Laboratoire de l'Informatique du Parallélisme

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

Constant Multipliers for FPGAs

Florent de Dinechin, Vincent Lefèvre

May 2000

Research Report N° 2000-18

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

Constant Multipliers for FPGAs

Florent de Dinechin, Vincent Lefèvre

May 2000

Abstract

This paper presents a survey of techniques to implement multiplications by constants on FPGAs. It shows in particular that a simple and well-known technique, canonical signed recoding, can help design smaller constant multiplier cores than those present in current libraries. An implementation of this idea in Xilinx JBits is detailed and discussed. The use of the latest algorithms for discovering efficient chains of adders, subtractors and shifters for a given constant multiplication is also discussed. Exploring such solutions is made possible by the new FPGA programming frameworks based on generic programming languages, such as JBits, which allow an arbitrary amount of irregularity to be implemented even within an arithmetic core.

Keywords: Constant multipliers, FPGAs, KCM

Résumé

Cet article discute différentes techniques qui permettent d'implémenter des multiplications par des nombres constants sur des réseaux de cellules reconfigurables *(FPGAs)*. Il montre en particulier qu'une technique classique, le recodage canonique en chiffres signés, permet de concevoir des multiplieurs qui sont plus petits que ceux que l'on trouve dans les bibliothèques standard. Une implantation dans le système de développement Xilinx JBits en est présentée et discutée. Puis nous discutons l'application aux FPGAs d'algorithmes plus récents de calcul de chaînes efficaces d'additions et décalages pour la multiplication par une constante donnée. L'étude de ce type de solution est rendue possible par les nouveaux environnements de développement pour FPGA qui, comme JBits, sont construits au dessus d'un langage de programmation classique: cela permet d'intégrer des optimisations d'une complexité arbitraire même dans le code d'un opérateur arithmétique.

Mots-clés: Multiplication par une constante, FPGA, KCM

1 Introduction

Multiplication by a constant value is very useful in computational cores such as filters and FFTs. From a hardware point of view, it is almost always a waste of space and time to use a generic multiplier to implement a constant multiplier. This is all the more true on reconfigurable systems, where constants may easily, well, be changed. The coefficients of a finite impulse response (FIR) filter, for example, may be adjusted during the life of this filter. However, as soon as the lifetime of a coefficient significantly exceeds the reconfiguration time, it makes sense to consider it a constant and to optimize the FPGA configuration - here the multipliers which input this constant – accordingly. In this work the word "constant" will thus denote a value that is constant between two reconfigurations.

The purpose of this article is to explore the design tradeoffs offered by current FPGAs for constant multiplication. There are two novel aspects to consider. First, the size of recent FPGAs makes it possible to compute on large numbers, up to 32 bit wide and more. This invites us to explore new techniques for the optimization of constant multipliers. Second, new FPGA design tools (such as PamDC [5], Xilinx JavaBits [6] or others [4]) embed a universal and powerful programming language in the hardware description language. In our case, this will allow a fair amount of constant-dependent optimization to be embedded in the program of a hardware constant multiplier core.

Section 2 provides a survey of the general problem of constant multiplication, with a focus on FPGA implementations. Our implementation of a variable size constant multiplier with constant-dependent routing is then detailed in section 3. The last section describes the roadmap of the work that this study suggests.

2 Constant multiplication: A survey

Most algorithms are just briefly introduced: a more detailed description will be found in the references given.

2.1 Multiple constant multiplications

The literature pays much attention to the case of *multiple constant multipliers* appearing in filters like FIR, FFTs, or other vector-product based filters for image or signal processing. Techniques such as Multiple Constant Multiplications [12], Distributed arithmetics [12, 10, 11] and Multiple Constant Multiplier Trees [1] (among other) have been developed to optimize the shift-and-add subexpressions globally between the multipliers. There is probably still a lot of interesting results to come in this field.

In the following of this paper, we will concentrate on the simpler case of a single, isolated constant multiplication. The main reason for that is one of simplicity. Nevertheless, considering on one side the difficulty of global optimization processes when the problem is large, and on the other side the need for locality and regularity in the routing for performance reasons, this work might also be useful in the context of multiple constant multipliers.

Notations We will denote k the constant, written on n bits, and x the variable to be multiplied, written on m bits.

2.2 Shift-and-add algorithms for single constant multiplication

In this section we consider various existing algorithms without regard to a hardware or FPGA implementation: the cost unit will be an addition or a subtraction.

2.2.1 Straightforward algorithm

The classical binary decomposition of the constant k gives us the most straightforward algorithm : if we write

$$k = \sum_{i=0}^{n-1} 2^i k_i$$

with $k_i \in \{0, 1\}$, then we have

$$kx = \sum_{i=0}^{n-1} 2^i x k_i \quad .$$

The product $2^i x$ is computed simply by shifting the binary decomposition of x to the left, and the number of actual additions in the previous sum is the number of 1s in the decomposition of k. Thus this methods generates between 0 and n additions, with an average of $n/2$.

2.2.2 Canonical signed recoding

A first variant of this algorithm (the origin of which is unclear acording to Hwang [7]) is to use some form of *recoding* of the bits of the constant.. The idea is to express the constant in a redundant digit system, typically $\{\overline{1}, 0, 1\}$ where $\overline{1}$ has the value -1. A number like 0111 (=1+2+4) may then be recoded as $100\overline{1}$ (=8-1). In the multiplication, a digit $\overline{1}$ is translated into a subtraction (which usually has the same cost as an addition). For any k there exists a canonical representation where at least one digit out of two is a zero, which means that at most $n/2$ additions are needed for the constant multiplication. It can also be shown that such recoding generates an average of $n/3$ additions.

2.2.3 Bernstein algorithm

The previous method, however, does not necessarily produce the shortest shift-and-add chain for a given constant multiplication (the problem is believed to be NP-complete). For example, if $k = 657$, one may check that $kx = (8x + x) + 8(8x + x) + 64(8x + x)$, which means that the product may be computed by only three additions and three shifts (re-using the value $8x + x$).

A well-known algorithm is due to Bernstein [2]. It is a branch-and-bound method testing recursively if k has numbers of the form $2^i - 1$ or $2^i + 1$ among its divisors. However the exponential complexity of this algorithm makes it impractical even for 32-bits constants.

2.2.4 Lefèvre algorithm

Lefèvre has therefore recently proposed a polynomial algorithm which is based on the discovery of patterns in the binary representation of k [8]. This algorithm gives better results than Bernstein's even for small constants, and allows constant multipliers up to several thousands of bits to be generated. So far it has only been implemented on microprocessors to produce efficient constant multiplications by very large numbers for a specific problem (the exhaustive worst-case search for the correct rounding of floating-point function [9]). One of the purposes of our study is to evaluate its suitability for an FPGA implementation, which could be useful for example in cryptography applications.

2.3 Some FPGA implementations

This section explores in more details the constant multiplier design space, in the case when the area cost unit is one 4-input look-up table (LUT), the elementary building block of FPGAs of the Xilinx 4000 and Virtex families.

2.3.1 The naive shift-and-add algorithm

There is a very straightforward implementation of this algorithm as an FPGA arithmetic core which leads to a very regular structure. The core consists of n stages. Each stage shifts the result of the previous stage by one bit, and either adds x to it or not, depending on the value of the corresponding bit in the binary code of k.

Although the final product will be coded on $m+n$ bits, it is easy to see that each adder needs only be of size m: as x (shifted) is added to the current partial sum, only the corresponding bits of this sum participate in the addition. Therefore the size of this core is $m \times n$.

Such a core is given in the current distribution of JBits as an example of run-time parametrizable (RTP) core [6]. Due to its simplicity and regularity, it is very fast to generate, although very wasteful in space.

2.3.2 The KCM algorithm

This algorithm, due to Chapman [3], is specifically adapted to LUT-based FPGAs. It is also the basis for Distributed Arithmetics approaches cited in 2.1. The idea is to break down the binary decomposition of x into 4-bit chunks (or, to express x in base 16):

$$x = \sum_{i=0}^{\lceil \frac{m}{4} \rceil} x_i.16^i \quad .$$

Now the product becomes

$$kx = \sum_{i=0}^{\lceil \frac{m}{4} \rceil} kx_i.16^i$$

and we have a sum of products kx_i, each of which can be computed by a $16 \times n+4$ bits look-up table, x_i being the address. Here the summation can take the form of an adder tree. The area cost is $\lceil \frac{m}{4} \rceil \times (n+4)$ LUTs for the tables, plus the adder tree of depth $\log_2 \lceil \frac{m}{4} \rceil$ (the adders being of growing size). In the best case, when m is a power of two (at least 8), the LUT cost of the adder tree is $(\frac{m}{4}-1)n+\frac{m}{2}\log_2\frac{m}{4}$, counting one full-adder cell per LUT.

The total LUT cost of the KCM is thus $(\frac{m}{2}-1)n+m+\frac{m}{2}\log_2\frac{m}{4}$.

These formulas do exactly match the Xilinx KCM core generator for Virtex [13]. Such a core is also present in the current JBits distribution, but only for 8×8 bit multiplier. Its size is 8×6 LUTs.

3 A variable size constant multiplier using canonical recoding

Recent FPGA development tools such as Xilinx JBits allow us to relax two constraints which oriented the design of the previous cores:

- routing may be arbitrarily irregular even within an arithmetic core, and
- the size of a core need not be known before compile time.

This section details and discuss the implementation in JBits of a new core generator for constant multiplication which exploits this new freedom. It is a rather straightforward implementation of the canonical recoding idea. The size of the generated cores is at most $n\times m/2$ LUTs, and in average $n\times m/3$ LUTs. These cores always have the form of a rectangle of n LUTs height, so they can still be integrated in datapaths.

This is, area-wise, a definite improvement over the naive method, and even over the KCM. The catch is that such a core generator needs to perform some kind of constant-dependent routing.

3.1 Overview

Our generator first computes the canonical recoding of the constant, and then instantiates a variable number of stages computing either an addition or a subtraction, depending on the bits of the recoding. Each stage i consists of m full adder cells, and adds the contribution of the i-th non-zero bit (either x or $-x$) to the partial sum shifted by the appropriate amount. The least significant bits of this partial sum may be output directly, as they don't appear in any subsequent operation. Figure 1 shows for example a multiplier by 221.

This core generator consists in less than 1000 lines of heavily commented Java. It is available for download at

`http://www.ens-lyon.fr/~fdedinec/recherche/`

3.2 Implementation details

We target Virtex chips, where the adders may be very efficiently implemented using the dedicated fast carry logic. On Fig. 1, each grey block is a LUT configured as a *full adder*, and on Fig. 2, each small square represents the output of a LUT (the LUTs are grouped by CLBs). In a column of CLBs it is thus possible to place two fast adder slices.

To implement the subtractions, we cannot straightforwardly use the classical two's complement notation $a-b=a+\overline{b}+1$, where the $+1$ is implemented as a

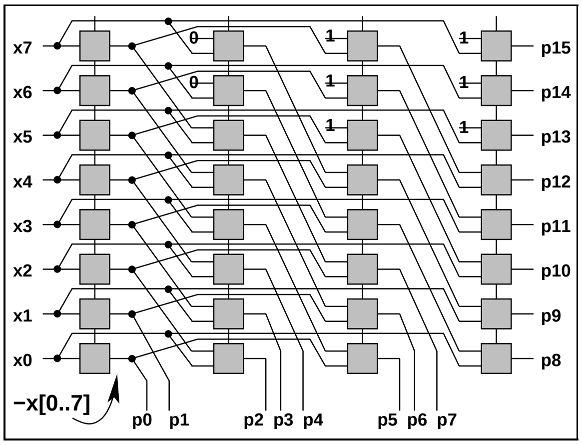

Figure 1: A 8×8-bit multiplier by 221, using the recoding $100\bar{1}00\bar{1}01$

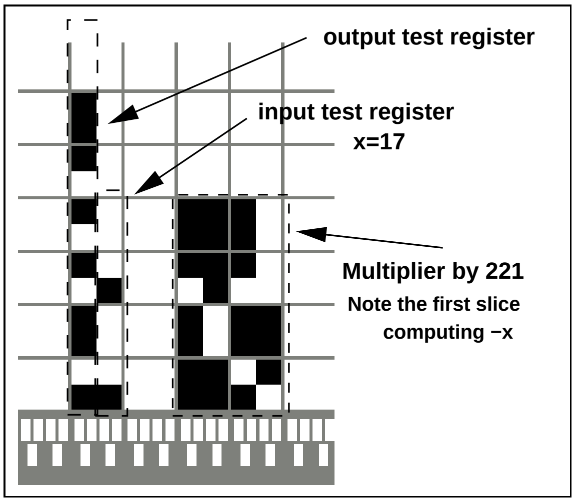

Figure 2: The state view of a test of the multiplier by 221 in Boardscope.

carry input to the adder. The problem is that we want each stage to be only of size m, *i.e.* to operate on the bits of the partial sum between the current bit i and $i+m$. A carry in on the ith stage should however be input on bit 0 of the partial sum, and would therefore potentially entail a carry propagation along the lower bits.

A solution would be to sum up all these carries in at compile time, and then start the sum (of x's and $\overline{x}$'s) with this initial value. We didn't find a clever way to do that without adding a slice of LUTs to the core, so we implemented another solution which has the same cost and other advantages: a first slice computes $-x$, and then all the slices are adders. Care must be taken however, when we know that the current partial sum is negative, to perform a sign extension of this partial sum, *i.e.* feed the free inputs with ones instead of zeroes (see Fig. 1). One advantage of this solution is that it makes the handling of two's complement signed numbers easy: as we already manipulate internally x and $-x$, operating on signed input and signed constants is only a matter of setting the sign extension bits properly (although this is not implemented yet).

The core generator only synthesizes this first $-x$ slice when needed, which lead to a small modification of the classical canonical recoding algorithm: it recodes 3 as $10\overline{1}$, which in our case is more expensive as 11. Our recoding avoids this case, sometimes saving a slice.

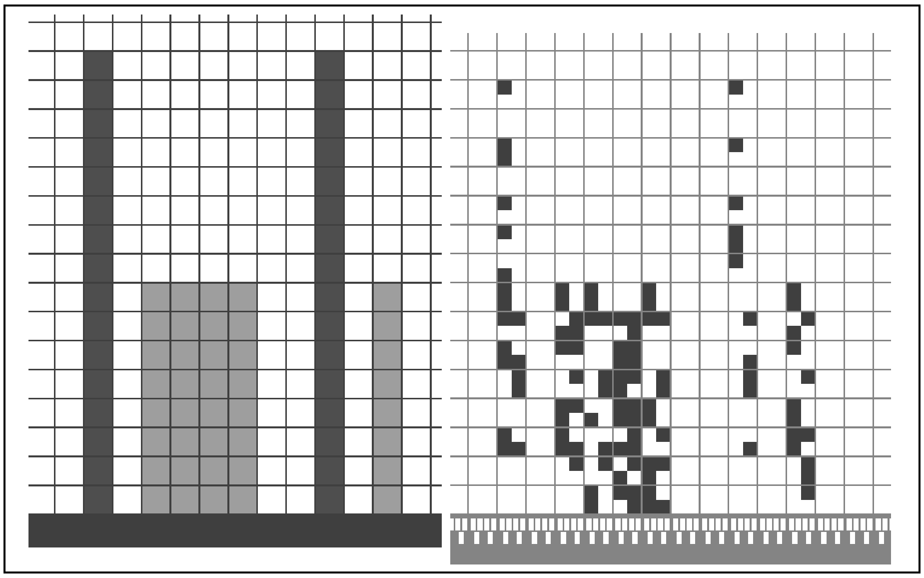

Figure 3: The core view and state view in Boardscope, for multipliers by $58995 = 100\overline{1}00110100\overline{1}0011$, and $57344 = (100\overline{1})2^{13}$

To implement the routing of the shifts, we first wrote a "stitcher" [6] core which was a simplified router able of doing only arbitrary shifts. Then Xilinx released a new version of JBits including an interface to a generic router, and we switched to this one. This allows a cleaner and safer interface of our core to other JBits object, ensuring that no routing resource conflict will occur. It also greatly simplified our code. The drawback is that, being more general, the Xilinx router is slower.

3.3 Experimental results

We extensively tested this constant multiplier core generator, currently only under simulation with the Boardscope tool. Unfortunately we have not yet been able to make any measure nor estimation of the speed of the resulting cores, as we don't own a Virtex board, and JBits doesn't take timing into account yet.

3.3.1 Area

We performed an exhaustive synthesis of all the possible multipliers by 16-bit constants. According to previsions, the smallest multipliers use up no CLB (a multiplication by 0 or a power of 2 is just wiring), the largest multiplier used

4×8 CLBs (*i.e.* 8 adders), and the average CLB count was 24.2, *i.e.* a little bit more than 16/3 adders of 16 bits, due to the fact that we sometimes add a slice to compute $-x$, and also to the wasted slice when the core uses an odd number of slices (because JBits core size is counted in CLBs of 4 LUTs, not in LUTs).

This is much better than the KCM. For example, our 221 multiplier, which is the biggest 8-bit multiplier that our generator will synthesize, is one third smaller than the JBits equivalent KCM (which in theory could be only 4 LUTs bigger, but is rounded up to the smallest CLB bounding box).

3.3.2 Synthesis time

The time currently needed to synthesize a constant multiplier core wouldn't probably qualify for a real-time reconfigurable core: although the time to compute the canonical recoding of the constant k is linear in n, which means practically less than a milisecond, computing the irregular routing takes about 500ms per slice (strangely enough, it's almost independent on the size of the adders) on a 400MHz Pentium, using Sun's Java Runtime Environment. This is to compare with the 400 ms it takes to instantiate a full 8bit KCM, whose implementation is more didactical than optimized.

Note however that we weren't using a just-in-time compiler, so some improvement may be expected there, depending on the evolution of Java technology.

3.4 Discussion

This constant multiplier generator is mostly intended to be a first demonstrator of the new possibilities offered by tools such as JBits. We should here briefly point several of its drawbacks. A more detailed discussion, especially of performance questions, can of course only occur in the context of an actual use.

Currently, the least significant bits of the result are not routed to a side of the core (contrary to what Fig 1 could lead to believe). These outputs are JBits "ports", accessible to the router without necessarily having to worry about their actual location. This is a very convenient feature, but not necessarily a desireable one from a performance point of view.

More importantly, our cores will be difficult to pipeline, both internally (due to the lack of registers) and externally (due to the constant-dependent timing). This is especially a problem as most filters involving constant multiplications may be heavily pipelined. It will be interesting to see how irregular operators behave in such a context, but this study is definitely out of the scope of this paper.

4 Work in progress

4.1 Bernstein and Lefèvre multipliers

We are currently investigating several other techniques to minimize the number of additions. The question is, what size can we expect? It is easy to build a worst-case k (*i.e.* the k which maximizes the number of adds or LUTs) for most variants of Bernstein's algorithm, and this worst case is of the order of $n/2$ additions. Lefèvre algorithm is more complex, and we haven't built a generic worst case. Exhaustive tests for n between 8 and 24 bits show that the algorithm never generates more than $0.4n$ adders. This is however a minor improvement over the $n/2$ of canonical recoding, all the more as

- the size of the adders is no longer constant, and it needs to be taken into account to evaluate the cost in LUTs of the worst case, and
- the routing is here even more irregular than in the canonical recoding case.

Experimental results [8] suggest that the average number of additions produced by Lefèvre's algorithm is $O(n^{0.85})$, which is encouraging. For $n = 32$, the average number of additions is 8, and for $n = 64$ it is 14.5. Here again we still have to evaluate the CLB count, since the size of the adders varies.

On the subject of routing, it is interesting to note that Bernstein algorithm only generates additions and subtractions involving the result of the previous stage and the initial x, which is easier to route as the more general shift-and-add chains of Lefèvre's algorithm. However Bernstein is impractical for n greater than 24, and we will concentrate on Lefèvre's algorithm. For a small n, it is always possible to put the two algorithms in competition and to chose the best result for the core implementation.

Considering the previous study, our current project is to

- adapt Lefèvre algorithm to the limitations of FPGAs, and evaluate the cost of its result more precisely (worst case and average, CLB count and not only add count),
- explore other algorithms (for example, using double base number systems, or compression algorithms *à la* Lempel-Ziv),
- implement the best algorithms in JBits,
- test the area and speed of these constant multipliers in situation (in FIRs or FFTs with real-world constants).

5 Conclusion

This paper is a survey of the constant multiplication problem in the case of FPGAs. Its contribution is to show that modern FPGA development frameworks such as JBits, being based on general-purpose programming languages, allow designers to consider arithmetic cores which are much more irregular than what was previously possible. Arbitrarily complex optimization can and must take place while programming the "hardware" part of reconfigurable systems.

We have detailed the example of a constant multiplier generator based on canonical recoding which, although conceptually simple, is much better in terms of LUT usage than the classical KCM approach. This example is hopefully the first in a series of novel arithmetic approaches for the configurable computing era.

References

[1] D. Benyamin, W. Luk, and J. Villasenor. Optimizing FPGA-based vector product designs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1999.

[2] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[3] K.D. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, May 1994.

[4] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object oriented circuit-generators in java. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1998.

[5] Digital Equipment Corporation. *PamDC: a C++ Library for the Simulation and Generation of Xilinx FPGA Designs*, 1997.

[6] S. A. Guccione and D. Levi. Run-time parameterizable cores. In *FPL'99*, pages 215–222, Glascow, August 1999. Springer Verlag, LNCS 1673.

[7] K. Hwang. *Computer Arithmetic Principles, Architecture and Design*. Wiley, New York, 1979.

[8] V. Lefèvre. Multiplication by an integer constant. LIP research report RR1999-06 (`ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/-RR/RR1999/RR1999-06.ps.Z`), Laboratoire d'Informatique du Parallélisme, Lyon, France, 1999.

[9] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[10] Mahesh Mehendale, S. D.Sherlekar, and G. Venkatesh. Synthesis of multiplier-less FIR filters with minimum number of additions. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 668–671, San Jose, CA USA, November 1995.

[11] Mahesh Mehendale, G. Venkatesh, and S. D.Sherleka. Optimized code generation of multiplication-free linear transforms. In *ACM IEEE Design Automation Conference*, page 41, Las Vegas, NV USA, November 1996.

[12] M. Potkonjak, M.B. Srivastava, and A. Chandrakasan. Efficient substitution of multiple constant multiplications by shifts and additions using iterative pairwise matching. In *ACM IEEE Design Automation Conference*, pages 189–194, San Diego, CA USA, June 1994.

[13] Xilinx Corporation. *Constant (K) Coefficient Multiplier Generator for Virtex*, March 1999.