Performance Evaluation of MAGIC-ReRAM Arithmetic Circuits for Low-Latency In-Memory Computing

Saeideh Nabipour*, Fatemeh Shirinzadeh*,
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, fatemeh.shirinzadeh, abhoy.kole, saeideh.shirinzadeh}@dfki.de, {kdatta, drechsler}@uni-bremen.de

***Abstract*—*In-memory computing (IMC)* with *Resistive Random Access Memory (ReRAM)* crossbars has emerged as a promising solution to overcome the von Neumann bottleneck by enabling computation inside memory arrays. This paper presents a unified benchmarking of multi-bit arithmetic circuits in the MAGIC logic style, encompassing seven adder and three multiplier architectures with operand sizes from 8 to 64 bits. Through a proposed parallel mapping methodology, we achieve latency reductions of up to 26.3× for adders and 361× for multipliers relative to state-of-the-art designs, realized through efficient crossbar-level utilization with modest hardware overhead. Our evaluation shows that *Brent–Kung (BK)* adders are most latency-efficient for larger sizes, while *Serial Prefix (SE)* adders excel for smaller ones and also offer superior hardware efficiency. In addition, Dadda multipliers achieve the lowest total latency, whereas array multipliers provide the best hardware efficiency among multipliers. By systematically quantifying the trade-offs between latency and memristor count, this work offers a detailed design-space exploration of arithmetic units for ReRAM-based IMC, yielding practical insights for future high-performance, memory-centric architectures.**

***Index Terms*—In-Memory Computing, Memristor, MAGIC Design Style, Arithmetic Circuits**

I. Introduction

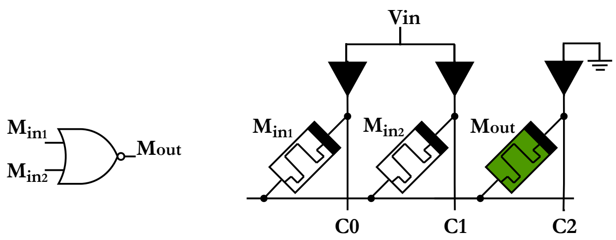
The rising demand for high-performance, energy-efficient computing highlights the limits of the von Neumann architecture, where frequent data transfers between memory and processors create a costly "memory wall." With transistor scaling slowing, new paradigms are required for *Artificial Intelligence (AI)*, big data, and real-time processing. *In-memory computing (IMC)* mitigates this bottleneck by performing computation directly in memory, reducing data movement and enabling massive parallelism. Among IMC technologies, *Resistive RAM (ReRAM)* is especially promising due to its high density, non-volatility, and ability to both store and process data [1]. ReRAM crossbars are thus well-suited for logic and arithmetic tasks. Several IMC logic styles have been proposed, including *implication (IMPLY)* [2], *Memristor Aided LoGIC (MAGIC)* [3], and *Majority (MAJ)* [4]. Among these, *MAGIC* is widely adopted since it realizes universal logic functions using only NOR and NOT gates, with minimal peripheral circuitry—making it suitable for large-scale IMC systems.

Arithmetic units such as adders and multipliers are fundamental building blocks of digital systems, forming the computational backbone of *Central Processing Units (CPUs)*, *Graphics Processing Units (GPUs)*, and accelerators for *Machine Learning (ML)* and *signal processing*. Although there is extensive research on optimizing these units in CMOS technology, their systematic exploration in the context of ReRAM-based IMC remains limited. Recent works have studied adders [5] and multipliers [6] separately using SIMPLER tool [7], establishing valuable baselines but constrained by SIMPLER's serialized execution model. Consequently, open questions remain about scalability, latency, and the true potential of exploiting crossbar-level parallelism in arithmetic unit design.

This paper provides a comprehensive benchmarking study of scalable arithmetic units, including various adders and multipliers, implemented using the MAGIC design style with a state-of-the-art mapping tool [8] that fully exploits parallel crossbar execution. In contrast to SIMPLER's row-reuse methodology, our approach leverages parallel mapping to drastically reduce latency while systematically analyzing hardware trade-offs. We evaluate various architectures and operand sizes, providing practical design guidelines for integrating arithmetic units into ReRAM-based IMC systems. The key contributions of this work are as follows:

- We present the broad comparative study of a scalable arithmetic unit including seven adder and three multiplier architectures implemented with MAGIC-based ReRAM crossbars, covering operand sizes from 8 to 64 bits.
- We demonstrate that our full-parallel mapping methodology achieves up to 26.3× latency reduction for adders and 361× for multipliers compared to SIMPLER, establishing the benefit of exploiting crossbar-level parallelism.
- We provide a quantitative evaluation of designs using two critical metrics—*total latency* and *memristor count*—to clarify speed versus hardware efficiency trade-offs.
- Under the MAGIC design style and the considered evaluation approach, our results show that SE adders are most efficient in hardware and latency for small bit-widths, while BK adders achieve superior latency for large bit-widths with reasonable area. For multipliers, Dadda is latency-optimal and Array is hardware-optimal.
- We systematically explore the design space and provide insights that may serve as practical guidelines for integrating arithmetic circuits into ReRAM-based IMC fabrics, thereby bridging the gap between device-level innovation and system-level deployment.

The remainder of this paper is organized as follows. Section II provides background on ReRAM-based IMC and the MAGIC design style. Section III describes the proposed synthesis and mapping methodology. Section IV presents experimental benchmarking results and comparative analysis.



(a) (b)

Fig. 1. Realization of a 2-input NOR gate using MAGIC

Finally, Section V concludes the paper and outlines directions for future work.

II. Background

A. MAGIC Design Style

MAGIC [3] is a stateful logic design that executes Boolean operations directly within memristor crossbars by encoding logic values as resistance states: a low resistance (R_{on}) represents logic '1', while a high resistance (R_{off}) denotes logic '0'. Unlike traditional CMOS logic, this approach eliminates the need for separate logic circuitry, as both inputs and outputs reside in the memory fabric itself. MAGIC natively supports NOT and NOR gates. Since NOR is a universal gate, arbitrary Boolean functions can be synthesized using combinations of NOT and NOR. Each operation proceeds in two steps: (*i*) *initialization*, where the output memristor is preset to R_{on} for inverting operations or R_{off} for non-inverting ones, and (*ii*) *evaluation*, where an input voltage (V_{in}) is applied across the inputs while the output is grounded, causing the memristor to switch to the correct resistance state that encodes the logical output. Fig. 1 shows a 2-input NOR gate realized with one row and three columns. More complex functions can be systematically constructed from NOT and NOR gates and mapped to crossbars using existing techniques [7], [9].

B. Adder Design

Binary adders are key components in digital arithmetic, enabling operations such as addition, subtraction, multiplication, and address calculation. Most architectures are hierarchically built from *full adders*, which combine two input bits and a carry-in to generate a sum and carry-out for multi-bit addition. The main distinction among adder designs lies in how carry signals are propagated—sequentially, in parallel, or through tree-based schemes to reduce delay. Regardless of the method, the *full adder* remains a universal module for constructing efficient adder structures. A full adder (inputs A, B, C_{in}) can be expressed in NOR form as,

$$Sum = (A' + B' + C_{in}')' + \big((A + B + C_{in})' + C_{\text{out}}\big)' \quad (1)$$

$$C_{\text{out}} = ((A + B)' + (B + C_{in})' + (C_{in} + A)')' \quad (2)$$

Here, Sum and C_{out} denote the sum and carry-out bits. The terms $(A + B + C_{in})'$, $(A + B)'$, and A' correspond to the NOR operations NOR(A, B, C_{in}), NOR(A, B), and NOR(A, A), respectively. To demonstrate ReRAM mapping, Fig. 2(a) shows a 4-bit *ripple carry adder (RCA)* built from cascaded full adders, with one block expressed as a NOR-tree gate in Fig. 2(c). This netlist is scheduled using the *As Late As Possible (ALAP)* algorithm and implemented in a 4×21 memristor crossbar [8], generating the final mapped design in Fig. 2(d). The full adder takes inputs (A, B, C_{in}) and generates outputs Sum and C_{out}, mapped to 0×20 and 1×20, respectively, completing in 14 cycles (7 initialization + 7 evaluation).

C. Multiplier Design

A conventional combinational multiplier operates in three distinct stages [10]:

1) ***Partial Product Generation (PPG):*** In this initial stage, individual partial products are created. For an $n \times n$ unsigned multiplier with input operands $A = \sum_{i=0}^{n-1} A_i 2^i$ and $B = \sum_{i=0}^{n-1} B_i 2^i$, where A_i and B_i represent the i^{th} least significant bits of A and B respectively, this stage calculates $A_i \times B_j$ for all relevant i and j.
2) ***Partial Product Reduction (PPR):*** The generated partial products are then summed together. This stage often involves an array of adders such as the carry-save adder array [11], the Wallace tree [12], and the Dadda tree [13], to efficiently reduce the number of partial products to two rows.
3) ***Carry Propagation Addition (CPA):*** The last stage involves a fast carry-propagate adder to sum the two remaining rows from the accumulation stage, generating the final product.

A variety of architectures for unsigned multiplication have been proposed to optimize speed, area, and power [14]–[16], with *array* and *tree-based* designs being the most common. *Array multipliers* use a regular matrix-like structure that simplifies layout and control but leads to long critical paths for large bit-widths. In contrast, *tree-based* multipliers such as the *Wallace* and *Dadda* trees reduce partial products hierarchically using *full adders (FAs)*, *half adders (HAs)*, or 4:2 compressors, enabling greater parallelism and lower latency. Each architecture presents trade-offs in complexity, performance, and resource use. To illustrate multiplier ReRAM mapping, Fig. 2(b) shows a 4×4 *Wallace tree* built from eight *FAs* across two reduction stages, showcasing how complex arithmetic functions can be constructed from full adder building blocks within the crossbar.

III. MAGIC-Based Mapping Methodology

This section outlines a framework for synthesis and mapping adder and multiplier designs in MAGIC-based memristor crossbars. As shown in Fig. 3, the flow has three main stages: design generation, logic synthesis, and crossbar mapping. In the first stage, the GENMUL framework [17] (Fig. 3 ❶) generates adder and multiplier architectures for operand sizes from 8 to 64 bits, based on four inputs: operand width, partial product generator, accumulator, and final-stage adder type. From these, various adder designs are extracted. The *Ripple Carry (RC)* adder is formally verified using RevSCA-2.0 (Fig. 3 ❷) [18], which applies word-level symbolic algebra. Finally, *combinational equivalence checking (CEC)* is performed with the ABC tool [19], using the verified RC adder as the reference.

In the synthesis stage, all verified adder and multiplier designs are first converted into a flattened format using Yosys synthesis tool as shown in Fig. 3 ❸ [20]. This transformation

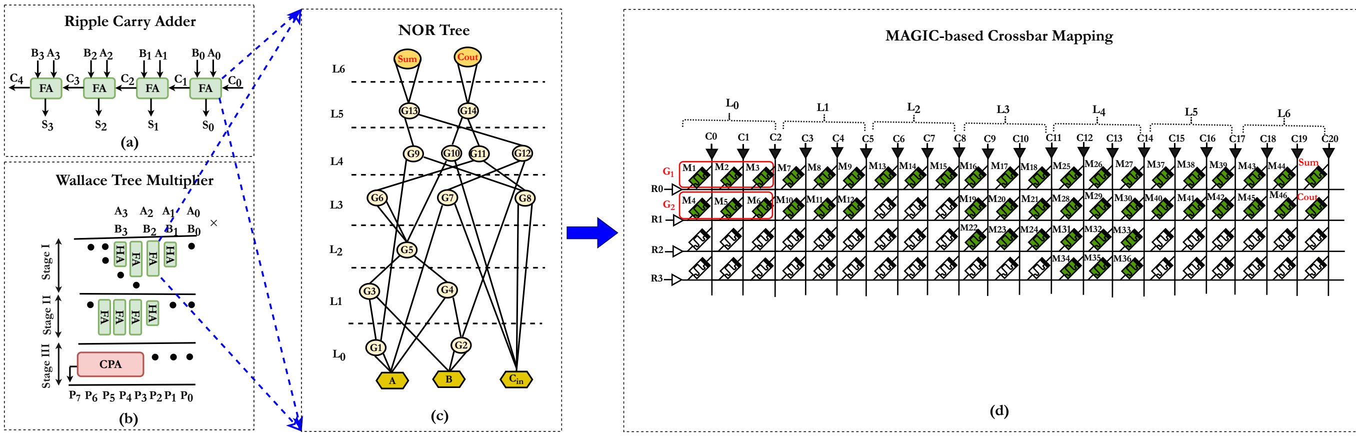


Fig. 2. MAGIC-based Mapping of Full Adder in Arithmetic Circuits. (a) Ripple Carry Adder (b) Wallace Tree Multiplier (c) Nor-tree Gate (d) Crossbar Mapping.

ensures compatibility with the ABC tool as shown in Fig. 3 ❹ for synthesizing the designs into NOR/NOT netlists. During the mapping stage, the MAGIC-based mapper tool proposed in [8] is utilized, as shown in Fig. 3 ❺. This tool incorporates three main algorithms. First, a level-wise scheduling algorithm assigns logic gates to their latest possible execution levels based on input dependencies. The resulting scheduled netlist is then mapped onto a memristor crossbar, enabling row-wise parallel evaluation of logic gates at each level. Finally, the corresponding micro-operations are generated and stored in a `.m` file. The mapping algorithm can evaluate gates with the total number of cycles, the required number of memristors, and the crossbar size as described below:

1) **Total Latency** : The total number of cycles needed to map a Boolean function using the proposed method in [8] is the sum of read, write, and evaluation latencies. The details of each latency are provided below:

$$L_{T_{\text{acc}}} = \sum_{stage \in \{read, write, eval\}} L_{stage} \tag{3}$$

a) **Read Latency**: For a NOT/NOR netlist consisting of G gates partitioned into L levels, reading all gates will require G cycles, so:

$$L_{read} = G \tag{4}$$

b) **Write Latency**: In each level of a NOT/NOR netlist, the outputs of all gates are initialized to logic one in parallel within a single cycle, and all gate inputs at each logic level are simultaneously written and aligned in the same columns, so:

$$L_{write} = 2L \tag{5}$$

c) **Evaluation Latency**: For a NOT/NOR netlist mapped in the crossbar, the evaluation cycles will be equal to the number of levels:

$$L_{eval} = L \tag{6}$$

2) **The number of memristors**: In the mapping process, Boolean functions are realized using only NOR gates within the crossbar to maximize parallelism and performance. A NOT gate is implemented by connecting both inputs of a NOR gate to the same signal, exploiting the identity $\text{NOR}(A, A) = \neg(A \vee A) = \neg A$. Given a NOR-only netlist with G gates and L logic levels, the total number of required memristors is:

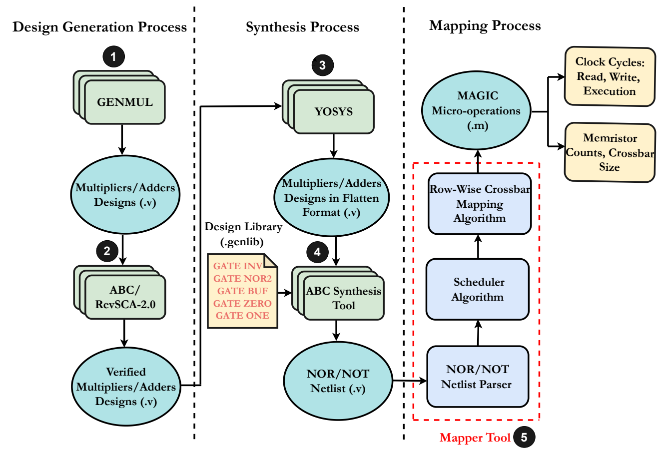


Fig. 3. Synthesis and Mapping Flow.

$$M = \sum_{\ell=1}^{L} 3 \cdot G_\ell \tag{7}$$

where G_L is the number of NOR gates at logic level L, and each NOR gate requires three memristors (two for inputs and one for output). This formulation ensures accurate resource estimation based on the gate distribution across levels.

3) **Crossbar Size (CBS)**: The crossbar dimensions are determined by the circuit's most complex logic level. The number of rows equals the maximum number of gates among all logic levels. Each gate requires $n+1$ columns: n for its inputs and one for its output, where n is the maximum fan-in among all gates. Thus, the total number of columns is equal to the required number of columns per level multiplied by the number of logic levels. So the crossbar size is:

$$CBS = Row \times Column = \left(\max_{0 \le \ell < L} (G_\ell) \right) \times (L \times 3) \tag{8}$$

For additional details on the synthesis and mapping methodology, readers are referred to [8], [21].

IV. Performance Evaluation & Discussion

This section presents the experimental results. The benchmark circuits were designed in Verilog, synthesized using the ABC tool [19], and subsequently mapped using the proposed method described in [8]. Tables I through IV summarize the evaluation results for seven different adder architectures and three types of unsigned multipliers: *Wallace Tree (WT)*, *Dadda Tree (DT)*, and *Array (AR)* multipliers, with bit-widths ranging from 8 to 64 bits. The evaluated adder architectures included *Ripple Carry (RC)*, *Carry Look-Ahead (CL)*, *Ladner-Fischer (LF)*, *Kogge-Stone (KS)*, *Brent-Kung (BK)*, *Carry-Skip (CK)*, and *Serial Prefix (SE)*. These were used independently and as the final stage in multiplier designs. In each table, the first two columns indicate the design name and the bit size. The next four columns report the synthesis results obtained using the proposed method in [8], namely the accurate total latency $L_{T_{\text{acc}}}$ (Eq. 3), the total latency L_T (calculated as $L_T = L_{\text{write}} + L_{\text{eval}}$), the number of memristors $\#M$ (Eq. 7), and the crossbar size CBS (Eq. 8). The subsequent three columns list the total latency (C_T), the number of gates ($\#G$), and the number of memristors ($\#M_S$), as reported by the SIMPLER tool [7] for NOR2 netlists, and applied in [5], [6]. Finally, the last two columns compare the results of the proposed method with SIMPLER by presenting the ratio of latency and crossbar size. All experiments were carried out on a system with an Intel® Core™ i7 1.70 GHz processor and 40 GB of RAM.

TABLE I
Synthesis and Mapping Results for MAGIC-based Adders

Designs	Bit Size	This work $L_{T_{\text{acc}}}$	This work L_T	This work #M	This work CBS	[5] C_T	[5] #G	[5] $\#M_S$	Impr. over [5] $\frac{C_T}{L_T}$	Impr. over [5] $\frac{\#M_S}{\#M}$
RC	8	217	113	281	16x83	94	91	50	0.83	0.17
CL		175	70	284	15x52	111	107	50	1.59	0.17
LF		170	54	302	12x40	106	102	50	1.96	0.16
KS		202	53	386	15x39	139	133	50	2.62	0.12
BK		168	58	287	11x43	100	97	50	1.72	0.17
CK		217	113	281	16x83	93	90	50	0.82	0.17
SE		**157**	74	**218**	9x55	85	83	50	1.15	0.22
RC	16	449	233	585	32x171	192	187	75	0.82	0.12
CL		351	134	588	31x100	325	310	75	2.43	0.12
LF		332	62	706	28x46	253	245	75	4.08	0.10
KS		431	58	970	38x43	338	322	75	5.83	0.07
BK		317	74	634	21x55	228	221	75	3.08	0.11
CK		449	233	585	32x171	191	186	75	0.82	0.12
SE		**309**	138	**450**	17x103	176	171	75	1.28	0.16
RC	32	913	473	1193	64x347	384	379	150	0.81	0.12
CL		703	262	1196	63x196	897	868	150	3.42	0.12
LF		693	78	1605	58x58	555	547	150	7.12	0.09
KS		985	69	2376	86x51	817	796	150	11.84	0.06
BK		**598**	90	1327	41x67	479	472	150	5.32	0.11
CK		913	473	1193	64x347	358	353	150	0.76	0.12
SE		613	266	**914**	33x199	351	347	150	1.32	0.16
RC	64	1841	953	2409	128x699	769	763	275	0.81	0.11
CL		1407	518	2412	127x388	2209	2152	275	4.27	0.11
LF		1438	82	3546	116x61	1225	1214	275	14.94	0.07
KS		2206	74	5544	182x55	1947	1907	275	26.31	0.04
BK		**1161**	106	2754	81x79	994	985	275	9.38	0.09
CK		1841	953	2409	128x699	834	827	275	0.88	0.11
SE		1221	522	**1842**	65x391	705	699	275	1.35	0.14

A. Benchmarking Synthesis for Adders

Table I presents the synthesis and mapping results for seven adder architectures with bit-widths ranging from 8 to 64 bits. The mapping analysis reveals the following:

- **8-bit adders**: The accurate total latency ranges from $L_{T_{\text{acc}}} = 157$ (SE) to 217 (RC/CK), while the number of memristors spans from 218 (SE) to 386 (KS). The SE design demonstrated the lowest accurate total latency (157), the fewest memristors (218), and a compact crossbar size of 9×55, making it the most efficient in both speed and hardware usage.
- **16-bit adders**: The accurate total latency ranges from $L_{T_{\text{acc}}} = 309$ (SE) to 449 (RC/CK), with memristor counts ranging from 450 (SE) to 970 (KS). The SE design achieved the lowest accurate total latency (309), the fewest memristors (450), and a relatively small crossbar size of 17×103.
- **32-bit adders**: The accurate total latency ranges from $L_{T_{\text{acc}}} = 598$ (BK) to 985 (KS), while the number of memristors varies from 914 (SE) to 2,376 (KS). The BK design recorded the lowest accurate total latency (598) with a moderate 41×67 crossbar, while the SE design used the fewest memristors (914) and a compact 33×199 layout.
- **64-bit adders**: The accurate total latency ranges from $L_{T_{\text{acc}}} = 1,161$ (BK) to $2,206$ (KS), and the number of memristors spans from 1,842 (SE) to 5,544 (KS). The BK design offered the lowest accurate total latency $(1,161)$ with an 81×79 crossbar, and the SE design achieved the fewest memristors (1,842) alongside a 65×391 crossbar size.

Overall, within our proposed design space, the SE architecture consistently excelled in hardware efficiency, maintaining the smallest memristor counts and compact crossbar sizes across all bit-width ranges, and providing the lowest accurate total latency for smaller adders. In contrast, for larger designs, the BK architecture leveraged its high degree of parallelism, allowing multiple computation stages to be executed simultaneously. This reduced the critical path length and enabled higher speed. Consequently, BK demonstrated superior latency performance for large adders while still keeping crossbar sizes reasonable, whereas SE retained its advantage in area efficiency.

B. Benchmarking Synthesis for Multipliers

Tables II through IV summarize the synthesis and mapping results for seven unsigned Wallace, Dadda, and Array multiplier architectures across bit-widths 8 to 64 bits. The mapping results show that:

- 8×8 **multipliers:** Across all three multiplier designs, the accurate total latency ranges from $L_{T_{\text{acc}}} = 875$ (DT–SE) to $1,136$ (WT–CK), the number of memristors ranges from #M = 1,483 (all AR designs) to #M = 2,333 (DT–KS), and the crossbar size spans from 20×273 (all AR designs) to 43×194 (WT–CK). The fastest design is DT–SE, the most hardware-efficient is any AR variant, and AR designs also have the most compact crossbars.
- 16×16 **multipliers:** The accurate total latency ranges from $L_{T_{\text{acc}}} = 3,455$ (DT–SE) to $4,177$ (WT–CK), the memristor count ranges from #M = 8,288 (AR–SE) to #M = 10,041 (WT–KS), and the crossbar size ranges from 32×671 (AR–SE) to 158×156 (WT–KS). DT–SE achieves the lowest latency, AR–SE the fewest memristors, and AR–SE also maintains the smallest crossbar.
- 32×32 **multipliers:** The accurate total latency ranges from $L_{T_{\text{acc}}} = 13,796$ (DT–SE) to $17,183$ (WT–CL), the memristor count spans from #M = 34,496 (AR–SE) to #M = 43,682 (WT–CL), and the crossbar size ranges from 64×1391 (AR–SE) to 649×195 (WT–KS). DT–SE leads in speed, AR–SE in hardware efficiency, and AR–SE again has the smallest crossbar.
- 64×64 **multipliers:** The accurate total latency ranges from $L_{T_{\text{acc}}} = 54,685$ (DT–BK) to $71,183$ (WT–CL), the memristor count ranges from #M = 140,672 (AR–SE) to #M = 186,372 (WT–CL), and the crossbar size spans from 128×2831 (AR–SE) to 2700×219 (WT–KS). DT–BK offers the lowest latency, AR–SE offers the fewest memristors, and the smallest crossbar.

TABLE II
SYNTHESIS AND MAPPING RESULTS FOR MAGIC-BASED WALLACE TREE MULTIPLIERS

		This work				[6]			Impr. over [6]	
Designs	**Bit Size**	$L_{T_{\text{acc}}}$	L_T	**#M**	**CBS**	C_T	**#G**	$\#M_S$	$\frac{C_T}{L_T}$	$\frac{\#M_S}{\#M}$
WT-RC		1092	120	2124	29x176	719	691	75	5.99	0.03
WT-CL		1090	96	2274	43x140	815	780	75	8.49	0.03
WT-LF		995	76	2150	43x111	758	725	75	9.97	0.03
WT-KS	8x8	1051	76	2297	42x110	799	762	75	10.51	0.03
WT-BK		996	78	2136	43x115	758	725	75	9.72	0.03
WT-CK		1136	132	2160	26x194	683	658	75	5.17	0.03
WT-SE		996	92	2049	36x135	721	693	75	7.84	0.03
WT-RC		4066	238	9301	63x354	3103	3064	175	13.04	0.01
WT-CL		4044	166	9717	85x246	3470	3409	175	20.90	0.01
WT-LF		3814	102	9517	150x150	3241	3198	175	31.78	0.01
WT-KS	16x16	4027	106	10041	158x156	3423	3369	175	32.30	0.01
WT-BK		3783	110	9379	152x163	3203	3160	175	29.12	0.01
WT-CK		4177	266	9404	58x396	3108	3069	175	11.69	0.01
WT-SE		3799	162	9088	88x240	3108	3069	175	19.20	0.01
WT-RC		15402	484	38079	157x723	12677	12596	350	26.20	0.009
WT-CL		17183	336	43682	350x501	13935	13771	350	41.46	0.008
WT-LF		14813	128	38833	645x189	13192	13100	350	103.06	0.009
WT-KS	32x32	15466	132	40493	649x195	13741	13624	350	104.85	0.008
WT-BK		14616	136	38261	615x201	12999	12909	350	95.58	0.009
WT-CK		15601	524	38327	142x783	12865	12780	350	24.55	0.009
WT-SE		14738	296	37544	206x441	21686	12604	350	73.28	0.009
WT-RC		58812	1014	151175	243x1518	50554	50393	700	49.86	0.004
WT-CL		71183	554	186372	805x827	54506	54166	700	98.38	0.003
WT-LF		57419	154	153085	2537x229	51984	51811	700	337.57	0.004
WT-KS	64x64	59269	148	157900	2700x219	53441	53224	700	361.08	0.002
WT-BK		56876	164	151583	2465x244	51421	53224	700	313.55	0.004
WT-CK		58954	1018	151505	227x1524	51095	50930	700	50.18	0.004
WT-SE		57210	556	149918	437x831	50605	50444	700	91.02	0.004

TABLE III
SYNTHESIS AND MAPPING RESULTS FOR MAGIC-BASED DADDA TREE MULTIPLIERS

		This work				[6]			Impr. over [6]	
Designs	**Bit Size**	$L_{T_{\text{acc}}}$	L_T	**#M**	**CBS**	C_T	**#G**	$\#M_S$	$\frac{C_T}{L_T}$	$\frac{\#M_S}{\#M}$
DT-RC		920	181	1978	35x135	689	664	75	3.81	0.03
DT-CL		943	149	2125	39x111	764	726	75	5.13	0.03
DT-LF		936	149	2096	40x111	719	690	75	4.82	0.03
DT-KS	8x8	1028	149	2333	40x111	739	709	75	4.96	0.03
DT-BK		920	154	2041	40x115	711	683	75	4.62	0.03
DT-CK		985	226	2030	28x169	683	658	75	3.02	0.03
DT-SE		**875**	161	1907	37x120	671	648	75	4.17	0.03
DT-RC		3534	325	8616	94x243	2892	2857	175	8.90	0.02
DT-CL		3658	277	9075	94x207	3163	3119	175	11.42	0.01
DT-LF		3578	197	9051	159x147	2986	2947	175	15.15	0.01
DT-KS	16x16	3870	197	9803	162x147	3134	3090	175	15.91	0.01
DT-BK		3509	205	8847	145x153	2970	2931	175	14.49	0.01
DT-CK		3650	394	8737	76x295	2883	2848	175	7.32	0.02
DT-SE		**3455**	289	8491	95x216	2856	2822	175	9.88	0.02
DT-RC		13926	597	35832	211x447	11874	11797	350	19.89	0.009
DT-CL		14280	541	36927	211x405	12867	12761	350	23.77	0.009
DT-LF		14079	257	37094	594x192	12191	12109	350	47.44	0.009
DT-KS	32x32	14862	250	39129	536x187	12665	12570	350	50.66	0.008
DT-BK		13828	266	36408	529x199	12113	12031	350	45.53	0.009
DT-CK		14123	697	36080	181x522	11899	11822	350	17.07	0.009
DT-SE		**13796**	545	35603	212x408	11792	11716	350	21.63	0.009
DT-RC		54946	1109	144824	419x831	48222	48067	700	43.49	0.004
DT-CL		55489	1061	146435	419x795	51890	51645	700	48.92	0.004
DT-LF		55304	286	147863	2455x214	49339	49173	700	172.53	0.004
DT-KS	64x64	57372	278	153215	2024x208	50763	50565	700	182.56	0.004
DT-BK		**54685**	310	146162	2151x232	48972	48808	700	157.97	0.004
DT-CK		55246	1214	145312	371x910	48375	48212	700	39.84	0.004
DT-SE		54752	1057	144403	421x792	48047	47892	700	45.46	0.004

Since the applied MAGIC-based mapping tool executes all independent operations within a logic level in parallel, architectures with smaller logic depth shorten the critical path, reduce latency, and thus provide a clear performance advantage. Dadda Tree multipliers, especially with the SE design in final stage, minimize sequential addition stages for partial product reduction, fully exploiting parallelism and achieving the lowest latency across most bit-widths. In contrast, the Array Tree SE design offers the best hardware efficiency, requiring the fewest memristors and smallest crossbars, though not always the fastest. Wallace Tree CL designs, with greater logic depth and less favorable mappings, incur the highest latency, memristor counts, and crossbar sizes, making them the least efficient.

Beyond benchmarking, in MAGIC-based IMC designs, latency-area trade-offs strongly affects design choices. For high-performance accelerators (such as AI/ML inference engines), faster circuits—like BK adders and Dadda multipliers—are preferable. In contrast, resource-limited embedded or edge devices benefit more from compact options, such as SE adders and array multipliers. A hybrid IMC system can even combine both: assigning fast MAGIC-based designs to performance-critical tasks, and compact ones to less critical operations. This underlines the importance of design-space exploration when optimizing IMC architectures.

C. Comparison with the Existing MAGIC-based Adders and Multipliers Benchmarks

This section compares the proposed MAGIC-based adder and multiplier designs with previously reported benchmarks in [5], [6] in terms of total latency (measured in cycles) and resource usage (memristor count). Performance improvements are expressed as ratios to existing designs, as summarized in the last two columns of Tables I through IV. The first ratio, $\frac{C_T}{L_T}$, measures the reduction in total latency. The second ratio, $\frac{\#M_S}{\#M}$, compares the number of memristors used in [5], [6] to the number of memristors in our implementation, thereby indicating the relative hardware size. For fairness, we evaluate latency under two scenarios: (i) total latency excluding read latency (L_T), and (ii) accurate total latency ($L_{T_{\text{acc}}}$) including read latency. Since the SIMPLER tool used in [5], [6] excludes read latency from its total latency calculation, we adopt the first scenario for direct comparison. In this case, our method demonstrates a significant reduction in latency. When read latency is included (second scenario), the total latency increases, but this provides a more precise and realistic measure of overall execution performance than the SIMPLER-based approach.

TABLE IV
SYNTHESIS AND MAPPING RESULTS FOR MAGIC-BASED ARRAY TREE MULTIPLIERS

		This work				[6]			Impr. over [6]	
Designs	**Bit Size**	$L_{T_{\text{acc}}}$	L_T	**#M**	**CBS**	C_T	**#G**	$\#M_S$	$\frac{C_T}{L_T}$	$\frac{\#M_S}{\#M}$
AR-RC		931	366	1483	20x273	721	693	75	1.97	0.05
AR-CL		931	366	1483	20x273	691	666	75	1.89	0.05
AR-LF		931	366	1483	20x273	689	664	75	1.88	0.05
AR-KS	8x8	931	366	1483	20x273	690	665	75	1.88	0.05
AR-BK		931	366	1483	20x273	688	663	75	1.88	0.05
AR-CK		931	366	1483	20x273	690	665	75	1.88	0.05
AR-SE		931	366	1483	20x273	688	663	75	1.88	0.05
AR-RC		4010	891	8376	45x667	3045	3002	175	3.42	0.02
AR-CL		4086	896	8514	43x671	3018	2976	175	3.37	0.02
AR-LF		4079	912	8495	41x683	3029	2987	175	3.32	0.02
AR-KS	16x16	4150	904	8703	43x677	3032	2989	175	3.35	0.02
AR-BK		4044	904	8422	37x677	3027	2985	175	3.35	0.02
AR-CK		4091	944	8446	38x707	3036	2993	175	3.22	0.02
AR-SE		3984	896	8288	32x671	3025	2983	175	3.37	0.02
AR-RC		14754	1851	34696	93x1387	12749	12653	350	6.89	0.01
AR-CL		14860	1863	34940	88x1396	12793	12695	350	6.87	0.01
AR-LF		14935	1880	35078	88x1409	12728	12632	350	6.77	0.009
AR-KS	32x32	15197	1872	35782	91x1403	12710	12614	350	6.79	0.009
AR-BK		14806	1864	34782	77x1397	12721	12625	350	6.83	0.01
AR-CK		14805	1863	34796	83x1396	12728	12632	350	6.83	0.01
AR-SE		14688	1856	34496	64x1391	12711	12615	350	6.85	0.01
AR-RC		56210	3771	141096	189x2827	52124	51921	700	13.82	0.004
AR-CL		56422	3792	141595	180x2843	52842	52632	700	13.94	0.004
AR-LF		56569	3791	141953	183x2842	52016	51814	700	13.72	0.004
AR-KS	64x64	57408	3800	144111	187x2849	52069	51866	700	13.70	0.004
AR-BK		56334	3784	141350	157x2837	52076	51873	700	13.77	0.004
AR-CK		56316	3783	141334	177x2836	52173	51969	700	13.79	0.004
AR-SE		56064	3776	140672	128x2831	52050	51847	700	13.79	0.004

MAGIC-based Adders: When comparing our designs against the benchmark design in [5], we observe that the KS architecture provides the largest performance gains. Specifically, for 8-bit adders our design achieves up to $2.62\times$ improvement in total latency efficiency; for 16-bit adders, the gain reaches $2.54\times$; for 32-bit adders, the improvement grows to $2.86\times$; and for 64-bit adders, the advantage becomes most pronounced at $26.31\times$. Across all bit-widths, these maximum speedups are consistently achieved by the KS architecture. In terms of hardware size, the ratio $\frac{\#M_S}{\#M}$ for these fastest designs is between 0.04 and 0.12, indicating that our design uses more memristors than the design in [5]. This increase in device count reflects a deliberate performance-driven design trade-off, enabling substantially lower latency and therefore significantly faster execution across all operand sizes.

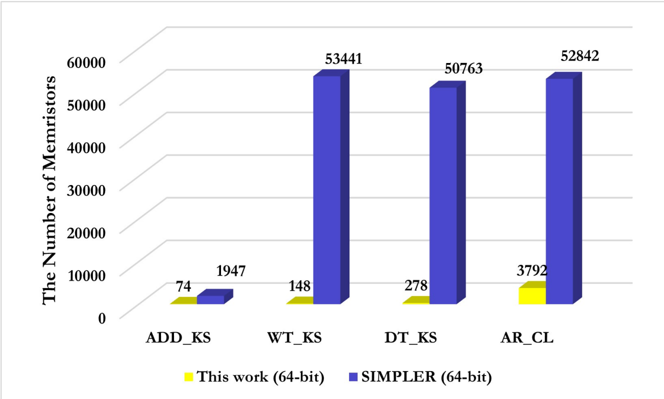


Fig. 4. Comparison of best-case latency improvements over SIMPLER tool, as reported in [5], [6].

MAGIC-based Multipliers: When compared to the benchmark design in [6], our WT multipliers achieve maximum total latency improvements of $9.97\times$, $32.30\times$, $41.46\times$, and $361.08\times$ for 8×8, 16×16, 32×32, and 64×64 multipliers, respectively, with the highest gains consistently achieved by the KS architecture. For DT multipliers, the corresponding improvements are $5.90\times$, $15.91\times$, $23.77\times$, and $182.56\times$, again with the KS design achieving the best results. For AR multipliers, the gains are more modest for smaller operand sizes, with improvements of $1.97\times$, $3.42\times$, $6.89\times$, and $13.82\times$ for 8×8, 16×16, 32×32, and 64×64 multipliers, respectively. Across all three architectures, the ratio $\frac{\#M_S}{\#M}$ for the fastest designs typically lies between 0.002 and 0.05, indicating that our implementations use more memristors than the benchmark design in [6]. This increase in hardware size reflects a performance-driven design trade-off, enabling substantial reductions in execution cycles, with improvements that become especially pronounced for larger operand sizes.

Fig. 4 summarize these results, showing the best-case latency improvements for 64-bit adders and 64×64 multipliers. The x-axes denote the architecture types, while the y-axes report total latency (cycles) and memristor usage. In the most favorable cases, our MAGIC-based designs achieve up to $26.31\times$ speedup for the 64-bit KS adder than [5], $361.08\times$ for the 64×64 Wallace Tree multiplier with KS adder, $182.56\times$ for the 64×64 Dadda Tree multiplier with KS adder, and $13.94\times$ for the 64×64 Array multiplier with CL adder, relative to [6]. Although this parallelism requires more memristors than SIMPLER-based implementations, it represents a deliberate trade-off favoring latency reduction over hardware cost. Importantly, the speed-up scales with operand size: deeper, tree-based designs such as KS adders and Wallace/Dadda multipliers exploit crossbar-level parallelism more effectively than [5], [6].

V. Conclusion

This work provides a comprehensive and comparative evaluation of MAGIC-based arithmetic circuits in ReRAM-based in-memory computing systems. Our results demonstrate that architectural choices strongly shape latency–area trade-offs: Brent–Kung (BK) adders offer the best latency for large operands, while Serial Prefix (SE) adders excel for smaller sizes and offer superior hardware efficiency. Among multipliers, Dadda designs achieve the lowest latency, whereas array multipliers provide the highest hardware efficiency. Compared to prior SIMPLER-based implementations, our fully parallel mapping achieves significant performance improvements, underscoring the importance of exploiting efficient crossbar-level utilization.

Beyond demonstrating substantial performance gains, this study establishes a practical methodology for mapping and evaluating arithmetic circuits using emerging memory technologies, and offers design guidelines for scalable IMC systems. Future work includes exploring approximate arithmetic for error-tolerant domains (e.g., AI/ML inference), extending the analysis to energy efficiency, and integrating these designs into larger-scale IMC accelerators. While our evaluation is limited to 64-bit operands, wider designs will require addressing larger crossbar dimensions and control complexity, which can be mitigated by hierarchical or partitioned crossbar designs. Investigating such scalability will be key to enabling system-level adoption of MAGIC-based arithmetic in future computing platforms.

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. Strukov, G.S.Snider, D. Stewart, and R. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[2] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[3] 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.

[4] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 427–432.

[5] C. K. Jha, A. Mahzoon, and R. Drechsler, "Investigating various adder architectures for digital in-memory computing using magic-based memristor design style," in *2022 IEEE International Conference on Emerging Electronics (ICEE)*. IEEE, 2022, pp. 1–4.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[11] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, ser. Oxford series in electrical and computer engineering. Oxford University Press, 2010.

[12] D. Jacobsohn, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, no. 6, pp. 754–754, 1964.

[13] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, pp. 349–356, 1965.

[14] Y. Yu, W. Pan, C. Tang, N. Yin, and Z. Yu, "Design of a high-speed, low-power ptl-cmos hybrid multiplier using critical-path evaluation model," *Electronics*, vol. 13, no. 7, p. 1284, 2024.

[15] P. Balasubramanian, D. L. Maskell, and N. Mastorakis, "Speed, energy and area optimized early output quasi-delay-insensitive array multipliers," *Plos one*, vol. 15, no. 2, p. e0228343, 2020.

[16] A. Rafiq and M. Jenihhin, "An optimized design of delay-and energy-efficient booth multiplier," *e-Prime-Advances in Electrical Engineering, Electronics and Energy*, vol. 9, p. 100698, 2024.

[17] 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.

[18] ——, "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.

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

[20] C. Wolf *et al.*, "Yosys open synthesis suite," 2016.

[21] L. Weingarten, K. Datta, A. Kole, M. Ambalapat, S. Nabipour, I. S. Gupta, and R. Drechsler, "A complete synthesis and verification approach for magic based in-memory computing," in *Proceedings of the International Conference on Design, Test & Technology of Integrated Systems (DTTIS)*, Athens, Greece, 2025.