Devices, Synthesis, Circuits, and Verification for Resistive RAM-Based Digital Computing-in-Memory

Chandan Kumar Jha*, Ankit Bende†, Xingyue Qian¶, Simranjeet Singh†, Weikang Qian¶,
Vikas Rana†, Farhad Merchant§, Rolf Drechsler*‡
*University of Bremen, Germany; † PGI-7, Forschungszentrum Jülich GmbH, Germany,
¶ Shanghai Jiao Tong University, China; ‡DFKI GmbH, Germany; §University of Groningen, The Netherlands
{chajha, drechsler}@uni-bremen.de, {a.bende, si.singh, v.rana,}@fz-juelich.de,
{qianxingyue, qianwk}@sjtu.edu.cn, f.a.merchant@rug.nl

***Abstract*—Electronic Design Automation (EDA) has enabled the development of multi-billion-transistor designs in modern chips. Hence, similar methodologies are required to make digital computing using emerging technologies feasible. While for transistors, the development has happened over nearly eight decades, for emerging technologies, parallel efforts are being taken to enable their rapid feasibility. Some methodologies have been developed from scratch, while several are taken from existing technologies and tailored for emerging technologies. This paper attempts to highlight the recent developments in the area of using Resistive Random Access Memory (RRAM) for performing digital Computing-in-Memory (CiM). We have divided the paper into four sections and follow a bottom-up approach. First, we discuss the RRAM devices and their properties that make them feasible for digital computing. Second, we discuss the method to perform the mapping of an arbitrary design onto the RRAM crossbar. Third, we discuss the automated methods to generate the netlists of the mapping on the RRAM crossbars. Last, we discuss the automated formal verification strategies that are being employed to ensure the correctness of the transformation process at each of the synthesis and the netlist generation stages.**

***Index Terms*—RRAM, computing-in-memory, synthesis, spice netlists, Boolean satisfiability, formal verification.**

I. Introduction

Electronic Design Automation (EDA) techniques need to be developed to make the non-von Neumann architecture scalable and feasible. These architectures are realized using a variety of traditional and emerging devices [1], [2]. Among these Resistive Random Access Memorys (RRAMs) are one such device that shows immense potential for Computing-in-Memory (CiM) [3], [4]. RRAMs can be used to perform computations both in analog and digital domains [5]–[8]. Analog computing using RRAM crossbars is mostly limited to Multiply and Accumulate (MAC) operation. However, any arbitrary Boolean function can be implemented on the RRAM crossbar using digital CiM. In this work, we focus on the recent developments in the digital CiM techniques.

RRAM devices can be configured to be in High Resistance State (HRS) or Low Resistance State (LRS) depending upon the applied voltage, which are treated as logic 0 and logic 1, respectively. The Boolean operations are performed using these logic values. Various design styles have been proposed to implement logic operations using RRAM crossbars like Memristor-Aided Logic (MAGIC), Majority, FELIX, etc. [9]–[11]. Regardless of the design style used, the basic idea is to implement Boolean functions that are functionally complete, i.e., any design can be mapped to these Boolean functions. For example, in [7], the authors experimentally demonstrated that OR and NOT operations can be performed on the RRAM crossbars using the MAGIC design style. The data dependencies within the Boolean function can be managed using cloning techniques, which enable data movement through memory [12].

The simulation model that captures the behavior of these devices has been developed to enable circuit simulations [13], [14]. These simulation models are used to implement circuits, perform energy and performance analysis, using RRAMs, and are simulated using a Spice simulator [15], [16]. This can help to simulate the basic Boolean gates. Traditional synthesis tools do not optimize for the mapping on RRAM crossbar arrays. Hence, custom mapping tools have been developed to generate an optimal mapping for the memristor crossbars [8], [17]–[19]. These tools focus on exploiting the architecture of the memristor crossbars to reduce the design costs, like memristor count, number of memristor crossbars, etc. Recently, there have also been efforts to generate automated Spice netlists for the circuit level simulations and analysis [20], [21]. In these works, the focus is on automatically mapping the design to the RRAM crossbars and generating the Spice netlist that allows for circuit-level analysis.

With the development of automated methods for performing digital CiM, the correctness of the outputs generated by these methods becomes essential. For smaller designs, the correctness can be guaranteed using simulations as the number of input patterns is small. However, these automated methods allow for the mapping of larger designs on the RRAM crossbar, making the process of simulation-based verification infeasible. Hence, formal verification-based methods are required to guarantee the correctness. There have been works focused on formal verification for the mapping as well as the Spice netlists generated for the digital CiM [22]–[24]. In this work, we discuss the recent developments of the steps involved in the design of digital CiM in detail.

In Section II, we discuss the experimental demonstration of the Boolean gates in the memristor crossbar. In Section III, we discuss the automated mapping methodologies used for implementing the design on the RRAM crossbars. In Section IV, we discuss about the netlists generation for the RRAM crossbars. In Section V, the formal verification to guarantee the correctness of the mapping and the netlists is discussed. We conclude the paper in Section VI.

II. Devices Using RRAMs

CiM using RRAM offers a path to overcome the memory bottleneck in modern computing architectures. While several theoretical and simulation-based studies have demonstrated logic execution using memristors, practical implementation on fabricated arrays has started to gain interest [7], [25], [26]. We present one of the recent works on experimental validation of MAGIC gates implemented on One-Transistor–One-Resistor (1T1R) TaO_x RRAM arrays and analyze their performance with a special focus on energy consumption, device-level variation, and gate reproducibility [7].

A. Array Fabrication and Characterization

To demonstrate the CiM capabilities of RRAM devices, an 8×4 array of 1T1R RRAM cells was fabricated by the authors and

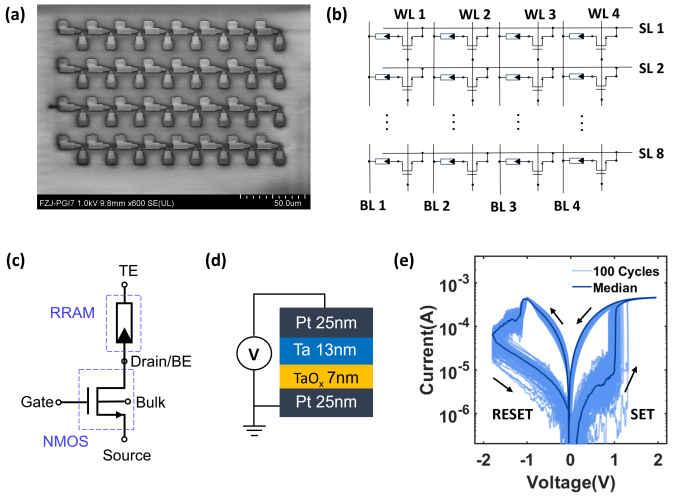

Fig. 1. (a) SEM image of the fabricated 8x4 1T1R RRAM, (b) schematic layout of the 8x4 1T1R array, (c) schematic layout of the 1T1R unit cell, (d) RRAM stack, and (e) I-V switching characteristics of the 1T1R RRAM.

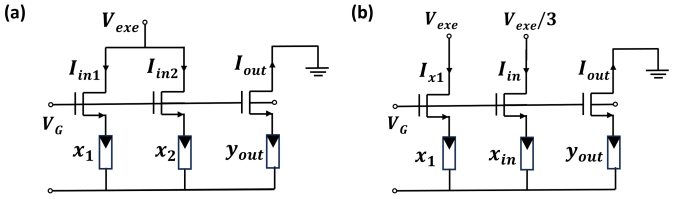

Fig. 2. Schematic layout of MAGIC (a) OR gate and (b) NOT gate.

integrated with a 180 nm CMOS technology platform [7]. The fabrication process commenced with Front-End-of-Line (FOEL) processed wafers containing NMOS transistors. Subsequently, the RRAM stack was integrated through sequential Radio Frequency (RF) sputtering and patterning of the bottom electrode, switching oxide, and top electrode layers. These layers were patterned using a standard back-etching approach with electron beam lithography to achieve nanoscale precision. A Scanning Electron Microscope (SEM) image of the fabricated array is presented in Fig. 1(a), and the corresponding layout schematic is shown in Fig. 1(b). Each 1T1R cell comprises an NMOS access transistor in series with an RRAM device, forming the basic computational unit, as illustrated in Fig. 1(c). The NMOS transistor enables selective access to individual cells and effectively mitigates sneak path currents. The RRAM device features a Pt/TaO_x/W/Pt stack in a crossbar architecture, as shown in Fig. 1(d), with a lateral device footprint of 100 nm $\times$ 100 nm. The RRAM devices exhibited counterclockwise switching as depicted in Fig. 1(e). The device exhibits low Cycle-to-Cycle (C2C) variation with a consistent HRS/LRS ratio of approximately 10, which makes them eligible for the implementation of logic gates.

B. MAGIC Logic Implementation

To demonstrate CiM using RRAM devices, the authors implemented and experimentally validated MAGIC-based OR and NOT gates using three 1T1R cells aligned in a single column of the fabricated array. The schematic layouts for the OR and NOT gate configurations are shown in Fig. 2(a) and Fig. 2(b), respectively. The logic operations consist of two phases: initialization and execution. During the initialization step, logical inputs are encoded into the

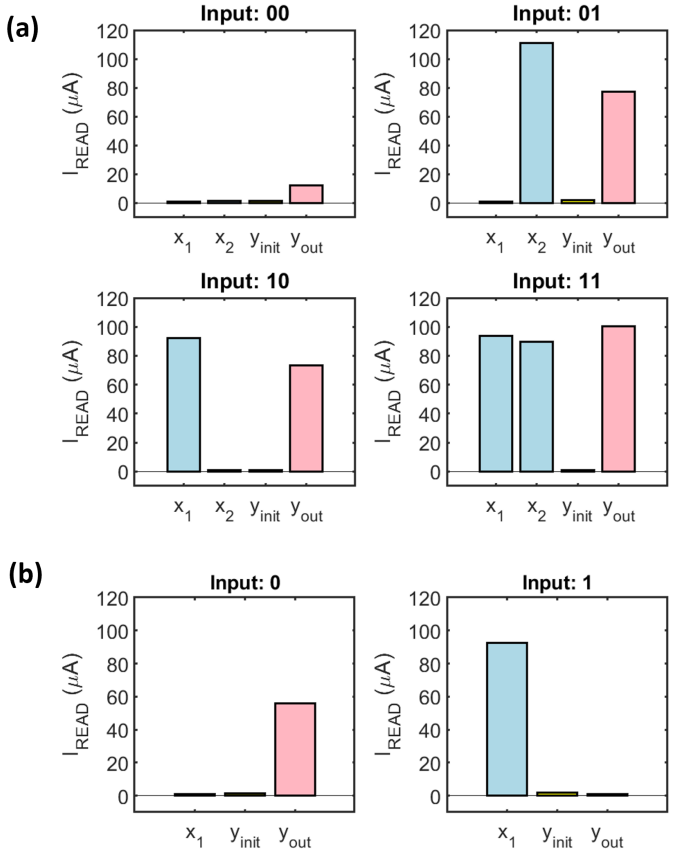

Fig. 3. Output read currents for all possible input combinations (a) logic OR operation and (b) logic NOT operation.

TABLE I
ENERGY BREAKDOWN FOR OR LOGIC OPERATION.

Input	Initialization Energy (nJ)	Execution Energy (nJ)	Read Energy (nJ)	% Initialization Energy	% Read Energy
00	**696**	**8**	0.07	99	0.003
01	**738**	**108**	2.8	87	0.1
10	**738**	**73**	2.8	91	0.1
11	**780**	**134**	5.6	85	0.2

resistive states of the memristors x_1 and x_2, while the output memristor y is initialized to logical "0". Logic "1" corresponds to the LRS, while logic "0" corresponds to the HRS. In the execution phase, voltage pulses are applied to the Source Lines (SLs) of the input memristors while the SL of the output memristor is grounded keeping the bit line at float. For the OR gate, both SL_1 and SL_2 receive the full execution voltage (V_{exe}), whereas in the NOT gate, SL_1 is connected to V_{exe} and SL_2 is biased at $V_{exe}/3$.

Fig. 3(a) and Fig. 3(b) present the measured output currents for OR and NOT gates after execution, respectively as shown in [7]. With V_{exe} = 3.3 V for OR and 1.5 V for NOT, correct logic outputs were consistently obtained. In the OR configuration, the output remained in the LRS state if either of the inputs was LRS, in line with expected MAGIC OR behavior. For the NOT gate, toggling the input confirmed successful logic inversion.

C. Energy Consumption of Logic Operations

To understand the energy profile of the logic operations, the authors also quantified the energy consumed during each operational

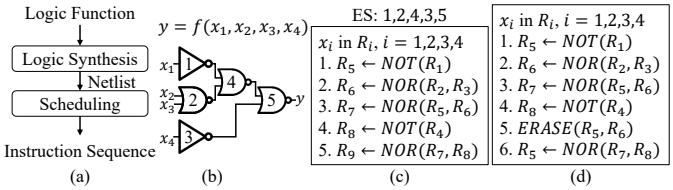

Fig. 4. Synthesis for digital computing-in-memory: (a) the basic synthesis flow; (b) an example of a NOR2 netlist; (c) an example of execution sequence (ES) and instruction sequence; (d) an example of instruction sequence when the cell number is limited.

stage of the logic OR operation—initialization, execution, and read. As shown in Table I, more than 85% of the total gate energy is consistently consumed during the initialization phase across all input cases. The execution energy ranges from 8 nJ to 134 nJ depending on the input combination, whereas the read energy remains under 6 nJ. This analysis underscores the dominance of the initialization step in energy consumption and highlights the importance of optimization strategies at this stage. The prior works mostly focused on optimizing the execution energy [16], [27], [28], however, the authors showed the importance of initialization energy in the computation process for the MAGIC design style [7].

These experimental results demonstrate successful in-memory logic execution using 1T1R TaO_x RRAM arrays with high fidelity and energy efficiency. The analysis confirms that initialization dominates energy consumption, while logic execution and read phases are significantly less demanding. This highlights the potential of optimization in initialization schemes for improving overall energy efficiency in memristor-based logic.

III. Synthesis Using RRAMs

The typical synthesis flow for RRAM-based digital CiM is shown in Fig. 4(a). The input is the target function such as multiplication, and the output is the instruction sequence given to the CiM hardware to compute the function. The flow consists of two main steps: logic synthesis and scheduling (also called mapping in some works). The logic synthesis step transforms the target function into a netlist of supported operations. For example, MAGIC [9] supports NOT and NOR operations, so the corresponding netlist can be a NOR2 netlists as shown in Fig. 4(b), where each operation is either a NOT operation or a 2-input NOR operation. The Primary Inputs (PIs) of the function, *e.g.*, x_1, x_2, x_3 and x_4, are stored in the RRAM array before the computation begins. By executing the operations in the netlist, we can obtain the values at the Primary Outputs (POs) of the function, *e.g.*, y. More details of logic synthesis will be introduced in Section III-A.

Then, scheduling determines the instruction sequence that instructs the hardware to execute the operations. Many works [8], [10], [18], [29]–[31] follow the Single Instruction Multiple Data (SIMD) setup where the computation on one input pattern is performed in the cells in one row (or column) and different rows (or columns) are used for computing the same function on different input patterns in parallel. This setup can achieve a high throughput, but the operations must be executed in serial. An operation can be executed once all its fan-ins have been executed. The scheduling process first determines the Execution Sequence (ES) of the operations. In the example in Fig. 4(c), the ES is $1, 2, 4, 3, 5$. Then, the process allocates the RRAM cell to store the output of each operation in the order of ES and generates the instruction(s) to execute it. For example, at clock cycle 3, $R_7 \leftarrow NOR(R_5, R_6)$ is scheduled, which executes a NOR operation on cell R_5 and R_6 that store the results of operations 1 and 2, respectively, and stores the result in R_7. In some circumstances, we need to insert cross-array copy or erase instructions before executing the operation.

Some other works [17], [19], [27] follow non-SIMD setup. In MAGIC [9], several operations in the netlist can be performed in parallel if their inputs are aligned in the same rows or columns. Hence, this setup can achieve smaller latency, but the computation on one input pattern may span more than one row and column. More details of scheduling for SIMD and non-SIMD setups will be introduced in Sections III-B1 and III-B2, respectively.

A. Logic synthesis

The logic synthesis step transforms the target function into a netlist of supported operations. Logic synthesis tools such as ABC [32] and Mockturtle [33] can automatically map a function expressed in various formats such as truth table to a baseline netlist. Since it takes energy and time to execute each operation, the main target of the logic synthesis step is to minimize the *size* of the netlist, *i.e.*, the number of operations. Most works apply existing commands from logic synthesis tools such as rewriting [34] and resubstitution [35] to the baseline netlist to reduce its size. The recent work [18] uses the widely-used logic optimization command sequence *resyn2* from ABC [32], which contains 10 optimization commands including rewriting and refactoring [36], to effectively reduce the size.

A few other works adjust the existing commands to fulfill the particular needs of their target hardware. For example, in [10], the authors modify the Majority-Inverter Graph (MIG) optimization algorithm to decrease the number of operations with multiple complemented inputs, since performing these operations requires additional clock cycles and RRAM cells on the target hardware [37].

B. Scheduling

1) SIMD Setup: Scheduling process in the SIMD setup determines the ES of the operations in the netlist and the RRAM cells to store their result, and then outputs the instruction sequence. Different instruction sequences have different performance after applied on hardware, so the target of scheduling is to obtain an instruction sequence of good performance. CiM hardware requires that the cells storing the inputs and output of an operation to locate in the same array when executing the operation. However, the size of the array is limited. If the inputs are originally stored in different arrays, we must first insert energy-consuming copy instructions to copy them to the same array before executing the operation. The insertion of copy instructions may diminish the benefit of CiM [8].

Hence, most scheduling algorithms aim at minimizing the number of cells required in the computation process, which is also called the memory footprint. For example, in Fig. 4(c), the memory footprint is 5 since 5 cells, *i.e.*, R_5 to R_9 are used. To achieve this target, the main effort is made on determining the ES. Given the ES, the corresponding instruction sequence that achieves minimum memory footprint can be easily generated by storing the operation result in the available cell with the smallest index. Several methods have been proposed to obtain the ES with minimized memory footprint. OptiSIMPLER [29] formulates a Boolean Satisfiability (SAT) problem and solves it with an Satisfiability Modulo Theories (SMT) solver. It can achieve the optimal memory footprint, but only works on small netlist due to the long runtime. To support larger netlists, other works apply heuristic methods. The scheduling algorithm in [10] is based on priority. At each clock cycle, the ready operation with highest priority is picked to be executed, where an un-scheduled operation becomes ready once all its fan-ins have been scheduled. The priority

of an operation is determined by the number of fan-ins that can be removed from memory after executing the operation and the level of its fan-outs. STAR [30] performs a depth-first search from the POs until reaching a ready operation. The traversal order among fan-ins is determined by the Strahler number [38], an optimistic estimation of the memory footprint required to obtain the fan-ins. In [31], the large netlist is first partitioned into several small netlists, which are then scheduled using an optimal scheduler. The scheduling results of the small netlists are finally combined to get the result of the large netlist.

For some architectures such as MAGIC [9], reusing the cells requires an extra instruction to erase them in bulk. For example, in Fig. 4(c), if there are only 8 cells available in a column, we need to modify the instruction sequence into the one in Fig. 4(d). The change is that we first insert an erase instruction at clock cycle 5 to erase R_5 and R_6 containing operation results that are no longer needed and then execute $R_5 \leftarrow NOR(R_7, R_8)$ at clock cycle 6. Hence, minimizing the number of erase instructions given array size limit is also a target of the scheduling process. STAR [30] uses a priority-based algorithm to achieve this target, with the priority set as the minimal number of operations that must be scheduled before erasing the operation.

Most previous works focus on computing with a single array. However, when the memory footprint exceeds the number of available cells in an array, we must use multiple arrays. For example, in Fig. 4(c), if there are only 4 cells in each column of an array, we cannot execute $R_5 \leftarrow NOT(R_1)$ at clock cycle 1 directly since R_1 and R_5 are from different arrays. Instead, we first need to copy x_1 from R_1 to R_6, which is located at the same array as R_5, at clock cycle 1, and then execute $R_5 \leftarrow NOT(R_6)$ at clock cycle 2 to perform the negation. As we mentioned, such cross-array copy instruction is energy-consuming, so minimizing the number of copy instructions is important given the array size limit. To achieve this target, a recent work, MASIM [18], proposes a priority-based scheduling algorithm followed by an iterative improvement flow. The priority-based scheduling algorithm uses the number of copy instructions that must be inserted before executing the current operation as the primary priority and the number of close partner pairs, a metric defined by the authors that can give a hint on the number of copy instructions needed in the future, as the secondary priority. The iterative improvement process randomly perturbs the ES and then obtains the updated instruction sequence using similar priority to further improve the result.

2) Non-SIMD Setup: The aim of the scheduling process in the non-SIMD setup is to minimize the latency in computing the target function on one input pattern. Similar to OptiSIMPLER [29] in the SIMD setup, SIMPLE [17] formulates an optimization problem to obtain the optimal latency for small netlists. Yadav [27] and SAID [19] use heuristic approaches to map larger netlists onto 2-dimension RRAM crossbar. Yadav [27] adopts a look-ahead strategy to reduce the number of intra-array copy operations inserted to align the operation inputs. SAID [19] divides a large netlist into small Boolean functions using a look-up table-based synthesis method and proposes a dedicated mapping method.

IV. RRAM-Based Circuits

Circuits implementing logic NOR and NOT gates using RRAM devices are designed by exploiting the programmable resistive states of the memristors within a crossbar array [9]. For a NOR gate, two RRAM cells represent the input operands, each programmed to either an HRS for logic '0' or an LRS for logic '1'. An additional RRAM

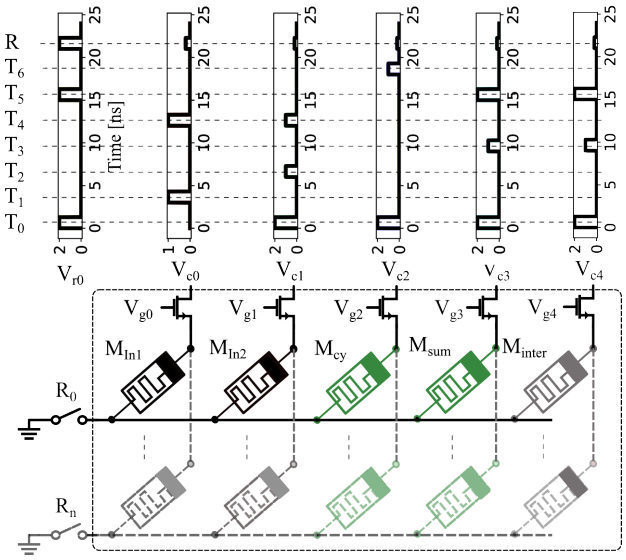

Fig. 5. Half-adder implementation using five memristors connected in series for the "10" input case (M_{in1} = 1, M_{in2} = 0). The sequence from T_0 to T_6 illustrates the seven cycles required to complete the half-adder operation, followed by an optional read cycle (R) to verify the final result. The dotted devices indicate that, similar to a SIMD architecture, the same operation can be executed concurrently across multiple rows (R_0 to R_n).

cell serves as the output node, initially set to LRS. By applying a specific voltage across the input and output lines, the output cell transitions to HRS only if both inputs are in HRS, thus realizing the NOR logic function directly through resistive state modulation. The NOT gate is implemented by connecting a single input RRAM cell and one output RRAM cell; the output cell is initially programmed to HRS and switches to LRS only if the input cell is in HRS, effectively inverting the input logic level. These gate designs rely on precise voltage control and the inherent switching dynamics of RRAM devices to perform logic operations inside the memory array, enabling compact and efficient in-memory computation without the need for conventional transistor-based logic circuits.

To achieve digital in-memory computing using RRAM devices, five different voltage sources are required, each tailored to specific aspects of the operation and ensuring the proper functioning of the MAGIC operation [15].

- ***Input Voltage:*** The input voltage determines the resistances of the RRAM devices used as inputs. It is mapped to 2.0V for storing '1' (LRS) and 0.0V for HRS (RRAM devices' default state).
- ***Read Voltage:*** The read voltage is applied to read the state of the output devices after all operations. It is set at 0.2V, relatively low compared to SET and RESET voltages, ensuring reliable read operations. There is potential for reducing energy consumption by further decreasing the read voltage.
- ***Initialization Voltage:*** The intermediate output devices, storing intermediate results, are initialized to LRS for accurate operation. This voltage configures them to LRS and is mapped to 2.0V for correct operation.
- ***Switch Voltage:*** During the execution cycle, only selected devices are active, isolated from others using a switch voltage. The switch voltage of the chosen devices is set to 2.0V (ON), while others remain at 0V (OFF), preserving their state during computation.
- ***Operation Voltage:*** During operation, specific voltages are applied to devices for NOR and NOT operations in the MAGIC design

style. Input devices are connected to 1.0V during NOT operation, while the output memristor is connected to the ground.

Fig. 5 illustrates the implementation of a half-adder for the input "10" ($in_0 = 1$, $in_1 = 0$) using five memristors arranged in a single row according to the mapping strategy presented in Listing 2. The implementation is executed over seven cycles, including an initialization cycle (T_0) and a reused cycle (T_4) required for reinitialization of memristors to store intermediate states. The top section of the figure shows the timing diagrams for the applied voltages across the row line (V_{r0}) and the five column lines (V_{c0} to V_{c4}) over the execution time. These voltage signals are selectively programmed at each time step to facilitate logic operations and state transitions in the memristive devices. The bottom part of the figure presents the schematic of the half-adder mapped to five memristors: two memristors (M_{In1}, M_{In2}) store the input values, while the memristors labeled M_{cy} and M_{sum} hold the carry and sum results, respectively. An additional memristor (M_{inter}) is utilized to store intermediate results necessary for completing the computation, and its reinitialization is indicated using a dotted connection in the schematic. Each memristor is accessed through a transistor controlled by gate voltages (V_{g0} to V_{g4}), which manage the programming and reading phases during execution. This structure enables in-memory computation of the half-adder using logic-in-memory principles, leveraging the state changes within memristors for performing logical operations directly within the memory array. Furthermore, this design aligns with a SIMD-like architecture where the same operation can be executed concurrently across different rows (r_n), allowing scalable and parallel computation. In [20], the authors proposed an automated methodology called MEMSPICE to generate the Spice netlist from the Verilog design. Internally, the tool uses the SIMPLER MAGIC tool to generate the mapping [8].

V. Verification Using RRAMs

In this section, we focus on the techniques used to ensure the correctness of the mapping and the circuits employed for CiM. Automated techniques are being explored, and complex designs are being mapped to memristor crossbars for CiM. The correctness of the transformations needs to be ensured at the mapping as well as the circuit level. While functional simulation can be used to guarantee correctness, it is very slow and does not scale well. For Spice circuits, performing a correctness check using spice simulation is further slow and is impractical even for smaller designs.

The research related to CiM was mainly focused only on synthesis and mapping. However, recently, the correctness of these mappings has also started to gain attention. One of the works that focused on generating correct mapping for the Very Long Instruction Word (VLIW) based on the ReVAMP architecture used the ABC tool internally to guarantee the correctness [39]. However, many popular mapping frameworks, like the SIMPLER MAGIC tool, do not guarantee correctness [8], [18], [19]. We discuss recent advancements in ensuring correctness for CiM. Most of these methods use SAT based methodologies to ensure the functional correctness. In general, the clauses generated from the golden model of the digital design are compared with the clauses generated from the mapping. These clauses are then fed to a SAT solver, which returns unsatisfiable if the designs are equivalent, or gives a counterexample for which the designs are different.

One of the first efforts in this direction was shown for Majority-based design [24]. The golden clauses are generated from the MIG. To generate the clauses for the micro-operations, the authors propose an intermediate data structure called ReRAM Sequence Graph (ReSG) from which the clauses are generated. The Python Z3 solver was used for comparing the clauses. The mapping of the golden design to the RRAM crossbar significantly alters the structure of the design. Hence, the authors showed that canonical representations that are independent of the design structure can be useful for verification purposes for the majority-based mapping [40]. The decision diagrams, like Binary Decision Diagrams (BDDs), Multiplicative Binary Moment Diagrams (*BMDs), and Kronecker Multiplicative BMDs (K*BMDs) are generated from the mapping as well as the golden design. Since these decision diagrams are canonical, the verification process becomes trivial as the decision diagram needs to be the same as that obtained from the mapping. It was shown that for interleaved ordering the proposed methodology consumes significantly less time for verification as compared to the Python Z3 solver method.

For the MAGIC design style, the SIMPLER MAGIC tool is one of the first open-source models to generate the mapping for the SIMD architecture. However, this tool provides no guarantees on the correctness of the mapping obtained. In [22], the authors proposed a methodology for the verification of the mapping obtained from the SIMPLER MAGIC tool. The clauses generated from the Verilog design were used as the golden reference. These clauses were then compared to the clauses generated from the mapping obtained from the tool using the Python Z3 solver. The authors identified a bug in the mapping. For the cases where the inputs and outputs were connected directly, the mapping obtained from the tool was incorrect. Since this was a specific bug, the authors also proposed a patch for the bug. In [41], a multi-input NOR-based mapping tool was proposed using the MAGIC design style. For the verification, the authors used a SAT based methodology.

While the prior methods focused on the verification of the mapping obtained from various tools, there were no automated frameworks for the generation of the Spice netlists until recently. There have been recent works that have proposed automated frameworks for the generation of Spice netlists [20], [21]. In [23], the authors showed the necessity of verification of Spice netlists generated from the tool proposed in [20]. The authors generated clauses from the Piece-Wise Linear (PWL) voltage files that are used to implement the gates using the RRAM crossbar. There were several conditions that were proposed, which needed to be satisfied for the generation of the clauses for the Boolean operations. These clauses were then compared with the clauses obtained from the golden Verilog design to guarantee the correctness.

While we have focused on the CiM using digital computations, analog computations using RRAM arrays are also being explored extensively [5], [6]. However, they suffer from variations which lead to challenges in performing analog computations using RRAMs [7], [42]. In [43], the author proposed a mathematical framework that can be used to guarantee the error bounds for matrix vector multiplication using RRAMs.

VI. Conclusion

In this work, we discussed the entire computing stack required to enable digital CiM using RRAMs. First, we discussed the fabrication of the RRAM devices and the implementation of digital gates on the RRAM crossbars. Second, we showed how complex digital designs can be mapped to the RRAM crossbars and optimized for SIMD and non-SIMD architectures. Third, we discussed how the mapped design of the RRAM crossbars can be automatically converted into a Spice netlist for circuit-level analysis. Lastly, we discussed the formal verification methods that are being developed to ensure correctness while transforming the digital design to the mapping on the crossbar

and the Spice netlists. Overall, we see that efforts are being made from devices to developing automated flows for the design using RRAMs. However, only a few design styles have been explored, and we believe that a lot of efforts are still required to make digital computing using RRAMs feasible.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) within the Projects PLiM (DR 287/35-2).

REFERENCES

[1] Z. Sun *et al.*, "A full spectrum of computing-in-memory technologies," *Nature Electronics*, vol. 6, no. 11, pp. 823–835, 2023.

[2] A. Chen *et al.*, *Emerging nanoelectronic devices*. John Wiley & Sons, 2014.

[3] S. Hamdioui *et al.*, "Memristor for computing: Myth or reality?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 722–731.

[4] W. Chen *et al.*, "Resistive-ram-based in-memory computing for neural network: A review," *Electronics*, vol. 11, no. 22, p. 3667, 2022.

[5] X. Chen *et al.*, "Rram-based analog in-memory computing : Invited paper," in *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2021, pp. 1–6.

[6] E. P.-B. Quesada *et al.*, "Experimental assessment of multilevel rram-based vector-matrix multiplication operations for in-memory computing," *IEEE Transactions on Electron Devices*, vol. 70, no. 4, pp. 2009–2014, 2023.

[7] A. Bende *et al.*, "Experimental validation of memristor-aided logic using 1t1r tao x rram crossbar array," in *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*. IEEE, 2024, pp. 565–570.

[8] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[9] S. Kvatinsky *et al.*, "MAGIC–memristor-aided logic," *Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[10] M. Soeken *et al.*, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference*, 2016, pp. 1–6.

[11] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–7.

[12] S. Singh *et al.*, "In-memory mirroring: Cloning without reading," in *2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC)*, 2024, pp. 1–6.

[13] S. Son *et al.*, "A comprehensive compact model for multilevel switching in taox-based memristive 1t-1r cells," *IEEE Access*, vol. 13, pp. 127 280–127 291, 2025.

[14] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[15] S. Singh *et al.*, "Should we even optimize for execution energy? rethinking mapping for magic design style," *IEEE Embedded Systems Letters*, vol. 15, no. 4, pp. 230–233, 2023.

[16] C. K. Jha *et al.*, "Imagin: Library of imply and magic nor-based approximate adders for in-memory computing," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 8, no. 2, pp. 68–76, 2022.

[17] R. Ben Hur *et al.*, "SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *International Conference on Computer-Aided Design*, 2017, pp. 225–232.

[18] X. Qian *et al.*, "MASIM: An energy-efficient multi-array scheduler for SIMD logic-in-memory architectures," in *International Conference on Computer-Aided Design*, 2025.

[19] V. Tenace *et al.*, "SAID: A supergate-aided logic synthesis flow for memristive crossbars," in *Design, Automation & Test in Europe Conference & Exhibition*, 2019, pp. 372–377.

[20] S. Singh *et al.*, "Memspice: Automated simulation and energy estimation framework for magic-based logic-in-memory," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 282–287.

[21] F. Seiler *et al.*, "Atomic: Automatic tool for memristive imply based circuit-level simulation and validation," in *2025 21st International Conference on Synthesis, Modeling, Analysis and Simulation Methods, and Applications to Circuits Design (SMACD)*, 2025, pp. 1–4.

[22] C. K. Jha *et al.*, "verisimpler: An automated formal verification methodology for simpler magic design style based in-memory computing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[23] C. K. Jha *et al.*, "verisim: Formal verification of spice netlists for magic-based logic-in-memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[24] K. Bhunia *et al.*, "Resg: A data structure for verification of majority-based in-memory computing on reram crossbars," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 6, pp. 1–24, 2024.

[25] L. Brackmann *et al.*, "Experimental verification and evaluation of non-stateful logic gates in resistive ram," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[26] B. Hoffer *et al.*, "Experimental demonstration of memristor-aided logic (magic) using valence change memory (vcm)," *IEEE Transactions on Electron Devices*, vol. 67, no. 8, pp. 3115–3122, 2020.

[27] D. Narayan *et al.*, "Look-ahead mapping of boolean functions in memristive crossbar array," *Integration*, 10 2018.

[28] P. L. Thangkhiew *et al.*, "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.

[29] D. Bhattacharjee *et al.*, *Synthesis and Technology Mapping for In-Memory Computing*. Singapore: Springer Nature Singapore, 2023, pp. 317–353.

[30] F. Wang *et al.*, "STAR: Synthesis of stateful logic in RRAM targeting high area utilization," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 5, pp. 864–877, 2021.

[31] X. Qian *et al.*, "A recursive partition-based in-memory simd computation scheduler for memory footprint minimization," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, pp. 2105–2118, 2025.

[32] B. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, 2010, pp. 24–40.

[33] M. Soeken *et al.*, "The EPFL logic synthesis libraries," 2022, arXiv:1805.05121v3.

[34] H. Riener *et al.*, "On-the-fly and DAG-aware: Rewriting boolean networks with exact synthesis," in *Design, Automation & Test in Europe Conference & Exhibition*, 2019, pp. 1649–1654.

[35] S.-Y. Lee *et al.*, "A simulation-guided paradigm for logic synthesis and verification," *TCAD*, vol. 41, no. 8, pp. 2573–2586, 2022.

[36] A. Mishchenko *et al.*, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[37] P.-E. Gaillardon *et al.*, "The programmable logic-in-memory (PLiM) computer," in *Design, Automation & Test in Europe Conference & Exhibition*, 2016, p. 427–432.

[38] D. Auber, "Using Strahler numbers for real time visual exploration of huge graphs," in *International Conference on Computer Vision and Graphics*, vol. 1, 2002, p. 3.

[39] D. Bhattacharjee *et al.*, "Contra: area-constrained technology mapping framework for memristive memory processing unit," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[40] K. Qayyum *et al.*, "Exploring the potential of decision diagrams for efficient in-memory design verification," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, 2024, pp. 502–506.

[41] S. Nabipour *et al.*, "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.

[42] J. He *et al.*, "Rvcomp: Analog variation compensation for rram-based in-memory computing," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 246–251.

[43] K. C. Coskun *et al.*, "Formal verification of error bounds for resistive-switching-based multilevel matrix-vector multipliers," in *2025 26th International Symposium on Quality Electronic Design (ISQED)*, 2025, pp. 1–8.