An Efficient Softcore Multiplier Architecture for Xilinx FPGAs

Martin Kumm, Shahid Abbas and Peter Zipf
University of Kassel, Germany
Digital Technology Group
Email: kumm@uni-kassel.de, shahid.abbas@student.uni-kassel.de, zipf@uni-kassel.de

Abstract

This work presents an efficient implementation of a softcore multiplier, i. e., a multiplier architecture which can be efficiently mapped to the slice resources of modern Xilinx FPGAs. Instead of dividing the multiplication into the generation of partial products and the summation using a compressor tree, as done in modern multipliers, an array-like architecture is proposed. Each row of the array generates a partial product which is directly added to results of previous rows using the fast carry chain. A radix-4 Booth encoding/decoding is used to reduce the I/O count of the partial product generation which makes it possible to map both, the Booth encoder and decoder, into a single 6-input look up table (LUT). Like a conventional Booth multiplier, this nearly halves the number of rows compared to a ripple carry array multiplier. In addition, the compressor tree is completely avoided and an efficient and regular structure retains that uses up to 50 % less slice resources compared to previous approaches and offers a multiply accumulate (MAC) operation without extra resources.

1. Introduction

Efficient multipliers are a key element for many applications in the domain of digital signal processing, image processing, scientific computing and many more. This demand on multipliers and their large complexity when mapped to the logic of field programmable gate arrays (FPGAs) forced the vendors to embed hard multiplier blocks and DSP blocks (which include pre and post addition) into the fabric of modern FPGAs. However, there are two main limitations when using embedded multipliers or DSP blocks: First, only a limited number is available and second, their word size is restricted. Furthermore, they have fixed locations and are often asymmetric in word size (like the 18×24 multiplier in Xilinx FPGAs). The limited word size can be extended by using several smaller multipliers and additional adders. A method called *tiling* was proposed to construct large multiplications using several DSP blocks and softcore multipliers in an efficient way [1], [2]. Here, the softcore multipliers are needed to "fill gaps" where a DSP block is too large. Besides the large multipliers, many applications require only small word size multipliers, like, e. g., image and video processing applications, where typical word sizes are 8 to 10 bit. Here, large parts of the DSP block are wasted. As a consequence, there is a demand for softcore multipliers whenever DSP blocks are not sufficient or their word size does not match to the requirements.

While a rich body of literature exists for parallel multipliers in application specific integrated circuits (ASICs), there is only few dealing with softcore multipliers on FPGAs. Kumar et al. presented a softcore multiplier structure which is based on a Booth multiplier [3] where the partial products are obtained from Booth recoding of one multiplicand and the compressor tree is replaced by a ripple-carry adder tree to benefit from the fast carry chain of the FPGA [4]. While this is suitable for the 4-input LUTs of old FPGAs, much more is possible on modern FPGAs.

A softcore multiplier architecture that exploits the low-level properties of modern FPGAs was presented by Parandeh-Afshar et al. [5]. They showed that two partial products of a Baugh-Wooley multiplier [6] can be generated and already compressed in a single stage of logic by using the two outputs of a look up table (LUT) and the fast carry chain. The number of remaining partial products is nearly halved like for the Booth multiplier which avoids the Booth decoding/encoding logic. However, a compressor tree is still required to obtain the final product. This architecture is reviewed in Section 2.1.

Another totally different concept is to compose a multiplier out of several smaller multipliers and additions. The idea of using small multipliers which fit

1063-6889/15 $31.00 © 2015 IEEE
DOI 10.1109/ARITH.2015.17

Table 1: 4×4 bit multiplication scheme

					a_3b_0	a_2b_0	a_1b_0	a_0b_0
$+$				a_3b_1	a_2b_1	a_1b_1	a_0b_1	
$+$			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
$+$		a_3b_3	a_2b_3	a_1b_3	a_0b_3			
$=$	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

into few FPGA LUTs was mentioned but not further investigated by de Dinechin et al. [7]. They proposed to map a 3×3 multiplier in six 6-input LUTs which work in parallel, leading to a delay of a single LUT. These partial results have to be bit-shifted and added which can be done using a compressor tree.

The compressor tree is a crucial part in multipliers. While it was believed for a long time that ripple carry adders are the best choice to realize a multi-input addition on FPGAs, it was demonstrated in the last years that the usage of compressor trees with generalized parallel counters (GPCs) as replacement of full-adders can improve this. While in early work, the delay could be reduced by compressor trees [8]–[11], it was shown recently that also the resources can be saved by exploiting the low level structure of FPGAs [12], [13]. However, the compressor tree typically occupies a significant part of the total resources of the multiplier.

In this work, an optimized softcore multiplier architecture is presented which completely avoids the compressor tree. This is done by merging the Booth recoding scheme of a Booth multiplier with the (ripple-carry) summation of the partial products in a single stage of LUTs, fast carry chains and (optionally) flip flops (FFs). The result is a regular array-like structure which is also capable to perform a multiply accumulate (MAC) operation. Besides the combinatorial implementation, a pipeline extension of the architecture is discussed which can be obtained by a few additional slice resources for FFs. Results are obtained from several synthesis experiments which are compared with Xilinx Coregen [14], the Parandeh-Afshar multiplier, which was ported to the Virtex 6 FPGA [5], and several variants of the idea presented in [7].

2. Review of FPGA Multiplier Architectures

In the following, previous multiplier architectures are reviewed. The presentation is done for unsigned numbers for the sake of simplicity. The extension to signed numbers is straight forward and is indicated where needed for the proposed multiplier.

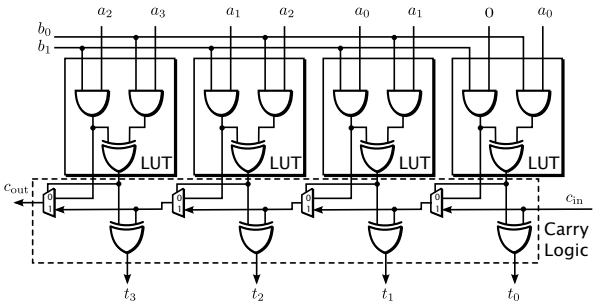


Figure 1: Slice configuration for the Parandeh-Afshar multiplier

2.1. Baugh-Wooley Multiplier

An $N \times M$ bit multiplication of two unsigned binary numbers $a = (a_{N-1} \dots a_1 a_0)$ and $b = (b_{M-1} \dots b_1 b_0)$, with $a_i, b_i \in \{0, 1\}$ can be written as

$$p = a \cdot b = a \cdot \left(\sum_{m=0}^{M-1} 2^m b_m \right) = \sum_{m=0}^{M-1} 2^m \underbrace{a \cdot b_m}_{\text{partial product}} \quad . \tag{1}$$

Hence, the partial products $a \cdot b_m$ can be obtained by a bitwise AND-operation which are then bit-shifted and added to get the final product. Take, e. g., a 4×4 bit multiplication, the product p is obtained by summing four partial products as shown in Table 1.

The probably most straight forward way to do this is realized with the ripple-carry array (RCA) multiplier. Its modification for two's complement inputs is better known as the Baugh-Wooley multiplier [6]. Here, a basic cell which includes an AND gate and a full adder (FA) is organized in a regular array where each row corresponds to a ripple carry adder (RCA) which adds the partial product of the current row (m in (1)) with the preceding results, yielding the product in the last stage.

The idea of Parandeh-Afshar et al. was to map two AND gates into the two-output LUT of the adaptive logic modules (ALMs) of modern Altera Stratix FPGAs (beginning from Stratix-II) and to use the full adders to add the partial results [5]. This structure can be directly transferred to the Xilinx FPGA slice which consists of four LUTs, fast carry chain logic and up to eight FFs. A slice mapping is illustrated in Figure 1 for the first two rows and the four least significant bit (LSB) columns of the example in Table 1. Here, the two AND gates are used to generate the partial products while the exclusive-or (XOR) gate in the LUT, together with the XOR gate and the multiplexer (MUX) in the carry logic form a full adder. With this

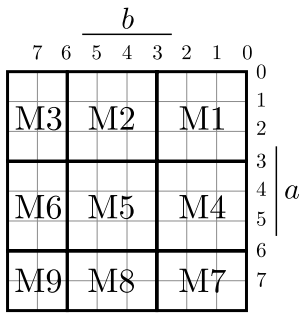


Figure 2: Tiling of a 8×8 multiplier using smaller multipliers

configuration, two partial products can be added which halves the number of partial products (for even word sizes). The sum of all partial products is computed by using their compressor tree method [8]. Note that the LSB positions of each partial product can be computed with a single AND gate which slightly reduces the carry-chain delay.

2.2. LUT-Multiplier-based Architecture

The idea of realizing an FPGA specific softcore multiplier based on partial results of several 3×3 multipliers which were mapped to the FPGA LUTs was proposed recently [7]. A 3×3 multiplier has six output bits and can be directly mapped to six 6-input LUTs by tabulating the multiplication result for each output bit. To build a larger multiplier the partial results from these basic multipliers have to be shifted and added. A simple graphical way to compute these shifts was introduced with the tiling method [1], [2]. Here, an $N \times M$ multiplication is visualized with a rectangle of size $N \times M$. Then, this rectangle is "tiled" with smaller multipliers until the complete rectangle is filled. The input bits of the small multipliers as well as the output shifts can be directly obtained from the rectangle. Take for example an 8×8 multiplier wich is represented by the rectangle in Figure 2. It can be covered by four 3×3 multipliers (M1, M2, M4 and M5), four 2×3 multipliers (M3, M6, M7 and M8) and one 2×2 multipler (M9). The 2×3 and 2×2 multipliers use less LUTs due to the reduced output word size of 5 and 4, respectively. Now, the inputs of a multiplier correspond to the coordinate ranges in a and b directions. The inputs of multiplier M6 for example are $a_{3\ldots5}$ and $b_{6\ldots7}$. The necessary shift of the multiplier result is equal to the sum of a and b coordinates of the upper right corner of the small multiplier. Multiplier M6 has to be shifted by $3+6=9$ bit as its upper right corner coordinate is $a=3$ and $b=6$. The complete product of the 8×8 multiplier example is

$$p = M1 + 2^3 M2 + 2^6 M3 + \ldots \quad (2)$$
$$\ldots + 2^3 M4 + 2^6 M5 + 2^9 M6 + \ldots \quad (3)$$
$$\ldots + 2^6 M7 + 2^9 M8 + 2^{12} M9 \quad (4)$$

where Mi corresponds to the result of multiplier i.

Besides the 3×3 basic multiplier, there are other possible configurations which might be interesting. On modern Xilinx FPGAs, the 6-input LUT can be configured as two 5-input LUTs with shared inputs. This allows to build a 3×2 multiplier with only five 5-input LUTs which are mapped to three 6-input LUTs (one 5-input LUT remains unused). Another way is to realize a 1×4 multiplier with four 5-input LUTs which are mapped to two 6-input LUTs. While these different basic multipliers use the LUT resources in a more efficient way, the number of partial products is increased significantly which increases the compressor tree. The three basic multipliers are evaluated experimentally in Section 4.

3. Proposed Architecture

Before the proposed multiplier is presented, some common concepts used in Booth multipliers are discussed.

3.1. Booth Multiplier

A well known multiplier and probably the most used multiplier in the ASIC domain is the Booth multiplier [3]. The main idea is to recode one of the inputs as follows:

$$\begin{aligned} b =& b_{M-1}2^{M-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 \\ =& b_{M-1}2^{M-1} + \ldots + b_2 2^2 + 2b_1 2^1 + \underbrace{-b_1 2^1 + b_0}_{\mathrm{BE}_0 = -2b_1 + b_0} \\ =& b_{M-1}2^{M-1} + \ldots \\ & \ldots + 2b_3 2^3 \underbrace{-b_3 2^3 + b_2 2^2 + 2b_1 2^1}_{\mathrm{BE}_2 = (-2b_3 + b_2 + b_1)2^2} + \mathrm{BE}_0 \\ =& \sum_{\substack{m=0 \\ m \text{ even}}}^{M} \mathrm{BE}_m 2^m \text{ with } \mathrm{BE}_m = -2b_{m+1} + b_m + b_{m-1} \end{aligned} \quad (5)$$

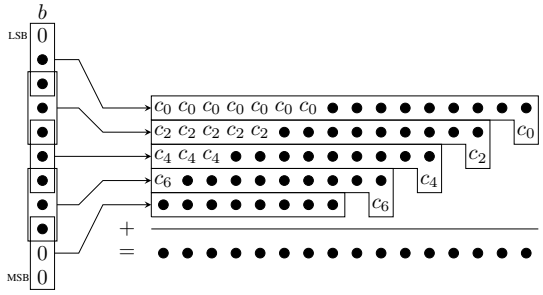
The multiplication becomes

$$a \cdot b = \sum_{\substack{m=0 \\ m \text{ even}}}^{M} a \cdot \mathrm{BE}_m 2^m \ . \quad (6)$$

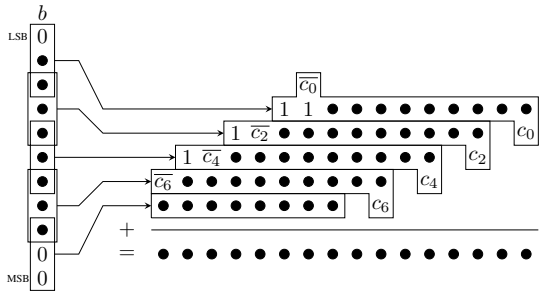
The values of the *Booth encoder* terms BE_m are in the range of $-2 \ldots 2$ as listed in Table 2. Hence, the

Table 2: Truth table of a Booth encoder

b_{m+1}	b_m	b_{m-1}	BE_m	z_m	c_m	s_m
0	0	0	0	1	0	0
0	0	1	1	0	0	0
0	1	0	1	0	0	0
0	1	1	2	0	0	1
1	0	0	-2	0	1	1
1	0	1	-1	0	1	0
1	1	0	-1	0	1	0
1	1	1	0	1	0	0



(a) Naive sign extension



(b) Optimized sign extension

Figure 3: Sign extension in Booth multipliers [15]

partial products $a \cdot \mathrm{BE}_m$ can be obtained by shifting a if $|\mathrm{BE}_m| = 2$ (encoded by $s_m = 1$), complementing the result if $\mathrm{BE}_m < 0$ (encoded by $c_m = 1$) or setting the partial product to zero in case that $\mathrm{BE}_m = 0$ (encoded by $z_m = 1$). This nearly halves the number of partial products and, thus, reduces the size of the compressor tree which typically follows.

Due to this coding, the partial products may become negative which requires a sign extension. The sign bit of each partial product is identical to the *complement* bit c_m according to Table 2. If the complement bit is set, the partial product has to be inverted and a '1' must be added to the LSB position. The naive approach is illustrated in the dot diagram in Figure 3(a). Here, each row of dots represents a partial product which is already inverted when the corresponding c_m is set. The conditional increment is done by adding the c_m bit to the LSB position. The sign extension is done by padding c_m bits up to the MSB position of the product. A well known technique to reduce the number of bits for sign extension is to replace a bit vector of identical bits by a vector of ones plus the inverted bit at LSB position:

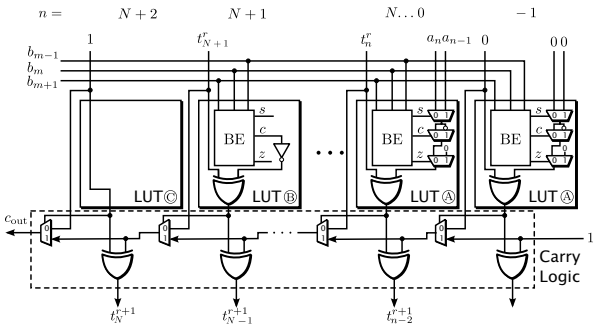


Figure 4: Slice configuration of the proposed multiplier

$$\begin{array}{cc|cccccccccc} & & 1 & 1 & 1 & 1 & \dots & 1 & 1 & 1 & 1 \\ + & & & & & & \dots & & & & \overline{x} \\ \hline = & \overline{x} & x & x & x & x & \dots & x & x & x & x \end{array}$$

The MSB bit of the result $\overline{x}$ has to be ignored. Applying this transformation to the dot diagram in Figure 3(a) results in several overlapping rows of constant '1' bits which can be added in advance, resulting in the dot diagram shown in Figure 3(b). This representation is typically used as input of a compressor tree that reduces the partial products to the final product. In the proposed architecture, each partial product row is generated in a chain of FPGA slices and previous intermediate results are directly added to the current partial product. This is detailed in the following.

3.2. Proposed FPGA Mapping

Each partial product row is mapped to a slice configuration as shown in Figure 4, the complete multiplier array is given in Figure 5. We distinguish between the three different LUT configurations Ⓐ, Ⓑ and Ⓒ. Except the two LUTs at the MSB positions $n = N+2$ and $n = N+1$ (leftmost LUTs in Figure 4), where n is the column index of the corresponding row, all LUTs are of configuration Ⓐ. The rightmost LUT in Figure 4 has the only task to add c_m to the LSB position. This is due to a limitation that the carry-in of a slice can only be set to constant '0' or constant '1' when the zero-input of the carry chain MUX bypasses the LUT. Using the shown input configuration, the rightmost LUT simply computes $\overline{z}c_m$ which is passed over the

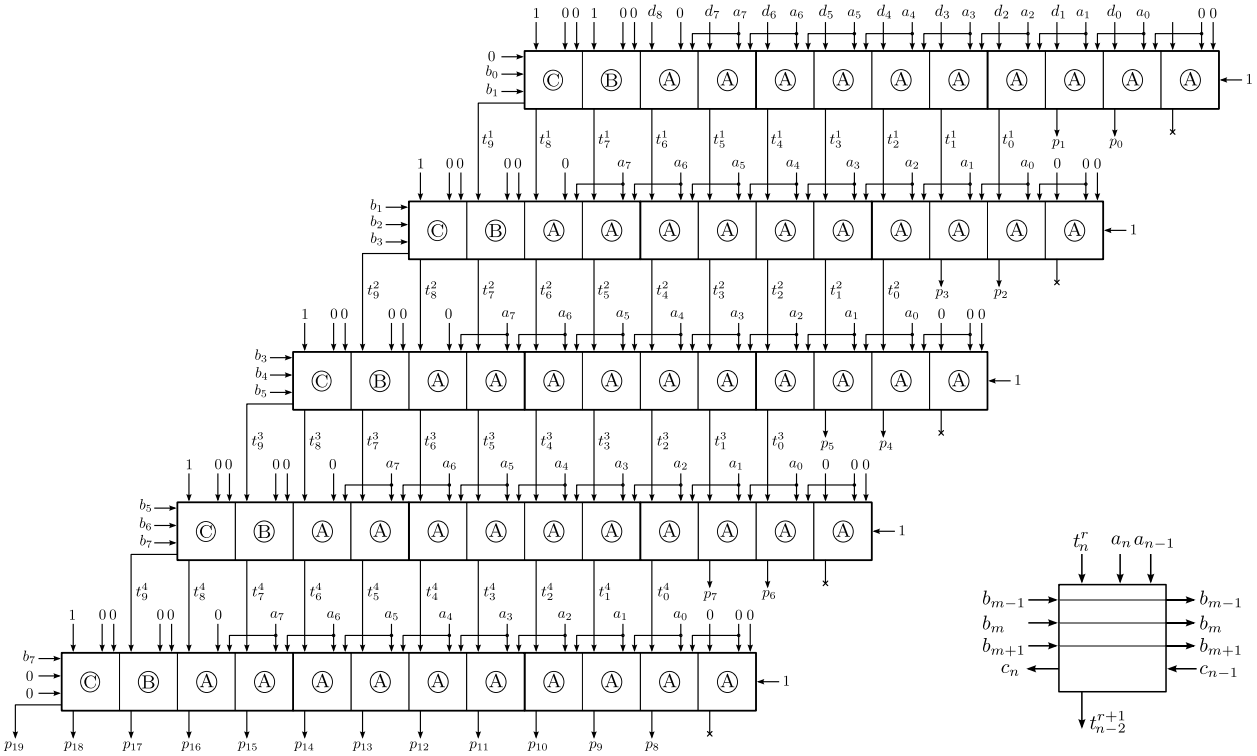


Figure 5: Overall architecture of the proposed multiplier with 8×8 bit unsigned inputs

carry chain MUX to the carry-in of the next higher bit position.

The following $N + 1$ LUTs (represented by the 2nd right LUT in Figure 4) realize the partial product generation and addition (the dots in Figure 3). As a Booth encoder has three inputs and three outputs, it can be replicated in each LUT Ⓐ without cost, saving an extra (high fanout) Booth encoder for each partial product. The bit shift, complement and zero setting (i. e., the Booth decoder) are performed in the same LUT which is represented by three multiplexers in Figure 4. The first MUX realizes the shift by selecting between a_n and a_{n-1} (a_{-1} is defined to be zero). The next MUX performs the conditional inversion in case of a complement and the last MUX sets the partial product to zero in case of $z = 1$. The input t_n^r denotes a result of a previous row (or the accumulate input d), where $r = m/2$ is the index of the row. This input is added to the partial product which leads to the result of the next row shifted by two bits t_{n-2}^{r+1}. With this configuration, the complete LUT is utilized with all inputs to compute the Booth encoding, the decoding as well as an 1-bit addition which is the key for an efficient FPGA implementation.

Each row is completed by two LUTs with configuration Ⓑ and Ⓒ for the sign extension. LUT Ⓑ is used to add $\overline{c_m}$ while LUT Ⓒ adds the constant '1' according to Figure 3(b). In the first row, the input t_{N+1}^0 is used to add the additional '1'.

Several such rows are cascaded to build the full multiplier as shown for an 8×8 multiplier in Figure 5. Each square represents a LUT and one bit of carry chain processing. The slices are indicated with thick lines. It can be seen that the structure is very regular and has a rectangular shape (which is drawn as a parallelogram in Figure 5 only for clarity). Note that some LUTs at MSB side can be removed as their result can be ignored. Modifications for signed input numbers are straight forward and identical to the modifications that are necessary in common Booth multipliers [15].

The number of required slices can be exactly predicted without synthesis. It is equal to the number of slices per row times the number of rows. Four columns can be mapped to the four LUTs of a slice leading to

$$\#\text{slices}(M,N) = \underbrace{\lceil N/4+1 \rceil}_{\text{slices per row}} \cdot \underbrace{\lfloor M/2+1 \rfloor}_{\text{no of rows}} . \quad (7)$$

Of course, if N (the word size of a) is not divisible by four, some LUTs of the MSB slices may be available for other logic. Note that for $N \neq M$, it might be better to swap the operands. For example, with $N = 9$ and $M = 8$, the multiplier requires 20 slices while for

$N = 8$ and $M = 9$ it requires only 15 slices.

In addition to the multiplication, the architecture offers a "free addition" with an $N + 1$ bit number (inputs $d_n = t_n^0$). This can be used to realize a MAC operation with no additional resources. This feature is useful in many DSP related applications and may be of particular interest for enhancing the word size of a hard multiplier block [1], [2], where bit shifted results have to be added.

The multiplier architecture can be easily pipelined by inserting FFs. A deep pipeline can be realized by placing FFs after each row. This costs less than expected on the first view as all t_n^r terms can be stored in the otherwise unused FFs of the slices. On Virtex 6 and the 7-series FPGAs, there are eight FFs per slice. The remaining four FFs can be used to delay the a_n inputs without cost in this case. The only remaining FFs are needed to synchronize the b_m inputs as well as the product bits produced in the first row. Most of them can profit from the shift register mode of the LUT (SRL16/SRLC32) [16]. Of course, one can omit pipeline stages for trading of the speed against the latency, which can be as high as the number of rows (which is $\lfloor M/2 + 1 \rfloor$).

3.3. Mapping to Altera FPGAs

Unfortunately, the proposed structure can not be directly transferred to Altera FPGAs, as only 4-input LUTs are available at each full adder input in modern architectures like the Stratix V or Arria 10. Extending the ALM by four additional multiplexers, it would be possible to route the output of the 5-input LUT and one ALM input to the inputs of each full adder. With this slight modification, the proposed multiplier would be possible.

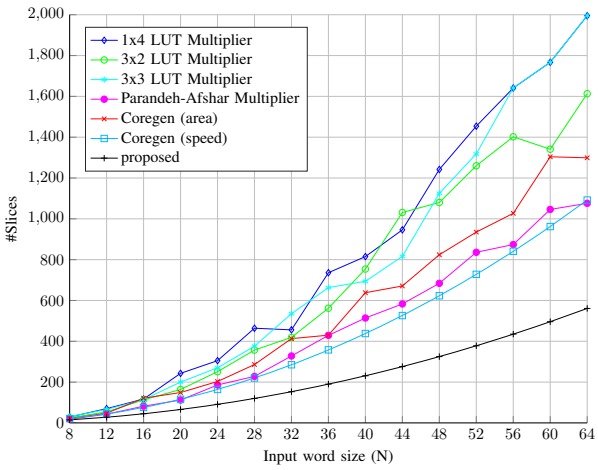
4. Results

We performed several synthesis experiments to compare the proposed multiplier with the state-of-the-art approaches. For that, the methods of Section 2.1 [5] and Section 2.2 [7] for the discussed variants with 3×3, 3×2 and 1×4 basic LUT multipliers as well as the Xilinx LogiCORE IP multiplier block as produced by Coregen [14] were considered. The input word sizes were varied in the range of $N = M = 8 \ldots 64$ in steps of 4 bit. All designs in the experiment use an unsigned number representation.

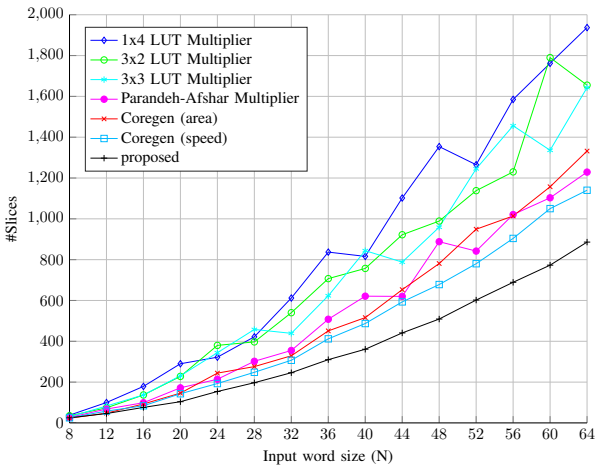
Due to its regular structure, the proposed multiplier was implemented in VHDL for arbitrary word sizes M and N and optional pipelining using VHDL `generate` statements and Xilinx primitives. The Baugh-Wooley multiplier [5] as well as the LUT based multipliers [7] require a compressor tree to add the partial products. For that, a VHDL code generator was implemented which uses the FloPoCo framework [17] and its built-in compressor tree synthesis [7]. The reason to use this heuristic compressor tree optimization in contrast to a recently proposed optimal integer linear programming (ILP) based optimization [13] is due to the large problem sizes. For the larger multipliers, more than 1000 partial product bits have to be compressed while the optimal method exceeded a time limit of one hour of computation time for problems larger than 100 bits [13]. Thanks to automatic pipelining in the FloPoCo framework, these designs could be easily pipelined. For that, the default target frequency of 400 MHz was used which produced pipeline depths between two (for $N = 8$) and seven stages (for $N = 64$). The Coregen multiplier offers the two different design goals 'speed' and 'area' [14]. As some speed optimized multipliers were smaller than the area optimized ones, we produced cores for both optimization goals. Pipelined multipliers were also generated using the 'Optimum pipeline stages' proposal of the Coregen tool, which was between two (for $N = 8$) and six stages (for $N = 64$).

All designs were synthesized using Xilinx ISE 13.4 with design goal 'speed' for a Virtex 6 FPGA (XC6VLX75T-FF784). The results are plotted in Figure 6. The combinatorial designs are shown on the left (Figure 6(a), (c) and (e)) and the pipelined designs are on the right (Figure 6(b), (d) and (f)). For each case, the number of total slices (Figure 6(a) and (b)), the percentage slice reduction of using the proposed multiplier compared to previous (Figure 6(a) and (b)) and the maximum frequencies obtained by trace (Figure 6(e) and (f)) are shown. To obtain the maximum frequency, inputs and outputs were registered (not counted in the slice plots).

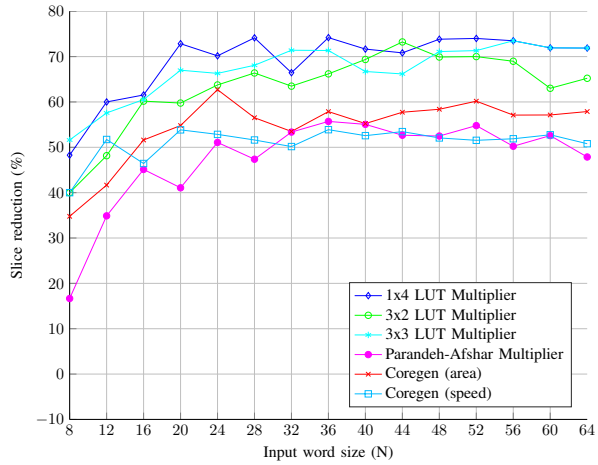
It can be observed that the proposed multiplier clearly outperforms all previous multipliers for any input word size $\geq$12 bit in terms of slice usage. In the combinatorial case about 50 % of the slices can be saved for word sizes $\geq$24 bit. In the pipelined case, the slice reduction is more than 25 % for word sizes $\geq$20 bit compared to designs with similar speed (Coregen 'area' and Parandeh-Afshar). A surprising result is that the second best method in terms of slice resources but one of the worst in terms of speed is Coregen with design goal 'speed', the design goal 'area' produced faster but larger designs. In terms of speed, the combinatorial implementation of the proposed multiplier is the slowest one. This is as expected as slow ripple-carry adders are used com-



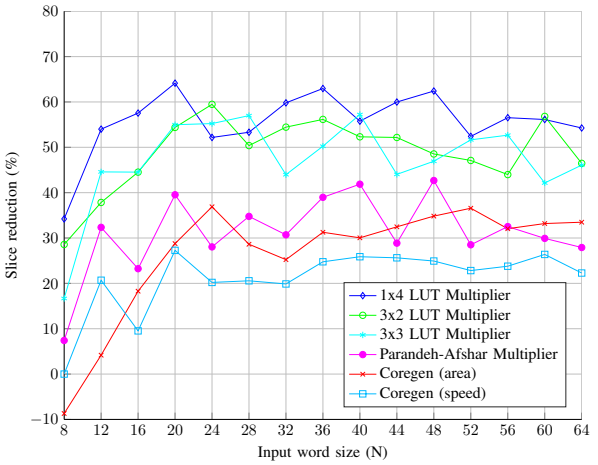
(a) Slices for combinatorial multipliers



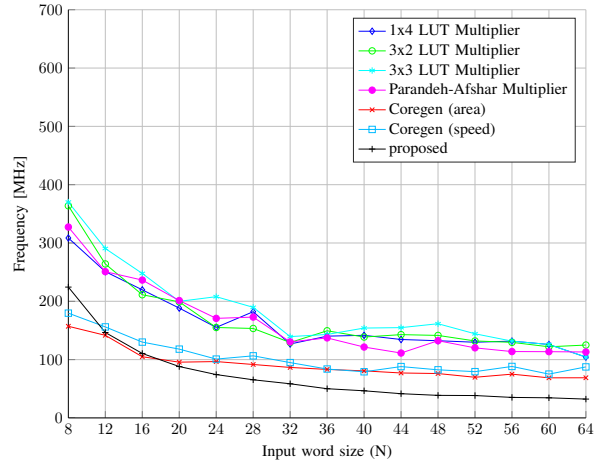
(b) Slices for pipelined multipliers



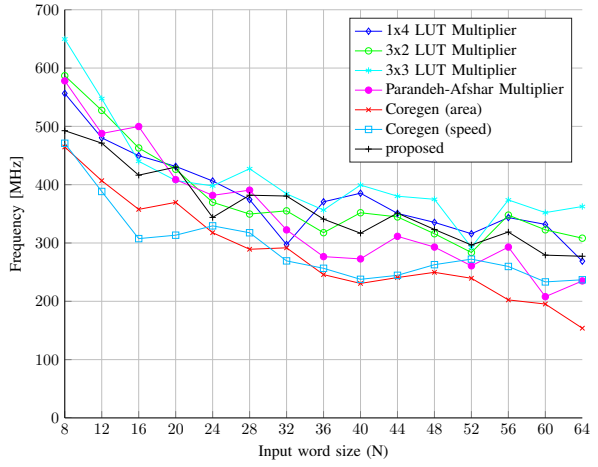
(c) Slice reduction for combinatorial multipliers



(d) Slice reduction for pipelined multipliers



(e) Speed for combinatorial multipliers



(f) Speed for pipelined multipliers

Figure 6: Slice resources and speed vs. input word size

pared to the compressor trees in the other methods. However, the pipelined implementation clearly shows a high performance which is always close to the fastest design.

For all respected LUT-based multipliers, the slice resources were similar with a slight preference for the 3×2 basic LUT multiplier. On average, using 3×2 basic LUT multipliers required 6.6 % and 10.3 % less LUTs compared to 3×3 and 1×4 basic LUT multipliers, respectively.

Of course, the situation would have looked different if a better compressor tree optimization was used. Average slice reductions of about 20 % were reported in [13] compared to the FloPoCo heuristic [7] used in this work. Assuming that half of the slice resources are required due to the compressor tree and that this reduction is applicable to the considered designs, the net effect would be a 10 % reduction, which is still worse compared to the proposed multiplier.

5. Conclusion

A multiplier architecture was proposed that is based on well known components from classical multiplier architectures which are organized in a way to yield a very efficient FPGA mapping. The design was compared to state-of-the-art softcore multipliers using a number of synthesis experiments for most practical word sizes. Compared to the best of the considered methods, slice reductions up to 50 % in the combinatorial and up to 30 % in the pipelined implementation were obtained, respectively. Besides these large slice reductions, the pipelined implementation provides state-of-the-art performance, which makes the multiplier attractive for many applications.

References

[1] F. de Dinechin and B. Pasca, "Large Multipliers with Fewer DSP Blocks," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2009, pp. 250–255.

[2] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for Floating-Point Double Precision and Beyond on FPGAs," *SIGARCH Computer Architecture News*, vol. 38, no. 4, pp. 73–79, Sep. 2010.

[3] A. D. Booth, "A Signed Binary Multiplication Technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, pp. 236–240, 1951.

[4] S. Kumar, K. Forward, and M. Palaniswami, "A Fast-Multiplier Generator for FPGAs," in *Interntational Conference on VLSI Design*. IEEE Comput. Soc. Press, 1995, pp. 53–56.

[5] H. Parandeh-Afshar and P. Ienne, "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 225–231.

[6] C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, 1973.

[7] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic Core Generation Using Bit Heaps," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2013, pp. 1–8.

[8] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting Fast Carry-Chains of FPGAs for Designing Compressor Trees," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2009, pp. 242–249.

[9] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor Tree Synthesis on Commercial High-Performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–19, 2011.

[10] T. Matsunaga, S. Kimura, and Y. Matsunaga, "Multi-Operand Adder Synthesis Targeting FPGAs," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E94-A, no. 12, pp. 2579–2586, Dec. 2011.

[11] ——, "An Exact Approach for GPC-Based Compressor Tree Synthesis," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96-A, no. 12, pp. 2553–2560, Dec. 2013.

[12] M. Kumm and P. Zipf, "*Efficient High Speed Compression Trees on Xilinx FPGAs*," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014.

[13] ——, "Pipelined Compressor Tree Optimization Using Integer Linear Programming," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–8.

[14] Xilinx, Inc., "Xilinx DS255 LogiCORE IP Multiplier v11.2, Data Sheet," pp. 1–13, Mar. 2010.

[15] G. W. Bewick, "Fast Multiplication: Algorithms and Implementation, Appendix A," Ph.D. dissertation, Stanford University, 1994.

[16] Xilinx, Inc., "Xilinx Virtex-6 Libraries Guide for HDL Designs," pp. 1–378, Jun. 2011.

[17] (2014) FloPoCo Project Website. [Online]. Available: http://flopoco.gforge.inria.fr