Multipartite Tables in JBits
for the Evaluation of Functions on FPGAs

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

Abstract

This paper presents the implementation, on Virtex FPGAs, of a core generator for arbitrary numeric functions in fixed-point format. The cores use the state-of-the-art multipartite table method, which allows input and output precisions in the range of 8 to 24 bits on current Virtex chips. The implementation uses the JBits API to embed elaborate optimisation techniques in the description of the hardware.

1 Introduction

Many FPGA applications require the evaluation of some unary functions such as trigonometric (sine/cosine, tangent...), algebraic (square or square root, cube or cubic root, ...) or transcendental (exponential or logarithm). The simplest implementation consists in tabulating all the values, which is impractical for precisions higher than a few bits, because the size of the table is exponential in the input size. There exist compact – but slow – implementations for some functions, such as CORDIC-like algorithms for trigonometric and exponential/logarithm. More general methods for function evaluation, based on polynomial approximation, rely on multipliers and are therefore less suited for implementation on multiplierless FPGAs (the study on recent FPGAs with hard multipliers, like the Virtex II series, is still to be done).

1.1 Table-based methods

Pioneered by Das Sarma and Matula [1], these methods reduce the size of a straightforward table implementation by exploiting the property of continuity of the function. The latest developments in this area is the multipartite method by Dinechin and Tisserand [2], for precisions of 8 to 24 bits. For precisions higher than 24 bits, Piñeiro *et al* recently demonstrated a method based on a second degree polynomial approximation using a dedicated squarer unit and a multiplier, which is more area-efficient and probably faster than the multipartite method [6].

This paper therefore focuses on the FPGA implementation of the multipartite method, for arbitrary functions, and for precisions of 8 to 24 bits. Our first contribution is to show that this method is well suited to FPGAs of the Xilinx family: In these circuits, the tables can be built efficiently out of the FPGA LUTs, and the built-in fast carry logic optimises the additions.

The second contribution of this paper is the use of the JBits framework [12] for implementing function evaluation cores on FPGAs.

1.2 JBits

JBits is an Application Programming Interface (API) developed by Xilinx for programming FPGAs of the 4000 and Virtex series. It comes as a set of Java classes. One of its main strong points is to allow a detailed, structural description of a circuit down to the CLB level. Its main drawback is that it imposes this structural description, even requiring the user to specify all the placement. However, the modern object-oriented features of Java, and the availability of a router [3] lighten the task to a very acceptable level.

Another strong point of the JBits framework is a tight and natural integration in the same language of the hardware and software parts of an application. Originally, this integration is mainly aimed at run-time reconfigurability: it allows to embed hardware objects in a software which reconfigures the hardware on the fly [7, 5, 8]. We did exploit this for testing our cores. However we also exploited it the opposite way, to embed elaborate optimisation techniques in the code for our hardware. We are thus able to produce cores where both the abstract architecture, and its FPGA implementation are optimised for the required function and precision.

To our knowledge, the function generator that we have developed is the most complex core ever developed in JBits, along with the DES implementation presented by Patterson [5]. We hope that our experience may be of benefit for prospective users of this technology.

In addition, this work involved developing JBits classes for the optimisation of arbitrary boolean functions, exploiting specifically the Virtex architecture. This paper describes the algorithms used in some detail, because these classes alone are probably of interest to the JBits community.

The work described in this paper is available for download under the GNU Public Licence at `www.ens-lyon.fr/LIP/Arenaire/News/JBits/`.

1.3 Outline of the paper

Section 2 describes the multipartite method used to generate architectures for arbitrary numeric functions. Section 3 discusses the JBits implementation details, focusing on the opportunities and constraints of this framework. Section 4 describes and comments the results of this methodology. Section 5 draws conclusions and suggests future works.

2 Multipartite function evaluation

2.1 The bipartite method

First presented by Das Sarma and Matula [1] in the specific case of the reciprocal function, this method consists in approximating the function by affine segments, as illustrated on Figure 1.

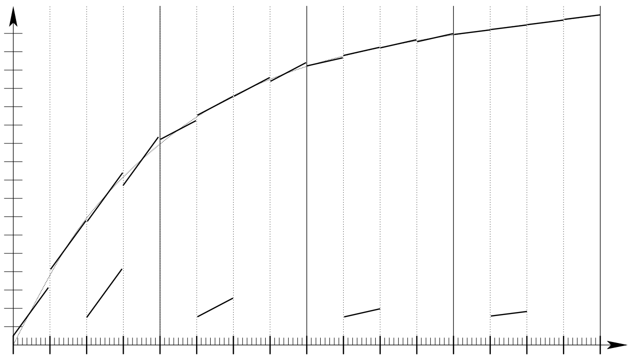


Figure 1. An example of bipartite approximation for 6-bit input: $w_I = 6$, $\alpha = 4$, $\gamma = 2$, $\beta = 2$.

The 2^α segments are indexed by the α most significant bits of the input word. To compute the 2^{w_I} values of the function (where w_I is the width of the input word), it is possible, for each segment, to tabulate one initial value, and to construct the other values of the segment by adding an offset, computed by a linear approximation using the $\beta = w_I - \alpha$ 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 Figure 1) such that the slope of the segments is considered constant on each larger interval. Thus there are only 2^γ tables of offsets, each containing 2^β offsets. Altogether, one needs to store $2^\alpha + 2^{\gamma+\beta}$ values, instead of $2^{\alpha+\beta}$ for a plain table, at the cost of an addition. The corresponding architecture is depicted on Figure 2.

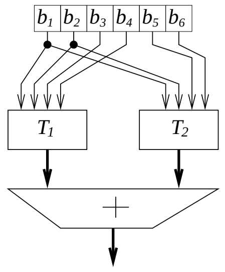


Figure 2. The corresponding architecture.

2.2 The multipartite method

It is possible to exploit the bipartite idea more than one time, splitting the input word into more subwords, and replacing the tables with even smaller ones. Schulte and Stine [11] and Muller [4] independently found two different ways to do it, and Dinechin and Tisserand [2] unified both approaches in an algorithm that explores the whole implementation space, leading to minimal table sizes. The typical multipartite architecture is presented on Figure 3. The algorithm presented in [2] (too complex to be detailed here) ensures that the cumulated approximation and rounding errors sum up to less than one LSB.

Precision	8 bits	12 bits	16 bits	20 bits
Total table size	224	1,552	8,960	50,176
Adders	2	3	3	3
Plain table size	2,048	49,152	1,048,576	20,971,520

Table 1. Hardware cost of the multipartite implementation of the sine function on $[0, \pi/4]$, versus plain table size (sizes in bits).

Table 1 presents some table reductions achieved by this algorithm. To measure the significance of this method, one should note that the sine cores offered by Xilinx, which use

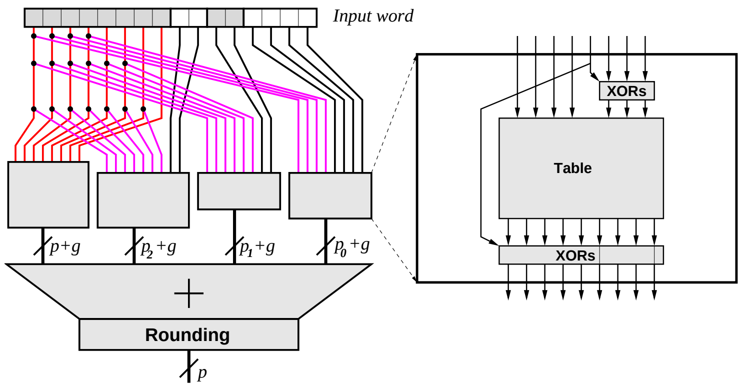


Figure 3. A multipartite architecture. The output of the tables are summed by a multi-operand adder. The XORs are a trick due to Schulte and Stine [11] that allows to halve the size of each table: Exploiting the symmetry of each segment to the function (see Figure 1), one of the input bit can be considered as a sign bit.

a plain table approach, have therefore a precision limit of 10 bits due to their size. In comparison, Figure 9 shows that a 16 bit sine core using our method occupies a small fraction of a Virtex 1000.

2.3 Multipartite tables and FPGAs

The initial implementation of the multipartite method was able to output VHDL which was then synthesised with Leonardo. Among the interesting results, it was remarked that:

- Architectures like that shown on Figure 3 lend itself to efficient implementation on Virtex devices, thanks to their LUT-based structure and the availability of fast adder circuitry;
- The size compression is so drastic that usually, it also entails a speed improvement over the plain table implementation, in spite of the adders;
- The VHDL synthetiser was able to compress the tables even further using logic optimisation techniques. This was measured as a *bits per LUT* factor [2] which could be as high as 18 (a LUT holds 16 bits, and some of them are used as multiplexers or adders).

These observations suggested that a specific core implementation should be developed to improve on these results. The JBits framework was chosen for this purpose. This is the subject of the rest of this paper.

3 JBits implementation

This section presents our JBits implementation of the previous multipartite method, first describing the general structure of the core, then focusing on a table compression heuristic and concluding with floorplanning considerations.

3.1 Overview of the architecture

Looking back at Figure 3 and Table 1, it can be seen that most area of the core will be dedicated to lookup tables, with the multi-operand adder and the rows of XORs occupying little area.

In the multipartite method, we need to add two to five values. In Virtex devices, the best option for a multi-operand adder of this kind is also the simplest: A row of simple adders using the built-in *fast carry logic*. This way the multi-operand adder is a convenient rectangular area. The XORs are also naturally built as columns of LUTs[1].

The rest of this section focuses on implementing a look-up table. Two JBits classes have been written for this purpose. The first one , described in Section 3.2, implements uncompressed, and therefore regularly placed and theoretically faster [10] tables. The second one, described in Section 3.3, compresses the table using Virtex-specific binary optimisation techniques. Its placement is no longer regular, which turns out to be a drawback in terms of JBits implementation : We had to write an ad-hoc placer, described in Section 3.4.

3.2 Uncompressed tables

To build a table addressed by w_I bits, we first consider independently each output bit, and fill the leaves of a multiplexer tree with the values of the bit according to the address. The multiplexer tree is a simple binary tree, each

[1]The reader will have noticed that we already can't avoid mentionning placement issues.

of its nodes being a multiplexer controlled by a bit of the address, and its leaves being LUTs used as small 4-bit addressed memories. The Virtex architecture provides multiplexers (called F5 and F6) specifically designed for arranging 2 or 4 LUTs in a CLB as a bigger look-up table with 5 or 6 input bits. Thus the level of the tree closest to the leaves uses F5 multiplexers, and the next level uses F6 multiplexers. Subsequent levels use LUTs as multiplexers, as shown on Figure 4.

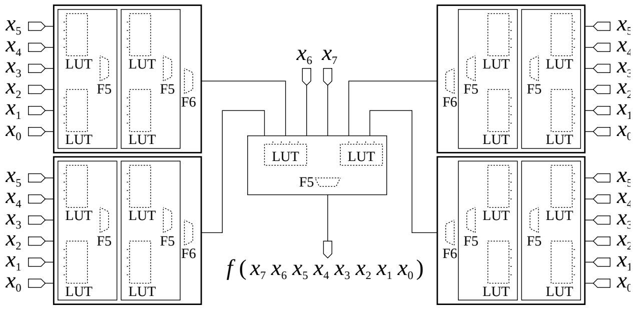


Figure 4. A Virtex multiplexer tree.

The placement of the CLB tree is then performed recursively, by placing each CLB between its sons (see Figure 5). This way, routing is simple and tables assemble in big rectangles.

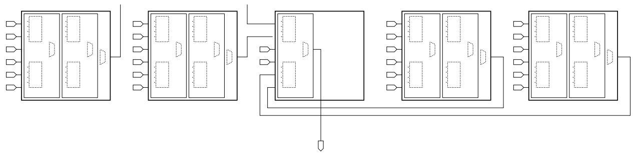


Figure 5. Placement of the multiplexer tree. This row corresponds to one output bit.

The tables built by this method are convenient to place and route. However Dinechin and Tisserand have shown that a VHDL synthesiser is able to compress them significantly [2]. The next step was therefore to study table compression. We developed a heuristic, described below, that gives excellent results with all kinds of tables. This heuristic is actually based on a more general heuristic for the optimization of binary functions, which uses classical techniques adapted to CLB trees in the Virtex architecture.

3.3 Table compression heuristic

Classically, each output bit of a table can be expressed as a boolean function of w_I variables. Well-known polynomial reduction algorithms may thus be applied to this function. The problem is well known to be NP-hard.

3.3.1 From tables to polynomials

The first step is to get a polynomial of the function from the (truth) table, and then, thanks to Karnaugh maps, a minimized expression of it: we build the boolean hypercube, then label each vertex by the value of the function at this point, and look for the biggest sub-hypercubes whose all vertices are labelled by 1. See Figure 6 for an example. This is obviously an exponential algorithm, but given the small size of the tables, this complexity is not noticeable. For 20-bit tables, it only lasts a couple of minutes.

3.3.2 From polynomials to multiplexer trees

Classical reduction techniques consist in trying to minimise the number of monomials in the polynomial. To target the particular architecture of our FPGAs, we will adapt these known algorithms so that they produce multiplexer trees with 4-input LUTs at the leaves, as used previously.

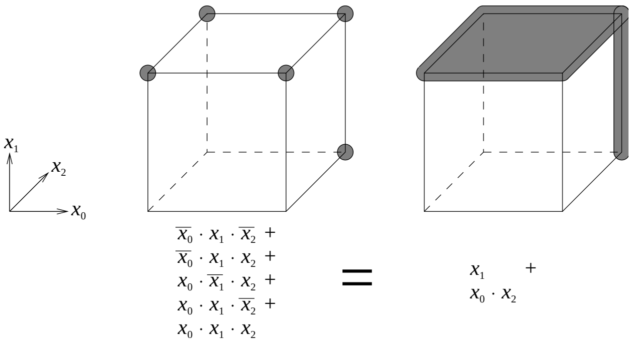

x_1
x_2
x_0
$\overline{x_0} \cdot x_1 \cdot \overline{x_2}$ +
$\overline{x_0} \cdot x_1 \cdot x_2$ +
$x_0 \cdot \overline{x_1} \cdot x_2$ +
$x_0 \cdot x_1 \cdot \overline{x_2}$ +
$x_0 \cdot x_1 \cdot x_2$
=
x_1 +
$x_0 \cdot x_2$

Figure 6. Polynomial reduction example.

To this effect we will focus on splitting a polynomial by a given variable x_K, which we call a *key variable*. Splitting can be achieved by Shannon's theorem:

$$\begin{aligned} p(x_0, .., x_K, .., x_{w_I-1}) = \\ & x_K \cdot p_1(x_0, .., x_{K-1}, x_{K+1}, .., x_{w_I-1}) \\ + \quad & \overline{x_K} \cdot p_0(x_0, .., x_{K-1}, x_{K+1}, .., x_{w_I-1}) \end{aligned}$$

The architectural interpretation of this equation is a multiplexer, controlled by x_K, with two sub-trees implementing the polynomials p_0 and p_1.

After splitting the polynomial, we simplify the resulting p_0 and p_1 using the Quine-McCluskey algorithm [9], and split them using another key variable.

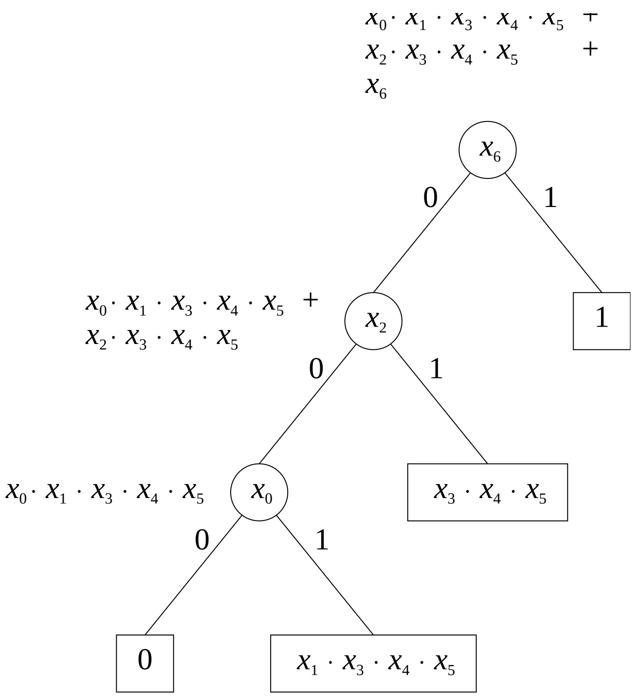
Eventually this algorithm produces functions of 4 variables which are leaves of the multiplexer tree, and will be mapped to LUTs in the FPGA.

The important part of the heuristic is now to choose the key variables and their order in order to get as small as possible a multiplexer tree.

3.3.3 Key variable selection

At each step, our heuristic is to choose the key variable that ensures the largest amount of simplification in the Quine-McCluskey simplification of p_0 and p_1. To achieve that we want to choose the key variable which plays a part in the largest number of smallest monomials.

We therefore count the occurences of each variable in the smallest monomials of the polynomial, and choose one of the most used ones. An example of this heuristic is given in Figure 7.



$x_0 \cdot x_1 \cdot x_3 \cdot x_4 \cdot x_5$ +
$x_2 \cdot x_3 \cdot x_4 \cdot x_5$ +
x_6
x_6
0
1
$x_0 \cdot x_1 \cdot x_3 \cdot x_4 \cdot x_5$ +
$x_2 \cdot x_3 \cdot x_4 \cdot x_5$
x_2
1
0
1
$x_0 \cdot x_1 \cdot x_3 \cdot x_4 \cdot x_5$
x_0
$x_3 \cdot x_4 \cdot x_5$
0
1
0
$x_1 \cdot x_3 \cdot x_4 \cdot x_5$

Figure 7. Reduction example.

3.4 Floorplanning

As far as a JBits implementation is concerned, the main drawback of the previous optimisation is that it produces a unbalanced tree of CLBs, which we cannot organise as regularly as previously. In VHDL we would simply leave the placement of this tree to the back-end tools. In JBits, however, we had to write an ad-hoc placer.

Our solution was to write a *CLB provider* class that works with a bitmap of the FPGA's area and centralises CLB allocation requests. It tries to avoid any empty space, by allocating the first free suitable space for a given order.

At the moment, the order established between the CLBs is simply a left-to-right, top-to-bottom linear order. Figure 8 is an example of this *CLB provider* placing a CLB tree.

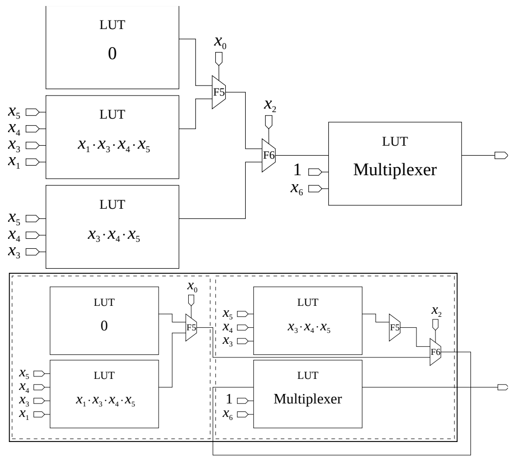


Figure 8. Placement example.

This algorithm does not even try to group CLBs from the same tree. In spite of this lack of regularity, however, the JBits router is always able to obtain a good, congestion free, routing. For this reason we did not try to improve on this rough placement. According to a study by Singh [10], however, more careful placement could lead to an increase in performance by up to 30%, and this will be the subject of further investigation.

3.4.1 Graphical interface

In order to synthesise rectangular cores, we have jointly developped a graphical interface to allow the user to specify the dimensions of the bounding rectangle, as it can be seen in the screenshot Figure 9.

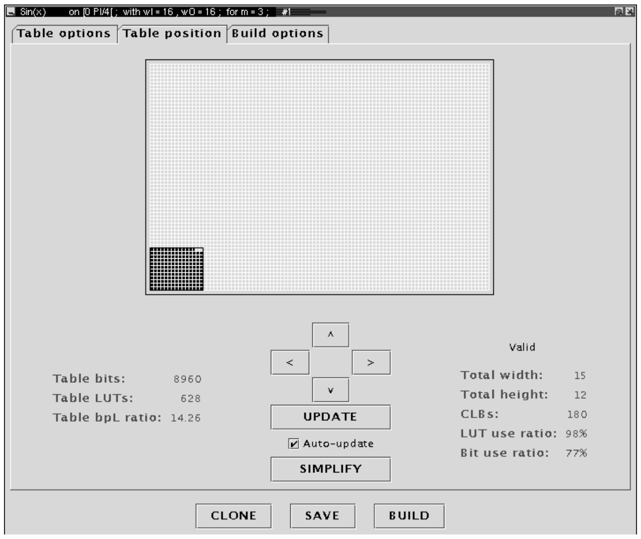
This interface actually comes on top of a previous interface to the multipartite method. It initially shows

- the adders, which will impose the minimal height of the core so that the fast carry logic can be used, and
- the required number of CLBs computed by the previous optimization.

The user then has the freedom of shaping the core rectangle to match its needs with respect to these constraints.

3.4.2 Class hierarchy

To sum up this section, Figure 10 describes the class hierarchy of our core generator. The classes have been written in such a way to allow easy reutilisation. Altogether, this represents 3 month of work for a postgraduate student.



Sin(x) on [0 PI/4[; with wI = 16 , wO = 16 ; for m = 3 ; #1

Table bits: 8960
Table LUTs: 628
Table bpL ratio: 14.26

[✓] Auto-update

Valid
Total width: 15
Total height: 12
CLBs: 180
LUT use ratio: 98%
Bit use ratio: 77%

Figure 9. Interface example: a 16-bit sine operator on a Virtex XCV1000.

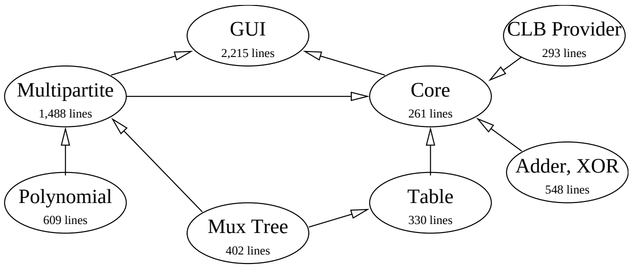


Figure 10. The class hierarchy, with code sizes.

4 Results

Some results of our multipartite core generator are given by Table 2. Synthesis was performed on a Pentium 400 with 512 MB of memory. For technical reasons, although we were able to test our cores on a Celoxica RC1000 board, we were unable to time them on this board so far. The frequencies given in this table are therefore only approximations obtained using the Xilinx FPGA Timing Analyser tool.

Three points need to be underlined:

- The optimisation algorithm provides an improvement of at least 20% in area, which compares favourably to the results obtained by Leonardo (optimising for area, medium effort). One reason for that may be that we performed binary optimisation with the target architecture in mind, instead of separating optimisation and technology mapping.
- The total compilation time is still acceptable, although slightly longer for optimized tables. Most of this time is spent in routing using JRoute. Unoptimised tables could be routed much faster by hand-writing the routing algorithm, but this would add a lot to the development time. It is doubtful, however, that cores using the multipartite method will qualify as *run-time reconfigurable cores* for precisions larger than 12 bits.
- The frequencies shows that the “smaller is faster” effect overcomes the negative effect of bad placement for all cores except the largest ones. This is consistent with Singh’s findings [10].

5 Conclusion and future work

There are three main conclusions to this work.

- Arbitrary numeric functions can be implemented efficiently in LUT-based FPGAs using the multipartite method. The core generator we developed can build a function evaluator for any function with up to 16 bits precision in seconds, and the resulting core will need only a fraction of the FPGA resources. This is in itself a great improvement over currently available cores.
- The JBits API is a great tool for developing such cores, because it allows one to integrate any kind of optimisation within the hardware description of the core. There is nothing in what we presented that could not have been done using a combination of Java, VHDL synthesiser, backend tools (for the placement), and makefiles. However we feel that the JBits approach is much more elegant for this specific problem.
- We are very glad we did not have to *route* the cores ourselves, and we wish we did not have to *place* them by ourselves. Manual placement is desirable to improve performance, it is easy when the architecture is regular, but it should be avoidable in the other cases. We hope that the next generations of JBits will include classes offering the functionality of our *CLB provider* class, improved with placement optimisation options.

Two natural directions of future work concern the aspect of performance.

- The speed of the bigger cores can very probably benefit from better placement.
- Pipelining should also be explored.

The JBits programs described above are available for download under the GNU Public Licence at `www.ens-lyon.fr/LIP/Arenaire/News/JBits/`.

function	12 bits sine		16 bits sine	
	uncompressed	compressed	uncompressed	compressed
memory	1,552 bits		8,960 bits	
size of tables	97 LUT	84 LUT	628 LUT	473 LUT
bits per LUT ratio	16.00	18.48	14.27	18.94
total size	160 LUT	147 LUT	710 LUT	555 LUT
frequency	52 MHz	39 MHz	36 MHz	35 MHz
reduction time	—	1”	—	15”
synthesis time	4”	4”	35”	20”
function	**16 bits exp**		**20 bits sine**	
	uncompressed	compressed	uncompressed	compressed
memory	11,520 bits		50,176 bits	
size of tables	810 LUT	643 LUT	3,573 LUT	2,546 LUT
bits per LUT ratio	14.22	17.91	14.04	19.70
total size	901 LUT	734 LUT	3,682 LUT	2,655 LUT
frequency	35 MHz	30 MHz	21 MHz	27 MHz
reduction time	—	20”	—	5’40”
synthesis time	40”	30”	9’	4’

Table 2. Timings, area and synthesis time of multipartite cores.

Acknowledgements

This work was partially supported by an ACI Jeunes Chercheurs of the French government, and Xilinx University Programme.

References

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

[2] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 128–135, Vail, Colorado, June 2001. Also available as LIP research report 2000-38.

[3] E. Keller. JRoute: A run-time routing API for FPGA hardware. In *7th Reconfigurable Architectures Workshop*, Cancun, Mexico, May 200O. LNCS 1800.

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

[5] C. Patterson. High performance DES encryption in Virtex(tm) FPGAs using JBits(tm). In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 2000.

[6] J. Piñeiro, J. Bruguera, and J.-M. Muller. Faithful powering computation using table look-up and a fused accumulation tree. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 40–47, Vail, Colorado, June 2001.

[7] J. Scalera and M. Jones. A run-time reconfigurable plug-in for the winamp MP3 player. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 2000.

[8] J. Scalera and M. Jones. Cores and anti-cores: Using JBits as part of a mainstream design flow. In *8th Reconfigurable Architectures Workshop*, San Francisco, USA, Apr. 2001.

[9] N. A. Sherwani. *Algorithms for VLSI physical design automation*. Kluwer Academic, 1993.

[10] S. Singh. Death of the RLOC? In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 2000.

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

[12] Xilinx Corporation. *The JBits 2.7 SDK for Virtex*, 2001.