2011 21st International Conference on Field Programmable Logic and Applications

MEASURING AND REDUCING THE PERFORMANCE GAP BETWEEN EMBEDDED AND SOFT MULTIPLIERS ON FPGAS

Hadi Parandeh-Afshar and Paolo Ienne

Communication and Computer Science Department
Federal Institute of Technology in Lausanne (EPFL)
email: {hadi.parandehafshar, paolo.ienne}@epfl.ch

ABSTRACT

To bridge the gap between FPGAs and ASICs for arithmetic dominated circuits, one key step is to improve multipliers on FPGAs. This is a key feature that FPGA vendors have tried to improve in recent years by embedding ASIC like multipliers in the DSP blocks. However, due to the limited number of DSP blocks in an FPGA, their fixed location and bit-width limitation, efficient soft logic implementation of multipliers is fundamental. This is the reason that FPGA vendors have enhanced the logic blocks architecture to improve certain arithmetic circuits such as adder tree, which is the basic part of a parallel multiplier. This paper has two main contributions: (1) The performance gap between embedded and soft multipliers is measured and the design space is explored and (2) the current performance gap is reduced by employing a number of target specific mapping and arithmetic transformation techniques. For this purpose, a multiplier generator tool is developed and two conventional multiplication techniques are implemented in this tool. We compare our multipliers with the ones that are generated by *Altera* core generator tool considering a wide range of bit-widths. Therefore, this paper can be used as a reference for the digital circuit designers to choose the right way of implementing multipliers on FPGAs based on their design constraints.

1. INTRODUCTION

Multiplication is a key operation in many Digital Signal Processing applications [1] and having fast multipliers on FPGAs is fundamental. This is the main reason that most of the current state-of-the-art FPGAs have embedded multipliers, which are called *DSP* blocks. The DSP blocks have ASIC-like structure and in addition to multipliers they implement some other few signal processing related circuits such as rounding and shifting. Such DSP blocks potentially can improve the delay, area and power of certain applications that perfectly utilize these resources. However, using the embedded multipliers does not always lead to a better performance [2]. Embedded multipliers implement certain fixed multiplication bit-widths [3][4] and for those applications that require different bit-widths, a considerable delay overhead is imposed when either a bigger multiplier is used to implement the smaller ones or smaller base multipliers are used to construct bigger multipliers. Moreover, since the DSP blocks have fixed locations in the FPGAs, there might be some constraints in the placement and routing of the applications that have a mixture of multiplier and non-multiplier logic. Finally, the number of DSP blocks is limited in an FPGA and for those applications that require more multipliers, we need to use soft logic for their implementation. This can be a limiting factor for performance, since the soft multiplier can be placed on the critical path of the application [2].

Alternatively, we can use the logic blocks of FPGAs for implementing multipliers. Such multipliers are called *soft multipliers* in this paper. The structure of logic blocks in recent FPGAs indicates the importance of efficient implementation of soft multipliers. The logic blocks in current FPGAs have fast carry chains and dedicated logic for fast addition, which are primarily intended to perform fast binary and ternary addition as well as multi-operand addition using fast ripple carry adders. It is worth mentioning that the main part of a multiplier is the *Partial Product Reduction Tree (PPRT)*, which is a multi-input addition.

In this paper, we explore the design space of multipliers on FPGAs and measure the performance gap between embedded and soft multipliers for different bit-widths. This study will help designers to choose the right implementation method based on the design constraints. For soft multipliers, we use a number of optimization techniques, which improve the performance of such multipliers. One contribution is to exploit the dedicated adders of logic blocks along with the small LUTs at the adder inputs to increase the logic block utilization and reduce the depth of the PPRT. Moreover, a number of arithmetic transformations are used for the signed multipliers in order to eliminate the sign extension parts of the *Partial Products (PPs)*. This will reduce the size of the PPRT. In addition to these techniques, we use [5] approach for the synthesis of the PPRT. This technique

978-0-7695-4529-5/11 $26.00 © 2011 IEEE
DOI 10.1109/FPL.2011.48

has already been proven to enhance multi-input additions on FPGAs and in this paper we will investigate its effectiveness in the framework of multiplier design, which includes both addition and non-addition logic. Moreover, the pipelining feature is added to this approach to be able to design pipelined multipliers. To explore the design space, we take different well-known VLSI multiplication methods and apply our techniques to them.

We developed a multiplier generator tool, which takes the multiplication method, operand widths, multiplier latency or number of pipeline levels, and being signed or unsigned as the input and generates the multiplier netlist in low level Verilog. This tool is similar to the wizards in current FPGA CAD tools that generate predefined and optimized arithmetic netlists for the target FPGAs. These tools provide the option of implementing the multipliers using both embedded multipliers and soft logic. We chose *Altera Stratix-III* FPGA for the experiments and the experimental results illustrated that our tool generates multipliers that are on average 27% faster than the ones that *Altera Quartus-II Megawizard* generates. Moreover, the experiments revealed that we are able to reduce the performance gap between embedded multipliers and soft multipliers from 54% to 22%. Moreover, one can use the proposed techniques and the reported results in this paper to choose the right implementation method based on the design requirements.

The structure of this paper is as follows. Related work is discussed in section 2. Section 3 explains some arithmetic and FPGA basics. In section 4, we present our proposed optimizations and implementation techniques. Experimental results are discussed in section 5. Finally section 6 concludes the paper.

2. RELATED WORK

A majority of recent work [6][7][8] discuss different methods to decompose large size multipliers using FPGA embedded multiplier blocks. With the introduction of asymmetric embedded multiplier blocks in recent Xilinx FPGAs [3], efficient decomposition of larger multipliers is challenging. In contrast to our work, none of these methods discuss soft logic implementation of the multipliers on FPGAs.

There are some work [9][10] in the area of improving the implementation of modular multipliers on FPGAs. However, modular multipliers are only used for public-key cryptography applications. In [9], Beuchat et al. suggests a method to take advantage of the dedicated carry logic available in modern FPGAs. In this paper, we design multipliers for 2's complement and sign magnitude numbering system.

In [11], Kumar et al. propose a multiplier generator for FPGAs which is based on Booth algorithm. Several features of FPGA architecture are used in this work to generate fast multipliers. However, the logic cell in current modern FPGAs is significantly different from the old FPGAs at that time. For example, the logic cells in current FPGAs have dedicated adder logic and carry chains that can be exploited to build efficient multipliers.

3. PRELIMINARIES

In this section, we briefly explain the arithmetic concepts that we use in this paper, including different multiplier approaches and compressor trees. For signed multiplication, we chose radix-4 Booth [12] and Baugh-Wooley [12] algorithms. Radix-4 has less PPs, while Baugh-Wooley has smaller PPs. Also in this section, we introduce the structure of the FPGA logic block that we use for the design of the multipliers.

3.1. Baugh-Wooley Multiplier

Baugh-Wooley [12] multiplication is based on the standard shift and add multiplication method and is used for the multiplication of signed numbers. The benefit of this multiplier is that the PPs are not sign extended. A Baugh-Wooley PPG for an $N \times N$-bit signed multiplier produces $N^2 + 1$ partial product bits, some of which are computed using a NAND gate rather than an AND gate, and the most significant output bit of the multiplier is inverted. One of the partial product bits is set to the constant value '1'. In this paper, by using a set of target specific mapping techniques, we merge each two PP generations into one logic block, which reduces the number of PPs by half.

3.2. Radix-4 Booth Multiplier

Radix-4 Booth [12] multiplication is a well-known VLSI multiplier design for 2's complement signed numbers, in which the number of PPs is half of the other array based multiplication schemes. In this method, the numbers that are multiplied are recoded and this way the number of PPs is reduced. The basic idea is to take every second bit, and multiply by ± 1, ± 2, or 0, instead of shifting and adding for every bit of the multiplier term and multiplying by 1 or 0. To Booth recode the multiplier term, the bits in blocks of three are considered, such that each block overlaps the previous block by one bit. The overlap is necessary so that we know what happened in the last block, as the MSB of the block acts like a sign bit. Figure 1 shows the PPG unit of this multiplier. It is obvious that the PPG unit in radix-4 is much more complex than the other two multipliers. However, this unit can fit into a 5-input logic block and therefore a single level of logic blocks suffices to implement all PPGs. Moreover, inherently the number of PPs is $\frac{\lceil N \rceil}{2}$ for an N$\times$N-bit signed multiplication and therefore the PPRT is faster and smaller. But, in contrast to Baugh-Wooley multiplier, each

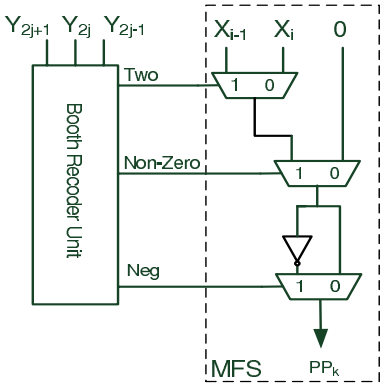

Fig. 1. Radix-4 Booth PPG unit. Each PPG output bit is a 5-input function, which has two shared bits with the previous/next PPG function.

PP should be sign extended and this will add some overhead to the PPRT. In this paper, we apply some arithmetic transformations, inspired from Baugh-Wooley multiplier, to the PPs, which will eliminate the sign extension parts of the PPs, and a few constant bits are added instead.

3.3. Compressor Tree

To compute the sum of two integers, a *Carry-Propagate Adder (CPA)* [12] such as ripple-carry and carry-select adders is used. To add more numbers, a compressor tree [12] is used. The compressor tree has a carry save structure, meaning that there is no carry dependency between the building blocks within the same level. In [5], *Generalized Parallel Counters (GPC)* are introduced as the building blocks that can be most efficiently mapped to the current FPGA logic blocks using the carry chains. At the end of the GPC tree a 3-input adder (ternary adder) is used to add the results. Current FPGAs from both Altera and Xilinx have support for efficient implementation of 3-input ripple carry adders. Furthermore, we added the pipelining feature to this compressor tree to be able to implement pipelined multipliers.

3.4. FPGA Logic Block Structure

We choose *Altera* FPGAs for the synthesis of multipliers. We believe that the same techniques can be used to implement fast multipliers on current *Xilinx* FPGAs, due to the fact that we use the specific features of the FPGAs for the implementation that are common between these two FPGA families. The logic block structure has not been changed from *Stratix-II* generation. For our experiments, we selected the *Stratix-III*, which is fabricated in a 65nm technology. The logic block in these series of FPGAs is called *Adaptive Logic Module (ALM)*. Ten ALMs are placed in an array, which is called *Logic Array Block (LAB)*. Each ALM has

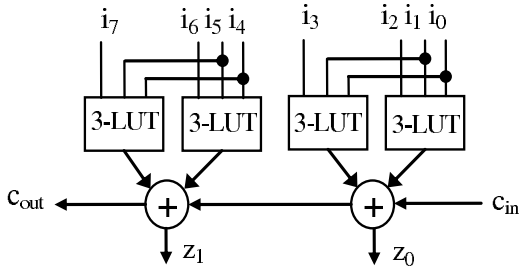

Fig. 2. ALM structure in arithmetic mode. The 3-LUTs with shared inputs can be configured as 2-LUTs with non-shared inputs.

two outputs and eight inputs. The LUT structure of ALM is fracturable and two adders and fast carry chain are the other major parts of ALM. Each ALM can operate in four different modes. In the normal mode, only LUTs are used and two different functions with some shared inputs can be implemented by an ALM. One possible configurations can be two 5-input functions with three shared inputs. This is an ideal case for mapping the PPG units of two adjacent PP bits, which have several shared inputs and each PPG is a function of at most 5 variables. This way the ALM is utilized much more efficiently than the other multipliers.

The other mode of ALM that we use in this paper is the *Arithmetic Mode*, which is shown in Figure 2. In this mode, the adders and the carry chains of the ALMs are exploited along with some LUTs as shown in this figure. Each adder is derived by two 3-LUTs, which have two shared inputs. The carry chain can be initiated either at the beginning of a LAB or in the middle, before the sixth ALM. We will use this mode of ALM to reduce the number of PPs in Baugh-Wooley multipliers.

4. PROPOSED DESIGN TECHNIQUES

To optimize the implementation of the mentioned multipliers on FPGAs, we have used some target specific mapping techniques and arithmetic oriented transformations. The mapping technique improves the logic density and halves the number of PPs in Baugh-Wooley multipliers; while the arithmetic transformations are applied to the PPs of radix-4 multiplier to remove the sign extension parts.

4.1. Target Oriented Optimization

As discussed earlier, the number of the PPs in Baugh-Wooley is twice that of the PPs in radix-4 multiplier. However, the PPG unit of these two multipliers is a set of simple two-input AND gates. The PPG unit in all mentioned multipliers can fit into one level of logic blocks, whereas in the Baugh-Wooley multipliers the logic blocks are underutilized. Considering the available resources and bandwidth of ALM in

arithmetic mode shown in Figure 2, we managed to merge the PPGs corresponding to two consecutive PPs into one logic block (ALM) level. As shown in Figure 3 (a), one can take the first two PPs, ignoring the first bit of the first PP, map the PPGs of the two PPs with the same bit position to the pair LUTs that derive the adder in the ALM. This is possible, since the two 3-LUTs in Figure 2 can be configured as 2-LUTs with non-shared inputs. Then, the following adder can be exploited to add the results of these two PPGs. This way, we are able to use a series of chained ALMs in arithmetic mode (1) to map the PPGs of two PPs and (2) to reduce two PPs into one. Figure 3 (b) shows the PPG mapping of the four corresponding bits of the first two PPs in Figure 3 (a) using this approach. The LUTs function as two input AND gates and the following adders sum-up the generated PPGs of the two PPs. The other interesting point in Figure 3 (a) is that the PPG units of two consecutive PPs have some common inputs, which reduces the number inputs to the ALMs. As an example in Figure 3 (a), the ALM, which is used for mapping the PPGs shown in the dotted box, only has five different inputs. Therefore, if all ten ALMs in a LAB are used for the PPs mapping, the total LAB input bandwidth would be less than or equal to 50. This is ideal, because based on our experiments the maximum input bandwidth of the LAB is 52 and therefore all ALMs in a LAB can be utilized for such design.

The drawback with this method is that the ALM chain that is used for reducing each two PPGs can be as long as the number of PP bits. This means that a ripple carry adder with the length of a PP is formed before going through the adder tree implementation. This will increase the delay of PPG unit and it is in contrast to the concept of compressor trees, which avoids carry propagation except for the final adder. To address this problem, we limited the ALM chain length to five, because the carry chain can be initialized either in the beginning of a LAB or in the middle before the sixth ALM. So the PPs are grouped in the 10-bit chunks, two per ALM. However, since the carry out of the last bit has to be considered for the addition in the PPRT, the inputs of the second half of the fifth ALM are grounded to be able to forward the carry out to the sum output of ALM. As a result, the number of bit positions of two PPGs that are placed in a group are constraint to 9 bits. Our experiments confirm that the delay of such block is comparable with the delay of the GPC blocks, which are the primitive components of compressor trees on FPGAs [5]. This guarantees to have uniform delays for the logic levels of both PPG and PPRT units, which allows having more precise pipelining.

4.2. Arithmetic Optimizations

In contrast to Baugh-Wooley, radix-4 multiplier PPs are sign extended. These repetitive sign bits create both delay and area overheads. To eliminate the sign extension parts of the

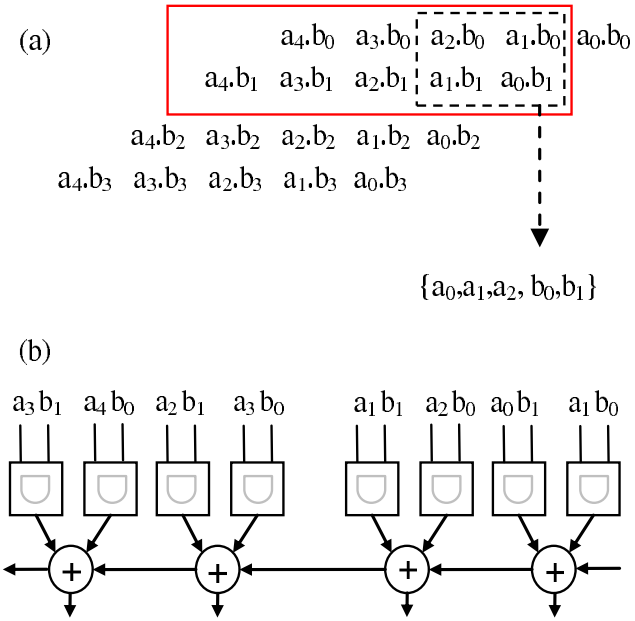

Fig. 3. (a) Each two adjacent PPGs can be merged and each two corresponding bits can be fit into an ALM with a required bandwidth of five. (b) Logic cell level representation of two merged PPGs.

PPs, we use the following transformations, which is similar to Baugh-Wooley technique. First, the sign extension part of each PP is added with $+1$ and then with -1, which logically does not change the PP value. However, when the sign part is added with $+1$, it is reduced to a single inverted sign bit ignoring the final carry bit as shown in Figure 4. Now, if this transformation is applied to the sign extension parts of all N PPs, then we will have N non-aligned constant numbers and N single inverted sign bits. Now, we can reduce the N constant numbers to only one number, by summing them up as shown in Figure 5. This step is performed automatically by our multiplier generator having the multiplier bit-widths as the input. Since the first bit of this constant number is aligned against the inverted sign bit of the first PP, we can append the constant number to the first PP as shown in Figure 6. The resulted value is then appended to the first PP from its sign bit position. Therefore, the number of bits rows in the PPRT will remain unchanged.

To summarize, with this technique we replace the whole sign parts of each PP with a single inverted sign bit, except for the first PP, where there are three sign bits in addition to a number of constant '0s' and '1s'.

4.3. Pipelining

A key feature in multiplier design on FPGAs is the possibility to insert pipeline registers within the layers of multi-

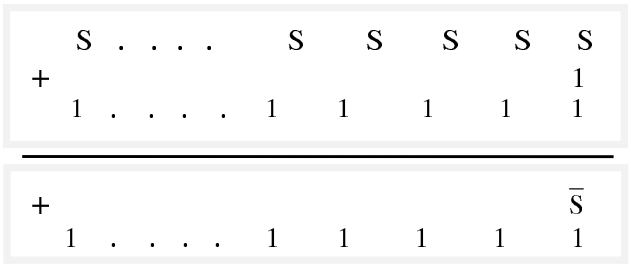

Fig. 4. Reducing the repetitive sign bits by adding with ± 1.

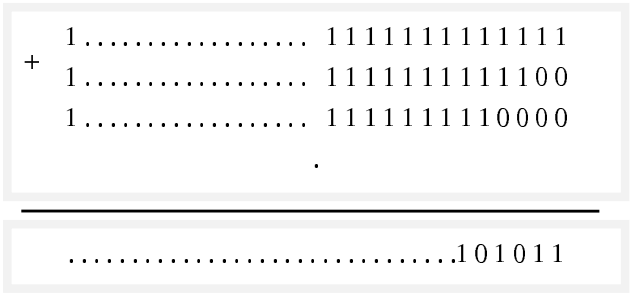

Fig. 5. Reducing the constant numbers to one number.

plier logic. For this purpose, we need to have an estimate of the number of logic levels to uniformly insert the registers. The compressor tree design method in [5], which we use for the implementation of the PPRT unit, does not address the pipelining issue of the compressor tree and only the combinatorial implementation is considered.

To estimate the number of logic levels, first, we define the term *logic level* as a set of independent logic units, where each logic unit includes one or more logic blocks. Except the local fast carry chains, no other interconnection wire is allowed within each logic unit. However, the length of the carry chain should not exceed a predefined threshold, which is called *Chain Length Threshold (CLT)*. The CLT is specified in a way that the delay of each logic layer remains below a delay threshold for the logic level. There are three different logic layers in the multiplier: the PPG layer, the carry-save layers in the PPRT and the final ternary adder.

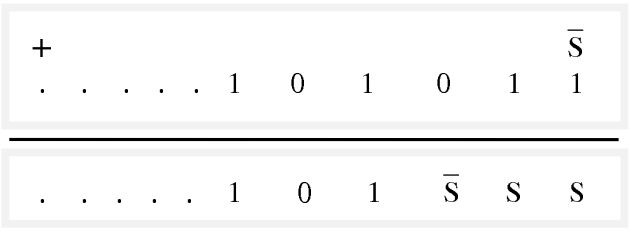

Fig. 6. Merging the constant number into the first partial product.

As mentioned in 3.3, the building blocks of compressor tree are GPCs, which form the carry-save layers of the PPRT. We set the logic unit delay threshold to the delay of GPCs. As the result, the number of logic levels are computed using equation (1).

$$LogicLevels = 1 + GPCLevels + \frac{FinalBitWidth}{CLT} \quad (1)$$

In this equation, one logic level is considered for the PPG. The contribution of the final adder to the logic levels is computed by dividing the adder bit width by the CLT. Once we estimated all the number of logic levels, we have to specify the place of pipeline registers based on the requested latency for the multiplier. The objective is to balance the number of logic blocks between registers to maximize the frequency.

5. EXPERIMENTAL RESULTS

We developed the multiplier generator tool in C++. Two multiplication algorithms including radix-4 Booth and Baugh-Wooley were implemented and the mentioned arithmetic transformations and target specific optimizations were applied. The pipelining feature was implemented as described in section 4.3. Our tool generates a Verilog netlist based on the multiplier specification provided by the user through the tool interface. For the experiments, we selected different bit-widths and compared the multipliers that are generated by our tool with the ones that are generated by *Altera Quartus Megawizard* tool.

Figure 7 compares the performance of radix-4 and Baugh-Wooley soft multipliers with the *Altera* soft multipliers. As shown, for all bit-widths, the radix-4 multipliers are faster than the *Altera* ones, while this is not the case for the Baugh-Wooley multipliers. There are two reasons that our radix-4 multipliers are faster than *Altera* multipliers: (1) the PPG delay in radix-4 multipliers is as short as the delay of a single LUT, while the PPG delay in *Altera* multipliers is as long as the delay of the final ripple carry adder and (2) the PPRT in radix-4 is efficiently implemented using carry save compressor trees, while for *Altera* multipliers the PPRT is implemented using 3-input carry propagate adders. Radix-4 multipliers are faster than the Baugh-Wooley multipliers for most of the bit-widths due to their faster PPG units.

The performance gain of our multipliers is obtained with the cost of consuming more FPGA logic cells. This area overhead is illustrated in Figure 8. The reason that the *Altera* multipliers are smaller than our multipliers is that the PPRT in *Altera* multipliers consumes less logic cells compared to the PPRT in our multipliers.

In Figure 9, we compare the performance of non-pipelined soft multipliers with the non-pipelined embedded multipliers of *Altera* FPGAs. Here, we assume that such multipliers are part of bigger circuits and therefore, the input/output

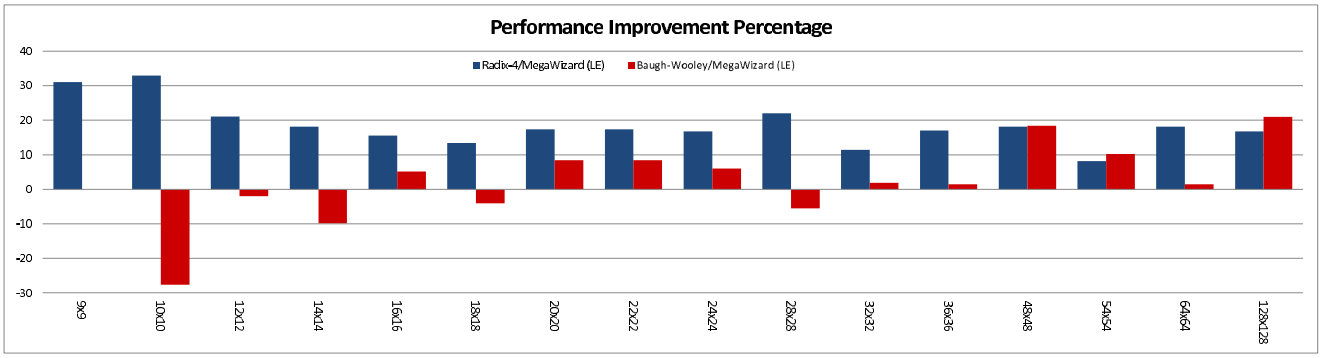

Fig. 7. The percentage of performance gain of non-pipelined radix-4 and Baugh-Wooley soft multipliers over non-pipelined *Altera* soft multipliers.

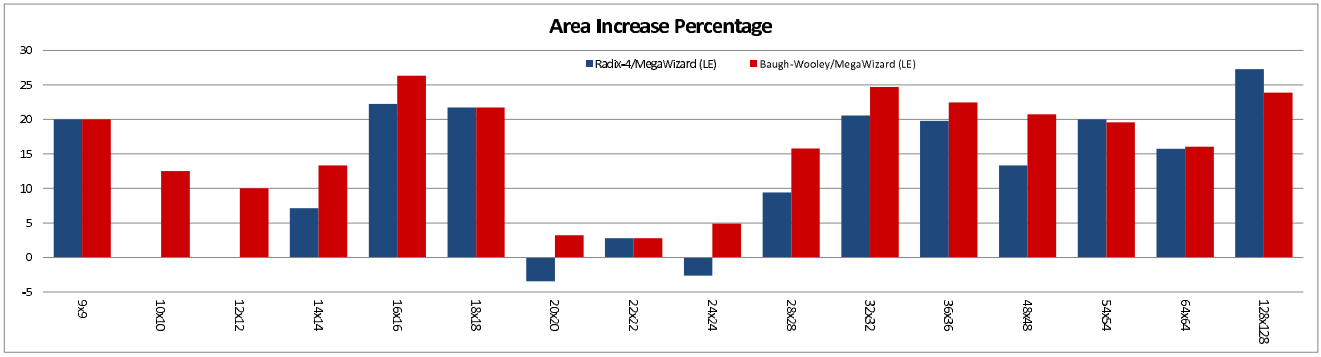

Fig. 8. The percentage of area (LAB) overhead of non-pipelined radix-4 and Baugh-Wooley soft multipliers over non-pipelined *Altera* soft multipliers.

connecting wires delays are includes in the overall delay of the multipliers. Since, embedded multipliers have fixed locations, the IO wires have longer delays and this constrains the delay advantage of embedded multipliers over soft multipliers. As shown in Figure 9, the radix-4 soft multipliers have better performance for the majority of bit-widths. Embedded multipliers with the bit-widths of smaller than 36 are exclusively implemented by the DSP blocks, while for the larger ones both DSP blocks and logic cells are involved.

Figure 10 illustrates the performance gap of pipelined soft multipliers with pipelined embedded multipliers. For this experiment, we registered the inputs and outputs of the multipliers and we inserted one stage of pipeline within the multipliers. Since, FPGA DSP blocks have optional registers on their inputs and outputs, the numbers that are reported in this figure do not include the input/output connecting wires delays. However, the IO delays may be added depending on the structure of the encompassing circuit. Ignoring the IO delays results in having faster embedded multipliers for the majority of the bit-widths, in contrast to what we see in Figure 9. Radix-4 soft multipliers are faster for small bit-widths, where larger embedded multipliers are used to map the smaller ones.

6. CONCLUSION

In this paper, we described a number of mapping techniques along with some arithmetic transformations, which improve the performance of soft multipliers significantly. To generate fast multipliers, specific features of FPGA logic blocks are considered to efficiently utilize the available resources. For the PPG unit of multipliers, in case of Baugh-Wooley, the pairs of PPGs are mapped into the same logic block and only one PP is generated rather than two using the dedicated adders and carry chains of logic blocks. For the PPRT unit, we apply a number of arithmetic transformations to the PPs of radix-4 multipliers to eliminate the sign extension parts. Moreover, we use [5] technique to efficiently map the PPRT of all multipliers on FPGAs using the carry-save compressor trees. Our experiments indicate that radix-4 multiplication with carry save PPRT unit is the right technique to implement soft multipliers on FPGAs and such multipliers always outperform *Altera* soft multipliers. Finally, the results revealed that the performance gap between embedded multipliers and soft multipliers is not siginificant.

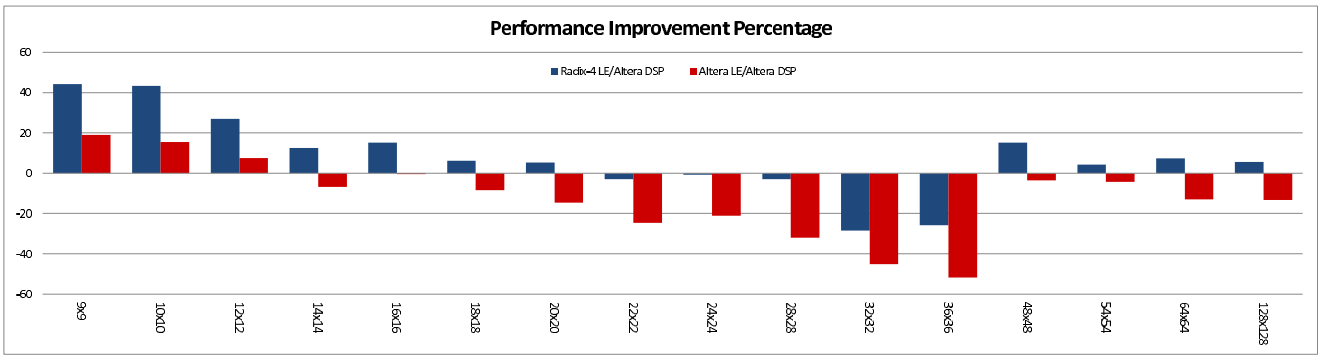

Fig. 9. The percentage of performance gain of non-pipelined radix-4 and *Altera* soft multipliers over non-pipelined *Altera* embedded multipliers.

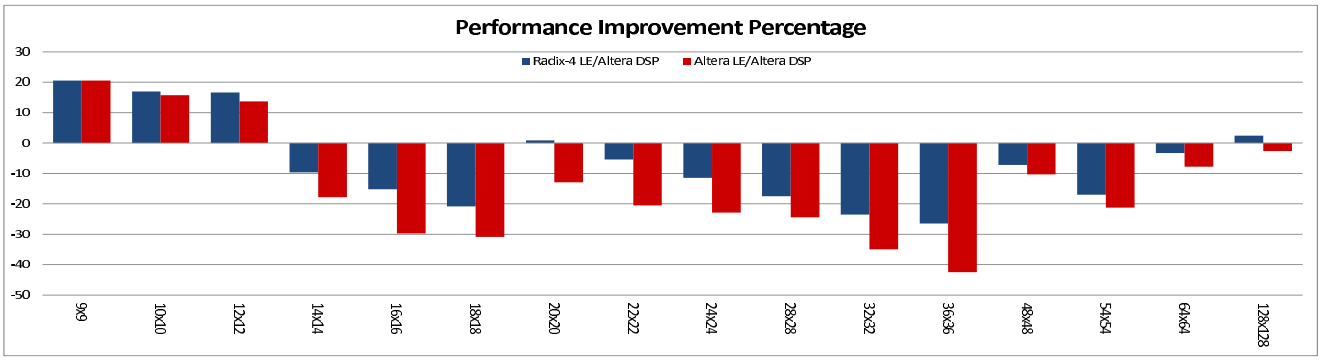

Fig. 10. The percentage of performance gain of 1-stage pipelined radix-4 and *Altera* soft multipliers over 1-stage pipelined *Altera* embedded multipliers.

7. REFERENCES

[1] R. Tessier and W. Burleson, *Reconfigurable Computing and Digital Signal Processing: Past, Present, and Future in Programmable Digital Signal Processors , Yu Wen Hu, ed., Marcel Dekker*. N.Y.: New York, 2002.

[2] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2006, pp. 21–30.

[3] *Virtex-5 User Guide*, Xilinx Inc., `http://www.xilinx.com/`.

[4] ALTERA, "Stratix II, III, and IV device handbooks," *Available online: http://www.altera.com/*.

[5] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting fast carry-chains of FPGAs for designing compressor trees," in *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications*, Prague, Aug. 2009, pp. 242–49.

[6] S. Srinath and K. Compton, "Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks," *in Proceedings of the 18th International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 51–58, Feb. 2010.

[7] F. Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," *in Proceedings of the 19th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 250–255, Aug 2009.

[8] S. Gao, D. Al-khalili, and N. Chabini, "Optimized realization of large-size two's complement multipliers on FPGAs," *in Proceddings of IEEE Northeast Workshop on Circuits and Systems*, pp. 494–497, Aug 2007.

[9] J. Beuchat and J. M. Muller, "Automatic generation of modular multipliers for FPGA applications," *in IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1600–1613, Dec 2008.

[10] D. N. Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler, "Efficient hardware architectures for modular multiplication on FPGAs," *in Proceedings of the 15th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 24–26, Aug 2005.

[11] S. Kumar, K. Forward, and M. Palaniswami, "A fast-multiplier generator for FPGAs," *in Proceedings of the 8th International Conference on VLSI Design*, pp. 53–56, Jan 1995.

[12] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Design*. New York: Oxford University Press, 2000.