Self-Reconfigurable Constant Multiplier for FPGA

JAVIER HORMIGO, Universidad de Málaga
GABRIEL CAFFARENA, Universidad CEU San Pablo
JUAN P. OLIVER, Universidad de la República
EDUARDO BOEMO, Universidad Autónoma de Madrid

Constant multipliers are widely used in signal processing applications to implement the multiplication of signals by a constant coefficient. However, in some applications, this coefficient remains invariable only during an interval of time, and then, its value changes to adapt to new circumstances. In this article, we present a self-reconfigurable constant multiplier suitable for LUT-based FPGAs able to reload the constant in runtime. The pipelined architecture presented is easily scalable to any multiplicand and constant sizes, for unsigned and signed representations. It can be reprogrammed in 16 clock cycles, equivalent to less than 100 ns in current FPGAs. This value is significantly smaller than FPGA partial configuration times. The presented approach is more efficient in terms of area and speed when compared to generic multipliers, achieving up to 91% area reduction and up to 102% speed improvement for the case-study circuits tested. The power consumption of the proposed multipliers are in the range of those of slice-based multipliers provided by the vendor.

Categories and Subject Descriptors: B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Constant multiplier, runtime reconfiguration, FPGA

ACM Reference Format:
Hormigo, J., Caffarena, G., Oliver, J. P., and Boemo, E. 2013. Self-reconfigurable constant multiplier for FPGA. *ACM Trans. Reconfig. Technol. Syst.* 6, 3, Article 14 (October 2013), 17 pages.
DOI: http://dx.doi.org/10.1145/2490830

1. INTRODUCTION

FPGA devices offer high computational power and reconfiguration capabilities that allow for complete customization and runtime adaptation of the circuit functionality. Many applications require these features in order to adjust the design to particular parameters at any given time step. Reconfiguration is a thriving field, but the long reconfiguration times and storage or generation of the new bitstream are still important issues [Compton et al. 2002; Dandalis and Prasanna 2005; Kalra and Lysecky 2010].

In signal processing, it is usual that one of the multiplier operands is a constant coefficient. Thus, it is possible to optimize its hardware structure to improve area

This work was partially supported by projects P07-TIC-02630 (Junta of Andalucía), TIN2006-01078 (Ministry of Education and Science of Spain), and USP-BS PPC05/2010 (University CEU San Pablo and Banco Santander).
Authors' addresses: J. Hormigo, Department of Computer Architecture, University of Málaga, Málaga, Spain; email: hormigo@ac.uma.es; G. Caffarena (corresponding author), Department of Information Technologies, University CEU San Pablo, Spain; email: gabriel.caffarenafernandez@ceu.es; J. P. Oliver, Facultad de Ingeniería, Universidad de la República, Uruguay; E. Boemo, Universidad Autónoma de Madrid, Spain.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1936-7406/2013/10-ART14 $15.00
DOI: http://dx.doi.org/10.1145/2490830

usage and computation speed with respect to conventional blocks [Gustafsson 2007; Xu et al. 2008]. However, in some applications, these constants may change in certain time steps, which prevents the use of standard constant multipliers [Bosí et al. 1999; Bouganis et al. 2009; Huang et al. 2008; Shoufan et al. 2010]. Some researchers [Chen and Chang 2009; Demirsoy et al. 2007; Turner and Woods 2004] have addressed this problem when the constant changes to several predefined values, as it does in FFT, DCT, filters, and many others. Other authors have proposed reconfigurable architectures for specific applications (i.e., FIR filters [Mahesh and Vinod 2010; Park et al. 2004]) that enable the use of a priori unknown constants. In this article, we propose a new reconfigurable constant multiplier that combines the advantages of the previous works, covering a wider range of applications (e.g., adaptive filters, neural networks, channel equalization, gain control, cryptography, etc.).

There are many approaches that tackle constant multiplier design, but not all of them are suitable for runtime reconfiguration to any constant value. For instance, the shift and add method produces efficient designs for most of the cases, although the resulting architectures and their features (i.e., area, delay, etc.) strongly depend on the constant value [Gustafsson et al. 2006; Nguyen and Chattejee 2000]. Other ideas are based on storing the results of partial products in lookup tables (LUT), which are added to compose the final multiplication value [Chapman 1996; Meher 2010; Wirthlin 2004]. In Wirthlin [2004], the application of such methods to FPGA designs are thoroughly studied. The main advantage of this method is that the architecture is fixed disregarding the value of the constants: only the values of the tables change. Thus, such multipliers are easier to design or to generate automatically. Therefore, they can be straightforwardly reconfigured if a different constant is required. Also, their area-time-power figure is known, and the same multipliers structure is utilized, independently of the constant value. In this article, we select this approach as the starting point to implement an on-the-fly reconfigurable constant multiplier.

Our proposed circuit enhances the regular LUT-based constant multiplier from Wirthlin [2004], enabling real-time reconfiguration of the constant with no restrictions on the possible constant values. The latter point is achieved by dedicating a portion of the multiplier to compute the contents of the LUTs, given a particular constant value. Thus, the multiplier is capable of reconfiguring itself. The circuit is able to self-reconfigure without implementing any standard FPGA reconfiguration techniques. Therefore, it can perfectly replace conventional multipliers in those applications where one of its operands changes its value only at particular time steps, being constant during long intervals of time (i.e., n clock cycles or n data computations). Domains that are suited well for this situation are cryptography, gain control, channel equalization, etc.

In this article, we use middle-cost Spartan-3 or Virtex-4 families Xilinx FPGA as the technology framework for validating the proposed ideas. Nevertheless, most of the concepts can be easily extended to high-end current FPGAs (which have different LUT size) (see Section 3.3). The work is divided into the following parts: Section 2 explains the fundamental ideas about LUT-based constant-coefficient multipliers optimized for FPGA devices. Section 3 deals with the architecture of the proposed self-reconfigurable multiplier. In Section 4, the implementation results and the comparison to other approaches are shown. Finally, conclusions are drawn in Section 5.

2. LUT-BASED CONSTANT MULTIPLIERS

In this section, we present the mathematical modeling which supports the design of LUT-based constant multipliers. Additionally, the specific architecture for FPGAs proposed in Wirthlin [2004] is shown, since it has been chosen as a starting point to develop our self-reconfigurable multiplier.

Table I. Content of Partial Product Tables

Unsigned		Signed	
d_i	Output	$d_{m/3}$	Output
000	$0 \cdot K$	000	$0 \cdot K$
001	$1 \cdot K$	001	$1 \cdot K$
010	$2 \cdot K$	010	$2 \cdot K$
011	$3 \cdot K$	011	$3 \cdot K$
100	$4 \cdot K$	100	$-4 \cdot K$
101	$5 \cdot K$	101	$-3 \cdot K$
110	$6 \cdot K$	110	$-2 \cdot K$
111	$7 \cdot K$	111	$-1 \cdot K$

2.1. Theory

First, let us assume that data is composed of natural numbers. All the concepts explained can be easily applied to fixed-point real numbers. The main idea is to decompose the nonconstant m-bit multiplicand (A) in digits of the same size (i.e., q bits, where $q < m$).

$$A = \sum_{i=1}^{\lceil m/q \rceil} d_i 2^{q \cdot (i-1)}, \tag{1}$$

where each digit complies with $0 \leq d_i < 2^q$. Therefore, the product of A by the constant K is expressed as

$$K \cdot A = \sum_{i=1}^{\lceil m/q \rceil} (K \cdot d_i) 2^{q \cdot (i-1)}. \tag{2}$$

The values of the partial products of all possible digits by the constant (i.e., from $K \cdot 0$ to $K \cdot (2^q - 1)$) are stored in tables. The final results are obtained by means of adding the partial products $K \cdot d_i$, applying the corresponding shifting, as shown in Equation (2). The value of parameter q allows a trade-off between the amount of memory and the number of addition operations. Thus, as long as the value of q increases, the number of additions required is reduced, but more memory is needed. For FPGA implementations, q is usually chosen to match the number of inputs of the FPGA LUT.

The extension of this method to signed constants is direct. The partial products must be stored in two's complement (TC) format, and the addition must use sign extension. If the multiplicand is represented using TC format, then only the most significant partial product lookup table has to be changed, since all digits (d_i) are unsigned except for the most significant one ($d_{\lceil m/q \rceil}$). Thus, for this digit, the values of the partial product from $-2^{q-1} \cdot K$ to $(2^{q-1} - 1) \cdot K$ have to be stored in the table. The correct storing order would be from $0 \cdot K$ to $(2^{q-1} - 1) \cdot K$ and from $-2^{q-1} \cdot K$ to $-1 \cdot K$. Table I shows an example of the values stored for unsigned and TC digits, supposing $q = 3$.

2.2. Architecture

There are several adder architectures able to perform the partial product summation. In this article, we focus on a cascaded array of carry-propagated adders. It provides a good trade-off between area and speed when implemented in FPGAs, and it can be directly pipelined. For the sake of clarity, the technique is illustrated for generic four-input LUTs ($LUT4$). The value q is selected to 3 instead of 4, since this allows for an important area reduction, as we will explain next.

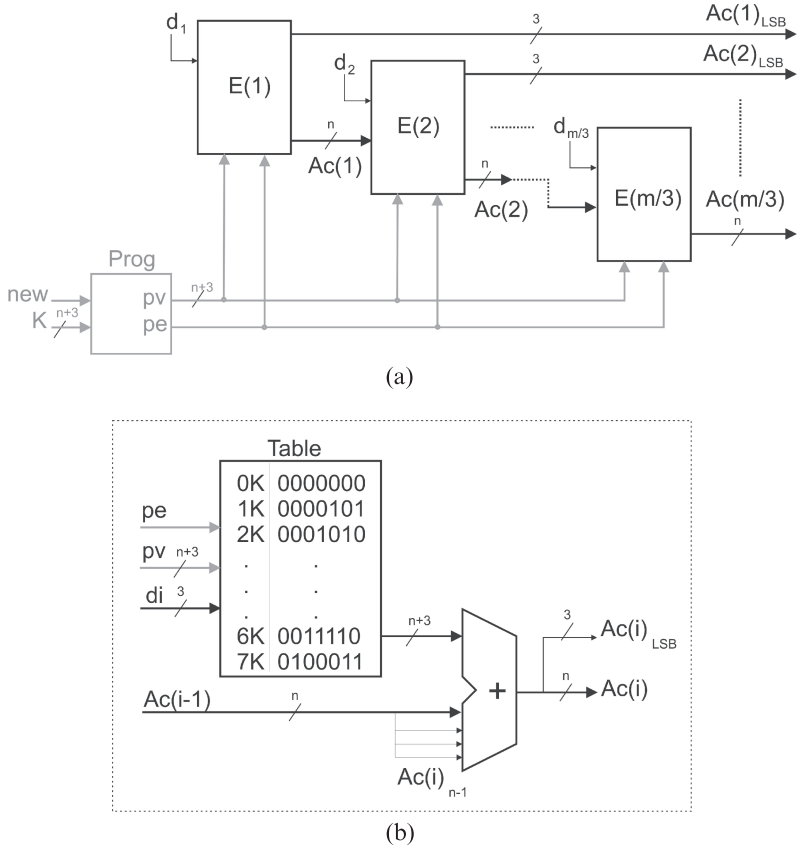

Fig. 1. Architecture for constant multiplier: (a) main blocks ($E(i)$) and additional circuit for reprogramming (in gray); (b) inner details of an $E(i)$ block particularized for $K = 5$.

Figure 1(a) displays the architecture selected for multiplying an n-bit constant by an m-bit multiplicand (black ink). Each block $E(i)$ processes a three-bit digit d_i and part of the accumulated partial product $Ac(i-1)$ from the previous stage. Internally, d_i is used to obtain the $(n+3)$-bit partial product $(K \cdot d_i)$ from the table (see Figure 1(b)). $Ac(i-1)$ is sign-extended and added to this partial product. As an example, Figure 1(b) shows the content of the table for $K = 5(0101)$. There are a total of $\lceil m/3 \rceil$ stages, each with an $(n+3)$-bit adder and a lookup table. Note that the three least significant bits of the accumulated sums $Ac(i)$ are directly connected to the final result (due to the shifting), and the remaining bits are the inputs of the next stage, $E(i+1)$). Therefore, the bit-width of the constant K determines the size of each stage $E(i)$, whereas the size of the multiplicand determines the number of stages required.

An efficient FPGA implementation of the previous scheme which reduces the overall size of the multiplier by 33% is presented in Wirthlin [2004]. The author combines the table and the addition operation corresponding to one bit within a single logic cell. A *LUT4* is used to implement a three-input table plus a half-addition, as shown in Figure 2, for bit position j. The three-bit table outputs the bit j of the partial product corresponding to digit d_i, which is added to the previous accumulated sum $(Ac(i)_j)$. The operation is completed by means of another half-addition (corresponding to the carry signal cin_j) implemented by using the specific FPGA carry-logic resource. It generates

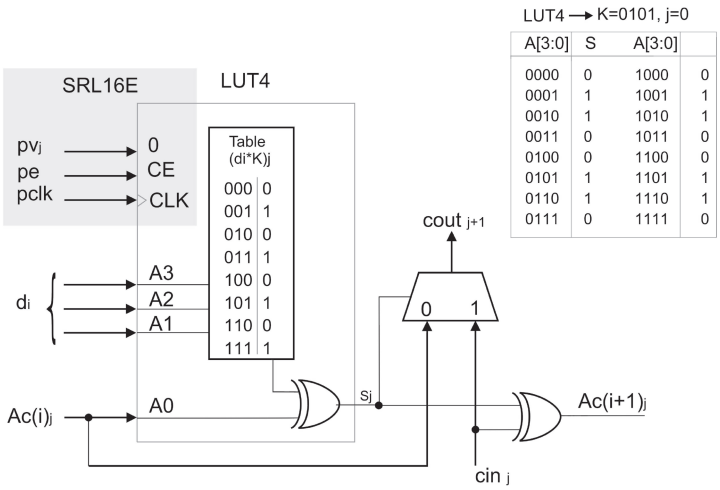

Fig. 2. LUT configuration for bit j within stage $E(i)$ (example for K = 5 and j = 0). Reconfiguration elements are in gray area.

the next accumulated sum $Ac(i+1)_j$ and carry $cout_j$. As an example, Figure 2 shows the values for the least significant bit (LSB) supposing $K = 5(0101)$.

Hence, in order to efficiently implement the architecture showed in Figure 1(b), each stage $E(i)$ is composed of $n+3$ of the circuits shown in Figure 2 working in parallel. All these circuits are interconnected through the carry chain and have the same input d_i. Each *LUT4* stores the bit j of the addition corresponding to all possible partial products and the previous accumulated sum. It must be stressed that all stages $E(i)$ are identical. The described architecture is valid for any constant value. Changing this parameter only involves modifying the data stored in the tables.

3. RUNTIME SELF-RECONFIGURATION

In this section, we propose a modification of the previous architecture to enable runtime self-reconfiguration. Two basic tasks must be carried out. First, a local mechanism to change the *LUT4*s that contain the partial products tables is included. Thus, the long transactional time involved in conventional FPGA reconfiguration is avoided. Second, the values to be stored in the *LUT4*s must be computed on-the-fly.

The first task can take advantage of the fact that in Xilinx FPGAs, the slices allow *K-LUT*s also to be used as shift registers (SRL-2^K blocks).[1] For example, in Spartan-3 and Virtex-4, it is possible to reprogram the *LUT4* by simply shifting the register 16 times. In order to add runtime reconfiguration capabilities to the constant multiplier just described, we use the *SRL16E* primitive instead of the *LUT4* primitive. The *SRL16E* has some extra inputs: shift enable (*CE*), serial input (0), and a clock signal (*clk*). The *LUT4* is loaded through the serial input by activating the shifting. After the new constant is entered, the shifting is disabled and the block acts as a constant multiplier. Self-reconfiguration can be achieved if a method for automatically obtaining the sequence of values for reconfiguration is developed. This is dealt with in the following sections.

[1] http://www.xilinx.com/support/documentation/index.htm

Table II. Content of *LUT4* Tables

A[3:0]		S	A[3:0]		S
d_i[2:0]	Ac(i)		d_i[2:0]	Ac(i)	
000	0	$0 \cdot K$	100	0	$4 \cdot K$
000	1	$\overline{0 \cdot K}$	100	1	$\overline{4 \cdot K}$
001	0	$1 \cdot K$	101	0	$5 \cdot K$
001	1	$\overline{1 \cdot K}$	101	1	$\overline{5 \cdot K}$
010	0	$2 \cdot K$	110	0	$6 \cdot K$
010	1	$\overline{2 \cdot K}$	110	1	$\overline{6 \cdot K}$
011	0	$3 \cdot K$	111	0	$7 \cdot K$
011	1	$\overline{3 \cdot K}$	111	1	$\overline{7 \cdot K}$

3.1. Unsigned Multiplicand

As we explained in Section 2.2, *LUT4* should be configured to implement the addition (i.e., the XOR) of the partial product $d_i \cdot K$ and the accumulated addition $Ac(i)$ (see Figure 2). Hence, since $0 \leq d_i < 8$, the partial products from $0 \cdot K$ to $7 \cdot K$ have to be stored if $Ac(i) = 0$. Otherwise, these values are negated. The position in the LUT of all these values depends on how the inputs $Ac(i)$ and d_i are connected. The easiest sequence to be serially generated occurs if $Ac(i)$ is connected to the least significant bit of the four inputs of the *LUT4*. This sequence corresponds to the partial products sorted in increasing order, negating those in even positions, as shown in Table II.

The generation of the programming sequence previously described is easily achieved by the following recurrence.

$$P(i+1) = \begin{cases} 0, & i = 0, \\ P(i) + 1, & i \text{ even}, \\ P(i), & i \text{ odd}, \end{cases} \tag{3}$$

$$S(i) = \begin{cases} P(i), & i \text{ even}, \\ \overline{P(i)}, & i \text{ odd}. \end{cases} \tag{4}$$

Figure 3 shows the circuit proposed for implementing Equations (3) and (4). Partial products (register P) are obtained serially by adding the constant (K) to the previous computed partial product, starting from the value 0, which corresponds to $0 \cdot K$. The next partial products are generated every two clock cycles until $7 \cdot K$ is reached. The XOR gates allow for selectively negating the partial products at each clock cycle. When a constant K is introduced and *new* is activated, the programing sequence is generated and signal *pe* (program enable) is activated during the 16 cycles. As an example, Figure 3 shows the programming values (pv) generated for $K = 5(0101)$. In each clock cycle, one column is produced, starting from the left to the right. Each row corresponds to the programming sequence (pv_j) associated with one *LUT4* according to its bit position (j). It can be seen that the highlighted row ($j = 0$) coincides with the programing values in Figure 2.

Figure 1 displays in gray how this element is connected to the constant multiplier in order to perform runtime self-reconfiguration. Now, each stage $E(i)$ is constructed using $n + 3$ *SRL16E* primitives instead of the *LUT4* of Section 2 (fixed constant case from Figure 2). As in the fixed case, these elements are arranged in parallel, with the carry signals connected from the least to the most significant one. The same d_i is shared by all of them. Signal *pe* (program enable) is connected on each stage $E(i)$ to all the *SRL16E* elements in order to allow the shift operation and, consequently, the reprogramming of the *LUT4*s. Bus pv also goes to all the stages, and each of its bits

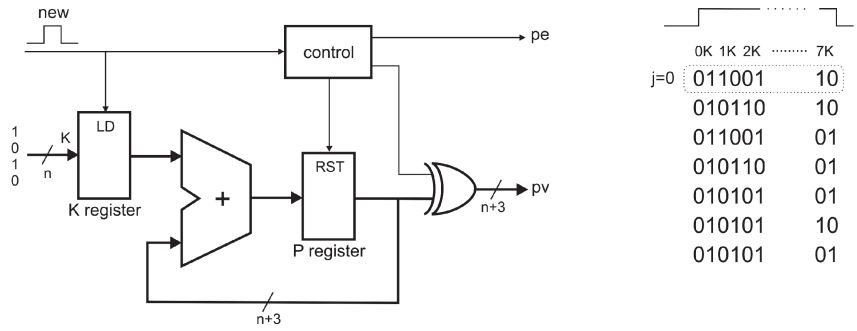

Fig. 3. Circuit for generating the new programming sequence (inverse order). Example for K = 5.

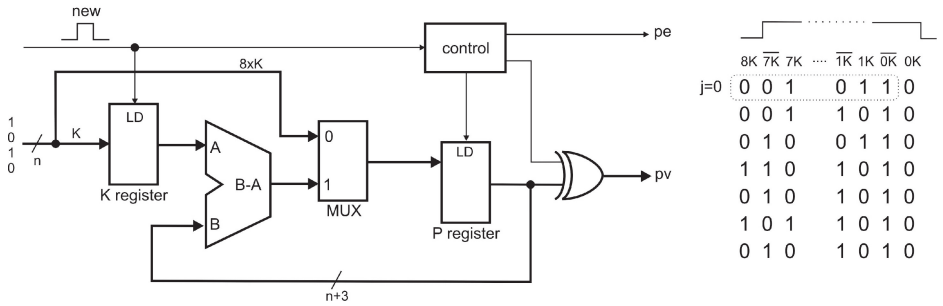

Fig. 4. Circuit for generating the new programming sequence (direct order). Example for K = 5.

is connected to the serial inputs of *SRL16E* (pv_j) according to its weight. Thus, on all stages $E(i)$, each row of values represented on Figure 3 is used to program one of the *SRL16E* elements, from $j = 0$ (top row) to $j = n + 2$ (lowest row). See how the values in row $j = 0$ from Figure 3 coincide with the values stored in the *LUT4* in Figure 2. When signal *new* is activated, the *LUT4*s are reloaded with the new programming values (according to the new constant) by shifting them during 16 clock cycles.

It must be noted that the *SRL16E* block shifts its bits from the least to the most significant ones. Thus, the programming sequence is generated in inverse order by the circuit just described. It is not difficult to design a circuit which generates the sequence correctly. It should start from $8 \cdot K$ value (by taking the constant shifted three bits to the left) and subtract K every two cycles to obtain the partial product sequence in correct order. Figure 4 shows the architecture for producing the programming sequence in direct order. In the architecture shown in Figure 3, the register P is initialized to zero by using a reset signal, whereas in the new *direct* architecture (Figure 4), this register should be initialized to $8{\cdot}K$. Thus, the reset signal cannot be used, and it is necessary to add a multiplexer at the input of register P. Therefore, as we will show in Section 4, the circuit implemented using this approach is slower than the corresponding one to the inverse method. For this reason, we use the inverse architecture, negating all the bits of the multiplicand before indexing the LUT. Thus, both approaches are now equivalent.

3.2. Signed Multiplicand

As aforementioned in Section 2, only the table corresponding to the most significant digit of the multiplicand is affected, since in TC, all bits have a positive weight except for the most significant one (the sign bit), which has a negative weight. Thus, a

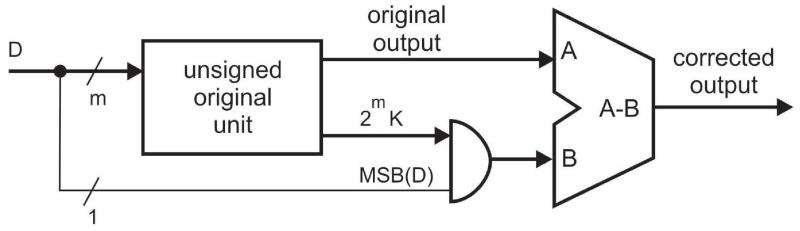

Fig. 5. Architecture for signed multiplier based on the unsigned one.

different partial product sequence should be computed ranging from $-4 \cdot K$ to $3 \cdot K$ and following the order described in Section 2.

In principle, this implies designing a different programming circuit for the last stage. As an alternative, we propose adding a correction step at the end of the signed multiplier to correct the result provided by the unsigned unit. Figure 5 shows the architecture used to build a signed multiplier based on the unsigned one. When the multiplicand is positive, the result from the unsigned unit does not require any correction. On the contrary, when the multiplicand is negative, its most significant bit is one, and then the value $K \cdot 2^{m-1}$ already has been added to the final product of the unsigned multiplier. This value should have been subtracted instead of added, since in TC representation format, this bit has a negative weight. Hence, the quantity $K \cdot 2^m$ (i.e., two times $K \cdot 2^{m-1}$) is subtracted from the output of the unsigned multiplier to obtain the correct result. Figure 5 depicts how the correction step is implemented by a final subtracter which is driven by the output of the unsigned multiplier (input A) and $2^m \cdot K$ or zero (input B), depending on the sign bit of the multiplicand.

3.3. Extension to Different FPGA Families

The ideas explained so far can be applied to any FPGA containing configurable logic blocks that can be used as lookup tables as well as shift registers. For instance, we find that this capability is supported by Xilinx FPGA families, such as Spartan-3, Virtex-4, Spartan-6, Virtex-6, Virtex-7, etc. We would like to stress that our approach is not limited to LUT4-based FPGAs and that 6-LUT-based devices can also be used. For these devices, the design presented in Section 2.2 can be applied by selecting q as 5 instead of 3. In Xilinx, the *LUT6* is implemented through two five-input *LUT*s (*LUT5*) combined by a multiplexer, and the shift register of M-slices is only associated to one of the *LUT5*s. Thus, if runtime reconfiguration is desired, q should be fixed to 4, and only one of the *LUT5*s would be used. In this case, the only difference with the architectures previously presented is that 32 cycles are required to complete the configuration of a new constant value. The runtime self-reconfiguration scheme could be applied also to other LUT-based FPGA devices, given that a mechanism for changing the content of the LUT at runtime is provided.

4. IMPLEMENTATION RESULTS AND COMPARISON

4.1. Area and Time Results

The advantages of the ideas proposed in this article are proved by implementing a parametric self-reconfigurable constant multiplier VHDL module. This block allows setting the size of both the constant and the multiplicand ($N \times M$). Also, signed or unsigned operations can be selected. The degree of pipeline depth is controlled by the number of partial product levels contained within each stage (E). For instance, $E = 1$ implies a finest-grain pipeline, and $E = \lceil m/3 \rceil$ leads to a combinational version.

Table III. Maximum Operation Frequency in MHz

E	1	2	3	4	5	6	1
N	Signed						Unsigned
6	278.0	198.2	140.5	108.8	88.8	75.0	278.6
12	259.2	186.8	134.6	105.2	86.4	73.3	278.6
18	257.1	176.6	129.2	101.9	84.1	71.6	278.6
24	238.2	167.4	124.3	98.8	82.0	70.1	256.5
32	216.9	156.6	118.2	94.9	79.3	68.1	232.0
48	183.0	138.7	107.7	88.1	74.5	64.5	194.8
64	158.2	124.5	98.9	82.1	70.2	61.2	167.8

Table IV. Occupied Area in Slices for Fully-Pipelined and Combinational Designs, Respectively

	Fully-Pipelined - Combinational						
M	6	12	18	24	32	48	64
N	Signed Operands						
6	40–32	64–51	89–70	114–89	152–118	216–165	304–222
12	64–54	99–85	136–116	172–147	228–194	322–271	446–364
18	88–75	134–118	184–161	231–204	302–269	426–376	585–505
24	111–97	170–152	230–207	288–262	378–345	532–482	727–647
32	144–125	216–196	293–267	366–338	478–445	672–622	915–835
48	210–182	318–285	428–388	537–491	700–646	983–903	1,332–1,212
64	277–240	424–375	572–510	719–645	940–848	1,320–1,185	1,786–1,590
N	Unsigned Operands						
6	36–29	60–48	85–67	110–86	149–115	212–162	298–219
12	57–48	93–79	129–110	166–141	221–188	314–265	437–358
18	78–66	125–109	181–152	236–195	320–260	459–367	573–496
24	100–85	158–140	218–195	276–250	366–333	518–470	711–635
32	127–109	201–180	276–251	350–322	462–429	655–606	896–819
48	183–158	292–261	402–364	510–467	674–622	954–879	1,302–1,188
64	241–208	387–343	535–478	682–613	903–816	1,279–1,153	1,747–1,558

The module has been synthesized with Xilinx ISE 9.2 and Spartan3E-5 devices using a wide range of parameters values ($N \times M \times E$). The results have been condensed in several tables where least-relevant parameters have been omitted.

In Table III, we present the maximum frequency of operation for the pipeline multiplier depending on the constant size (N) and the pipeline depth (E). The width of the multiplicand is omitted since it only affects the total number of stages, not the frequency, except for a combinational design. For the unsigned case, only results for fully-pipelined designs are shown, since the others are very similar to those of signed multipliers.

Table IV shows the occupied area (slices) for different sizes and signed/unsigned operands. We only show the extreme cases, that is, fully-pipelined and combinational, since the differences in area for other values of E are relatively small. The area values shown in Table IV cover the Wirthlin LUT-based multiplier [Wirthlin 2004] as well as the newly added blocks: the programmer and the correction stage for the signed case. The area increase with respect to the nonreconfigurable constant multiplier proposed by Wirthlin is shown in Table V (the value of M does not affect this quantity). Thus, the efficient architecture proposed by Wirthlin for performing constant multiplication has been modified to also support constant reconfiguration with a reasonable area increase.

Regarding delay, our proposal does not modify the critical path of the constant multiplier proposed by Wirthlin, since the hardware for reconfiguration is independent of

Table V. Increment of Area (Slices) with Respect to Nonreconfigurable Constant Multiplier

N	6	12	18	24	32	48	64
	Area in Slices						
signed	15	24	33	43	55	80	106
unsigned	12	18	24	31	39	56	74

Table VI. Reconfiguration Time

N	6	12	18	24	32	48	64
	Reconfiguration Time in ns						
direct	69	74	79	84	91	104	117
reverse	65	65	65	69	76	89	102

Table VII. Relative Increment of Maximum Operation Frequency (%) Achieved Using *OURS* for Signed Operand

M	6	12	18	24	32	48	64	ALL
N	*OURS* vs. *SLICE*							
6	7.1	21.4	32.4	44.7	56.2	79.3	102.3	
12	5.9	13.2	23.5	35.0	45.7	67.2	88.6	
18	13.0	19.4	22.5	33.9	44.5	65.8	87.1	
24	12.0	18.0	20.1	24.0	33.9	53.6	73.3	
32	11.0	16.4	18.3	21.7	21.9	39.9	57.9	
48	8.7	13.2	14.9	17.8	17.9	18.0	33.2	
64	7.0	10.9	12.4	14.8	15.0	15.0	27.4	
ALL								31.8
	OURS vs. *MULT*							
6	−2.9	−2.9	−2.9	−0.7	15.8	19.5	36.5	
12	−9.5	−9.5	−9.5	0.6	11.3	14.0	35.3	
18	−10.2	−10.2	−10.2	7.8	18.3	21.0	38.2	
24	−14.9	−7.5	−0.2	8.4	18.2	34.2	48.9	
32	−9.6	−6.9	−0.2	7.7	16.6	22.2	35.6	
48	−21.4	−19.5	−13.9	3.1	3.1	4.0	14.4	
64	−22.3	−17.4	−15.0	−1.1	−1.1	−1.1	24.9	
ALL								4.9

the processing data path. The only time penalty is the one required for reprogramming the LUTs when it is necessary to change the constant. The configuration time is shown in Table VI for several sizes of the constant and the two programming schemes (i.e., direct or reverse, see Section 3). The *direct* scheme requires about 20% more time for reconfiguration than the *reverse* one. It can be seen that the reconfiguration process is extremely fast, in the order of 100 ns.

As a last step, we compare the proposed design (*OURS*) to standard multipliers implemented using Xilinx Coregen v10. Multipliers based on both slices (*SLICES*) and MULT blocks (*MULT*) are considered. Note that one of the multiplier's inputs is registered in order to use the operator as a programmable constant multiplier.

Tables VII and VIII contain the relative increase of the maximum operation frequency (decrease for negative values) obtained using the architecture proposed in this article instead of each of the standard multipliers, for signed and unsigned operands, respectively. Except when comparing with *MULT* for signed operands, where there are some negative values, the use of our proposal always increases the maximum operation frequency. In general, this improvement is bigger when the size of the operand

Table VIII. Relative Increment of Maximum Operation Frequency (%) Achieved Using *OURS* for Unsigned Operand

M	6	12	18	24	32	48	64	ALL
N	*OURS* vs. *SLICE*							
6	5.2	20.7	31.9	44.8	56.5	79.6	102.7	
12	13.8	20.7	31.9	44.8	56.5	79.6	102.7	
18	22.4	29.3	31.9	44.8	56.5	79.6	102.7	
24	20.7	27.0	29.4	33.4	44.2	65.4	86.6	
32	18.7	24.4	26.5	30.2	30.4	49.6	68.8	
48	15.7	20.5	22.3	25.3	25.5	25.6	41.7	
64	13.5	17.7	19.2	21.8	22.0	22.0	35.2	
ALL								39.6
	OURS vs. *MULT*							
6	−2.7	−2.7	−2.7	0.9	17.9	19.7	38.2	
12	−2.7	−2.7	0.9	9.6	21.0	22.5	46.9	
18	−2.7	−2.7	9.6	18.2	29.6	31.1	49.7	
24	−7.0	0.9	10.1	18.1	28.6	44.5	60.3	
32	−3.4	0.8	9.2	16.3	25.9	30.7	45.0	
48	−16.3	−13.4	9.7	9.7	9.7	10.7	21.7	
64	−16.7	−11.5	4.9	4.9	4.9	4.9	32.6	
ALL								12.9

(M) increases. The speed improvements with respect to *SLICES* are very significant, ranging from 5.9% to 102.3% for signed, and from 5.2% to 102.7% for unsigned, with a mean of 31.8% and 39.6%, respectively. Compared to *MULT*, the embedded blocks are faster in approximately half of the signed cases and in less than a third of the unsigned ones. It must be noted that for values of N and M bigger than or equal to 18 bits, only 20% of *MULT* perform better than *OURS* for the signed case; as for the unsigned case, all of *OURS* are faster. The speedup ranges from −22.3% to 48.9% with a mean of 4.9% for the signed case, and from −16.7% to 60.3% with a mean of 12.9% for the unsigned case. This improvement, although smaller than that of *SLICES*, is very valuable, since it should be remembered that *MULT* uses specialized multipliers directly implemented on silicon, whereas *OURS* uses general logic resources. Also, it must be noted that *MULT* is beating our approach in terms of speed for small-size multipliers mainly.

In the second experiment, we have synthesized each type of multiplier unit with 49 combinations of $N \times M$, different pipeline depths, and 26 frequencies ranging from 50 MHz to 300 MHz. An XC3S1600E-5 device is utilized as the technological framework. The comparison is made by selecting the number of pipeline stages which produces multipliers with minimum areas that comply with a certain operation frequency constraint. For the sake of fairness, the area comparison is made by using the metric that accounts for the maximum number of multiplier units that fit the device [Bouganis et al. 2009; Caffarena et al. 2009], that is, the total number of resources available in the device divided by the number of resources used in the design, where the resources could be slices or dedicated multipliers.

Figures 6 and 7 display the maximum number of multiplier units versus frequency results for several combinations of parameters. When one architecture is not able to achieve the required frequency, a zero value is represented. For the signed case, we can see in Figure 6 that *OURS* clearly outperforms *SLICES* in area-time for the majority of cases, whereas it requires much fewer resources than *MULT* implementations. On the other hand, the unsigned case from Figure 7 shows a similar trend with slightly better results.

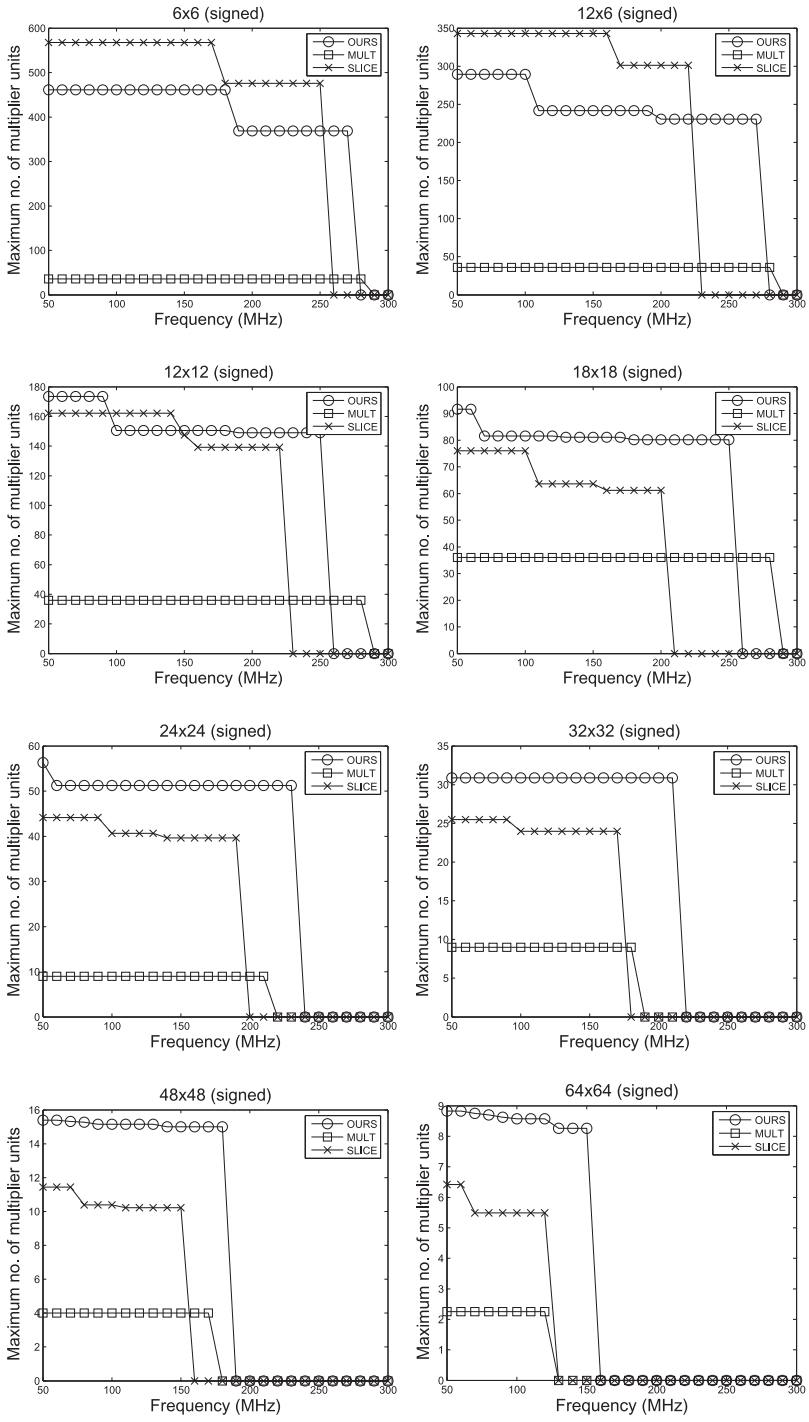

Fig. 6. Maximum number of multiplier units on an XC3S1600E-5 device versus clock frequency (MHz) for different $N \times M$ signed multipliers.

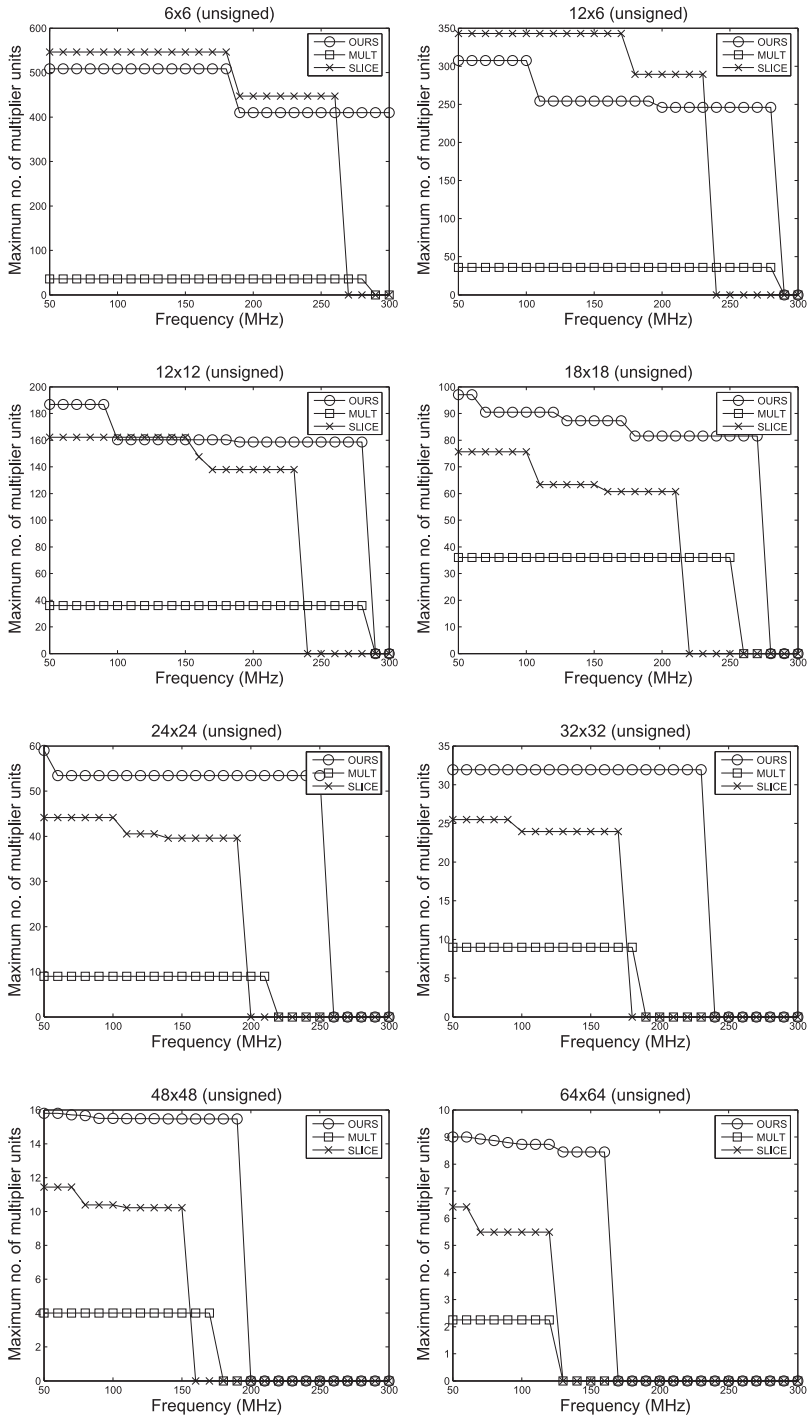

Fig. 7. Maximum number of multiplier units on an XC3S1600E-5 device versus clock frequency (MHz) for different $N \times M$ unsigned multipliers.

Table IX. Average Area Reduction (%) Achieved for Signed

M	6	12	18	24	33	48	64	
N	*OURS* vs. *SLICE*							
6	−24.12	−16.29	−12.67	−9.36	−7.84	−4.26	−1.34	
12	−29.34	2.53	3.39	5.60	6.92	7.71	10.13	
18	−29.30	0.23	18.08	19.77	20.32	21.91	23.35	
24	−31.11	−0.94	18.38	19.65	20.10	20.89	23.05	
32	−41.12	−6.85	14.27	16.23	20.41	21.91	23.94	
48	−30.17	−1.80	18.53	19.98	24.12	30.21	32.92	
64	−32.35	−2.65	17.21	19.82	24.10	31.39	34.11	
ALL								3.92
	OURS vs. *MULT*							
6	91.42	85.89	80.33	87.44	83.79	84.21	84.53	
12	85.49	76.75	68.22	79.83	74.36	74.95	75.22	
18	79.59	67.96	55.95	72.23	64.62	65.81	66.08	
24	86.65	79.37	72.11	82.52	77.77	78.25	78.52	
32	81.63	72.30	63.15	76.93	70.83	71.57	71.79	
48	83.00	74.23	65.61	78.46	72.69	73.62	73.91	
64	82.63	73.85	65.24	78.30	72.41	73.50	74.07	
ALL								76.13

Table X. Average Area Reduction (%) Achieved for Unsigned

M	6	12	18	24	33	48	64	ALL
N	*OURS* vs. *SLICE*							
6	−8.01	−3.44	0.59	3.17	5.57	9.35	12.05	
12	−21.68	8.46	10.13	12.50	13.78	15.10	17.46	
18	−22.78	5.41	24.42	24.02	24.97	26.42	27.97	
24	−26.00	2.71	21.48	22.70	23.61	24.76	26.71	
32	−33.49	−2.58	15.80	19.04	23.15	24.99	27.17	
48	−28.79	0.49	16.07	21.20	25.35	31.95	34.59	
64	−30.84	−1.16	18.63	21.15	25.48	32.61	35.35	
ALL								9.5
	OURS vs. *MULT*							
6	92.21	87.27	82.49	88.75	85.65	86.19	86.47	
12	86.27	78.10	70.34	81.24	76.14	76.91	77.18	
18	80.36	69.42	58.45	73.69	66.68	67.85	68.14	
24	87.10	80.14	72.93	83.25	78.76	79.33	79.56	
32	82.39	73.36	62.97	77.69	71.81	72.68	72.97	
48	83.23	74.78	64.40	79.00	73.37	74.28	74.59	
64	82.79	74.26	65.77	78.64	72.90	73.99	74.56	
ALL								77.33

The experimentation embraces a total of 1,274 $f_{ck} \times N \times M$ scenarios tested for each type of operand (signed and unsigned). For signed multipliers, there are 84 cases where *OURS* is the only feasible implementation for a given frequency and 56 cases where *MULT* is the only candidate. This never happens to *SLICE* implementations, since they are the slowest for all experiments. *OURS* has the smallest area for 647 cases, *MULT* for 56 cases, and *SLICE* for 245 cases corresponding to small-size multipliers. Considering unsigned multipliers, there are 148 cases where *OURS* is the only option and 18 cases where *MULT* is the only candidate. *OURS* has the smallest area for 784 cases, *MULT* for 18 cases, and *SLICE* for 184 cases corresponding to small-size multipliers.

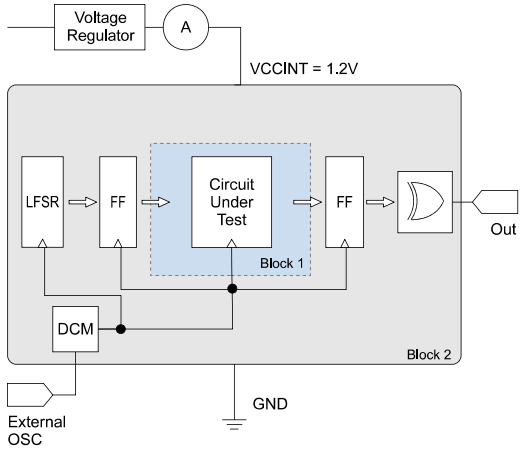

Fig. 8. Power measurement experimental setup.

The overall area results for signed multipliers are condensed in Table IX. It display the average reduction in comparison to *SLICE* and *MULT* when our approach is used. For each combination of $N \times M$, the mean of the area reduction achieved for all frequencies tested is shown. To calculate this mean, only results corresponding to frequencies which are reached for both compared designs are used. The numbers yield that our approach outperforms the other implementations substantially for bitwidths equal to or bigger than 12 bits. For the signed case, the mean area reduction when compared to *SLICE* ranges from −41.12% to 34.11%, with a mean for all sizes of 3.92%. However, if only multipliers with bitwidths equal to or bigger than 12 are considered, it can be observed that *OURS* outperforms *SLICE* for most cases, and the overall area reduction rises to 14.9. For bitwidths from 18 bits, the mean area improvement is 21.6%. The area improvement with respect to *MULT* is more significant, and it ranges from 55.95% to 91.42%, with an overall mean of 76.32%. That means that the number of multipier units that could be implemented in one device goes from 2× to 11× when using *OURS* instead of *MULT*. These improvements are slightly greater for the unsigned case, as shown in Table X.

4.2. Power Consumption Results

In the previous section, the area-time properties of the proposed multiplier were presented, so in order to complete the characterization of the multiplier, in this section, power consumption is analyzed.

The proposed design (*OURS*) is compared to standard multipliers implemented using Xilinx Coregen. The comparison only includes slice-based multipliers (*SLICES*), since MULT blocks have a reduced power consumption. The power consumption is obtained though experimental measurements using a Digilent Spartan 3 Board with a Xilinx Spartan 3 XC3S200-FT256 FPGA device. This board is not specifically designed to perform power measurements; therefore, some modifications were made in order to measure the internal core power consumption; thus, IO power was not measured. The on-board 1.2-volts regulator was removed and substituted by a circuit that includes an external regulator and a serial shunt. A calibration procedure of the shunt resistor and the measurement probes was performed. The voltage across the shunt resistor was measured with a Tektronix TDS3052C oscilloscope and with a Fluke 45 multimeter, having a relative error in the measures of less than 1.5%.

Table XI. Power Consumption Results: *OURS* vs. *SLICES*

Size	OURS[1] (mW)	SLICES[2] (mW)	Power reduction (%)
18×18	46.6	45.4	2.6
24×24	73.4	72.3	1.5
32×32	119.1	114.0	4.3

[1] Fully pipelined
[2] Level of pipelining recommended by vendor

Figure 8 shows the block diagram of the experimental setup that is based on connecting the *circuit under test* (block 1) to an *on-chip test vector generator* (block 2). The circuit under test is a signed integer multiplier with registered inputs and outputs. The test vector generator is composed of a digital clock manager (DCM), a linear feedback shift register (LFSR) that generates pseudorandom data, and a parity function that limits the number of FPGA outputs to one. This experimental setup minimizes the use of inputs and outputs and their influence on the design [Oliver and Boemo 2011; Wilton et al. 2004].

The board has an external oscillator of 50 MHz, but the DCM output was fixed to 100 MHz for all the experiments. The multiplier input is changed every clock cycle, and the constant is changed every 1,024 clock cycles. The proposed multipliers (*OURS*) are fully pipelined to meet the clock frequency, while the Coregen multipliers (*SLICES*) have five pipeline stages, as recommended by the Coregen tool.

The power consumption of the DCM, the LFSR, and the parity generator was measured separately and then subtracted from the total measured power. Table XI shows the power consumption of the circuits under test.

The results yield that the power consumption of the proposed multiplier is in the range of the multipliers provided by Xilinx.

5. CONCLUSIONS

The use of standard optimized constant multipliers is not suitable for applications where constants change. This situation forces the use of generic multipliers. This article presents as an alternative idea a constant multiplier able to reconfigure itself in runtime to change the constant value with no restriction. Thus, this design could substitute generic multipliers in such cases. The configuration time of the proposed architecture is shorter than the partial reconfiguration times required by FPGA devices. It does not use any storage for programming data, and its size is easily parameterizable. Compared to generic multipliers based on slices, it clearly outperforms those implementations in terms of area and speed and poses similar power consumption features. Compared to embedded blocks, our approach is faster in most cases (especially for unsigned operands), and it allows for implementing many more multiplier units in the same device.

We regard the application of the proposed reconfigurable constant multiplier to floating-point arithmetic as an interesting future research line. A floating-point multiplier is composed of a fixed-point multiplier with added hardware blocks to deal with the exponents of the multiplicands as well as with the normalization and rounding of the final results. Thus, the multiplier proposed in this article could replace the fixed-point multiplier in the floating-point architecture, being only necessary to provide the value of the constant in a floating-point format where there is information of both mantissa and exponent. The mantissa value would be used as the constant value to program the fixed-point architecture proposed in this article, and the exponent would be stored to be added to the exponent of the variable input.

Also, another research line is the assessment of this new architecture in modern FPGA devices based on LUTs with more than four inputs.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers of previous versions of this work.

REFERENCES

Bosí, B., Bois, G., and Savaria, Y. 1999. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 7,* 3, 299–308.

Bouganis, C.-S., Park, S.-B., Constantinides, G. A., and Cheung, P. Y. K. 2009. Synthesis and optimization of 2D filter designs for heterogeneous FPGAs. *ACM Trans. Reconfigurable Technol. Syst. 1*, 24:1–24:28.

Caffarena, G., López, J., Leyva, G., Carreras, and Nieto-Taladriz, O. 2009. Architectural synthesis of fixed-point DSP datapaths using FPGAs. *Int. J. Reconfigurable Comput.* 1–14.

Chapman, K. 1996. Constant coefficient multipliers for the "xc4000e". Tech. rep. XAPP 054, Xilinx Corporation, San Jose, CA.

Chen, J. and Chang, C.-H. 2009. High-level synthesis algorithm for the design of reconfigurable constant multiplier. *IEEE Trans. Comput.-Aid. Des. Integ. Circuits Syst. 28,* 12, 1844–1856.

Compton, K., Li, Z., Cooley, J., Knol, S., and Hauck, S. 2002. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. VLSI Syst. 10,* 3, 209–220.

Dandalis, A. and Prasanna, V. 2005. Configuration compression for FPGA-based embedded systems. *IEEE Trans. VLSI Syst. 13,* 12, 1394–1398.

Demirsoy, S., Kale, I., and Dempster, A. 2007. Reconfigurable multiplier blocks: Structures, algorithm and applications. *Circuits, Syst. Signal Process. 26,* 6, 793–827.

Gustafsson, O. 2007. Lower bounds for constant multiplication problems. *IEEE Trans. Circuits Syst. II 54,* 11, 974–978.

Gustafsson, O., Dempster, A. G., Johansson, K., Macleod, M. D., and Wanhammar, L. 2006. Simplified design of constant coefficient multipliers. *Circuits, Syst. Signal Process. 25,* 2, 225–251.

Huang, X., Liang, C., and Ma, J. 2008. System architecture and implementation of MIMO sphere decoders on FPGA. *IEEE Trans. VLSI Syst. 16,* 2, 188–197.

Kalra, R. and Lysecky, R. 2010. Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems. *IEEE Trans. VLSI Syst. 18,* 4, 671–674.

Mahesh, R. and Vinod, A. 2010. New reconfigurable architectures for implementing FIR filters with low complexity. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst. 29,* 2, 275–288.

Meher, P. 2010. Novel input coding technique for high-precision LUT-based multiplication for DSP applications. In *Proceedings of the 18th IEEE/IFIPVLSI System on Chip Conference (VLSI-SoC)*. 201–206.

Nguyen, H. and Chattejee, A. 2000. Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis. *IEEE Trans. VLSI Syst. 8,* 4, 419–424.

Oliver, J. and Boemo, E. 2011. Power estimations vs. power measurements in Cyclone III devices. In *Proceedings of the Southern Conference on Programmable Logic*. 87–90.

Park, J., Jeong, W., Mahmoodi-Meimand, H., Wang, Y., Choo, H., and Roy, K. 2004. Computation sharing programmable FIR filter for low-power and high-performance applications. *IEEE J. Solid-State Circuits 39,* 2, 348–357.

Shoufan, A., Wink, T., Molter, H., Huss, S., and Kohnert, E. 2010. A novel cryptoprocessor architecture for the McEliece public-key cryptosystem. *IEEE Trans. Comput. 59,* 11, 1533–1546.

Turner, R. and Woods, R. 2004. Highly efficient, limited range multipliers for LUT-based FPGA architectures. *IEEE Trans. VLSI Syst. 12,* 10, 1113–1118.

Wilton, S., Ang, S., and Luk, W. 2004. The impact of pipelining on energy per operation in field-programmable gate arrays. In *Field-Programmable Logic and Application*. Lecture Notes in Computer Science, Vol. 3203. Springer-Verlag, Berlin, 719–728.

Wirthlin, M. 2004. Constant coefficient multiplication using look-up tables. *J. VLSI Signal Process. Syst. 36,* 1, 7–15.

Xu, F., Chang, C.-H., and Jong, C.-C. 2008. A new approach to versatile subexpressions sharing in multiple constant multiplications. *IEEE Trans. Circuits Syst. 55,* 2, 559–571.

Received November 2012; revised April 2013; accepted May 2013