Salim Ullah, Semeen Rehman, Muhammad Shafique, Akash Kumar

High-Performance Accurate and Approximate Multipliers for FPGA-based Hardware Accelerators

Erstveröffentlichung in / First published in:

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2022, 41 (2), S. 211-224. IEEE. ISSN 1937-4151.

DOI: https://doi.org/10.1109/TCAD.2021.3056337

Final edited form was published in "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems". 41 (2), S. 211 - 224. ISSN: 1937-4151.
https://doi.org/10.1109/TCAD.2021.3056337

High-Performance Accurate and Approximate Multipliers for FPGA-based Hardware Accelerators

Salim Ullah, Semeen Rehman, Muhammad Shafique and Akash Kumar

***Abstract*—Multiplication is one of the widely used arithmetic operations in a variety of applications, such as image/video processing and machine learning. FPGA vendors provide high-performance multipliers in the form of DSP blocks. These multipliers are not only limited in number and have fixed locations on FPGAs but can also create additional routing delays and may prove inefficient for smaller bit-width multiplications. Therefore, FPGA vendors additionally provide optimized soft IP cores for multiplication. However, in this work, we advocate that these soft multiplier IP cores for FPGAs still need better designs to provide high-performance and resource efficiency. Towards this, we present generic area-optimized, low-latency accurate and approximate softcore multiplier architectures, which exploit the underlying architectural features of FPGAs, i.e., look-up table (LUT) structures and fast carry chains to reduce the overall critical path delay and resource utilization of multipliers. Compared to Xilinx multiplier LogiCORE IP, our proposed unsigned and signed accurate architecture provides up to 25% and 53% reduction in LUT utilization, respectively, for different sizes of multipliers. Moreover, with our unsigned approximate multiplier architectures, a reduction of up to 51% in the critical path delay can be achieved with an insignificant loss in output accuracy when compared with the LogiCORE IP. For illustration, we have deployed the proposed multiplier architecture in accelerators used in image and video applications, and evaluated them for area and performance gains. Our library of accurate and approximate multipliers is open-source and available online at https://cfaed.tu-dresden.de/pd-downloads to fuel further research and development in this area, facilitate reproducible research, and thereby enabling a new research direction for the FPGA community.**

***Index Terms*—Approximate Computing, Multipliers, High-Performance, Reduced-Area, Accelerators, Neural Networks**

I. Introduction

Multiplication is one of the basic arithmetic operations, used extensively in the domain of digital signal and image processing. FPGA vendors, such as Xilinx and Intel, provide DSP blocks to achieve fast multipliers [1]. Despite the high performance offered by these DSP blocks, their usage might not be efficient in terms of overall performance and area requirements for some applications. Table I compares the critical path delays (CPDs) and lookup tables (LUTs) utilization of two different implementations of Reed-Solomon and JPEG encoders* for Virtex-7 series FPGA using Xilinx Vivado. The routing delay caused by the location of the allocated DSP blocks has resulted in higher latency for DSP-based implementation of Reed-Solomon encoder. For small applications, it may be possible to perform manual Floorplanning to optimize an application's overall performance. However, for complex applications with contending requirements for FPGA resources, it may not be possible to optimize the placement of required FPGA resources manually to enhance performance gains. Similarly, the implementation of the JPEG-encoder shows a large number of DSP blocks (56% of the total available DSP blocks) utilization. Such applications can exhaust the available DSP blocks, making them less available/unavailable for performance-critical operations of other applications executing concurrently on the same FPGA, and thereby necessitating the LUT-based multipliers. Similar results about the DSP blocks utilization and overall application performance are also reported by [41]. Therefore, the orthogonal approach of having logic-based soft multipliers along with DSP blocks is important for obtaining overall performance gains in different implementation scenarios. That is why Xilinx and Intel also provide logic-based soft multipliers [2], [3].

Techniques like [4]–[7] present modular approach of designing bigger FPGA-based multipliers using smaller blocks. However, such techniques prove to be useful for relatively smaller bit-width multipliers; and for the relatively higher bit-width multipliers, they consume more FPGA resources. For example, the logic-based implementation of an accurate 8×8 multiplier on Virtex-7 FPGA using Vivado, with default synthesis options, consumes 71 LUTs. Whereas, the modular implementation of an accurate 8×8 multiplier using accurate 4×4 multipliers consumes 82 LUTs.

Walters [8] and Kumm et al. [9] have used the modified Booth's algorithm for area efficient radix-4 multiplier implementations using 6-input LUTs and associated carry chains of Xilinx FPGAs. Their implementations avoid partial products compressor trees and have large critical path delays. Parandeh-Afshar et al. have also used the Booth's and Baugh-Wooley's multiplication algorithms for area-efficient multiplier implementation using Altera FPGAs [10]. However, to reduce the effective length of carry chains, their implementation limits the length of the adaptive logic modules (ALM) to five, which results in the underutilization of the FPGA resources. Moreover, this feature of limiting the carry chain to five ALMs cannot be achieved without wasting resources, with current FPGAs from other vendors, such as those provided by Xilinx [11]. Parandeh-Afshar et al. have also proposed a partial products compressor tree using Altera FPGAs [12]. However, their proposed implementation of generalized parallel counters (GPCs), underutilizes LUTs in two consecutive ALMs.

Among other available multiplier design options, the conventional shift-and-add [13], serial and serial/parallel multi-

TABLE I: Comparison of logic vs DSP blocks based implementations for Reed-Solomon Decoder and JPEG Encoder

Design	DSP Blocks Enabled			DSP Blocks Disabled		
	CPD [ns]	LUTs	DSP Blocks	CPD [ns]	LUTs	DSP Blocks
Reed-Solomon Dec.	4.68	2797	22	4.47	2839	0
JPEG Enc.	8.85	14780	631	9.88	71362	0

S. Ullah and A. Kumar are with the Technische Universität Dresden, Germany. E-mail: (salim.ullah, akash.kumar)@tu-dresden.de

S. Rehman and M. Shafique are with the Technische Universität Wien, Austria. E-mail: (semeen.rehman, muhammad.shafique)@tuwien.ac.at

*Source codes from *http://opencores.org/projects*

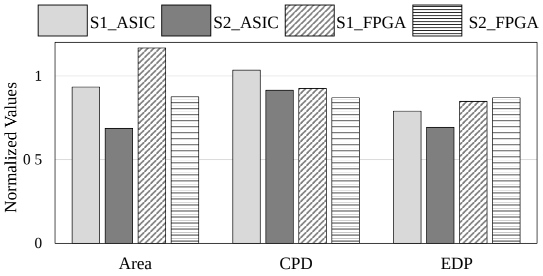


Fig. 1: Cross-platform comparison of area, latency and EDP. Results are normalized to the corresponding results of ASIC- and FPGA-based accurate multipliers.

pliers address the low area requirements but offer very high critical path delays. The commonly used Wallace [14] and Dadda [15] design-based parallel multipliers have high area requirements for achieving low output latencies by parallel addition of partial products. Considering the characteristics of the Booth and Wallace/Dadda multiplier schemes, a fast hybrid multiplier architecture using Radix-4 recoding has been proposed in [16] for ASIC-based systems. In this paper, we show that the performance of these logic-based soft multipliers can be further improved by utilizing efficient techniques for partial products encoding and their reduction. For example, compared to the multiplier implementation proposed in [9], our proposed accurate multiplier implementation offers a reduction of up to 52% in the critical path delay.

As has been demonstrated by a large body of works like [17], [38], [39], a wide range of applications do not require accurate intermediate computations and their operations can be approximated to further improve performance and energy efficiency. These applications have inherent resilience to approximation induced errors and thereby demonstrate the ability to produce viable outputs despite some of the input-data/intermediate computation being incorrect or approximate. Examples of such applications can be found in the domains of image/signal processing, machine learning and various other probabilistic algorithms. For the area-optimized and performance-efficient hardware acceleration of such applications, both accurate and approximate multipliers can be utilized. Using the principles of *approximate computing*, works in [18]–[21], [22]–[28] and [32] suggest the use of functional approximations for designing different types of approximate adders and multipliers with different performance gains.

However, most of the state-of-the-art accurate and approximate multiplier architectures consider only ASIC-based systems. Due to the inherent architectural differences between FPGAs and ASICs, these ASIC-based multiplier designs provide limited or no performance gains when directly synthesized for the FPGA-based systems. To further emphasize the need for designing FPGA-based approximate modules, we present the following motivational case study comparing the efficiency of both ASIC-based and FPGA-based implementations of state-of-the-art approximate multipliers that have been originally designed for ASIC-based systems.

A. Motivational Case Study

Fig. 1 compares the ASIC-based area, critical path delay (CPD), and Energy-Delay-Product (EDP) of two state-of-the-art approximate multipliers, *"S1"*, presented in [25], and *"S2"*, described in [23], with their FPGA-based implementations. The ASIC-based implementation results have been obtained from the corresponding papers ([25] and [23]), whereas for the FPGA-based implementations, the Xilinx Vivado tool for the Virtex-7 family has been used. Further, to evaluate the efficacy of the approximate designs, we have normalized these results to the implementation results of corresponding ASIC-based and FPGA-based accurate multipliers, respectively. As shown by the analysis results, the gains offered by the ASIC-based implementation are not proportionally translated to the corresponding FPGA-based implementation. For example, the area and EDP gains offered by $S1$ and $S2$ are reduced for the corresponding FPGA-based implementations – in fact, approximate implementation of $S1$ consumes more FPGA resources than the corresponding accurate design. However, the CPD is further reduced for both FPGA-based implementations. This lack of similar performance gains for the FPGA-based systems is the result of the architectural differences between ASICs and FPGAs. In ASIC-based designs, logic gates are deployed for the implementations of different logic circuits; thus, a full control over resource utilization at a fine granularity is possible. However, FPGA-based computational blocks are composed of entirely different entities, i.e., look-up tables (LUTs), where configuration bits are used to implement an individual circuit. This poses a **research challenge** of defining LUTs-based optimizations to implement accurate and approximate multipliers with significant performance gains, which can help in the realization of efficient FPGA-based hardware accelerators for error-resilient applications.

B. Our Novel Contributions

To address the above research challenge, we extend our prior work in [33] and present our methodology of defining LUT-level optimization for implementing accurate and approximate multipliers for FPGA-based systems. The overall contributions of this article are as follows:

- *An Accurate Unsigned Multiplier Design:* Utilizing the 6-input LUTs and associated fast carry chains of the state-of-the-art FPGAs, we present a scalable, area-optimized and reduced latency architecture of accurate multiplier.
- *Single Step Partial Products Generation and their Addition*: The proposed implementation of accurate multiplier fuses the generation and mutual addition of two consecutive partial product rows into one stage. For an $\text{N}_{\text{Multiplicand}}\times\text{M}_{\text{Multiplier}}$ multiplier, it results in the concurrent generation of (N+2)-bits long $\left\lceil \frac{M}{2} \right\rceil$ processed partial products (PPP).
- *Efficient Partial Product Reduction Tree*: For an N×M multiplier, *our automated tool flow* organizes the PPPs in $\left\lceil \frac{M}{6} \right\rceil$ groups. Each group can contain a maximum of three PPPs. Using 6-input LUTs and the associated carry chains, our methodology then deploys either ternary or binary adders for the mutual addition of PPPs in each group. The total number of stages required to find final accurate product is defined by Eq. 1.

$$No.\ of\ stages\ =\ \left\lceil log_3(\frac{M}{2}) \right\rceil + 1 \tag{1}$$

- *Accurate Signed Multiplier:* Utilizing the Baugh Wooley multiplication algorithm [37], we extend the *"Single Step Partial Products Generation and their Addition"* technique to compute $\left\lceil \frac{M}{2} \right\rceil$ PPPs for an N×M signed multiplier. Our methodology adds the generated PPPs to compute the final product by employing our proposed partial product reduction method.

- *An Approximate 4×2 Unsigned Multiplier* as a building block to implement higher-order approximate multipliers. The proposed approximate 4×2 multiplier completely utilizes the six inputs of a LUT of the state-of-the-art FPGAs.
- *An Approximate 4×4 Unsigned Multiplier*: To reduce the number of output errors, we perform different FPGA-specific optimizations on the approximate 4×2 multiplier and generate an approximate and asymmetric 4×4 multiplier.
- *An Approximate Ternary Adder for Summation of the Generated Approximate Partial Products*

Fig. 2 presents an overview of our proposed methodology in achieving these contributions. Using the primary logic resources of FPGAs (related preliminaries discussed in Section II), we present the accurate multiplier design in Section III. To reduce the multiplier critical path delay, we then analyze the utilized 3:1 compressors (ternary adders) and use various *LUT-level optimization* techniques to design different approximate multipliers in Sections V and VI. Finally, a thorough analysis of the output accuracy and performance gains of the proposed multipliers compared to the state-of-the-art multipliers is presented in Section VII. Our accurate and approximate architectures provide up to a 25% reduction in the total utilized LUTs when compared with the area-optimized Xilinx LogiCORE IP [2] for different sizes of multipliers. Moreover, the proposed approximate architecture achieves a reduction of up to 51% in the multiplier critical path delay when compared with the area-optimized LogiCORE IP. For the error characterization of our proposed multipliers, we have used the following quality metrics: (a) the number of error occurrences, (b) maximum error magnitude, (c) average relative error, and (d) number of maximum error occurrences. These metrics are commonly used by the literature for the quality analysis of approximate arithmetic circuits [22], [23], [29].

The RTL and behavioral models of these accurate and approximate multipliers are open-source and available online at *https://cfaed.tu-dresden.de/pd-downloads*. This will not only facilitate reproducing the results, but will also enable further research and development at higher abstraction layers.

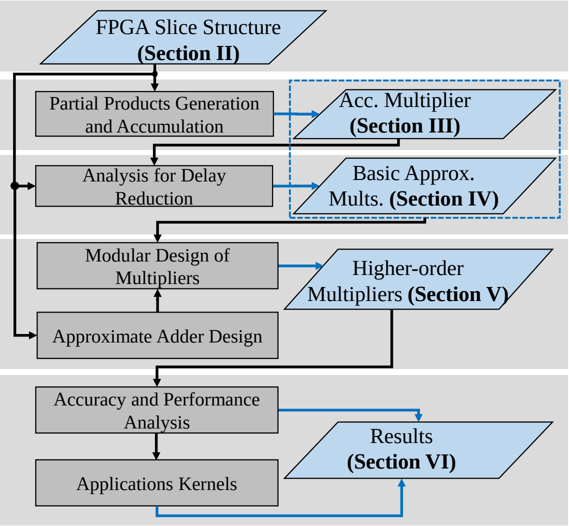

FPGA Slice Structure
(Section II)
Partial Products Generation and Accumulation
Acc. Multiplier (Section III)
Analysis for Delay Reduction
Basic Approx. Mults. (Section IV)
Modular Design of Multipliers
Higher-order Multipliers (Section V)
Approximate Adder Design
Accuracy and Performance Analysis
Results (Section VI)
Applications Kernels

Fig. 2: Summary of the proposed methodology

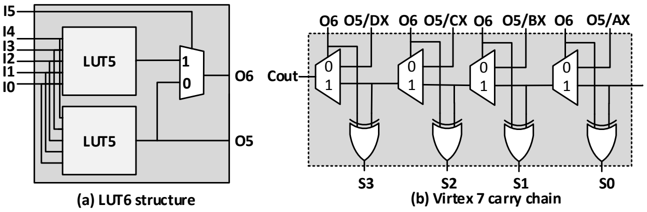


Fig. 3: Xilinx FPGA slice structure [11]

II. Preliminaries

A. Xilinx FPGA Slice Structure

State-of-the-art FPGAs, such as those provided by Xilinx and Intel, utilize 6-input LUTs to implement combinational and sequential circuits. In this manuscript, we have used Xilinx FPGAs for the implementation of all designs. However, our proposed methodology is generic and can be implemented on FPGAs from other vendors, such as Intel, which also uses fracturable 6-input LUTs and carry chains.

A slice in the configurable logic block (CLB) of Xilinx's 7-series FPGAs have four 6-input LUTs (commonly referred as *LUT6_2*) along with eight flip-flops for registering LUTs outputs and a single 4-bit long carry chain [11]. A *LUT6_2* can be used to implement either a single 6-bit combinational function, using O6 output bit, or two 5-bit combinational functions, using O5 and O6 output bits, by defining an INIT value which describes all the possible input combinations for which a logic value "1" is required at the output. For example, an INIT value of *0000000000000002(hex)* for *LUT6_2* defines to produce outputs *O5 = 1 & O6 = 0* for input combination *100001*. Besides the implementation of combinational functions, these 6-input LUTs are also used for controlling the associated carry chain, as shown in Fig. 3(b). The carry chain implements a carry-lookahead adder using O5 as the carry-generate signal and O6 as the carry-propagate signal as described by Eq. 2 and Eq. 3. The carry-generate signals for the carry chain can also be provided by the external bypass signals AX – DX.

$$S_i = P_i \oplus C_i \tag{2}$$

$$C_{i+1} = G_i + P_i \cdot C_i \tag{3}$$

B. Baugh-Wooley's Multiplication Algorithm

Compared to unsigned multiplication, all the partial products in a signed multiplication must be properly sign-extended to compute the accurate product. Baugh-Wooley's multiplication algorithm [37] eliminates the need for computing and communicating sign-extension bits by encoding the sign information in the generated partial products. For an $N \times M$ signed multiplier, Eq. 4 describes the respective operands in 2's complement representation. Eq. 5 illustrates the generation of the signed partial products to compute the final product *'P'*. Baugh-Wooley's multiplication algorithm rewrites the negative partial product terms, as described in Eq. 6, to eliminate the need for explicit sign-extension bits. The $a_x\overline{b_y}$ term in the

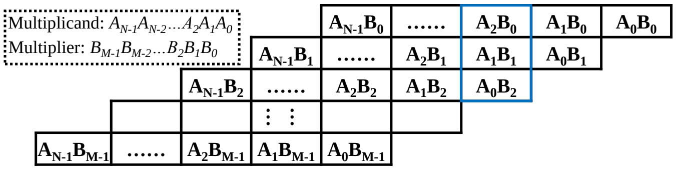


Fig. 4: N × M Basic multiplier design

equation, $for\ x \in [0\ ..\ N-1]\ and\ y \in [0\ ..\ M-1]$, de-notes the 1's complement of the corresponding partial product term. For example, for an 8×8 signed multiplier, Eq. 7 represents the signed partial products according to Baugh-Wooely's algorithm.

$$\begin{aligned} A &= -a_{N-1}2^{N-1} + \sum_{n=0}^{N-2} a_n 2^n \\ B &= -b_{M-1}2^{M-1} + \sum_{m=0}^{M-2} b_m 2^m \end{aligned} \tag{4}$$

$$\begin{aligned} P = {} & a_{N-1}b_{M-1}2^{N+M-2} + \sum_{n=0}^{N-2}\sum_{m=0}^{M-2} a_n b_m 2^{n+m} \\ & - 2^{N-1}\sum_{m=0}^{M-2} a_{N-1}b_m 2^m - 2^{M-1}\sum_{n=0}^{N-2} b_{M-1}a_n 2^n \end{aligned} \tag{5}$$

$$\begin{aligned} P = {} & a_{N-1}b_{M-1}2^{N+M-2} + \sum_{n=0}^{N-2}\sum_{m=0}^{M-2} a_n b_m 2^{n+m} \\ & + 2^{N-1}\sum_{m=0}^{M-2} \overline{a_{N-1}b_m} 2^m + 2^{M-1}\sum_{n=0}^{N-2} \overline{b_{M-1}a_n} 2^n \\ & + 2^{N-1} + 2^{M-1} + 2^{N+M-1} \end{aligned} \tag{6}$$

$$\begin{aligned} P_8 = {} & a_7b_7 2^{14} + \sum_{n=0}^{6}\sum_{m=0}^{6} a_n b_m 2^{n+m} + 2^7 \sum_{m=0}^{6} \overline{a_7b_m} 2^m \\ & + 2^7 \sum_{n=0}^{6} \overline{b_7a_n} 2^n + 2^8 + 2^{15} \end{aligned} \tag{7}$$

III. Generic Area-Optimized Low-Latency Unsigned Accurate Multiplier Architecture

The proposed implementation of accurate multiplier is based on the basic method of multiplying two multi-bit numbers $A_{(N\text{-bits})}$ and $B_{(M\text{-bits})}$, as shown in Fig. 4. The multiplication results in the generation of M, N-bit partial products with required shifting. Fig. 5 exhibits the elemental steps of our proposed implementation to realize an accurate multiplier. It includes the following operations:

1) *Organization of Partial Products (PPs)*: We have used the 6-input LUTs for computing the required partial products by performing AND operation between every bit of multiplier and multiplicand. However, to enhance the utilization of LUTs, our *automated methodology* groups every two consecutive partial products. Each group contains the second partial product shifted left by a single bit position relative to the first partial product. Further, the partial products in every group are computed and mutually added in one single step. However, in every group, there are two partial product terms, for example, A_0B_0 and $A_{N-1}B_1$ in the first group, which are not added with any other partial product term in their respective group. Moreover, due to the limited number of input/output pins of LUTs in modern FPGAs, it is not possible to group more than two partial product terms. For example, the generation and addition of partial products A_2B_0, A_1B_1 and A_0B_2, as shown by the blue box in Fig. 4, cannot be performed in a single step.
2) *LUTs and Carry Chain Assignment*: In this step, our methodology assigns the 6-input LUTs and the associated carry chains to each group of partial products, as shown by the computational blocks Type-A and Type-B in Fig. 6. An instance of a block, either Type-A or Type-B, denotes a 6-input LUT with an associated adder and carry chain cell (CC). Fig. 7(a) shows the functionality of the LUT of block Type-A. The output signals O5 and O6 are passed to the corresponding carry chain as carry generate ($\mathrm{G_i}$) and carry propagate ($\mathrm{P_i}$) signals respectively. The LUT configuration for block Type-B, in Fig. 7(b), uses O5 for the computation of the least significant partial product term in each row. The generate signal for the carry chain element corresponding to block type-B is constant '0' and provided by the external bypass signal (AX–DX), as already described in Fig. 3. The associated carry chain uses Eq. 2 and Eq. 3 for the generation of sum bit (S_i) and carry out (C_{i+1}) bits. The completion of this stage of our implementation results in the generation of (N+2)-bits long $\left\lceil \frac{M}{2} \right\rceil$ processed partial products (PPP).
3) *Re-arrangement and reduction of PPPs*: Our implementation utilizes ternary and binary adders for reducing PPPs to a final product. Modern FPGAs provide the capability of implementing a ternary adder as a ripple carry adder (a 3:1 compressor for the simultaneous reduction of 3 partial products). Our automated methodology arranges the PPPs in multiple groups; with the intention of placing three distinct PPPs in each group. Depending on the value of M, in Fig. 4, a group may have one, two or three PPPs. Our implementation then utilizes 3:1 and 2:1 compressors for reducing PPPs in each group.

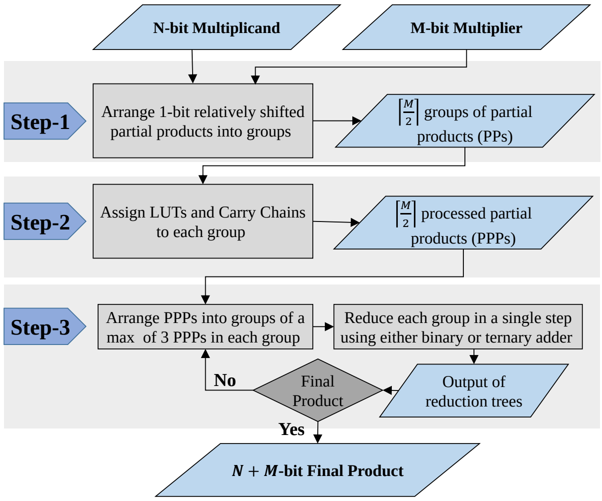

N-bit Multiplicand
M-bit Multiplier
Step-1
Arrange 1-bit relatively shifted partial products into groups
$\left\lceil \frac{M}{2} \right\rceil$ groups of partial products (PPs)
Step-2
Assign LUTs and Carry Chains to each group
$\left\lceil \frac{M}{2} \right\rceil$ processed partial products (PPPs)
Step-3
Arrange PPPs into groups of a max of 3 PPPs in each group
Reduce each group in a single step using either binary or ternary adder
No
Final Product
Output of reduction trees
Yes
N + M-bit Final Product

Fig. 5: Proposed design flow of accurate multipliers

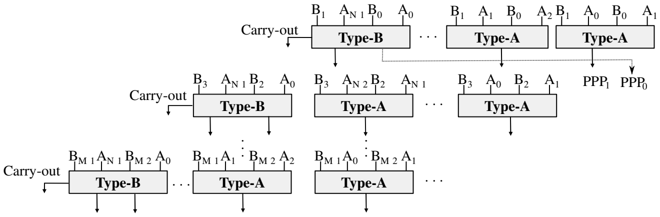
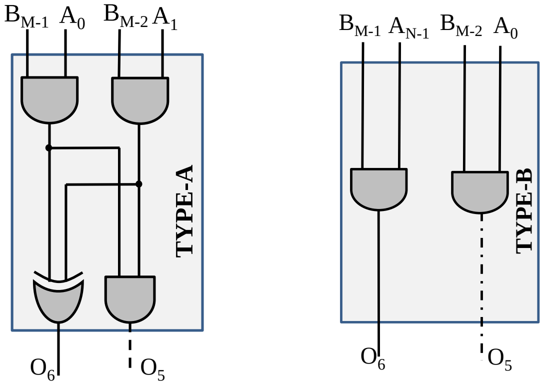


Fig. 6: Partial products generation for an N × M proposed accurate unsigned multiplier



(a) LUT of Type-A

(b) LUT of Type-B

Fig. 7: LUTs Configuration for accurate unsigned multiplier

The PPPs reduction phase may produce new partial sums, which are again grouped and passed through 3:1 and 2:1 compressors. This process is repeated until the final product is obtained. For example, for a 16 × 16 multiplier, 8 PPPs are generated. The grouping and reduction of these PPPs to compute the final product is described in Fig. 8

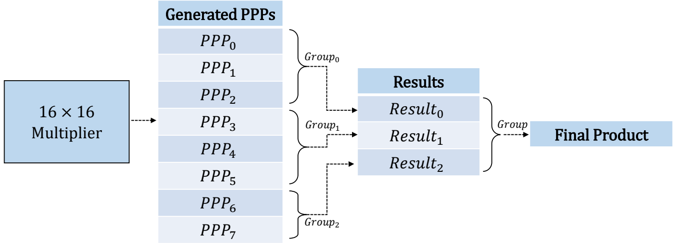


Fig. 8: Grouping of PPPs to compute final product for a 16×16 multiplier

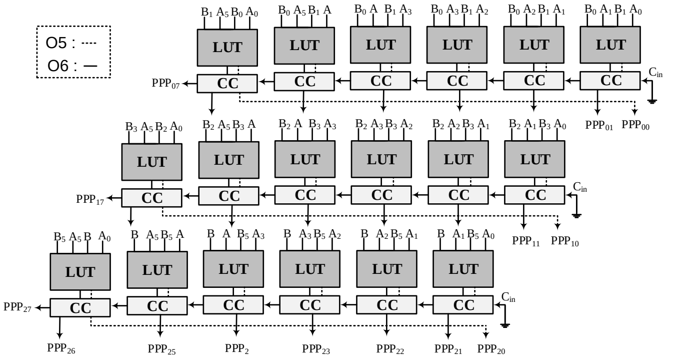


Fig. 9: Virtex 7 FPGA slice based representation of processed partial products generation for a 6 × 6 accurate unsigned multiplier

Fig. 9 shows the mapping of the proposed implementation for a 6×6 accurate multiplier on Xilinx 7-series FPGAs. However, the same implementation can also be ported to the newer versions of FPGAs, such as Virtex UltraScale+. As described previously, a computational block of Fig. 6 is equivalent to a LUT and the one bit cell (CC) of associated carry chain in Fig. 9. Table II defines the *Type-A* LUT configuration for the generation and summation of partial product bits $A_Y B_Y$ and $A_X B_X$ of Fig. 9. The LUT initially performs the logical *AND* operation on $(A_Y,\ B_Y)$ and $(A_X,\ B_X)$ and then produces the O5 (generate) and O6 (propagate) signals. The values *O5 = 0x8000* and *O6 = 0x7888* accommodates only four input values. As discussed in Section II, the INIT value for LUT6 to produce *O5 = 0x8000* and *O6 = 0x7888* will be *0x7888788880008000*. The INIT value for *Type-B* LUT configuration is *0xFFFFFFFF80008000* and its configuration is shown in Table III.

As shown in Fig. 9, three 8-bit long processed partial products have been generated in the first stage of multiplication. Our proposed automated methodology organizes these PPPs in a single group and utilizes ternary addition for computing the final product. The ternary adder in Fig. 10 shows the computation of final product bits P_1–P_4 by adding three partial products in one step. The carry out of the slice is forwarded to the carry chain in next slice for computing other product bits. Since the proposed implementation relies on the efficient utilization of the available LUTs and carry chain in a slice, therefore, the LUTs required by an $N \times M$ multiplier can be estimated even without synthesizing the design using Eq. 8.

$$No.\ of\ required\ LUTs < \left\lfloor \frac{M}{4} \right\rfloor \times (N+4) + \left\lceil \frac{M}{2} \right\rceil \times N \quad (8)$$

TABLE II: Type-A LUT configuration

A_Y	B_Y	A_X	B_X	A_XB_X	A_YB_Y	$A_XB_X + A_YB_Y$		O6 (Hex)	O5 (Hex)
						Sum (O6)	Carry (O5)		
0	0	0	0	0	0	0	0	8	0
0	0	0	1	0	0	0	0		
0	0	1	0	0	0	0	0		
0	0	1	1	1	0	1	0		
0	1	0	0	0	0	0	0	8	0
0	1	0	1	0	0	0	0		
0	1	1	0	0	0	0	0		
0	1	1	1	1	0	1	0		
1	0	0	0	0	0	0	0	8	0
1	0	0	1	0	0	0	0		
1	0	1	0	0	0	0	0		
1	0	1	1	1	0	1	0		
1	1	0	0	0	1	1	0	7	8
1	1	0	1	0	1	1	0		
1	1	1	0	0	1	1	0		
1	1	1	1	1	1	0	1		

B_{M-1}	A_{N-1}	B_{M-2}	A_0	A_0B_{M-2}	$A_{N-1}B_{M-1}$	Sum (O6)	Carry (O5)	O6 (
0	0	0	0	0	0	0	0	
0	0	0	1	0	0	0	0	
0	0	1	0	0	0	0	0	
0	0	1	1	1	0	0	1	
0	1	0	0	0	0	0	0	
0	1	0	1	0	0	0	0	
0	1	1	0	0	0	0		
0	1	1	1	1	0	0		
1	0	0	0	0	0	0		
1	0	0	1	0	0	0		
1	0	1	0	0	0	0		
1	0	1	1	1	0	0		
1	1	0	0	0	1	1		
1	1	0	1	0	1	1		
1	1	1	0	0	1	1		
1	1	1	1	1	1	1		

0 8

IV. Generic Area-Optimized Accurate Multiplier

Utilizing the proposed design fl and Baugh-Wooley's multiplicatic Eq. 6, we present our novel desigr plier. For an $N \times M$ signed multipli ates only $\left\lceil \frac{M}{2} \right\rceil$ signed PPPs. This f plementation is similar to the commonly used radix-4 Booth's multiplication algorithm, which halves the total number of generated partial products [42]. Further, Baugh-Wooley's algorithm eliminates the need for extra sign-extension bits, which help realize a resource-efficient implementation of the multiplier. Fig. 11 presents the graphical representation of Baugh-Wooley's algorithm. As shown, the last partial product row and the most significant term in all other partial product rows are complemented. To accommodate the generation of these complemented terms, we update our proposed design flow with three new LUTs configurations. Fig. 12 presents the new configurations of LUTs. Utilizing these configurations, Fig. 13 presents the *'LUTs and Carry Chain Assignment'* step of our proposed methodology for an $N \times M$ signed multiplier. After generating all signed PPPs, we utilize the *'Re-arrangement and reduction of PPPs'* step of our proposed

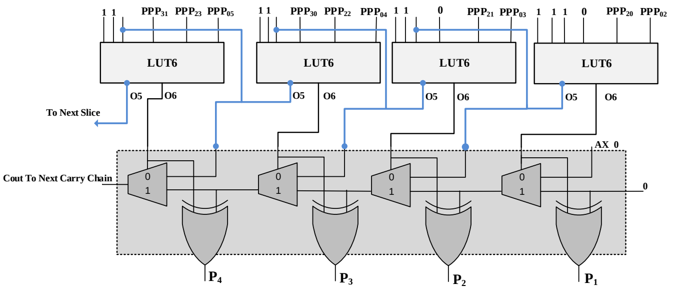


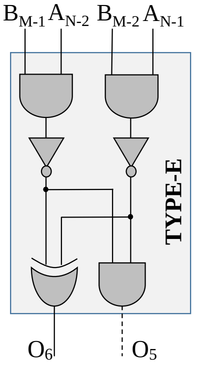
Fig. 10: Virtex 7 FPGA slice-based ternary adder: computation of final product bits P_1–P_4 for a 6×6 accurate multiplier

Multiplicand: $A_{N-1} \ldots A_1 A_0$
Multiplier: $B_{M-1} \ldots B_1 B_0$

				$\overline{A_{N-1}B_0}$	$A_{N-2}B_0$	...	$A_{M-1}B_0$	...	A_1B_0	A_0B_0
			$\overline{A_{N-1}B_1}$	$A_{N-2}B_1$	...	$A_{M-1}B_1$	...	A_1B_1	A_0B_1	
		$\overline{A_{N-1}B_2}$	$A_{N-2}B_2$	...	$A_{M-1}B_2$	...	A_1B_2	A_0B_2		
				⋮						
1	$A_{N-1}B_{M-1}$	$\overline{A_{N-2}B_{M-1}}$	...	$\overline{A_{M-1}B_{M-1}}$	...	$\overline{A_1B_{M-1}}$	$\overline{A_0B_{M-1}}$			
				1			1			

Fig. 11: Baugh-Wooley's N × M signed multiplier design

methodology to compute the final product. Further, the 1's at



(c) LUT of Type-E

Fig. 12: LUTs Configuration for accurate signed multiplier

V. Approximate Multipliers Architecture

The proposed accurate multiplier design utilizes ternary and binary adders for the addition of generated partial products, as shown in the previous section. The utilization of ternary adders enables resource-efficient implementations. However, the dependency of every element of the carry chain on the carry-generate signal from its preceding cell, as shown in Fig. 10, diminishes the performance of a ternary adder. For example, the implementation of an 8-bit ternary adder (three operands) on Virtex-7 FPGA using Vivado has a 37% higher critical path delay compared to an 8-bit binary adder (two operands). Kumm et al. have also reported similar observations about the reduced performance of ternary adders with different bit-widths [40]. Towards this end, we apply various approximation techniques to realize high-performance and resource-efficient approximate multipliers. In the following sections, we first present designs of elementary approximate multipliers followed by the architecture of an approximate adder for implementing higher-order approximate multipliers using sub-multipliers.

A Performance/Area Optimized Elementary Multiplier Module, targeted for FPGAs, should efficiently utilize the available *LUT6_2* structure and the associated carry chains in a given FPGA. The 2×2 multipliers, as used by [23] and [25], under-utilize *LUT6_2* and therefore has been excluded from the list of potential design options for the elementary multipliers. The only two potential multiplier designs, which utilize all the inputs of a *LUT6_2*, are 3×3 and 4×2 multipliers. However, a 3×3 multiplier is not a feasible option for the implementation of higher order multipliers, e.g. 4×4 and 8×8 multipliers. A 4×4 multiplier requires one 3×3, one 1×4 and

Final edited form was published in "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems". 41 (2), S. 211 - 224. ISSN: 1937-4151.
https://doi.org/10.1109/TCAD.2021.3056337

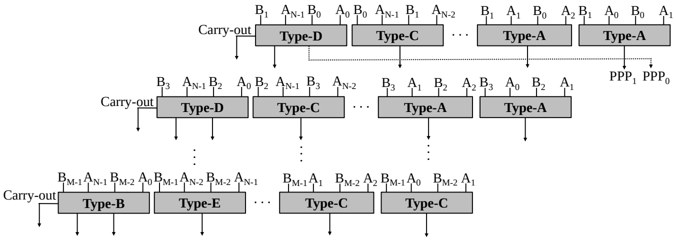

B_1 A_{N-1} B_0 A_0 B_0 A_{N-1} B_1 A_{N-2} B_1 A_1 B_0 A_2 B_1 A_0 B_0 A_1
Carry-out
Type-D
Type-C
. . .
Type-A
Type-A
PPP_1 PPP_0
B_3 A_{N-1} B_2 A_0 B_2 A_{N-1} B_3 A_{N-2} B_3 A_1 B_2 A_2 B_3 A_0 B_2 A_1
Carry-out
Type-D
Type-C
. . .
Type-A
Type-A
B_{M-1} A_{N-1} B_{M-2} A_0 B_{M-1} A_{N-2} B_{M-2} A_{N-1} B_{M-1} A_1 B_{M-2} A_2 B_{M-1} A_0 B_{M-2} A_1
Carry-out
Type-B
Type-E
. . .
Type-C
Type-C

Fig. 13: Partial products generation for an N × M proposed accurate signed multiplier

one 3×1 multipliers [4]. This limited applicability of a 3×3 multiplier results in filtering it out from our selection of an elementary multiplier module. The only feasible elementary design is a 4×2 multiplier, which thoroughly utilizes lookup tables of state-of-the-art FPGAs. A 4×4 multiplier can be implemented using two instances of a 4×2 multiplier and an adder. This paper uses 4×2 multiplier as the elementary block for designing higher order approximate multipliers.

A. Approximate Design of 4 × 2 Multiplier

An accurate 4×2 multiplier generates a 6-bit output with the following optimized logic equations for A($A_3A_2A_1A_0$) and B(B_1B_0) as multiplicand and multiplier, respectively:

$$P_0 = B_0A_0 \tag{9}$$

$$P_1 = B_1'B_0A_1 + B_1B_0'A_0 + B_1A_1'A_0 + B_0A_1A_0' \tag{10}$$

$$\begin{aligned} P_2 = {} & B_1'B_0A_2 + B_1B_0'A_1 + B_0A_2A_1' \\ & + B_1A_2'A_1A_0' + B_1A_2A_1A_0 \end{aligned} \tag{11}$$

$$\begin{aligned} P_3 = {} & B_1'B_0A_3 + B_1B_0'A_2 + B_1A_3'A_2A_1' \\ & + B_0A_3A_2'A_1' + B_1B_0A_3'A_2'A_1A_0 \\ & + B_0A_3A_2A_1 + B_0A_3A_1A_0' \end{aligned} \tag{12}$$

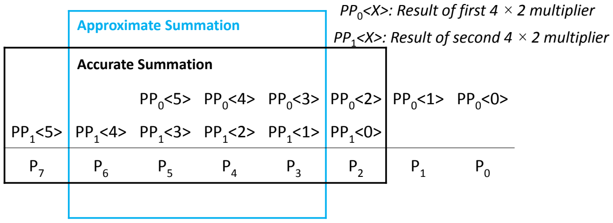
$$\begin{aligned} P_4 = {} & B_1B_0'A_3 + B_1A_3A_2'A_1' + B_1A_3A_2'A_0' \\ & + B_1B_0A_3'A_2A_1 \end{aligned} \tag{13}$$

$$P_5 = B_1B_0A_3A_2 + B_1B_0A_3A_1A_0 \tag{14}$$

As P_0, P_1, and P_2 each depends on less than six shared variables, i.e., A_0, A_1, A_2, B_0, and B_1, any two of these three least significant product bits can be generated using a single LUT6_2. The remaining four product bits will require four separate LUTs for implementation. An area and energy efficient approximation is to accommodate the six product bits in four LUTs i.e. a single slice. Truncation of P_0 limits the output error to the least significant product bit and the final output accuracy to 75% with maximum error magnitude of '1' for all input combinations. Approximation of any other product bit results in a higher magnitude of error in the final output. The proposed approximate design of 4×2 multiplier uses 4 LUTs for its implementation by truncating 'P_0' and generating 'P_1' and 'P_2' by a single LUT6_2.

B. Approximate Design of a 4 × 4 Multiplier

The approximate design of a 4×4 multiplier requires two 4×2 multipliers, consuming eight LUTs for generating partial products. For multiplicand A($A_3A_2A_1A_0$) and multiplier B($B_3B_2B_1B_0$), the first 4×2 multiplier takes A($A_3A_2A_1A_0$) & $B(B_1B_0)$ and the second 4×2 multiplier occupies A($A_3A_2A_1A_0$) & $B(B_3B_2)$ as input operands.


PP_0<X>: Result of first 4 × 2 multiplier
PP_1<X>: Result of second 4 × 2 multiplier
Approximate Summation
Accurate Summation

		PP_0<5>	PP_0<4>	PP_0<3>	PP_0<2>	PP_0<1>	PP_0<0>
PP_1<5>	PP_1<4>	PP_1<3>	PP_1<2>	PP_1<1>	PP_1<0>		
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Fig. 14: 4×4 using 4×2 multipliers

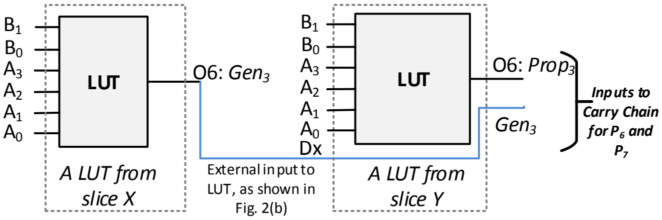


Fig. 15: Implementation of *Gen₃* and *Prop₃* for P_6 and P_7

As shown by the black box in Fig. 14, the accurate summation of the approximate partial products generated by the two 4×2 multipliers requires the use of two carry chains[†]. Therefore, the approximate 4×4 multiplier, with accurate summation of partial products, requires 16 LUTs[‡] (2 LUTs wasted by the second carry chain). Due to the truncation of PP_0<0> and PP_1<0> in Fig. 14, this 4×4 multiplier implementation has an average relative error of 0.049 with an error probability of 0.375 for a uniform input distribution. However, the proposed design performs approximate addition along with FPGA-specific optimizations of second 4×2 multiplier and uses one single carry chain for partial products summation, as shown by the blue rectangle in Fig. 14. Our optimizations not only provides area gains but also significantly improves the total number of error cases by having only 6 erroneous outputs. Our proposed optimization uses three LUT6_2s for the implementation of required *Carry Propagate* and *Carry Generate* signals to compute P_3, P_4 and P_5 product bits. Since PP_1<4> and PP_1<5> share same six operands, therefore our design does not compute PP_1<4> and PP_1<5> explicitly for subsequent addition by the carry chain. The proposed approach, as shown in Fig. 15, computes the respective *Carry Propagate 'Prop₃'* and *Carry Generate 'Gen₃'* signals for the computation of P_6 and P_7 directly from the multiplier and multiplicand bits by implicitly generating PP_1<4> and PP_1<5>. This implicit implementation of PP_1<4> and PP_1<5> saves one LUT as compared to their explicit computation. In order to improve the output accuracy, the recovered LUT is then assigned for the accurate realization of P_0 and P_2. Since the computation of P_3 is also dependent on the carry-out from P_2, therefore, the corresponding LUT for P_3 besides, using PP_0<3> and PP_1<1> also utilize A_0, B_2 and PP_0<2> to resolve the effect of the missing carry-out from P_2. As carry propagate and carry Generate signals cannot be '1' simultaneously, all the cases where A_0, B_2, PP_0<2>, PP_0<3> and PP_1<1> are all '1' concurrently, will generate an error. To limit the error occurrences to a single product bit, P_3, we propose to correctly compute the carry Generate signal

[†] A carry chain is 4-bit wide in Xilinx 7 series FPGAs.

[‡] Each position of a carry chain is controlled by a corresponding LUT6_2.

TABLE IV: 4×4 multiplier error values

Multiplier	Multiplicand	Accurate Product	Approximate Result	Difference
5	15	75	67	8
6	7	42	34	8
6	15	90	82	8
7	15	105	97	8
13	13	169	161	8
15	5	75	67	8

only. This decision limits the error to P_3 only with a fixed error magnitude of "8".

Tables IV and V present the input operands with erroneous outputs and INIT values employed by each LUT along with input/output pins configuration respectively. It is noteworthy that depending upon an application's input data, the proposed 4×4 multiplier may produce better result due to its asymmetric nature and the values presented in Table IV only show the maximum number of possible error occurrences for uniform distribution of all input cases. Our proposed multiplier does not generate erroneous outputs for highlighted inputs, in Table IV, with multiplier and multiplicand mutually swapped. For achieving better output quality results, the proposed approach suggests an initial analysis of input data, before multiplication, to decide operands for multiplier and multiplicand. The asymmetric nature of the proposed multiplier and the analysis of input data for achieving better output accuracy are further explored in Section VII.

VI. Designing Higher Order Approximate Multipliers

We have used the modular approach of implementing higher-order approximate multipliers using submultipliers. For example, as shown in Fig. 16, the results of four M × M submultipliers are added together to implement a 2M × 2M multiplier. The modular approach provides a broader design space by using various accurate/approximate submultipliers to implement a higher-order accurate/approximate multiplier. Furthermore, various accurate and approximate adders can be utilized to add together the results (referred to as sub-products: *SPs*) of submultipliers to obtain the final accurate/approximate product.

Utilizing the modular approach, we have used four instances of our proposed approximate 4×4 multiplier and the accurate ternary adder, shown in Fig. 10, to implement an approximate 8×8 multiplier, referred to as Ca. For example, the $SP<4>$ – $SP<7>$ from $A_L \times B_L$, $SP<0>$ – $SP<3>$ from $A_H \times B_L$ and $SP<0>$ – $SP<3>$ from $A_L \times B_H$ are added in one single step to produce final product bits P_4 – P_7 for an 8×8 multiplier. The $O5$ output of the fourth LUT6 and the Cout of the carry chain in Fig. 10 are routed to the next slice for generation of higher order product bits. The same process can be repeated for the implementation of arbitrary sizes of higher order multipliers.

TABLE V: LUTs' inputs and outputs pins configuration for approximate 4×4 Multiplier

LUT	LUT Input Pins Configuration						INIT value (Hex)	LUT Output Pins Configuration	
	I5	I4	I3	I2	I1	I0		O6	O5
LUT_0	1	B_1	B_0	A_2	A_1	A_0	*B4CCF00066AACC00*	$PP_0<2>$	$PP_0<1> = P_1$
LUT_1	B_1	B_0	A_3	A_2	A_1	A_0	*C738F0F0FF000000*	$PP_0<3>$	
LUT_2	B_1	B_0	A_3	A_2	A_1	A_0	*07C0FF0000000000*	$PP_0<4>$	
LUT_3	B_1	B_0	A_3	A_2	A_1	A_0	*F800000000000000*	$PP_0<5>$	
LUT_4	1	B_3	B_2	A_2	A_1	A_0	*B4CCF00066AACC00*	$PP_1<2>$	$PP_1<1>$
LUT_5	B_3	B_2	A_3	A_2	A_1	A_0	*C738F0F0FF000000*	$PP_1<3>$	
LUT_6	B_3	B_2	A_3	A_2	A_1	A_0	*F800000000000000*	Gen_3	
LUT_7	1	1	$PP_0<2>$	B_2	B_0	A_0	*5FA05FA088888888*	P_2	P_0
LUT_8	1	$PP_1<1>$	$PP_0<3>$	B_2	A_0	$PP_0<2>$	*007F7F80FF808000*	$Prop_0$	Gen_0
LUT_9	1	1	1	1	$PP_1<2>$	$PP_0<4>$	*6666666688888880*	$Prop_1$	Gen_1
LUT_{10}	1	1	1	1	$PP_1<3>$	$PP_0<5>$	*6666666688888880*	$Prop_2$	Gen_2
LUT_{11}	B_3	B_2	A_3	A_2	A_1	A_0	*07C0FF0000000000*	$Prop_3$	

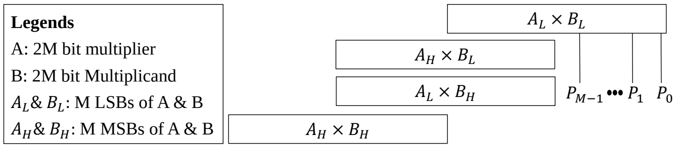


Fig. 16: Designing higher order multipliers from lower order multipliers

For the approximate addition of the results of accurate/approximate submultipliers to implement higher-order multipliers, we also present a novel approximate adder. Our proposed adder adds the results of three submultipliers simultaneously without using carry-out from the preceding bit locations. The gate-level diagram of our proposed adder is shown in Fig. 17. Utilizing the proposed adder and four instances of our proposed approximate 4x4 multiplier, Fig. 18 shows the implementation of an approximate 8x8 multiplier, referred to as Cc. Each blue box represents an instance of the proposed approximate adder implemented using a LUT6_2. As shown, the four least and most significant product bits are obtained without using addition. The proposed approximate addition further improves the performance of the realized higher-order multiplier by reducing its critical path and the number of LUTs employed during the summation of the results of submultipliers.

VII. Results and Discussion

A. Experimental Setup and Tool Flow

All presented multipliers have been implemented in VHDL and synthesized for the 7VX330T device of Virtex-7 family using Xilinx Vivado 17.4. For PDP and EDP calculations, Vivado Simulator and Power Analyzer tools have been used. Our methodology implements each design multiple times with a different critical path constraint in each iteration to

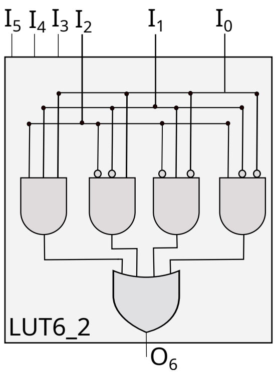


Fig. 17: Proposed approximate adder for implementation of higher-order multipliers: LUT6_2-based representation

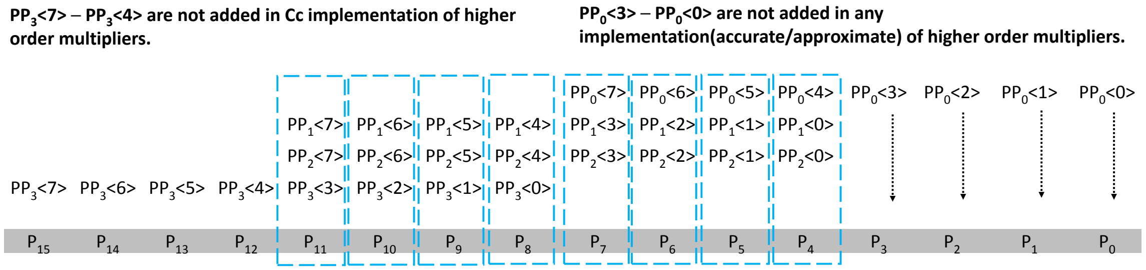

PP3<7> – PP3<4> are not added in Cc implementation of higher order multipliers.
PP0<3> – PP0<0> are not added in any implementation(accurate/approximate) of higher order multipliers.
PP0<7> PP0<6> PP0<5> PP0<4> PP0<3> PP0<2> PP0<1> PP0<0>
PP1<7> PP1<6> PP1<5> PP1<4> PP1<3> PP1<2> PP1<1> PP1<0>
PP2<7> PP2<6> PP2<5> PP2<4> PP2<3> PP2<2> PP2<1> PP2<0>
PP3<7> PP3<6> PP3<5> PP3<4> PP3<3> PP3<2> PP3<1> PP3<0>
P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0

Fig. 18: 8×8 approximate multiplier and its approximate summation

produce precise area (LUTs), critical path delay, and dynamic power consumption values. In each implementation-iteration, our automated tool flow adjusts the new critical path constraint according to the critical path-slack obtained from the previous iteration. The total number of implementation-iterations, performed by our tool flow to provide the final critical path, resource utilization, and dynamic power consumption information of a design, is adjustable. For this paper, we have kept the maximum number of implementation-iterations at *10*.

To evaluate the efficacy of the proposed accurate multipliers, we compare it to the existing standard multipliers, such as Xilinx LogiCORE Multiplier *IP* (area and speed optimized) [2], Booth Multipliers (*S3*) [9], Xilinx's default K-Map solved optimized multiplier (*S4*) [31], Wallace Tree (S5) [14], Dadda (*S6*) [15] multipliers, and signed multipliers (*S7*) [34] and (*S8*) [35]. We compare the proposed approximate multipliers for performance gains and output accuracies with S1 [25], S2 [23], library of 8-bit approximate multipliers EvoApprox8b [30], precision-reduced 8×8 multiplier with four LSBs of the final product rounded to zero and Xilinx accurate multiplier IP [2].

The designed multipliers have also been implemented for the image smoothing accelerator of the SUSAN application and a multilayer perceptron for classification of the MNIST dataset to record the area savings offered by our novel multipliers.

B. Evaluation and Characterization of Designed Multipliers

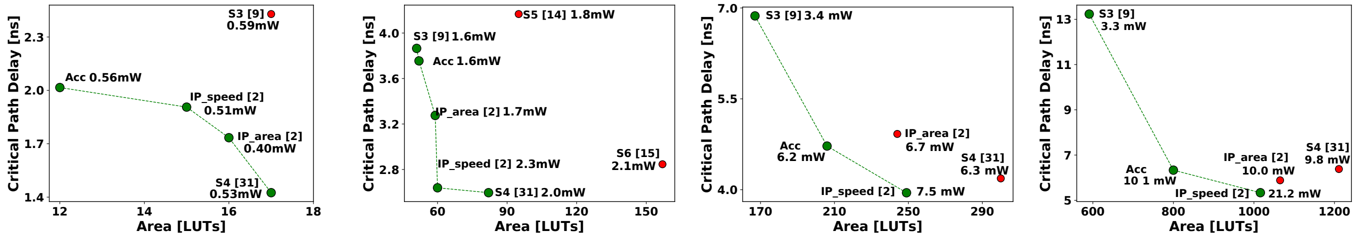
Table VI presents the implementation results of our proposed accurate (*Acc*) and approximate (*Approx*) unsigned multipliers. The N×N *Acc_acc* multipliers have been implemented utilizing four instances of $\frac{N}{2} \times \frac{N}{2}$ proposed accurate multipliers *Acc* and ternary adders. Similarly, the N×N *Acc_app* multipliers use four instances of $\frac{N}{2} \times \frac{N}{2}$ accurate multiplier *Acc* and the proposed approximate adder for the summation of partial products. For approximate 8×8 and 16×16 multipliers *Ca* and *Cc*, all sub-multipliers are approximate.

The 4×4 *Approx* multiplier offers reduced latency and energy consumption than the corresponding *Acc* multiplier. As shown by the results, the approximate summation of partial products has helped in significantly reducing the latency and energy consumption of *Cc* and *Acc_app* multipliers. Further, the proposed *Acc* multiplier consume less number of LUTs and has reduced latency for larger multipliers. This reduction in resource utilization and latency is due to the proposed method of partial product generation and their summation. For larger multipliers, the proposed accurate multiplier produces less number of partial products and computes final product in a fewer number of stages than the corresponding approximate multipliers.

1) Performance comparison of the proposed accurate multiplier with the state-of-the-art accurate multipliers: Fig. 19 compares the area and critical path delay requirements of different unsigned accurate multipliers for different bit-widths. The proposed *Acc* multiplier always lie on the area-delay Pareto fronts of different sizes multipliers. The *S5* and *S6*

TABLE VI: Area, latency and PDP results of proposed unsigned multipliers

Multiplier Size	Design	Area [LUTs]	Latency [ns]	PDP [pJ]
4x4	Acc	12	2.016	1.127
	Approx	12	1.564	0.649
8x8	Acc	52	3.755	6.099
	Acc_acc	57	3.508	6.910
	Acc_app	56	2.388	5.886
	Ca	57	3.130	4.732
	Cc	56	1.982	3.546
16x16	Acc	206	4.721	29.627
	Acc_acc	225	5.594	32.253
	Acc_app	224	4.301	28.448
	Ca	245	4.979	26.495
	Cc	240	2.375	16.155



(a) 4×4 (b) 8×8 (c) 16×16 (d) 32×32

Fig. 19: Area and critical path delay results of different accurate unsigned multipliers

Final edited form was published in "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems". 41 (2), S. 211 - 224. ISSN: 1937-4151.
https://doi.org/10.1109/TCAD.2021.3056337

TABLE VII: Performance comparison of proposed signed multiplier

Design	4x4			8x8			16x16			32x32		
	LUTs	CPD [ns]	PDP [pJ]	LUTs	CPD [ns]	PDP [pJ]	LUTs	CPD [ns]	PDP [pJ]	LUTs	CPD [ns]	PDP [pJ]
Proposed	14	2.59	1.31	54	4.37	7.26	208	5.28	31.14	803	7.35	63.75
S7 [34]	12	2.15	1.09	40	4.25	5.14	144	7.64	21.15	544	15.181	44.56
S8 [35]	18	1.65	1.13	66	2.8	6.06	243	4.48	26.52	928	6.08	57.50
Vivado IP Area Opt. [2]	30	2.91	2.25	88	3.45	9.07	326	5.04	35.25	1102	6.79	95.21
Vivado IP Speed Opt. [2]	18	2.14	1.06	74	3.54	5.73	286	4.27	33.91	1103	5.81	104.46

TABLE VIII: Error analysis of 8×8 approximate multipliers

Error Description	Approximate Architectures							
	Ca	Cc	Acc_app	S1 [25]	S2 [23]	Mult(8,4)	P8_1	P8_2
Maximum Error Magnitude	2312	8288	8160	7225	14450	15	509	1521
Average Error	54.19	1592.26	1579.12	1354.69	903.12	6.50	127.25	380.25
Average Relative Error	0.0029	0.13	0.13	0.14	0.032	0.0037	0.026	0.069
Error Occurrences	5482	52731	52437	53375	30625	53248	48896	61056
Maximum Error Occurrences	14	1	2	31	1	2048	1	1

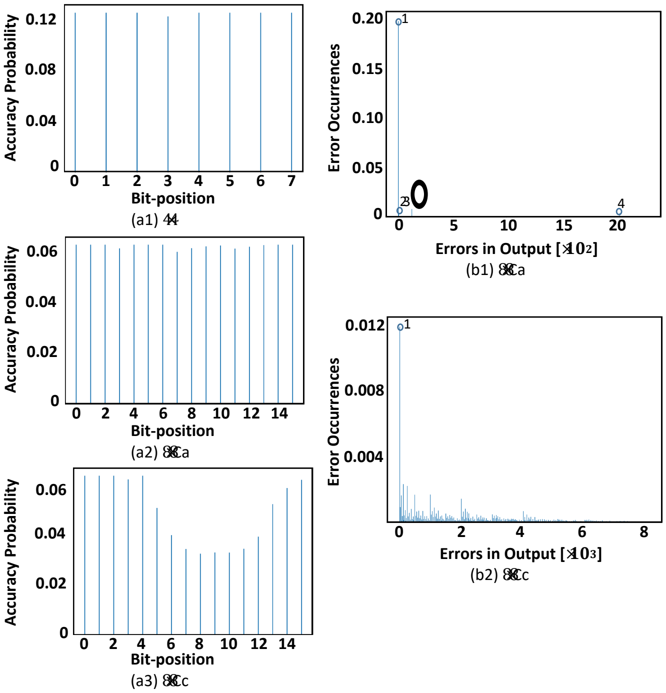


Fig. 20: Probability of error in individual product bits: (a) Normalized bit histograms of different multipliers, (b) Normalized probability mass functions (PMFs) of different multipliers

implementations consume a large number of LUTs, and have high critical path delays as shown in Fig. 19(b); therefore, they have not been considered for designing higher order multipliers. Even though the *S3* occupies a reduced number of LUTs for higher order multipliers, the sequential generation-addition of partial products, in *S3*, results in higher critical path delays. The dynamic power consumptions of all of the above implementations have also been shown along with each implementation.

Table VII compares the LUTs utilization, CPD, and PDP of the proposed signed multiplier with state-of-the-art signed multipliers for different bit-widths. Compared to the *S8* design and Vivado area- and speed-optimized signed multiplier IPs, the proposed accurate multiplier implementation always requires fewer LUTs. For example, the proposed 8×8 implementation offers a 38.63% reduction in LUT utilization when compared with Vivado area-optimized IP. Similarly, the energy consumption (PDP) of our proposed multiplier is also less than the Vivado area- and speed-optimized multiplier IPs. The *S7* design is more resource-efficient than the proposed implementation; however, *S7* has, on average, a higher CPD than our proposed design. For example, compared to the 16×16 *S7* design, our implementation offers a 30.8% reduction in the multiplier's critical path delay.

2) Error analysis of the proposed approximate multipliers: Table VIII presents an error analysis of our designed approximate multipliers in comparison with the state-of-the-art approximate multipliers and precision-reduced 8×8 multipliers Mult(8,4), P8_1, and P8_2. For Mult(8,4), the four LSBs of the final product are rounded to zero, and all other bits of the final product are computed accurately. The P8_1 and P8_2 precision-reduced multipliers truncate one and two bits of each operand, respectively. These precision-reduced operands are then utilized by corresponding smaller accurate multipliers (7×7 and 6×6) to compute the product, which is shifted by an appropriate number of bits (2-bits and 4-bits) to calculate the final product. The proposed multiplier *Ca* outperforms the existing approximate multipliers in terms of maximum error magnitude, average error, error occurrences and maximum error occurrences. The approximate multiplier *Cc* has a higher maximum error magnitude compared to the state-of-the-art S1 [25], however, the maximum error occurs only once for *Cc* while it occurs 31 times for S1 [25]. The precision-reduced Mult(8,4) has highest number of maximum error occurrences. Regardless of its low average relative error, its high resource utilization, 350 LUTs, filters it out in Pareto analysis. The precision-reduced P8_1 and P8_2 multipliers offer reduced utilization of resources due to the use of smaller accurate multipliers. However, P8_1 and P8_2 have the highest number of error occurrences. The average error and average relative error of P8_1 and P8_2 are also higher than the Ca multiplier.

To explore the erroneous bit values with their effect on final output and the frequency of error occurrences, Fig. 20 represents the normalized bit accuracy histograms and the normalized number of unique error occurrences for proposed multipliers. Our novel design restricts the errors to limited bits only. Except *Cc* multiplier, all other multipliers have few distinct errors. The low probability of getting accurate bit values for *Cc* is due to the highly-inaccurate approximate addition of the partial products. Such type of architectures, with limited distinct errors, can be easily configured to have an error-correction circuitry that can be turned on/off according to applications' requirements.

3) Performance comparison of the proposed approximate multipliers with the state-of-the-art multipliers: Fig. 21 compares the resource utilization, critical path delay (CPD), and power delay product (PDP) of our proposed accurate and ap-

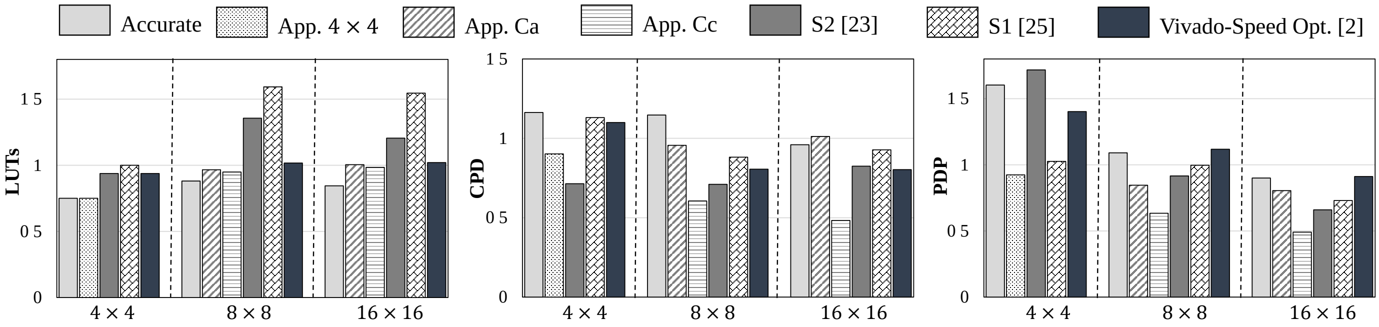


Fig. 21: LUTs utilization, critical path delay and energy consumption of accurate and approximate unsigned multipliers. Results are normalized to the corresponding results of Vivado area-optimized multiplier IP [2].

TABLE IX: Performance comparison of proposed multipliers with DSP blocks-based multipliers. The CPD and PDP are in *ns* and *pJ*, respectively.

Design	4 × 4				8 × 8				16 × 16				32 × 32			
	LUTs	**DSPs**	**CPD**	**PDP**	**LUTs**	**DSPs**	**CPD**	**PDP**	**LUTs**	**DSPs**	**CPD**	**PDP**	**LUTs**	**DSPs**	**CPD**	**PDP**
Approximate 4 × 4	12	0	1.564	0.63	-	-	-	-	-	-	-	-	-	-	-	-
Approximate Ca	-	-	-	-	57	0	3.13	4.73	245	0	4.98	26.50	1013	0	6.98	58.84
Approximate Cc	-	-	-	-	56	0	1.98	3.55	240	0	2.38	16.16	992	0	3.02	33.04
Accurate signed	14	0	2.58	1.31	54	0	4.37	7.26	208	0	5.28	31.14	803	0	7.35	63.75
Accurate unsigned	12	0	2.016	1.13	52	0	3.76	6.10	206	0	4.72	29.63	800	0	6.33	64.10
Vivado IP Area opt. [1]	0	1	3.355	4.47	0	1	3.57	5.45	0	1	3.57	5.91	**611**	**1**	7.28	56.41
Vivado IP Speed opt. [1]	0	1	3.355	4.47	0	1	3.54	5.45	0	1	3.68	6.00	0	**4**	6.86	22.37

proximate multipliers with Vivado speed-optimized multiplier IP and state-of-the-art approximate multipliers. These results have been normalized to the corresponding results of Vivado area-optimized multiplier IP. As shown, the proposed accurate multiplier and approximate multiplier Cc provide better resource utilization compared to the other implementations. For example, compared to the area-optimized multiplier IP, the accurate multiplier provides up to a 25% reduction in LUTs utilization. The approximate multipliers S1 and S2 consume more LUTs than the Vivado multiplier IPs. The proposed approximate Cc multiplier trades the output accuracy to achieve significant reductions in the critical path delay and energy consumption compared to the other implementations. For example, compared to the Vivado area-optimized multiplier IP, approximate multiplier Cc renders up to 51% and 50% reductions in the critical path delay and energy consumption, respectively.

We also compare our proposed accurate and approximate designs with DSP blocks-based multipliers. These results are presented in Table IX. To provide a thorough comparison, we have explored the various synthesis optimization strategies provided by the Xilinx Vivado synthesis tool for DSP blocks-based multipliers, such as area/speed optimization and unsigned/signed operations. However, the performance metrics of DSP blocks-based multipliers do not have a significant difference between unsigned and signed numbers based operations; therefore, we have shown the results for only signed numbers-based DSP blocks. The proposed approximate multiplier Cc has a lower CPD than DSP blocks-based multipliers for various bit-widths. Compared to the proposed approximate Ca and accurate multiplier implementations, the DSP blocks-based multipliers have higher CPD and PDP values for lower bit-widths multipliers, such as 4×4 and 8×8. For example, the proposed 4 × 4 signed multiplier offers a 23% and 70.6% reduction in CPD and PDP values, respectively, when compared with the area-optimized 4 × 4 DSP block-based multiplier. The DSP blocks-based multiplier's degraded performance is because the DSP48E1 slice in 7 series FPGAs (used for all experiments in this work) hosts a 25 × 18 multiplier and is not optimized for smaller multipliers. According to the design recommendations of Xilinx Vivado [1], LUTs-based soft multipliers should be used for implementing lower bit-widths multipliers. Our proposed accurate and approximate multipliers provide a feasible trade-off between accuracy, performance, and resource utilization for such scenarios. For higher-order multipliers, such as 16×16 and 32×32, the DSP blocks-based multipliers provide reduced CPD and PDP values than our proposed accurate multipliers. However, compared to the proposed 32 × 32 multipliers, the area-optimized DSP block-based multiplier utilizes one DSP slice and 611 LUTs. The corresponding speed-optimized 32 × 32 IP utilizes 4 DSP slices.

The utilization of DSP blocks along with a large number of LUTs for DSP blocks-based multipliers call for the orthogonal approach of defining resource-efficient soft multiplier architectures for multiplier-intensive applications, such as artificial neural networks, implemented on a small FPGA. Towards this end, we experimented on a small multilayer perceptron (MLP) to classify the MNIST dataset [44]. The inference accuracy of the dataset using the single-precision floating-point number is 97%. The corresponding inference accuracy using 8-bit fixed-point quantization is 96.6%, resulting in an insignificant drop in output accuracy. To evaluate the performance metrics of the quantized MLP implementation on FPGA, we implemented a single layer of the MLP on Xilinx Zynq UltraScale+ MPSoC (xczu3eg-sbva484-1-e device). The implementation consists of instantiating 20 neurons with 128 input activations. The experiment results using 8 × 8 DSP blocks-based multipliers and the proposed signed multiplier are presented in Table X. As shown, the DSP blocks-based implementation utilizes 54.5% and 34.6% of the total available DSP blocks and LUTs, respectively, on the FPGA. However, the corresponding

Final edited form was published in "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems". 41 (2), S. 211 - 224. ISSN: 1937-4151.
https://doi.org/10.1109/TCAD.2021.3056337

TABLE X: MLP implementation results on FPGA

Design	LUTs Utilization [%]	DSP_utilization [%]	CPD [ns]
DSP blocks	34.6	54.5	53.4
Proposed signed multiplier	46.4	0.0	57.2

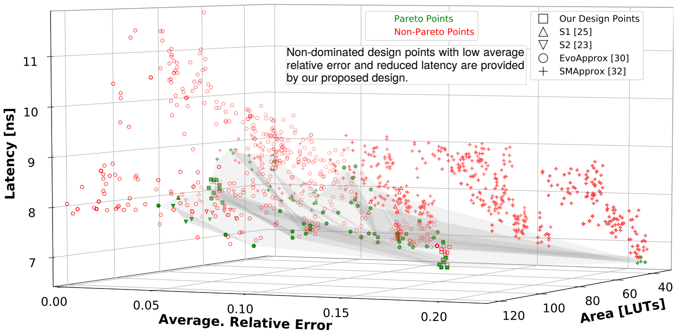


Fig. 22: Pareto optimal analysis of the proposed approximate 8×8 multipliers with state-of-the-art designs

implementation using our proposed multiplier utilizes only 46.4% of the total available LUTs. It is obvious to use both DSP blocks and soft multipliers to realize high-performance accelerators on resource-constrained FPGAs.

Finally, to provide a more exhaustive analysis of the proposed approximate multipliers, Fig. 22 compares the average relative error, the total number of utilized LUTs, and the critical path delay of all configurations of the proposed 8 × 8 approximate multipliers and state-of-the-art approximate multipliers S1 [25], S2 [23], EvoApprox8b [30] and SMApproxlib [32]. To minimize the design space evaluation time, we have obtained these results by synthesizing and implementing each design point only once, i.e., without utilizing the iterative implementation technique described in Section VII-A. The Pareto optimal analysis reveals that the number of non-dominated points reported by Evoapprox8b in [30] has significantly reduced for FPGA-based implementation. This analysis is in accordance with our observation of ASIC-based approximations less effective in producing comparable results for FPGA-based systems. For the 500 8 × 8 multiplier implementations provided by the EvoApprox8b library, only 8.6% of designs lie on the Pareto surface. Similarly, for the SMApprox library, only 6% of the implementations are non-dominated design points. However, our proposed approximate multipliers provide a better trade-off between resource utilization, critical path delay, and average relative error by offering 43.75% design points that lie on the Pareto surface. These results are summarized in Table XI. We have also performed the hypervolume analysis [43] of the Pareto design points to identify each implementation's dominance in the design space. The hypervolume indicator quantitatively computes the volume of the design space's dominant portion. As shown in Table XI, our proposed designs provide the maximum exclusive hypervolume contribution. Therefore, despite having only 14 solutions out of 93 non-dominated design points, our implementations offer the maximum coverage of the entire design space. Our proposed design points offer a better reduction in the average relative error, LUTs utilization, and critical path delay in the multi-objective design space of Fig. 22.

4) Quality Evaluation for Application Kernels: The proposed multipliers are also tested for the multiply-mode-based image blending application and SUSAN application-based image smoothing accelerator to observe degradation in the final output accuracy. For the image blending filter, we have utilized our approximate multipliers in the Python-based behavioral model of the application and used it for ten random test images from USC-SIPI Database [36]. In comparison to the accurate multiplier-based filter, the *Ca* and *Cc* multipliers-based filters produce an average of 51.9 dB and 31.6 dB *Peak signal-to-noise ratio* (PSNR) values. Fig. 23 presents the visual output along with respective PSNR values for a single image.

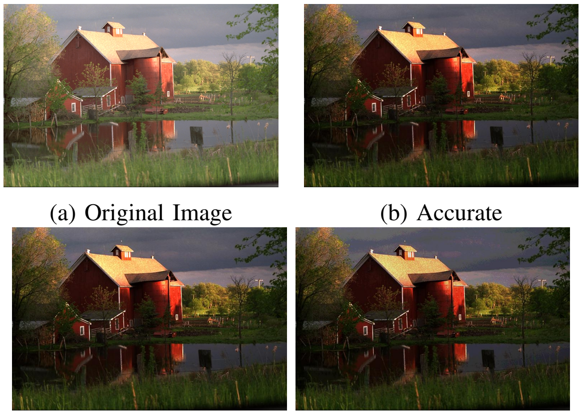

(a) Original Image
(b) Accurate
(c) *Ca*
(d) *Cc*

Fig. 23: Image blending application (multiply-mode); the PSNR values of the approximate multipliers-based filters are computed with respect to the accurate multiplier-based filter: *Ca* PSNR=51.9 dB, *Cc* PSNR=31.6 dB

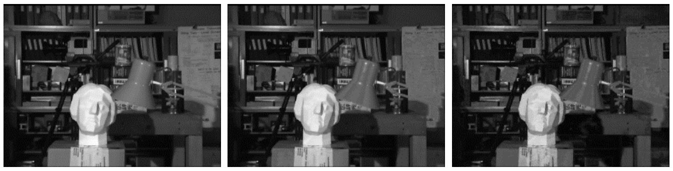

Accurate
Ca
Cc

Fig. 24: Accurate and approximate multipliers-based SUSAN image smoothing accelerator output

We also synthesized the SUSAN application-based image smoothing accelerator with Vivado's default multiplier *S4* and our proposed *Ca* and *Cc* multipliers using Xilinx Vivado. Our approximations offers 17%, and 17.2% area gains for *Ca* and *Cc* multipliers respectively with insignificant output quality loss. Fig. 24 and Table XII contrast the output visual qualities and the PSNR values of SUSAN image smoothing accelerator, using proposed approximate multipliers, accurate multiplier and state-of-the-art multipliers *S1* and *S2* respectively. The results show that our designed approximate multipliers provide better visual quality outputs and PSNR values than those

TABLE XI: Summary of Pareto optimal analysis of different 8×8 multipliers

Design	Total Design Points	Pareto Points	% Pareto Points	Max. Exclusive Hypervolume
S1 [25]	16	1	6.25	0.0004
S2 [23]	16	4	25.00	0.0445
EvoApprox [30]	500	43	8.60	0.0521
SMApprox [32]	512	31	6.05	0.0164
Ours Approximate	32	14	43.75	0.0656
Total	1076	93	-	-

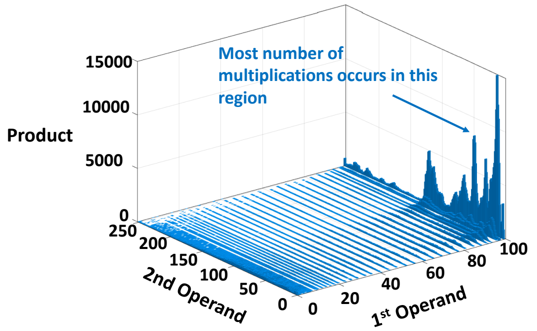


Fig. 25: Analysis of input image: SUSAN application 8×8 multiplication histogram

displayed by the *S2* multiplier. The approximate multiplier *S1*, apparently, produces better PSNR value than those produced by *Ca* and *Cc*. However, the input values analysis, in Fig. 25, of the image under consideration shows that most of the multiplications during the image smoothing process are limited to a narrow band and increasing the multiplication output accuracy for this band can increase the accelerator's output quality. Exploiting the asymmetric nature of our proposed multiplier, the mutual swapping of all input values to our approximate multipliers for SUSAN image smoothing accelerator and input-image under consideration results in enhanced output qualities with higher PSNR values as shown in Table XII. Hence depending upon the input-data and the application under analysis, *Ca, Cc* or Ca_s, Cc_s can be deployed for achieving desired area, latency, EDP gains with required output accuracy.

TABLE XII: PSNR values of 8×8 approximate multipliers

Multiplier Architecture	SUSAN Accelerator PSNR [dB]
Accurate	∞
Ca	33.716
Cc	25.602
S1	47.493
S2	17.944
Ca_s (Ca Swapped Inputs)	59.119
Cc_s (Cc Swapped Inputs)	27.366

VIII. Conclusion

In this paper, we have used the 6-input lookup tables and the associated carry chains of modern FPGAs to propose area-optimized, high-performance softcore accurate and approximate multipliers. Compared to the area-optimized multiplier IP provided by Vivado, our proposed accurate unsigned and signed multipliers provide up to 25% and 53% reduction in the total utilized LUTs, respectively. For an M × N accurate multiplier, our proposed methodology generates only $\frac{N}{2}$ partial products in parallel. It then utilizes ternary and binary adders to add the generated partial products to compute the final product. We have used the modular approach for implementing higher-order approximate multipliers using our proposed approximate 4×2 and 4×4 multipliers. Towards this end, we have also presented the design of a novel resource-efficient and high-performance ternary adder for adding the results of submultipliers. Our approximate multipliers provide up to 51% reduction in the critical path delay when compared with Vivado's area-optimized multiplier IP. Our proposed multipliers can be utilized in implementing resource-efficient high-performance accelerators for different applications. We have also applied our proposed multipliers in different real-world applications and tested for output quality and performance gains. We also intend to extend our proposed methodology of defining LUT-level optimizations for designing other resource-efficient and high-performance accurate and approximate arithmetic circuits, such as multiplier–accumulator (MAC) and dividers. We provide our open-source library of the accurate and approximate multiplier at https://cfaed.tu-dresden.de/pd-downloads to assist reproducible results.

References

[1] Xilinx 7 Series DSP48E1 Slice https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[2] Xilinx LogiCORE IP v12.0 https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf

[3] Integer Arithmetic IP Cores User Guide https://www.altera.com/en_US/pdfs/literature/ug/ug_lpm_alt_mfug.pdf

[4] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes and B. Popa, "Arithmetic core generation using bit heaps," 2013 23rd International Conference on Field programmable Logic and Applications, Porto, 2013, pp. 1-8.

[5] J. Beuchat and J. Muller, "Automatic Generation of Modular Multipliers for FPGA Applications," in IEEE Transactions on Computers, vol. 57, no. 12, pp. 1600-1613, Dec. 2008.

[6] Ahmet Kakacak, Aydin Emre Guzel, Ozan Cihangir, Sezer Gren, and H. Fatih Ugurdag. 2017. "Fast Multiplier Generator for FPGAs with LUT based Partial Product Generation and Column/row Compression,". in Integr. VLSI J. 57, C 2017, 147-157.

[7] M. Kumm, J. Kappauf, M. Istoan and P. Zipf, "Resource Optimal Design of Large Multipliers for FPGAs," 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH), London, 2017, pp. 131-138.

[8] E. G. Walters, "Array Multipliers for High Throughput in Xilinx FPGAs with 6-Input LUTs" in Computers, vol. 5, no. 4, 2016.

[9] M. Kumm, S. Abbas and P. Zipf, "An Efficient Softcore Multiplier Architecture for Xilinx FPGAs," 2015 IEEE 22nd Symposium on Computer Arithmetic, Lyon, 2015, pp. 18-25.

[10] H. Parandeh-Afshar and P. Ienne, "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs," 2011 21st International Conference on Field Programmable Logic and Applications, Chania, 2011, pp. 225-231.

[11] 7 Series FPGAs Configurable Logic Block https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[12] H. Parandeh-Afshar, P. Brisk and P. Ienne, "Exploiting fast carry-chains of FPGAs for designing compressor trees," 2009 International Conference on Field Programmable Logic and Applications, Prague, 2009, pp. 242-249.

[13] B. Parhami, "Computer Arithmetic Algorithms and Hardware Designs", 2000

[14] C. S. Wallace, "A Suggestion for a Fast Multiplier," in IEEE Transactions on Electronic Computers, vol. EC-13, no. 1, pp. 14-17, Feb. 1964.

[15] L. Dadda, "Some schemes for parallel multipliers", in Alta frequenza, vol. 34, no. 5, 1965.

[16] B. Millar, P. E. Madrid and E. E. Swartzlander, "A fast hybrid multiplier combining Booth and Wallace/Dadda algorithms," [1992] Proceedings of the 35th Midwest Symposium on Circuits and Systems, Washington, DC, USA, 1992, pp. 158-165 vol.1.

[17] V. K. Chippa, S. T. Chakradhar, K. Roy and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2013, pp. 1-9.

[18] A. K. Verma, P. Brisk and P. Ienne, "Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design," 2008 Design, Automation and Test in Europe, Munich, 2008, pp. 1250-1255.

[19] M. Shafique, W. Ahmad, R. Hafiz and J. Henkel, "A low latency generic accuracy configurable adder," 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2015, pp. 1-6.

[20] V. Gupta, D. Mohapatra, A. Raghunathan and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 32, no. 1, pp. 124-137, Jan. 2013.

[21] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," DAC Design Automation Conference 2012, San Francisco, CA, 2012, pp. 820-825.

[22] K. Bhardwaj, P. S. Mane and J. Henkel, ”Power- and area-efficient Approximate Wallace Tree Multiplier for error-resilient systems,” Fifteenth International Symposium on Quality Electronic Design, Santa Clara, CA, 2014, pp. 263-269.

[23] P. Kulkarni, P. Gupta and M. Ercegovac, ”Trading Accuracy for Power with an Underdesigned Multiplier Architecture,” 2011 24th Internatioal Conference on VLSI Design, Chennai, 2011, pp. 346-351.

[24] S. Hashemi, R. I. Bahar and S. Reda, ”DRUM: A Dynamic Range Unbiased Multiplier for approximate applications,” 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2015, pp. 418-425.

[25] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, J. Henkel and J. Henkel, ”Architectural-space exploration of approximate multipliers,” 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2016, pp. 1-8.

[26] C. Liu, J. Han and F. Lombardi, ”A low-power, high-performance approximate multiplier with configurable partial error recovery,” 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2014, pp. 1-4.

[27] C. Lin and I. Lin, ”High accuracy approximate multiplier with error correction,” 2013 IEEE 31st International Conference on Computer Design (ICCD), Asheville, NC, 2013, pp. 33-38.

[28] J. Mody, R. Lawand, R. Priyanka, S. Sivanantham and K. Sivasankaran, ”Study of approximate compressors for multiplication using FPGA,” 2015 Online International Conference on Green Engineering and Technologies (IC-GET), Coimbatore, 2015, pp. 1-4.

[29] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan and K. Roy, ”IMPACT: IMPrecise adders for low-power approximate computing,” IEEE/ACM International Symposium on Low Power Electronics and Design, Fukuoka, 2011, pp. 409-414.

[30] V. Mrazek, R. Hrbacek, Z. Vasicek and L. Sekanina, ”EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 258-261.

[31] Vivado Design Suite User Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug910-vivado-getting-started.pdf

[32] S. Ullah, S. S. Murthy and A. Kumar, “SMApproxlib: Library of FPGA-based Approximate Multipliers”, in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, 2018, pp. 1-6.

[33] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, A. Kumar, “Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators”, in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, 2018, pp. 1-6.

[34] S. Ullah, H. Schmidl, S. S. Sahoo, S. Rehman and A. Kumar, ”Area-optimized Accurate and Approximate Softcore Signed Multiplier Architectures,” in IEEE Transactions on Computers, doi: 10.1109/TC.2020.2988404.

[35] S. Ullah, T. D. A. Nguyen and A. Kumar, ”Energy-Efficient Low-Latency Signed Multiplier for FPGA-based Hardware Accelerators,” in IEEE Embedded Systems Letters, doi: 10.1109/LES.2020.2995053.

[36] SIPI Image Database (2019) http://sipi.usc.edu/database/database.php?volume=misc

[37] C. R. Baugh and B. A. Wooley, ”A Two’s Complement Parallel Array Multiplication Algorithm,” in IEEE Transactions on Computers, vol. C-22, no. 12, pp. 1045-1047, Dec. 1973, doi: 10.1109/T-C.1973.223648

[38] S. Mittal, “A Survey of Techniques for Approximate Computing”, in 2016 ACM Computing Surveys.

[39] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, J. Henkel, “Cross-layer approximate computing: From logic to architectures”, in 2016 53rd Annual Design Automation Conference (DAC).

[40] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf and U. Meyer-Baese, “Multiple constant multiplication with ternary adders,” in 2013 23rd International Conference on Field programmable Logic and Applications, Porto.

[41] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203-215, Feb. 2007.

[42] A. D. Booth, “A Signed Binary Multiplication Technique,” in The Quarterly Journal of Mechanics and Applied Mathematics 1951.

[43] E. Zitzler, D. Brockhoff and L. Thiele, “The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration,” In International Conference on Evolutionary Multi-Criterion Optimization, pp. 862-876. Springer, Berlin, Heidelberg, 2007.

[44] MNIST-cnn. (2016). [Online]. Available: https://github.com/integeruser/MNIST-cnn

Salim Ullah is a Ph.D. student at the Chair for Processor Design, Technische Universität Dresden. He has completed his BSc and MSc in Computer Systems Engineering from the University of Engineering and Technology Peshawar, Pakistan. His current research interests include the Design of Approximate Arithmetic Units, Approximate Caches, and Hardware Accelerators for Deep Neural Networks.

Semeen Rehman is currently with the Technische Universität Wien (TU Wien), Faculty of Electrical Engineering as a tenure- track Assistant Professor. Before that, she was a Postdoctoral Researcher with the Technische Universität Dresden (TU Dresden) and Karlsruhe Institute of Technology (KIT), Germany, since 2015. In July 2015, she received her Ph.D. from Karlsruhe Institute of Technology (KIT), Germany. Her main research interests include dependable systems, cross-layer design for error resiliency with a focus on run-time adaptations, emerging computing paradigms, such as approximate computing.

Muhammad Shafique (M’11 - SM’16) is a professor of Computer Architecture and Robust Energy-Efficient Technologies (CARE-Tech.) at the Institute of Computer Engineering, TU Wien, Austria since Nov. 2016. He received his Ph.D. in Computer Science from Karlsruhe Institute of Technology (KIT), Germany, in Jan. 2011. His research interests are in computer architecture, power-/energy-efficient systems, robust computing, hardware security, Brain-Inspired computing trends like Neuromorphic and Approximate Computing.

Akash Kumar (SM’13) received the joint Ph.D. degree in electrical engineering and embedded systems from the Eindhoven University of Technology, Eindhoven, The Netherlands, and the National University of Singapore (NUS), Singapore, in 2009. From 2009 to 2015, he was with NUS. He is currently a Professor with Technische Universität Dresden, Dresden, Germany, where he is directing the Chair for Processor Design. His current research interests include the Design, Analysis, and Resource Management of Low-Power and Fault-Tolerant Embedded Multiprocessor Systems.