Small Multiplier-based Multiplication and Division Operators for Virtex-II Devices

Jean-Luc Beuchat, Arnaud Tisserand

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Small Multiplier-based Multiplication and Division Operators for Virtex-II Devices

Jean-Luc Beuchat, Arnaud Tisserand

No 4494

July 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4494--FR+ENG

Small Multiplier-based Multiplication and Division Operators for Virtex-II Devices

Jean-Luc Beuchat, Arnaud Tisserand

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n ° 4494 — July 2002 — 14 pages

Abstract: This paper presents integer multiplication and division operators dedicated to Virtex-II FPGAs from Xilinx. Those operators are based on small 18×18 multiplier blocks available in the Virtex-II device family. Various trade-offs are explored (computation decomposition, radix, digit sets, ...) using specific VHDL generators. The obtained operators lead to speed improvements up to 18% for multiplication and 40% for division compared to standard solutions only based on CLBs.

Key-words: Multiplication, division, FPGA, Virtex-II family.

(Résumé : tsvp)

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International : +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International : +33 4 76 61 52 52

Opérateurs de multiplication et de division pour la famille Virtex-II construits à partir de petits multiplicateurs

Résumé : Cet article présente des opérateurs de multiplication et division entières destinés aux FPGA de la famille Virtex-II de Xilinx. Ces opérateurs sont basés sur les petits blocs de multiplication 18×18 disponibles sur les Virtex-II. Différents compromis sont étudiés (décomposition des calculs, base, ensemble de chiffres, ...) grâce à des générateurs automatiques de code VHDL. Les opérateurs obtenus conduisent à des gains en vitesse de 18% pour la multiplication et 40% pour la division par rapport aux solutions standard utilisant uniquement des CLB.

Mots-clé : Multiplication, division, FPGA, famille Virtex-II.

1 Introduction

Many algorithms and architectures have been developed for implementing multiplication and division in FPGAs. Those algorithms mainly use solutions proposed for standard integrated circuits. They are based on very low-level basic elements such as full adder and NAND/XOR gates.

Some recent FPGAs embed hardwired small multipliers blocks. For instance, Virtex-II devices from Xilinx include many 18×18 multipliers. ORCA Series 4 from Lattice or Stratix from Altera FPGAs also embed small multipliers. These new basic elements may lead to more efficient arithmetic operators.

This paper deals with the design and the optimization of integer multiplication and division operators based on a combination of small multiplier blocks and configurable logic blocks for the Virtex-II devices. Section 2 briefly describes Virtex-II features used in this work. The multiplication algorithms, operators and implementation results are described in Section 3, and Section 4 deals with division. Finally, Section 5 presents some conclusions and future prospects.

2 The Virtex-II Family

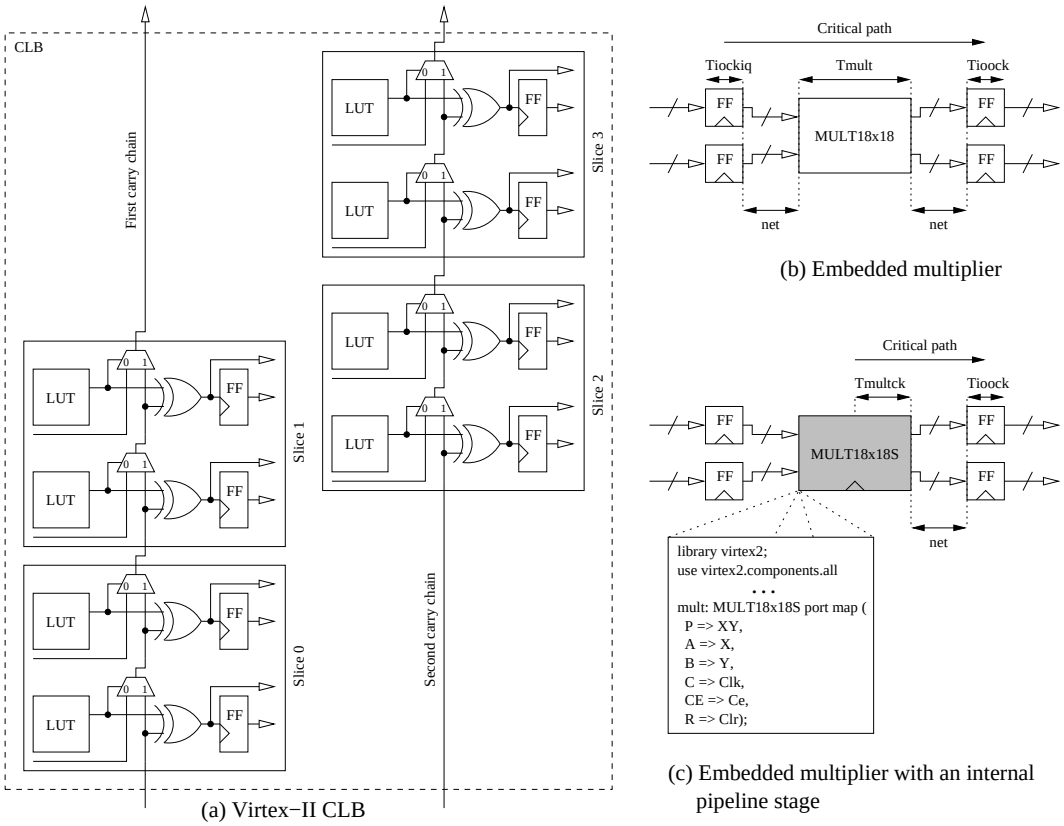
Virtex-II configurable logic blocks (CLBs) provide functional elements for synchronous and combinatorial logic. Each CLB includes four slices containing basically two 4-input look-up tables (LUT), two storage elements, and fast carry logic dedicated to addition and subtraction. A CLB has two separate carry chains, whose height is two bits per slice, running upward (Figure 1a).

Virtex-II circuits embed many 18×18 two's complement multipliers (the MULT18x18 blocks), each of them supporting two input ports 18-bit signed or 17-bit unsigned wide (Figure 1b). Furthermore, each multiplier has an internal pipeline stage. Surprisingly, this feature is poorly documented in the Virtex-II data sheet[1] and synthesis tools seem unable to automatically deal with it. The MULT18x18S component, available in the library of Synplify Pro, allows us to write multipliers that take advantage of this characteristic (Figure 1c).

3 Unsigned Multiplication

This section presents the design of efficient unsigned integer multipliers based on the MULT18x18 blocks available in Virtex-II devices. The algorithms studied here

[1] There is only one table with the switching characteristics of a pipelined MULT18x18 block.



(a) Virtex−II CLB

(b) Embedded multiplier

(c) Embedded multiplier with an internal pipeline stage

Figure 1: Virtex-II arithmetic features overview.

involve splitting the operands into two parts. A m-bit integer X is decomposed into a lower part X_0 and a higher part X_1 such that

$$X = X_0 + 2^n X_1 = \sum_{i=0}^{n-1} x_i 2^i + 2^n \sum_{i=0}^{m-n-1} x_{n+i} 2^i,$$

where $n < m$. Paragraphs 3.1, 3.2, and 3.3 describe three architectures requiring from one to four MULT18x18 blocks. Evaluation and comparison of these methods are summarized in paragraph 3.4.

3.1 Divide-and-Conquer Approach (4 MULT18x18 Blocks)

The well-known divide-and-conquer approach (see for example [6]) requires four small multiplications and two additions to compute the product XY (Figure 2) and is based on the following equation:

$$(X_1k + X_0)(Y_1k + Y_0) = X_1Y_1k^2 + (X_1Y_0 + X_0Y_1)k + X_0Y_0,$$

where $k = 2^n$. Note that the n least significant bits are obtained directly from a MULT18x18 block. Synplify Pro uses this method to synthesize a multiplier that is larger than the MULT18x18 width. However, X_0 and Y_0 respectively contain the 17 least significant bits of X and Y. Our experiments will demonstrate that this choice does not always lead to the fastest circuit. Therefore, we have written a VHDL generator allowing the user to specify the width of X_0, X_1, Y_0, and Y_1.

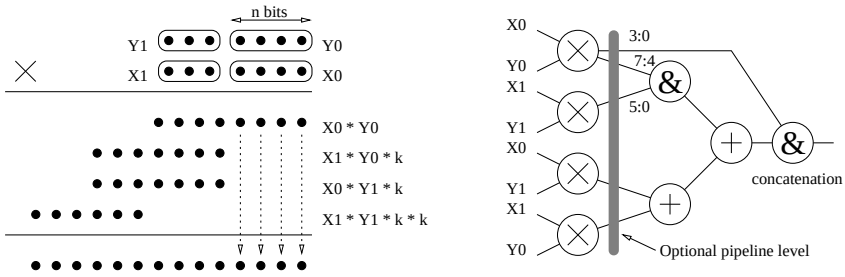


Figure 2: The divide-and-conquer method (4 MULT18x18 blocks).

3.2 Two-Way Method (3 MULT18x18 Blocks)

The two-way method, originally proposed by Karatsuba and Ofman [3], allows us to spare a MULT18x18 block and is based on the rewriting:

$$(X_1k + X_0)(Y_1k + Y_0) = X_1Y_1(k^2 - k) + (X_1 + X_0)(Y_1 + Y_0)k + X_0Y_0(1 - k).$$

It involves three multipliers, namely X_1Y_1, $(X_1 + X_0)(Y_1 + Y_0)$, and X_0Y_0, three adders, and two subtracters (Figure 3). However, this method has a small drawback in that $(X_1 + X_0)$ and $(Y_1 + Y_0)$ require $n + 1$ bits. In [4], Knuth suggested an improvement by writing:

$$(X_1k + X_0)(Y_1k + Y_0) = X_1Y_1(k^2 + k) - (X_1 - X_0)(Y_1 - Y_0)k + X_0Y_0(1 + k).$$

Knuth was able to replace $X_1 + X_0$ by $X_1 - X_0$. At the price of a few additional logic, we can always subtract the lesser from the greater and save the extra bit [7]. However, we will not consider this option in the following.

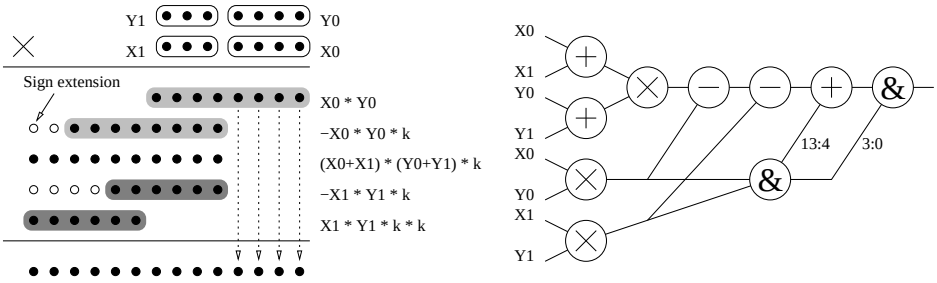


Figure 3: The two-way method (3 MULT18x18 blocks).

3.3 Using a Single MULT18x18 Block

When the size m of the operands is slightly greater than 17, the divide-and-conquer approach leads to a waste of resources. For example, if X and Y are 20-bit unsigned integers, Synplify allocates a MULT18x18 block for the 3×3 multiplication X_1Y_1. In such cases, we suggest an architecture involving a single MULT18x18 which carries out the product X_1Y_0 (Figure 4a) and some additional logic for the few other partial products computation and addition. We combine the remaining x_iy_j terms and build $m-17$ partial products as depicted in Figure 4a. These partial products PP_i are summed using a tree of carry-propagate adders (Figure 4b). Carry-save adders are not useful in Virtex FPGAs because of the fast carry logic. Formally, PP_i is a $(2m-i-1)$-bit integer defined by:

$$\mathrm{PP}_i = \sum_{j=0}^{m-1-i} x_i y_j 2^{i+j} + \sum_{j=i+1}^{m-1} x_j y_{m-1-i} 2^{j+m-i-1}.$$

The difficulty lies in the determination of the width of the intermediate sums. Let us notice that

$$\begin{aligned}\max(\mathrm{PP}_i) &= \sum_{j=0}^{m-1-i} 2^{i+j} + \sum_{j=i+1}^{m-1} 2^{j+m-i-1} \\ &= \sum_{j=0}^{m-1-i} 2^{i+j} + \sum_{j=m-i}^{2m-2i-2} 2^{i+j} = 2^i\left(2^{2m-2i-1}-1\right).\end{aligned}$$

Therefore,

$$\begin{aligned}\max(\mathrm{PP}_i + \mathrm{PP}_{i+1}) &= 2^{2m-i-1} + 2^{2m-i-2} - 2^{i+1} - 2^i \quad (1)\\ &= 10\,\underbrace{11\cdots11}_{2m-2i-4\times}\,01\,\underbrace{0\cdots0}_{i\times} \quad (2)\end{aligned}$$

and $\mathrm{PP}_i + \mathrm{PP}_{i+1}$ is a $(2m-i)$-bit integer. Assume that $m = 21$ and consider the sum $(\mathrm{PP}_0 + \mathrm{PP}_1) + (\mathrm{PP}_2 + \mathrm{PP}_3)$. From (2), we deduce

$$\begin{aligned}&\max\Big(\underbrace{(\mathrm{PP}_0 + \mathrm{PP}_1)}_{42\text{ bits}} + \underbrace{(\mathrm{PP}_2 + \mathrm{PP}_3)}_{40\text{ bits}}\Big)\\ &\quad = 10\underbrace{11\cdots11}_{38\times}01 + 10\underbrace{11\cdots11}_{34\times}0100.\end{aligned}$$

Hence, the sum is a 42-bit integer. We have written a VHDL generator that uses such rules to compute the sum of the PP_i terms. While the solution illustrated in Figure 4c seems easier to implement, it leads to larger and slower design. The synthesis tools can not guess the properties of the partial products and allocate full-adders whose inputs remain equal to zero.

3.4 Implementation Results

We have written a C library which generates VHDL descriptions of the architectures described above. The VHDL code was synthesized with Synplify Pro 7.0.3 and implemented on a Virtex-II XC2V500-6 device using Xilinx Alliance Series 4.1.03i. In the following, m and n respectively denote the total size of the operands and the size of their lower part.

The first experiment compares the three solutions (Figure 5). The divide-and-conquer strategy leads to the smallest circuits in terms of slices number, except for

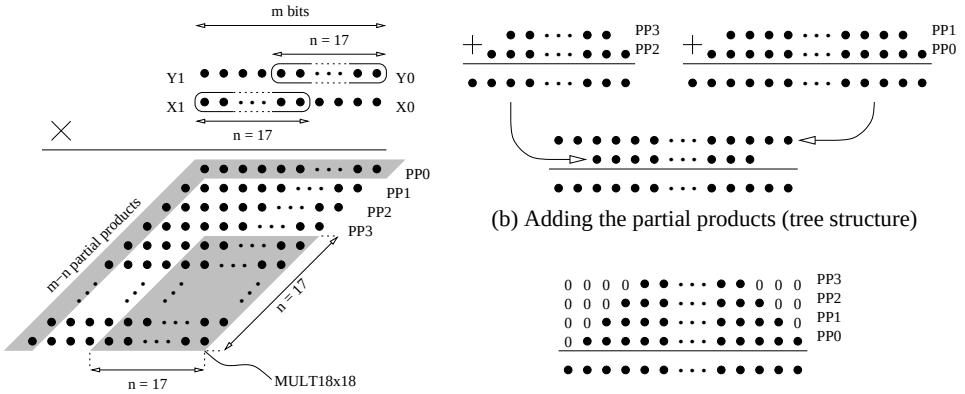


Figure 4: Multiplication with a single MULT18x18 block.

$m = 18$ and $m = 19$ where the single MULT18x18 architecture is smaller. The two-way method is disappointing: though it spares a small multiplier, it is slower and the additional logic is twice bigger than the divide-and-conquer approach. Finally, the architecture described in paragraph 3.3 is a good trade-off between size and period when m is close to 17.

An interesting point is that the proposed single MULT18x18 architecture is still efficient for $m = 23$, which is the mantissa size in IEEE standard floating-point numbers (a survey on floating-point can be found in [2]). We have implemented two versions of the 23×23 unsigned multiplier involved in the floating-point multiplication with two pipeline stages and the MULT18x18S primitive. The divide-and-conquer approach implies 50 slices, four MULT18x18S blocks, and has a period τ equal to 6 ns. Our algorithm needs 182 slices, a single MULT18x18S block and is a little bit slower ($\tau = 8$ ns). We plan to study floating-point arithmetic on FPGAs.

Let us now study the impact of n on the divide-and-conquer method. We obtain the smallest circuits when $n = 17$ (Figure 6). This result is not surprising: remember that the n least significant bits come directly from a MULT18x18 block. Consequently, the larger n is, the smaller the last adder of Figure 2 becomes. For a combinatorial circuit, choosing $n = m/2$ leads however to a faster circuit. When we introduce a pipeline stage by replacing all MULT18x18 blocks by MULT18x18S, our experiments show that $n = 17$ is the best solution: the size is exactly the same one that in the combinatorial case (that's why it is not plotted in Figure 6) and the period doesn't depend anymore on the value of n.

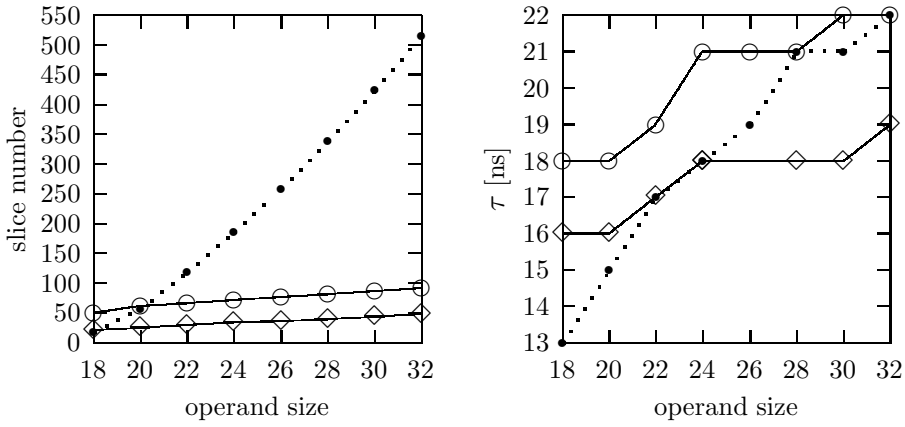


Figure 5: Number of slices and period τ of various unsigned multipliers on a XC2V500-6 device; $\diamond$: divide-and-conquer algorithm; X_0 and Y_0 are 17-bit integers (4 MULT18x18 blocks); $\circ$: two-way method (3 MULT18x18 blocks); $\bullet$: algorithm described in paragraph 3.3 (1 MULT18x18 block).

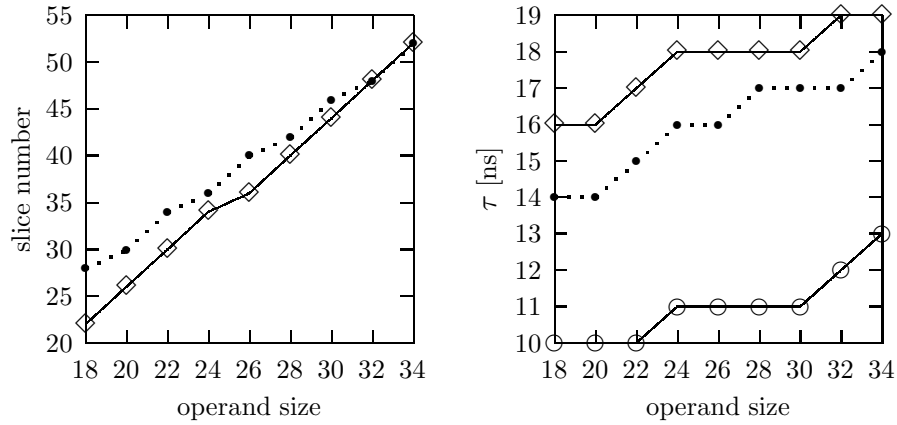


Figure 6: Impact of the choice of n on the divide-and-conquer method; $\diamond$: $n = 17$; $\bullet$: $n = m/2$; $\circ$: $n = 17$ with MULT18x18S.

The architectures described in this section only need limited changes so as to perform signed multiplication. The use of MULT18x18 blocks can lead to waste resources for small operand width (i.e. $n << 17$). For the specific case of very

small operand width (up to 5 bits), two multiplications can be shared in the same MULT18x18 block.

4 Unsigned Division

The most common division implementation in processors is the digit-recurrence algorithm named SRT from the initials of Sweeney, Robertson, and Tocher. SRT and other division algorithms and implementations can be found in a complete survey [5]. Digit-recurrence algorithms retire a fixed number of quotient bits in every iteration. SRT dividers are similar to the "paper-and-pencil" method, but they only use limited comparisons to speed-up the quotient selection. SRT dividers are typically of low complexity, utilize small area, but have relatively large latencies. A complete book is devoted to digit-recurrence algorithms [1].

Figure 7 presents a standard radix-r SRT iteration architecture ($r = 2^k$). There are $t = \lceil n/k \rceil$ iterations in the division of n-bit integers. Two additional cycles are required (1 before and 1 after the iterations) to check input values (division by zero and scaling) and to convert the signed-digit quotient to a standard radix-2 notation. The division x/d produces k bits of the quotient q at every iteration. The quotient digit q_j is represented using a radix-r notation. The first residual w_0 is initialized to x. At iteration j, the residual w_j is shifted by k bits left (it produces rw_j). Based on a few most significant bits of rw_j and d (n_r and n_d-bit large respectively), one can deduce the next quotient digit q_{j+1} using the quotient digit selection table (Qsel). Finally, the product $q_{j+1} \times d$ is subtracted to rw_j to form the next residual w_{j+1}.

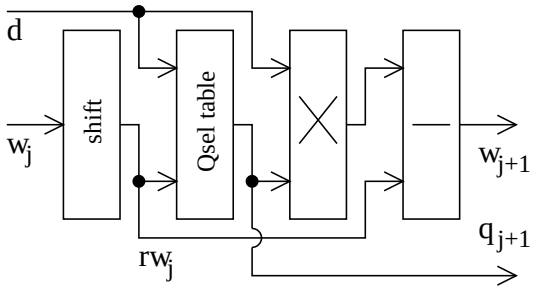


Figure 7: SRT division iteration architecture.

4.1 Virtex-II SRT Dividers Implementations

A SRT dividers generator was developed in C++. Based on the operator parameters (operands width, radix, quotient and residual representations), the program checks the parameters combination, generates the quotient digit selection table and the synthesizable VHDL code of the complete operator description. There is no pipeline level inside the generated iteration description.

There are several important parameters in a SRT divider. The radix $r = 2^k$ plays a major role. For large values of k the iteration number t is small but each iteration is complex (large Qsel tables, complex product $q_{j+1} \times d$). High radices (larger than 16 in our case) lead to very huge quotient digit selection tables and seem to be impracticable.

The radix-2^k quotient is represented using a signed-digit redundant number system to ensure that next quotient digit determination is possible only based on a few most significant n_r and n_d bits of the residual and divisor respectively. Several digit sets $\{-\alpha, -\alpha+1, \ldots, 0, \ldots, \alpha-1, \alpha\}$ can be used for radix-2^k notation depending on the values of k and α. For instance, with a radix-4 representation, the minimally redundant digit set is $\{-2, -1, 0, 1, 2\}$ ($\alpha = 2$) and the maximally redundant digit set is $\{-3, -2, -1, 0, 1, 2, 3\}$ ($\alpha = 3$). The digit set used for the quotient is an important design decision. High values of α lead to simpler quotient digit selection (smaller values of $n_r + n_d$ as the address width of the Qsel table) but also to more complex product $q_{j+1} \times d$. In this work, we only use distributed RAM (based on the LUTs inside the CLBs) for the implementation of the selection tables. Furthermore, the specific RAM blocks (called BRAM in Virtex devices) can be used for the Qsel tables.

Usually, another important parameter is the residual w_j representation. For VLSI implementation, a redundant number system such as carry-save is used for w_j to accelerate the $q_{j+1} \times d - rw_j$ subtraction. For Xilinx FPGAs implementation, a non-redundant number system such as the two's complement is sufficient for the residual because of the fast-carry logic available in Virtex-II devices.

In this work we compare a standard radix-2 division (noticed "std r2") and several SRT dividers based on MULT18x18 blocks (noticed "r?/?"). Radix 4, 8 and 16 have been investigated with several quotient digit sets. For each solution, 16, 24, 32 and 40-bit operands operators have been generated. The different solutions investigated in this work are summarized in Table 1.

A radix-16 iteration architecture has been generated but it leads to a huge area (the quotient digit selection table is then a 14-bit address table). It requires more

solution name	std r2	r4/2	r4/3	r8/5	r8/6	r16/12
radix	2	4	4	8	8	16
α	1	2	3	5	6	12
MULT18x18 blocks	no	yes	yes	yes	yes	yes
n_r, n_d	na	4,3	6,3	7,4	6,4	7,7

Table 1: Summary of the generated and tested division operators.

2 700 slices of the XC2V500-6 (more than 70% of the slice count). So, we limited our study to radices smaller or equal to 8.

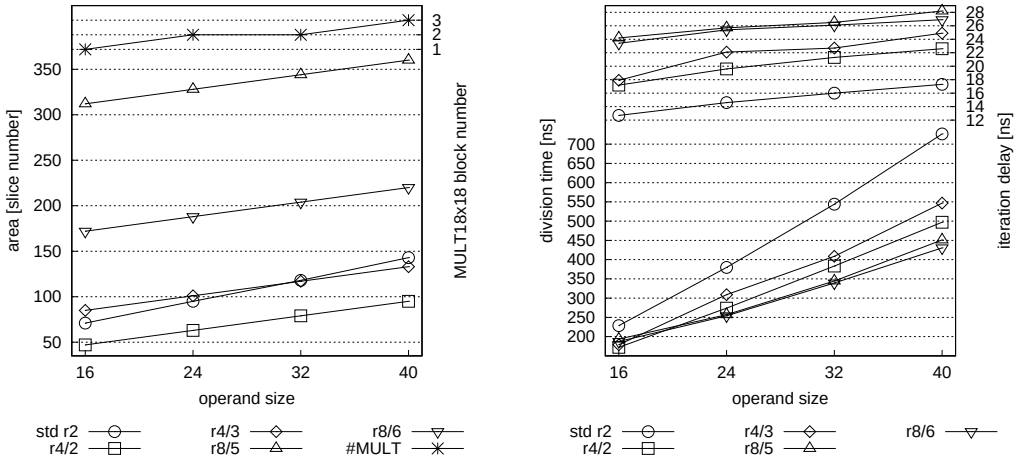


Figure 8: Implementation results on Virtex-II of the generated division operators (area on left side, speed on right side).

Figure 8 presents the implementation results of the corresponding generated operators. The left part of Figure 8 presents the area results. Depending on the solution ("std r2" or "r?/?") and the operands width, the total area used in the device is the sum of the slice number and the number of MULT18x18 blocks reported on the figure. For instance, a radix-8 and α=6 solution leads to 172 slices and 1 MULT18x18 block for 16-bit operands and to 204 slices and 2 MULT18x18 blocks for 32-bit operands. The standard radix-2 solution does not use any MULT18x18 block. Speed achievements are presented on the right part of Figure 8. There are two curves sets. The five

lower ones represent the total division time while the five upper ones represent one iteration delay. All delays have a linear growth with the operand size n. Significant speed improvements are achieved using MULT18x18 blocks for the $q_{j+1} \times d$ product. Indeed, for 40-bit operands a radix-8 solution based on 3 MULT18x18 blocks is more than 40% faster than the standard radix-2 solution.

5 Conclusion

In this paper, improved architectures and implementations of integer multiplication and division operators for Virtex-II FPGAs have been presented. Those operators are based on a combination of small hardwired 18x18 multipliers and CLBs. Dedicated VHDL generators have been developed to provide a wide parameter space exploration. Various trade-offs between speed and area have been explored. A significant speed improvement is possible using MULT18x18 blocks. Cunningly using a few MULT18x18 blocks combined to a few additional logic leads to very efficient operators.

For unsigned multiplication, a computation decomposition based on a single MULT18x18 and some additional logic can lead to faster designs than with 4 blocks. For operands size slightly larger than 17, the single MULT18x18 block solution leads to smaller and faster multipliers. A radix-8 SRT division architecture with a product generator based on MULT18x18 blocks leads to a up to 40% speed improvement compared to a standard radix-2 solution.

Further research is needed to explore other parameters. The presented solutions and tools will be extended to signed integers. The presented multiplication solution can be derived to close operators such as multiplication and accumulation and square operators while SRT division can be extended to square-root. For division operators various coding of the signed quotient digits are possible (two's complement, sign-magnitude, one-hot codings...). Some codings will reduce the selection table size while they require more complex $q_{j+1} \times d$ product generation. Floating-point operators will also be another future research direction.

Acknowledgments

The authors would like to thank the "Ministère Français de la Recherche" (grant # 1048 CDR 1 "ACI jeunes chercheurs"), the "Fonds National Suisse de la Recherche Scientifique", and the Xilinx University Program for their support.

References

[1] M.D. Ercegovac and T. Lang. *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[2] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[3] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Phys. Doklady*, 7(7):595–596, January 1963.

[4] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 2nd edition, 1981.

[5] S.F. Oberman and M.J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.

[6] B. Parhami. *Computer Arithmetic*. Oxford University Press, 2000.

[7] D. Zuras. More On Squaring and Multiplying Large Integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399