Some Improvements on Multipartite Table Methods

Florent de Dinechin , Arnaud Tisserand
École Normale Supérieure de Lyon - INRIA
46 allée d'Italie, 69364 Lyon, France
{Florent.de.Dinechin, Arnaud.Tisserand}@ens-lyon.fr

Abstract

This paper presents an unified view of most previous table-lookup-and-addition methods: bipartite tables, SBTM, STAM and multipartite methods. This new definition allows a more accurate computation of the error entailed by these methods. Being more general, it also allows an exhaustive design space exploration which has been implemented, and leads to tables smaller than previously published ones by up to 50%. Some results have been synthesised for Virtex FPGAs, and are discussed in this paper.

1 Introduction

Table-lookup-and-addition methods, such as the bipartite method, have been the subject of much recent attention [4, 1, 5, 6, 3]. They allow to compute commonly used functions with low accuracy (currently up to 24 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, and also for providing initial seed values to iterative methods such as the Newton-Raphson algorithms for division and square root [2] which are commonly used in the floating-point units of current processors.

This paper clarifies some of the cost and accuracy questions which are incompletely formulated in previous papers. It also unifies two complimentary approaches to multipartite tables, by Stine and Schulte [6], and Muller [3]. It completely defines the implementation space for multipartite tables, which allows us to provide a methodology for selecting the best implementation that fullfills arbitrary accuracy and cost requirements. This methodology has been implemented and is demonstrated on a few examples.

After some notations and definitions in Section 2, Section 3 presents previous table-lookup-and-addition methods, and unifies them as a general multipartite method. Section 4 shows how to explore the design space in order to select the best multipartite implementation full-filling a given accuracy requirement. Section 5 defines the content of tables. Section 6 presents our implementation and its results. Section 7 discusses the results and concludes.

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. The reader should keep in mind that all the following work can (and must) be straightforwardly extended to arbitrary closed, semi-closed or open intervals (the reciprocal, for example, is typically computed on $[1, 2[\rightarrow]0.5, 1]$). A general presentation would degrade readability without increasing the interest of the paper. Our implementation, however, allows such arbitrary combinations.
- 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 $\{0, ..., 2^p - 1\}$ 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 $\{0, ..., 2^{w_I} - 1\}$, and we will express the real number $x \in [a, b[$ that it codes by $x = a + (b - a)X/2^{w_I}$.

2.2 Errors

Usually, three different kinds of error affect the global error of an evaluation of f:

- The input discretisation (or quantisation) error measures the fact that an input number usually represents a small interval of values centered around this number.

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 IEEE

- 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.

- The output discretisation (or rounding) error measures the difference between the approximated function and the closest machine-representable value.

In the following, we will ignore the question of input discretisation, by considering that an input number only represents itself as an exact mathematical number. A discussion about quantisation errors should come before or after the implementation presented here.

3 Table-and-addition methods

3.1 The bipartite method

First presented by Das Sarma and Matula [4] in the specific case of the reciprocal function, and generalised by Schulte and Stine [5, 6] and Muller [3], this method consists in approximating the function by affine segments, as illustrated on Figure 1.

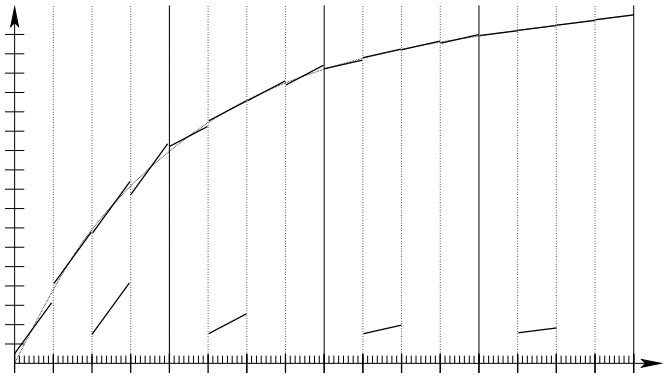

Figure 1. The bipartite approximation

The 2^α segments are selected by the α most significant bits of the input word. Instead of tabulating the 2^{w_I} values of the function, it is possible, for each segment, to tabulate one initial value, and to construct the other values by adding, to this initial values, an offset defined by the $w_I - \alpha$ least significant bits of the input word.

The idea behind the bipartite method is to group the segments into 2^γ (with $\gamma < \alpha$) larger intervals (4 on the figure) such that the slope of the segments is considered constant on each larger interval. 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 $2^{\alpha+\beta}$.

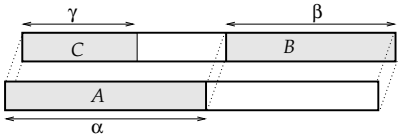

Figure 2. Bipartite input word decomposition

In all the following, we will call *Table of Initial Values (TIV)* the table that stores the initial points of each segment. This table will be addressed by a sub-word 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 sub-words of the input word: C (the γ most significant bits) and B (the β least significant bits). Figure 2 depicts this decomposition of the input word.

Previous authors [3, 6] 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 [5] that it is possible to exploit the symmetry of the segments on each small interval (see Figure 3, which is a zoom view on Figure 1) to halve the size of the TO: They store in the TIV the value of the function in the middle of the small interval, and in the TO the offsets for a half segment. 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* [5]).

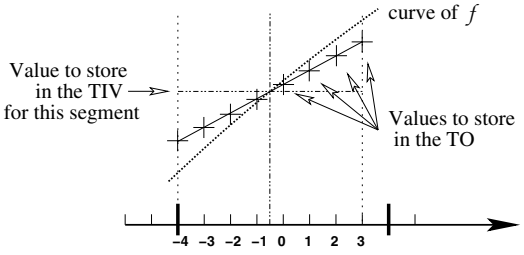

Figure 3. Segment symmetry

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 **IEEE**

3.3 Multipartite methods

In another paper [6], Schulte and Stine 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 sub-word B can be decomposed (as seen on Figure 4) into m sub-words 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 discretisation errors (one per table) which will sum up to a larger global discretisation error, unless the smaller tables have a bigger output accuracy (and thus are bigger). We will formalise this later.

Schulte and Stine have termed this method STAM, for *Symmetric Table and Addition Method*. It can still be improved: Note in Equation (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 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 [3] 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 $2p+1$ sub-words X_1,...,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 sub-words.

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 doesn't address the rounding issue.

3.4 A general multipartite method

Investigating what is common to Schulte and Stine's STAM and Muller's multipartite methods leads us to define a decomposition into sub-words that generalises both:

- The input word is split into two sub-words A and B of respective sizes α and β with $\alpha + \beta = w_I$ (see Fig. 4).
- The most significant sub-word A addresses the TIV.
- The least significant sub-word B will be used to address $m \geq 1$ TOs.
  - B will in turn be decomposed into m sub-words B_0,..., B_{m-1}, the least significant being B_0.
  - A sub-word B_i starts at position p_i and consists of β_i bits (see Fig. 4). We have $p_0 = 0$ and $p_{i+1} = p_i + \beta_i$.
  - The sub-word B_i is used to address the TO_i, along with a sub-word 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.

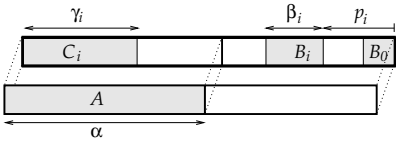

Figure 4. Multipartite input word decomposition

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 4.2. 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, Schulte and Stine's STAM [6] is a multipartite decomposition where all the C_i's are equal, and Muller's multipartite approach [3] is a specific case of our decomposition where the γ_i are multiples of constant integers.

It should be clear that general decompositions are more promising than Stine and Schulte's in that they allows to reduce the accuracy of the slope involved in the TOs (and thus

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 IEEE

their size). They are also more promising than Muller's, as they are more flexible (for example the size of the input word needs not be a multiple of some $2p+1$). Our methodology will also be slightly more accurate than both in the error analysis. Section 6 will show these improvements.

3.5 Other table-lookup and addition methods

In addition to the previous works, we should also mention Wong and Goto [7] who have presented a subtle approximation method which takes into consideration second-order terms, and leads to an architecture involving additions before and after the table lookups. Our generalised multipartite method, however, will prove better both area and delay, as it will be exposed in Section 6.

Of interest is also the work from Hassler and Tagaki [1], who have presented a method based on partial product arrays (PPAs) which is radically different from all the previous methods based on Taylor/linear approximation. Stine and Schulte show in [6] that their methods are more area and time efficient, so we do not elaborate here further.

4 Choosing a multipartite decomposition

Having defined in Section 3.4 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 term of speed or area) that full-fills the accuracy requirement known as *faithful rounding*, which will be presented in the following section. Section 4.2 shows that the approximation error can be computed accurately with only very few operations, which allows us to define in Section 4.3 an efficient exploration algorithm for an arbitrary cost function.

4.1 Faithful rounding and guard bits

Like previous authors, we want to implement the function f with faithful rounding: The computed result should be one of the two machine numbers closest to the mathematical result. In other words, the result should differ from the true result by less than one unit in the last place. Therefore we define the maximum output error as the value of the least significant bit of the output: $\epsilon_f = (d-c)2^{-w_O}$. We thus need to ensure that the total implementation error will be smaller than ϵ_f. For this purpose, we will need to compute with an internal precision which is higher than the final precision: We will add g "guard" bits to the tables to ensure this internal precision.

The final error will then be the sum of three terms:

- A mathematical approximation error, whose maximum value will be noted ϵ_{approx} and will be computed exactly in Section 4.2.
- The rounding error when filling each table, $\epsilon_{rt} \leq (m+1)\epsilon_t$ where $(m+1)$ is the number of tables and ϵ_t is the maximum rounding error when filling one table: $\epsilon_t = (d-c)2^{-w_O-g-1}$.
- The rounding error when rounding the sum of the tables to w_O output bits. Its maximum value is $\epsilon_{rf} = (d-c)2^{-w_O-1}$ in a straightforward implementation, but a trick due to Das Sarma and Matula [4] allows to improve it to $\epsilon_{rf} = (d-c)(2^{-w_O-1} - 2^{-w_O-g-1})$. This trick will be presented in Section 5.

Finally, the condition to ensure faithful rounding, $\epsilon_{rt} + \epsilon_{rf} + \epsilon_{\text{approx}} < \epsilon_f$ is rewritten $g > -w_O - 1 + \log_2((d-c)m) - \log_2((d-c)2^{-w_O-1} - \epsilon_{\text{approx}})$.

As the next section shows, ϵ_{approx} is a function of the decomposition $\mathcal{D}$ of the input word. If $\epsilon_{\text{approx}} \geq (d-c)2^{-w_O-1}$, $\mathcal{D}$ is unable to provide the required output accuracy. Otherwise the previous inequation gives us the number g of extra bits that ensures faithful rounding:

$$g = \left\lceil -w_O - 1 + \log_2 \frac{(d-c)m}{(d-c)2^{-w_O-1} - \epsilon_{\text{approx}}} \right\rceil \quad (2)$$

Our experiments show that it is very often possible to decrease this value by one and still keep faithful rounding, but we are unable to provide a solid argument for that.

4.2 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 discretisation. 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 $s(Ci)$. Figure 5 (another zoom of Figure 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

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 IEEE

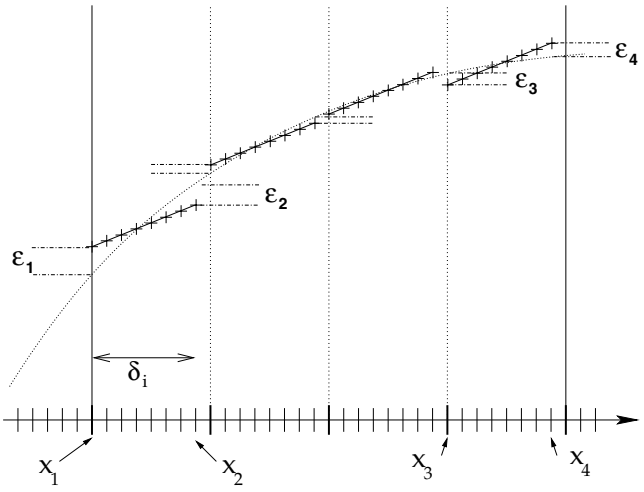

Figure 5. Computing the approximation error

the segment slope is constant. The minimum $\epsilon_i^{\mathcal{D}}(C_i)$ of this maximum error is obtained when

$$\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4 \;=\; \epsilon_i^{\mathcal{D}}(C_i) \tag{3}$$

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:

$$s_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) + f(x_4) - f(x_3)}{2\delta_i} \tag{4}$$

$$\epsilon_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) - f(x_4) + f(x_3)}{4} \tag{5}$$

where (using the notations of Section 2.1)

$$\begin{aligned}
\delta_i &= (b-a)2^{-w_I+p_i}(2^{\beta_i}-1) && (6)\\
x_1 &= a + (b-a)2^{-\gamma_i}C_i && (7)\\
x_2 &= x_1 + \delta_i && (8)\\
x_3 &= x_1 + (b-a)(2^{-\gamma_i} - 2^{-w_I+p_i+\beta_i}) && (9)\\
x_4 &= x_3 + \delta_i && (10)
\end{aligned}$$

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:

$$\epsilon_i^{\mathcal{D}} = \max(|\epsilon_i^{\mathcal{D}}(0)|, |\epsilon_i^{\mathcal{D}}(2^{\gamma_i}-1)|) \tag{11}$$

In practice, it is easy to compute this approximation error by implementing equations (5) to (11). Altogether it represents a few floating-point operations per TO_i.

[1] Not surprisingly, the slope that minimises the error is the average value of the slopes on the borders of the interval. Previous authors considered the slope in the midpoint of this interval.

4.3 An algorithm for choosing a decomposition

1. Choose the number of tables m. A larger m means smaller tables, but more additions.
2. Enumerate the decompositions.
3. For each decomposition $\mathcal{D}$, compute the approximation errors entailed by each TO_i as seen in Section 4.2, and sum them to get $\epsilon_{\text{approx}}^{\mathcal{D}} = \sum_{i=0}^{m-1} \epsilon_i^{\mathcal{D}}$. Keep only a set of possible decompositions for which this error is smaller than the maximum admissible error ϵ_f.
4. For each possible decomposition, compute the number g of extra accuracy bits using Equation (2), and evaluate the size and speed of the implementation. Section 4.4 gives formulas for the memory size in bits.
5. Synthesise 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 TOi 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 small 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 γ_{min} such that for any $\gamma_i \leq \gamma_{\text{min}}$ this error is larger than the required output precision. These $\gamma_{\text{min}}(p_i, \beta_i)$ 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 $\gamma_{\text{min}} < \gamma_i < \alpha$. 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.

4.4 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.

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 **IEEE**

The size in bits of the TIV is simply $2^{\alpha}(w_O + g)$. The TO$_i$s have a smaller range than the TIV: Actually the range of 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 $C_i = 2^{\gamma_i} - 1$:

$$r_i = \max(|s_i(0) \times \delta_i|, |s_i(2^{\gamma_i} - 1) \times \delta_i|) \quad (12)$$

The number of output bits of TO$_i$ is therefore

$$w_i = \lceil w_O + g + \log_2(r_i/(d-c)) \rceil \quad (13)$$

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 multi-operand adders are the subject of current investigation. Section 6 will present some results for Virtex FPGAs, showing that the bit counts presented above allows a predictive enough evaluation of the actual costs.

5 Filling the tables

5.1 The mathematical values

An initial value TIV(A) provided by the TIV for an input sub-word A will be used on an interval $[x_l, x_r]$ defined (using the notations of Sections 2.1 and 4.2) by:

$$x_l = a + (b-a)2^{-\alpha}A \quad (14)$$

$$x_r = x_l + \sum_{i=0}^{m-1} \delta_i \quad (15)$$

On this interval, each TO$_i$ provides a constant slope, as its C_i is a sub-word of A. The approximation error, which is the sum of the $\epsilon_i^{\mathcal{D}}(C_i)$ defined by Equation (5), 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) is:

$$\widetilde{\text{TIV}}(A) = \frac{f(x_l) + f(x_r)}{2} \quad (16)$$

The TO$_i$ values before rounding are (see Figure 3):

$$\widetilde{\text{TO}_i}(C_i B_i) = s(C_i) \times 2^{-w_I+p_i}(b-a)(B_i + \frac{1}{2}) \quad (17)$$

5.2 Rounding considerations

This section reformulates the techniques employed by Stine and Schulte in [6] and using an idea that seems to appear first in the paper by Das Sarma and Matula [4].

The purpose is to fill our tables in such a way to ensure that their sum (which we compute on $w_O + g$ bits) always has an implicit 1 as its $(w_O + g + 1)$-th bit. This reduces the final rounding error from $\epsilon_{rf} = 2^{-w_O-1}$ to $\epsilon_{rf} = 2^{-w_O-1} - 2^{-w_O-g-1}$.

To achieve this trick, we remark that there are two ways to round a real number to $w_O + g$ bits with an error smaller than $\epsilon_t = 2^{-w_O-g-1}$. 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$_i$s. 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 TOis are

$$\text{TO}_i(C_i B_i) = \left\lfloor \frac{2^{w_O+g}}{d-c} \widetilde{\text{TO}_i}(C_i B_i) \right\rfloor \quad (18)$$

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

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m-1}{2} + 2^{g-1} \right\rfloor \quad (19)$$

and if m is even:

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m}{2} + 2^{g-1} \right\rfloor \quad (20)$$

6 Implementation and results

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 synthesisable VHDL.

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 IEEE

n	f	m	α	β	α_i	β_i	tables	size	ref size
16	sin	1	10	6	5	6	$17.2^{10} + 7.2^{10}$	24576	32768
		2	8	8	7,4	3,5	$19.2^{8} + 10.2^{9} + 8.2^{8}$	12032	20480
		3	8	8	7,6,4	2,3,3	$18.2^{8} + 9.2^{8} + 7.2^{8} + 4.2^{6}$	8960	17920
		4	8	8	7,6,4,4	2,2,2,2	$19.2^{8} + 10.2^{8} + 8.2^{7} + 6.2^{5} + 4.2^{5}$	8768	na
	2^x	1	10	6	5	6	$16.2^{10} + 6.2^{10}$	22528	24576
		2	8	8	7,4	3,5	$17.2^{8} + 9.2^{9} + 6.2^{8}$	10496	14592
		3	8	8	7,6,4	2,2,4	$17.2^{8} + 9.2^{8} + 7.2^{7} + 5.2^{7}$	8192	13568
		4	8	8	7,6,5,4	2,2,2,2	$18.2^{8} + 10.2^{8} + 8.2^{7} + 6.2^{6} + 4.2^{5}$	8704	na
	$\frac{1}{x}$	1	10	5	7	5	$16.2^{10} + 6.2^{11}$	28672	24576
		2	9	6	7,6	3,3	$18.2^{9} + 9.2^{9} + 6.2^{8}$	15360	16896
		3	9	6	8,7,5	2,2,2	$17.2^{9} + 8.2^{9} + 6.2^{8} + 4.2^{6}$	14592	15872
24	sin	1	15	9	8	9	$25.2^{15} + 10.2^{16}$	1474560	1474560
		2	13	11	10,7	4,7	$27.2^{13} + 14.2^{13} + 10.2^{13}$	417792	581632
		3	12	12	11,9,6	3,4,5	$27.2^{12} + 14.2^{13} + 12.2^{12} + 8.2^{10}$	282624	425984
		4	12	12	11,10,9,6	2,2,3,5	$27.2^{12} + 14.2^{12} + 12.2^{11} + 10.2^{11} + 8.2^{10}$	221184	360448
		5	12	12	11,10,9,8,6	2,2,2,3,3	$27.2^{12} + 14.2^{12} + 12.2^{11} + 10.2^{10} + 8.2^{10} + 5.2^{8}$	212224	356352

Table 1. Best decomposition characteristics and table sizes for 16-bit and some 24-bit operands

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 Figure 6 for an example of output).

The ability to actually fill the tables also helps to characterise the real quality of the final approximation. For instance, we can point some small problems such as the non-monotonicities (see for instance Fig. 5 around $x = x_3$). These non-monotonicities are never bigger than one LSB thanks to faithful rounding, but we have never seen any mention to this problem in the literature.

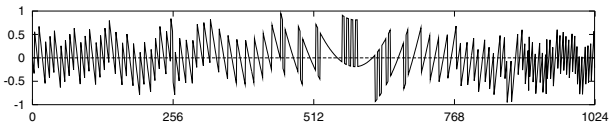

Figure 6. Measured error (10-bit sine and $m = 2$)

6.1 Comparison with previous works

Table 1 presents the best decomposition obtained for 16-bit and some 24-bit operands for a few functions. In this table, we compare our results with the best known results from the work of Schulte and Stine [5, 6]. We can notice a size improvement up to 50%. 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.

Results for 24-bit operands should also be compared to the ATA architecture published by Wong and Goto for this specific case [7]. They use 6 tables for a total of 513536 bits, and altogether 9 additions. Our results are thus both smaller and faster. However it should be noted that 5 of the 6 tables in their architecture have the same content, which means that a sequential access version to a unique table is possible (provided the issue of rounding is studied carefully). This sequential architecture would involve only 149376 bits of tables, but it would be five times slower.

6.2 FPGA implementation results

The target architecture is the Virtex device family from Xilinx. More precisely, we use a XCV400 FPGA with a speed grade of -4 (the slowest one). The synthesised operator is considered as a combinatorial block. No pipelining is performed in this work (it is a future work). All operators have been synthesised using Synplify, the place and route operations are performed using Xilinx tools.

Table 2 presents some implementation results of multipartite tables. 16-bit values have been considered for sin and 2^x functions. For each function, all possible values for the number of TOis m have been considered with respect to accuracy requirements. The reported metrics are the number of LUTs (look up tables, the basic cells of the FPGA), the operator delay, the synthesis time T_{synth} and a compres-

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 **IEEE**

sion factor CF. The compression factor is the ratio number of bits / number of LUTs, it measures the compression capabilities of the optimiser. Using the algorithm presented in Section 4.3, the time required to compute the optimal decomposition is always negligible compared to T_{synth}.

f	m	#LUTs	delay	T_{synth}	CF
sin	1	1375	43 ns	122 s	19.4
	2	799	42 ns	50 s	16.4
	3	628	39 ns	34 s	15.6
	4	664	41 ns	37 s	14.3
2^x	1	1839	43 ns	206 s	16.7
	2	959	39 ns	67 s	16.6
	3	864	42 ns	52 s	14.5
	4	801	39 ns	54 s	14.4

Table 2. Virtex FPGA implementation (16-bit)

These results show that when the number of TOis 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. $m = 2$ to $m = 3$ for the sine 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 compensated by the tables size reduction. We can notice a similar behavior for the operator delay.

The synthesis time decreases when m increases. This is due to the fact that the synthesis tool optimises the tables size using logical minimisation tools. The compression factor CF, the number of bits / number of LUTs ratio, is more or less constant (just a slight decrease). This fact can be used to predict the size after synthesis on the FPGA from the table size in bits. From these tables we can deduce that the synthesiser perform some optimisation inside the table, because each LUT in a Virtex FPGA can only store 16 bits of memory (cf Xilinx documentation). We think that common small sub-words are shared over close words. It can lead to a compression factor larger than 16. The compression factor decreases when m increases because the minimisation potential is smaller on small tables than on larger ones. The low level optimisation of tables values will be one of our future work in this field.

7 Conclusion

We have presented several contributions to table-lookup-and-additions methods. The first one is to unify and generalise two complimentary approaches to multipartite tables, by Stine and Schulte, and Muller. The second one is to give a method for optimising 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 synthesised for Virtex FPGAs. Our method provides up to 50% smaller solutions than ones of the best literature results.

Future work includes completing the tools by allowing more accurate, technology-dependent area and speed estimations, reducing non-monotonicities, and investigating some low-level optimisations in the synthesis of the tables.

There are also functions for which this methodology will not work. It is easy to see that 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. The solution in such cases is to break 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. Our tool should accommodate such cases, as well as the case of arbitrary functions which do not satisfy the convexity hypothesis we have assumed in this paper.

References

[1] H. Hassler and N. Tagaki. Function evaluation by table look-up and addition. In S. Knowles and W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, 1995.

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

[3] J.M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, 1999.

[4] D. Das Sarma and D.W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995.

[5] M.J. Schulte and J.E. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, August 1999.

[6] J.E. Stine and M.J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2):167–177, 1999.

[7] W.F. Wong and E. Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3):453–457, March 1995.

Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)
1063-6889/01 $10.00 © 2001 **IEEE**