A Comprehensive Synthesis and Verification Approach for RRAM-Based Neuromorphic Computing

Fatemeh Shirinzadeh* fatemeh.shirinzadeh@dfki.de
Abhoy Kole* abhoy.kole@dfki.de
Kamalika Datta*‡ kdatta@uni-bremen.de
Saeideh Shirinzadeh*† saeideh.shirinzadeh@dfki.de
Rolf Drechsler*‡ drechsler@uni-bremen.de

*German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
†Fraunhofer Institute for Systems and Innovation Research (ISI), Karlsruhe, Germany
‡Institute of Computer Science, University of Bremen, Germany

***Abstract*—Resistive RAM (RRAM) has emerged as a promising technology for in-memory computing by enabling storage and computation within the same physical substrate. While its analog computation capability, particularly the multiply-accumulate (MAC) operation, has been effectively used in neuromorphic systems, its potential for logic synthesis remains underexplored. Logic synthesis using MAC not only unlocks new efficiency gains but also aligns with hardware already present in neuromorphic accelerators. In this work, we present the first automated framework for evaluating arbitrary Boolean functions on standard RRAM crossbars using highly parallel MAC operations. The proposed method introduces a logic computation core for RRAM-based neuromorphic architectures without requiring additional hardware, leveraging existing peripheral circuitry. To ensure functional correctness, we further integrate a formal verification approach based on equivalence checking via SAT solvers. Experimental results on standard benchmarks demonstrate substantial reductions in computation cycles and improved efficiency compared to existing RRAM-based logic synthesis methods, highlighting the practical potential of MAC-based logic in emerging computing systems.**

***Index Terms*—In-Memory Computing, Logic Synthesis, MAC Operation, Formal Verification.**

I. Introduction

Resistive RAM (RRAM) is a non-volatile memory technology whose internal resistance can be switched between low and high states to represent binary bits [1]. This abrupt switching property enables performing basic logic operations and therefore allows to unify memory and processing units known as in-memory computing. In addition to logic operations, RRAM allows to perform the *multiply-accumulate (MAC)* operation driven by measurements based on current flowing memory columns with tuned device conductivities. This enables highly parallel computation of matrix-vector multiplication which is essential for neuromorphic structures [2]–[4].

Various automated approaches for logic-in-memory computing with RRAM devices have been proposed. These methods usually exploit different universal logic operations that are executable within RRAM devices such as *material implication(IMPLY)* [5], resistive majority of three *(MAJ)* [6], [7], and *Memristor-Aided LoGIC(MAGIC)* [8].

However, MAC has been partially used for logic computation only in a few approaches despite its advantages for parallelization. In [9], MAC operation is utilized in a BDD-based approach to evaluate logic functions on RRAM crossbar. The approach uses only two crossbar rows and thus cannot explore the full parallelism potential of MAC. More recently, a MAC-based SAT solver was proposed [10] for evaluating *Conjunctive Normal Form (CNF)* representations which is not efficient as a solution for logic synthesis due to high computational complexity.

While various approaches have been proposed for RRAM-based logic-in-memory computation, most current methods depend on universal logic primitives such as IMPLY, MAJ, or MAGIC design. Nevertheless, these approaches often face challenges such as limited scalability due to the large number of required write cycles and inefficient handling of parallelization. The MAC operation, commonly used in memristive architectures, offers a promising alternative by enabling efficient parallel evaluation of Boolean functions within RRAM crossbars. This paper aims at leveraging MAC for logic synthesis while exploring its parallelism potential. To the best of our knowledge, we introduce a novel MAC-based design and mapping approach for the first time, combined with a formal verification framework, to enhance both efficiency and reliability in RRAM-based in-memory computing.

Our proposed design introduces a logic computation core within RRAM-based neuromorphic units, utilizing available peripheral circuitry, sense amplifiers, ADC/DAC, and other required control elements without imposing additional overhead. The approach requires a one-time crossbar initialization procedure, which remains effective for all input variations. While the initialization process contributes to overall latency, our method outperforms state-of-the-art approaches in scenarios requiring frequent evaluations, as it eliminates the need for regular memory refreshing or intermediate value updates.

Beyond the design aspect, this work also addresses the critical challenge of verification in RRAM-based in-memory computing. As in-memory designs grow in complexity, ensuring computational correctness becomes increasingly important. Traditional validation techniques, such as manual inspection or simulation, become impractical for large-scale designs with numerous inputs and outputs. We use equivalence checking to compare the function descriptions of conventional logic netlists with

the micro-operations mapped to the RRAM crossbar, ensuring their functional correctness. While recent studies have explored formal verification for memristive logic primitives in RRAM devices [11], [12], or focused on verifying the OIG netlist of MAC operations [13] but verification techniques specifically targeting MAC-based computation on RRAM crossbars are being performed for the first time in this paper.

As emerging applications such as edge AI, real-time signal processing, and embedded neuromorphic systems demand faster and more energy-efficient computing, RRAM-based in-memory architectures are gaining increasing attention. However, the lack of scalable logic synthesis methods and trustworthy verification flows for such systems remains a significant bottleneck. Existing logic-in-memory schemes often fall short in exploiting parallelism or ensuring functional correctness at scale. Meanwhile, MAC operations remain an underutilized opportunity for logic processing. By enabling reliable and parallel Boolean computation using MAC operations, our work bridges this gap and opens the door to reconfigurable, low-latency, and verifiable computing directly within memory fabrics.

II. Background

A. MAC Operation

RRAM's unique analog computation capabilities distinguish it from other emerging memory technologies, particularly in neural network implementations. These capabilities enable efficient MAC operations, which are crucial for matrix-vector multiplication.

In the crossbar structure shown in Fig. 1, the resistive values of RRAM devices are initialized with $a_{j,k}^{-1}$. By applying voltages $x_1, \ldots, x_n$ to the rows, up to m MAC operations can be performed simultaneously across m crossbar columns.

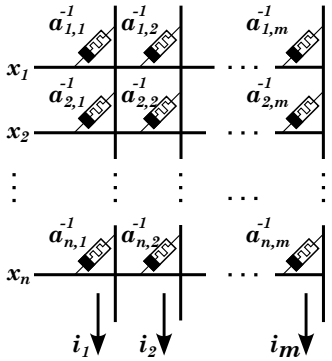


Fig. 1: MAC computation in RRAM crossbar

The outputs are the currents flowing in the corresponding crossbar columns, which is the sum of currents in each RRAM device, i.e. $i_j = \sum_{k=1}^{n} x_k \cdot a_{j,k}$. This can be denoted as $I = XA$, where $I = (i_1, \ldots, i_m)$, $x = (x_1, \ldots, x_n)$ and the matrix A is defined as follows:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \ldots & a_{1,m} \\ a_{2,1} & a_{2,2} & \ldots & \ldots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \ldots & a_{n,m} \end{bmatrix} \quad (1)$$

Thus, m MAC operations, each involving n multiplications, can be computed in parallel within a single cycle. While MAC operations are extensively used in neuromorphic computing with RRAM, their potential for logic applications remains largely untapped. This paper explores their use in computing Boolean functions through our proposed RRAM crossbar mapping approach.

B. Boolean Satisfiability (SAT)

The *Boolean Satisfiability Problem (SAT)* is a fundamental NP-complete problem in computer science, where the goal is to determine whether a Boolean formula can be satisfied by some assignment of its variables [14]. If such an assignment exists, the formula is *satisfiable (SAT)*; otherwise, it is *unsatisfiable (UNSAT)* [15].

To solve combinatorial problems using SAT, formulas are typically converted into *Conjunctive Normal Form (CNF)*, a conjunction of clauses, where each clause is a disjunction of literals. This standard form allows efficient processing by modern SAT solvers, which are widely used in logic synthesis and formal verification.

C. Related Works

Many existing approaches to logic synthesis with RRAM devices rely on gate-level circuits derived from function descriptions such as *Binary Decision Diagrams (BDDs)* [16], *AND-Inverter Graphs (AIGs)* [17], and *Majority-Inverter Graphs (MIGs)* [18], as well as other graph-based representations. In [5], material implication was demonstrated using resistive devices, and a memristive NAND gate capable of realizing any Boolean function was introduced. This foundational work opened new directions for non-von Neumann computing architectures by enabling computation within memory arrays.

MIGs were introduced in [19] as a logic representation using only the majority function and negation, which simplifies the design of logic circuits and field-programmable gate arrays (FPGAs). MIGs are particularly suitable for RRAM-based circuit synthesis since resistive majority operations can be directly executed [6]. A comprehensive synthesis method leveraging BDDs, AIGs, and MIGs for resistive in-memory computing was presented in [20]. This method supports parallel computing on multi-row crossbar architectures and provides alternative implementations to optimize metrics such as the number of RRAM devices, operations, area, and delay. Experimental results demonstrated substantial improvements in area and latency. However, these designs face complexity challenges due to conflicting write operations across multiple devices, requiring additional resources and longer computation times to maintain data integrity.

MAGIC-based logic synthesis was explored in [21], where the authors proposed a method to map multi-output Boolean functions onto RRAM crossbars using NOR gates. The technique included gate scheduling to reduce hardware overhead and achieved notable reductions in cycle count and energy consumption. Nevertheless, the proposed crossbar optimization strategies did not account for sneak path issues, which can compromise system reliability and functionality.

In terms of verification, various in-memory techniques have been developed, primarily targeting Majority- and MAGIC-based logic styles. These methods aim to ensure the correctness of operation sequences generated by automated crossbar mapping tools. For Majority-based designs, formal methods have employed SMT solvers such as CVC4 [22] and Z3 [11], and BBD [23], using intermediate representations such as ReRAM Sequence

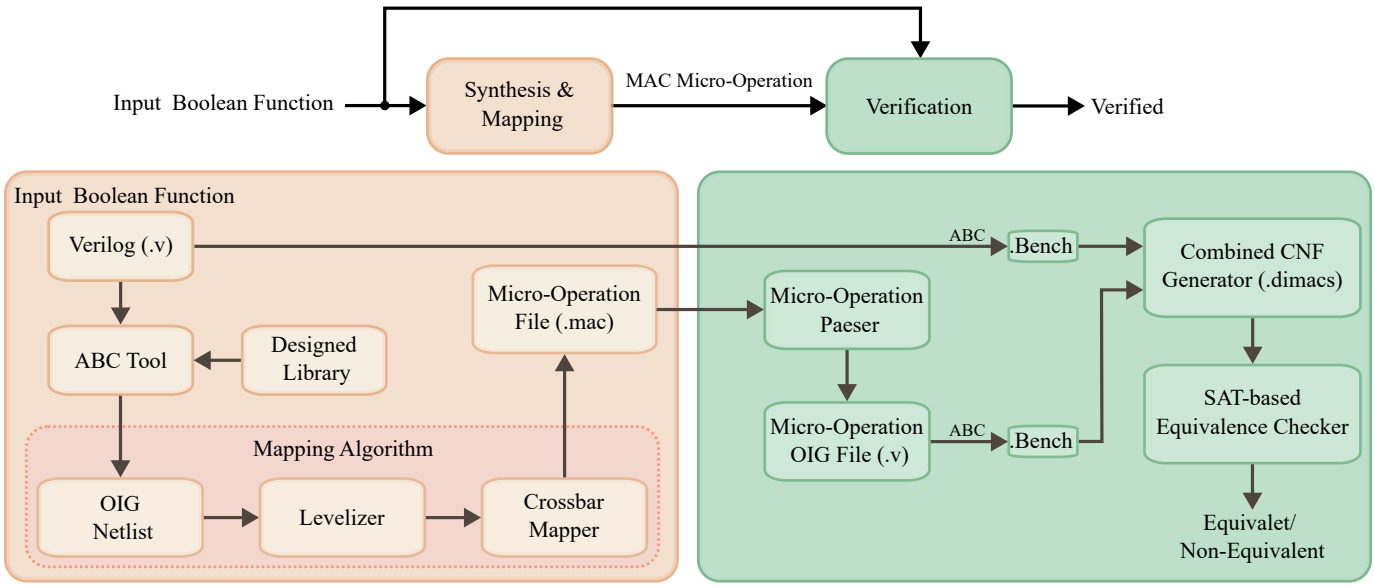


Fig. 2: The overall flow of MAC-based Synthesis and Verification

Graphs [11] and ReRAM Matrices [23]. In contrast, MAGIC-based verification remains relatively underexplored. A recent work, veriSimpler [12], addresses formal verification of NOR-netlists produced by the Simpler mapping method [24], using SAT-based equivalence checking with Z3. While this marks a significant step forward, it does not perform verification at the level of individual mapping operations. Thus, there remains a need for a more holistic synthesis, mapping, and verification framework for MAGIC-based in-memory computing [25].

III. Proposed Mapping and Verification Methodology

In this section, we present our approach for mapping arbitrary Boolean functions to generate MAC micro-operations in RRAM-based crossbar architectures. We also describe the corresponding verification methodology used to ensure the correctness of these micro-operations. An overview of the entire process is illustrated in Fig. 2.

A. MAC-Based Mapping Methodology

The proposed methodology adopts the *sum of products (SoP)* approach for performing MAC operations. To leverage its structural advantages, we employ the *Or-Inverter Graph (OIG)* as the underlying functional representation.

An OIG is a directed acyclic graph composed of three distinct types of nodes. The first type, which has no outgoing edges, represents a terminal node that acts as the primary output. The second type, lacking incoming edges, functions as the primary input. The third type consists of nodes with n incoming edges (where n is 10 or fewer) and a single outgoing edge, representing a Boolean OR operation.

The OR nodes are connected by two types of edges: a regular edge representing the actual functionality and a complementary edge representing its negation. More formally, an OIG is defined as follows:

Definition 1: An *OR Inverter Graph (OIG)* over the primary input variables $X = \{x_1, x_2, \ldots, x_n\}$ and the primary output variables $Y = \{y_1, y_2, \ldots, y_m\}$ is a directed acyclic graph $H = (V, E)$ with the following characteristics:

- A finite set of nodes $V = V_X \cup V_H \cup V_Y$, where V_X and V_Y are terminal nodes that specify the primary input nodes, and primary output nodes, respectively, and $V_H = \{v_{h1}, v_{h2}, \ldots, v_{hk}\}$ are non-terminal nodes representing a logical OR operation.
- An edge $e \in E$ between a source node $u \in V$ and a target node $v \in V$ can be either a regular edge or a complement edge. Specifically, an edge e is represented as $(u, (v \times p))$, where $u \notin V_Y$ and $v \notin V_X$. Here, p denotes the type of edge: $p = 1$ for a regular edge that signifies the actual functionality, and $p = 0$ for a complementing edge that indicates the negation of this functionality.

The depth of the OIG corresponds to its total number of levels. In this paper, the OIG is optimized to minimize depth by prioritizing OR gates with larger fan-ins at lower levels whenever possible.

As shown earlier in Fig. 2, the process begins with a Boolean function specified in Verilog format. Using the ABC tool [17], we employ a custom-designed library to map the function into an OIG representation. Next, a levelized intermediate file is generated to support MAC-based mapping. In this stage, all nodes are analyzed by examining their inputs and outputs and are then organized into levels based on output dependencies. This levelization facilitates the identification of nodes and their corresponding connections at each computation stage.

Upon generating a levelized netlist, our MAC-based mapper transfers values to the RRAM crossbar. For a Boolean function with n primary inputs and m primary outputs, assuming the synthesized netlist comprises N gates and L levels, The required write cycles or initialization cycles for synthesizing a function are equal to N, i.e:

$$\#\text{Initialization Cycle} = N. \tag{2}$$

In our method, initialization cycles for different inputs are executed only once, unlike in the MAJ and MAGIC-based methods. Although the increased frequency of write cycles might seem like a disadvantage, it becomes an advantage in the context of a larger processor. All gates in each level can be computed in one MAC

cycle. However, we add an extra cycle per level for intermediate delays, caused by the column currents being fed into ADCs and written back into the crossbar for the next level. Primary outputs are not mapped to the rows and can be read from the columns. Total MAC cycles are calculated as:

$$\#\text{MAC Cycle} = (2 \times L) - 1. \tag{3}$$

Hence, the total number of cycles required for evaluating is given below:

$$\#\text{Total Evaluation Cycle} = N + (2 \times L) - 1. \tag{4}$$

The size of the RRAM crossbars varies depending on the Boolean function type. If a Boolean function has independent outputs, i.e. its primary outputs that are not influenced by any other primary output as input, then the number of required rows in the crossbars is:

$$\#\text{Rows} = 2 \times (n + N - m). \tag{5}$$

Every variable here, either a primary input or an intermediate input of gates, and their complement, requires two rows within the crossbars. Similarly, the output of each level, along with its complement, is mapped to subsequent rows. For dependent primary outputs, the calculation for the required number of rows is:

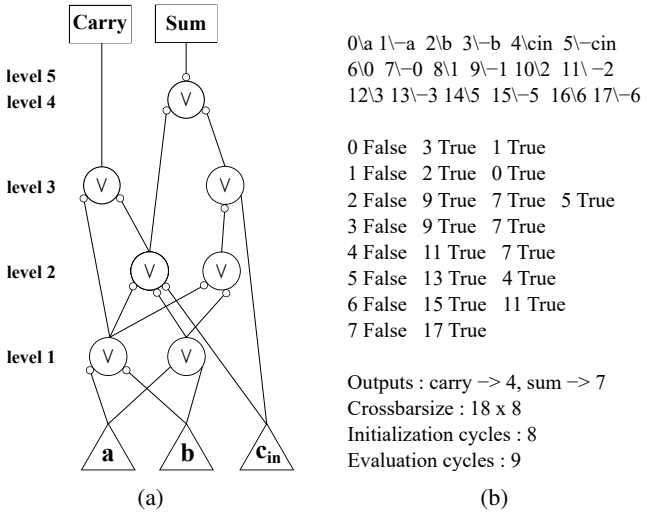
$$\#\text{Rows} = 2 \times (n + N + d - m). \tag{6}$$

Here, d denotes the number of primary outputs that are dependent on each other. The number of required columns will be equal to N since each gate is implemented in a separate column. This value remains consistent for both scenarios.

B. MAC netlist representation

This section describes the micro-operation file format and the algorithm that generates it from a levelized OIG netlist. The tool takes this netlist as input and generates a ***.mac file***, which can be directly mapped to the crossbars. Generating the *.mac file* requires three types of instructions:

1) **Primary Input mapping instruction**: This instruction provides information about the primary inputs and the rows to which they, along with their complements (denoted by '-'), are mapped. The syntax is as follows:
   `<row0> <var0> <row1> <-var0> <row2> <var1> ...`
   where `<row>` denotes the wordline, and `<var>` signifies the primary input variable name.
2) **Column to row mapping instruction**: Since a levelized netlist requires outputs and their complements from some levels as inputs for the next level, this instruction maps a column's (gate's) output to a row for further processing. It shows the connection of the crossbar's columns to the rows of the crossbars. The syntax of the instruction is:
   `<row0> <col0> <row1> <-col0> ...`
   In this notation, `<row>` represents the row, and `<col>` indicates the column linked to that row.
3) **RRAM initialization instruction**: Each OR gate in the OIG is mapped to a column in the crossbars. To evaluate these gates, we initialize the RRAM for the variables in that column. This instruction sets the RRAM, specified by row and column, to the low resistance state. It follows the syntax:
   `<col> False <row1> True <row1> True ...`



(a) (b)

Fig. 3: (a) The OIG Graph of Full adder, and (b) Fulladder.mac file

Here, `<col>` represents the column, and `<row>` represents the row. This constitutes a column-parallel operation where RRAM devices in a column are activated concurrently by applying False to the column and then True to the row of the intended RRAM devices.

For instance, Fig. 3a illustrates the OIG graph of a Full Adder. In this graph, triangles represent primary inputs, while circles denote OR gates with varying fan-ins—up to three in this example, though the number may vary depending on the Boolean function. Each bubble indicates the negation of its corresponding input, and rectangles signify the outputs.

The OIG for this Full Adder comprises three inputs: a, b, and c_{in}, and produces two outputs: sum and carry. The graph spans five levels and includes seven OR gates, concluding with a NOT gate at the final level. This intermediate representation serves as the input for subsequent stages, enabling the generation of micro-operation files for crossbar mapping.

Fig. 3b presents the micro-operations generated for the Full Adder. The first instruction defines the row connections for primary inputs and their complements: a, $-a$, b, $-b$, c_{in}, and $-c_{\text{in}}$ are assigned to rows 0, 1, 2, 3, 4, and 5, respectively. Next, a column-to-row mapping is performed, where the 0th column is mapped to the 6th row, and its complement is mapped to the 7th row, maintaining a sequential pattern. Ultimately, the primary outputs are located in columns 4 and 7.

IV. MAC-BASED MICRO-OPERATION VERIFICATION

Referring back to Fig. 2, the overall verification methodology involves two representations of the same function: the Verilog-based Boolean function, which serves as the golden model, and the MAC-based micro-operation representation, considered the Design Under Test (DUT). The verification process begins by converting the micro-operations into an OIG netlist file, a gate-level representation. An equivalence checker, employing an SAT solver, is then used to determine whether the golden model and the DUT are functionally equivalent.

A. Verification Methodology

The fundamental idea behind SAT-based equivalence checking is to express the equivalence problem as a Boolean satisfiability instance, which is then evaluated by an SAT solver. If the solver

returns SAT, it indicates a difference between the two designs; otherwise, an UNSAT result confirms that they are functionally identical. From a circuit design perspective, this process starts by constructing a miter circuit between the golden model and the DUT. This miter circuit is then converted into a Boolean representation suitable for SAT-based analysis. The transformation is carried out by propagating Boolean expressions from inputs to outputs, progressively integrating the logic functions defined by individual gates. To enable efficient SAT solving, the Boolean formula is reformulated *CNF* using Tseitin transformation, ensuring that the resulting CNF remains proportional in size to the original formula.

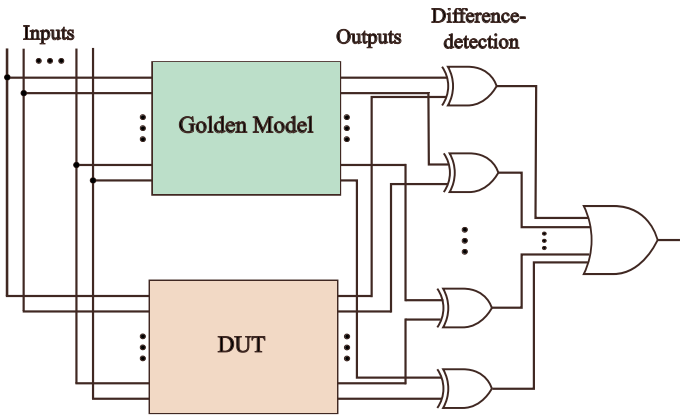


Fig. 4: The layout of a miter circuit for SAT-based verification

Fig. 4 illustrates the overall structure of the miter circuit used for equivalence checking. Corresponding inputs from the golden model and the Design Under Test (DUT) are connected to ensure they receive identical values. Each pair of outputs is compared using XOR gates, and the results are aggregated through a single OR gate. If the OR gate outputs a logical '1', it indicates there is at least one mismatch between the models.

To build this structure, intermediate bench files are generated for both the golden model and the DUT. A Python-based framework then merges the two representations and constructs the miter circuit within a combined CNF. This CNF is subsequently passed to the Z3 SAT solver, which verifies whether the two models are functionally equivalent.

V. Experimental Results

To present the experimental results of our proposed method, which includes both MAC-based mapping and its verification, we divide this section into two parts. The first part provides a comparative evaluation of the mapping process, while the second focuses on the verification results, demonstrating the accuracy and precision of our approach.

All experiments were conducted using the ISCAS-85 and IWLS 2005 benchmark sets [26], [27] on a machine equipped with an Intel® Core™ i7 2.10 GHz processor and 8GB of main memory.

A. Evaluation of MAC-Based Mapping

Table I presents the results of our MAC-based synthesis method on the IWLS benchmarks, including benchmark names, number of primary inputs and primary outputs (#PI/PO), and key performance metrics: number of levels (#L), crossbars size, write cycle, MAC cycle, and total cycle. It also compares the total cycle over the MAJ-based [28], and the MAGIC-based [29] methods. Our MAC-based method demonstrates significant improvements over both MAJ and MAGIC-based designs, achieving an average total cycle reduction of **50.60%** and **40.30%**, respectively.

Similarly, Table II provides results for the ISCAS-85 benchmarks. The performance metrics are structured similarly, with additional comparisons between serial and parallel MAGIC-based designs. Our method outperforms MAJ and serial MAGIC, reducing the total cycle by **12.67%** and **46.33%**, respectively. However, compared to parallel MAGIC, our method does not show an improvement in cycle count. Nevertheless, MAC-based computing proves to be more efficient in scenarios requiring repeated function evaluations. Fig. 5 shows that the MAC-based approach outperforms both MAJ and parallel MAGIC, even in the worst-case `c499` benchmark. As the number of iterations increases, the total cycle count of the MAC-based method decreases significantly compared to the alternatives.

Another key advantage of our method lies in the reduced number of write cycles, as depicted in Figure 6. In function synthesis, the number of initialization cycles is a crucial parameter. With the MAJ-based function, all inputs must be initialized for each iteration or various inputs, whereas for the MAC-based function, only one-time initialization is required. This immensely improves the endurance of our proposed method. Unfortunately, the write cycle data from [21] could not be included in Figure 6, as it did not provide information about write cycles.

In summary, our experimental results highlight the efficiency of the MAC-based approach in most scenarios, particularly in reducing total cycles and write cycles. While parallel MAGIC remains competitive in terms of cycle count, MAC-based computing offers notable advantages in endurance and repeated function evaluations, making it a strong candidate for scalable in-memory computing applications.

B. Verification and Accuracy Analysis

Ensuring the correctness of MAC-based micro-operations is crucial for reliable computation. To this end, we use an SAT-based approach to verify their functional equivalence against the original Boolean functions.

Table III presents the verification results, comparing the proposed method with the MAJ-based approach. The first three columns list the benchmark names, along with the number of *Primary Inputs (PI)* and *Primary Outputs (PO)*. The next two columns report the performance of our method in terms of the number of variables and clauses generated. The following two columns show the runtime (in CPU seconds) required by the Z3 solver to verify functional equivalence for both equivalent and non-equivalent cases. The final two columns present the corresponding runtimes for the MAJ-based method [11]. A timeout was assumed for instances that could not be solved within two hours.

For each benchmark, we first generate MAC-based micro-operations, which are directly mapped onto the crossbars. The verification process then compares these micro-operations against the original Verilog representation of the function.

To evaluate the accuracy of our method in detecting non-equivalence, we introduced random modifications by inserting or deleting logic operations in the crossbar micro-operations, while keeping the golden model unchanged. In all such cases, the SAT solver correctly detected the discrepancy, confirming non-equivalence.

TABLE I: Comparing results of the proposed method for IWLS 2005

Benchmark		Proposed					MAJ [28]	MAGIC [29]
Name	#PI/PO	#L	Crossbars($r \times c$)	Write Cycle	MAC Cycle	Total Cycle	Total Cycle	Total Cycle
9sym_d	9/1	11	112×48	48	21	69	176	138
con1f1	7/2	4	42×16	16	7	23	48	42
exam1_d	3/1	4	16×6	6	7	13	30	42
exam3_d	4/1	3	24×8	8	7	15	30	54
max46_d	9/1	12	234×109	109	23	132	404	114
newill_d	8/1	6	46×15	15	11	26	80	78
newtag_d	8/1	4	26×5	5	7	12	41	42
rd53f1	5/3	6	84×40	40	11	51	81	48
rd73f1	7/3	11	192×92	92	21	113	175	96
rd84f1	8/1	9	136×61	61	17	78	153	126
sao2f1	10/1	7	64×23	23	13	46	91	84
sym10_d	10/1	11	116×49	49	21	80	221	168
t481_d	16/1	7	82×26	26	13	39	46	174
xor5	5/1	6	32×12	12	11	23	45	54

TABLE II: Comparing results of the proposed method for ISCAS 85

Benchmark		Proposed					MAJ [28]	MAGIC [21]	
Name	#PI/PO	#L	Crossbars($r \times c$)	Wrire C.	MAC C.	Total C.	Total C.	serial	parallel
c432	36/7	17	248×91	91	33	124	408	361	222
c499	41/32	15	766×374	374	33	407	254	692	178
c880	60/26	14	566×249	249	43	292	412	610	180
c1355	41/32	14	802×360	374	29	403	300	683	156
c1908	33/25	23	742×338	338	45	383	448	702	240
c2670	233/140	20	1156×478	478	39	517	755	1149	180
c3540	50/22	30	1548×735	735	59	794	1173	1583	324
c5315	178/123	31	2856×1348	1348	61	1409	1826	2138	300
c6288	32/32	88	3750×1875	1875	175	2050	1801	3077	720
c7552	207/108	31	3048×1398	1398	61	1459	2252	2473	258

As expected, verifying equivalence generally requires more time than detecting non-equivalence. This is because, in equivalent cases, the SAT solver must exhaustively explore the solution space to confirm functional equivalence. In contrast, in non-equivalent cases, the solver can terminate as soon as a mismatch is found.

The results demonstrate that our method consistently outperforms the MAJ-based design style, achieving average improvements of **82.35%** in equivalent cases and **92.07%** in non-equivalent cases. The SAT solver efficiently determines solutions (i.e., SAT or UNSAT) for most benchmarks, including large ones like `c3540` and `c7552`. However, the benchmark `c6288`, a 16-bit array multiplier circuit, requires significantly longer runtimes due to its large number of clauses. Specifically, `c6288` includes 240 full-adder and half-adder units, forming extensive XOR networks. These increase the complexity of the SAT instance and enlarge the search space, leading to notably longer solving times for this benchmark.

Overall, the results validate the robustness and efficiency of our SAT-based verification approach for MAC-based micro-operations. The significant runtime improvements, particularly for larger and more complex circuits, demonstrate the scalability of our method and its suitability for practical use in in-memory computing applications.

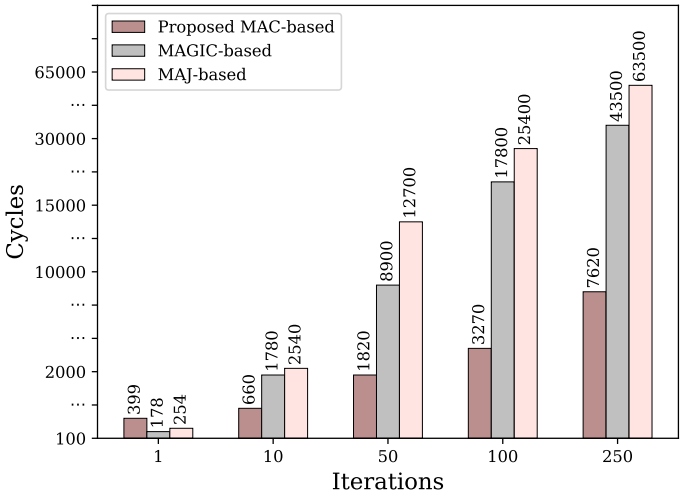


Fig. 5: Evaluation Cycle Analysis for a single function (`c499` [26])

VI. Conclusion

This work introduced a comprehensive MAC-based synthesis and verification framework designed to enhance the computational

TABLE III: Comparing results of the proposed method / Equivalent and Non-Equivalent cases

	Benchmark			Proposed Method				MAJ-based Method [11]	
	Name	#PI	#PO	Variables	Clauses	Equivalence(s)	Non-equivalence(s)	Eqivalence(s)	Non-equivalence(s)
ISCAS-85	c17	5	2	36	72	0.0103	0.0074	-	-
	c432	36	7	581	1326	0.0153	0.0099	2.013	2.331
	c499	41	32	1822	4479	0.0379	0.0147	9.541	10.673
	c880	60	26	1295	3088	0.0431	0.0142	-	-
	c1355	41	32	1846	4551	0.0769	0.0251	-	-
	c1908	33	25	1485	3694	0.0425	0.0137	10.636	10.333
	c3540	50	22	3707	9411	0.7668	0.0254	time>2 Hour	time>2 Hour
	c5315	178	123	6029	14591	0.2050	0.0349	-	-
	c6288	32	32	7688	19188	time>2 Hour	0.0470	time>2 Hour	time>2 Hour
	c7552	207	108	6517	15639	0.1990	0.0346	-	-
IWLS-2005	9sym_d	9	1	232	528	0.0164	0.0088	0.059	0.59
	alu4_98	14	8	3939	10020	0.2600	0.0283	-	-
	con1f1	7	2	93	198	0.0146	0.0072	0.053	0.053
	exam1_d	3	1	35	77	0.0138	0.0068	0.039	0.039
	exam3_d	4	1	43	93	0.0136	0.0072	0.049	0.049
	max46_d	9	1	572	1432	0.0228	0.0092	0.152	0.152
	newill_d	8	1	96	202	0.0129	0.0079	0.089	0.089
	newtag_d	8	1	57	103	0.0132	0.0071	0.042	0.042
	rd53f1	5	3	187	452	0.0150	0.0072	0.141	0.141
	rd73f1	7	3	432	1065	0.0204	0.0081	0.179	0.179
	rd84f1	8	1	312	754	0.0173	0.0077	0.258	0.258
	sao2f1	10	1	153	341	0.0138	0.0072	0.258	0.258
	sym10_d	10	1	264	622	0.0160	0.0084	0.062	0.062
	xor5	5	1	70	148	0.0137	0.0060	0.033	0.033

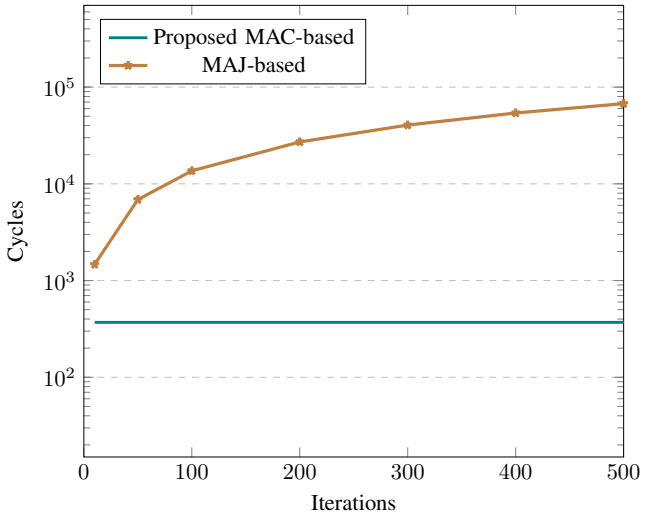


Fig. 6: Total Write Cycle Analysis for a single function (c499 [26])

efficiency and functional reliability of in-memory computing systems. Through experimental evaluation, we demonstrated that the proposed approach significantly outperforms existing methods, such as Majority (MAJ)-based and MAGIC-style designs, in terms of total computation cycles, write efficiency, and scalability across a diverse set of benchmarks.

In addition to synthesis improvements, we presented a formal verification strategy leveraging SAT-based techniques to ensure the correctness of the generated MAC-based micro-operations. The verification results validate the functional integrity of our approach, accurately identifying both equivalent and non-equivalent implementations, even in the presence of structural errors or for large-scale circuits. The proposed method efficiently detects functional discrepancies when errors are introduced, highlighting its robustness for practical applications. These findings establish MAC-based design as a promising paradigm for logic-in-memory architectures, combining computational efficiency with formal correctness. The proposed framework lays the groundwork for future advances in scalable and verifiable in-memory computing systems.

ACKNOWLEDGEMENT

This work is partly supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2 and SH 1917/1-2) and by the Indo-German project funded by Department of Science and Technology (DST), India, Federal Ministry of Education and Research (BMBF) and German Academic Exchange Service (DAAD), Germany (DST TPN No. 86669, DAAD No. 57682048).

REFERENCES

[1] L. O. Chua and M. K. Sung, "Memristive devices and systems," *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209–223, February 1976.

[2] M. Prezioso *et al.*, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[3] O. Krestinskaya, K. N. Salama, and A. P. James, "Learning in memristive neural network architectures using analog backpropagation circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 2, pp. 719–732, 2018.

[4] L. Xia *et al.*, "Technological exploration of RRAM crossbar array for matrix-vector multiplication," *Journal of Computer Science and Technology*, vol. 31, no. 1, pp. 3–19, Jan 2016.

[5] 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.

[6] 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.

[7] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 948–953.

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

[9] S. Fröhlich, S. Shirinzadeh, and R. Drechsler, "Multiply-accumulate enhanced bdd-based logic synthesis on RRAM crossbars," in *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020*. IEEE, 2020, pp. 1–5.

[10] F. Shirinzadeh, A. Deb, S. Shirinzadeh, A. Kole, K. Datta, and R. Drechsler, "In-memory sat-solver for self-verification of programmable memristive architectures," in *37th International Conference on VLSI Design and 23rd International Conference on Embedded Systems, VLSID 2024, Kolkata, India, January 6-10, 2024*. IEEE, 2024, pp. 384–389.

[11] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler, "Automated equivalence checking method for majority based in-memory computing on reram crossbars," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2023, p. 19–25.

[12] C. Jha, K. Qayyum, K. C. Coşkun, S. Singh, M. Hassan, R. Leupers, F. Merchant, and R. Drechsler, "veriSIMPLER: An Automated Formal Verification Methodology for SIMPLER MAGIC Design Style Based In-Memory Computing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–11, 2024.

[13] F. Shirinzadeh, K. Datta, S. Shirinzadeh, A. Kole, and R. Drechsler, "Towards formal verification for mac-based in-memory computing," in *Proceedings of the 33rd Asian Test Symposium (ATS 2024)*. IEEE, 2024.

[14] S. A. Cook, "The complexity of theorem-proving procedures, stoc'71: Proceedings of the third annual acm symposium on theory of computing," 1971.

[15] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

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

[18] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2015.

[19] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[20] S. Shirinzadeh, M. Soeken, P. 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. [Online]. Available: https://doi.org/10.1109/TCAD.2017.2750064

[21] P. L. Thangkhiew, R. Gharpinde, and K. Datta, "Efficient mapping of boolean functions to memristor crossbar using magic nor gates," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 8, pp. 2466–2476, 2018.

[22] S. Froehlich and R. Drechsler, "Generation of verified programs for in-memory computing," in *Digital System Design (DSD-2022)*, 2022.

[23] K. Qayyum, A. Kole, K. Datta, M. Hassan, and R. Drechsler, "Exploring the potential of decision diagrams for efficient in-memory design verification," in *Proceedings of the Great Lakes Symposium on VLSI 2024, GLSVLSI 2024, Clearwater, FL, USA, June 12-14, 2024*. ACM, 2024, pp. 502–506.

[24] 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.

[25] K. Datta, Deb, F. Shirinzadeh, A. Kole, S. Shirinzadeh, and R. Drechsler, "Verification of in-memory logic design using reram crossbars," in *21st IEEE Interregional NEWCAS Conference, NEWCAS 2023, Edinburgh, United Kingdom, June 26-28, 2023*. IEEE, 2023, pp. 1–5.

[26] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the iscas-85 benchmarks: a case study in reverse engineering," *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[27] C. Albrecht, "Iwls 2005 benchmarks," Tech. Rep., Jun. 2005.

[28] S. Fröhlich and R. Drechsler, "LiM-HDL: HDL-based synthesis for in-memory computing," in *Design, Automation & Test in Europe*, 2022.

[29] K. Datta, S. Shirinzadeh, P. L. Thangkhiew, I. Sengupta, and R. Drechsler, "Unlocking sneak path analysis in memristor based logic design styles," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 793–800.