Fast and Scalable MAGIC-Based Wallace Tree Multiplier for In-Memory Computing

Saeideh Nabipour*, Kamalika Datta*†, Abhoy Kole*, Saeideh Shirinzadeh*‡, Rolf Drechsler*†
* Cyber-Physical Systems, DFKI GmbH, Germany
† Institute of Computer Science, University of Bremen, Germany
‡ Fraunhofer Institute for Systems and Innovation Research (ISI), Karlsruhe, Germany
{saeideh.nabipour, abhoy.kole, saeideh.shirinzadeh}@dfki.de, {kdatta, drechsler}@uni-bremen.de

***Abstract*—The growing demand for high-performance, real-time computation in data-intensive applications is increasingly constrained by the Von Neumann bottleneck. *In-memory computing (IMC)*, particularly through memristor-based technologies such as *Memristor-Aided loGIC (MAGIC)*, offers a promising solution by enabling logic operations directly within memory arrays. While prior research has demonstrated basic Boolean logic with memristors, arithmetic operations such as multiplication remain latency-bound due to sequential logic execution and inefficient crossbar utilization. This work introduces a scalable and efficient MAGIC-based Wallace Tree multiplier architecture tailored for in-memory computing. By integrating an optimized 3:2 compressor and leveraging a state-of-the-art synthesis-to-micro-operation mapping tool, our approach significantly reduces latency and improves parallelism within memristor crossbars. Experimental evaluations across 4- to 64-bit unsigned Wallace Tree multipliers show consistent improvements in speed and scalability. The proposed architecture presents a practical and fully scalable design for next-generation in-memory arithmetic systems.**

***Index Terms*—In-Memory Computing, Wallace Tree Multiplier, Memristor, MAGIC Design Style, Low-Latency Arithmetic**

I. INTRODUCTION

Traditional von Neumann architectures face inherent data movement bottlenecks due to separate memory and processing units. *In-memory computing (IMC)* using memristor-based crossbars addresses this challenge by enabling computation directly within the memory, thereby offering high density and energy-efficient operations. The *Memristor-Aided loGIC (MAGIC)* design style [1] supports Boolean operations such as NOR and NOT using only memristors, providing a compact and efficient computation model. Although significant progress has been made in optimizing basic logic operations through MAGIC [**?**], [2]–[5], implementing efficient arithmetic units remains a major challenge. Multipliers, in particular, are computationally demanding and are fundamental components in microprocessors, *Digital Signal Processors (DSPs)*, embedded systems [6], and *Convolutional Neural Networks (CNNs)* [7]. However, existing memristor-based implementations still suffer from high latency, area overhead, and limited scalability in memristor-based implementations.

To address these challenges, we propose a novel MAGIC-based Wallace Tree multiplier optimized for in-memory implementation. The Wallace Tree is selected over the traditional array and Dadda designs due to its lower computational depth and higher parallelism [8], [9], making it ideal for memristive crossbars. Our work introduces the first MAGIC-compatible implementation of a CMOS 3:2 compressor [10], which is employed in the partial product reduction stage. This innovation reduces both latency and memristor count. These compressors are integrated using an automatic mapping method [11] that converts Boolean logic into crossbar-compatible micro-operations. Unlike the *As Late As Possible (ALAP)* scheduling in [11], which delays operations to optimize resource usage and power, we adopt an *As Soon As Possible (ASAP)* strategy to enable faster execution, increased parallelism, and reduced latency. We evaluate the proposed architecture on unsigned multipliers with operand widths ranging from 4 to 64 bits. The results show a significant reduction in latency compared to existing designs [6], [12]–[16], establishing the practicality and scalability of our method. The main contributions of this work are:

- We design a low-latency MAGIC-based in-memory unsigned Wallace Tree multiplier that incorporates optimized 3:2 compressors to improve structural regularity and minimize mapping latency.
- We extend a state-of-the-art automatic mapping methodology to incorporate high-performance unsigned Wallace Tree multipliers into MAGIC-based crossbar arrays at the micro-operation level, achieving enhanced parallelism and lower latency.
- We evaluate the proposed architecture by implementing unsigned Wallace Tree multipliers with input widths ranging from 4 to 64 bits, generating detailed micro-operation schedules for comprehensive performance analysis.

The rest of this paper is organized as follows. Section II provides the necessary preliminary and related work. In Section III, we describe the synthesis and mapping process; Section IV then details the proposed MAGIC-based Wallace Tree multiplier. In Section V, the experimental results are discussed, followed by concluding remarks in Section VI.

II. PRELIMINARY AND RELATED WORKS

A. MAGIC Design Style

Memristor-Aided loGIC (MAGIC) design style, proposed in [17], uses memristor resistance states to represent logic states, enabling efficient Boolean computation within crossbar arrays. MAGIC supports all basic logic gates; however, only NOR and NOT can be directly mapped to crossbar rows or columns. Since NOR is a universal gate, any logic function can be constructed using a sequence of NOR gates. A MAGIC operation involves two steps: (1) initialization, where the output memristor is preset based on the gate type (R_{off} (0) for non-inverting gates and R_{on} (1) for inverting ones); and (2) evaluation, where an input voltage (V_{in}) triggers the output to switch based on input logic states. Fig. 1(a) shows

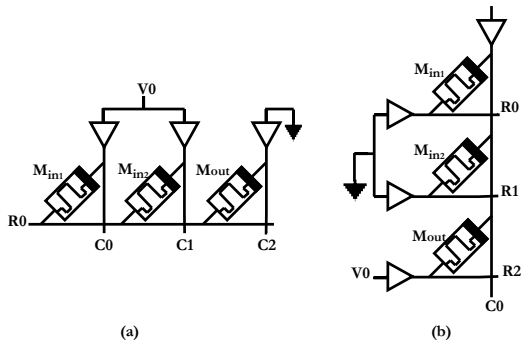

Fig. 1. MAGIC implementation of 2-inputs NOR gate (a) row-wise mapping (b) column-wise mapping.

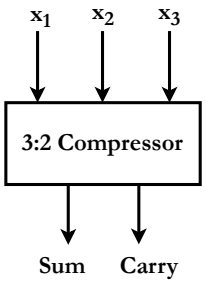

Fig. 2. The schematic of 3:2 compressor.

a 2-input NOR gate using one row and three columns. The symmetric crossbar structure also enables column-wise NOR operations, as shown in Fig. 1(b). Complex functions can be composed of NOR and NOT gates and mapped using existing techniques [4], [5], [18], [19].

B. Wallace Tree Multiplier

The Wallace Tree multiplier is a widely used architecture in high-performance processors and memory units due to its efficiency in accelerating multiplication operations [8], [9]. It performs the multiplication of two n-bit operands through three principal stages:

1) ***Partial Product Generation (PPG):*** Each bit of the multiplier is logically ANDed with each bit of the multiplicand, resulting in n^2 partial products.
2) ***Partial Product Reduction (PPR):*** The partial products are systematically reduced using a parallel tree structure composed of half adders and full adders, grouped by their respective weight positions. The resulting carry and sum outputs are then propagated to the next level of the reduction tree.
3) ***Carry Propagation Addition (CPA):*** The final two rows of partial products are summed using a fast carry-propagate adder, which constitutes the critical path and determines the overall latency of the multiplier.

By leveraging parallelism in the reduction stage, the Wallace Tree architecture minimizes the depth of the critical path and achieves significant improvements in speed, making it well-suited for integration in high-speed arithmetic units.

C. Compressor 3:2

A compressor in digital circuits is a logic component designed to perform multi-input addition while minimizing the number of output signals [20]. This is achieved by generating a compact sum along with associated input and output carry bits. Specifically, an $m{:}n$ compressor (where $n < m$) takes m equally weighted partial product bits as inputs and generates their sum as an n-bit output, along with carry. Consider two unsigned n bit inputs, $A = \sum_{i=0}^{n-1} A_i 2^i$ and $B = \sum_{i=0}^{n-1} B_i 2^i$. The product P between A and B is calculated as:

$$P = A{\cdot}B = p_{2n-1}2^{2n-1} + \cdots + p_0 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} A_i B_j 2^{i+j}. \quad (1)$$

The computation of P involves summing partial products $A_i B_j$, each weighted by 2^{i+j}. To improve structural regularity, reduce latency, and lower energy consumption during the partial product reduction phase, compressors are employed instead of traditional full adders [21], [22]. The 3:2 compressor is one of the most widely used compressors in digital circuit design. As shown in Fig. 2, it takes three input bits and outputs a sum and a carry bit, making it a key component in efficient multiplier architectures.

D. Related Works

Several studies have explored memristor-based multipliers using logic primitives such as MAGIC, Majority, and IMPLY [6], [12], [16], [23], [24]. However, many of these designs are not well-suited for integration with memory arrays due to their lack of support for automatic generation of micro-operations, which limits both scalability and programmability. For example, IMPLY-based logic often requires strictly ordered operations and additional control circuits, making it difficult to implement efficiently in memory arrays designed for fast, parallel processing. While some studies evaluate gate counts and computation cycles on memristive crossbars, they typically rely on manual or semi-automated mapping techniques and do not fully address the challenges of micro-operation scheduling. For instance, [16] uses the SIMPLER tool [5] for gate-level synthesis but does not provide analysis at the crossbar level. Similarly, [6] proposes an energy-efficient in-memory Wallace Tree multiplier using majority logic in SOT-MRAM, yet it lacks a method for extracting micro-operations. Other works [12], [23], [24] use IMPLY logic, but face limitations in scalability and practical system integration. For example, [13] maps IMPLY logic to crossbars but requires significant architectural modifications, such as additional control switches and a work resistor (R_G).

III. The Synthesis and Mapping Process

This section outlines the complete framework for the synthesis and mapping of Wallace Tree multipliers in MAGIC-based memristor crossbar. As illustrated in Fig. 3, the process involves two main stages: synthesis and mapping. In the synthesis stage, Boolean functions for unsigned Wallace Tree multipliers (ranging from 4 to 64 bits), described in Verilog, are first formally verified using the RevSCA-2.0 framework

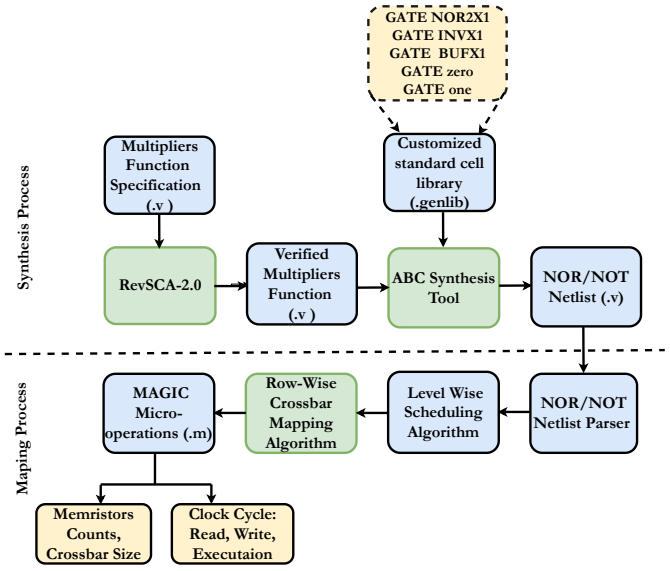

Fig. 3. The overall synthesis and crossbar-level mapping process in MAGIC-based logic design.

[25], a word-level verification method based on symbolic computer algebra. The verified designs are then synthesized into NOT and NOR gate netlists using the ABC synthesis tool [26]. The gate library used includes MAGIC-compatible primitives: NOT ($Y = \neg A$), NOR ($Y = \neg A.\neg B$), buffer ($Y = A$), constant zero ($Y = CONST0$), and constant one ($Y = CONST1$). Although ABC [26] optimizes the gate counts, it does not consider in-memory logic constraints such as placement or timing [5]. In the mapping stage, an *ASAP* scheduling strategy is applied as detailed in Algorithm 1, to assign logic gates to their earliest possible execution level based on input dependencies. In contrast to the previously adopted *ALAP* scheduling approach in [11], this method promotes faster execution, higher parallelism, and reduced overall latency.

A case study of a 4-bit Wallace Tree multiplier with a ripple-carry adder in the final stage shows that *ASAP* scheduling achieves a total latency of 271 cycles, consisting of 153 read cycles, 88 write cycles, and 30 evaluation cycles. In contrast, *ALAP* scheduling results in a higher total latency of 296 cycles, with the same 153 read cycles but increased write and evaluation cycles—101 and 42 respectively. The increased latency under *ALAP* scheduling is due to higher logic levels and reduced concurrency, which delay operations and increase write and evaluation cycles, resulting in overall performance degradation. The resulting scheduled netlist is then mapped to the crossbar using the mapping tool proposed in [11], allowing a row-wise parallel evaluation of the gates within each level. The corresponding micro-operations are then generated and saved in a `.m` file. For further details on the synthesis and mapping processes, we refer the reader to [11].

Algorithm 1 ASAP Scheduling Algorithm

Input: Logic netlist $G(V, E)$ with gates V and dependencies E
Input: Primary inputs $PI \subseteq V$
Output: Level assignment $L(g)$ for each gate $g \in V$
for $g \in V$ **do**
  $L(g) \leftarrow$ undefined
end for
for all $g \in PI$ **do**
  $L(g) \leftarrow 1$
end for
Perform topological sort on G to obtain ordered list $S = [g_1, g_2, \ldots, g_n]$
for all $g \in S$ **do**
  if $g \in PI$ **then**
    continue
  else
    Let $\text{Pred}(g) = \{p \in V \mid (p \rightarrow g) \in E\}$
    $L(g) \leftarrow 1 + \max\{L(p) \mid p \in \text{Pred}(g)\}$
  end if
end for
return Level assignment $L(g)$ for all $g \in V$

IV. The Proposed MAGIC-Based Wallace Tree Multiplier

A. Architectural Design

Fig. 4 illustrates the structure of a 4×4 Wallace Tree multiplier, where intermediate computations C_{ij} and S_{ij} are labeled for various i and j. Multiplying two 4-bit numbers generates four shifted partial products, which are then summed to yield an 8-bit result. At each reduction stage, full adders (or half adders) compress three (or two) bits into a sum and carry, which propagate to the next level. Partial products P_{ij} are generated using AND gates and grouped into sums S_{ij} and carries C_{ij} as shown in Fig. 4. The critical path of a full adder (defined as the longest delay through logic gates) determines the overall delay. Let t_{XOR}, t_{AND}, and t_{OR} denote the delays of XOR, AND, and OR gates, respectively. The *Sum* output delay is $t_S = 2t_{\text{XOR}}$, and the *Carry* delay is $t_C = t_{\text{AND}} + t_{\text{OR}}$. Since XOR gates are typically slower, the overall delay is approximated as $t_{\text{critical}} \approx 2t_{\text{XOR}}$. A 3:2 compressor, functionally equivalent to a full adder, can offer reduced delay and lower power consumption.

Two optimized CMOS-based 3:2 compressor architectures are proposed in [10], as illustrated in Fig. 5. The first design (Fig. 5(a)) utilizes two XOR gates and a $2:1$ multiplexer, resulting in a critical path delay equivalent to two XOR gates. The second design (Fig. 5(b)) employs XOR-XNOR modules along with two multiplexers, achieving a shorter critical path comprising one XOR-XNOR module and a single multiplexer through improved internal signal routing. Both compressor architectures have been synthesized and successfully mapped to memristive crossbar arrays using the MAGIC-based methodology presented in [11]. As shown in Table I, the second design outperforms both the conventional full adder and the first compressor design by achieving lower total latency (L_T) (calculated as the sum of read, write and evaluation latencies) and requiring fewer memristors (#Mem), while maintaining the same crossbar size (CBS). Accordingly, we adopt the second design for partial product reduction

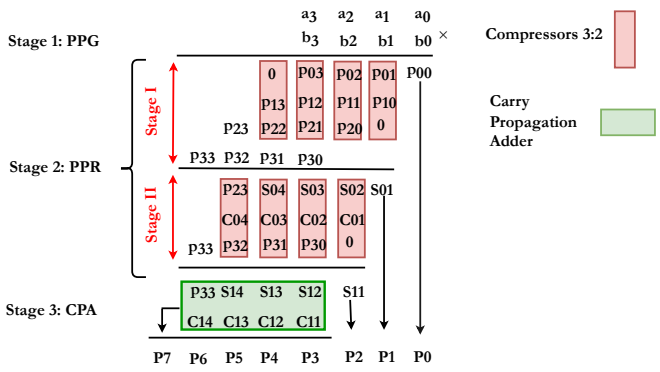

Fig. 4. The schematic of 4×4 Wallace tree multiplier.

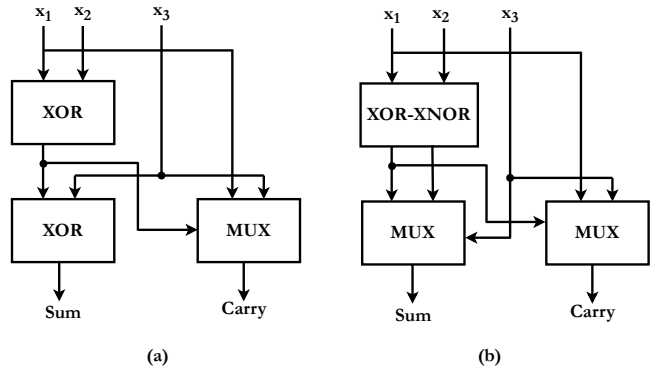

Fig. 5. Design variants of the 3:2 compressor proposed in [10], (a) Design 1, (b) Design 2.

stage in the Wallace Tree multiplier. The implemented logic functions are as follows [10]:

$$Sum = (x_1 \oplus x_2) \cdot \overline{x_3} + \overline{(x_1 \oplus x_2)} \cdot x_3 \quad (2)$$

$$Carry = (x_1 \oplus x_2) \cdot x_3 + \overline{(x_1 \oplus x_2)} \cdot x_1 \quad (3)$$

As illustrated in Fig. 4, the partial product reduction phase (Stage 2, consisting of two sub-stages I and II) employs 3:2 compressors (red rectangles) to perform parallel additions, where sums and carries propagate to the next sub-stage. In sub-stage II, the non-grouped partial products (P_{ij}) are added to the sums and carries from sub-stage I. Each sub-stage executes four parallel additions, and the final stage (Stage 3) completes the operation using a 4-bit adder (indicated by the green rectangle).

TABLE I
MAGIC-BASED SYNTHESIS AND MAPPING RESULTS FOR THE FULL ADDER AND 3:2 COMPRESSOR DESIGNS FROM [10]

Design	L_T	#Mem	CBS
Conventional Full Adder	37	48	4x21
3:2 Compressor (Design 1)	36	42	4x21
3:2 Compressor (Design 2)	34	39	4x21

L_T : total latency, #Mem : number of memristors, CBS : crossbar size.

B. Micro-operation Generation of MAGIC-based 3:2 Compressor

Figure 6 presents the MAGIC-based implementation of the 3:2 compressor used in the partial product reduction stage of the proposed Wallace tree multiplier. It includes a NOR gate tree as shown in Fig. 6(a) generated using the proposed ASAP scheduling algorithm. The corresponding crossbar-level realization is illustrated in Fig. 6(b), which highlights the mapping of logic operations across the crossbar. The associated micro-operations are provided in Listing 1, where the netlist is organized into seven topologically sorted levels, with each line commented using the # symbol for clarity. The compressor operates without any intermediate buffer, taking three inputs ($x1$, $x2$, and $x3$) and generating two outputs: carry, located at crossbar position 0×20, and sum, at 1×14 in the crossbar. The implementation uses a 4×21 crossbar and completes in 34 cycles, comprising 13 read cycles, 14 write cycles, and 7 evaluation cycles. The 4×4 Wallace tree multiplier integrates eight such 3:2 compressors organized across two hierarchical reduction stages, as depicted in Fig. 4. This demonstrates the scalability of the proposed methodology, enabling systematic and efficient implementation of larger bit-width multipliers by composing modular compressor blocks within a crossbar framework. The micro-operations for the 3:2 compressor, implemented in MAGIC logic from a NOR2-based netlist, are described below:

1) At Level 0, three primary inputs are inverted, and a NOR2 gate operates on two of them. To implement this, NOT gates are placed in rows 0 to 2, each processing one of the inputs $/X_1$, $/X_2$, and $/X_3$, using columns 0 and 1 for input and column 2 for output. A NOR2 gate is placed in row 3 to compute the NOR of $/X_1$ and $/X_2$, sharing the same columns for input and output. All gates share columns to enable parallel execution and are evaluated simultaneously once mapped.
2) At Level 1, two NOR2 gates take the primary inputs along with the memristor values from the previous level. Specifically, the first gate reads the primary input $/X_2$ and the memristor value in row 0, column 2 ($\neg X_1$), while the second gate reads the primary input $/X_1$ and the memristor value at row 1, column 2 ($\neg X_2$). Once the input mapping is complete, each gate is evaluated.
3) Level 2 contains a single NOR2 gate that reads inputs from 0×5 and 1×5. The output memristor is initialized to `True`, and the gate evaluation follows.
4) Level 3 consists of single NOT and NOR2 gates. The first gate takes its input from 0×8, and the second gate takes its inputs from 0×8 and primary input $/X_3$. After initialization, the output memristors are initialized to `True`, and the gates are evaluated accordingly.
5) Level 4 consists of two NOR2 gates. The first gate takes inputs from 2×2 and 0×10, then copies them to location 0×12 and 0×13. The second gate reads inputs from 3×2 and 1×11, and writes them to 1×12 and 1×13. Output memristors are initialized to `True`, and the gates are evaluated accordingly.
6) At Level 5, a single NOR gate is used. It reads its input from 0×14 and 1x11, then writes to 0×15 and 0×16. Once initialized, the gate proceeds to evaluation.
7) Level 6 consists of a single NOT gate, which reads its input from 0×17 and then writes to 0×18 and 0×19.

The output memristors are initialized to `True`, and gate evaluation follows.

```
name CMP_3_2
input X1, X2, X3
output Sum, Carry

Outputs are placed at:
Carry -> 0x20
Sum   -> 1x14

Buffers are placed at:

#Level 0
0 False 0 /X1 1 /X1 2 True
1 False 0 /X2 1 /X2 2 True
2 False 0 /X3 1 /X3 2 True
3 False 0 /X1 1 /X2 2 True
#Level 1
0 False 3 /X2 4 0x2 5 True
1 False 3 /X1 4 1x2 5 True
#Level 2
0 False 6 0x5 7 1x5 8 True
#Level 3
0 False 9 0x8 10 0x8 11 True
1 False 9 /X3 10 0x8 11 True
#Level 4
0 False 12 2x2 13 0x10 14 True
1 False 12 3x2 13 1x11 14 True
#Level 5
0 False 15 0x14 16 1x11 17 True
#Level 6
0 False 18 0x17 19 0x17 20 True
```

Listing 1. Compressor 3:2 Mico-Operation File

V. Experimental Evaluation

This section presents the experimental results. The circuits are designed in Verilog, synthesized using the ABC tool [26], and subsequently mapped using the proposed tool from [11]. All experiments are conducted on a system with an Intel i7-4750U CPU (1.70 GHz) and 40 GB of RAM.

A. Benchmarking Synthesis

Table II summarizes the synthesis and mapping results for seven unsigned Wallace Tree multiplier architectures across multiple bit-widths (4 to 64 bits). The evaluated designs include: *Ripple Carry (RC)*, *Carry Look-Ahead (CL)*, *Lander-Fischer (LF)*, *Kogge-Stone (KS)*, *Brent-Kung (BK)*, *Carry-Skip (CK)*, and *Serial Prefix Adder (SE)*. Each design is analyzed in terms of read latency (L_R), write latency (L_W), evaluation latency (L_E), and total latency ($L_T = L_R + L_W + L_E$), along with the number of memristors (`#Mem`) and crossbar size (`CBS`). Let G denote the total number of gates in the NOT/NOR netlist and L the number of logic levels. In this context, *reading* latency refers to sequentially accessing all G gates in the netlist, which requires G cycles in total. *Writing* latency corresponds to initializing the outputs of all gates and writing all gate inputs in parallel across levels, leading to $2L$ cycles. Finally, *evaluation* latency denotes the actual computation of logic values across the crossbar, which proceeds level by level and therefore requires L cycles. All these latencies are expressed in clock cycles.

For 4- to 64-bit designs, WT-LF and WT-CK achieve low latency (228–40,128 cycles), reduced memristor usage (350–34,048), and compact crossbars (9×70–3341×68), demonstrating high efficiency and scalability. In contrast, WT-RC exhibits the highest latency (271–46,720 cycles), the highest memristor count (404–39,168), and the largest crossbar (8×88–4284×87), indicating lower efficiency. With increasing bit-width, WT-LF and WT-CK offer better scalability and lower latency. However, this latency reduction increases the crossbar size in one dimension due to row-wise operations, but using both row- and column-wise logic could improve area efficiency. To better show these results, Fig. 7 compares total latency across all bit-widths and multiplier architectures. The x-axis represents multiplier types, and the y-axis shows latency in cycles, including a zoomed-in view for 4-bit and 8-bit designs. LF and CK multipliers achieve lower latency due to their parallel prefix structures, while RC and CL scale less efficiently under MAGIC-based synthesis due to sequential carry propagation. These results highlight the superior scalability of prefix-based Wallace Tree multipliers.

TABLE II
Synthesis and Mapping Results for Proposed MAGIC-based Unsigned Wallace Tree Multiplier

Designs	Bit Size	Mapping Results					
		L_R	L_W	L_E	L_T	#Mem	CBS
WT-RC	4	153	88	30	271	404	8x88
WT-CL		138	70	24	232	365	10x70
WT-LF		133	70	25	228	350	9x70
WT-KS		153	72	26	251	400	9x72
WT-BK		137	70	25	232	360	9x70
WT-CK		133	70	25	228	350	9x70
WT-SE		153	78	28	259	400	9x78
WT-RC	8	612	176	60	848	612	39x87
WT-CL		552	140	48	740	552	39x87
WT-LF		532	140	50	722	532	30×65
WT-KS		612	144	52	808	612	30×65
WT-BK		548	140	50	738	548	30×65
WT-CK		532	140	50	722	532	30×65
WT-SE		612	156	56	824	612	30×65
WT-RC	16	2448	704	240	3392	2448	190×87
WT-CL		2208	560	192	2960	2208	190×87
WT-LF		2128	560	200	2888	2128	148×68
WT-KS		2448	576	208	3232	2448	148×68
WT-BK		2192	560	200	2952	2192	148×68
WT-CK		2128	560	200	2888	2128	148×68
WT-SE		2448	624	224	3296	2448	148×68
WT-RC	32	9792	2112	720	12624	9792	902×87
WT-CL		8832	1680	576	11088	8832	902×87
WT-LF		8512	1680	600	10792	8512	704×68
WT-KS		9792	1728	624	12144	9792	704×68
WT-BK		8768	1680	600	11048	8768	704×68
WT-CK		8512	1680	600	10792	8512	704×68
WT-SE		9792	1872	672	12336	9792	704×68
WT-RC	64	39168	5632	1920	46720	39168	4284×87
WT-CL		35328	4480	1536	41344	35328	4284×87
WT-LF		34048	4480	1600	40128	34048	3341×68
WT-KS		39168	4608	1664	45440	39168	3341×68
WT-BK		35072	4480	1600	41152	35072	3341×68
WT-CK		34048	4480	1600	40128	34048	3341×68
WT-SE		39168	4992	1792	45952	39168	3341×68

L_R : read latency, L_W : write latency, L_E : evaluation latency, L_T : total latency, #Mem : number of memristors, CBS : crossbar size.

B. Comparison with Existing Multipliers

Table III compares the proposed 4×4 and generalized $n \times n$ MAGIC-based Wallace Tree multipliers, using the LF adder at the CPA stage, with existing in-memory multiplier architectures. The analysis focuses on latency (cycles) and resource usage (cell count), with improvements reported as ratios relative to existing designs. Our architecture supports complete parallelism at each logic level, with the crossbar dimensions determined by the circuit's most complex level. Specifically, the number of rows corresponds to the maximum

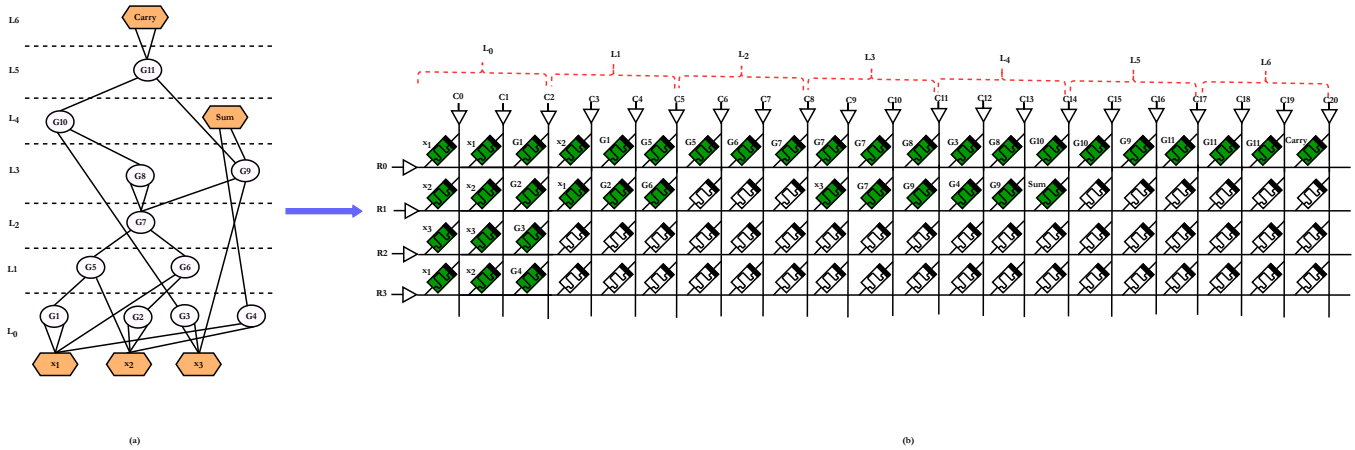

Fig. 6. MAGIC-based 3:2 compressor for the reduction stage of Wallace tree multiplier. (a) NOR gate tree (b) Crossbar mapping.

TABLE III
COMPARISON OF PROPOSED DESIGN WITH EXISTING MULTIPLIERS

Multiplier	Logic Design	4 × 4		Improvement vs		n × n	
		#Cycle	#Cells	#Cycle	#Cells	#Cycle	#Cells
Shift and Add [12]	IMPLY Gate	195	200	1.98	0.88	$15n^2 - 11n - 1$	$15n^2 - 9n - 1$
Semi-Serial Adder Based [13]	IMPLY Gate	102	38	1.04	0.16	$(\log_2 n)(10n + 2) + 4n + 2$	$2n^2 + n + 2$
MultPIM [14]	Minority + NOT	139	49	1.41	0.21	$n(\log_2 n) + 14n + 3$	$14n - 7$
Wallace Tree [6]	Majority Gate	32	128	0.32	0.56	$6\left(\log_2\left(\frac{n^2}{4}\right)\right) + 4\left(\log_2\left(2\left(n - \log_2 n\right)\right)\right) + (n-2)\left(\log_2 n - 2\right) + 10$	$8n^2 + 48\log_2\left(\frac{n}{4}\right)$
Full Precision Fixed Point [15]	NOR Gate	158	75	1.61	0.33	$13n^2 - 14n + 6$	$20n - 5$
Wallace Tree (WT-LF) SIMPLER [16]	NOR Gate	140	144	1.42	0.64	NA	NA
Conventional Design	NOR Gate	132	405	1.3	1.8	$(\sum_{l=0}^{L-1} G_l) + 4L$	$\left(\max_{0\le l<L}(G_l)\right) \times (L \times 3)$
Proposed Design	NOR Gate	98	225	-	-	$(\sum_{l=0}^{L-1} G_l) + 4L$	$\left(\max_{0\le l<L}(G_l)\right) \times (L \times 3)$

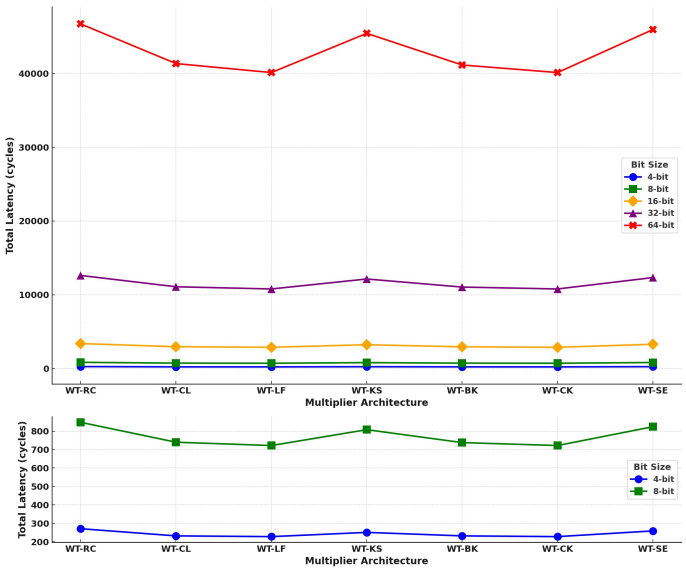

Fig. 7. Latency comparison across different unsigned Wallace tree multiplier.

number of gates in any logic level. Each gate occupies $n + 1$ columns, n for inputs and one for output, where n represents the maximum gate fan-in. For a NOT/NOR-based netlist with G gates distributed across L levels, the total number of read cycles is $\sum_{l=0}^{L-1} G_l$, where G_l denotes the number of gates at level l. At each level, gate inputs and outputs are written to the crossbar simultaneously and logic evaluation is completed in L cycles. Compared to IMPLY-based designs such as Shift-and-Add [12] (195 cycles) and the Semi-Serial Adder [13] (102 cycles) for 4×4 multiplication, which inherently process bits sequentially, our fully parallel implementation achieves significantly lower latency at 98 cycles, albeit with increased area due to the larger crossbar.

Compared to MultPIM [14], which uses a minority + NOT design and completes execution in 139 cycles using 49 cells, our approach achieves significantly faster performance. Although majority-based Wallace Tree designs [6] demonstrate low latency (32 cycles), they operate at the logic abstraction level and do not account for crossbar-specific constraints. In contrast, our method provides a fully mappable micro-operation-level implementation. Compared to other NOR-based designs such as Full-Precision Fixed Point [15] (158 cycles, 75 cells) and the SIMPLER-based Wallace Tree [16] (140 cycles, 144 cells), the proposed method consistently outperforms in both latency and scalability by leveraging fast 3:2 compressors and structured micro-operation scheduling. In addition, we used GENMUL [27], an open source multiplier generator that implements a conventional design that uses full adders for the partial reduction stage of the product. The simulation results demonstrate that our design achieves lower latency and reduced area compared to conventional implementations, confirming its performance benefits. Moreover, while many previous designs exhibit latency growth on the order of $O(n^2)$ or worse latency that escalates rapidly for larger multipliers, our method demonstrates a much more gradual increase in latency with increasing size, enabling efficient implementation of larger multipliers. It achieves 25% lower latency on average, with a trade-off in area overhead.

VI. Conclusion

This paper presents a scalable, high-performance Wallace Tree multiplier for memristor-based in-memory computing using the MAGIC design style. By incorporating an optimized 3:2 compressor and a state-of-the-art mapping tool, the proposed approach addresses key limitations in latency, crossbar efficiency, and scalability in existing approaches. The proposed design approach is evaluated for 4- to 64-bit operands and is directly mapped to executable micro-operations in memristor crossbars, enabling highly parallel in-memory multiplication. Although this parallelism incurs some area cost, it offers significant performance benefits for compute-intensive applications. Experimental results demonstrate a 25% average reduction in latency, making the design suitable for next-generation *processing-in-memory (PIM)* platforms. Future work will explore hybrid row/column logic, gate reuse, and hierarchical mapping strategies to improve area efficiency, as well as extend the architecture to support signed multiplication, floating-point operations, and approximate computing. As memristors evolve, such architectures will help unify logic and memory for energy-efficient design.

Acknowledgement

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2) and SH 1917/1-2.

References

[1] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[2] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta, "A scalable in-memory logic synthesis approach using memristor crossbar," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 355–366, 2017.

[3] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 225–232.

[4] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 7, pp. 1422–1435, 2018.

[5] R. Ben-Hur, R. Rotem, , A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[6] V. Lakshmi, J. Reuben, and V. Pudi, "A novel in-memory wallace tree multiplier architecture using majority logic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 3, pp. 1148–1158, 2021.

[7] D. Esposito, A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra, "Approximate multipliers based on new approximate compressors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4169–4182, 2018.

[8] D. R. Gandhi and N. N. Shah, "Comparative analysis for hardware circuit architecture of wallace tree multiplier," in *2013 international conference on intelligent systems and signal processing (ISSP)*. IEEE, 2013, pp. 1–6.

[9] F. U. D. Farrukh, C. Zhang, Y. Jiang, Z. Zhang, Z. Wang, Z. Wang, and H. Jiang, "Power efficient tiny yolo cnn using reduced hardware resources based on booth multiplier and wallace tree adders," *IEEE Open Journal of Circuits and Systems*, vol. 1, pp. 76–87, 2020.

[10] S. Veeramachaneni, K. M. Krishna, L. Avinash, S. R. Puppala, and M. Srinivas, "Novel architectures for high-speed and low-power 3-2, 4-2 and 5-2 compressors," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. IEEE, 2007, pp. 324–329.

[11] S. Nabipour, K. Datta, L. Weingarten, A. Kole, and R. Drechsler, "Multi-input magic synthesis and verification for in-memory computing design," in *2025 IEEE 55th International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE Computer Society, 2025, pp. 178–183.

[12] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[13] D. Radakovits, N. TaheriNejad, M. Cai, T. Delaroche, and S. Mirabbasi, "A memristive multiplier using semi-serial imply-based adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1495–1506, 2020.

[14] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Multpim: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1647–1651, 2021.

[15] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[16] C. K. Jha and R. Drechsler, "Benchmarking multiplier architectures for magic based in-memory computing," in *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*. IEEE, 2023, pp. 1–5.

[17] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[18] D. Yadav, P. Thangkhiew, and K. Datta, "Look-ahead mapping of boolean functions in memristive crossbar array," *Integration*, vol. 64, pp. 152–162, 2019.

[19] F. Lalchhandama, M. Sahani, V. M. Srinivas, I. Sengupta, and K. Datta, "In-memory computing on resistive ram systems using majority operation," *Journal of Circuits, Systems and Computers*, vol. 31, no. 4, 2022.

[20] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and analysis of approximate compressors for multiplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984–994, 2014.

[21] Hsiao, Jiang, and Yeh, "Design of high-speed low-power 3-2 counter and 4-2 compressor for fast multipliers," *Electronics Letters*, vol. 34, no. 4, pp. 341–343, 1998.

[22] C.-H. Chang, J. Gu, and M. Zhang, "Ultra low-voltage low-power cmos 4-2 and 5-2 compressors for fast arithmetic circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 10, pp. 1985–1997, 2004.

[23] L. E. Guckert, "Memristor-based arithmetic units," Ph.D. dissertation, 2016.

[24] L. Guckert and E. E. Swartzlander, "Dadda multiplier designs using memristors," in *2017 IEEE International Conference on IC Design and Technology (ICICDT)*. IEEE, 2017, pp. 1–4.

[25] A. Mahzoon, D. Große, and R. Drechsler, "Revsca-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1573–1586, 2021.

[26] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[27] A. Mahzoon, D. Große, and R. Drechsler, "Genmul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques: Selected Papers from the 14th International Workshop on Boolean Problems*. Springer, 2021, pp. 177–191.