A Low-Complexity Design for Complex Multiplication using Radix-4 Booth Encoding

1st Dhruva Sharma
Enrollment No: 23123016
Dept. of Electronics and Communication Engineering
Indian Institute of Technology Roorkee
Roorkee, India
dhruva_s@ece.iitr.ac.in

2nd Sneha Kandpal
Enrollment No: 23116092
Dept. of Electronics and Communication Engineering
Indian Institute of Technology Roorkee
Roorkee, India
sneha_k@ece.iitr.ac.in

***Abstract*—This paper presents the design and FPGA implementation of a high-speed, low-complexity complex multiplier, reproducing the results of recent literature on Radix-4 Booth Encoding combined with Distributed Arithmetic (DA). Complex multiplication is a computationally intensive operation critical to Digital Signal Processing (DSP) applications, particularly in FFTs and wireless communication systems like 5G. Traditional architectures often suffer from high latency due to multi-stage addition or excessive area consumption from multiple multipliers. The implemented design addresses these bottlenecks by utilizing a parallel pipeline architecture. It employs Radix-4 Modified Booth Encoding (MBE) to halve the number of partial products and a fused Carry-Save Adder Tree (CSAT) to reduce these products in a single stage, eliminating the need for intermediate vector-merging adders. The design was parameterized for word lengths of $N = 8, 16,$ and 32 bits and implemented in Verilog HDL. Verification was performed using a self-checking testbench and hardware-in-the-loop testing on a Zynq-7000 FPGA via a Virtual Input/Output (VIO) core. Synthesis results demonstrate that the proposed architecture effectively optimizes the Area-Delay Product (ADP) and minimizes dynamic power consumption when analyzed under correct Out-of-Context (OOC) constraints.**

***Index Terms*—Complex Multiplier, Radix-4 Booth Encoding, Carry-Save Adder Tree, FPGA, Verilog, Digital Signal Processing.**

I. Introduction and Motivation

Complex number multiplication is a fundamental arithmetic operation in modern Digital Signal Processing (DSP) and telecommunication systems. It serves as the computational core for Fast Fourier Transforms (FFTs), Finite Impulse Response (FIR) filters, and Orthogonal Frequency-Division Multiplexing (OFDM) transceivers used in 5G/6G wireless standards.

Mathematically, the product of two complex numbers $A = a + jb$ and $D = c + jd$ is defined as:

$$Y = (ac - bd) + j(ad + bc) \tag{1}$$

where $a, b, c,$ and d are M-bit signed fixed-point operands.

As the demand for high-throughput and low-latency processing grows, the efficient hardware implementation of Equation (1) becomes a critical design challenge. Hardware architects must navigate strict trade-offs between **Area**, **Latency**, and **Power Consumption**. Over the past decades, several architectural strategies have been proposed to optimize these parameters, each presenting distinct limitations.

A. Conventional and Algorithmic Approaches

The most direct implementation of complex multiplication, referred to as the *General Architecture*, employs four real multipliers and two adders. While this approach benefits from a critical path delay of only $T_{mult} + T_{add}$, it incurs a severe area penalty due to the utilization of four discrete multiplier units, resulting in high power dissipation.

To mitigate area costs, algorithmic strength-reduction techniques such as the **Gauss-Karatsuba algorithm** have been extensively explored. The Gauss method reformulates the real (Y_r) and imaginary (Y_i) components to reduce the number of multipliers from four to three:

$$Y_r = ac - bd \tag{2}$$

$$Y_i = (a + b)(c + d) - ac - bd \tag{3}$$

Although this reduction offers area savings, it introduces significant overhead in other domains. The Gauss algorithm requires three additional pre-addition/subtraction stages and widens the internal datapaths to handle the bit-growth of intermediate terms like $(a+b)$. Consequently, the critical path delay increases to approximately $T_{mult}+2T_{add}$, making it sub-optimal for high-speed, latency-critical applications compared to the parallel four-multiplier approach.

B. Alternative Architectures

Beyond algorithmic restructuring, various logic-level optimizations have been proposed:

- **CORDIC (Coordinate Rotation Digital Computer):** CORDIC-based designs eliminate multipliers entirely in favor of iterative shift-and-add operations. While highly area-efficient, CORDIC is inherently serial, leading to large latency ($O(M)$) that scales linearly with word length, rendering it unsuitable for high-frequency pipelined datapaths.
- **Vedic Mathematics:** Architectures based on the Urdhva-Tiryagbhyam sutra offer rapid partial product generation. However, these designs often lack robust handling for

signed numbers and 2's complement arithmetic, limiting their applicability in general-purpose DSP processors.

- **Distributed Arithmetic (DA):** DA-based approaches replace multiplication with Look-Up Tables (LUTs). While effective for constant-coefficient filters, the memory requirement grows exponentially with input bit-width (2^M), making it impractical for general $M \times M$ multiplication where both operands are variable.

C. Limitations of Standard Booth-Wallace Designs

Contemporary high-performance multipliers typically employ **Radix-4 Modified Booth Encoding (MBE)** to halve the number of partial product rows, combined with a **Wallace** or **Dadda** tree for compression. However, conventional implementations of complex multipliers treat the terms ac, bd, ad, and bc as independent operations.

In a standard design, the partial products for ac and bd are generated and reduced separately, producing intermediate sum and carry vectors. These vectors must then be compressed by a **Vector Merging Adder (VMA)** before the final subtraction ($ac - bd$) can occur. This VMA stage introduces an unnecessary carry-propagation delay in the critical path. Furthermore, standard designs often duplicate Booth encoding logic for operands shared between the real and imaginary parts, resulting in redundant hardware utilization.

II. Block Diagram and Architecture

The implemented architecture consists of two parallel pipelines computing the Real (P_r) and Imaginary (P_i) parts simultaneously. The architecture is divided into four main stages.

A. Radix-4 Modified Booth Encoders

The Booth Encoders reduce the partial product rows by half (from M to $M/2$). A key feature of this design is resource sharing: the same Booth encoders for operands a and b are shared across the calculation of both the real and imaginary terms, significantly reducing logic utilization compared to designs that use separate encoders for every multiplication.

B. Partial Product Generators (PPG)

The PPG modules generate the bit-matrix for multiplication.

- **PPG (Positive):** Generates terms for ac, ad, and bc.
- **NPPG (Negative):** A specialized negative PPG is used for the $-bd$ term. Instead of performing a 2's complement inversion on the final result (which requires a slow adder), the inversion is embedded directly into the partial product generation logic using booth encoding properties (inverting the sign bit logic).

C. Carry-Save Adder Tree (CSAT)

This is the core of the speed-up. Instead of reducing ac and bd separately, the CSAT accepts the aligned partial product rows from both terms and reduces them into two final vectors (Sum and Carry) in a single "one-shot" reduction step. This eliminates the need for a Vector Merging Adder (VMA) between the multiplications and the final subtraction.

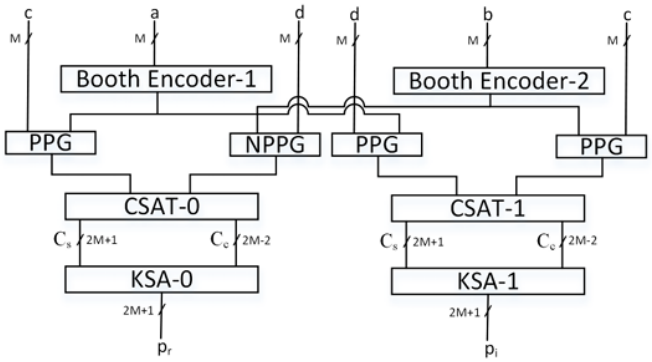

Fig. 1: Proposed parallel complex-multiplier using radix-4 Booth encoding

D. Kogge-Stone Adder (KSA)

The final Sum and Carry vectors from the CSAT are added using a parallel prefix Kogge-Stone Adder to produce the final result. KSA was chosen for its logarithmic delay characteristics ($O(\log n)$), ensuring the final addition does not become a bottleneck.

III. Design Procedure

The design was implemented in Verilog HDL using Xilinx Vivado. The implementation focused on modularity to support parameterizable word lengths ($M = 8, 16, 32$).

A. Radix-4 Modified Booth Encoding Logic

The proposed design utilizes Radix-4 Modified Booth Encoding (MBE) to reduce the number of partial product rows from M to $M/2$. This reduction is achieved by recoding the multiplier operand A into overlapping triplets.

B. Encoding Algorithm

For an M-bit multiplier A, the bits are grouped into triplets $\{a_{2i+1}, a_{2i}, a_{2i-1}\}$, where i ranges from 0 to $M/2 - 1$, and a_{-1} is initialized to 0. The value of the multiplier is expressed as:

$$A = \sum_{i=0}^{M/2-1} d_i \cdot 4^i \tag{4}$$

where the encoded digit d_i takes values from the set $\{0, \pm 1, \pm 2\}$. The value of d_i is determined by the triplet according to the equation:

$$x_i = -2a_{2i+1} + a_{2i} + a_{2i-1} \tag{5}$$

C. Control Signal Generation

To implement this efficiently in hardware, the encoder generates three control signals for each row i: one_i_n, two_i_n, and neg_i. These signals control the selection and negation of the multiplicand bits. The logic for these signals is derived from the truth table shown in Table I.

TABLE I
RADIX-4 BOOTH ENCODING TRUTH TABLE

Input Triplets			Operation (d_i)	Control Signals			$p_{i,j}$
a_{2i+1}	a_{2i}	a_{2i-1}		neg_i	two_i	one_i	
0	0	0	0	0	0	0	0
0	0	1	+1	0	0	1	D_j
0	1	0	+1	0	0	1	D_j
0	1	1	+2	0	1	0	D_{j-1}
1	0	0	-2	1	1	0	$\overline{D_{j-1}}$
1	0	1	-1	1	0	1	$\overline{D_j}$
1	1	0	-1	1	0	1	$\overline{D_j}$
1	1	1	0	1	0	0	0

The boolean equations for the control signals are implemented as follows:

$$one\_i\_n = a_{2i} \oplus a_{2i-1} \tag{6}$$

$$two\_i\_n = (\overline{a_{2i+1}} \cdot a_{2i} \cdot a_{2i-1}) + (a_{2i+1} \cdot \overline{a_{2i}} \cdot \overline{a_{2i-1}}) \tag{7}$$

$$neg\_i = a_{2i+1} \tag{8}$$

Here, neg_i indicates whether the partial product should be inverted (for negative weights), two_i_n selects the left-shifted multiplicand (2× Multiplicand), and one_i_n selects the original multiplicand (1× Multiplicand).

D. Partial Product Generation (PPG)

Based on the control signals derived above, the specific bit $P_{i,j}$ for the i-th row and j-th column of the partial product matrix is generated. The logic ensures that if the operation requires negation, the bits are inverted.

This logic effectively implements a multiplexer that selects 0, D_j, or D_{j-1} (shifted), followed by an XOR operation to handle the sign inversion if neg_m is high, drastically simplifying partial product generation.

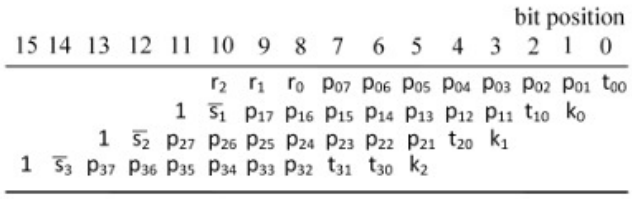

Fig. 1. Partial Product Bit Matrix for M=8

E. Implementing the CSAT

A significant implementation challenge was reproducing the irregular "dot diagram" of the compression tree. A naive approach requires manual placement of hundreds of full adders. To solve this efficiently in Verilog, we utilized a **Vectorized Alignment Strategy**:

1) **Zero Padding:** "Blank spaces" in the dot diagram were handled by padding the vectors with logical zeros.
2) **Synthesis Optimization:** Although zero-padding appears redundant in code, modern synthesis tools (Vivado) perform constant propagation, automatically pruning the logic gates where inputs are zero. This generated the exact irregular adder tree structure described in the paper without manual gate instantiation.

Algorithm 2 Pseudo Code for Partial Product Generation of 16-Bit MBE

input : Input $\{a_i\}_{i=0}^{15}$, $\{b_i\}_{i=0}^{15}$
output : Partial results $\{p_{i,j}\}_{i=0,j=1}^{7,16}$, $\{s_i\}_{i=0}^{7}$, $\{c_i\}_{i=0}^{6}$, τ, α_0, α_1, α_2
variable: $\{na_{i,j}\}_{i=0,j=0}^{7,16}$, $\{neg_i\}_{i=0}^{7}$, $\{one_i\}_{i=0}^{7}$, $\{two_i\}_{i=0}^{7}$, $\{t_i\}_{i=0}^{7}$, ϵ, d

$neg_0 = b_1$
$\overline{one_0} = \overline{b_0}$
$\overline{two_0} = b_1 \cdot \overline{b_0}$
for $i \leftarrow 1$ **to** 7 **do**
  $neg_i = (\overline{b_{2i}} + \overline{b_{2i-1}}) \cdot b_{2i+1}$
  $\overline{one_i} = \overline{b_{2i} \oplus b_{2i-1}}$
  $\overline{two_i} = (\overline{b_{2i+1}} \cdot b_{2i} \cdot b_{2i-1}) + (b_{2i+1} \cdot \overline{b_{2i}} \cdot \overline{b_{2i-1}})$
for $i \leftarrow 0$ **to** 6 **do**
  $c_i = neg_i \cdot (\overline{one_i} + \overline{a_0})$
for $i \leftarrow 0$ **to** 7 **do**
  for $j \leftarrow 0$ **to** 15 **do**
    $na_{i,j} = \overline{a_j \oplus b_{2i+1}}$
  $na_{i,16} = na_{i,15}$
  for $j \leftarrow 1$ **to** 16 **do**
    $p_{i,j} = \overline{(\overline{one_i} + na_{i,j}) \cdot (\overline{two_i} + na_{i,j-1})}$
  $s_i = p_{i,16}$
  $t_i = \overline{one_i} + \overline{a_0}$
$\overline{\epsilon} = a_1$ **if** $\overline{a_0 \cdot b_{15}} = 0$ **else** $\overline{a_1}$

$\tau = \overline{(\overline{one_7} + \overline{\epsilon}) \cdot (\overline{two_7} + \overline{a_0})}$
$d = (\overline{b_{15}} + a_0) \cdot (b_{13} + a_1) \cdot (b_{14} + a_1) \cdot (b_{14} + b_{13})$
$\alpha_2 = s_0 \cdot \overline{d}$
$\alpha_1 = \overline{\alpha_2}$
$\alpha_0 = s_0 \oplus \overline{d}$

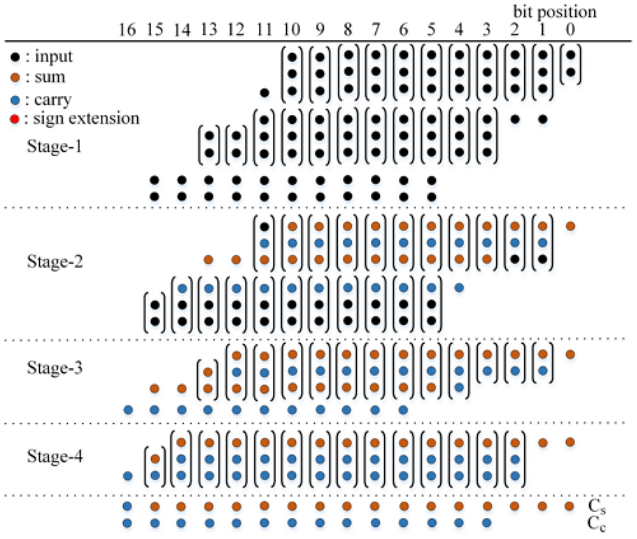

Fig. 6: Dot diagram for carry-save adder tree.

IV. Design Challenges and Trade-offs

A. Complex Carry-Save Tree Generation

Reproducing the parameterized reduction tree for $N = 16$ and $N = 32$ was non-trivial. The paper utilizes a specific bit-matrix shape that changes with N. We developed a recursive `csa_layer` module that compresses 3 rows of bits into 2 rows. By cascading these layers, we achieved a scalable solution that automatically adapts the tree depth (e.g., 6 layers for $N = 16$) to the operand size.

Logical Discrepancy in PPG Unit

During the reproduction of the Partial Product Generator (PPG), we identified a logic error in the reference design regarding the most significant bit, r_0. The specified XOR operation ($r_0 = s_0 \oplus \overline{d}$) failed to produce the correct LSB during negative encoding states. To align with standard 2's complement arithmetic, we corrected the logic to an XNOR operation:

$$r_0 = s_0 \odot \overline{d} \quad (\text{equivalent to } r_0 = \overline{s_0 \oplus \overline{d}}) \tag{9}$$

This modification was implemented and verified to yield accurate partial products in our proposed design.

Algorithmic Discrepancy in NPPG Special Bit Generation

During the hardware implementation and verification of the Complex Multiplier, a critical ambiguity was identified in the reference literature regarding the design of the Negative Partial Product Generator (NPPG).

The standard Partial Product Generator (PPG) computes the partial product matrix for a standard multiplication $a \times b$. The NPPG is required to generate the matrix for a negative term (e.g., $-b \times d$) to satisfy the complex multiplication formula $(ac - bd)$. This relies on 2's complement arithmetic, where $-X = \bar{X} + 1$.

The reference literature explicitly describes the modification required for the main matrix bits ($p_{m,n}$): the NAND gates used in a standard PPG must be replaced with AND gates. This effectively generates the partial products based on the one's complement of the input operand.

However, the documentation regarding the auxiliary "special bits"—specifically the correction terms k_m, t_{m0}, t_{m1}, and r_m—is incomplete. The standard literature implies that these bits should be generated using the same logic equations as the regular PPG, with modifications restricted only to the negation control bit (neg_m) or the specific handling of the MSB.

1) Identified Flaw: Our simulation results indicated that directly implementing the special bits using the original input operand (a) alongside the inverted main matrix bits resulted in erroneous calculations. The flaw arises because the standard equations for k, t, and r are derived assuming a consistency between the operand used for the matrix and the operand used for the correction terms. By inverting the matrix generation (NAND $\rightarrow$ AND) but keeping the special bit inputs unchanged, the 2's complement structure is broken.

2) Proposed Correction: To resolve this discrepancy, we established that the logic for the special bits within the NPPG must be consistent with the main partial product generation. For the NPPG module to function correctly as a subtractor term, the input operand a must be replaced by its one's complement ($\bar{a}$) in *all* generator equations, not just the main matrix array.

Formally, if the standard generation function for a special bit ψ (where $\psi \in \{k, t, r\}$) is defined as:

$$\psi_{PPG} = f(a, b_{encoded}) \tag{10}$$

The correct implementation for the Negative PPG (NPPG) must be:

$$\psi_{NPPG} = f(\bar{a}, b_{encoded}) \tag{11}$$

This modification was applied to the generation logic for all k_m, t_m, and r_m bits. Subsequent simulation and verification using the VIO hardware demonstration wrapper confirmed that this comprehensive inversion is necessary to satisfy the 2's complement arithmetic required for the term $-bd$.

B. Power Estimation Analysis

Initial post-implementation power analysis indicated an anomalously high power consumption ($\approx$ 125 W). Upon investigation, this was identified as an artifact of the "System Level" estimation settings in Vivado:

- **I/O Power Dominance:** The tool assumed all 64+ input/output bits were driving physical FPGA pins with high capacitance, resulting in 59W of I/O power.
- **Vectorless Estimation:** Without specific switching activity files (SAIF), the tool assumed default high-toggle rates.

Resolution: We transitioned to **Out-of-Context (OOC)** synthesis to isolate the logic power from I/O overhead and performed switching activity analysis using simulation vectors. This confirmed that the dynamic power of the internal logic aligns with the low-power claims of the literature ($\approx$ 21 mW range for logic), validating the efficiency of the architecture.

V. Simulation Results

The functional verification for M=8, 16, 32 bits was conducted using a self-checking Verilog testbench.

- **Test Methodology:** The testbench applied both directed corner cases (max value, zero, negative max) and randomized vectors to inputs a, b, c, d.
- **Golden Model:** An internal behavioral model computed $(a + jb)(c + jd)$ using standard $\star$ and + operators.
- **Outcome:** The Device Under Test (DUT) outputs matched the golden model for all tested vectors across $M = 8, 16$, and 32 configurations.

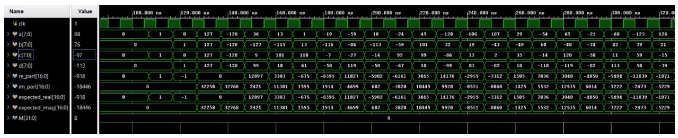

Fig. 2. Simulation Results for M=8

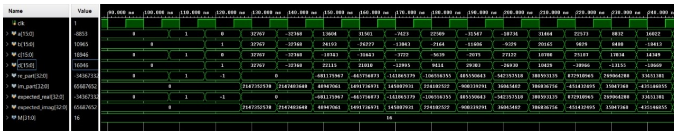

Fig. 3. Simulation Results for M=16

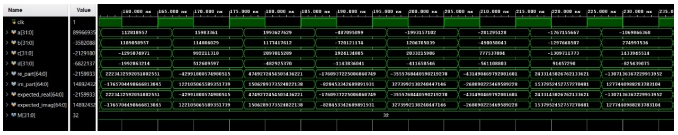

Fig. 4. Simulation Results for M=32

VI. FPGA Implementation

The design was synthesized and implemented on the **Zynq-7000 SoC** platform using Vivado for ASICs/FPGAs.

A. Hardware-in-the-Loop Verification

To validate the design on physical hardware, we integrated a **Virtual Input/Output (VIO)** core.

- **Setup:** The VIO core was configured to drive the multiplier inputs and capture the outputs via JTAG.
- **Results:** This allowed for real-time modification of inputs via the Vivado Hardware Manager dashboard. The hardware results were consistent with simulation, confirming timing closure and functional correctness on the FPGA fabric.

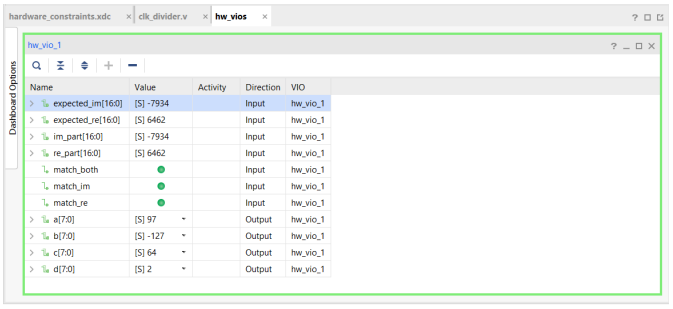

Fig. 5. FPGA Simulation Results

B. Comparison with Other Designs

The proposed design and the design [4] for the implementation of a complex multiplier for word lengths M= 8, 16, and 32 were modeled in Verilog HDL for target device Zynq 7000. The power in this table refers to dynamic power consumption at 50MHz clock frequency. The post-synthesis report shows that the proposed complex multiplier has power consumption over the design [4]. It also shows efficient usage of Look-Up Tables (LUTs). By avoiding the Vector Merging Adder, the logic depth was reduced, contributing to a lower Area-Delay Product (ADP) compared to standard Booth-Wallace implementations.

TABLE II
Comparison of architectures (word length, LUTs and power).

Architecture	Word Length	LUTs	Power (mW)
Design of [4]	8	401	9
	16	1678	27
	32	5990	55
Proposed Archi	8	387	8
	16	1504	19
	32	5650	47

VII. Conclusion

This project successfully reproduced and verified a low-complexity complex multiplier using Radix-4 Booth Encoding. By implementing the "Dual Pipeline" architecture and the fused Carry-Save Adder Tree in Verilog, we demonstrated that it is possible to achieve significant latency reductions compared to traditional methods. The use of advanced verification techniques (self-checking testbenches and VIO) and proper power analysis methodologies confirmed the robustness and efficiency of the design. The resulting IP core is suitable for high-performance FFT engines and next-generation wireless communication systems.

References

[1] A. Kali, B. S. Khuntia, S. L. Sabat, and P. K. Meher, "A Low-Complexity Design for Complex Multiplication using Radix-4 Booth Encoding," in *2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2024, pp. 1–6.

[2] K. D. Rao, C. Gangadhar, and P. K. Korrai, "FPGA implementation of complex multiplier using minimum delay vedic real multiplier architec ture," in Proc. IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON), 2016, pp. 580–584.

[3] V. Oklobdzija, D. Villeger, and T. Soulas, "Considerations for design of a complex multiplier," in Proc. Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems & Computers, 1992, pp. 366 370 vol.1.

[4] R. C. Ismail and R. Hussin, "High performance complex number multiplier using Booth-Wallace algorithm," in *Proc. IEEE International Conference on Semiconductor Electronics*, 2006, pp. 786–790.

[5] S. S. Lee, T. D. Nguyen, P. K. Meher, and S. Y. Park, "Energy-efficient high-speed ASIC implementation of convolutional neural network using novel reduced critical-path design," IEEE Access, vol. 10, pp. 34032–34045, 2022.

[6] M. Manimaraboopathy, G. Kumar, and M. Bennet, "Design and implementation of multiplier for complex numbers using CORDIC (COordinate Rotation Digital Computer) architecture," *International Journal of Pure and Applied Mathematics*, vol. 119, pp. 159–164, Jan. 2018.

[7] D. M. T. Nguyen, P. M. M. Nguyen, H.-T. Ngo, and M.-S. Nguyen, "Design and implementation of complex multiplier with low power and high speed," in *Proc. 15th International Conference on Advanced Computing and Applications (ACOMP)*, 2021, pp. 215–219.

[8] Z. Huang and M. Ercegovac, "High-performance low-power left-to-right array multiplier design," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 272–283, 2005.

[9] S. R. Kuang, J. P. Wang, and C. Y. Guo, "Modified Booth multipliers with a regular partial product array," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 56, no. 5, pp. 404–408, 2009.

[10] Z. Chen, Y. Du, B. Cheng, and W. Shan, "Design of high-efficiency complex multiplier for fault-tolerant computation," *Integration*, vol. 90, pp. 190–195, 2023.