multipartite methods architectures make sense).

- The approximation error due to a TO_i is actually only dependent on the function evaluated, the input precision, and the three parameters p_i, β_i, and γ_i of this TO_i. It is therefore possible to compute all these errors only once and store them in a three-dimensional array $\epsilon_{\text{TO}}[p][\beta][\gamma]$. The size of this array is at most 24^3 double-precision floating-point numbers.
- For a given pair (p_i, β_i), this error grows as γ_i decreases. There exists a $\gamma_{\min}$ such that, for any $\gamma_i \leq \gamma_{\min}$, this error is larger than the required output precision. These $\gamma_{\min}(p_i, \beta_i)$ may also be computed once and stored in a table.

Finally, the enumeration of the (p_i, β_i) is limited by the relation $p_{i+1} = p_i + \beta_i$ and the enumeration on γ_i is limited by $\gamma_{\min} < \gamma_i < \alpha$. Note that we have only left out decompositions which were unable to provide faithful rounding. It would also be possible, in addition, to leave out decomposition whose area is bigger than the current best. This turns out not to be needed.

The rest of this section details the steps of this algorithm.

4.3 Computing the Approximation Error

Here, we consider a monotonic function with monotonic derivative (i.e., convex or concave) on its domain. This is not a very restrictive assumption: It is the case, after argument reduction, of all the functions studied by previous authors.

The error function we consider here is the difference $\varepsilon(x) = f(x) - \widetilde{f}(x)$ between the exact mathematical value and the approximation. Note that other error functions are possible, for example, taking into account the input discretization. The formulas set up here would not apply in that case, but it would be possible to set up equivalent formulas.

Using these hypotheses, it is possible to exactly compute, using only a few floating-point operations in double precision, the minimum approximation error which will be entailed by a TO_i with parameters p_i, β_i, and γ_i, and also the exact value to fill in these tables as well as in the TIV to reach this minimal error.

The main idea is that, for a given (p_i, β_i, γ_i), the parameters that can vary to get the smallest error are the slope $s(Ci)$ of the segments and the values $\text{TIV}(A)$. With our decomposition, several $\text{TIV}(A)$ will share the same $s(Ci)$. Fig. 8 (another zoom of Fig. 1) depicts this situation.

As the figure suggests, with our hypothesis of a monotonic (decreasing on the figure) derivative, the approximation error is maximal on the borders of the interval on which the segment slope is constant. The minimum $\epsilon_i^{\mathcal{D}}(C_i)$ of this maximum error is obtained when

$$\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4 = \epsilon_i^{\mathcal{D}}(C_i) \tag{6}$$

with the notations of the figure. This system of equations is easily expressed in terms of $s(C_i)$, p_i, β_i, γ_i, TIV, and f. Solving this system gives the optimal slope[1] and the corresponding error:

$$s_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) + f(x_4) - f(x_3)}{2\delta_i}, \tag{7}$$

$$\epsilon_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) - f(x_4) + f(x_3)}{4}, \tag{8}$$

where (using the notations of Section 2.1)

1. Not surprisingly, the slope that minimizes the error is the average value of the slopes on the borders of the interval. Previous authors considered the slope at the midpoint of this interval.

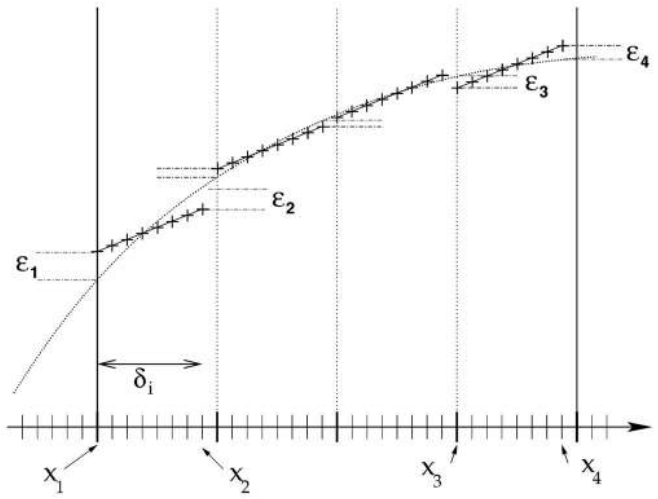

Fig. 8. Computing the approximation error.

$$\delta_i = (b-a)2^{-w_I+p_i}(2^{\beta_i}-1), \tag{9}$$

$$x_1 = a + (b-a)2^{-\gamma_i}C_i, \tag{10}$$

$$x_2 = x_1 + \delta_i, \tag{11}$$

$$x_3 = x_1 + (b-a)(2^{-\gamma_i} - 2^{-w_I+p_i+\beta_i}), \tag{12}$$

$$x_4 = x_3 + \delta_i. \tag{13}$$

Now, this error depends on C_i, that is, on the interval on which the slope is considered constant. For the same argument of convexity, it will be maximum either for $C_i = 0$ or for $C_i = 2^{\gamma_i} - 1$. Finally, the maximum approximation error due to TO_i in the decomposition $\mathcal{D}$ is:

$$\epsilon_i^{\mathcal{D}} = \max(|\epsilon_i^{\mathcal{D}}(0)|, |\epsilon_i^{\mathcal{D}}(2^{\gamma_i}-1)|). \tag{14}$$

In practice, it is easy to compute this approximation error by implementing (8) to (14). Altogether, it represents a few floating-point operations per TO_i.

4.4 Computing the Number of Guard Bits

The condition to ensure faithful rounding, $\epsilon_{\mathrm{rnd\_m\_tables}} + \epsilon_{\mathrm{rnd\_final}} + \epsilon_{\mathrm{approx}}^{\mathcal{D}} < \epsilon_{\mathrm{total}}$ is rewritten using (2), (3), (4), and (5) as:

$$g > -w_O - 1 + \log_2((d-c)m) - \log_2((d-c)2^{-w_O-1} - \epsilon_{\mathrm{approx}}^{\mathcal{D}}).$$

If $\epsilon_{\mathrm{approx}}^{\mathcal{D}} \geq (d-c)2^{-w_O-1}$, $\mathcal{D}$ is unable to provide the required output accuracy. Otherwise, the previous inequality gives us the number g of extra bits that ensures faithful rounding:

$$g = \left\lceil -w_O - 1 + \log_2 \frac{(d-c)m}{(d-c)2^{-w_O-1} - \epsilon_{\mathrm{approx}}^{\mathcal{D}}} \right\rceil. \tag{15}$$

Our experiments show that it is very often possible to decrease this value by one and still keep faithful rounding. This is due to the actual worst-case rounding error in each table being smaller than the one assumed above, thanks to the small number of entries for each table. This question will be discussed in Section 5.2.

4.5 The Sizes of the Tables

Evaluating precisely the size and speed of the implementation of a multipartite decomposition is rather technology dependent and is out of the scope of the paper. We can, however, compute exactly (as other authors) the number of bits to store in each table.

The size in bits of the TIV is simply $2^{\alpha}(w_O + g)$. The TO_is have a smaller range than the TIV: Actually, the range of $\mathrm{TO}_i(C_i, *)$ is exactly equal to $|s_i(C_i) \times \delta_i|$. Again, for convexity reasons, this range is maximum either on $C_i = 0$ or $C_i = 2^{\gamma_i} - 1$:

$$r_i = \max(|s_i(0) \times \delta_i|, |s_i(2^{\gamma_i}-1) \times \delta_i|). \tag{16}$$

The number of output bits of TO_i (without the guard bits) is therefore

$$w_i = \left\lceil w_O + g - \log_2\left(\frac{d-c}{r_i}\right) \right\rceil. \tag{17}$$

In a symmetrical implementation of the TO_i, the size in bits of the corresponding table will be $2^{\gamma_i+\beta_i-1}(w_i - 1)$.

The actual costs (area and delay) of implementations of these tables and of multioperand adders are technology dependent. We present in Section 5.4 some results for Virtex-II FPGAs, showing that the bit counts presented above allow a predictive enough evaluation of the actual costs.

4.6 Filling the Tables

4.6.1 The Mathematical Values

An initial value $\mathrm{TIV}(A)$ provided by the TIV for an input subword A will be used on an interval $[x_l, x_r]$ defined (using the notations of Sections 2.1 and 4.3) by:

$$x_l = a + (b-a)2^{-\alpha}A, \tag{18}$$

$$x_r = x_l + \sum_{i=0}^{m-1} \delta_i. \tag{19}$$

On this interval, each TO_i provides a constant slope, as its C_i is a subword of A. The approximation error, which is the sum of the $\epsilon_i^{\mathcal{D}}(C_i)$ defined by (8), will be maximal for x_l and x_r (with opposite signs).

The TIV exact value that ensures that this error bound is reached is therefore (before rounding):

$$\widetilde{\mathrm{TIV}}(A) = \frac{f(x_l) + f(x_r)}{2}. \tag{20}$$

The TO_i values before rounding are (see Fig. 3):

$$\widetilde{\mathrm{TO}}_i(C_iB_i) = s(C_i) \times 2^{-w_I+p_i}(b-a)\left(B_i + \frac{1}{2}\right). \tag{21}$$

4.6.2 Rounding Considerations

This section reformulates the techniques employed by Stine and Schulte in [4] and using an idea that seems to appear first in the paper by Das Sarma and Matula [1].

The purpose is to fill our tables in such a way as to ensure that their sum (which we compute on $w_O + g$ bits)

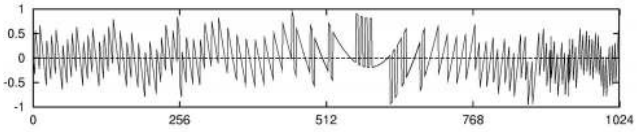

Fig. 9. Measured error (10-bit sine and $m = 2$).

always has an implicit 1 as its $(w_O + g + 1)$th bit. This reduces the final rounding error from $\epsilon_{\text{rnd\_final}} = 2^{-w_O-1}$ to $\epsilon_{\text{rnd\_final}} = 2^{-w_O-1} - 2^{-w_O-g-1}$.

To achieve this trick, remark that there are two ways to round a real number to $w_O + g$ bits with an error smaller than $\epsilon_{\text{rnd\_table}} = 2^{-w_O-g-1}$. The natural way is to round the number to the nearest $(w_O + g)$-bit number. Another method is to truncate the number to $w_O + g$ bits and assume an implicit 1 in the $(w_O + g + 1)$th position.

To exploit the symmetry, we will need to compute the opposite of the value given by a TO_i. In two's complement, this opposite is the bitwise negation of the value, plus a 1 at the LSB. This leads us to use the second rounding method for the TO_i. Knowing that its LSB is an implicit 1 means that its negation is a 0 and, therefore, that the LSB of the opposite is also a 1. We therefore don't have to add the sign bit at the LSB. We store and bitwise negate the $w_i + g - 1$ bits of the TO_i and assume in all cases an implicit 1 at the $(w_O + g + 1)$th position.

Now, in order to reach our goal of always having an implicit 1 at the $(w_O + g + 1)$th bit of the sum, we need to consider the parity of m, the number of TO_is. If m is odd, the first rounding method is used for the TIV, if m is even, the second method is used. This way we always have $\lfloor m/2 \rfloor$ implicit ones, which we simply add to all the values of the TIV to make them explicit.

Finally, after summing the TIV and the TO_i, we need to round the sum, on $(w_O + g)$ bits with an implicit 1 at the $(w_O + g + 1)$th bit, to the nearest number on w_O bits. This can be done by simply truncating the sum (at no hardware cost), provided we have added half an LSB of the final result to the TIV when filling it.

Summing it up, the integer values that should fill the TO_is are

$$\text{TO}_i(C_iB_i) = \left\lfloor \frac{2^{w_O+g}}{d-c} \widetilde{\text{TO}}_i(C_iB_i) \right\rfloor \tag{22}$$

and the values that should fill the TIV are, if m is odd:

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m-1}{2} + 2^{g-1} \right\rfloor \tag{23}$$

and, if m is even:

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m}{2} + 2^{g-1} \right\rfloor. \tag{24}$$

4.7 Implementation

The methodology presented above has been implemented in a set of Java and C++ programs. These programs enumerate the decompositions, choose the best one with respect to accuracy and size, compute the actual values of the tables, and, finally, generate synthesizable VHDL.

TABLE 1
The Functions Tested, with Actual Values of w_I and w_O for 16-Bit Precision

function	input	output	w_I	w_O
sin	$[0, \pi/4[$	$[0, 1[$	16	16
2^x	$[0, 1[$	$[1, 2[$	16	15
$1/x$	$[1, 2]$	$[1/2, 1]$	15	15

Our tools also perform various additional checks. Storing the $\widetilde{\text{TIV}}$ and $\widetilde{\text{TO}}_i$, they measure the actual value of $\epsilon^{\mathcal{D}}_{\text{approx}}$. We find that the predicted values are indeed accurate to 10^{-7}. They similarly compute the maximal final error and check that this error is really smaller than the expected accuracy (see Fig. 9 for an example of output).

5 Results

This section studies the size and area of operators obtained using this methodology. The functions used are given in Table 1 with their input and output intervals. Some of these functions are identical to those in [4]. Notice that the bits that are constant over the input or output interval are not counted in w_I or w_O. Also notice that the output interval for the sine function is not the image of the input interval (which would be $[0, 1/\sqrt{2}[$), but, rather, a larger interval which will allow easy argument reduction using trigonometric identities.[2]

5.1 Comparison with Previous Work

Tables 2 and 3 present the best decomposition obtained for 16-bit and some 24-bit operands for a few functions. In these tables, we compare our results with the best-known results from the work of Schulte and Stine [4]. We can notice a size improvement up to 50 percent. The size for $1/x$ and $m = 1$ is larger than the reference size. After investigation, this is due to rounding errors compensating, in this specific case leading to an overestimated g.

5.2 Further Manual Optimization

The results obtained by the automatic method presented above can usually be slightly improved, up to 15 percent in terms of table sizes. The reason is that the automatic algorithm assumes that worst-case rounding will be attained in filling the tables, which is not the case. As we have many TO_is with few entries (typically, 2^5 to 2^8 entries for 16-bit operands in Table 2), there is statistically a good chance that the sum of the table-filling rounding errors is significantly smaller than predicted. This is a random effect which can only be tested by an exhaustive check of the architecture. However, in a final stage, it is worth trying several slight variations of the parameters, which can be of two types:

2. The specification, in the two papers by Schulte and Stine [3], [4] of the input and output intervals for the sine function is inconsistent. The input interval should probably read $[0, \pi/4[$ instead of $[0, 1[$. The output mapping is also unclear. Therefore, the comparisons concerning the sine function in this paper may be misleading.

TABLE 2
Best Decomposition Characteristics and Table Sizes for 16-Bit Operands

f	m	α	β	γ_i	β_i	g	tables	size	ref size
sin	1	10	6	5	6	1	$17.2^{10}+6.2^{10}$	23552	32768
	2	8	8	6 5	4 4	3	$19.2^{8}+10.2^{9}+6.2^{8}$	11520	20480
	3	8	8	7 5 4	2 3 3	2	$18.2^{8}+9.2^{8}+7.2^{7}+4.2^{6}$	8064	17920
	4	8	8	6 6 5 5	2 2 2 2	3	$19.2^{8}+10.2^{7}+8.2^{7}+6.2^{6}+4.2^{6}$	7808	na
2^x	1	10	6	5	6	1	$16.2^{10}+6.2^{10}$	22528	24576
	2	8	8	7 5	3 5	2	$17.2^{8}+9.2^{9}+6.2^{9}$	12032	14592
	3	8	8	7 6 4	2 3 3	2	$17.2^{8}+9.2^{8}+7.2^{8}+4.2^{6}$	8704	13568
	4	8	8	7 6 5 4	2 2 2 2	2	$17.2^{8}+9.2^{8}+7.2^{7}+5.2^{6}+3.2^{5}$	7968	na
$1/x$	1	10	5	7	5	1	$16.2^{10}+6.2^{11}$	28672	24576
	2	9	6	7,6	3,3	3	$18.2^{9}+9.2^{9}+6.2^{8}$	15360	16896
	3	9	6	8,7,5	2,2,2	2	$17.2^{9}+8.2^{9}+6.2^{8}+4.2^{6}$	14592	15872

TABLE 3
Best Decomposition Characteristics and Table Sizes for 24-Bit Operands

f	m	α	β	γ_i	β_i	g	tables	size	ref size
sin	1	15	9	7	9	3	$27.2^{15}+11.2^{15}$	1245184	1998848
	2	13	11	10 6	4 7	3	$27.2^{13}+13.2^{13}+9.2^{12}$	364544	753664
	3	12	12	10 9 6	3 4 5	4	$28.2^{12}+15.2^{12}+12.2^{12}+8.2^{10}$	233472	610304
	4	12	12	10 10 8 7	2 2 4 4	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{11}+7.2^{10}$	201728	507904
	5	12	12	10 10 9 7 6	2 2 2 3 3	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{10}+9.2^{9}+6.2^{8}$	189440	491520
	6	12	12	10 10 9 8 7 5	2 2 2 2 2 2	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{10}+9.2^{9}+7.2^{8}+5.2^{6}$	190016	na
2^x	1	15	9	8	9	1	$24.2^{15}+9.2^{16}$	1376256	1474560
	2	13	11	10 8	5 6	2	$25.2^{13}+12.2^{14}+7.2^{13}$	458752	581632
	3	12	12	11 9 7	3 4 5	3	$26.2^{12}+14.2^{13}+11.2^{12}+7.2^{11}$	280576	425984
	4	12	12	11 10 8 8	2 3 3 4	3	$26.2^{12}+14.2^{12}+12.2^{12}+9.2^{10}+6.2^{11}$	234496	360448
	5	12	12	11 10 9 8 8	2 2 2 3 3	3	$26.2^{12}+14.2^{12}+12.2^{11}+10.2^{10}+8.2^{10}+5.2^{10}$	211968	356352
	6	12	12	11 10 9 8 8 8	2 2 2 2 2 2	3	$26.2^{12}+14.2^{12}+12.2^{11}+10.2^{10}+8.2^{9}+6.2^{9}+4.2^{9}$	207872	na
$1/x$	1	15	8	9	8	5	$28.2^{15}+13.2^{16}$	1769472	1933312
	2	14	9	11 8	3 6	3	$26.2^{14}+12.2^{13}+9.2^{13}$	598016	884736
	3	13	10	12 10 8	2 3 5	4	$27.2^{13}+14.2^{13}+12.2^{12}+9.2^{12}$	421888	688128
	4	13	10	11 11 10 9	2 2 3 3	4	$27.2^{13}+14.2^{12}+12.2^{12}+10.2^{12}+7.2^{11}$	382976	524880
	5	13	10	11 11 10 9 8	2 2 2 2 2	5	$28.2^{13}+15.2^{12}+13.2^{12}+11.2^{11}+9.2^{10}+7.2^{9}$	379392	651264

- g may be decremented (as this is a variation of one parameter, it should actually be automatically performed).
- Some of the γ_i can be decremented (meaning less accurate slope). Remark that this negative effect on the mathematical error may be compensated by the halving of the number of values in the corresponding TO_i, which doubles the expected distance to the worst-case rounding.

Table 4 gives the example of a lucky case, with 11.5 percent improvement in size. These values of the γ_i even produce a method error of more than 0.5 ulp, which the lucky rounding compensates.

TABLE 4
Effect of Manual Optimization of the Parameters (Sine, 16 Bits, $m = 4$)

	γ_i	g	size	max. measured error in ulp
automatic	6 6 5 5	3	7808	0.915
fine-tuned	6 **5 4 3**	3	**6912**	0.939

Such a size improvement is reflected in the FPGA implementation: See Table 8 in Section 5.4.

5.3 Multipartite Are Close to Optimal among Order-One Methods

We remark in Table 2 and Table 3 that, for large values of m, the parameter α is close to $w_I/2$. Consider the family of linear (order-one) approximation schemes using some α bits of the input to address a TIV. There is an intrinsic lower bound on α for this family and it is the α for which the (mathematical) approximation error prevents faithful rounding. Generally speaking, this bound is about $w_I/2$, as given by the Taylor formula (and $w_I/3$ for order-two methods, etc.). This bound can be computed for each function exactly and we find that the best multipartite decomposition almost always exhibits the smallest α compatible with a faithful approximation.

Combined with the observation that the main contribution to the total size is always the TIV, this allows us to claim that our best multipartite approximations are close to the global optimal in the family of linear approximation schemes. More accurately, even the availability of a costless

TABLE 5
Virtex-II FPGA Implementation for Some Functions (16-Bit)

f	m	slices	(% of chip)	delay	T_{synth}	CF
sin $[0, \pi/4[$	1	711	13.9%	24.2 ns	49 s	16.6
	2	311	6.1%	21.2 ns	22 s	16.9
	3	265	5.2%	22.3 ns	16 s	15.2
	4	280	5.5%	24.8 ns	16 s	13.9
2^x	1	677	13.2%	25.7 ns	44 s	16.6
	2	376	7.3%	24.4 ns	25 s	16.0
	3	283	5.5%	22.8 ns	16 s	15.4
	4	295	5.8%	21.2 ns	18 s	15.6
$\frac{1}{x}$	1	821	16.0%	27.6 ns	69 s	17.4
	2	490	9.6%	28.1 ns	36 s	15.7
	3	474	9.3%	27.1 ns	27 s	15.4

perfect multiplier to implement a linear scheme will remove only the TO_is, which accounts for less than half the total size.

5.4 FPGA Implementation Results

In this section, the target architecture is a Virtex-II 1000 FPGA from Xilinx (XC2V1000-fg456-5). All the synthesis, place, and route processes have been performed using the Xilinx ISE XST 5.2.03i tools. The generated VHDL operators have been optimized for area with a high effort (the results are very close using a speed target). Area is measured in number of slices (two LUTs with four address bits per slice in Virtex-II devices), there are 5,120 available slices in a XC2V1000. The delay is expressed in nanoseconds. We also report the delay of the operator and its complete synthesis time (including place and route optimizations) T_{synth}. The compression factor CF is the ratio number of bits/number of LUTs; it measures the additional compression capabilities of the logical optimizer. In the target FPGAs, look-up tables may hold 16 bits, so a CF larger than 16 indicates such a size improvement.

Table 5 presents some synthesis results for the functions described in Table 1. The time required to compute the optimal decomposition (using the algorithm presented in Section 4.2) is always negligible compared to T_{synth}.

Table 6 details the evolution of area and delay with respect to input size for the sine function. Note that, in the Xilinx standard sine/cosine core [18], which uses a simple tabulation, the input size is limited to 10 bits, meaning an 8-bit table after quadrant reduction.[3]

Some results for 24-bit are also given in Table 7, showing that 24-bit precision is the practical limit of multipartite methods on such an FPGA. The economical limit, of course, is probably less than 20 bits, as suggested by Table 6.

These results show that, when the number of TO_is m increases, the operator size (the number of LUTs) decreases. The size gain is significant when we use a tripartite method ($m = 2$) instead of a bipartite one ($m = 1$). For larger values of m, this decrease is less important. Sometimes, a slight increase is possible for even larger values of m (e.g., $m = 3$ to $m = 4$ for the sine and 2^x function). This is due to the extra cost of the adder with an additional input, the XOR gates, and the sign extension mechanism that is not

3. Using on-chip RAM blocks, the simple table approach allows up to 16 bits, meaning $w_I = 14$ after quadrant reduction.

TABLE 6
Virtex_II FPGA Implementation of the Sine Function for Various Sizes

f	w_I	m	slices	delay	T_{synth}	CF
sin $[0, \pi/4[$	8	2	19	16.6 ns	7 s	5.6
	12	3	76	18.0 ns	12 s	9.5
	16	4	280	24.8 ns	18 s	13.9
	20	5	1209	34.5 ns	96 s	16.5
	24	5	4954	43.0 ns	660 s	19.1

compensated by the tables size reduction. This is also reflected in the operator delay.

The compression factor CF is more or less constant (just a slight decrease with m). This fact can be used to predict the size after synthesis on the FPGA from the table size in bits. As each LUT in a Virtex FPGA can only store 16 bits of memory, we can deduce from these tables that the synthesizer performs some logic optimization inside each table. The compression factor decreases when m increases because the minimization potential is smaller on small tables than on larger ones. The synthesis time also decreases when m increases.

We investigated in [19] the use of ad hoc table-compression techniques. For this, we used JBits, a low-level hardware description language developed by Xilinx. Compression factors of up to 19 could be obtained for 16-bit and 20-bit sines at the expense of two months of development.

An important remark is that smaller operators turn out to be faster on FPGAs: Using a multipartite compression improves both speed and area.

5.5 Comparisons with Higher-Order Methods

Results for 24-bit operands should also be compared to the ATA architecture published by Wong and Goto for this specific case [15]. They use six tables for a total of 868,352 bits and, altogether, nine additions. Our results are thus both smaller and faster. However, it should be noted that five of the six tables in their architecture have the same content, which means that a sequential access version to a unique table should be possible (provided the issue of rounding is studied carefully). This sequential architecture would involve only about 16Kbits of tables, but it would be five times slower.

The remainder of this section compares with recently published methods involving multipliers. Such methods have to be used for $w_I > 24$ bits: If we consider that the maximum admissible table size is 2^{12} entries, this limit is reached by the multipartite approach for $w_I = 24$. Our aim here is to give a more quantitative idea of the domains of

TABLE 7
Virtex_II FPGA Implementation of the Sine Function (24-Bit)

f	m	slices	delay	T_{synth}	CF
sin $[0, \pi/4[$	1	640%	–	–	–
	2	170%	–	–	–
	3	105%	–	–	–
	4	101%	–	–	–
	5	4954	43.0 ns	660 s	19.1
	6	5004	36.1 ns	520 s	19.0

TABLE 8
Effect of Fine-Tuning on Virtex-II Implementation

f	m	automatic slices	automatic delay	fine-tuned slices	fine-tuned delay
sin on $[0, \pi/4[$	4	280	24.8 ns	**258**	**24.6** ns

TABLE 9
Comparison with Defour et al.'s Approach

f	$w_I = w_O$	table size in [16]	multipartite size
sin $[0, \pi/4[$	15	3536	4608 ($m = 4$)
	20	15936	40704 ($m = 4$)
	24	71776	210728 ($m = 4$)

relevance of the various methods. Of course, this will depend on the function and on the target hardware.

The method published recently by Defour et al. uses two small multipliers in addition to tables and adders [16]. Note that recent FPGAs include small $18 \times 18 \rightarrow 35$-bit multipliers which can be used for Defour et al.'s architecture. This method has several restrictions: It is more rigid than the multipartite approach as it uses a decomposition of the input word into five subwords of the same size. As a consequence, for some functions, it is unable to provide enough precision for faithful rounding when $w_O = w_I$. Table 9 gives some comparisons of this method with the multipartite approach. We chose $m = 4$ so that the number of additions is the same in both approaches. According to this table, a multipartite approach will be preferred for precisions smaller than 15 bits and Defour et al.'s approach will be preferred for precisions greater than 20 bits, as far as size only is considered. For $w_I = 15$, Defour et al.'s tables are still smaller, but the size of the multipliers will compensate, so multipartite should be both smaller and faster. If speed is an issue, the delay of the multipliers will play in favor of multipartite tables.

The architecture by Piñeiro et al. [17] involves a squarer and a multiplier and 12,544 bits of tables for 24-bit $1/x$. An FPGA implementation sums up to 565 slices, which is only slightly more than our 16-bit implementation at 474 slices. This again suggests that multipartite methods are not relevant for more than 16 bits of precision, as far as size only is concerned.

Finally, we have recently compared a second-order approach using two multipliers and the multipartite approach on the specific case of addition/subtraction in the logarithm number system. The functions involved are then $\log_2(1 + 2^x)$ and $\log_2(1 - 2^x)$ and a restricted form of faithful rounding is used. In this case, we have only one point of comparison, corresponding to about 12 bits of precision. The multipartite approach is better both in terms of speed and area in this case [8].

In all these cases, it should be noted that the simplicity and generality of the multipartite approach may be a strong argument. Implementing a new function is a matter of minutes from the start down to VHDL code. This code is then efficiently synthesized, at least on FPGAs, because it only contains additions. Comparatively, approaches relying on multiplications need much more back-end work, typically requiring to design custom operators as Piñeiro et al. does.

5.6 Limits of the Method

5.6.1 Nonmonitonicities

Our approach maximizes the approximation error (within the bounds of faithful rounding) to minimize the hardware cost. This has the drawback of entailing nonmonotonicities at some of the borders between intervals: See, for instance, Fig. 8 around $x = x_3$. These nonmonotonicities are never bigger than one ulp thanks to faithful rounding. It is a problem of all the faithful approximation schemes, but the multipartite method as presented makes it happen quite often.

The subject of monotonicity in the context of bipartite tables has been studied by Iordache and Matula [7]. They reverse-engineered the AMD K6-2 implementation of fast reciprocal and reciprocal square root instructions, part of the 3D-Now instruction set extensions. They found that bipartite approximations were used, that the reciprocal was monotonic, and that the reciprocal square root was not. They also showed that the latter could be tuned to become monotonic, at the expense of a larger table size (7.25 KB instead of 5.5). This tuning involves increasing the output size of the TIV and an exhaustive exploration of what value these extra bits should take.

In general, if monotonicity is an important property, it can be enforced simply in a multipartite approximation by using appropriate slopes and TIV values. For instance, monotonically increasing functions with decreasing derivatives (as on our figures) may use the slope on the right of the interval instead of the middle, ensuring that the approximation is monotonic. This means a larger maximum approximation error, however. Rounding errors can then be kept within bounds that ensure monotonicity by increasing g as in [7]. All this entails increased hardware cost. A general and systematic study of this question remains to be done.

5.6.2 Infinite Derivative

There are also functions for which this methodology will not work. The square root function on $[0, 1[$, for example, although it may perfectly be stored in a single table, has an infinite derivative in 0 which breaks multipartite methods. We have never seen any mention of this problem in the literature, either. One solution in such cases is to split the input interval into two intervals $[0, 2^{-\zeta}[$ (on which the function is tabulated in a single table) and $[2^{-\zeta}, 1[$, where the multipartite method is used. The optimal ζ can probably be determined by enumeration.

6 Conclusion

We have presented several contributions to table-lookup-and-additions methods. The first one is to unify and generalize two complimentary approaches to multipartite tables by Stine and Schulte and by Muller. The second one is to give a method for optimizing such bipartite or multipartite tables which is more accurate than what could be previously found in the literature. Both these improvements have been implemented in general tools that can generate optimal multipartite tables from a wide range of specifications (input and output accuracy, delay, area).

These tools output VHDL which has been synthesized for Virtex FPGAs. Our method provides up to 50 percent smaller solutions than ones of the best literature results. This paper also discusses the limits of this approach. By comparing with higher-order methods, we conclude that multipartite methods provide the best area/speed tradeoff for precisions from 8 to 16 bits.

With the observation that multipartite methods are optimal among first-order methods, this paper leaves little room for improvement in such methods. Future work should now aim at providing methods for exploring the design space of higher-order approximations with the same ease as multipartite methods allow for first-order approximations.

Acknowledgments

This work was partially supported by an ACI grant from the French Ministry of Research and Education, and by Xilinx University Programme. The authors would like to thank the anonymous referees for many useful remarks and comments.

References

[1] D. Das Sarma and D. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 17-28, 1995.
[2] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 10-16, 1995.
[3] M. Schulte and J. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 842-847, Aug. 1999.
[4] J. Stine and M. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. 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] J.-M. Muller, *Elementary Functions, Algorithms and Implementation.* Boston: Birkhauser, 1997.
[7] C. Iordache and D.W. Matula, "Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow!" *Electronic Notes in Theoretical Computer Science,* vol. 24, 1999.
[8] J. Detrey and F. de Dinechin, "A VHDL Library of LNS Operators," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers,* Oct. 2003.
[9] F. de Dinechin and A. Tisserand, "Some Improvements on Multipartite Table Methods," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 128-135, June 2001.
[10] D. Matula, "Improved Table Lookup Algorithms for Postscaled Division," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 101-108, June 2001.
[11] F.J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. Computers,* vol. 37, no. 2, Feb. 1988.
[12] D.M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System," *IEEE Trans. Computers,* vol. 39, no. 11, Nov. 1990.
[13] J.N. Coleman and E.I. Chester, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 702-715, July 2000.
[14] B. Lee and N. Burgess, "A Dual-Path Logarithmic Number System Addition/Subtraction Scheme for FPGA," *Proc. Field-Programmable Logic and Applications,* Sept. 2003.
[15] W. Wong and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 453-457, Mar. 1995.
[16] D. Defour, F. de Dinechin, and J.-M. Muller, "A New Scheme for Table-Based Evaluation of Functions," *Proc. 36th Asilomar Conf. Signals, Systems, and Computers,* Nov. 2002.
[17] J.A. Piñeiro, J.D. Bruguera, and J.-M. Muller, "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 40-47, June 2001.
[18] *Sine/Cosine Lookup Table V4.2,* Xilinx Corp., Nov. 2002, www.xilinx.com/ipcenter/.
[19] F. de Dinechin and J. Detrey, "Multipartite Tables in Jbits for the Evaluation of Functions on FPGAs," *Proc. IEEE Reconfigurable Architecture Workshop, Int'l Parallel and Distributed Symp.,* Apr. 2002.

Florent de Dinechin received the DEA from the École Normale Supérieure de Lyon (ENS-Lyon) in 1993 and the PhD degree from the Université de Rennes 1 in 1997. After a postdoctoral position at Imperial College, London, he is now a permanent lecturer at ENS-Lyon in the Laboratoire de l'Informatique du Parallèlisme (LIP). His research interests include computer arithmetic, software and hardware evaluation of functions, computer architecture, and FPGAs. He is a member of the IEEE and the IEEE Computer Society.

Arnaud Tisserand received the MSc degree and the PhD degree in computer science from the École Normale Supérieure de Lyon, France, in 1994 and 1997, respectively. He is with the French National Institue for Research in Computer Science and Control (INRIA) and the Laboratoire de l'Informatique du Parallèlisme (LIP) in Lyon, France. He teaches computer architecture and VLSI design at the École Normale Supérieure de Lyon, France. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE.