Some Optimizations of Hardware Multiplication by Constant Matrices

Nicolas Boullis, Arnaud Tisserand

To cite this version:

Nicolas Boullis, Arnaud Tisserand. Some Optimizations of Hardware Multiplication by Constant Matrices. IEEE Transactions on Computers, 2005, 54 (10), pp.1271-1282. 10.1109/TC.2005.168 . lirmm-00113092

HAL Id: lirmm-00113092

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00113092

Submitted on 10 Nov 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Some Optimizations of Hardware Multiplication by Constant Matrices

Nicolas Boullis and Arnaud Tisserand, *Member*, *IEEE*

Abstract—This paper presents some improvements on the optimization of hardware multiplication by constant matrices. We focus on the automatic generation of circuits that involve constant matrix multiplication, i.e., multiplication of a vector by a constant matrix. The proposed method, based on number recoding and dedicated common subexpression factorization algorithms, was implemented in a VHDL generator. Our algorithms and generator have been extended to the case of some digital filters based on multiplication by a constant matrix and delay operations. The obtained results on several applications have been implemented on FPGAs and compared to previous solutions. Up to 40 percent area and speed savings are achieved.

Index Terms—Computer arithmetic, multiplication by constants, common subexpressions sharing, FIR filter.

1 Introduction

Important optimizations of the speed, area, and power consumption of circuits can be achieved by using dedicated operators instead of general ones whenever possible. Multiplication by constant is a typical example. Indeed, if one operand of the multiplication is constant, one can use some shifts and additions/subtractions to perform the operation instead of using a complete multiplier. This usually leads to smaller, faster, and less power-consuming circuits.

Applications involving multiplication by constant are common in digital signal processing, image processing, control, and data communication. Finite impulse response (FIR) filters, discrete cosine transform (DCT), and discrete Fourier transform (DFT), for instance, are central operations in high-throughput systems and they use a huge amount of such operations. Their optimization widely impacts the performance of the global system that uses them. In [1], there is an analysis of the frequency of such operations.

The problem of the optimization of multiplication by constant has been studied for a long time. For instance, the famous recoding presented by Booth in [2] can simplify both the multiplications by constants and the complete multiplications. This recoding and the algorithm proposed by Bernstein in [3] were widely used on processors without a multiplication unit.

The main goal in this problem is the minimization of the computation quantity. The multiplication by constant problem seems to be simple, but its resolution is a hard problem due to its combinatorial properties. This problem can occur in more or less complex contexts. In the case of a single multiplication of one variable by one small constant, it may be possible to explore the whole parameter space. But, in the case of the multiplication of several variables by several constants, the space to explore is so huge that one has to use heuristics.

A first solution proposed to optimize multiplication by constant was the use of the constant recoding, such as Booth's. This solution just avoids long strings of consecutive ones in the binary representation of the constant. Better solutions are based on the factorization of common subexpressions, simulated annealing, tree exploration, pattern search methods, etc.

Our work deals with multiplication of constant matrix, i.e., one useful form of the multiplication of several variables by several constants. A lot of applications involve such linear operations. This method is based on constants recoding followed by some dedicated common subexpression factorization algorithms. We also extended our method to the case of some digital filters. Our solution is able to handle filters based on constant matrix multiplication and delay operations (such as FIR filters). The proposed method was implemented in a VHDL generator. The generated results for several applications have been implemented on Xilinx FPGAs (field programmable gate arrays) and compared to other solutions. Some significant improvements have been obtained: up to 40 percent area saving in the DCT case and from 20 percent up to 30 percent in the case of some FIR filters, for instance.

This paper is an extended version of the paper [4] presented at the 16th IEEE Symposium on Computer Arithmetic (ARITH16) in June 2003. It is organized as follows: The problem is presented in Section 2. In Section 3, some related works are presented. Our algorithm is presented in Section 4. The developed generator and the target architectures are discussed in Section 5. The results of the implementation of some applications and their comparison to other solutions are presented in Section 6. Finally, the specific case of digital filters is presented in Section 7.

2 Problem Definition

In this paper and in the related works, the central problem is the substitution of complete multipliers by an optimized

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

Manuscript received 24 Nov. 2003; revised 24 Mar. 2005; accepted 6 Apr. 2005; published online 16 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0225-1103.

sequence of shifts and additions and/or subtractions. We focus on integers, but all the results can be easily extended to other fixed-point representations.

All the values are represented using a standard unsigned radix-2 notation or two's complement unless it is specified. The notation $x \ll k$ denotes the k-bit left shift of the variable x (i.e., $x \times 2^k$). As we look at hardware implementations, we assume that shift is just routing and that addition and subtraction have the same area and speed cost.

As an example, let us compute p as the product of the input variable x by the constant $c = 111463 = 11011001101100111_2$. The simplest algorithm uses the distributive property of multiplication. There is one addition of x (after some potential shift) for each one in the binary representation of c. In the case $c = 111463$, it leads to 10 additions:

$$\begin{aligned}111463x = &(x \ll 16) + (x \ll 15) + (x \ll 13) + (x \ll 12) \\ &+ (x \ll 9) + (x \ll 8) + (x \ll 6) + (x \ll 5) \\ &+ (x \ll 2) + (x \ll 1) + x.\end{aligned}$$

The central point in this problem is the minimization of the total number of operations. It can be significantly reduced by using a recoding of the constant and/or subexpression elimination and sharing. The theoretical complexity of this problem still seems to be unknown.

Depending on the target application, this problem can occur at different levels of complexity. It starts with the multiplication of one variable by one constant. After, the multiple constant multiplication (MCM) problem appears with the multiplication of one variable by several constants [5]. In this present work, we deal with a more general version of this problem with the multiplication of one variable vector by one constant matrix: the constant matrix multiplication. We also deal with the case of some digital filters that involve multiplication by a constant matrix and delay operations.

3 Related Works

There are at least four types of methods to address the multiplication by constant problem:

- direct recoding methods,
- evolutionary methods,
- cost-function-based search methods, and
- pattern search methods.

3.1 Direct Recoding Methods

The recoding of the multiplier operand is very frequently used in multipliers. The famous Booth's recoding [2] replaces long strings of ones by values with more zeros. The modified Booth recoding is often used in variable multipliers because it reduces the area of the operators. See [6] or [7] for the use of Booth or modified Booth recodings for multiplication. But, the Booth recoding is generally not used in constant multipliers because the number of nonzero digits of the recoded operand is not minimal.

A minimal recoding ensures that the number of nonzero digits in the recoded value is as small as possible. In the radix-2 signed digit (SD) representation, the digits belong to the set $\{\bar{1} = -1, 0, 1\}$. A number is said to be in the canonical signed digit (CSD) format if no two nonzero digits are consecutive; such a code is minimal. Using a minimal recoding, such as CSD, on an n-bit unsigned value, the number of nonzero digits is bounded by $(n+1)/2$ and it tends asymptotically to an average value of $n/3 + 1/9$, as shown in [8]. For our example, using CSD recoding we have: $111463 = 11011001101100111_2 = 100\bar{1}0\bar{1}0100\bar{1}0\bar{1}00\bar{1}_2$ and the product $p = c \times x$ is reduced to seven additions/subtractions:

$$\begin{aligned}111463x = &(x \ll 17) - (x \ll 14) - (x \ll 12) + (x \ll 10) \\ &- (x \ll 7) - (x \ll 5) + (x \ll 3) - x.\end{aligned}$$

The KCM algorithm [9] was specifically designed for LUT-based FPGAs (LUT means look-up table). It decomposes the binary representation of the variable into 4-bit chunks (a radix-16 representation). Each partial product, deduced by the product of the constant by one radix-16 digit of the variable, is read in a small multiplication table. Those tables are addressed by one radix-16 digit, which perfectly fulfills the 4-input LUT resources of the target FPGAs. There is a more general version of this decomposition problem with distributed arithmetic. For instance, in [10], distributed arithmetic was used on a 16-point DCT operator with an area saving of 17 percent compared to the direct implementation of the whole computation.

There are some recent works on the use of high-radix recoding. For instance, in [11], the authors implement some FIR filters using a radix-8 representation with punctured coefficients. Those coefficients are represented using digits in the set $\{0, \pm1, \pm2, \pm4\}$ instead of the set $\{0, \pm1, \pm2, \pm3, \pm4\}$. This is a lossy representation, so they have to deal with some additional accuracy requirements. In our case, we want to study this problem for a lossless representation, but this approach seems to be interesting for future research.

The recoding of the constants using sum of power of two (SOPOT) values is a standard method. In this method, the theoretical coefficients are quantified to values that can be expressed using a small number of nonzero bits (compared to the whole word length). This method is often used in signal processing filters, see [12] and [13] for recent filter applications.

Another recoding solution was proposed with the use of multiple-radix representations and especially with the double-base number system (DBNS) [14]. In this solution, the authors use both radices 2 and 3 simultaneously, i.e., the values are expressed by $a = \sum_{i,j} a_{i,j} 2^i 3^j$ with $a_{i,j} \in \{0, 1\}$. This multiple-radix representation, sometimes useful in some analog circuits, does not seem to be efficient in the multiplication by constant problem in digital circuits. In [15], multiple-radix or mixed-radix representations have been used in the implementation of FIR multirate converters. A small area gain is reported using this kind of representation.

3.2 Evolutionary Methods

Some evolutionary methods, such as evolutionary graph generation [16], have been proposed to generate arithmetic circuits and especially for constant multipliers. These methods based on genetic algorithms seem to provide very poor results. For instance, in [16], the results are slightly better than the straightforward CSD encoding, which is

very far from the best known results. Furthermore, it seems that these methods are limited to the problem of multiplication by a few constants and have never been used to produce more complex circuits.

3.3 Cost-Function-Based Search Methods

The algorithm presented by Bernstein in [3] allows some intermediate values that are used only once in recoding methods to be reused. A more detailed and corrected version of this algorithm can be found in [17]. The algorithm, based on a tree exploration, defines three kinds of operations: $t_{i+1} = (t_i \ll k)$, $t_{i+1} = (t_i \pm x)$, and $t_{i+1} = ((t_i \ll k) \pm t_i)$. A cost can be specified for each operation according to the target technology. The cost function used to guide the exploration is the sum of the costs of all the involved operations. This algorithm only shares some common subexpressions. For our example, $p = c \times x$ with $c = 111463$, this algorithm gives a 5-addition solution:

$$t_1 = (((x \ll 3) - x) \ll 2) - x,$$
$$t_2 = t_1 \ll 7 + t_1,$$
$$p = (((t_2 \ll 2) + x) \ll 3) - x.$$

There are some other cost-function-based search methods such as simulated annealing. In [18], this technique was used to produce multiplication by a small set of constants. The same multiplier is used for a small set of different coefficients. This problem is different from ours.

In [19], a greedy algorithm is used to determine a solution with a low total operation cost. A 28 percent average area saving is achieved on some controllers and elliptic filters. This solution seems to be limited due to local attraction of the greedy algorithm.

3.4 Pattern Search Methods

Most of the pattern search methods are based on the same general idea. The algorithm recursively builds a set of constants to be optimized. This set is initialized with the recoded initial constants (generally using the CSD format). The different methods differ in the way they match the common subexpressions and the way they share and reuse them.

The multiple constant multiplication (MCM) solution presented in [5] performs a tree exploration with selection of matching parts of the SD representation of the constants. This paper is the most cited one and it presents a lot of details about the algorithm as well as about the comparisons.

In [20], the matches between constants are represented using a graph. The exploration and some transformations of this graph are used to produce a specific form of FIR filters with a reduced number of adders/subtractors while controlling the operator delay.

A solution based on an algebraic formulation of the possible matches between constants is presented in [21]. Unfortunately, the authors use random filters for their tests without specifying the coefficients. So, it is difficult to compare their results to other solutions.

A recent work [22] proposes sharing digits in the CSD representation of the coefficient matrix both in a horizontal and in a vertical way. This solution allows circuits to be designed with 10 percent fewer adders/subtracters than the straightforward CSD horizontal subexpression factorization.

In [23], a pattern search method is proposed. Some optimizations on the result architecture are done such as the transformation of multiple subtractions of the same value into the negation of this value and several adders. This kind of optimization can lead to significant improvement in ASICs where subtractors are larger than adders. This is not the case in our FPGAs.

In [24], a factorization method based on the selection of the best pair of matching digits is used. This solution can be easily extended to the selection of common parts of words larger than two digits.

We will base our solution on extensions and improvements of the algorithms presented in [25] and [26]. A detailed description of this idea is presented below. One can notice that, among all the abundant bibliography about the multiplication by constant problem, there is no general solution to the multiplication by constant matrix problem.

4 Proposed Algorithms

4.1 Lefèvre's Algorithm

In 2001, Lefèvre proposed a new algorithm to efficiently multiply a variable integer by a given set of integer constants [25]. As a special case, this algorithm was used to multiply a variable by a single constant.

4.1.1 Definitions

The principle of the algorithm is to handle a list of constants to be optimized and to find a "pattern" that appears several times in the set of constants. The constants are recoded using the CSD format in the very beginning. A pattern is a sequence of digits in $\{\bar{1}, 0, 1\}$. The number of nonzero digits in the pattern is called its weight.

A pattern P is said to occur in a constant C with a shift α when, for each 1 in position k of P, there is a 1 in position $k + \alpha$ in C and, for each $\bar{1}$ in position k of P, there is a $\bar{1}$ in position $k + \alpha$ in C. And, a pattern is said to occur negatively when there is a $\bar{1}$ in C for each 1 in P and a 1 in C for each $\bar{1}$ in P. This last point is one of the main differences between the two papers, [25] and [26]. Lefèvre's algorithm allows us to use patterns negatively, which leads to slightly better optimizations.

When two occurrences of the same pattern or of different patterns match the same nonzero digit of the constant, the two occurrences are said to conflict. For example, in the number $51 = 10\bar{1}010\bar{1}_2$, the pattern $10\bar{1}$ occurs positively with shift 0, negatively with shift 2, and positively with shift 4. The first and third occurrences both conflict with the second one. And, the pattern 10001 occurs negatively with shift 0 and positively with shift 2. Those occurrences overlap, but do not conflict. Moreover, every occurrence of the $10\bar{1}$ pattern conflicts with every occurrence of the 10001 pattern.

4.1.2 Description of the Algorithm

The principle of the algorithm can be described by the pseudocode presented in Algorithm 1.

Algorithm 1 : Principle of Lefèvre's algorithm.

While $\left(\begin{array}{l}\text{there are some patterns with weight} \geq 2 \\ \text{and at least 2 non-conflicting occurrences}\end{array}\right)$
    choose a pattern with maximal weight and 2 non-conflicting occurrences
    remove the chosen pattern from both occurrences
    add the pattern as a new constant

Then, multiplication by each constant in the final set can be implemented in the usual way: For each 1 ($\bar{1}$) in position p, add (subtract) x shifted by p bits to the left. And then, by rolling back the algorithm, each constant can be computed by shifts and additions/subtractions, with roughly one addition/subtraction for each chosen occurrence of a pattern.

On our previous example $p = c \times x$, Lefèvre's algorithm gives a solution with only four additions:

$$t_1 = (x \ll 3) - x,$$
$$t_2 = (t_1 \ll 2) - x,$$
$$p = (t_2 \ll 12) + (t_2 \ll 5) + t_1.$$

4.2 Extensions and Enhancements to Lefèvre's Algorithm

Mathematically speaking, Lefèvre's algorithm deals with the multiplication of a number by a constant vector. The first thing to do is to extend it for the multiplication of a vector by a constant matrix. This extension is rather straightforward: We simply replace each constant with a constant vector. Patterns are then replaced by vectors of patterns and shifts are performed componentwise. With this algorithm, it is possible to share all kinds of expressions.

For example, let us consider the computation of $y_1 = 5x_1 + 5x_2 + x_3$ and $y_2 = 5x_1 + 5x_2 + 4x_3$. The algorithm will first share the computation of $5x_1 + 5x_2$ between y_1 and y_2. After that, it will share $x_1 + x_2$ in $(5x_1 + 5x_2) = 4(x_1 + x_2) + (x_1 + x_2)$, effectively sharing the multiplication by 5 between x_1 and x_2. This example shows that the algorithm deals with both dimensions of the constant matrix.

A detailed description of our extended algorithm is given in the Appendix. This description, in C-like pseudocode, presents the overall behavior of our algorithm.

One point is kept unspecified in Lefèvre's algorithm: Which maximal pattern and which occurrences should we choose? In his original implementation, Lefèvre simply chose the first maximal pattern he found, with the first two occurrences. This solution is probably not the best, so we tried to find something better.

The first idea was to find all the maximal-weight patterns with at least two nonconflicting occurrences and all their occurrences. And then, we try to choose a set of patterns and, for each pattern, a set of at least two occurrences such that two chosen occurrences (of the same pattern or of different patterns) do not conflict. The choice is performed in order to maximize the gain in the weight of all the constants; with a constant with weight w and i occurrences, we gain $(i-1)(w-1)$ times its weight. As all the chosen patterns have the same maximal weight, we want to maximize the sum, for each pattern, of the number of occurrences diminished by one.

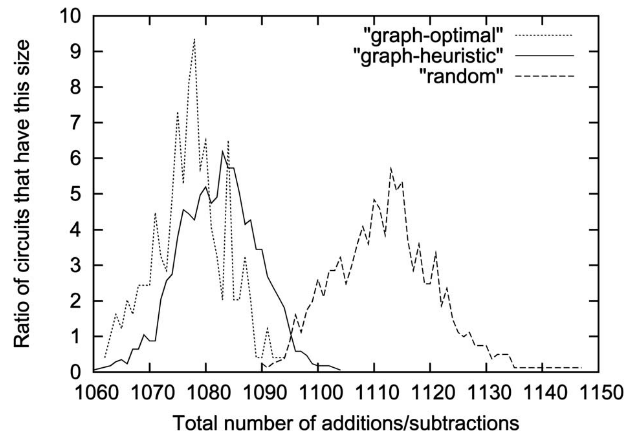

Fig. 1. Heuristics influence distributions (from many optimizations of one large 2D IDCT operator).

We tried three different solutions for this. The first one, called "random," and which is the closest to the original algorithm, is to recursively choose, at random, a pattern with two nonconflicting occurrences and to remove everything that conflicts with these occurrences. The second one, called "graph-heuristic," is to recursively choose a pattern with a maximal set of nonconflicting occurrences and a minimal set of conflicts with the other patterns and then remove everything that conflicts with these occurrences. And the third one, called "graph-optimal," is to build all the maximal sets of patterns and nonconflicting occurrences and to choose the best one. This last solution can be very computationally intensive.

We tried to compare those three solutions, by running them several times for the same constant matrix: a huge standard 8×8 points 2D IDCT (inverse DCT) operator with 14-bit words. The results in Fig. 1 show that the "graph-optimal" and "graph-heuristic" are roughly equivalent and better than the "random," with a tiny advantage to "graph-optimal." The time required to generate these results is less than one minute for "graph-heuristic" and "random," while it can grow to hours for "graph-optimal." Hence, we generally choose the "graph-heuristic" solution so we can perform lots of tries (thanks to its speed) and then choose the best solution. Similar results have been obtained using other applications.

4.3 Beyond the Mathematical Optimization

The improvements described above only deal with the minimization of the total number of additions and subtractions. Translated to hardware, this is not enough. Some additions and subtractions can be reordered without changing their total number thanks to properties such as associativity and commutativity.

First of all, one may want to have a small circuit. When three numbers a, b, and c are added, the order in which they are added influences the size of the adders. For example, if a and b are narrow numbers, while c is wide, computing $(a+b)+c$ leads to a smaller circuit than $(a+c)+b$ or

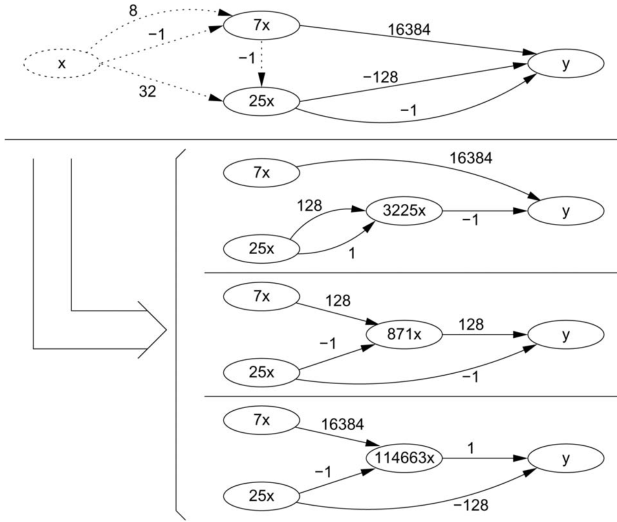

Fig. 2. Postoptimization of the multiplication by 111463 for area target.

$(b + c) + a$. Hence, for space optimization, it is generally better to add the narrowest numbers first.

On the other hand, one may want to have a fast circuit. The order in which three numbers are added also influences the worst-case delay of the circuit. For example, if a and c are available early while b is available late, the result is available earlier if we compute $(a + c) + b$ than $(a + b) + c$ or $(b + c) + a$. Hence, for speed optimization, it is preferable to add the earliest available values first.

Those optimizations are performed in two steps. At first, we unconstrain the order of the operations as much as possible. When a value is used only once, its computation is merged into the value that uses it. For example, if we had $t_1 = (x \ll 2) - x$ and $t_2 = (x \ll 6) - (t1 \ll 2) + x$, it becomes $t_2 = (x \ll 6) - (x \ll 4) + (x \ll 2) + x$. Similarly, if a term is composed of only one term, its computation is also merged into any value that uses it. This part is done by a traversal of the dataflow graph that represents the computation.

After removing all those useless constraints, we want to set new constraints that meet our goal of high speed or low area. In the data-flow graph, this is done by splitting nodes with more than two terms. We do this with a hierarchical traversal of the graph: A node is only considered after all its ancestors. This is possible because our dataflow graphs are acyclic. When we meet a node with more that two terms during the traversal, we extract two of its terms to make a new node, as illustrated in Fig. 2. We consider all the pairs of terms of the node. Each such pair can be assembled to a new intermediate value. It is possible to symbolically compute each corresponding value and deduce how wide the corresponding adder would be. And, as we are using a hierarchical traversal, we can also compute when each of those values would be available.

If we want to optimize for area, we select a pair from among those that require the smaller adder. On the other hand, if we want to optimize for speed, we select it from among those that would be available the earliest. Then, the corresponding new node is generated and replaces the two former terms: Now, there is one fewer term. We iterate that extraction of two terms until the considered node has only two terms.

Back to our example in Fig. 2, there are three possibilities that use $3225x$, $871x$, or $114663x$ as new intermediate values. Obviously, computing $871x$ requires a smaller adder than the other two; that solution would be the chosen one for area optimization. About speed, all three intermediate values would be available after three adder steps (from the input x), so they are equivalent. This simple example shows that the postoptimizations lead to significant improvements. In Section 6, a larger example (based on an IDCT operator) confirms these improvements using postoptimizations (see Table 5).

When we try to optimize for area, if several possibilities are equivalent, we choose among them with the speed criterion, so the circuit is not uselessly slow. The opposite is, of course, true as well.

Moreover, the algorithmic optimization is not enough. We need to generate some real circuits. Hence, we decided to generate some VHDL code. Although it may work for any target, our VHDL code generator is currently optimized for Xilinx FPGAs. So, additions and subtractions are performed using the dedicated fast carry-propagate adders and subtractors. The generator is able to produce VHDL code for fully parallel circuits or for digit-serial circuits with radix 2^n for any n. Only parallel architectures are available when delays are involved (e.g., filters).

5 Implementation

Our implementation is mainly in two parts. The first part performs the mathematical optimization, with our extended and enhanced version of Lefèvre's algorithm. This part was written in C++ and is approximately 1,500 lines long. This part is not a program by itself, but a collection of simple classes that can be easily interfaced with any C++ program. Hence, it would be easy, for example, to interface this with a program that computes coefficients for FIR filters. Then, the user would simply choose the type of filter and the required frequencies and attenuations and the program would compute the coefficients and generate some efficient VHDL code for it.

After the mathematical optimization, everything is implemented as plug-ins. Hence, there are, for example, plug-ins that optimize the order of the additions and subtractions or plug-ins that generate the output VHDL code. This structure with plug-ins makes the whole thing very modular. Hence, if someone wants, for example, to generate some Verilog code or some assembly language code for a DSP, it is sufficient to write a new output plug-in. Then, if someone wants to get pipelined circuits, a new pipelining plug-in can be written and it can then be used with any output plug-in. Those plug-ins are also written in C++. The collection of plug-ins is currently approximately 2,500 lines long.

The plug-ins communicate between themselves and with the main program with simple interfaces that describe the circuit as a data-flow graph. In this representation, vertices represent mathematical values. Hence, there are vertices for input values, for output values, and also for intermediate

```
#include "decompose.h"
#include <iostream>

int main(int argc, char **argv) {

  int value[] = {111463};

  Expr::initialize(argc, argv);

  CombinationSet<int> work(1);
  work.addLine(value);

  Key<int> *k;
  k = work.findBestPattern();
  while ((k!=NULL)&&(k->weight()>1)) {
    work.applyKey(*k, &std::cout);
    delete k;
    k = work.findBestPattern();
  }
  work.finish();
  return 0;
}
```

Fig. 3. Multiplication by 111463 optimization source code.

values. Then, there is an edge, from vertex x to vertex y, tagged with $(shift, sign, delay)$, if x shifted $shift$ bits to the left and a delay of $delay$ clock cycles is a positive or negative part (according to $sign$) of y. The $delay$ part is used for filters or pipelined circuits. This representation has the quality of being independent of the desired output.

Let us give a simple example of how this can be used and what is generated. As a simple example, we will consider building a constant multiplier by 111463. The corresponding source code and generated VHDL are presented in Fig. 3 and Fig. 4, respectively.

6 Results and Comparisons

The syntheses of this section have been performed using Xilinx ISE XST 4.2.03i tools for a Virtex XCV200-5 FPGA.

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_arith.all;

entity MULT_111463 is
  port (
    x1  : in  std_logic_vector (7 downto 0);
    y1  : out std_logic_vector (24 downto 0)
  );
end MULT_111463;

architecture struct of MULT_111463 is
  signal t3     : std_logic_vector (22 downto 0);
  signal t2     : std_logic_vector (15 downto 0);
  signal t1     : std_logic_vector (22 downto 0);
begin -- struct
-- t3     = 20641*x1
  t3 <= ( t2(15)&t2(15)&t2(15 downto 0) & "00000" )
       +( t1(22 downto 0) );
-- t2     = 129*x1
  t2 <= ( x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)
          &x1(7)&x1(7 downto 0) )
       +( x1(7)&x1(7 downto 0) & "0000000" );
-- t1     = 16513*x1
  t1 <= ( t2(15)&t2(15)&t2(15)&t2(15)&t2(15)&t2(15)
          &t2(15)&t2(15 downto 0) )
       +( x1(7)&x1(7 downto 0) & "00000000000000" );
-- y1     = 111463*x1
  y1 <= ( t1(21 downto 0) & "000" )
       -( t3(22)&t3(22)&t3(22 downto 0) );
end struct;
-- Number of additions: 4
-- Estimated number of FA cells:  61
```

Fig. 4. Multiplication by 111463 generated VHDL.

TABLE 1
Number of Additions/Subtractions Comparison for Some 1D 8-Point DCT Operators

operator	initial	[5]	[24]	our
DCT 8b	300	94	73	56
DCT 12b	368	100	84	70
DCT 16b	521	129	114	89
DCT 24b	789	212	—	119

TABLE 2
Synthesis Results of Some 1D 8-Point DCT Generated Operators

operator	synthesis		generator	
	# slices	delay [ns]	# $\pm$	# FA
DCT 8b	401 (17%)	19.5	56	739
DCT 12b	647 (27%)	21.7	70	1202
DCT 16b	1085 (46%)	25.7	89	2009
DCT 24b	2106 (89%)	27.9	119	3934

The operators are not pipelined. The area is measured in number of slices (two LUTs with 4 address bits per slice in Virtex devices). The required area compared to the 2,352 available slices in an XVC200 device is also reported in parentheses. The delay is expressed in nanoseconds. The number of additions/subtractions and the number of FA cells are computed by our generator (see the two last lines of Fig. 4); the number of FA cells is only an estimation (assuming the use of carry ripple adders).

Only a few papers give enough elements to compare to our solutions. In [5] and [24], there are useful values for the DCT application. Table 1 presents the number of additions/subtractions for the 1D 8-point DCT for several word sizes. Our generator improves the previous results from 17 percent to 44 percent. Table 2 gives the synthesis results for the corresponding generated operators.

We performed some other comparisons on some error-correcting codes from [5] and [24]: the 8×8 Hadamard matrix transform, $(16, 11)$ Reed-Muller, $(15, 7)$ BCH, and $(24, 12, 8)$ Golay codes. The comparison with the previous works in [5] and [24] is presented in Table 3 and the corresponding synthesis results are presented in Table 4. These results show that, for very simple operators such as a small BCH code, some improvements are still possible. In the case of the 8×8 Hadamard matrix transform, we obtained the same results as in the previous work [5].

Table 5 presents the synthesis results of the same IDCT operator with the three possible postoptimizations of our generator: none, area, or speed. The operator is a 1D 8-point IDCT for 14-bit constants and 8-bit inputs. From the same initial additions/subtractions number, the optimizations presented in Section 4.3 lead to significant improvements, 40 percent for the speed optimization for instance. The generation time for all these operators is around a few seconds on a standard desktop computer.

In Section 4.3, we explained that our generator can produce digit-parallel as well as digit-serial circuits using different output plug-ins. Table 6 presents the synthesis results of a 1D 8-point IDCT operator for several solutions: digit-parallel and radix-2, 4, 8, 16, 64, and 256 digit-serial versions. Digit-serial implementations lead to small area

TABLE 3
Number of Additions/Subtractions Comparison for Some Error-Correction Benchmarks

operator	initial	[5]	[24]	our
8×8 Had.	56	24	—	24
$(16, 11)$ R.-M.	61	43	31	31
$(15, 7)$ BCH	72	48	47	44
$(24, 12, 8)$ Golay	76	—	47	45

TABLE 4
Synthesis Results for Some Error-Correction Benchmarks

operator	synthesis		generator	
	# slices	delay [ns]	# ±	# FA
8×8 Had.	128 (5%)	11.9	24	240
$(16, 11)$ R.-M.	39 (1%)	12.1	31	86
$(15, 7)$ BCH	461 (19%)	18.2	44	861
$(24, 12, 8)$ Golay	63 (2%)	12.2	45	136

and short cycle time operators. But, in order to fairly compare digit-serial versus digit-parallel solution, we should compare with the pipelined parallel operator.

In [27], an algorithm for designing multiplication by matrix operators is presented. The proposed algorithm has been tested on $n \times n$ matrices with random 8-bit integer coefficients. In Fig. 5, we compare our generator with the results from [27] (only values for n between 2 and 6 are reported in [27]). Our complete results on random matrices with 8-bit integer coefficients are reported in Table 7. The average number of additions and subtractions and its standard deviation have been evaluated on 100 random matrices for each size. The reported generation time is the average value for the generation time of one matrix. Our results show slightly better performances.

7 Extension to Digital Filters

Digital filters are a very specific case of multiplication by a constant matrix. They are linear combinations of the input, delayed several times:

$$y[t] = \sum_{i=0}^{n} a_i x[t-i],$$

where $x[i]$ is the ith value of the sampled signal x.

Such filters are generally implemented using one of two different kinds of architectures. The first one delays the input to compute all the $x[t-i]$ and then computes their linear combination (the multiplication by the constant matrix). The second one computes all the $a_i x[t-i]$ and then delays them and adds them to form the result as depicted in Fig. 6. In signal processing, the first form of the filter is called the *direct form*, while the second one is called the *transposed form*. We call the gray part of Fig. 6 the multiplication block (MB).

These implementation solutions consider the computations to be independent and do not allow sharing results between consecutive computations. We extended our algorithm to be able to apply such optimizations. As an example, let us consider the following trivial low-pass FIR filter:

$$y[t] = x[t] + 5x[t-1] + 5x[t-2] + x[t-3].$$

The direct form of the filter leads to three delay units to compute the $x[t-i]$ and then five additions to compute $y[t]$ (Fig. 7A). This can be reduced to three delay units and four additions by using the symmetry of the coefficients (Fig. 7B). The transposed form leads to one addition to compute the values $x[t]$ and $5x[t]$ and then three delay units and three additions to compute $y[t]$, which gives a total of three delay units and four additions (Fig. 7C).

If we allow sharing of intermediate results between computations using our generator, we can first compute $z[t] = x[t] + x[t-1]$, which requires one delay unit and one addition, and then compute $y[t] = z[t] + 4z[t-1] + z[t-2]$, which requires two delay units and two additions; this gives a total of three delay units and three additions (Fig. 7D). It is, of course, equivalent to first computing $z'[t] = x[t] + 4x[t-1] + x[t-2]$ and then $y[t] = z'[t] + z'[t-1]$; this is the architecture found by our generator (Fig. 7E). An extract of the generated VHDL code corresponding to this last architecture is shown in Fig. 8.

TABLE 5
Influence of the Generator Optimizations on a 1D 8-Point IDCT Operator

operator	synthesis		generator	
	# slices	delay [ns]	# ±	# FA
IDCT	769 (32%)	30.2	81	1382
IDCT area	665 (28%)	19.8	81	1196
IDCT speed	666 (28%)	18.0	81	1241

TABLE 6
Synthesis Results for 1D 8-Point Digit-Parallel and Digit-Serial IDCT Operators

operator	# slices [ns]	delay
parallel	614	40
serial radix-2	85	22
serial radix-4	153	36
serial radix-8	194	46
serial radix-16	242	47
serial radix-64	349	47
serial radix-256	446	48

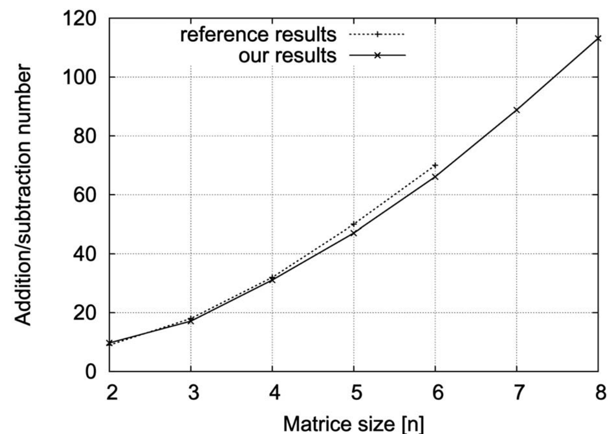

Fig. 5. Comparison of the results from [27] with ours on $n \times n$ random matrices with 8-bit integer coefficients.

TABLE 7
Results for $n \times n$ Random Matrices with 8-Bit Integer Coefficients

n	average # ±	standard deviation	generation time [s]
2	9.7	1.3	.04
3	17.1	0.9	.07
4	31.2	2.2	.14
5	47.1	3.3	.37
6	66.1	4.0	.80
7	88.9	5.3	1.54
8	113.2	6.7	2.69
9	141.6	7.0	4.55
10	172.4	8.5	7.28
11	207.1	10.8	10.62
12	241.6	11.9	16.21
13	279.6	13.3	23.86
14	322.9	17.0	32.59
15	370.0	20.0	44.94
16	412.4	19.4	57.30

Of course, this trivial example only shows that it may be possible to reduce the computational cost of an FIR filter by sharing some results between consecutive computations. It is not supposed to establish a rule about how efficient it is; this will be shown by implementing some real FIR filters.

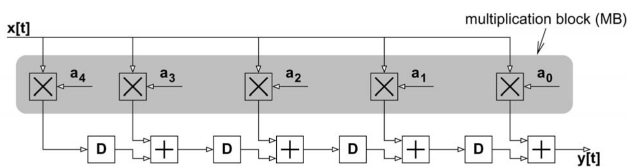

Fig. 6. The transposed form of an FIR filter.

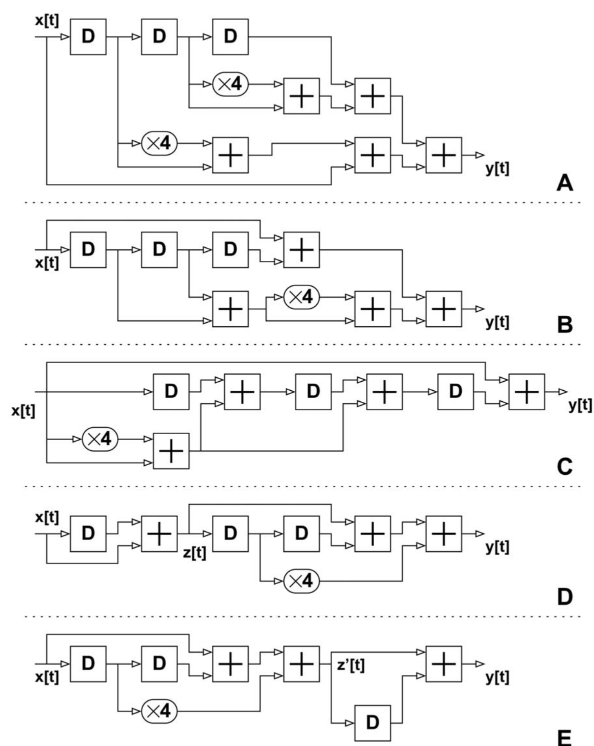

Fig. 7. FIR (1, 5, 5, 1) architectures.

```
...
architecture struct of fir1551 is
...
begin
  x1_d(0) <= x1;
  dl1: for k in 0 to 1 generate
    r1: reg generic map (n=>8)
           port map (Ck=>clk,i=>x1_d(k),o=>x1_d(k+1));
  end generate;
  t2_d(0) <= t2;
  dl2: for k in 0 to 0 generate
    r2: reg generic map (n=>11)
           port map (Ck=>clk,i=>t2_d(k),o=>t2_d(k+1));
  end generate;
-- t1    = x1+x1[2]
  t1    <=  SXT(x1_d(2) , 9) + SXT(x1 , 9);
-- t2    = x1+4*x1[1]+x1[2]
  t2    <=  SXT(x1_d(1) & "00" , 11) + SXT(t1 , 11);
-- y1    = x1+5*x1[1]+5*x1[2]+x1[3]
  y1    <=  SXT(t2 , 12) + SXT(t2_d(1) , 12);
end struct;
```

Fig. 8. FIR (1, 5, 5, 1) generated VHDL (extract).

For this new extension, we use the Z transform of the FIR filter, which is very common in digital signal processing: FIR filters are represented by polynomials in Z^{-1}. For example, our (1, 5, 5, 1) FIR filter is represented by the polynomial $1 + 5Z^{-1} + 5Z^{-2} + Z^{-3}$. In those polynomials, multiplying a signal by Z^{-1} means delaying that signal by one delay unit. Such a polynomial represents one single FIR filter. But, as our algorithm is already able to deal with several inputs and several outputs, we thought it would be useful to be able to deal with filters with several inputs and outputs. Such complex filters may be used, for example, to implement a digital audio equalizer or a digital DTMF (dual-tone-multi-frequency) decoder.

Therefore, an extension works by replacing the matrix of constants by a matrix of polynomials in Z^{-1}. Then, if we perform the optimization exactly as before, no pattern is shared between subsequent computations; this corresponds to the direct form of the filter. To implement such sharings, we must allow multiplication of the patterns by Z^{-1}, just as we allowed shifting them to the left. By doing so, the introduced delays are not taken into account for the optimization and only the number of additions/subtractions is optimized.

This generally results in a huge increase of the number of registers, with little to no gain to the number of additions/subtractions. This may be acceptable when programming some DSP processors, but it is not for hardware implementations. To prevent that huge increase, it is possible to set a limit to the number of multiplication of a pattern by Z^{-1}. Thus, it is possible to control the number of added registers.

TABLE 8
Specifications of the Example Filters Presented in [28] and Matlab Command Used to Generate the Coefficients (Attenuation and Ripple Values Are Theoretical Values)

filter	# tap	normalized frequencies		stop-band attenuation	pass-band ripple
		pass-band	stop-band		
example 1	25	0.15	0.25	46.0 dB	0.05 dB
	remez(24, [0 0.3 0.5 1], [1 1 0 0])				
example 2	59	0.021	0.07	61.7 dB	0.2 dB
	remez(59, [0 0.042 0.14 1], [1 1 0 0], [1 14])				

TABLE 9
Comparison of the Implementation of Low-Pass FIR Filters from [28]

Filter	coefficients # bits		# ±				implemented filters	
			[28] results		our results		stop-band	pass-band
	total	non-zero	MB	total	MB	total	attenuation	ripple
[28] example 1	9	2 (3)	11	36	6	30	43.8 dB	0.05 dB
[28] example 2	14	3 (4)	57	116	33	89	60.5 dB	0.2 dB

In [28], the optimization of low-pass FIR filters using sum of power of two (SOPOT) coefficients is presented. The method is demonstrated on two filters. The specifications of these two filters are reported in Table 8. This table also report the `remez` Matlab function call used to generate the theoretical coefficients of the filters.

In Table 9, we compare the implementation results from [28] with our method. For our generator input we use the optimized SOPOT coefficients presented in [28] in order to achieve the same stop-band attenuation and pass-band ripple values. On the first example from [28], nine digits ({-1, 0, 1}) SOPOT coefficients are used with at most two nonzero digits expect for large values where three digits are allowed. In the second example, 14 bits SOPOT coefficients are used with at most three (or four) nonzero digits. In Table 9, two values are reported for the number of addition/subtraction: *total* for the whole filter and *MB* only for the multiplication block of the transposed form (see Fig. 6).

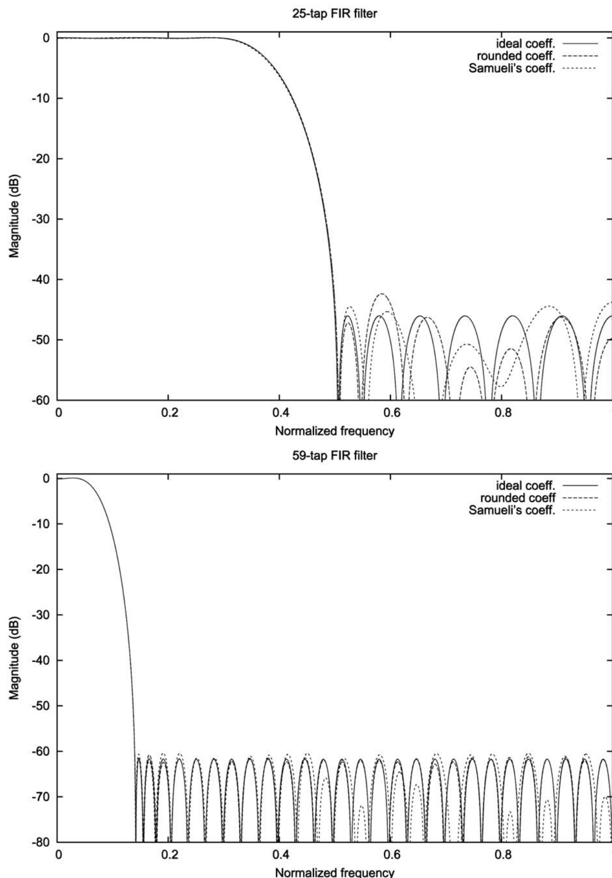

Fig. 9. Frequency response of the filters compared with [28].

The normalized frequency response of the two filters (theoretical, rounded, and generated filters) are reported in Fig. 9.

We also implemented in FPGA some low-pass FIR filters with specifications derived from [20]. The corresponding results are presented in Table 11. The specifications of those filters are presented in Table 10. The coefficients have been generated using the `remez` Matlab functions $c_1 = \texttt{remez}(\#\text{tap}, [0 f_p f_s 1], [1100])$ and $c_2 = \texttt{round}((2 \wedge \text{width}) * c_1)$. The values reported in Table 10 represent the complete filter, while the number of adders reported in [20] only represent the multiplication block, an additional adder should added for each tap of the filter.

For each filter from [20], we tried to implement it with a delay limit (denoted by DL in the result tables) set to 0 (no sharing between consecutive calculations), 1, 2, or ∞ and the resulting VHDL code was optimized for speed using Xilinx ISE XST 5.2.03i tools for a Virtex-II 1000 FPGA (XC2V1000-5) on 1.7 GHz Pentium4 PC with 1GB RAM. The operators are not pipelined. The required area, compared to the 5,120 available slices in a XC2V1000 device, is reported in parentheses. We also report the delay of the operator and

TABLE 10
Low-Pass FIR Filters Specifications from [20]

filter	f_p	f_s	#tap	width
FIR 1	0.15	0.25	60	14
FIR 2	0.15	0.20	60	16
FIR 3	0.10	0.15	60	14
FIR 4	0.10	0.12	100	18

TABLE 11
FPGA Synthesis Results of the Low-Pass FIR Filters from [20]

filter	DL	# slices	delay [ns]	T_{synth} [s]
FIR 1	0	1364 (26%)	37.2	51
	1	1377 (26%)	36.2	57
	2	1392 (27%)	35.2	57
	∞	2033 (39%)	33.5	84
FIR 2	0	1655 (32%)	38.4	60
	1	1762 (34%)	37.1	67
	2	1911 (37%)	35.8	76
	∞	4141 (80%)	36.5	215
FIR 3	0	1293 (25%)	34.8	49
	1	1420 (28%)	32.3	60
	2	1503 (30%)	32.1	77
	∞	2238 (43%)	32.9	105
FIR 4	0	3126 (61%)	41.7	114
	1	3247 (63%)	47.7	129
	2	3486 (68%)	41.8	145
	∞	(>250%)	—	—

TABLE 12
Parks-McClellan Filter Coefficients Specification

coefficient	value
$h(0)$	-0.00933078669
$h(1)$	0.07628237421
$h(2)$	0.03135623682
$h(3)$	0.01374432164
$h(4)$	-0.00948598843
$h(5)$	-0.03358586396
$h(6)$	-0.04680063247
$h(7)$	-0.03819695824
$h(8)$	-0.00271831937
$h(9)$	0.05563093697
$h(10)$	0.12405515375
$h(11)$	0.18473033065
$h(12)$	0.22024453765

TABLE 13
Number of Additions/Subtractions Comparison for the Parks-McClellan Filter

operator	initial	[22]	our
8 bits	35	32	24
16 bits	72	70	46

its total synthesis time (including place and route optimizations) using high effort constraints.

In Table 11, one can see that the operator period is generally reduced when increasing the delay limit (DL). This effect is due to the additional registers that break long paths in the circuit. This permits reorganization of the schedule with a shorter critical path. But, when the number of additional registers is too large, new long lines of routing are involved in the circuit. This explains the larger delay that sometimes occurs for large values of DL.

We did another experiment on an FIR filter from [22], implemented both with 8 and 16 bits of accuracy. This filter is based on the Parks-McClellan design of a low-pass 26-tap FIR filter with pass-band and stop-band edges at 0.2 and 0.25, respectively. The corresponding coefficients are presented in Table 12 (only the first 13 coefficients are reported because of the symmetry); those

TABLE 14
Parks-McClellan Filters Synthesis Results

operator	DL	# slices	delay [ns]	T_{synth} [s]
8 bits	0	297 (6%)	27.0	22
	1	306 (6%)	27.7	23
	2	275 (5%)	22.4	22
	∞	415 (8%)	22.4	25
16 bits	0	711 (13%)	34.4	33
	1	747 (14%)	31.2	34
	2	755 (15%)	33.3	35
	∞	1569 (31%)	33.5	66

coefficients can be computed using Matlab with the command $\texttt{remez}(25, [0 0.2 0.25 1], [1 1 0 0])$.

The comparison of our results with those from [22] is presented in Table 13. Our solution leads to a reduction of the operation count of about 25 percent for 8-bit coefficients and 34 percent for 16-bit coefficients. The results of the FPGA implementation of generated architectures are presented in Table 14.

Even with no sharing between consecutive computations, our algorithm already gives a very small number of operations (less than two additions/subtractions per tap). This gives little room for improvement. Hence, when the delay limit rises, the number of operations does not shrink much, while many registers are added to share intermediate results. This explains why the size rises with the delay limit. On the other hand, the delay is generally reduced, around 9 percent on average and up to 17 percent. This proves that this sharing is still useful when speed is a main concern.

8 Conclusion

A new algorithm for the problem of multiplication by constant was presented. We generalized the previous results by dealing with the problem of the optimization of multiplication of one vector by one constant matrix. Our algorithm is based on extensions and enhancements of previous algorithms from [25] and [26]. Compared to the best previous results, our solution leads to a significant drop in the total number of additions/subtractions, up to 40 percent.

```
procedure optimize(args: constants):
  constants := [csd(constants)]
  names := ["y"]
  share := search_best_shares(constants)
  while share != FAILURE
    (pattern,occurrences) := share
    pattern_name := new_name()
    for occ in occurrences
      (index,shift,sign) := occ
      constants[index] := constants[index] - sign * pattern << shift
      display(names[index]," := ",names[index],sign,"(",pattern_name," << ",shift,")")
    constants := constants + [pattern]
    names := names + [pattern_name]
    share := search_best_shares(constants)
  # all optimizations are done
  # ``flush'' the last constants
  for index in [1..size(constants)]
    c := constants[index]
    for rank in [0..size(c)]
      if c[rank] = 1
        display(names[index]," := ",names[index]," + (x << ",rank,")")
      else if c[rank] = -1
        display(names[index]," := ",names[index]," - (x << ",rank,")")
    display(names[index]," := 0")
```

Fig. 10. Pseudocode of procedure `optimize`.

```
function search_best_shares(lines):
  best_gain := 2
  best_patterns := {}
  for (index1,index2) in [1..size(lines)]^2
      such that index1 <= index2
    l1 := lines[index1]
    l2 := lines[index2]
    for (delay1,delay2) in [(0,0)..(+infinity,0)] union [(0,0)..(0,+infinity)]
        such that l1 div (Z-1)^delay1 != 0
             and l2 div (Z-1)^delay2 != 0
             and (index1 != index2 or delay1 <= delay2)
      ll1 := l1 div (Z-1)^delay1
      ll2 := l2 div (Z-1)^delay2
      for (shift1,shift2) in [(0,0)..(+infinity,0)] union [(0,0)..(0,+infinity)]
          such that l1 >> shift1 != 0
               and l2 >> shift2 != 0
               and (index1 != index2 or delay1 != delay2 or shift1 < shift2)
        line1 := ll1 >> shift1
        line2 := ll2 >> shift2
        for sign in {+,-}
          pattern := line1 & sign * line2
          share := (pattern,[(index1,delay1,shift1,+),(index2,delay2,shift2,sign)])
          if estimated_gain(share) >= best_gain and index1 = index2
            # eliminate internal conflicts
            true_patterns := eliminate_internal_conflicts(pattern,sign,shift1,
                                                  shift2,delay1,delay2)
          else
            # there is no internal conflict : this is a true pattern
            true_patterns := {pattern}
          for word in true_patterns
            share := (word,[(index1,delay1,shift1,+),(index2,delay2,shift2,sign)])
            gain := estimated_gain(share)
            if gain >= best_gain
              if gain > best_gain
                best_shares := {}
                best_gain := gain
              # Normalization of the pattern
              share := normalize(share)
              best_shares := best_shares union {share}
  return best_shares
```

Fig. 11. Pseudocode of function `search_best_shares`.

```
function eliminate_internal_conflicts(args: pattern,sign,shift1,shift2):
  conflicts := pattern << shift1 & sign * pattern << shift2
  while conflicts != 0
    pattern := pattern - least_significant_digit(conflicts) >> shift1
    conflicts := pattern << shift1 & sign * pattern << shift2
  return pattern
```

Fig. 12. Pseudocode of function `eliminate_internal_conflicts`.

We implemented this algorithm in a VHDL generator. Based on a simple mathematical description of the computation, the generator produces an optimized VHDL code for Xilinx FPGAs. At the moment, the generated operators are nonpipelined parallel or digit-serial ones. We will extend our generator to produce pipelined circuits to reach higher clock frequencies.

We also extended our algorithm and generator to the case of some digital filters. We are now able to handle filters involving a multiplication by constant matrix and delay operations (such as FIR filters). In the case of a 26-tap 16-bit FIR filter, a 34 percent reduction of the operation count is achieved, compared to recent results from [22]. These first results on filter optimization are promising; we now plan to work on the synthesis of filters in the near future.

We want to extend our algorithm and generator to standard-cell-based ASICs. The way to implement the adders/subtracters would widely impact the performance of the complete operator. The optimization required for low-power consumption may also change our solutions.

Another area to explore in the future is the use of lossy representations, such as [11]. In a lot of applications, the models include some approximations and the quantization of the coefficients. It may be a good idea to allow small perturbations of the coefficients.

Appendix

Detailed Algorithms

Figs. 10, 11, and 12 are C-like pseudocode versions of our extended algorithm presented in Section 4. Procedure `optimize`, Fig. 10, is the main entry point.

Acknowledgments

The authors would like to thank the "*Ministère Français de la Recherche*" (grant # 1048 CDR 1 "*ACI jeunes chercheurs*") and the Xilinx University Program for their support. They also want to thank the anonymous reviewers. Their comments and corrections were very useful in improving the paper.

References

[1] D.J. Magenheimer, L. Peters, K.W. Pettis, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Trans. Computers*, vol. 37, no. 8, pp. 980-990, Aug. 1988.

[2] A.D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical Applications of Math.*, vol. IV, no. 2, pp. 236-240, 1951.
[3] R. Bernstein, "Multiplication by Integer Constants," *Software—Practice and Experience*, vol. 16, no. 7, pp. 641-652, July 1986.
[4] N. Boullis and A. Tisserand, "Some Optimizations of Hardware Multiplication by Constant Matrices," *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH 16)*, J. -C. Bajard and M. Schulte, eds., pp. 20-27, June 2003.
[5] M. Potkonjak, M.B. Srivastava, and A.P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 2, pp. 151-165, Feb. 1996.
[6] M.D. Ercegovac and T. Lang, *Digital Arithmetic.* Morgan Kaufmann, 2003.
[7] M.J. Flynn and S.F. Oberman, *Advanced Computer Arithmetic Design.* Wiley-Interscience, 2001.
[8] R.I. Hartley, "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677-688, Oct. 1996.
[9] K.D. Chapman, "Fast Integer Multipliers Fit in FPGAs," *EDN Magazine*, May 1994.
[10] S. Yu and E.E. Swartzlander, "DCT Implementation with Distributed Arithmetic," *IEEE Trans. Computers*, vol. 50, no. 9, pp. 985-991, Sept. 2001.
[11] P. Boonyanant and S. Tantaratana, "FIR Filters with Punctured Radix-8 Symmetric Coefficients: Design and Multiplier-Free Realizations," *Circuits Systems Signal Processing*, vol. 21, no. 4, pp. 345-367, 2002.
[12] C.K.S. Pun, S.C. Chan, K.S. Yeung, and K.L. Ho, "On the Design and Implementation of FIR and IIR Digital Filters with Variable Frequency Characteristics," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 11, pp. 689-703, Nov. 2002.
[13] S.C. Chan and W.L.K.L. Ho, "Multiplierless Perfect Reconstruction Modulated Filter Banks with Sum-of-Powers-of-Two Coefficients," *Signal Processing Letters, IEE*, vol. 8, no. 6, pp. 163-166, 2001.
[14] V.S. Dimitrov, G.A. Jullien, and W.C. Miller, "Theory and Applications of the Double-Base Number System," *IEEE Trans. Computers*, vol. 48, no. 10, pp. 1098-1106, Oct. 1999.
[15] J. Li and S. Tantaratana, "Multiplier-Free Realizations for FIR Multirate Converters Based on Mixed-Radix Number Representation," *IEEE Trans. Signal Processing*, vol. 45, no. 4, pp. 880-890, Apr. 1997.
[16] N. Homma, T. Aoki, and T. Higuchi, "Evolutionary Graph Generation System with Transmigration Capability and Its Application to Arithmetic Circuit Synthesis," *IEE Proc.*, vol. 149, no. 2, pp. 97-104, Apr. 2002.
[17] P. Briggs and T. Harvey, "Multiplication by Integer Constants," technical report, Rice Univ., 1994.
[18] M.F. Mellal and J.-M. Delosme, "Multiplier Optimization for Small Sets Of Coefficients," *Proc. Int'l Workshop Logic and Architecture Synthesis*, pp. 13-22, Dec. 1997.
[19] H.T. Nguyen and A. Chatterjee, "Number-Splitting with Shift-and-Add Decomposition for Power and Hardware Optimization in Linear DSP Synthesis," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 4, pp. 419-424, Aug. 2000.
[20] H.-J. Kang and I.-C. Park, "FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 8, pp. 770-777, Aug. 2001.
[21] M. Martínez-Peiró, E.I. Boemo, and L. Wanhammar, "Design of High-Speed Multiplierless Filters Using a Nonrecursive Signed Common Subexpression Algorithm," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 3, pp. 196-203, Mar. 2002.
[22] A. Vinod, E.-K. Lai, A. Premkumar, and C. Lau, "FIR Filter Implementation by Efficient Sharing of Horizontal and Vertical Common Subexpresions," *Electronics Letters*, vol. 39, no. 2, pp. 251-253, Jan. 2003.
[23] A. Yurdakul and G. Dündar, "Fast and Efficient Algorithm for the Multiplierless Realisation of Linear DSP Transforms," *IEE Proc. Circuits, Devices, and Systems*, vol. 149, no. 4, pp. 20-211, Aug. 2002.
[24] A. Matsuura, M. Yukishita, and A. Nagoya, "A Hierarchical Clustering Method for the Multiple Constant Multiplication Problem," *IEICE Trans. Fundamentals of Electronics, Comm., and Computer Sciences*, vol. E80-A, no. 10, pp. 1767-1773, Oct. 1997.
[25] V. Lefèvre, "Multiplication par une Constante," *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, vol. 13, nos. 4-5, pp. 465-484, 2001.
[26] R. Paško, P. Schaumont, V. Derudder, S. Vernalde, and D. Ďuračková, "A New Algorithm for Elimination of Common Subexpressions," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58-68, Jan. 1999.
[27] A.G. Dempster, O. Gustafsson, and J.O. Coleman, "Towards an Algorithm for Matrix Multiplier Blocks," *Proc. European Conf. Circuit Theory Design*, Sept. 2003.
[28] H. Samueli, "An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Power-of-Two Coefficients," *IEEE Trans. Circuits and Systems*, vol. 36, no. 7, pp. 1044-1047, July 1989.

Nicolas Boullis was a student at the École Normale Supérieure de Lyon, where he learned computer science. During his studies, he showed some interest in various research topics in computer science, such as multicast in active networks, designing some arithmetic blocs for FPGA, design of asynchronous dividers, and, more recently, hardware multiplication by constants, which is the topic of his PhD thesis (which he will be defending in mid-2005). He now works at the École Centrale de Paris as a system engineer.

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

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.