The 33rd Asian Test Symposium (ATS 2024)

Towards Formal Verification for MAC-based In-Memory Computing

Fatemeh Shirinzadeh* fatemeh.shirinzadeh@dfki.de
Kamalika Datta*‡ kdatta@uni-bremen.de
Saeideh Shirinzadeh*† saeideh.shirinzadeh@dfki.de
Abhoy Kole* abhoy.kole@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) is a non-volatile memory technology with an abrupt switching property that enables it to perform basic logic operations. RRAM also possesses analog computational features by means of the so-called Multiply and Accumulate (MAC) operation that can be performed in all memory columns simultaneously. The MAC operation is particularly interesting for neuromorphic computing as it enables highly parallelized calculation of complex matrix-vector multiplications on standard RRAM crossbars.**

So far, several forms of universal logic are executed within RRAM devices, which have been the basis for a variety of logic-in-memory synthesis approaches. Recent research has addressed the mapping of logical functions to RRAM crossbars using the MAC operation, which allows for the facilitation of RRAM-based neuromorphic architectures with a basic logical core. Recently, a few formal verification methods have been introduced, which are tailored for synthesis approaches using certain RRAM logic primitives, such as in-memory styles based on the three-input majority operation and NOR gates. This paper analyzes these methods and, for the first time, proposes a verification method customized for MAC-based in-memory computing. A case study has been conducted to compare the proposed method with the existing methods, which reveals the superior performance of our method.

***Index Terms*—In-Memory Computing, Multiply-and-accumulate(MAC), Formal verification**

I. Introduction

RRAM or the so-called memristors have been a subject of interest in recent research due to their advantageous properties enabling new computing paradigms and architectures. Logic-in-memory computing has been explored by various methods [1]–[3]. that allow to evaluate arbitrary logic functions on standard RRAM crossbars by means of executing sequences of resistive logic primitives. These methods allow to combine memory and computational core and therefore alleviate the increasing issues in the current computer architectures which are caused by dissimilar progress rate of storage and processing units. However, current logic-in-memory approaches are still inadequate for practical use due to the resulting latency and complex control requirements when compared to conventional CMOS-based logic.

In addition to logic-in-memory architectures, use of RRAM is widely studied within neuromorphic architectures. Resistive memories allow to efficiently perform complex matrix-vector multiplications which are frequently required in neural networks and can be manipulated to mimic the behavior of natural synaptic architectures [4], [5].

Contrary to in-memory computing architectures, which use logic switching property of RRAM, neuromorphic structures exploit the analog computational features of RRAM crossbars. Fast evaluation of matrix-vector multiplications is conducted by setting conductivities of RRAM devices to desired values and applying appropriate voltages to memory wordlines. Results of this operation are then measured as currents flowing within each crossbar column, which is known as *Multiply and Accumulate (MAC)* operation [6].

Utilization of MAC or memristive logic primitives for computation on RRAM crossbars both require verification. Ensuring that a computation executes the desired functionality is essential. While manual inspection or simulation-based techniques are commonly used for validation, they are primarily effective for small-scale designs with limited inputs and outputs. As in-memory designs grow larger and more complex, these methods become less applicable. In such cases, equivalence checking becomes crucial, providing the advanced verification needed to confirm the correctness of crossbar mappings. This approach involves comparing the function description of the traditional logic network with the memristive operations on the crossbars to determine if they achieve the same functionality. Recently, a few research works have introduced formal verification for memristive in-memory computing based on execution of logic primitives in RRAM devices. However, formal verification of MAC-based computing on RRAM crossbar has not been addressed yet. This paper, for the first time, to the best of our knowledge, presents an approach

979-8-3315-2916-1/24/$31.00 ©2024 IEEE

for the MAC-based computational style due to its significance for both in-memory and neuromorphic architectures.

The rest of this paper is structured as follows. Section II explains the preliminary concepts required to understand the paper. Section III describes the existing works on formal verification of memristive logic-in-memory computing. The proposed method is explained in Section IV, followed by the experimental results and comparisons in Section V. Finally, Section VI concludes the paper.

II. Background

A. RRAM Crossbars

RRAM devices are among the most promising candidates for realizing in-memory computing architectures to overcome the longstanding problem of the memory wall. RRAM is a two-terminal memristive device whose internal resistance can be switched between a Low Resistance State (LRS or logic 1) and a High Resistance State (HRS or logic 0). RRAM devices are typically created in a crossbar structure, with each device formed at the junction of horizontal (bitline) and vertical (wordline) nanowires. Computation within RRAM devices has been performed using different fundamental logic operations.

Memristor-Aided LoGIC (MAGIC) is a logic design proposed to execute the NOR function within memristive crossbars [3]. MAGIC uses previously initialized input devices and realizes NOR within an output device initially storing a known logic value [7], [8]. Though all gates can be implemented using the MAGIC design approach, only NOR and NOT gates can be mapped to the RRAM crossbars. To map larger Boolean functions to the crossbar, it is necessary to express them as NOR gates and NOT gates first [7], [8].

In [9], a *Resistive Majority Operation* (MAJ) enabled by RRAM devices was introduced. According to MAJ, the resistive state of an RRAM device is switched from its current value r to $\bar{r} = p\bar{q} + pr + \bar{q}r$, where p and $\bar{q}$ represent the values applied to its top and bottom terminals, respectively. In [10], a Programmable Logic-in-Memory (PLiM) architecture based on the MAJ operation design is introduced, where RRAM devices are used for both storage and computation, enabling the integration of logic operations directly into the memory array.

Another work proposes a multi-objective algorithm for optimizing RRAM-based logic using *Majority Inverter Graphs (MIGs)* [11]. This approach significantly reduces the number of computational steps and improves efficiency in terms of both the required number of RRAM cells and computational steps in MAJ-based realizations.

In addition to performing the mentioned logic gates, RRAM can also execute analog computations, enabling it to carry out the *MAC* operation. MAC operations are widely employed in numerous applications, including neural networks and neuromorphic computing, to speed up complex matrix multiplications. RRAM crossbars offer an excellent platform for implementing MAC operations thanks to their unique capabilities. Considering that the resistive values of the RRAM devices in the crossbars are initialized with $a_{j,k}^{-1}$, m MAC operations can be conducted simultaneously within m crossbars columns by applying the voltages $x_1, \ldots, x_n$ to n rows.

The outputs are the currents in the crossbar columns, which are the sum of the currents through each RRAM device, as shown below:

$$i_j = \sum_{k=1}^{n} a_{j,k} \cdot x_k \tag{1}$$

This can be expressed as $I = Ax$, where $I = (i_1, \ldots, i_m)^T$, $x = (x_1, \ldots, x_n)^T$, and the matrix A is defined as:

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

Thus, within a single cycle, m MAC operations are computed in parallel, each involving n multiplications. Some recent work have also explored MAC-based computing for performing verification within RRAM crossbars [12].

B. Boolean Satisfiability (SAT)

The *Boolean Satisfiability Problem (SAT)* is considered as one of the classic NP-complete problems in computer science, for which there is no known algorithm for solving it in polynomial time [13]. The task here is to find out a possible set of variable assignments for a given Boolean formula such that the formula is evaluated to TRUE. If there exists such an assignment then the formula is said to be satisfiable (sat); otherwise, it is unsatisfiable (unsat) [14]. For solving any combinatorial problem, it must be first encoded into *Conjunctive Normal Form (CNF)*. A clause is defined as disjunction of literals. A literal can be a variable either in complemented or uncomplemented form. A formula is said to be in CNF if it can be represented as conjunction of clauses, i.e. conjunction of disjunction of literals.

Example 1: Let $f_n = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge (\overline{x_3})$.

A function f_n is represented in terms of CNF in Example 1. It has three clauses and three variables. The first clause has three literals, the second has two literals and the third clause has a single literal. One possible satisfying assignment for this function is $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$.

III. Verification for in-memory computing

In this section, we discuss various in-memory verification techniques that exist in the literature for Majority and MAGIC-based logic design styles. The main idea behind in-memory verification is to verify the correctness of the operations list generated through automated crossbar mapping programs. For executing any function on the crossbar we require a sequence of steps to be performed; therefore, it is of vital importance to verify the correctness of the same.

A. Verification for Majority-based mapping techniques

There exist many works in the literature that use Majority-based in-memory logic design [11], [15], [16]. Although these works particularly focus on delay and area optimization, they do not focus on verification. Some recent works, however, have targeted verification of Majority-based logic design [17]–[19]. In [17] the authors verified the generated crossbar operations using formal methods. The verification is performed in two steps, firstly the proof of purity is carried out followed by proof of equivalence. The authors have used CVC4, which is an open-source theorem

prover for *Satifiability Modulo Theories (SMT)*, due to its performance benefits. In [18], the authors propose an automated method for verifying the list of crossbar operations generated by their mapping tool. They have used the Z3 solver for verification. In another recent work [19], the authors have performed verification of adder circuits generated using automated mapping tool using *Binary Decision Diagram* (BDD). They have also carried out performance comparison with the Z3 solver.

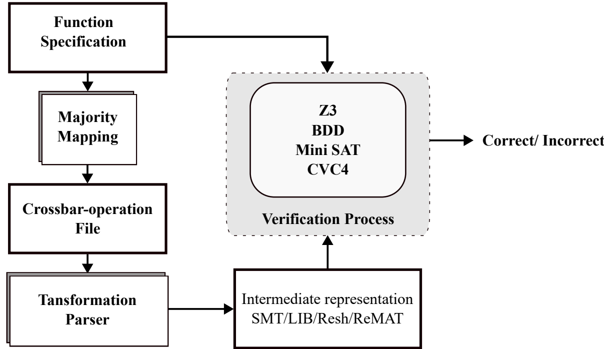

Fig. 1: Majority-based verification methodology

Fig. 1 shows the overall verification methodology for Majority-based mapping. From the input specification, the mapping tool first generates a sequence of operations for crossbar mapping. The goal is to verify the correctness of this sequence of operations. To perform this, a transformation parser converts the list of operations to an intermediate representation. Various intermediate representations have been proposed in the literature, viz. *ReRAM Sequence Graph* (ReSG) [18], *ReRAM Matrix* (ReMAT) [19], etc. Then the input specification and the intermediate form are fed to some specific proof engine like BDD, or SAT. Finally, the outcome determines whether the two specifications are equivalent or not.

B. *Verification for Magic-based mapping techniques*

The MAGIC design style is based on the execution of NOR gate operations on the RRAM crossbar. There exist many works in the literature that explore MAGIC -based designs for mapping boolean functions to RRAM crossbars [8], [20]–[22]. The main idea here is to first represent a function in terms of a netlist of NOR gates, and then use a mapping algorithm to translate this into MAGIC gate operations on the crossbar. Although some methods discuss generating a list of crossbar mapping operations, they are often very specific and lack a general representation of the operations list. As a result, many of these methods rely on manual inspection for verification.

A recent work [23] has explored the formal verification of a particular MAGIC-based mapping method, viz. Simpler [22], for the first time. This method essentially verifies the NOR operations generated by Simpler against the original input specification in Verilog. For both the input specifications and the NOR netlists, first Boolean Satisfiability formulas are generated, and then the Z3 SAT solver is used to verify the equivalence between the two. This method claims to identify certain bugs in mapping that were not considered in the Simpler method. Even though this method provides a verification methodology for MAGIC-based designs for the first time, it does not verify the functions at the mapping-operation level. Hence further research is needed to have a complete synthesis, mapping and verification strategy for MAGIC-based designs [24].

IV. Verification for MAC-based in-memory computing

In this paper, we proposed MAC Verification, which is the first verification Methodology for the MAC-based mapping in RRAM crossbars to the best of our knowledge. In general, our method performs equivalence checking between the golden reference Verilog file and the *OR Inverter Graph (OIG)* netlist generated from the synthesis process. This section briefly presents the process of mapping an arbitrary Boolean function onto the crossbar. It then addresses the verification methodology in detail.

A. *MAC-based Function Mapping in RRAM Crossbars*

Fig. 2 shows the overall scheme of the MAC-based mapping process on the crossbars. The process begins with the ABC tool [25], which maps an arbitrary Boolean function to an *OIG* using a designed library. The OIG is then levelized, and then, a C++ framework is used to map it onto the crossbars. As described in Section II-A, the preferred methodology for utilizing the MAC operation is the *Sum of Products (SoP)*. Thus, the OIG is implemented as a fundamental function representation to leverage these advantages.

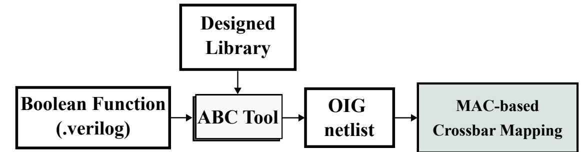

Fig. 2: MAC-Based Crossbar Mapping Overview

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 that represents the actual functionality and a complementary edge that represents 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.

Listing 1: Half adder's OIG netlist

```
module half_adder (a, b, sum, carry );
  input  a, b;
  output sum, carry;
  wire new_n5_, new_n6_, new_n7_;

  OR200   g0(.A(b), .B(a), .Y(new_n5_));
  OR211   g1(.A(b), .B(a), .Y(new_n6_));
  OR200   g2(.A(new_n6_), .B(new_n5_), .Y(new_n7_));
  NOT     g3(.A(new_n7_), .Y(sum));
  NOT     g4(.A(new_n5_), .Y(carry));

endmodule
```

The depth of OIG is determined by the total number of levels in the graph. in this paper, the OIG graph is optimized to minimize depth. Whenever possible, it prioritizes OR gates with larger fan-in at fewer levels, rather than using OR gates with smaller fan-in but more levels.

As a starting point, we use a Boolean function expressed in Verilog as an input file. A custom-designed library is then applied to accurately map each Boolean function onto an OIG graph structure, utilizing the ABC tool. In the next step, a levelized intermediate list for MAC-based mapping is generated. In the next step, a levelized intermediate list for MAC-based mapping is generated, which can be directly mapped onto the crossbars.

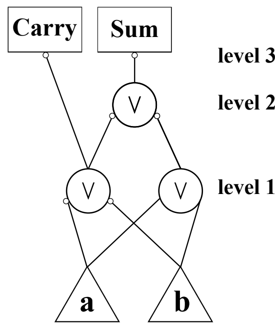

Fig. 3: The OIG graph of Half adder

As an example, Fig. 3 illustrates the OIG graph of a Half-Adder. It consists of three levels with three two-input OR gates and two NOT gates. Listing 1 shows the corresponding netlist for this graph. In the netlist, inputs, outputs, and wires are first introduced, followed by the description of the digital gates. The format for an OR gate is as follows: the first number after "OR" indicates the number of fan-ins (inputs) to the gate. This is followed by a sequence of binary digits equal to the number of fan-ins, which specifies how the inputs are connected. A "1" indicates a direct input connection to the gate, while a "0" denotes the negated input. Afterward, the name of the gate in the circuit is provided, followed by the input variables in parentheses, and the last variable within the parentheses represents the gate's output. For instance, the first OR gate is a two-input gate with negated inputs, $\bar{b}$ and $\bar{a}$, and its output is $new_n5_$. For NOT gates, the netlist simply specifies the input and its corresponding output.

B. Overall Verification Methodology

As the OIG can be directly mapped to the MAC-netlist to be executed on the crossbars, in this paper, we focus on verifying the OIG graph against our original Verilog benchmark. This choice is crucial because the accuracy of the OIG graph directly impacts the reliability of the entire process, ensuring that the subsequent implementation steps are based on a solid foundation.

The overall verification methodology is depicted in Fig. 4, which involves two representations of the same function: the Verilog-based Boolean function, serving as the golden model, and the parsed OIG graph representation, considered as the Design Under Test (DUT). Initially, the OIG graph, which is a gate-level representation is transformed into a behavioral-level representation. To verify functional equivalence between the golden model and the DUT, an equivalence checker leveraging a SAT solver [26] is used.

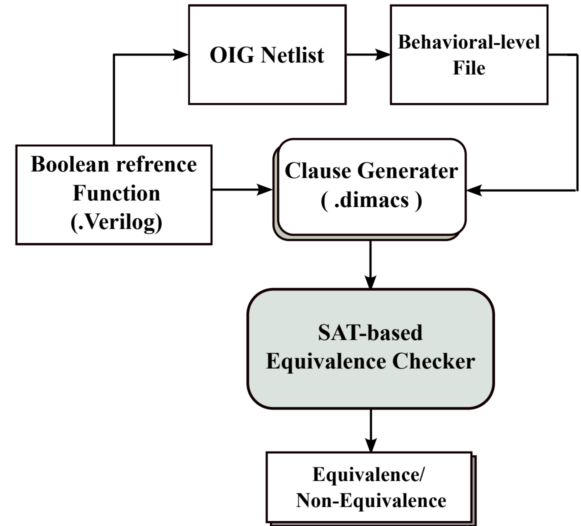

Fig. 4: Proposed verification methodology

The core concept of SAT-based equivalence checking involves formulating the design equivalence problem as a Boolean satisfiability instance, which is then solved by an SAT solver. If the solver finds a solution (SAT), it indicates a discrepancy between the designs; otherwise, they are equivalent (UnSAT). From a circuit perspective, the process begins by constructing a miter circuit between the golden model and the DUT. This miter circuit is then converted into a Boolean format suitable for the SAT solver. The entire circuit is transformed into a Boolean formula by traversing from inputs to outputs, progressively combining the Boolean function presented by the gates. To make it compatible with SAT solvers, the formula is converted into CNF using the Tseitin transformation, which produces an equisatisfiable CNF that remains linear in size relative to the original formula.

Fig. 5 shows the general structure of the miter circuit. The corresponding inputs from the golden model and the DUT are connected, ensuring that both circuits receive identical input values. An XOR gate is added for each pair of corresponding outputs to detect differences. The outputs of all XOR gates are then combined using an OR gate. If the OR gate outputs a '1', it indicates at least one difference exists.

To create this structure, an intermediate bench file is generated for both the golden model and the DUT. A Python framework is then used to produce a combined CNF, including the miter circuit, which is fed into the SAT solver to determine whether they are equivalent or not.

V. Evaluation of In-Memory Verification Effort

In order to validate our proposed method, a framework has been developed in Python 3.8. All experiments are applied to the

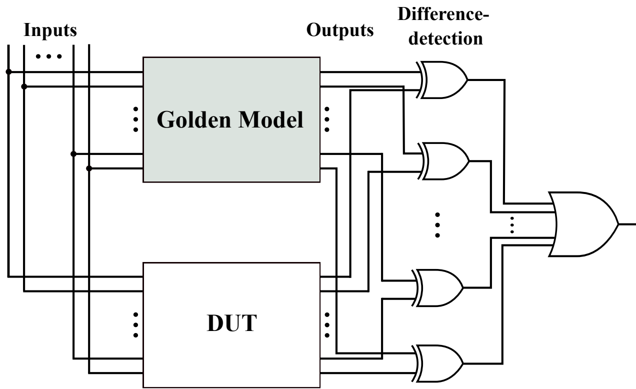

Fig. 5: The layout of a miter circuit for SAT-based verification

ISCAS-85, and IWLS 2005 benchmark sets [27], [28], and are carried out on a machine having an Intel(R) core(TM) i7 2.10 GHz processor and 8GB of main memory.
For all experiments, a timeout was assumed if the solver could not succeed in solving an instance within 2 hours. Table I presents the obtained results. The first three columns provide the benchmark names along with the number of *Primary Inputs (PI)* and *Primary Outputs (PO)*. The next two columns show the performance of our method in terms of the number of variables and clauses. The final two columns report the runtime, in CPU seconds, required by the Z3 solver to verify the equivalence between two functions. The runtimes are given separately for cases where the functions are equivalent and non-equivalent. For the considered benchmarks, we first generate the OIGs and this is checked against the original verilog representation of the function. The OIGs can be then efficiently mapped to MAC operations.

To verify the proposed method's accuracy for non-equivalent cases, we modified the DUT by randomly inserting or deleting logic operations in the crossbar file while keeping the golden model unchanged. The SAT solver confirmed that the two functions were not functionally equivalent.

As expected, the results demonstrate that the runtime for finding equivalent cases is significantly higher than for non-equivalent cases. This is because, in equivalent cases, the SAT solver must check all possible states to verify functional equivalence, whereas, in non-equivalent cases, the solver exits the process as soon as the first difference is detected.

Based on the results shown in Table I, the SAT solver efficiently determines solutions (i.e., SAT or UnSAT) for most benchmarks, including large ones like c3540 and c7552. However, c6288, which represents a 16-bit array multiplier circuit, requires significantly longer run times. This is due to its substantially larger number of clauses compared to other benchmarks. Specifically, c6288 incorporates 240 full-adder and half-adder functions, resulting in complex and extensive XOR networks. The complexity of these networks leads the Z3 solver to explore a much larger search space, which contributes to the notably increased solution times for c6288.

Overall, our proposed formal verification approach effectively identifies both equivalence and non-equivalence between the original Boolean function and its corresponding OIG graph netlist. The results also confirm that our method is scalable and effective.

VI. Conclusion

As verification is an important aspect in any design process in this paper we propose a verification strategy for MAC-based in-memory computing for the first time to the best of our knowledge. We exploit the OIG graph to map functions into RRAM crossbar efficiently. Firstly, the mapping tool generates the required OIG netlist. The OIG netlist is then converted into behavioral level representation which is verified against the golden response using the state-of-the-art SAT solver. Experiments were conducted to evaluate the performance of our verification strategy. Experiments reveal that the performance of our MAC-based verification is better in terms of runtime compared to other verification methods based on Majority and Magic design styles.

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] 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.

[2] 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)*. Ieee, 2016, pp. 427–432.

[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] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, p. 13, 2013.

[5] 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.

[6] M. Ali, A. Jaiswal, S. Kodge, A. Agrawal, I. Chakraborty, and K. Roy, "Imac: In-memory multi-bit multiplication and accumulation in 6t sram array," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2521–2531, 2020.

[7] P. L. Thangkhiew, R. Gharpinde, P. V. Chowdhary, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *2016 11th International Design & Test Symposium (IDT)*. IEEE, 2016, pp. 142–147.

[8] N. Talati, S. D. Gupta, P. S. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Trans. on Nanotechnology*, vol. 15, pp. 635–650, 2016.

[9] 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.

[10] 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.

[11] 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.

[12] F. Shirinzadeh, 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.

TABLE I: Results of the Proposed Method / Equivalent and Non-Equivalent Cases

	Benchmark			Variables	Clauses	Equivalent Cases	Non-Equivalent Cases
	Name	#PI	#PO			runtime(s)	runtime(s)
ISCAS-85	c17	5	2	36	72	0.023	0.0068
	c432	36	7	557	1257	0.103	0.0097
	c499	41	32	1696	4101	0.102	0.0134
	c880	60	26	1239	2920	0.042	0.0124
	c1355	41	32	1696	4101	0.065	0.0135
	c1908	33	25	1437	3534	0.073	0.0182
	c2670	233	140	2898	6443	0.045	0.0171
	c3540	50	22	3523	8821	0.541	0.0218
	c5315	178	123	5865	14141	0.183	0.0298
	c6288	32	32	7621	18988	timeout	0.0417
	c7552	207	108	6292	14956	0.184	0.0315
IWLS-2005	9sym_d	9	1	227	523	0.0081	0.0074
	alu4_98	14	8	3749	9501	0.135	0.0237
	con1f1	7	2	93	198	0.0073	0.0070
	exam1_d	3	1	36	77	0.0071	0.0068
	exam3_d	4	1	43	93	0.0070	0.0068
	max46_d	9	1	549	1399	0.0105	0.0090
	newill_d	8	1	95	199	0.0071	0.0070
	newtag_d	8	1	55	103	0.0069	0.0068
	rd53f1	5	3	178	427	0.0077	0.0075
	rd73f1	7	3	425	1045	0.0107	0.0085
	rd84f1	8	1	307	737	0.0093	0.0078
	sao2f1	10	1	154	342	0.0080	0.0072
	sym10_d	10	1	255	595	0.0085	0.0080
	xor5	5	1	68	145	0.0069	0.0067

[13] S. A. Cook, “The complexity of theorem-proving procedures, stoc’71: Proceedings of the third annual acm symposium on theory of computing,” 1971.

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

[15] 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

[16] 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, pp. 2 250 071:1–2 250 071:24, 2022.

[17] S. Froehlich and R. Drechsler, “Generation of verified programs for in-memory computing,” in *Digital System Design (DSD-2022)*, 2022, pp. 1–6.

[18] 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.

[19] 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.

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

[21] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, “A staircase structure for scalable and efficient synthesis of memristor-aided logic,” in *Asia and South Pacific Design Automation Conference*, 2019, p. 237–242.

[22] 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.

[23] 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.

[24] 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.

[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] L. d. Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[27] 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.

[28] C. Albrecht, “Iwls 2005 benchmarks,” Tech. Rep., Jun. 2005.