

Depth Optimization of Binary Field $GF(2^m)$ Multipliers using MAGIC-Based 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—Binary field $GF(2^m)$ multiplication is a key operation in cryptographic and error-correcting applications, including elliptic-curve cryptography, coding theory, and selected post-quantum schemes. *Memristor-Aided Logic (MAGIC)* enables computation directly within memristive crossbars, potentially reducing data-movement overhead while introducing execution constraints due to logic dependencies and resource limitations. This paper presents an automated framework for implementing $GF(2^m)$ multipliers on MAGIC-based *In-Memory Computing (IMC)* architectures. To reduce arithmetic dependency depth, a *Reduction-Aware (RA)* recursive Karatsuba formulation is proposed that integrates recombination and modular reduction into a unified accumulation network while exploiting XOR sharing and balanced XOR realization. Furthermore, a *Dependency-Aware Cell Reuse (DACR)* mapping strategy improves logical crossbar utilization through reclamation and reuse of intermediate cells. The framework integrates arithmetic generation, logic synthesis, and MAGIC-based mapping, and is evaluated for field sizes ranging from $GF(2^{16})$ to $GF(2^{1024})$. Experimental results generally show substantial reductions in logical execution cycles and allocated memristor count compared with Schoolbook multiplication, particularly for larger fields. These results highlight arithmetic dependency-depth reduction as a key optimization objective for large-field $GF(2^m)$ multipliers under the adopted MAGIC mapping model.

Index Terms—In-Memory Computing, MAGIC, $GF(2^m)$ Multiplication, Karatsuba Multiplication, Cryptographic Accelerators

I. INTRODUCTION

The increasing cost of data movement between memory and processing units has become a major performance and energy bottleneck in modern computing systems, giving rise to the well-known memory wall [1], [2]. To address this challenge, *In-Memory Computing (IMC)* performs computation directly within memory arrays. Among emerging IMC technologies, *Memristor-Aided Logic (MAGIC)* [3] enables functionally complete computation using NOR and NOT operations executed directly inside *Resistive Random-Access Memory (ReRAM)* crossbars.

Many cryptographic and coding applications, including *Elliptic Curve Cryptography (ECC)*, error-correcting codes, and emerging *Post-Quantum Cryptography (PQC)* schemes, rely heavily on arithmetic over binary extension fields $GF(2^m)$ [4]–[7]. Among these operations, multiplication dominates the computational complexity and is repeatedly executed during encryption, signature generation, decoding, and key-exchange procedures. Such computations generate a large number of intermediate operands and partial products that must be stored, transferred, and processed throughout execution. In conventional ASIC and FPGA accelerators, this results in substan-

tial data movement between memory and computation units, contributing to both performance and energy overheads. Consequently, implementing $GF(2^m)$ arithmetic directly within memristive memory arrays has the potential to improve both performance and energy efficiency by reducing the frequent movement of operands and intermediate results between memory and computation units. Among stateful IMC paradigms, MAGIC is particularly attractive for $GF(2^m)$ multiplication because independent logic operations can be evaluated in parallel within a ReRAM crossbar, making execution latency primarily dependent on logic depth.

However, most existing $GF(2^m)$ multiplier architectures were originally developed for conventional digital platforms, such as ASICs and FPGAs, where optimization primarily targets area, gate count, and critical-path delay. These objectives do not directly align with MAGIC-based in-memory computing, where execution latency is largely determined by logic dependency depth and hardware efficiency depends on crossbar resource utilization. Consequently, architectures that are efficient on conventional digital platforms are not necessarily optimal for memristive crossbars. To address this challenge, we propose a dependency-aware framework for MAGIC-based $GF(2^m)$ multiplication. The framework targets dependency reduction at both the arithmetic and mapping levels. At the arithmetic level, a *Reduction-Aware (RA)* optimization integrates recursive Karatsuba decomposition, reduction-integrated accumulation, shared XOR extraction, and balanced XOR realization to reduce arithmetic dependency depth. At the mapping level, a *Dependency-Aware Cell Reuse (DACR)* strategy dynamically reclaims and reuses intermediate storage cells to reduce memristor counts and crossbar size while preserving execution parallelism. The main contributions of this work are:

- A RA Karatsuba optimization framework for $GF(2^m)$ multiplication that minimizes arithmetic dependency depth through reduction-integrated accumulation, XOR sharing, and balanced XOR realization.
- A DACR mapping strategy that reduces memristor utilization and crossbar size without increasing execution latency.
- An automated end-to-end framework for generation, synthesis, and mapping of $GF(2^m)$ multipliers onto MAGIC-based memristive crossbars, evaluated for field sizes ranging from $GF(2^{16})$ to $GF(2^{1024})$.

The rest of this paper is organized as follows. Section II provides the necessary background and related work. Section III presents the proposed RA multiplier architecture. Section IV describes the synthesis and DACR-enhanced parallel row-

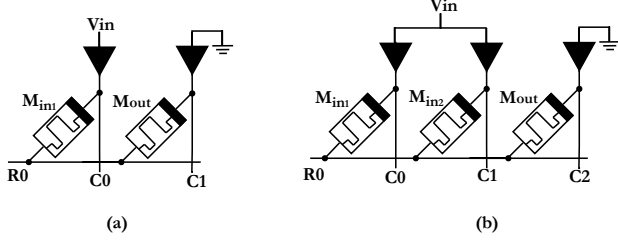


Fig. 1. Realization of NOT and 2-input NOR gate using MAGIC Design [3]

wise MAGIC mapping framework. Section V presents the experimental results, followed by concluding remarks in Section VI.

II. BACKGROUND

A. MAGIC Design Style

MAGIC [3] enables stateful logic computation in memristor-based IMC systems by storing logical values directly in memristor resistance states, where the *Low Resistance State (LRS)* represents logic ‘1’ and the *High Resistance State (HRS)* represents logic ‘0’. Computation is performed directly within ReRAM crossbars using functionally complete NOR and NOT operations, as illustrated in Fig. 1(a) and (b). Before execution, the output memristor is initialized to logic ‘1’, and the applied input voltages determine whether its state remains unchanged or switches to logic ‘0’. Outputs can be reused as inputs to subsequent operations, enabling the realization of any Boolean function [8]–[10].

B. Galois Field Arithmetic

A *Galois Field* $GF(2^m)$ is a finite field widely used in digital systems due to its compatibility with binary logic [11]–[13]. An element is represented in polynomial basis form as

$$A = \sum_{i=0}^{m-1} a_i \alpha^i, \quad a_i \in 0, 1, \quad (1)$$

where α is a root of an irreducible polynomial $p(x)$ of degree m . The irreducible polynomial defines the field and is used to reduce multiplication results whose degree exceeds $m-1$. Low-weight irreducible polynomials are commonly preferred because they simplify modular reduction and improve hardware efficiency. The irreducible polynomials used in this work are listed in Table I. A polynomial basis representation is adopted because $GF(2^m)$ arithmetic is naturally expressed using XOR and AND operations, which can be efficiently synthesized into the NOT/NOR logic primitives supported by MAGIC-based in-memory computing.

1) *Galois Field Multiplication*: Multiplication in $GF(2^m)$ is performed by multiplying the operand polynomials and reducing the result modulo an irreducible polynomial $f(\alpha)$ of degree m . For two field elements A and B ,

$$A \cdot B = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \alpha^{i+j}. \quad (2)$$

Higher-degree terms are reduced using

TABLE I
IRREDUCIBLE POLYNOMIALS USED FOR $GF(2^m)$ MULTIPLICATION
[14]–[17].

m	Irreducible polynomial $p(x)$	m	Irreducible polynomial $p(x)$
16	$x^{16} + x^5 + x^3 + x + 1$	32	$x^{32} + x^7 + x^3 + x^2 + 1$
64	$x^{64} + x^4 + x^3 + x + 1$	127	$x^{127} + x + 1$
128	$x^{128} + x^7 + x^2 + x + 1$	163	$x^{163} + x^7 + x^6 + x^3 + 1$
233	$x^{233} + x^{74} + 1$	256	$x^{256} + x^{10} + x^5 + x^2 + 1$
283	$x^{283} + x^{12} + x^7 + x^5 + 1$	409	$x^{409} + x^{87} + 1$
571	$x^{571} + x^{10} + x^5 + x^2 + 1$	1024	$x^{1024} + x^{19} + x^6 + x + 1$

$$\alpha^m \equiv \sum_{k=0}^{m-1} c_k \alpha^k \pmod{f(\alpha)}. \quad (3)$$

Multiplication is the dominant operation in many $GF(2^m)$ -based cryptographic and coding applications. The conventional Schoolbook method generates and accumulates all pairwise partial products, resulting in quadratic complexity. For large field sizes, recursive algorithms such as Karatsuba multiplication [18] are commonly used to reduce computational complexity.

2) *Karatsuba Multiplication*: Karatsuba multiplication [18] reduces multiplication complexity by recursively decomposing the input polynomials into two sub-polynomials. Let

$$f(x) = f_0(x) + f_1(x)x^k, \quad g(x) = g_0(x) + g_1(x)x^k,$$

where $k = \lceil n/2 \rceil$. Defining $\alpha = f_0(x)g_0(x)$, $\beta = f_1(x)g_1(x)$, and $\gamma = (f_0(x) + f_1(x))(g_0(x) + g_1(x))$, the product can be expressed as follows [19]:

$$f(x)g(x) = (1 + x^k)\alpha + x^k\gamma + x^k(1 + x^k)\beta. \quad (4)$$

Equation (4) represents the Karatsuba recombination step, in which the recursively computed partial products α , β , and γ are combined to reconstruct the final product polynomial. In $GF(2^m)$, this recombination is realized primarily through XOR operations and is followed by modular reduction, making these stages a major contributor to logic depth in MAGIC-based implementations. Since addition in $GF(2^m)$ is equivalent to bitwise XOR, only three recursive multiplications (α , β , and γ) are required instead of four. Recursively applying the decomposition reduces the multiplication complexity from $O(n^2)$ to $O(n^{\log_2 3}) \approx O(n^{1.585})$.

C. Related Work

The implementation of arithmetic operations in memristive IMC architectures has attracted significant attention in recent years. One of the earliest demonstrations of finite-field computation using ReRAM-based stateful logic was ReVAMP [20], which established the feasibility of GF arithmetic in IMC systems. However, ReVAMP focused on relatively small field sizes and did not introduce arithmetic-level optimizations for scalable finite-field multiplication. Within MAGIC-based IMC, several studies investigated arithmetic primitives such as adders [21], integer multipliers [22], and scheduling-aware arithmetic mapping [23]. Despite these advances, existing works primarily target conventional integer arithmetic or small finite-field examples. To the best of our knowledge, no prior work has systematically investigated large-field $GF(2^m)$ multipliers in MAGIC-based IMC, nor explored the impact of recursive multiplier decomposition and reduction-network optimization on execution latency and hardware cost. This

work addresses this gap through a comprehensive study of large-field $GF(2^m)$ multipliers over field sizes ranging from $GF(2^{16})$ to $GF(2^{1024})$.

III. PROPOSED REDUCTION-AWARE KARATSUBA APPROACH

Karatsuba multiplication reduces the number of recursive multiplications from four to three at each decomposition level. However, the partial products generated by Karatsuba multiplication must be recombined and subsequently reduced modulo the irreducible polynomial. In $GF(2^m)$, both operations are primarily realized using XOR computations, whose dependency depth increases with recursion. Consequently, the latency benefits of reduced multiplication count may be partially offset by the XOR-dominated recombination and reduction networks in MAGIC-based IMC. To address this limitation, we propose a *Reduction-Aware (RA)* and dependency-optimized realization of recursive Karatsuba multiplication. Unlike the conventional approach, which performs recombination and modular reduction as separate stages, the proposed method integrates reduction information directly into the accumulation process. Let

$$r_i(x) = x^i \bmod p(x) \quad (5)$$

denote the reduced representation of monomial x^i with respect to the irreducible polynomial $p(x)$. Furthermore, let

$$\mathcal{I}_i = t \mid x^t \text{ appears in } r_i(x) \quad (6)$$

denote the set of output coefficients affected by x^i after reduction. Instead of generating the complete unreduced product and subsequently constructing a separate reduction network, each intermediate coefficient is directly accumulated into all output coefficients indexed by $\mathcal{I}_i$. Consequently, recombination and reduction are merged into a single accumulation stage, eliminating redundant XOR operations and shortening dependency chains. The resulting accumulation network is further optimized through shared XOR extraction and balanced XOR realization. Repeated XOR subexpressions are computed once and reused through shared intermediate nodes, while balanced XOR trees reduce the dependency depth of XOR accumulations. Together, these optimizations reduce logic redundancy and dependency depth, which are the primary factors governing execution latency in MAGIC-based IMC architectures.

A. Algorithm and Dependency Analysis

Algorithm 1 summarizes the proposed flow. Recursive Karatsuba decomposition is applied until the minimum decomposition size is reached, where direct Schoolbook multiplication is used. The generated coefficients are accumulated directly into their reduced output locations according to $\mathcal{I}_i$, after which the resulting accumulation network is optimized through shared XOR extraction and balanced XOR realization. While Karatsuba multiplication reduces the number of recursive multiplications, execution latency in MAGIC-based IMC is also influenced by the dependency depth of the XOR-dominated recombination and modular-reduction networks. The dependency depth of the conventional accumulation network, denoted by D_{CA} , can be expressed as

$$D_{CA} = D_{\text{mult}} + D_{\text{recomb}} + D_{\text{reduce}} \quad (7)$$

Algorithm 1 Reduction-Aware Dependency-Optimized Karatsuba

Input: $f(x), g(x) \in GF(2^m)$, irreducible polynomial $p(x)$
Output: $C(x) = f(x)g(x) \bmod p(x)$
if $n \leq n_{\text{leaf}}$ **then**
 | **return** Schoolbook(f, g)
end
 $k \leftarrow \lceil n/2 \rceil$ Decompose $f = f_0 + f_1x^k$ and $g = g_0 + g_1x^k$
 $\alpha \leftarrow K(f_0, g_0)$, $\beta \leftarrow K(f_1, g_1)$, $\gamma \leftarrow K(f_0 + f_1, g_0 + g_1)$
foreach c_i in $\{(1+x^k)\alpha, x^k\gamma, x^k(1+x^k)\beta\}$ **do**
 | Accumulate c_i directly into all C_t with $t \in \mathcal{I}_i$
end
Extract repeated XOR subexpressions and replace them by shared nodes
foreach output coefficient C_t **do**
 | $C_t \leftarrow \text{BalancedXOR}(C_t)$
end
return C

where D_{mult} , D_{recomb} , and D_{reduce} denote the logic depths associated with recursive multiplication, Karatsuba recombination, and modular reduction, respectively. In the proposed approach, recombination and reduction are merged into a single accumulation network and subsequently optimized through shared XOR extraction and balanced XOR realization. The resulting dependency depth can be expressed as

$$D_{RA} = D_{\text{mult}} + D_{\text{opt}} \quad (8)$$

where D_{opt} denotes the depth of the optimized accumulation network after applying reduction-integrated accumulation, XOR sharing, and balanced XOR realization. Since reduction-integrated accumulation eliminates redundant XOR operations and balanced XOR realization shortens XOR dependency chains, D_{opt} is generally smaller than $D_{\text{recomb}} + D_{\text{reduce}}$. Consequently, the proposed approach reduces the dependency depth associated with recombination and reduction, resulting in $D_{RA} < D_{CA}$, a shorter critical path, and lower MAGIC execution latency.

IV. OVERALL METHODOLOGY

This section presents the framework used to generate, synthesize and map $GF(2^m)$ multiplier designs onto MAGIC-based memristor crossbars. As shown in Fig. 2, the methodology consists of three main steps: GF multiplier generation and validation ①, synthesis of these circuits to generate NOT/NOR netlist ②, and finally mapping of generated netlists to extract design metrics ③. Each of these stages is discussed in detail in the next subsections.

A. Design Generation and Validation

The proposed framework automatically generates Verilog specifications of $GF(2^m)$ multipliers for multiple field sizes using Python-based code generation scripts. Starting from the mathematical description of finite-field multiplication and the selected irreducible polynomial, the framework constructs synthesizable RTL implementations corresponding to Schoolbook multiplication, two-way Karatsuba decomposition (Karatsuba-2), three-way Karatsuba decomposition (Karatsuba-3), four-way Karatsuba decomposition (Karatsuba-4), and their RA dependency-optimized variants. To ensure functional correctness, each generated $GF(2^m)$ multiplier was validated using

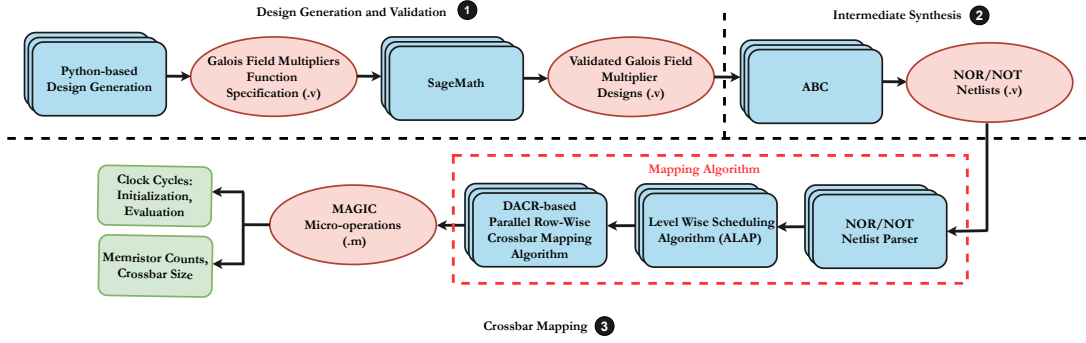


Fig. 2. Overall Framework of the Proposed Synthesis and Mapping Flow for MAGIC-Based $GF(2^m)$ Multipliers.

an automatically generated testbench with 10,000 randomly generated input operand pairs. For a field size m , the input operands were uniformly sampled from the range $[0, 2^m - 1]$. Functional simulations were performed using *Icarus Verilog* (*iverilog*), while the expected multiplication results were independently computed using SageMath-based $GF(2^m)$ arithmetic [24]. The outputs generated by the RTL implementation were compared against the corresponding reference results, and no mismatches were observed during validation.

B. Intermediate Synthesis

In the synthesis stage, as shown in Fig. 2 ②, all validated GF multiplier designs are synthesized using ABC [25] into a NOR-only netlist, where NOT operations are represented as NOR gates with identical inputs. This representation simplifies row-wise MAGIC mapping and increases parallelism. This process preserves functional correctness while enabling execution on memristor-based architectures.

C. DACR-Enhanced MAGIC Mapping Flow

As illustrated in Fig. 2 ③, the synthesized NOR/NOT netlist is first analyzed to extract logic dependencies. Subsequently, an *As Late As Possible* (ALAP) scheduling strategy assigns each gate to its latest feasible execution level while preserving data dependencies. Gates within the same level are independent and can therefore be executed in parallel using the row-wise MAGIC mapping framework of [26], [27]. This process generates the corresponding sequence of MAGIC micro-operations and enables the evaluation of dependency depth, execution latency, memristor utilization, and crossbar size. Although row-wise mapping effectively exploits logic-level parallelism, dedicated storage allocation for intermediate signals can lead to inefficient crossbar utilization. To address this limitation, the proposed *Dependency-Aware Cell Reuse* (DACR) technique is introduced below.

1) *Dependency-Aware Cell Reuse* (DACR): Conventional row-wise MAGIC mapping allocates a dedicated memristor cell to each intermediate signal and retains the allocation throughout execution. Consequently, many cells remain occupied even after their stored values are no longer needed. This increases storage requirements, particularly for large arithmetic circuits with numerous intermediate signals. To improve crossbar utilization, the proposed *Dependency-Aware Cell Reuse* (DACR) technique tracks signal lifetimes and reclaims a level's storage region only after all its outputs have completed their final uses and none is a primary output.

Algorithm 2 summarizes the DACR procedure. First, dependency counts are computed for all intermediate signals in the

synthesized NOR/NOT netlist. Mapping proceeds one logic level at a time, with gates placed in separate rows of a three-column region for parallel execution. Before mapping a new level, eligible regions are examined in chronological order, and the earliest reusable region is selected; otherwise, a new region is allocated. Dependency counts are updated after the complete level has executed. A region becomes reusable only when all signals generated by its assigned level have zero remaining dependencies and none is a primary output. Reclamation at level boundaries prevents reuse of cells still required by the current level. This strategy reduces allocated resources while preserving the ALAP schedule and circuit functionality. Under the assumed parallel reinitialization model, the selected region is prepared within the existing initialization cycle (L_I), leaving total execution latency (L_T) unchanged.

Fig. 3(a) shows a synthesized NOR/NOT logic graph and its equivalent Verilog description, while Fig. 3(b) illustrates the corresponding DACR-based parallel row-wise MAGIC mapping. Regions whose signals have completed their final uses and contain no primary outputs become eligible for reuse by subsequent logic levels. For this example, DACR preserves execution latency at 8 cycles (4 initialization and 4 evaluation cycles) while reducing memristor count from 24 to 18 and logical crossbar dimensions from 2×12 to 2×9 relative to the same mapping without reuse.

V. PERFORMANCE EVALUATION & DISCUSSION

This section evaluates the effectiveness of the proposed RA Karatsuba decomposition for implementing large-field $GF(2^m)$ multipliers on MAGIC-based memristive crossbars. The evaluation considers binary fields ranging from $m = 16$ to $m = 1024$, including conventional cryptographic field sizes and larger fields for assessing scaling behavior. To assess the impact of decomposition granularity, three architectures are investigated: two-way Karatsuba (K2), three-way Karatsuba (K3), and four-way Karatsuba (K4). For each architecture, the proposed RA optimization integrates accumulation and reduction, shared-XOR extraction, and balanced-XOR realization to reduce logic dependencies. All benchmark circuits were automatically generated in Verilog using a Python-based design flow, synthesized into NOR/NOT logic using ABC [25], and subsequently mapped onto memristive crossbars using the DACR-enhanced parallel row-wise MAGIC mapping framework described in Section IV. All experiments were conducted on a machine equipped with an Intel® Core™ i7 processor operating at 1.70 GHz and 40 GB of RAM.

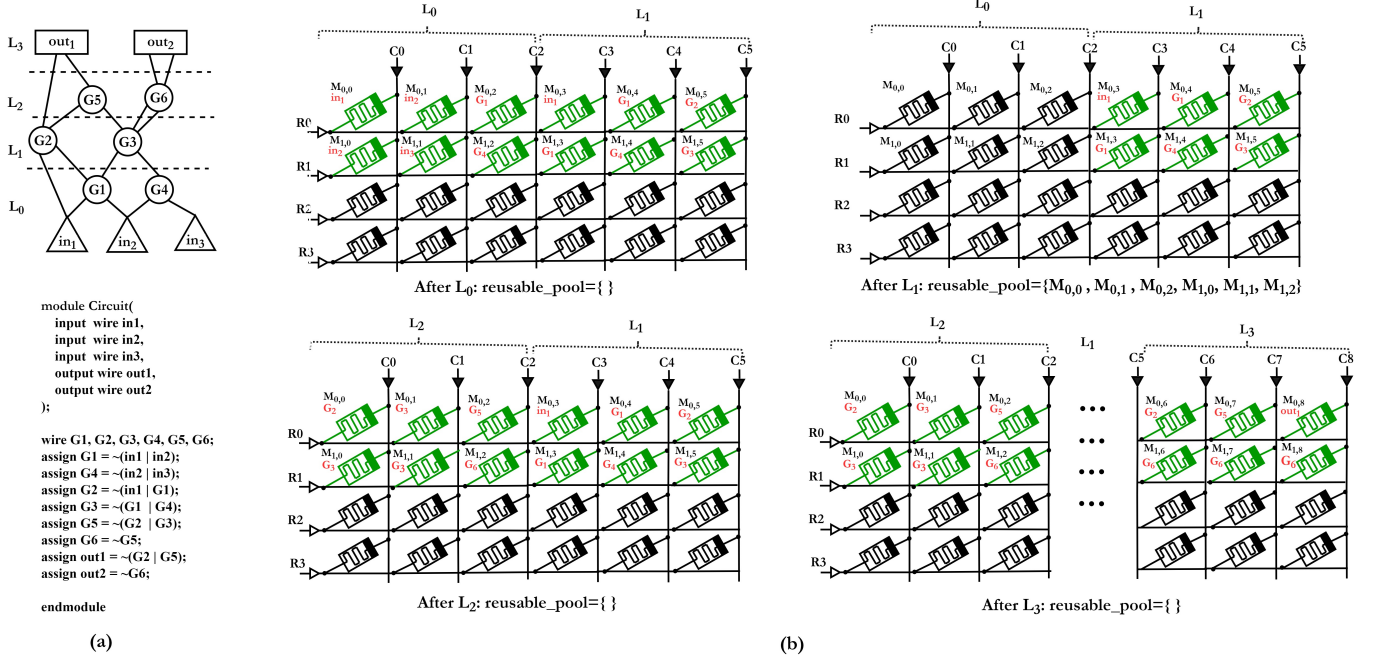


Fig. 3. Simple Example of DACR-Based Parallel Row-Wise Mapping. (a) NOR/NOT graph and equivalent Verilog description. (b) MAGIC-based crossbar mapping.

Algorithm 2 Dependency-Aware Cell Reuse (DACR)

Input: Level-scheduled netlist N , primary outputs PO

Output: Signal-to-cell mapping M

Compute remaining-use counts $DepCount(s)$

$reuse_pool \leftarrow \emptyset$

foreach level L in execution order **do**

Take earliest region R from $reuse_pool$, or allocate a new three-column region

Mark R occupied with outputs of L

foreach gate $g \in L$ **do**

Map $out(g)$ to an output cell in a distinct row of R

end

Initialize R ; execute L in parallel

Update $DepCount$ for all inputs consumed by L

foreach occupied region Q **do**

if all current outputs have $DepCount = 0$ and none is in PO **then**

Mark Q reusable; add to $reuse_pool$

end

end

end

return M

A. Impact of DACR Optimization

Table II evaluates the proposed DACR strategy on Schoolbook $GF(2^m)$ multipliers for field sizes from $m = 16$ to $m = 1024$. The reported metrics are total execution latency (L_T), defined as the sum of initialization and evaluation latencies, memristor count ($\#M$), and logical crossbar dimensions (CBS), compared with the row-wise MAGIC mapping framework of [26], which does not employ cell reuse. DACR preserves the arithmetic structure, logic dependencies, and ALAP execution schedule. Under the assumed parallel

reinitialization model, reclaimed cells are initialized within existing initialization cycles, leaving L_T unchanged. By reclaiming cells associated with expired intermediate signals, DACR reduces allocated storage requirements. Across the evaluated field sizes, memristor count and each crossbar dimension decrease by approximately 20%, corresponding to approximately 36% reduction in the dimension product. The absolute resource savings increase for larger fields. For example, for the $GF(2^{1024})$ Schoolbook multiplier, DACR reduces the memristor count from 21.0M to 16.8M and the crossbar dimensions from 10192×5812 to 8153×4649 . These results demonstrate improved logical resource utilization without additional execution cycles under the adopted mapping model; peripheral and controller overheads are not included.

TABLE II
IMPACT OF DACR ON MAGIC-BASED $GF(2^m)$ SCHOOLBOOK MULTIPLIERS

m	L_T	#M		CBS	
		Without DACR [26]	DACR	Without DACR [26]	DACR
16	153	5577	4461	103×143	82×114
32	248	21437	17149	279×232	223×185
64	440	83578	66862	598×412	478×329
127	790	323042	258433	1230×740	984×592
128	811	331614	265291	1240×759	992×607
163	1039	535430	428344	1582×973	1265×778
233	1436	1086729	869383	2275×1345	1820×1076
256	1581	1318820	1055056	2503×1482	2002×1185
283	1759	1609118	1287294	2774×1648	2219×1318
409	2492	3347252	2677801	4035×2335	3228×1868
571	3487	6538579	5230863	5662×3268	4529×2614
1024	6200	21003484	16802787	10192×5812	8153×4649

B. Evaluation of Proposed Binary Field Multipliers on MAGIC

Table III summarizes the synthesis and MAGIC mapping results for Schoolbook multiplication and the proposed reduction-aware Karatsuba architectures over binary fields

TABLE III
SYNTHESIS AND MAPPING RESULTS OF $GF(2^m)$ MULTIPLIERS

m	Schoolbook					Proposed K2-RA					Proposed K3-RA					Proposed K4-RA				
	L_I	L_E	L_T	#M	CBS	L_I	L_E	L_T	#M	CBS	L_I	L_E	L_T	#M	CBS	L_I	L_E	L_T	#M	CBS
16	114	39	153	4461	82x114	106	36	142	4202	68x106	106	36	142	4551	72x106	124	42	166	4564	67x124
32	185	63	248	17149	223x185	133	45	178	12528	218x133	133	45	178	14024	241x133	124	42	166	13307	242x124
64	329	111	440	66862	478x329	160	54	214	38260	550x160	133	45	178	44446	805x133	169	57	226	41950	527x169
127	592	198	790	258433	984x592	178	60	238	114998	1738x178	160	54	214	132963	2057x160	169	57	226	123677	2130x169
128	607	204	811	265291	992x607	187	63	250	118334	1666x187	169	57	226	136572	2024x169	178	60	238	127393	1961x178
163	778	261	1039	428344	1265x778	214	72	286	184599	2388x214	169	57	226	213619	3410x169	214	72	286	213590	2249x214
233	1076	360	1436	869383	1820x1076	205	69	274	320070	4313x205	178	60	238	370126	5164x178	214	72	286	352067	3794x214
256	1185	396	1581	1055056	2002x1185	214	72	286	360039	4786x214	196	66	262	459961	6149x196	223	75	298	392185	4047x223
283	1318	441	1759	1287294	2219x1318	232	78	310	432771	5428x232	196	66	262	523842	7283x196	232	78	310	479191	5579x232
409	1868	624	2492	2677801	3228x1868	232	78	310	775944	9461x232	205	69	274	961221	13085x205	214	72	286	852781	11686x214
571	2614	873	3487	5230863	4529x2614	277	93	370	1319334	14119x277	205	69	274	1663245	24726x205	259	87	346	1480516	17072x259
1024	4649	1551	6200	16802787	8153x4649	277	93	370	3293087	37781x277	232	78	310	4293995	56030x232	277	93	370	3577642	37364x277

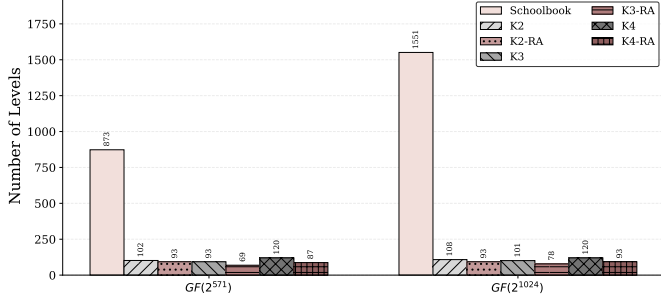


Fig. 4. Logic-depth comparison of Schoolbook, conventional Karatsuba, and proposed RA multipliers for $GF(2^{571})$ and $GF(2^{1024})$.

ranging from $GF(2^{16})$ to $GF(2^{1024})$. The reported metrics include initialization latency (L_I), evaluation latency (L_E), total latency ($L_T = L_I + L_E$), memristor count (#M), and logical crossbar size (CBS). All implementations are mapped using the DACR-enhanced row-wise MAGIC framework.

Compared with Schoolbook multiplication, the proposed architectures generally reduce latency and memristor count. Using the unweighted mean of per-field percentage reductions over the 12 evaluated sizes, K2-RA, K3-RA, and K4-RA reduce total latency by approximately 67.9%, 70.5%, and 67.0%, respectively. K3-RA achieves the lowest or joint-lowest latency for most field sizes, whereas K4-RA is fastest at $m = 32$. At $m = 16$, K4-RA increases latency by 8.5%, and K3-RA and K4-RA slightly increase memristor count. Latency reductions become more pronounced for larger fields.

In terms of mapped storage requirements, K2-RA achieves the lowest memristor count across all evaluated sizes. Using the same averaging method, K2-RA, K3-RA, and K4-RA reduce memristor count by approximately 55.4%, 48.1%, and 51.3%, respectively. Crossbar dimensions exhibit different trade-offs and do not directly represent physical area. Overall, K3-RA generally favors latency, K2-RA minimizes allocated memristors, and K4-RA offers an intermediate latency-resource trade-off for larger fields.

To assess the additional contribution of RA, Fig. 4 compares Schoolbook multiplication, conventional Karatsuba multipliers (K2, K3, and K4), and their corresponding RA-enhanced variants for $GF(2^{571})$ and $GF(2^{1024})$. Conventional Karatsuba decomposition substantially reduces logic depth relative to Schoolbook multiplication, while RA further reduces depth for each evaluated decomposition factor. K3-RA achieves the lowest depth in both fields. These comparisons support the combined benefit of reduction-integrated accumulation, XOR sharing, and balanced XOR realization, although they do not

isolate the contribution of each optimization individually.

VI. CONCLUSION AND FUTURE WORK

This paper presented a framework for implementing large-field $GF(2^m)$ multipliers on MAGIC-based IMC architectures. A *Reduction-Aware (RA)* recursive Karatsuba optimization was proposed to reduce arithmetic dependency depth through integrated accumulation and reduction, shared XOR extraction, and balanced XOR realization. In addition, a *Dependency-Aware Cell Reuse (DACR)* strategy was introduced to improve crossbar resource utilization through reuse of intermediate storage cells. Experimental results for binary fields ranging from $GF(2^{16})$ to $GF(2^{1024})$ demonstrated substantial improvements over Schoolbook multiplication. K3-RA generally achieved the lowest latency, with an average reduction of 70.5% and a maximum of 95%, while K2-RA consistently required the fewest memristors. DACR reduced memristor count and each logical crossbar dimension by approximately 20%, preserving execution cycles under the assumed parallel reinitialization model. Overall, the results identify arithmetic dependency depth as a key optimization objective for MAGIC-based $GF(2^m)$ multiplication. Future work will extend the framework toward complete cryptographic accelerators for ECC and post-quantum cryptography, realistic tiled crossbars, and energy-aware optimization.

ACKNOWLEDGEMENT

This research has been supported by the German Research Foundation (DFG) with project PLiM (DR 287/35-1, DR 287/35-2, and SH 1917/1-2), and by the German Ministry for Research, Technology and Space (BMFTR) with project ExaVerse (grant number 01IW25003).

REFERENCES

- [1] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design & Test*, vol. 34, no. 2, pp. 39–50, 2016.
- [2] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 2002.
- [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] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland, 1977.
- [5] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.
- [6] D. Bhattacharjee and A. Chattopadhyay, "Efficient binary basic linear algebra operations on reram crossbar arrays," in *2017 30th international conference on VLSI design and 17th international conference on embedded systems (VLSID)*. IEEE, 2017, pp. 277–282.
- [7] D. Niu, Y. Xiao, and Y. Xie, "Low power memristor-based reram design with error correcting code," in *17th Asia and South Pacific Design Automation Conference*. IEEE, 2012, pp. 79–84.

- [8] 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.
- [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] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.
- [11] S. Lin and J. Li, *Fundamentals of Classical and Modern Error-Correcting Codes*. Cambridge University Press, 2021.
- [12] S. Nabipour, J. Javidan, and G. Zare Fatin, "Error detection mechanism based on bch decoder and root finding of polynomial over finite fields," *Journal of mathematics and computer science*, vol. 12, no. 04, pp. 271–281, 2014.
- [13] C. Paar and J. Pelzl, *Understanding cryptography*. Springer, 2010, vol. 1.
- [14] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.
- [15] Information Technology Laboratory (National Institute of Standards and Technology), *Digital Signature Standard (DSS)*. US Department of Commerce, National Institute of Standards and Technology, 2013.
- [16] G. Banegas, R. Custódio, and D. Panario, "A new class of irreducible pentanormals for polynomial-based multipliers in binary fields," *Journal of Cryptographic Engineering*, vol. 9, no. 4, pp. 359–373, 2019.
- [17] G. Seroussi, *Table of low-weight binary irreducible polynomials*. Hewlett-Packard Laboratories Palo Alto, California, 1998.
- [18] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digit numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.
- [19] I. van Hoof, "Space-efficient quantum multiplication polynomials for binary finite fields with sub-quadratic toffoli gate count," *Quantum Inf. Comput.*, vol. 20, no. 9&10, pp. 721–735, 2020.
- [20] S. Mandal, D. Bhattacharjee, Y. Tavva, and A. Chattopadhyay, "Reram-based in-memory computation of galois field arithmetic," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2018, pp. 1–6.
- [21] 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.
- [22] 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.
- [23] S. Nabipour, F. Shirinzadeh, K. Datta, A. Kole, S. Shirinzadeh, and R. Drechsler, "Performance evaluation of magic-ReRAM arithmetic circuits for low-latency in-memory computing," in *Nordic Circuits and Systems Conference 2025*, 2025.
- [24] W. Stein *et al.*, "Sage: Open source mathematical software," 2008.
- [25] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.
- [26] 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.
- [27] 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.