


FPGA Implementation of Fast Binary Multiplication Based on Customized Basic Cells

Abd Al-Rahman Al-Nounou

(Department of Computer Engineering,


Jordan University of Science and Technology, Irbid, Jordan

 <https://orcid.org/0000-0003-2213-9540>, Amalnounou15@cit.just.edu.jo)

Osama Al-Khaleel


(Department of Computer Engineering,

Jordan University of Science and Technology, Irbid, Jordan

 <https://orcid.org/0000-0003-0585-2619>, oda@just.edu.jo)

Fadi Obeidat


(Cadence Design Systems, Austin, Texas, USA

 <https://orcid.org/0000-0002-8731-0989>, fobeidat@cadence.com)

Mohammad Al-Khaleel

(Department of Mathematics, Khalifa University, Abu Dhabi, UAE

Department of Mathematics, Yarmouk University, Irbid, Jordan

 <https://orcid.org/0000-0001-6266-373X>, mohammad.alkhaleel@ku.ac.ae)

Abstract: Multiplication is considered one of the most time-consuming and a key operation in wide variety of embedded applications. Speeding up this operation has a significant impact on the overall performance of these applications. A vast number of multiplication approaches are found in the literature where the goal is always to achieve a higher performance. One of these approaches relies on using smaller multiplier blocks which are built based on direct Boolean algebra equations to build large multipliers. In this work, we present a methodology for designing binary multipliers where different sizes customized partial products generation (CPPG) cells are designed and used as smaller building blocks. The sizes of the designed CPPG cells are 2×2 , 3×3 , 4×4 , 5×5 , and 6×6 . We use these cells to build 8×8 , 16×16 , 32×32 , 64×64 , and 128×128 binary multipliers. All of the CPPG cells and the binary multipliers are described using the VHDL language, tested, and implemented using XILINX ISE 14.6 tools targeting different FPGA families. The implementation results show that the best performance is achieved when cell 3×3 is used and Virtex-7 FPGA is targeted. The binary multipliers that are designed using the proposed CPPG cells achieve better performance when compared with the binary multipliers presented in the literature. As an application that utilizes the proposed multiplier, a Multiply-Accumulate (MAC) unit is designed and implemented in Spartan-3E. The implementation results of the MAC unit demonstrate the effectiveness of the proposed multiplier.

Keywords: Binary multiplier, Parallel Multiplier, Customized Cells, FPGA, VHDL

Categories: B.2.4, B.6.0

DOI: 10.3897/jucs.86282

1 Introduction

Binary multipliers play a significant role in microprocessor systems [Karthikeyan and Jagadeeswari, 2021][Kamrani and Heikalabad, 2021][Balasubramanian et al., 2021]. They are considered one of the critical components when it comes to performance in various kinds of Embedded Components, Digital Signals Processors, and Logical Units that execute different applications such as image processing, arithmetic, and filtering. Therefore, several research studies have been done to enhance the multiplication operation in order to improve the speed, the power consumption [Khaleqi Qaleh Jooq et al., 2021][Mounica et al., 2021], as well as the area [Sakthimohan and Deny, 2021]. Likewise, the pace of modern equipment has been increased, so the demand for high-performance systems has been rocketed [Seferlis et al., 2021]. As a result, any FPGAs vendors provide special Logic Cores for adders and multipliers. They use memory blocks and carry chains to enhance the performance of the multiplication operation [Véstias, 2021]. The multiplier blocks which are embedded in the FPGA families are fixed in the size. Therefore, bigger multipliers are built using a different number of these blocks as shown in [Langhammer and Pasca, 2021, Singh et al., 2021]. There are three main steps to perform the conventional multiplication operation as presented in [Mao et al., 2021][Hasan and Kort, 2007]. These steps are the partial products generation, the partial products reduction, and finally, the partial products summation to obtain the final product. For example, the authors in [BN and HG, 2021] show that the basic array multiplier for two N -bit binary numbers is designed such that each bit in the multiplicand is multiplied by each bit in the multiplier to generate N^2 1-bit wide partial products. This requires N^2 AND gates. Full adders are then used in a two-dimensional structure to add the partial products. However, fewer and larger size partial products can be obtained by processing more than one bit from the multiplicand and the multiplier at a time. For instance, processing two bits from the multiplicand and the multiplier results in having $\frac{N^2}{4}$ 4-bit wide partial products. Similarly, the resulting partial products are to be added to generate the final product. However, larger adders are needed as the partial products are wider in this case.

Generally, binary multipliers are categorized into two main categories: parallel and serial (iterative) multipliers. In both cases, the optimization of the partial products generation and the summation of the partial products have a big impact on the performance of the multiplier [Rafiee et al., 2021]. Partial products can be generated serially or in parallel by using AND gates at the bit level or by using customized partial products generation (CPPG) cells at higher levels. The partial products summation phase is directly related to the way how they are generated. The array and tree addition schemes are used for the parallel generation case. While serial addition or accumulation fits with the serial generation case.

Much larger hardware is required in the parallel multipliers in order to reduce the multiplication latency. To generate the 1-bit wide or the n -bit wide partial products concurrently in the parallel multipliers, many AND gates or CPPG cells are required. In this case, a parallel addition structure, that uses many adders, can be used to obtain the final product [Christilda and Milton, 2021][Lee and Burgess, 2003][Beuchat and Tisserand, 2002] in a shorter time compared to the iterative multipliers.

This work designs high-speed and low-area binary multipliers to be used in any application where the multiplication operations have a big impact on the overall performance of the application. Different size customized partial products generation (CPPG) cells are proposed to be used as building blocks in the binary multipliers. These CPPG

cells are designed following the traditional logic design and Boolean algebra. The impact of these cells on the performance is studied by using them in designing different size binary multipliers and testing their performance.

Our proposed multipliers are exact multipliers that produce exact results. They are not approximate multipliers. Approximate multipliers are smaller and faster than exact multipliers. However, they produce inexact results. Therefore, they are suitable for only applications that do not require exact results. If the approximate multiplier gives results far away from the exact ones, an application that uses this multiplier will not give the expected output. Thus, the efficiency of an approximate multiplier that is being used in an application must be verified such that the application still works properly and correctly. This is not the case with the exact multipliers, which can be used in any application, and the expected output of the application is still achieved.

The rest of this paper is organized as follows: Section 2 provides a summary of some related works in the literature. Section 3 provides a detailed explanation of the design methodology of the CPPG cells and how they are used to design binary multipliers. The experimental results and the comparisons are presented and discussed in Section 4. Finally, the conclusions are provided in Section 5.

2 Background and literature review

Many research studies in the literature address the binary multiplication operation and its implementation. Some of these studies focus on improving the multiplication process by the enhancement of the partial products generation only. On the other hand, there are several research studies that improve the multiplication process by enhancing the partial products generation and the partial products summation.

In [Asati and Chandrashekhar, 2009], the authors explore a new structure for multiplying two binary numbers using a hierarchical design approach. They design a (16×16) unsigned multiplier which is considered an array of array multiplier that has better performance than the array multiplier. This category of the multiplier is built based on smaller multiplication cells. A (2×2) multiplication cell is designed using the traditional combinational logic circuit design approach where truth table and Karnaugh map are used to generate optimized Boolean equations that model the design. The (2×2) multiplication cell is then used to design a (4×4) that is used to design an (8×8) multiplication cell. Finally, the (8×8) multiplication cell is used to design a (16×16) binary multiplier. The design is implemented using CMOS technology and is compared with the 16-bit Booth encoded Wallace tree multiplier, which is implemented in [Fadavi-Ardekani, 1993]. The design shows an improvement in terms of latency and power consumption with a doubled number of transistors. However, they do not investigate arbitrary or bigger multipliers other than the (16×16) multiplier.

The authors in [Das and Rahaman, 2010], pursue the same technique that is followed in [Asati and Chandrashekhar, 2009]. They use the 2×2 multiplier cell to design signed numbers binary multiplier. The partial products are added using parallel addition and the design is compared with the Baugh Wooley multiplier which is implemented in [Baugh and Wooley, 1973]. The proposed methodology provides better performance with larger area and higher power dissipation. Also, their proposed design expends more power rather than the Baugh Wooley multiplier. Similarly, they do not generalize their method for larger multipliers. Only 16×16 multiplier is investigated as in [Asati and Chandrashekhar, 2009].

A 128×128 multiplier is proposed in [Nagaraju and Reddy, 2014]. The authors invest the technique of ancient mathematics to implement and enhance their multiplier. They design the 128×128 multiplier by using 64×64 multiplier blocks, which are built from 3.96 ns 4×4 Vedic multiplier blocks. Better performance is achieved in comparison to the conventional multipliers.

An energy-efficient radix-16 Booth multiplier is proposed in [Mounica et al., 2021]. The design is used for both signed and unsigned numbers. The delay and energy are improved by optimizing the partial product generation unit. Two architectures of the multiplier are presented. One is non-pipelined and the other is pipelined. The signed, unsigned and combined (signed/unsigned) versions of the multiplier are investigated.

A sequential 8×8 multiplier is presented in [Hameed and Kathem, 2021]. An iterative addition approach is defined such that the number of iterative additions, required to generate the final product, is reduced. All of the intermediate shift operations, required in the conventional serial multiplication, are replaced by only one shift operation applied to the final result. Both of the proposed and conventional multipliers are implemented and compared in terms of time delay.

The work in [Fonseca et al., 2005] puts more effort to enhance the performance of binary multiplication using carry-save addition array structure presented in [Kim et al., 1998]. On the other hand, the authors use a type of 2×2 cell to generate the partial products and they build a hybrid multiplier. Their approach is based on the 2's complement encoding. Signed and unsigned operands are investigated. The authors compare their results with the modified Booth multiplier in terms of power, area, and delay. They use the Altera Quartus II and target the Altera Stratix FPGA family. The power consumption is reduced by 25% with area and delay penalty.

A radix-4 multiplexer-based array binary multiplier is presented in [Al-Khaleel et al., 2006]. The multiplier is used in designing a scalable and large moduli multiplier for public-key cryptographic systems. The authors target Virtex-4 FPGA family. To fit up to 512-bit modular multiplications on a single FPGA device, the authors use a novel partitioning and pipeline folding scheme.

A new architecture for the 8-bit Radix-4 modified Booth multiplier is presented in [Rafiq et al., 2021]. The proposed multiplier runs at a frequency of 500 MHz with low power and delay. A new encoder is used to reduce the first partial product array circuitry. The proposed multiplier is synthesized using Cadence Virtuoso and 90nm process technology.

The authors in [Singh and Singh, 2016a] utilize the work in [Singh and Singh, 2016b] to implement a 16×16 Vedic multiplier based on investing the architecture of the Brent Kung adder. The authors design a 2×2 Vedic multiplier and use it to build a 4×4 Vedic multiplier. Then, they used the 4×4 Vedic multiplier to design an 8×8 Vedic multiplier. Finally, they use four blocks of the 8×8 Vedic multiplier to design their 16×16 Vedic multiplier. They compare their work with Vedic multiplier using MUX-based adder.

A (64-bit) Vedic multiplier is presented in [Karthik et al., 2021]. The authors use half adders for adding the partial products. They achieve better performance in area and delay in comparison with the Array and Booth multipliers.

The authors in [Atre and Alshewimy, 2017] utilize both Wallace Tree and Modified Booth's Algorithm structure for designing different size multipliers. The sizes of the operands are 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit. Wallace Tree decreases the number of sequential summation steps to achieve better speed. They achieve an improvement in the time delay with some area penalty.

A technique of building a larger multiplier from smaller blocks is used in [Jais and Palsodkar, 2016]. A 64-bit Vedic multiplier is designed starting based on a 2-bit Vedic

multiplier. The authors target three different FPGA families: Spartan-6, Virtex-5, and Virtex-6. Verilog HDL is used to describe the designs and the results are compared with three familiar multipliers: Karatsuba, Vedic-Karatsuba, and Optimized Vedic multiplier.

Parallel computation of the partial products is investigated in [Naqvi, 2017] to design a 64-bit Vedic multiplier. The Carry-save adder and concatenation technique are used to enhance the performance of this multiplier. Three different FPGA devices (Spartan-3, Virtex-5, and Virtex-6) are targeted and a comparison between the proposed multiplier and four related works is made.

A carry-save adder is used instead of a parallel adder to design a modified Vedic multiplier for binary numbers in [Akhter and Chaturvedi, 2019]. The authors use four (8×8 -bit) Vedic multiplier blocks and 16-bit carry-save addition to build their 16×16 bit modified multiplier.

An 8-bit Vedic multiplier by using Urdhva Tiryaghyam sutra is proposed in [Chandrashekara and Rohith, 2019]. A modified carry-save adder is used in the addition phase. The work is compared against the array, the Wallace Tree, and the Booth multipliers. The proposed design shows better area and delay results compared with similar works in literature.

An approximate multiplier, which is efficiently deployed on FPGAs, is presented in [Van Toan and Lee, 2020]. The main goal is to achieve low power, small area, and high speed; while maintaining acceptable accuracy such that an application that utilizes the multiplier still gives proper output. An image processing application is used to verify the effectiveness and applicability of the multiplier.

The authors of [Moss et al., 2019] present a two-speed, radix-4, serial-parallel multiplier. The latency of their design depends on the multiplier value. This is because their multiplier is basically a modified radix-4 Booth multiplier that adds only the nonzero Booth encodings and skips over the zero operations. VerilogHDL and Intel Cyclone-V FPGA are used to implement the proposed multiplier. The implementation results, depending on the input set, show better performance over the standard parallel Booth multiplier in terms of area-time.

After a careful study of the related work presented in the literature, one can conclude that some works design a 2×2 multiplication cell using the traditional logic design and Boolean algebra. The 2×2 cell is used in these works to generate the partial products in a binary multiplier. Some works use such cell to design larger cells like 4×4 and 8×8 and then use them to design larger multipliers. However, up to the knowledge of the authors, no work in literature builds cells larger than 2×2 using the traditional logic design and Boolean algebra and uses them in designing large binary multipliers. Especially, when it comes to larger bit-width multipliers such as the 128×128 multiplier. Therefore, in this work, different sizes of customized partial products generation (CPPG) cells are designed using the traditional logic design and Boolean algebra. The CPPG cells are then used in designing 8×8 , 16×16 , 32×32 , 64×64 , and 128×128 binary multipliers. VHDL description for each CPPG cell and for each multiplier is developed. The VHDL is portable and can be implemented in any FPGA or ASIC platform. However, since we only have access to Xilinx platform, our implementations are done in Xilinx FPGAs using Xilinx CAD tools.

3 The proposed binary multipliers

In the proposed methodology, five customized partial products generation cells (CPPG) are designed using the traditional digital logic design approach, which is used in [Al-

Khaleel et al., 2011a], [Al-Khaleel et al., 2011b], [Al-Khaleel et al., 2013], and [Al-Khaleel et al., 2015] to obtain optimized Boolean equations. The optimization guarantees a minimal number of product terms and a minimal number of literals within each term. The number of the product terms affects the number of the AND gates, the number of the OR gates, and the size of the OR gates. While the number of literals within each term affects the size of the AND gates. The sizes of the proposed CPPG cells are 2×2 , 3×3 , 4×4 , 5×5 , and 6×6 . A $r \times r$ CPPG cell is used to multiply r bits from the multiplier by r bits from the multiplicand to generate one partial product of $2r$ bits. Multiple instances of each cell are used to concurrently generate the partial products in the binary multiplier. The generated partial products are then added to generate the final product. The cells are used to design binary multipliers with different operand sizes.

3.1 Using the 2×2 CPPG cell to design $n \times m$ Multiplier

The optimized Boolean functions for the 2×2 CPPG cell are given in Equation 1. This cell multiplies the 2-bit numbers $(a_1a_0$ and $b_1b_0)$ and produces the 4-bit partial product $c_3c_2c_1c_0$.

$$\begin{aligned} c_3 &= a_1a_0b_1b_0, \\ c_2 &= a_1b_1\bar{b}_0 \vee a_1\bar{a}_0b_1b_0, \\ c_1 &= \bar{a}_1a_0b_1 \vee a_1\bar{b}_1b_0 \vee a_1\bar{a}_0b_1b_0 \vee a_0b_1\bar{b}_0, \\ c_0 &= a_0b_0. \end{aligned} \tag{1}$$

To generalize the approach for any multiplier size, the sizes of the operands of the multiplier A and B are considered to be n -bit and m -bit, respectively. Both n and m must be multiple of 2 integers in this case. If this condition is not valid, then zeros can be padded to the left of the number such that the sizes of the operands are multiple of 2. A can be represented as shown by Equation 2.

$$A = (A_{n-1}A_{n-2}A_{n-3}A_{n-4} \dots A_3A_2A_1A_0)_2. \tag{2}$$

Furthermore, A can be decomposed as given in Equation 3.

$$\begin{aligned} A &= (A_{n-1}A_{n-2} \underbrace{00 \dots 00}_{(n-2)\text{zeros}})_2 + (00A_{n-3}A_{n-4} \underbrace{00 \dots 00}_{(n-4)\text{zeros}})_2 \\ &+ \dots + (\underbrace{00 \dots 00}_{(n-4)\text{zeros}} A_3A_200)_2 + (\underbrace{00 \dots 00}_{(n-2)\text{zeros}} A_1A_0)_2. \end{aligned} \tag{3}$$

Replace the k zeros on the right side of each term with 2^k to get Equation 4.

$$A = 2^{n-2}(A_{n-1}A_{n-2})_2 + 2^{n-4}(A_{n-3}A_{n-4})_2 + \dots + 2^2(A_3A_2)_2 + 2^0(A_1A_0)_2. \tag{4}$$

Let $X_{\frac{k}{2}} = (A_{k+1}A_k)_2$; $k = 0, 2, 4, \dots, n - 2$. Then A can be expressed as in Equation 5.

$$A = 2^{n-2}X_{\frac{n}{2}-1} + 2^{n-4}X_{\frac{n}{2}-2} + \dots + 2^2X_1 + 2^0X_0 = \sum_{i=0}^{\frac{n}{2}-1} 2^{2i}X_i. \tag{5}$$

Similarly, B can be represented and decomposed as given by Equations 6 and 7, respectively.

$$B = (B_{m-1}B_{m-2}B_{m-3}B_{m-4} \dots B_3B_2B_1B_0)_2, \tag{6}$$

$$B = (B_{m-1}B_{m-2} \underbrace{00 \dots 00}_{(m-2)\text{zeros}})_2 + (00B_{m-3}B_{m-4} \underbrace{00 \dots 00}_{(m-4)\text{zeros}})_2 + \dots + (\underbrace{00 \dots 00}_{(m-4)\text{zeros}} B_3B_200)_2 + (\underbrace{00 \dots 00}_{(m-2)\text{zeros}} B_1B_0)_2. \tag{7}$$

Replace the k zeros on the right side of each term with 2^k to get Equation 8.

$$B = 2^{m-2}(B_{m-1}B_{m-2})_2 + 2^{m-4}(B_{m-3}B_{m-4})_2 + \dots + 2^2(B_3B_2)_2 + 2^0(B_1B_0)_2. \tag{8}$$

Let $Y_{\frac{k}{2}} = (B_{k+1}B_k)_2$; $k = 0, 2, 4, \dots, m - 2$. Then B can be expressed by Equation 9.

$$B = 2^{m-2}Y_{\frac{m}{2}-1} + 2^{m-4}Y_{\frac{m}{2}-2} + \dots + 2^2Y_1 + 2^0Y_0 = \sum_{j=0}^{\frac{m}{2}-1} 2^{2j}Y_j. \tag{9}$$

Hence, the final product C can be obtained by Equation 10.

$$\begin{aligned} C &= A \times B \\ &= \sum_{i=0}^{\frac{n}{2}-1} 2^{2i}X_i \times \sum_{j=0}^{\frac{m}{2}-1} 2^{2j}Y_j \\ &= \sum_{i=0}^{\frac{n}{2}-1} \sum_{j=0}^{\frac{m}{2}-1} 2^{2i}X_i \times 2^{2j}Y_j = \sum_{i=0}^{\frac{n}{2}-1} \sum_{j=0}^{\frac{m}{2}-1} 2^{2(i+j)}X_iY_j. \end{aligned} \tag{10}$$

X_iY_j represents the partial product number $\frac{n}{2}i + j$, which is represented as $PP_{\frac{n}{2}i+j}$. Therefore, C can be represented in terms of the partial products as in Equation 11.

$$C = \sum_{i=0}^{\frac{n}{2}-1} \sum_{j=0}^{\frac{m}{2}-1} 2^{2(i+j)}PP_{\frac{n}{2}i+j}. \tag{11}$$

The total number of the partial products before shifting is $\frac{n}{2} \times \frac{m}{2} = \frac{mn}{2^2}$. These partial products are $PP_{\frac{n}{2}i+j}$ for $(i, j) = (0, 0), (0, 1), \dots, (0, \frac{m}{2} - 1), (1, 0), (1, 1), \dots, (\frac{n}{2} - 1, 0), (\frac{n}{2} - 1, 1), \dots, (\frac{n}{2} - 1, \frac{m}{2} - 1)$. As illustrated in Figure1, which presents the structure of an 8×8 multiplier using the 2×2 CPPG cell, each of these none shifted partial products is generated using one 2×2 CPPG cell in *Level0* of the structure. Therefore, the total number of the 2×2 CPPG cells that are required to generate the none shifted partial products is $\frac{mn}{2^2}$. These none shifted partial products are shifted and then added in the addition stage according to Equation 11. The shifted partial products are the outputs from *Level0* of the structure. They are obtained using Equation 12 where the superscript l represents the level that outputs the shifted partial products, which is in this case

Level0 (i.e. $l = 0$).

$$PP_{\frac{n}{2}i+j}^l = 2^{2(i+j)} PP_{\frac{n}{2}i+j},$$

for $l = 0$ and $(i, j) = (0, 0), (0, 1), \dots, (0, \frac{m}{2} - 1), (1, 0), (1, 1), \dots, (\frac{n}{2} - 1, 0), (\frac{n}{2} - 1, 1), \dots, (\frac{n}{2} - 1, \frac{m}{2} - 1)$. (12)

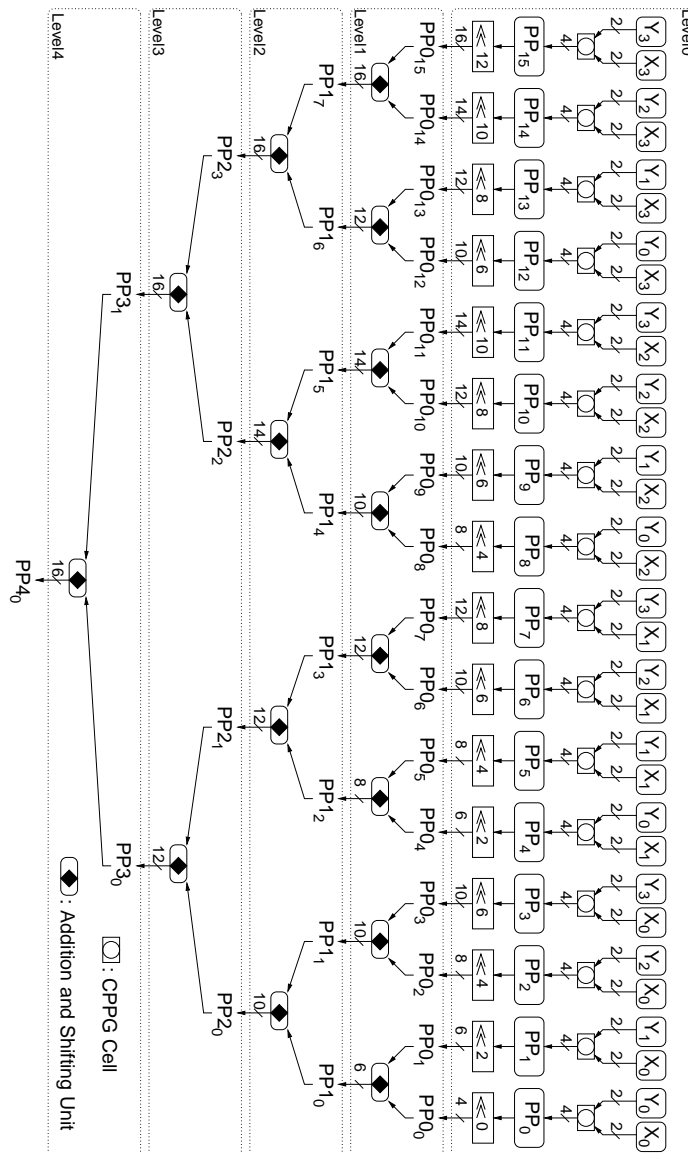


Figure 1: The structure of an 8×8 multiplier using the 2×2 CPPG cell.

For the addition phase, the tree structure is adopted to enhance the speed as illustrated in Figure 1. The shifted partial products from *Level0* are inputs to the first level (*Level1*) of the addition phase. The total number of inputs to *Level1* must be powers of 2. Therefore, if the number of the shifted partial products is not powers of 2, additional partial products with 0 values must be padded with the shifted partial products to form the inputs to *Level1*. The number of the additional partial products D is given by Equation 13. This means that the total number of inputs to *Level1* is $\frac{mn}{2^2} + D$. It should be pointed out that padding the 0 values additional partial products do not affect the performance or the area of the design as the design tools trim out any none used hardware. For example, if an adder is instantiated within the VHDL code and it happens that the inputs to the adder have 0 values, then the tool trims out this adder and it will not be part of the final design. The goal of having the number of the inputs to *Level1* powers of 2 is to maintain a regular structure that facilitates generic VHDL coding.

$$D = 2^{\lceil \log_2 \frac{mn}{2^2} \rceil} - \frac{mn}{2^2} \quad (13)$$

All of the shifted partial products from *Level0* and the additional padded partial products are added in the first level (*Level1*) of the addition phase such that each two consecutive partial products are added using one binary adder. That is PP_1^0 is added with PP_0^0 , PP_3^0 is added with PP_2^0 , and so on. In general, PP_{2t}^0 is added with PP_{2t+1}^0 for $t = 0, 1, 2, \dots, (\frac{mn}{2^2} + D)/2 - 1$. There are $(\frac{mn}{2^2} + D)/2$ output results from *Level1* of the addition phase, which are the inputs to *Level2*. Hence, there are $(\frac{mn}{2^2} + D)/4$ output results from *Level2* of the addition phase, which are the inputs to *Level3* and so on. In general, for a given level l in the addition stage, there are $(\frac{mn}{2^2} + D)/2^{l-1}$ inputs and $(\frac{mn}{2^2} + D)/2^l$ outputs. This means that the number of adders in a given level l is $(\frac{mn}{2^2} + D)/2^l$. The outputs of a given level l in the addition phase are obtained based on equation 14.

$$PP_t^l = (PP_{(2t+1)}^{l-1} + PP_{2t}^{l-1}), \text{ for } t = 0, 1, 2, 3, \dots, (\frac{mn}{2^2} + D)/2^l - 1. \quad (14)$$

In the last level ($l = L$), there are two results to be added. Hence, for the last level, it can be stated that $((\frac{mn}{2^2} + D)/2^{L-1} = 2)$. It can be easily shown that solving for L , which is an integer that represents the number of the levels in the addition phase, one gets $L = \log_2(\frac{mn}{2^2} + D)$.

3.2 Using the 3×3 CPPG cell to design $n \times m$ multiplier

The optimized Boolean functions for the 3×3 CPPG cell are obtained in a similar way to that used for the 2×2 CPPG cell. They are presented in Equation 15. This cell multiplies the 3-bit numbers $(a_2a_1a_0)$ and $(b_2b_1b_0)$ and produces the 6-bit partial product $c_5c_4c_3c_2c_1c_0$.

$$\begin{aligned}
 c_5 &= a_2 a_1 a_0 b_2 b_0 \vee a_2 a_0 b_2 b_1 b_0 \vee a_2 a_1 b_2 b_1, \\
 c_4 &= a_2 \bar{a}_1 a_0 b_2 \bar{b}_1 b_0 \vee a_2 \bar{a}_1 \bar{a}_0 b_2 b_1 b_0 \vee \bar{a}_2 a_1 a_0 b_2 b_1 \vee a_2 a_1 \bar{b}_2 b_1 b_0, \\
 &\quad \vee a_2 \bar{a}_0 b_2 \bar{b}_1 b_0 \vee a_2 \bar{a}_1 b_2 b_1 \bar{b}_0 \vee a_2 a_1 a_0 b_1 b_0 \vee a_2 b_2 \bar{b}_1 \bar{b}_0, \\
 c_3 &= \bar{a}_2 a_1 a_0 \bar{b}_2 b_1 b_0 \vee a_2 \bar{a}_1 a_0 b_2 \bar{b}_1 b_0 \vee \bar{a}_2 a_1 b_2 \bar{b}_1 \vee a_2 \bar{a}_1 \bar{a}_0 b_2 b_1 b_0 \vee a_2 \bar{a}_1 b_2 b_1 \bar{b}_0, \\
 &\quad \vee a_2 \bar{a}_1 \bar{b}_2 b_1 b_0 \vee \bar{a}_2 a_1 \bar{a}_0 b_2 \vee a_2 \bar{b}_2 b_1 \bar{b}_0 \vee a_1 b_2 \bar{b}_1 \bar{b}_0 \vee a_1 \bar{a}_0 b_2 b_0 \vee a_2 a_0 b_1 \bar{b}_0, \\
 c_2 &= a_2 \bar{a}_1 \bar{a}_0 b_2 b_1 b_0 \vee a_2 \bar{a}_0 b_2 \bar{b}_1 b_0 \vee a_2 \bar{a}_1 \bar{b}_2 b_1 b_0 \vee \bar{a}_2 a_1 \bar{a}_0 b_1 \vee a_2 \bar{b}_2 \bar{b}_1 b_0, \\
 &\quad \vee \bar{a}_1 a_0 b_2 \bar{b}_0 \vee a_1 \bar{b}_2 b_1 \bar{b}_0 \vee a_1 \bar{a}_0 b_1 \bar{b}_0 \vee a_0 b_2 \bar{b}_1 \bar{b}_0 \vee a_2 a_0 \bar{b}_2 b_0 \vee \bar{a}_2 a_0 b_2 b_0, \\
 c_1 &= \bar{a}_1 a_0 b_1 \vee a_1 \bar{a}_0 b_0 \vee a_1 \bar{b}_1 b_0 \vee a_0 b_1 \bar{b}_0, \\
 c_0 &= a_0 b_0.
 \end{aligned} \tag{15}$$

For generality, the sizes of the operands A and B are assumed to be n -bit and m -bit, respectively. Both n and m should be multiple of 3 integers in this case. If this condition is not valid, then again, zeros are padded to the left of the number to make the size of the operands multiple of 3. A can be represented as given by Equation 16.

$$A = (A_{n-1} A_{n-2} A_{n-3} \dots A_2 A_1 A_0)_2. \tag{16}$$

Moreover, A can be decomposed as shown in Equation 17.

$$\begin{aligned}
 A &= (A_{n-1} A_{n-2} A_{n-3} \underbrace{000 \dots 000}_{(n-3)\text{zeros}})_2 + (\underbrace{000 A_{n-4} A_{n-5} A_{n-6}}_{(n-6)\text{zeros}} \underbrace{000 \dots 000}_{(n-6)\text{zeros}})_2 \\
 &\quad + \dots + (\underbrace{000 \dots 000}_{(n-6)\text{zeros}} A_5 A_4 A_3 000)_2 + (\underbrace{000 \dots 000}_{(n-3)\text{zeros}} A_2 A_1 A_0)_2.
 \end{aligned} \tag{17}$$

Replacing the k zeros on the right side of each term with 2^k implies Equation 18.

$$\begin{aligned}
 A &= 2^{n-3} (A_{n-1} A_{n-2} A_{n-3})_2 + 2^{n-6} (A_{n-4} A_{n-5} A_{n-6})_2 + \dots + 2^3 (A_5 A_4 A_3)_2 \\
 &\quad + 2^0 (A_2 A_1 A_0)_2.
 \end{aligned} \tag{18}$$

Assume $X_{\frac{k}{3}} = (A_{k+2} A_{k+1} A_k)_2$; $k = 0, 3, 6, \dots, n-3$, then A can be expressed as in Equation 19.

$$A = 2^{n-3} X_{\frac{n}{3}-1} + 2^{n-6} X_{\frac{n}{3}-2} + \dots + 2^3 X_1 + 2^0 X_0 = \sum_{i=0}^{\frac{n}{3}-1} 2^{3i} X_i. \tag{19}$$

Similarly, B can be represented and decomposed as given by Equations 20 and 21, respectively.

$$B = (B_{m-1} B_{m-2} B_{m-3} \dots B_2 B_1 B_0)_2, \tag{20}$$

and

$$\begin{aligned}
 B = & (B_{m-1}B_{m-2}B_{m-3}\underbrace{000\dots000}_{(m-3)\text{zeros}})_2 + (000B_{m-4}B_{m-5}B_{m-6}\underbrace{000\dots000}_{(m-6)\text{zeros}})_2 \\
 & + \dots + (\underbrace{000\dots000}_{(m-6)\text{zeros}}B_5B_4B_3000)_2 + (\underbrace{000\dots000}_{(m-3)\text{zeros}}B_2B_1B_0)_2.
 \end{aligned} \tag{21}$$

Replacing the k zeros on the right side of each term with 2^k leads to Equation 22.

$$\begin{aligned}
 B = & 2^{m-3}(B_{m-1}B_{m-2}B_{m-3})_2 + 2^{m-6}(B_{m-4}B_{m-5}B_{m-6})_2 \\
 & + \dots + 2^3(B_5B_4B_3)_2 + 2^0(B_2B_1B_0)_2.
 \end{aligned} \tag{22}$$

Assume $Y_{\frac{k}{3}} = (B_{k+2}B_{k+1}B_k)_2$; $k = 0, 3, 6, \dots, m - 3$ then B can be expressed by Equation 23.

$$B = 2^{m-3}Y_{\frac{m}{3}-1} + 2^{m-6}Y_{\frac{m}{3}-2} + \dots + 2^3Y_1 + 2^0Y_0 = \sum_{j=0}^{\frac{m}{3}-1} 2^{3j}Y_j. \tag{23}$$

Hence, the final product C can be obtained by Equation 24.

$$\begin{aligned}
 C = A \times B = & \sum_{i=0}^{\frac{n}{3}-1} 2^{3i}X_i \times \sum_{j=0}^{\frac{m}{3}-1} 2^{3j}Y_j = \sum_{i=0}^{\frac{n}{3}-1} \sum_{j=0}^{\frac{m}{3}-1} 2^{3i}X_i \times 2^{3j}Y_j \\
 & = \sum_{i=0}^{\frac{n}{3}-1} \sum_{j=0}^{\frac{m}{3}-1} 2^{3(i+j)}X_iY_j.
 \end{aligned} \tag{24}$$

X_iY_j represents the partial product number $\frac{n}{3}i + j$, which is represented as $PP_{\frac{n}{3}i+j}$. Therefore, C can be represented in terms of the partial products as in Equation 25.

$$C = \sum_{i=0}^{\frac{n}{3}-1} \sum_{j=0}^{\frac{m}{3}-1} 2^{3(i+j)}PP_{\frac{n}{3}i+j}. \tag{25}$$

In this case, the total number of the partial products before shifting is $\frac{n}{3} \times \frac{m}{3} = \frac{mn}{3^2}$. These partial products are:

$PP_{\frac{n}{3}i+j}$ for $(i, j) = (0, 0), (0, 1), \dots, (0, \frac{m}{3} - 1), (1, 0), (1, 1), \dots, (\frac{n}{3} - 1, 0), (\frac{n}{3} - 1, 1), \dots, (\frac{n}{3} - 1, \frac{m}{3} - 1)$.

Again, the process is illustrated in Figure 2 where the structure of an 8×8 multiplier using the the 3×3 CPPG cell is presented. Since 8 is not multiple of 3, a 0 is padded to the left of the most significant bit of multiplier and to the left of the most significant bit of the multiplicand. As a result both n and m become 9-bit in size. Each of the none shifted partial products is generated using one 3×3 CPPG cell in *Level0* of the structure. Therefore, the total number of the 3×3 CPPG cells that are required to generate the none shifted partial products is $\frac{mn}{3^2}$. These none shifted partial products are shifted and then added in the addition stage according to Equation 25. The shifted partial products are

the outputs from *Level0* of the structure. They are obtained using Equation 26 where the superscript l represents the level that outputs the shifted partial products, which is in this case *Level0* (i.e. $l = 0$).

$$PP_{\frac{n}{3}i+j}^l = 2^{3(i+j)} PP_{\frac{n}{3}i+j},$$

for $l = 0$ and $(i, j) = (0, 0), (0, 1), \dots, (0, \frac{n}{3} - 1), (1, 0), (1, 1), \dots, (\frac{n}{3} - 1, 0), (\frac{n}{3} - 1, 1), \dots, (\frac{n}{3} - 1, \frac{n}{3} - 1)$. (26)

The number of the shifted partial products from *Level0* is 9 which is not powers of 2. Therefore, additional partial products with 0 values must be padded with the shifted partial products to form the inputs to *Level1*. The number of the additional partial products D is given by Equation 27. In this case $D = 7$. This means that the total number of inputs to *Level1* is $\frac{mn}{3^2} + D = 16$.

$$D = 2^{\lceil \log_2 \frac{mn}{3^2} \rceil} - \frac{mn}{3^2} \quad (27)$$

Again, all of the shifted partial products from *Level0* and the additional padded partial products are added in the first level (*Level1*) of the addition phase such that each two consecutive partial products are added using one binary adder. That is PP_1^0 is added with PP_0^0 , PP_3^0 is added with PP_2^0 , and so on. In general, PP_{2t}^0 is added with PP_{2t+1}^0 for $t = 0, 1, 2, \dots, (\frac{mn}{3^2} + D)/2 - 1$. There are $(\frac{mn}{3^2} + D)/2$ output results from *Level1* of the addition phase, which are the inputs to *Level2*. Hence, there are $(\frac{mn}{3^2} + D)/4$ output results from *Level2* of the addition phase, which are the inputs to *Level3* and so on. In general, for a given level l in the addition stage, there are $(\frac{mn}{3^2} + D)/2^{l-1}$ inputs and $(\frac{mn}{3^2} + D)/2^l$ outputs. This means that the number of adders in a given level l is $(\frac{mn}{3^2} + D)/2^l$. The outputs of a given level l in the addition phase are obtained based on equation 28.

$$PP_t^l = (PP_{(2t+1)}^{l-1} + PP_{2t}^{l-1}), \text{ for } t = 0, 1, 2, 3, \dots, (\frac{mn}{3^2} + D)/2^l - 1. \quad (28)$$

In the last level ($l = L$), there are two results to be added. Hence, for the last level, it can be stated that $((\frac{mn}{3^2} + D)/2^{L-1} = 2)$. It can be easily shown that solving for L , which is an integer that represents the number of the levels in the addition phase, one gets $L = \log_2(\frac{mn}{3^2} + D)$.

It should be emphasized again that all of the extra unnecessary hardware, due to the padded 0's to the operands or the padded 0 values additional partial products, is trimmed out, by the CAD tool, in the final design. The final design after the tool trims out the unnecessary hardware is shown in Figure 3

We follow the same previous techniques for designing binary multipliers using 4×4 , 5×5 and 6×6 CPPG cells¹.

The aforementioned technique is generalized in Algorithm 1 for any CPPG cell size $r \times r$ and arbitrary operands sizes n and m .

¹ The VHDL code for each of the CPPG cells is posted at [Al-Nounou, 2022] for the community.

Algorithm 1 Using $r \times r$ CPPG cell to design $n \times m$ Multiplier.**Input:** Operand A , Operand B , n : size of A , m : size of B , r : size of CPPG cell**Output:** $C = A \times B$ **Require:** n, m multiple of r , otherwise, zeros padded to the left of the number

```

1: if  $A = A_{n-1}A_{n-2}A_{n-3}A_{n-4} \dots A_3A_2A_1A_0$  then
2:    $A = 2^{n-r}A_{n-1}A_{n-2}A_{n-3} \dots A_{n-r+2}A_{n-r+1}A_{n-r}$ 
      $+ 2^{n-2r}A_{n-r-1}A_{n-r-2}A_{n-r-3} \dots A_{n-2r+2}A_{n-2r+1}A_{n-2r}$ 
      $+ \dots + 2^{2r}A_{3r-1}A_{3r-2}A_{3r-3} \dots A_{2r+2}A_{2r+1}A_{2r}$ 
      $+ 2^rA_{2r-1}A_{2r-2}A_{2r-3} \dots A_{r+2}A_{r+1}A_r$ 
      $+ 2^0A_{r-1}A_{r-2}A_{r-3} \dots A_2A_1A_0$ 
3: end if
4: if  $B = B_{m-1}B_{m-2}B_{m-3}A_{m-4} \dots B_3A_2B_1B_0$  then
5:    $B = 2^{m-r}B_{m-1}B_{m-2}B_{m-3} \dots B_{m-r+2}B_{m-r+1}B_{m-r}$ 
      $+ 2^{m-2r}B_{m-r-1}B_{m-r-2}B_{m-r-3} \dots B_{m-2r+2}B_{m-2r+1}B_{m-2r}$ 
      $+ \dots + 2^{2r}B_{3r-1}B_{3r-2}B_{3r-3} \dots B_{2r+2}B_{2r+1}B_{2r}$ 
      $+ 2^rB_{2r-1}B_{2r-2}B_{2r-3} \dots B_{r+2}B_{r+1}B_r$ 
      $+ 2^0B_{r-1}B_{r-2}B_{r-3} \dots B_2B_1B_0$ 
6: end if
7: for  $s = 1, 2, 3, \dots, \frac{n}{r} - 2, \frac{n}{r} - 1, \frac{n}{r}$  do
8:    $k = n - rs$ 
9:    $X_{\frac{k}{r}} = A_{k+r-1}A_{k+r-2}A_{k+r-3} \dots A_{k+2}A_{k+1}A_k$ 
10:   $A = \sum_{i=0}^{\frac{n}{r}-1} 2^{ri} X_i$ 
11: end for
12: for  $s = 1, 2, 3, \dots, \frac{m}{r} - 2, \frac{m}{r} - 1, \frac{m}{r}$  do
13:    $k = m - rs$ 
14:    $Y_{\frac{k}{r}} = B_{k+r-1}B_{k+r-2}B_{k+r-3} \dots B_{k+1}B_{k+1}B_k$ 
15:    $B = \sum_{j=0}^{\frac{m}{r}-1} 2^{rj} Y_j$ 
16: end for
17:  $D = 2^{\lceil (\log_2 \frac{mn}{r^2}) \rceil} - \frac{mn}{r^2}$ 
18: for  $i = 0$  to  $\frac{n}{r} - 1$  do
19:   for  $j = 0$  to  $\frac{m}{r} - 1$  do
20:      $PP_{\frac{n}{r}i+j} = X_i Y_j$ 
21:      $PP_{\frac{n}{2}i+j}^0 = 2^{2(i+j)} PP_{\frac{n}{2}i+j}$ 
22:   end for
23: end for
24: Pad  $D$  additional partial products of 0 values to the shifted partial products.
25:  $L = \log_2(\frac{mn}{r^2} + D)$ 
26: for  $l = 1$  to  $L$  do
27:   for  $t = 0$  to  $(\frac{mn}{r^2} + D)/2^l - 1$  do
28:      $PP_t^l = (PP_{(2t+1)}^{l-1} + PP_{2t}^{l-1})$ 
29:   end for
30: end for
31:  $C = PP_0^L((m+n) - 1 \text{ downto } 0)$ 

```

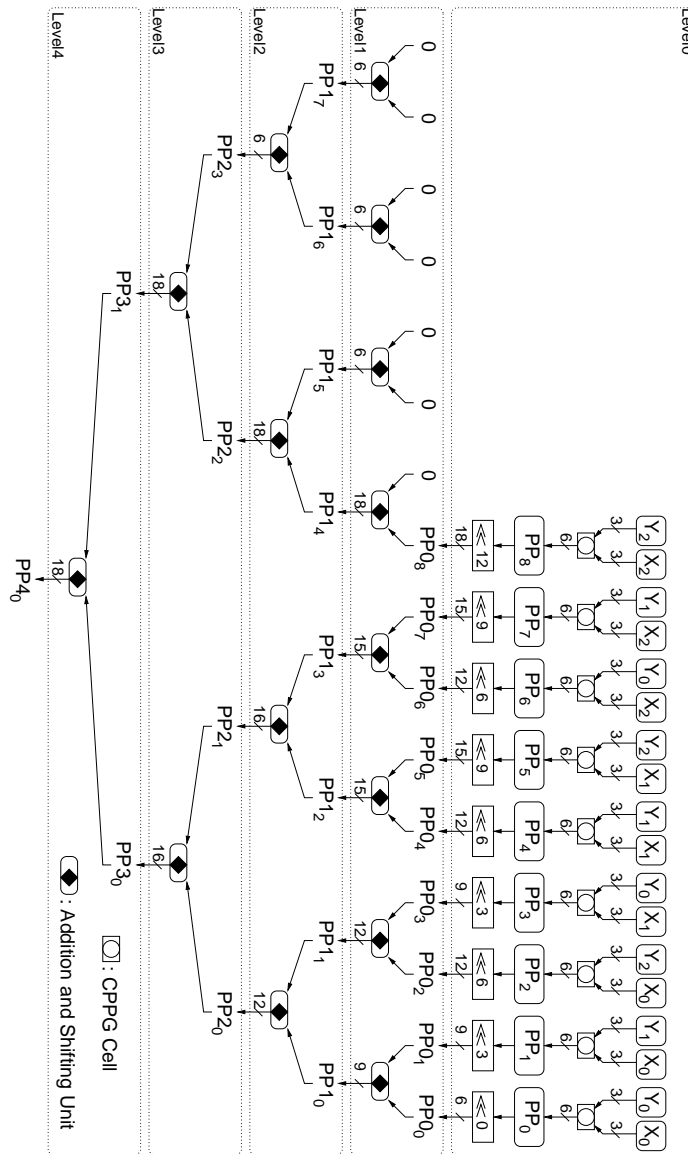


Figure 2: The structure of an 8×8 multiplier using the 3×3 CPPG cell.

4 Results and discussions

The proposed CPPG cell presented in Section 3 are used to design 8×8 , 16×16 , 32×32 , 64×64 , and 128×128 binary multipliers. All of the proposed CPPG cells and the designed binary multipliers are described and functionally verified using VHDL and Xilinx ISE 14.6 simulation tools. In addition, the proposed CPPG cells and the designed

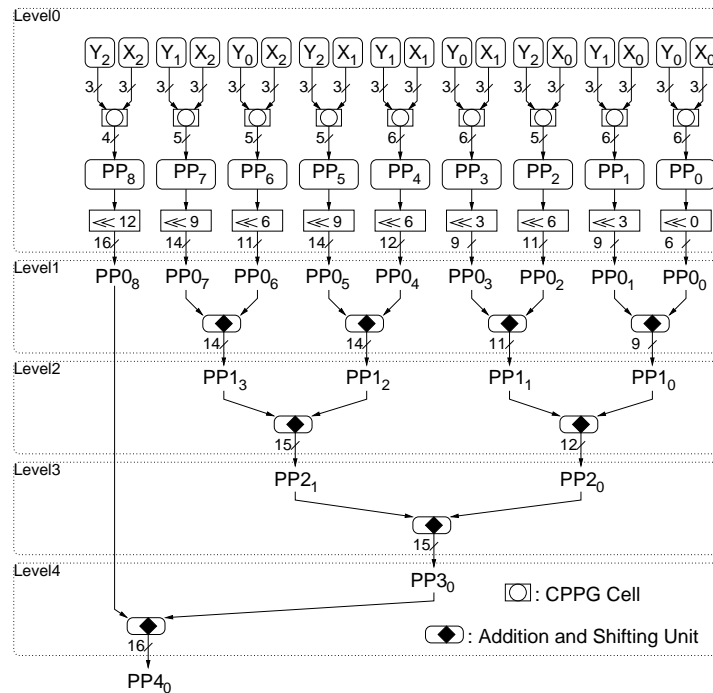


Figure 3: The structure of an 8×8 multiplier using the 3×3 CPPG cell after the tool trims out unnecessary hardware.

binary multipliers are synthesized and implemented using Xilinx ISE 14.6 synthesis and implementation tools. Different FPGA families and devices are targeted.

4.1 Implementation of the proposed CPPG cells

The proposed CPPG cells are implemented using different FPGA families. The implementation results are listed in Table 1. As expected, the area and delay increase with the size of the cell. Generally, the best delay is achieved when the Virtex-7 FPGA family is targeted. The delay results are reported after the synthesis process of the tool. The implementation results for the 128×128 and 64×64 binary multipliers using the different CPPG cells targeting the Virtex-7 FPGA family are listed in Table 2. While, those for the 32×32 and 16×16 binary multipliers are listed in Table 3.

It can be observed that, for all investigated multipliers, the best area is achieved when utilizing the 3×3 CPPG cell. While, the best delay for the 128×128 , the 64×64 , and the 32×32 multipliers is achieved when utilizing the 4×4 CPPG cell. For the 16×16 multiplier, the best delay is achieved when utilizing the 5×5 CPPG cell. The second best area for the 128×128 and the 64×64 multipliers is achieved with the 4×4 CPPG cell. While the second best area for the 32×32 and the 16×16 multipliers is achieved with the 2×2 CPPG cell. The second best delay for the 128×128 , the 64×64 , and the 32×32 multipliers is achieved when utilizing the 2×2 CPPG cell. On the other hand, the second best delay for the 16×16 multiplier is achieved with the 4×4 CPPG cell. Utilizing the 6×6

| FPGA | Metric | CPPG cell size | | | | |
|----------|---------------------|----------------|-------|-------|-------|-------|
| | | 2×2 | 3×3 | 4×4 | 5×5 | 6×6 |
| Virtex-5 | Area (Slice LUTs) | 4 | 6 | 32 | 199 | 785 |
| | Area (Slices) | 2 | 2 | 11 | 74 | 344 |
| | Delay (<i>ns</i>) | 3.885 | 4.129 | 5.595 | 6.996 | 9.778 |
| Virtex-6 | Area (Slice LUTs) | 2 | 5 | 39 | 134 | 463 |
| | Area (Slices) | 1 | 2 | 19 | 54 | 156 |
| | Delay (<i>ns</i>) | 0.989 | 1.13 | 2.953 | 3.57 | 4.913 |
| Virtex-7 | Area (Slice LUTs) | 2 | 5 | 39 | 135 | 458 |
| | Area (Slices) | 1 | 3 | 21 | 54 | 154 |
| | Delay (<i>ns</i>) | 0.921 | 1.023 | 2.595 | 3.217 | 4.73 |

Table 1: Implementation results for the proposed CPPG cells over different FPGA families

| CPPG cell | 128×128 | | | 64×64 | | |
|-----------|-------------|---------------|---------------------|-------------|---------------|---------------------|
| | Area (LUTs) | Area (Slices) | Delay (<i>ns</i>) | Area (LUTs) | Area (Slices) | Delay (<i>ns</i>) |
| 2×2 | 137947 | 39278 | 15.388 | 19457 | 6113 | 11.952 |
| 3×3 | 71356 | 20031 | 16.608 | 12126 | 4214 | 12.448 |
| 4×4 | 80293 | 28733 | 15.081 | 16247 | 5674 | 11.673 |
| 5×5 | 114471 | 39157 | 17 | 20616 | 7265 | 13.325 |
| 6×6 | 235362 | 74207 | 15.853 | 42178 | 15181 | 12.985 |

Table 2: Implementation results for the 128×128 and 64×64 binary multipliers using the proposed CPPG cells and targeting the Virtex-7 FPGA family.

| CPPG cell | 32×32 | | | 16×16 | | |
|-----------|-------------|---------------|---------------------|-------------|---------------|---------------------|
| | Area (LUTs) | Area (Slices) | Delay (<i>ns</i>) | Area (LUTs) | Area (Slices) | Delay (<i>ns</i>) |
| 2×2 | 3135 | 1044 | 9.232 | 675 | 314 | 7.056 |
| 3×3 | 2267 | 808 | 9.475 | 508 | 243 | 7.821 |
| 4×4 | 3567 | 1184 | 8.865 | 807 | 364 | 7.023 |
| 5×5 | 4762 | 1673 | 9.456 | 1172 | 419 | 6.82 |
| 6×6 | 10249 | 3557 | 10.089 | 2300 | 716 | 7.304 |

Table 3: Implementation results for the 32×32 and 16×16 binary multipliers using the proposed CPPG cells and targeting the Virtex-7 FPGA family.

CPPG cell reduces the number of partial products and the complexity of the addition phase. Theoretically, this would result in better performance. Unexpectedly, the implementation results show that the worst performance is achieved whenever the 6×6 CPPG cell is used. The huge complexity of the Boolean equations of the 6×6 CPPG cell can be a justification for this drawback in performance. It happens that the 3×3 CPPG cell provides an acceptable reduction in the number of partial products while maintaining low hardware complexity.

4.2 Comparisons with works in literature

The proposed binary multipliers in this work are compared with different works presented in the literature. For a fair comparison, the size of the proposed binary multipliers is chosen to be equal to the size of the binary multipliers presented in the literature. In addition, the implementation environment is chosen to be the same as that used in the work from the literature. That is, the FPGA family used to implement the proposed binary multiplier is chosen to be the same FPGA family that is used to implement the binary multipliers presented in the literature. Each work from the literature that is to be used for the comparison purpose is represented with a symbol according to Table 4.

| Symbol | Work |
|--------|-----------------------------------|
| W_a | [Akhter and Chaturvedi, 2019] |
| W_b | [Sharma, 2015] |
| W_c | [Mani et al., 2015] |
| W_d | [Atre and Alshewimy, 2017] |
| W_e | [Chaudhary and Kularia, 2016] |
| W_f | [Bais and Khan Ali, 2016] |
| W_g | [Solanki et al., 2021] |
| W_h | [Chandrashekara and Rohith, 2019] |
| W_i | [Van Toan and Lee, 2020] |
| W_j | [Naqvi, 2017] |
| W_k | [Jais and Palsodkar, 2016] |
| W_l | [Wallace, 1964] |
| W_m | [Waters and Swartzlander, 2010] |
| W_n | [Asif and Kong, 2015a] |
| W_o | [Asif and Kong, 2015b] |
| W_p | [Saha et al., 2018] |
| W_q | [Fritz and Fam, 2017] |
| W_r | [Murugeswari and Mohideen, 2014] |

Table 4: Symbols that are used to represent some works from the literature.

Table 5 lists the comparison results for operand sizes of 4 and 8. While, Table 6 lists the comparison results for operand sizes of 16, 32, 64, and 128. The results reported for our designs are based on utilizing the 2×2 CPPG cell. In these two tables, if a result is not reported for a reference then we use 'not reported' (NR) in the corresponding cell. Generally, the proposed multipliers outperform those presented in the literature in terms of delay with an area penalty in some cases.

Based on Tables 5 and 6, the proposed 16×16 multiplier, over Virtex-4 FPGA, achieves 21.8% reduction in delay and 47.9% reduction in area comparing to its counterpart in [Atre and Alshewimy, 2017]. It is also clear that over Spartan-3E FPGA, it beats the 16×16 multiplier of [Solanki et al., 2021] in terms of delay with 18% extra area penalty. The authors in [Solanki et al., 2021] mention that the minimum periods in their 8×8 and 16×16 multipliers are 9.29 and 10.27 ns, respectively. The cycles needed to output the product are 16 and 32 cycles, respectively. Hence, the delay of their

| Operand size | Work | FPGA | Area (Slices) | Area (LUTs) | Delay (ns) |
|-------------------|-----------------|------------------|---------------|-------------|------------|
| 4 | W_a | Virtex-4 | NR | 30 | 8.54 |
| | <i>Proposed</i> | | 43 | 84 | 7.557 |
| | W_a | Spartan-3 | NR | 30 | 15.42 |
| | <i>Proposed</i> | | 42 | 83 | 13.382 |
| | W_b (WT) | Spartan-3E | 22 | 39 | 18.534 |
| | W_b (Dadda) | | 23 | 39 | 17.864 |
| | <i>Proposed</i> | | 15 | 27 | 12.738 |
| | W_a | Spartan-6 | NR | 24 | 10.43 |
| | <i>Proposed</i> | | 14 | 39 | 8.568 |
| | W_c (WT) | Artix-7 | NR | 26 | 11.109 |
| | W_c (IWT) | | NR | 22 | 8.437 |
| | <i>Proposed</i> | | NR | 25 | 4.706 |
| | 8 | W_d (Radix-32) | Virtex-4 | 254 | NR |
| W_a | | NR | | 153 | 13.87 |
| <i>Proposed</i> | | 76 | | 137 | 12.247 |
| W_a | | Spartan-3 | NR | 153 | 28.73 |
| W_e | | | 105 | 192 | 28.669 |
| W_f (WT) | | | 97 | 181 | 28.601 |
| <i>Proposed</i> | | 75 | 137 | 20.694 | |
| W_g | | Spartan-3E | NR | 161 | 148.64 |
| W_b (WT) | | | 116 | 205 | 34.629 |
| W_b (Dadda) | | | 124 | 218 | 30.75 |
| <i>Proposed</i> | | | 75 | 127 | 16.595 |
| W_a | | Spartan-6 | NR | 125 | 18.46 |
| W_h (Booth) | | | NR | 216 | 25.86 |
| W_h (Array) | | | NR | 130 | 23.106 |
| W_h (WT) | | | NR | 145 | 16.478 |
| W_h (Vedic) | | | NR | 128 | 14.219 |
| W_i (LUT-Based) | | | NR | 204 | 5.15 |
| W_i (DSP-Based) | | | NR | 371 | 6.25 |
| <i>Proposed</i> | | | 57 | 122 | 11.208 |
| W_c (WT) | | Artix-7 | NR | 205 | 26.501 |
| W_c (IWT) | | | NR | 128 | 11.314 |
| <i>Proposed</i> | | | NR | 144 | 6.281 |

Table 5: Comparisons with different works presented in literature for operand sizes of 4 and 8.

designs is computed as $(16 \times 9.29 = 148.64ns$ for the 8×8 multiplier) and $(32 \times 10.27 = 328.64ns$ for the 16×16 multiplier). In fact, their major concern is low-power design.

The proposed multipliers in [Van Toan and Lee, 2020] are approximate multipliers that do not provide exact results as produced by the proposed multipliers in this work. These approximate multipliers can be used in applications where exact results are not

| Operand size | Work | FPGA | Area (Slices) | Area (LUT's) | Delay (ns) |
|-----------------|-------------------|------------------|---------------|--------------|------------|
| 16 | W_d (Radix-32) | Virtex-4 | 726 | NR | 20.35 |
| | <i>Proposed</i> | | 378 | 667 | 15.913 |
| | W_f (WT) | Spartan-3 | 444 | 819 | 51.090 |
| | <i>Proposed</i> | | 373 | 672 | 32.174 |
| | W_g | Spartan-3E | NR | 473 | 328.64 |
| | <i>Proposed</i> | | 377 | 578 | 22.234 |
| | W_i (LUT-Based) | Spartan-6 | NR | 923 | 7.46 |
| | W_i (DSP-Based) | | NR | 372 | 6.40 |
| <i>Proposed</i> | 323 | | 673 | 17.26 | |
| 32 | W_d (Radix-32) | Virtex-4 | 2400 | NR | 27.95 |
| | <i>Proposed</i> | | 1877 | 3135 | 20.242 |
| | W_f (WT) | Spartan-3 | 2021 | 3765 | 88.114 |
| | <i>Proposed</i> | | 1593 | 2896 | 41.397 |
| | W_i (LUT-Based) | Spartan-6 | NR | 3499 | 11.30 |
| | W_i (DSP-Based) | | NR | 1496 | 18.04 |
| | <i>Proposed</i> | | 1179 | 3152 | 21.31 |
| 64 | W_d (Radix-32) | Virtex-4 | 8770 | NR | 46.11 |
| | W_e (Array) | | 4653 | 9110 | 1525.592 |
| | W_e (Booth) | | 8097 | 16192 | 138.25 |
| | W_e (Vedic) | | 1817 | 2289 | 29.967 |
| | <i>Proposed</i> | | 8458 | 13773 | 24.442 |
| | W_j | Virtex-5 | NR | 8185 | 21.243 |
| | W_k | | NR | 2622 | 21.985 |
| | <i>Proposed</i> | | 4305 | 11210 | 17.176 |
| | W_k | Virtex-6 | NR | 2622 | 15.77 |
| | W_j | | NR | 8185 | 12.717 |
| | <i>Proposed</i> | | 5944 | 19482 | 12.137 |
| | W_k | Spartan-6 | NR | 2622 | 24.932 |
| | <i>Proposed</i> | | 6627 | 19551 | 22.328 |
| | 128 | W_d (Radix-32) | Virtex-4 | 31960 | NR |
| <i>Proposed</i> | | 44517 | | 63724 | 32.184 |

Table 6: Comparisons with different works presented in literature for operand sizes of 16, 32, 64, and 128.

mandatory for the application to produce the expected output. However, the authors use the core generator of the Xilinx Tools to generate two IP cores for comparison purposes. The first one is a LUT-Based exact multiplier and the second one is a DSP-Based exact multiplier.

For the LUT-Based core, the area is reported in terms of the number of LUTs and the number of CARRY4 embedded components, which consists of four 2×1 -MUXs and four XOR gates. Therefore, the area of each CARRY4 embedded component can be estimated to be 8 LUTs. As a result, the area of the LUT-Based core is the number

of the reported LUTs plus the number of reported CARRY4 components multiplied by 8. For example, for the 16×16 LUT-Based multiplier, the authors reported that the area is $(299 \text{ LUTs} + 78 \text{ CARRY4})$. This means that the estimated area in LUTs is $(299 + 78 \times 8 = 923 \text{ LUTs})$.

For the DSP-Based core, the area is reported in terms of the number of LUTs and the number of utilized DSP embedded components. The DSPs in Xilinx FPGAs are known to be 18×18 in size. The estimated area of a DSP component in terms of LUTs can be easily extracted from the Core Generator tool itself. It is founded that the area for an 18×18 DSP core is 364 LUTs. As a result, the area of the DSP-Based core is the number of the reported LUTs plus the number of the reported DSP components multiplied by 364. For example, for the 16×16 LUT-Based multiplier, the authors reported that the area is $(8 \text{ LUTs} + 1 \text{ DSP})$. This means that the estimated area in LUTs is $(8 + 1 \times 364 = 372 \text{ LUTs})$.

From Tables 5 and 6, it is clear that the proposed multipliers in this work achieves smaller area comparing to the LUT-Based 8×8 , the 16×16 , and the 32×32 multipliers presented in [Van Toan and Lee, 2020]. The proposed 8×8 multiplier is smaller in area than the 8×8 DSP-Based multiplier of [Van Toan and Lee, 2020]. While, the 16×16 , and the 32×32 DSP-Based multipliers of [Van Toan and Lee, 2020] are smaller than the proposed ones. When it comes to the delay, it looks that the multipliers of [Van Toan and Lee, 2020] are faster for all operand sizes. However, while investigating the area of the 18×18 DSP, we noticed that the authors in [Van Toan and Lee, 2020] have reported the delay of their designs after registering the inputs and the output. This means that their delay is the minimum period of the design and it is not a combinational path delay that involves the delay of the IO buffers as in our case. The IO buffers delay is the routing delay from the FPGA pins to the ports of the design. This delay varies according to where the tool places the design inside the FPGA. When registering the inputs and the output of our 8×8 multiplier, we get a minimum period of 7.877 ns over Spartan-6 FPGA. Also, if we exclude the IO buffer delay, then the delays of our combinational 8×8 , 16×16 , and the 32×32 multipliers over Spartan-6 FPGA become 6.94 ns , 10.89 ns , and 14.52 ns , respectively. Moreover, Two points should be raised here. The first one is that the multipliers of [Van Toan and Lee, 2020] are generated IP cores that are part of the CAD tool itself and these IP cores can be generated for operand size of up to a maximum 64×64 and not more than that, which limits the scalability of the multiplication in [Van Toan and Lee, 2020]. While the proposed approach in this work is scalable and is not limited to any operand size. The second point is that using these embedded IP cores limits the portability of the HDL code to Xilinx FPGA platforms. While in the proposed approach, the HDL can be implemented in any FPGA or ASIC platform.

Over Spartan-3E FPGA and comparing to the 4×4 Wallace Tree (WT) and Dadda multipliers of [Sharma, 2015], the proposed multiplier achieves 31.2% and 28.69% improvement in speed, respectively. The area is also reduced by 31.8% and 34.7%, respectively. For the size of 8×8 over Spartan-3E FPGA, the proposed multiplier also shows better results in terms of area and speed compared to those WT and Dadda multipliers of [Sharma, 2015]. In fact, there is 35.3% and 39.5% reduction in the area with 52% and 46% reduction in delay, respectively.

Furthermore, the proposed 8×8 multiplier over Spartan-6 FPGA achieves 56.6%, 51.4%, and 31.9% improvement in speed compared to the Booth, Array, and WT multipliers of [Chandrashekara and Rohith, 2019], respectively. In this case, the area of the proposed multiplier is reduced by 43.5% compared to the Booth multipliers and slightly reduced compared to the Array and WT multipliers.

The proposed 8×8 multiplier, over Artix-7 FPGA, achieves 44.4% reduction in delay with only 11.1% increase in the area compared to the Improved Wallace Tree (IWT) multiplier of [Mani et al., 2015]

Over Spartan-3, the proposed 16×16 multiplier outperforms its counterpart WT multiplier in [Bais and Khan Ali, 2016] in terms of area and speed. It is 1.58 times faster and 1.19 times smaller. On the other hand, the proposed 32×32 multiplier is 2.12 times faster and 1.26 smaller compared with the 32×32 WT multiplier in [Bais and Khan Ali, 2016].

The 64×64 Booth and Array multipliers in [Chaudhary and Kularia, 2016] are also compared with the proposed multiplier over Virtex-4 FPGA. The results show that the proposed multipliers achieve much better speed with a considerable area penalty, especially in the case of the Array multiplier.

The proposed 64×64 multiplier shows minor speed improvement over the multiplier in [Naqvi, 2017] when targeting Virtex-6 FPGA. However, over Virtex-5 FPGA, it shows 19.14% speed improvement with 26.98% extra area.

Targeting Virtex-4 FPGA, a 46.3% speed improvement is achieved by the proposed 128×128 multiplier over the multiplier in [Atre and Alshewimy, 2017]. However, there is a 28.2% area penalty in this case.

The Area-Time product for the results in Tables 5 and 6 are calculated and listed in Tables 7 and 8. Although the proposed multipliers show improvement in speed compared to other similar works presented in the literature, one finds in some cases that the Area-Time product for the proposed multipliers is higher than its counterpart presented in the literature. This is due to the fact that in these cases, the proposed multiplier achieves better speed with a considerable extra area penalty.

4.3 Power comparison

The proposed 8×8 multiplier is implemented in Spartan-3E FPGA and compared, in terms of power, with other 8×8 multipliers from literature. In order to get the power analysis from the tool, the proposed 8×8 multiplier is implemented such that the inputs and the outputs are registered. The inputs are registered with 8-bit parallel in parallel out (PIPO) registers. The output is also registered with a PIPO register. However, the register at the output is 16-bit input and 8-bit output. This means that to get the final product out, two cycles are needed. The power results of the proposed multiplier and for other works from literature are listed in Table 9 where (CBWT) refers to Counter-Based Wallace Tree and (RCWT) refers to Reduced Complexity Wallace Tree. The results of other multipliers are taken from [Solanki et al., 2021].

The proposed 8×8 multiplier outperforms all other multipliers in terms of power with an acceptable increase in clock cycle delay in some cases. For example, there is 25.39% reduction in power compared to [Solanki et al., 2021] with only 14.06% increase in clock cycle delay. It can be figured out that although the proposed 8×8 multiplier shows a small improvement in power compared to [Wallace, 1964], it shows 20.65% reduction in area. In fact, the proposed 8×8 multiplier outperforms all other multipliers, listed in Table 9, in terms of area.

4.4 Utilizing the proposed multiplier in a MAC Unit

The Multiply-Accumulate (MAC) unit is a basic and essential digital component in most microprocessors and Digital Signal Processors (DSPs). The MAC unit is used to efficiently accelerate the computations of FIR or FFT/IFFT, which are required by data-intensive applications such as filters, orthogonal frequency-division multiplexing algorithms, and channel estimators [Hoang et al., 2010]. As shown by Figure 4, the main

| Operand size | Work | FPGA | A×T (Slices.ns) | A×T (LUTs.ns) |
|-------------------|-----------------|------------------|-----------------|---------------|
| 4 | W_a | Virtex-4 | NA | 256 |
| | <i>Proposed</i> | | 325 | 635 |
| | W_a | Spartan-3 | NA | 463 |
| | <i>Proposed</i> | | 562 | 1111 |
| | W_b (WT) | Spartan-3E | 408 | 723 |
| | W_b (Dadda) | | 411 | 697 |
| | <i>Proposed</i> | | 191 | 344 |
| | W_a | Spartan-6 | NA | 250 |
| | <i>Proposed</i> | | 120 | 334 |
| | W_c (WT) | Artix-7 | NA | 289 |
| | W_c (IWT) | | NA | 186 |
| | <i>Proposed</i> | | NA | 118 |
| | 8 | W_d (Radix-32) | Virtex-4 | 3904 |
| W_a | | NA | | 2122 |
| <i>Proposed</i> | | 931 | | 1678 |
| W_a | | Spartan-3 | NA | 4396 |
| W_e | | | 3010 | 5504 |
| W_f (WT) | | | 2774 | 5177 |
| <i>Proposed</i> | | | 1552 | 2835 |
| W_g | | Spartan-3E | NA | 23931 |
| W_b (WT) | | | 4017 | 7099 |
| W_b (Dadda) | | | 3813 | 6704 |
| <i>Proposed</i> | | | 1245 | 2108 |
| W_a | | Spartan-6 | NA | 2308 |
| W_h (Booth) | | | NA | 5586 |
| W_h (Array) | | | NA | 3004 |
| W_h (WT) | | | NA | 2389 |
| W_h (Vedic) | | | NA | 1820 |
| W_i (LUT-Based) | | | NA | 1051 |
| W_i (DSP-Based) | | | NA | 2319 |
| <i>Proposed</i> | | | 639 | 1367 |
| W_c (WT) | | Artix-7 | NA | 5433 |
| W_c (IWT) | | | NA | 1448 |
| <i>Proposed</i> | | | NA | 904 |

Table 7: Comparisons of Area-Time product with different works presented in literature for operand sizes of 4 and 8.

components of a MAC unit are Multiplier, Adder, and Accumulator. The Efficiency of the multiplier highly affects the overall performance of the MAC. The proposed multiplier is utilized in designing an 8-bit MAC unit to demonstrate its efficiency. Once the MAC finishes computation, the result is output in 2 cycles (8 bits/cycle). A VHDL code description for the MAC unit is developed and the design is implemented target-

| Operand size | Work | FPGA | A×T (Slices.ns) | A×T (LUTs.ns) |
|-----------------|-------------------|------------------|-----------------|---------------|
| 16 | W_d (Radix-32) | Virtex-4 | 14774 | NA |
| | <i>Proposed</i> | | 6015 | 10614 |
| | W_f (WT) | Spartan-3 | 22684 | 41843 |
| | <i>Proposed</i> | | 12001 | 21621 |
| | W_g | Spartan-3E | NA | 155447 |
| | <i>Proposed</i> | | 8382 | 12851 |
| | W_i (LUT-Based) | Spartan-6 | NA | 6886 |
| | W_i (DSP-Based) | | NA | 2381 |
| | <i>Proposed</i> | | 5575 | 11616 |
| 32 | W_d (Radix-32) | Virtex-4 | 67080 | NA |
| | <i>Proposed</i> | | 37994 | 63459 |
| | W_f (WT) | Spartan-3 | 178078 | 331749 |
| | <i>Proposed</i> | | 65945 | 119886 |
| | W_i (LUT-Based) | Spartan-6 | NA | 39539 |
| | W_i (DSP-Based) | | NA | 26988 |
| | <i>Proposed</i> | | 25124 | 67169 |
| 64 | W_d (Radix-32) | Virtex-4 | 404385 | NA |
| | W_e (Array) | | 7098580 | 13898143 |
| | W_e (Booth) | | 1119410 | 2238544 |
| | W_e (Vedic) | | 54450 | 68594 |
| | <i>Proposed</i> | | 206730 | 336640 |
| | W_j | Virtex-5 | NA | 173874 |
| | W_k | | NA | 57645 |
| | <i>Proposed</i> | | 73943 | 192543 |
| | W_k | Virtex-6 | NA | 41349 |
| | W_j | | NA | 104089 |
| | <i>Proposed</i> | | 72142 | 236453 |
| | W_k | Spartan-6 | NA | 65372 |
| | <i>Proposed</i> | | 147968 | 436535 |
| | 128 | W_d (Radix-32) | Virtex-4 | 1916961 |
| <i>Proposed</i> | | 1432735 | | 2050893 |

Table 8: Comparisons of Area-Time product with different works presented in the literature for operand sizes of 16, 32, 64, and 128.

ing Spartan-3E FPGA. The implementation results are listed in Table 10 and compared against the results of different MAC units, which are based on different multiplier designs from the literature. The implementation results show that the MAC unit of this work outperforms the different MAC units in terms of area and power, and power-delay product. For example, the MAC with the proposed multiplier achieves 18.3% reduction in power, 32.2% reduction in area, and 36.3% improvement in power-delay product compared to the MAC in [Solanki et al., 2021]. Hence, the proposed multiplier improves the efficiency of the MAC unit.

| Multiplier | Power (mW) | Dynamic power (mW) | Mini. period (ns) | Maxi. Freq. (MHz) | Area (LUTs) | Power delay product (pJ) |
|-----------------|------------|--------------------|-------------------|-------------------|-------------|--------------------------|
| W_l (WT) | 70.93 | 37.06 | 10.14 | 98.57 | 184 | 719.23 |
| W_m (RCWT) | 117.23 | 35.80 | 11.87 | 84.23 | 184 | 1391.63 |
| W_n (CBWT) | 123.72 | 42.20 | 12.70 | 78.69 | 190 | 1571.24 |
| W_o | 88.67 | 7.59 | 9.57 | 104.42 | 164 | 848.57 |
| W_p | 88.35 | 7.28 | 9.47 | 105.52 | 162 | 836.67 |
| W_q | 88.16 | 7.09 | 11.83 | 84.48 | 169 | 1042.93 |
| W_r | 88.21 | 7.14 | 9.53 | 104.93 | 165 | 840.64 |
| W_g | 87.12 | 6.06 | 9.29 | 107.55 | 161 | 809.34 |
| <i>Proposed</i> | 65.00 | 3.10 | 10.81 | 92.52 | 146 | 702.52 |

Table 9: Comparison of power for 8×8 multipliers over Spartan-3E FPGA.

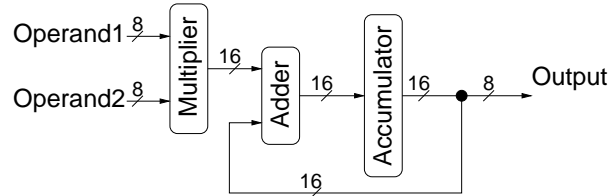


Figure 4: A block diagram for the MAC unit.

| Multiplier | Power (mW) | Dynamic power (mw) | Minimum period (ns) | Area (LUTs) | Power delay product (pJ) |
|-----------------|------------|--------------------|---------------------|-------------|--------------------------|
| W_l (WT) | 126.72 | 45.16 | 11.73 | 236 | 1486.42 |
| W_m (RCW) | 126.55 | 45.00 | 12.23 | 240 | 1547.7 |
| W_n (CBW) | 129.63 | 48.04 | 12.60 | 238 | 1633.33 |
| W_o | 93.91 | 12.77 | 14.82 | 242 | 1391.74 |
| W_p | 93.81 | 12.66 | 14.82 | 242 | 1390.26 |
| W_q | 90.97 | 9.86 | 14.83 | 230 | 1349.08 |
| W_r | 93.39 | 12.25 | 14.82 | 242 | 1384.03 |
| W_g | 87.44 | 6.38 | 14.82 | 242 | 1295.86 |
| <i>Proposed</i> | 71.00 | 38.00 | 11.62 | 164 | 825.02 |

Table 10: Comparison of 8-bit MAC using different multipliers over Spartan-3E FPGA.

5 Conclusions

Customized partial products generation (CPPG) cells are designed and used as building blocks for binary multipliers. Different size binary multipliers have been designed using these cells. All of the cells and the binary multipliers are described using VHDL. The

functionality of the cells and the binary multiplier is verified using Xilinx ISE 14.6 tools. Moreover, the cells and the multipliers are synthesized and implemented targeting different Xilinx FPGA families. Although larger size cells reduce the number of partial products, the hardware complexity of the cells increases in a way that negatively affects the performance of the multiplier. Over Virtex-7 FPGA family, the best area is achieved when using the 3×3 cell for all multipliers. While, the best delay for the 128×128 , the 64×64 , and the 32×32 multipliers is achieved when utilizing the 4×4 CPPG cell. For the 16×16 multiplier, the best delay is achieved when utilizing the 5×5 CPPG cell. The proposed multipliers have comparable performance results with those presented in the literature in terms of area, delay, and power. The efficiency of the proposed multiplier is verified by utilizing it in designing an 8-bit MAC unit, which shows better performance than similar MAC units in literature when implemented in Spartan-3E FPGA.

Acknowledgements

Osama Al-Khaleel acknowledges the support from AMD-XILINX University Program to Jordan University of Science and Technology (JUST).

References

- [Akhter and Chaturvedi, 2019] Akhter, S. and Chaturvedi, S. (2019). Modified binary multiplier circuit based on vedic mathematics. In 2019 6th International Conference on Signal Processing and Integrated Networks (SPIN), pages 234–237.
- [Al-Khaleel et al., 2011a] Al-Khaleel, O., Al-Khaleel, M., Al-Qudah, Z., Papachristou, C. A., Mhaidat, K., and Wolff, F. G. (2011a). Fast binary/decimal adder/subtractor with a novel correction-free bcd addition. In 2011 18th IEEE International Conference on Electronics, Circuits, and Systems, pages 455–459.
- [Al-Khaleel et al., 2015] Al-Khaleel, O., Al-Qudah, Z., Al-Khaleel, M., Bani-Hani, R., Papachristou, C., and Wolff, F. (2015). Efficient hardware implementations of binary-to-bcd conversion schemes for decimal multiplication. *Journal of Circuits, Systems and Computers*, 24(2):1–21.
- [Al-Khaleel et al., 2013] Al-Khaleel, O., Al-Qudah, Z., Al-Khaleel, M., and Papachristou, C. (2013). High performance fpga-based decimal-to-binary conversion schemes for decimal arithmetic. *Microprocessors and Microsystems*, 37(3):287–298.
- [Al-Khaleel et al., 2011b] Al-Khaleel, O., Al-Qudah, Z., Al-Khaleel, M., Papachristou, C. A., and Wolff, F. G. (2011b). Fast and compact binary-to-bcd conversion circuits for decimal multiplication. In 2011 IEEE 29th International Conference on Computer Design (ICCD), pages 226–231.
- [Al-Khaleel et al., 2006] Al-Khaleel, O., Papachristou, C., Wolff, F., and Pekmestzi, K. (2006). Fpga-based design of a large moduli multiplier for public-key cryptographic systems. In 2006 International Conference on Computer Design, pages 314–319.
- [Al-Nounou, 2022] Al-Nounou, A. A.-R. (2022). VHDL code for the CPPG cells. http://www.just.edu.jo/~oda/research/comp_arith/binary/multipliers/alnounou/CPPG_Cells/.
- [Asati and Chandrashekhara, 2009] Asati, A. and Chandrashekhara (2009). A high-speed, hierarchical 16×16 array of array multiplier design. 2009 International Multimedia, Signal Processing and Communication Technologies, IMPACT 2009, pages 161–164.
- [Asif and Kong, 2015a] Asif, S. and Kong, Y. (2015a). Analysis of different architectures of counter based wallace multipliers. In 2015 Tenth International Conference on Computer Engineering Systems (ICCES), pages 139–144.

- [Asif and Kong, 2015b] Asif, S. and Kong, Y. (2015b). Design of an algorithmic wallace multiplier using high speed counters. In 2015 Tenth International Conference on Computer Engineering Systems (ICCES), pages 133–138.
- [Atre and Alshewimy, 2017] Atre, S. G. M. E. and Alshewimy, M. A. M. (2017). Design and implementation of new delay-efficient/configurable multiplier using fpga. In 2017 12th International Conference on Computer Engineering and Systems (ICCES), pages 8–13.
- [Bais and Khan Ali, 2016] Bais, K. and Khan Ali, Z. (2016). Design of a high-speed wallace tree multiplier. *International Journal of Engineering Sciences & Research Technology*, 5(6):476–480.
- [Balasubramanian et al., 2021] Balasubramanian, P., Nayar, R., and Maskell, D. L. (2021). Approximate array multipliers. *Electronics*, 10(5):630.
- [Baugh and Wooley, 1973] Baugh, C. and Wooley, B. (1973). A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, C-22(12):1045–1047.
- [Beuchat and Tisserand, 2002] Beuchat, J.-L. and Tisserand, A. (2002). Small multiplier-based multiplication and division operators for virtex-ii devices. In *International Conference on Field Programmable Logic and Applications*, pages 513–522. Springer.
- [BN and HG, 2021] BN, M. K. and HG, R. (2021). Array multiplier and cia based fir filter for dsp applications. *International Research Journal on Advanced Science Hub*, 3:52–59.
- [Chandrashekara and Rohith, 2019] Chandrashekara, M. N. and Rohith, S. (2019). Design of 8 bit vedic multiplier using urdhva tiryagbhyam sutra with modified carry save adder. In 2019 4th International Conference on Recent Trends on Electronics, Information, Communication Technology (RTEICT), pages 116–120.
- [Chaudhary and Kularia, 2016] Chaudhary, I. and Kularia, D. (2016). Design of 64 bit high speed vedic multiplier. 5:4090–4096.
- [Christilda and Milton, 2021] Christilda, V. D. and Milton, A. (2021). Speed, power and area efficient 2d fir digital filter using vedic multiplier with predictor and reusable logic. *Analog Integrated Circuits and Signal Processing*, pages 1–11.
- [Das and Rahaman, 2010] Das, D. and Rahaman, H. (2010). A novel signed array multiplier. In 2010 International Conference on Advances in Computer Engineering, pages 19–23.
- [Fadavi-Ardekani, 1993] Fadavi-Ardekani, J. (1993). Mxn booth encoded multiplier generator using optimized wallace trees. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):120–125.
- [Fonseca et al., 2005] Fonseca, M., da Costa, E., Bampi, S., and Monteiro, J. (2005). Design of a radix-2m hybrid array multiplier using carry save adder format. In *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design, SBCCI '05*, page 172–177, New York, NY, USA. Association for Computing Machinery.
- [Fritz and Fam, 2017] Fritz, C. and Fam, A. T. (2017). Fast binary counters based on symmetric stacking. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2971–2975.
- [Hameed and Kathem, 2021] Hameed, A. S. and Kathem, M. J. (2021). Design and implementation of a fast sequential multiplier based on iterative addition architecture. *IOP Conference Series: Materials Science and Engineering*, 1076(1):012040.
- [Hasan and Kort, 2007] Hasan, O. and Kort, S. (2007). Automated formal synthesis of wallace tree multipliers. In 2007 50th Midwest Symposium on Circuits and Systems, pages 293–296. IEEE.

- [Hoang et al., 2010] Hoang, T. T., Sjalander, M., and Larsson-Edefors, P. (2010). A high-speed, energy-efficient two-cycle multiply-accumulate (mac) architecture and its application to a double-throughput mac unit. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(12):3073–3081.
- [Jais and Palsodkar, 2016] Jais, A. and Palsodkar, P. (2016). Design and implementation of 64 bit multiplier using vedic algorithm. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 0775–0779. IEEE.
- [Kamrani and Heikalabad, 2021] Kamrani, H. and Heikalabad, S. R. (2021). Design and implementation of multiplication algorithm in quantum-dot cellular automata with energy dissipation analysis. *The Journal of Supercomputing*, 77(6):5779–5805.
- [Karthik et al., 2021] Karthik, T. S., Manasa, K., and Raju, Y. S. (2021). Design and implementation of 64 bit high speed vedic multiplier. *INTERNATIONAL JOURNAL OF ADVANCE SCIENTIFIC RESEARCH AND ENGINEERING TRENDS*, 6:37–42.
- [Karthikeyan and Jagadeeswari, 2021] Karthikeyan, S. and Jagadeeswari, M. (2021). Performance improvement of elliptic curve cryptography system using low power, high speed 16 x 16 vedic multiplier based on reversible logic. *Journal of Ambient Intelligence and Humanized Computing*, 12(3):4161–4170.
- [Khaleqi Qaleh Jooq et al., 2021] Khaleqi Qaleh Jooq, M., Ahmadinejad, M., and Moaiyeri, M. H. (2021). Ultraefficient imprecise multipliers based on innovative 4: 2 approximate compressors. *International Journal of Circuit Theory and Applications*, 49(1):169–184.
- [Kim et al., 1998] Kim, T., Jao, W., and Tjiang, S. (1998). Arithmetic optimization using carry-save-adders. In *Proceedings of the 35th Annual Design Automation Conference, DAC '98*, page 433–438, New York, NY, USA. Association for Computing Machinery.
- [Langhammer and Pasca, 2021] Langhammer, M. and Pasca, B. (2021). Folded integer multiplication for fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 160–170.
- [Lee and Burgess, 2003] Lee, B. and Burgess, N. (2003). Improved small multiplier based multiplication, squaring and division. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 91–97. IEEE.
- [Mani et al., 2015] Mani, K. R., Tessly, T., Alphy, M., Anju, R., and Riboy, C. (2015). Fpga implementation of an efficient high speed wallace tree multiplier. In *International Conference on Emerging Trends in Technology and Applied Sciences (ICETTAS 2015)*, pages 9–14.
- [Mao et al., 2021] Mao, W., Li, K., Xie, X., Zhao, S., Li, H., and Yu, H. (2021). A reconfigurable multiple-precision floating-point dot product unit for high-performance computing. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1793–1798. IEEE.
- [Moss et al., 2019] Moss, D. J. M., Boland, D., and Leong, P. H. W. (2019). A two-speed, radix-4, serial-parallel multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):769–777.
- [Mounica et al., 2021] Mounica, Y., Kumar, K. N., Veeramachaneni, S., et al. (2021). Energy efficient signed and unsigned radix 16 booth multiplier design. *Computers & Electrical Engineering*, 90:106892.
- [Murugeswari and Mohideen, 2014] Murugeswari, S. and Mohideen, S. K. (2014). Design of area efficient and low power multipliers using multiplexer based full adder. In *Second International Conference on Current Trends In Engineering and Technology - ICCTET 2014*, pages 388–392.
- [Nagaraju and Reddy, 2014] Nagaraju, G. and Reddy, G. V. S. (2014). Design and implementation of 128 x 128 bit multiplier by ancient mathematics. *International Journal of Engineering Research & Technology (IJERT)*, 03:1363–1366.

- [Naqvi, 2017] Naqvi, S. Z. H. (2017). Design and simulation of enhanced 64-bit vedic multiplier. In 2017 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT), pages 1–4.
- [Rafiee et al., 2021] Rafiee, M., Pesaran, F., Sadeghi, A., and Shiri, N. (2021). An efficient multiplier by pass transistor logic partial product and a modified hybrid full adder for image processing applications. *Microelectronics Journal*, page 105287.
- [Rafiq et al., 2021] Rafiq, A., Chaudhry, S. M., Awan, K. S., and Usman, M. (2021). An efficient architecture of modified booth multiplier using hybrid adder. In 2021 International Bhurban Conference on Applied Sciences and Technologies (IBCAST), pages 648–656.
- [Saha et al., 2018] Saha, A., Pal, R., Naik, A. G., and Pal, D. (2018). Novel cmos multi-bit counter for speed-power optimization in multiplier design. *AEU - International Journal of Electronics and Communications*, 95:189–198.
- [Sakthimohan and Deny, 2021] Sakthimohan, M. and Deny, J. (2021). An efficient design of 8x8 wallace tree multiplier using 2 and 3-bit adders. In *Proceedings of International Conference on Sustainable Expert Systems: ICSES 2020*, volume 176, page 23. Springer Nature.
- [Seferlis et al., 2021] Seferlis, P., Varbanov, P. S., Papadopoulos, A. I., Chin, H. H., and Klemeš, J. J. (2021). Sustainable design, integration, and operation for energy high-performance process systems. *Energy*, page 120158.
- [Sharma, 2015] Sharma, A. (2015). Fpga implementation of a high speed multiplier employing carry lookahead adders in reduction phase. *International Journal of Computer Applications*, 116(17):27–31.
- [Singh and Singh, 2016a] Singh, N. and Singh, M. (2016a). Design and implementation of 16 x 16 high speed vedic multiplier using brent kung adder. *International Journal of Science and Research (IJSR)*, 5:239–242.
- [Singh and Singh, 2016b] Singh, N. and Singh, M. (2016b). Performance evaluation of 8-bit vedic multiplier with brent kung adder. *International Journal of Current Engineering and Technology*, 6(6):2086–2090.
- [Singh et al., 2021] Singh, N., Verma, G., and Khare, V. (2021). Power estimation and validation of embedded multiplier based on ann and regression technique. *Journal of Circuits, Systems and Computers*, page 2250086.
- [Solanki et al., 2021] Solanki, V., Darji, A. D., and Singapuri, H. (2021). Design of low-power wallace tree multiplier architecture using modular approach. *Circuits, Systems, and Signal Processing*, 40(9):4407–4427.
- [Van Toan and Lee, 2020] Van Toan, N. and Lee, J.-G. (2020). Fpga-based multi-level approximate multipliers for high-performance error-resilient applications. *IEEE Access*, 8:25481–25497.
- [Véstias, 2021] Véstias, M. P. (2021). Field-programmable gate array. In *Encyclopedia of Information Science and Technology*, Fifth Edition, pages 257–270. IGI Global.
- [Wallace, 1964] Wallace, C. S. (1964). A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17.
- [Waters and Swartzlander, 2010] Waters, R. S. and Swartzlander, E. E. (2010). A reduced complexity wallace multiplier reduction. *IEEE Transactions on Computers*, 59(8):1134–1137.