ATOMIC: Automatic Tool for Memristive IMPLY based Circuit-level Simulation and Validation

Fabian Seiler*, Peter M. Hinkel+, Axel Jantsch*, and Nima TaheriNejad+*
*Technische Universität Wien (TU Wien), Austria, +Heidelberg University, Germany
fabian.seiler@tuwien.ac.at, peter.hinkel@stud.uni-heidelberg.de,
axel.jantsch@tuwien.ac.at, nima.taherinejad@ziti.uni-heidelberg.de

***Abstract*—Memristor-based In-Memory Computation (IMC) is one of the promising candidates for the post-CMOS era, which comes in many flavors. Processing in Array (PIA) is a relatively new approach, and substantially different than traditional CMOS-based logic design. Consequently, there is a lack of publicly available CAD tools for memristive PIA design and evaluation. Here, we present ATOMIC: an Automatic Tool for Memristive IMPLY-based Circuit-level Simulation and Validation. Using our tool, a large portion of the simulation, evaluation, and validation process can be performed automatically, drastically reducing the development time for memristive PIA systems from multiple days to a few hours. With ATOMIC the effect of non-idealities can be analyzed, which is important but often unconsidered in the design phase. We evaluated State-of-the-Art (SoA) adder algorithms and compared their stability when calculated under non-ideal conditions. The code is available at https://github.com/fabianseiler/ATOMIC.**

I. INTRODUCTION

There is a growing emphasis on new computing technologies such as In-Memory Computation (IMC) based on memristors. This approach is a promising candidate for a More-than-Moore era since it has the ability to circumvent the Von Neumann bottleneck. Processing in Array (PIA) is a relatively new flavor of IMC that is substantially different than traditional CMOS-based logic design since it is only compatible with stateful logic forms such as Material Implication (IMPLY) [1] or Memristor-Aided Logic (MAGIC) [2]. We illustrated the typical development process in Figure 1, where we focus on IMPLY as it is the most reliable stateful logic [3]. We used $a \vee b$ as an example function to showcase each step, starting from the logic design and implementation as algorithm $\mathcal{A}$, which must be at least validated to confirm functionality. After that, the control logic must be synthesized, which is then used for various simulations on circuit-level to extract relevant criteria such as the energy consumption or stability regarding non-idealities. The control logic generation, circuit-level simulations, and especially the analysis of non-ideal behavior are the most time-intensive but necessary parts [4]. When carried out manually, the process can take multiple hours to days, even for experienced researchers. As there is a lack of publicly available Computer-Aided Design (CAD) tools for memristive PIA design and evaluation we propose **ATOMIC**, an automated Python tool to optimize the development of IMPLY-based algorithms. With this tool, a large portion of the validation, simulation, and evaluation process, as well as the processing and illustration of the data, is now

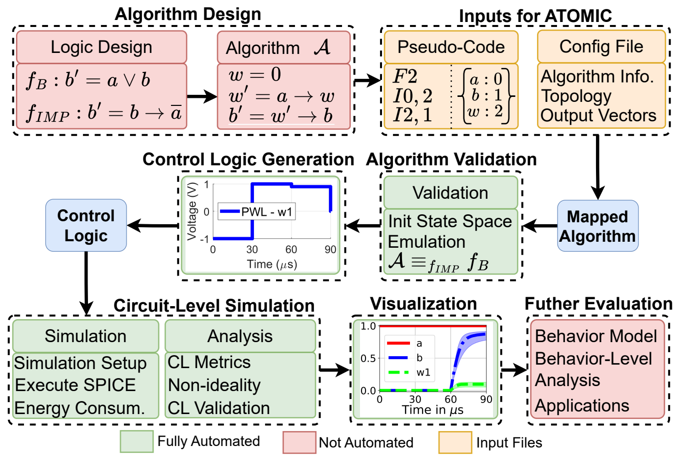
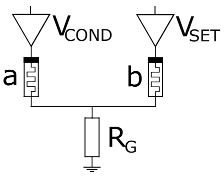


Fig. 1: Design process for algorithms in memristive IMPLY logic, with $a \vee b$ as example function. The green blocks are fully automated within ATOMIC.

completely automated as highlighted in green in Figure 1. Since real-world memristors experience non-idealities such as variation, studying their effect is an important topic, that is often disregarded in the State-of-the-Art (SoA). With our tool, the analysis of non-ideal deviations can be assessed automatically, which we utilize to compare the stability of SoA IMPLY adder algorithms.

II. BACKGROUND

Memristors are novel devices that can store data in a nonvolatile fashion, using their resistive states R_{on} and R_{off}, which can be interpreted as logical values [5]. Other advantages include low power consumption, fast writing time, and small dimensions [4], [5]. The inherent state changes when the applied voltage exceeds a certain threshold (v_{on} & v_{off}), forming a hysteresis curve. A commonly used model to simulate memristive circuits is VTEAM [6], which can accurately represent memristors ($< 1.5\%$ mean squared error [1]). The main limitation of this model is its inability to accurately capture non-ideal behavior. The most common and relevant deviations of real memristors are the variation of resistive states and threshold voltages [3]. IMPLY is a stateful logic that can be used for PIA, which was selected because of its higher stability compared to other approaches [3], [7]. The basic structure to perform IMPLY operations is shown



(a)

a	b	$b' = a \rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

(b)

Fig. 2: IMPLY operation [1]: (a) Gate structure, (b) Truth table

in Figure 2(a). The two memristors are connected to a common resistor which must fulfill the requirement $R_{on} \ll R_G \ll R_{off}$. When the voltages V_{COND} and V_{SET} are applied to a and b, the operation $b' = a \rightarrow b$ is executed. To allow for correct operations the voltages must satisfy the condition $V_{COND} < V_C < V_{SET}$, where V_C is the inherent threshold voltage of the memristor [1], [5]. The truth table of this operation is shown in Figure 2(b), where the inputs correspond to the resistive states of the memristors prior to the operation. The initial state of the b memristor is overwritten by the result of the operation, which increases the complexity when designing IMPLY-based functions. There are four commonly used IMPLY adder structures: 1) serial; 2) parallel; 3) semi-serial; and 4) semi-parallel, that each offer distinct advantages, rendering them competitive [7]. Here we present an automated framework that automates a large chunk of the development process of IMPLY algorithms in common topologies. We present a non-ideality analysis to cover this important but often disregarded design aspect.

III. ATOMIC Framework

In this section, we discuss the presented ATOMIC framework that automates a large part of the development process for IMPLY-based algorithms. This pipeline includes four steps: (A) validation of the algorithm, (B) control logic generation and topology mapping, (C) circuit-level simulation, and (D) graphical representation of results. An overview of the structure and project flow is illustrated in Figure 3, where each step is executed sequentially and the specified outputs are generated. To evaluate an IMPLY algorithm, only a config file containing general information (*.json*) and a step-by-step description of the algorithm in pseudo-code are required. The proposed pseudo-code consists of either $False$ or IMPLY operations that are applied to specified memristors. This is denoted by $\mathtt{F}\langle m_1 \rangle$ to reset memristor m_1 or $\mathtt{I}\langle m_1 \rangle$,$\langle m_2 \rangle$ to evaluate $m_1 \rightarrow m_2$. An example of this is illustrated in Figure 1. Memristor model and IMPLY-specific parameters can be adjusted to individual requirements. A detailed explanation of how to run and extend the project can be found in the corresponding technical document [8].

A. Validation of the Algorithm

The first step in creating an IMPLY-based function is the development of the specified boolean logical expression f_B. This function must be expressed using only the complete logic set $\{\rightarrow, \perp\}$ to comply with the constraints of IMPLY logic. We denote this expression as f_{IMP} which shares the truth table with f_B. Another important aspect to consider is that the initial state of an implied memristor is overwritten by the operation's result, which means that the sequence of operations is highly important. We represent this sequence of operations via the algorithm $\mathcal{A}$. As the design of such algorithms is highly complex, we introduce a procedure that emulates $\mathcal{A}$ and validates the equivalence $f_B \equiv f_{IMP}$ after the last step. The state space consists of the number of memristors m used in the algorithm. Each memristor is represented by a vector $\vec{x}_i$ of length $2^{|\mathcal{I}|}$, where $\mathcal{I}$ represents the set of input memristors. The vector of each input memristor is initialized with a bit pattern that corresponds to the enumerated column index of the input in the truth table. All other memristors are marked as uninitialized to verify if they are properly reset within $\mathcal{A}$.

Each operation in the algorithm is sequentially applied to the corresponding vectors $\vec{x}_i$, which are updated accordingly. We implemented the vectors and update functions using NumPy, to allow for a parallelized emulation to reduce the computation time. After the emulation of the algorithm, $\vec{x}_i$ of each output memristor is equivalence checked with the specified output. After the emulation of the algorithm, the state vector $\vec{x}_i$ of each output memristor is compared to the specified output of the algorithm. If all output states are equal to their specification, the algorithm is considered valid. In the case that differences occur a detailed error message and the state history after every operation is given, to allow for fast debugging on why the algorithm does not lead to the expected result.

B. Control Logic Generation & Topology Mapping

To evaluate the behavior of the previously validated algorithms on circuit-level, the mapping of the algorithm to a specific topology within the crossbar array [7] and the generation of control logic is required. The control logic consists of n steps, equal to the length of algorithm $\mathcal{A}$, each lasting a time of Δ_S. For each step, all memristors that are used in the current operation must be connected to the common row/column of the topology via a switch. Depending on the operation, a specific voltage must be applied to the memristors. For IMPLY-operations, the pre-defined V_{Set} and V_{Cond} are applied for Δ_S to the memristors specified in the pseudo-code. This is done for every step of the algorithm.

Since we implemented the serial, semi-serial, and semi-parallel IMPLY topologies up to two operations in parallel are possible. The different computational sections can be connected with additional switches outside the crossbar array, which require additional logic signals. After determining the operation voltages for the algorithm, the topology-specific control logic signals in the form of Piece Wise Linear (PWL) files are transferred to the simulator and saved to the output folder, to allow for easy debugging.

C. Circuit-level Simulation

In this section, we explain each part of the general simulation procedure, which uses the PyLTSpice library as an interface to SPICE, and showcase three experiments we implemented on top of it.

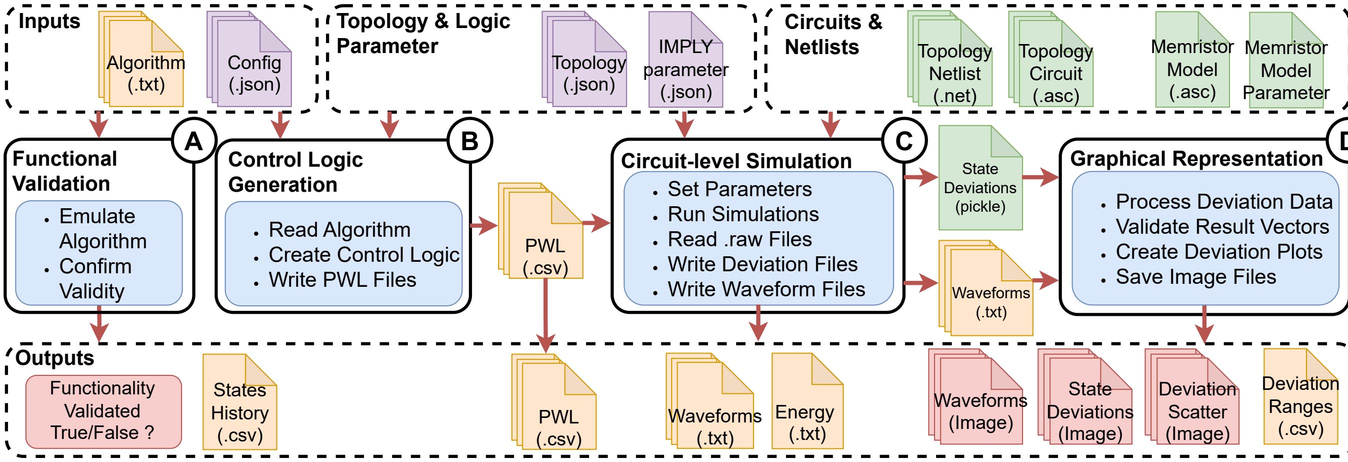


Fig. 3: Overview of the ATOMIC framework and automated process.

1) Simulation Procedure: Any circuit simulation in this project can be broken down into three parts. First, the simulation parameters are used to create a netlist of the targeted topology for simulation. Those parameters include the input states of the memristors as well as the values of the resistive states, and the voltage thresholds for the memristor model. The second part consists of executing SPICE simulations via the PyLTSpice interface, where the updated netlist with modified parameters and the simulator type can be selected. After the execution is finished, the output files created by LTSpice are stored locally to allow for debugging. In the last step the `.raw` files, which contain the waveforms of all the memristors, and the `.log` file are parsed, and the data is processed for further calculations. This simulation procedure is the underlying methodology of the following experiments.

2) Implemented Experiments: We implemented three exemplary experiments that are highly important for evaluating memristive circuits. As the non-ideality evaluation of Processing in Memory (PIM) circuits is often overlooked in the SoA, we lay additional emphasis on evaluating the impact of deviating memristor behavior. As this procedure is unfeasible when simulated by hand, our tool provides automated scripts that utilize the aforementioned simulation procedure to automate this process.

Energy Consumption: As the energy consumption measurements of memristive circuits are crucial for circuit-level evaluation, we propose an automated simulation that evaluates the energy of all possible input combinations C and provides the average, as shown in Equation (1). This is achieved by summing up the energy consumption for each memristor (m) at a specific input combination $c \in C$. The energy is then calculated by integrating the voltage-current product of the memristor over the time ($n\Delta_S$) required by the algorithm.

$$\text{Energy} = \frac{1}{|C|} \sum_{c \in C} \sum_{k=1}^{m} \int_{0}^{n\Delta_S} (V_k \times I_k)\ dt \qquad (1)$$

Deviation Experiments: We implemented automated experiments that check if an IMPLY algorithm yields valid results on circuit-level when specific properties are deviated from the ideal model by a certain degree. We focused on the variation of the resistive states R_{on} & R_{off} and the threshold voltages v_{on} & v_{off}, as those are the most prevalent and impactful deviations. Both non-idealities can be analyzed separately or combined. The validity of an algorithm at a certain maximum deviation d is checked by simulating all pairwise variations of $\pm d\%$ for both parameters (resistive states or threshold voltages). This is done for all possible input combinations C. The maximal difference between expected and non-ideal normalized output states is denoted as $Diff_{max}(d)$. This is done for all outputs.

$$Diff_{max}(d) = \max_{c \in C} |out(c) - out_{\pm d}(c)| \qquad (2)$$

Following conventions we denote an algorithm under the deviation d valid if $D_{max}(d) < 0.33$. With this approach, we evaluate all potential worst-case variations and validate them for all input combinations so that the algorithms are thoroughly evaluated on the circuit-level. We also implemented a combined analysis, where both the resistive states and the threshold voltages are varied, leading to a validation matrix. We use this approach in Section IV to compare SoA IMPLY adder algorithms.

D. *Graphical Representation*

To further utilize the data gained through simulation, we implemented a post-processing procedure and various illustration methods to extract and highlight important information, such as circuit-level validation or visualizations of the non-ideality effect on specific input combinations. Different configurations are available to highlight either the range of the output states or the validity of individual combinations using a color-coded scatter plot. The general validity of algorithms over increasing maximum deviations d is calculated by Equation (2) and color-coded. This illustration can give a good insight into potential requirements for the fabrication process. To illustrate the effect of specific deviations on a specific waveform, we included a method that finds the minimum and maximum of each waveform at every time step. This can be used to analyze what parameters, such as the frequency or voltage levels, are sensitive to non-idealities. Examples of figures of SoA algorithms are shown in [8].

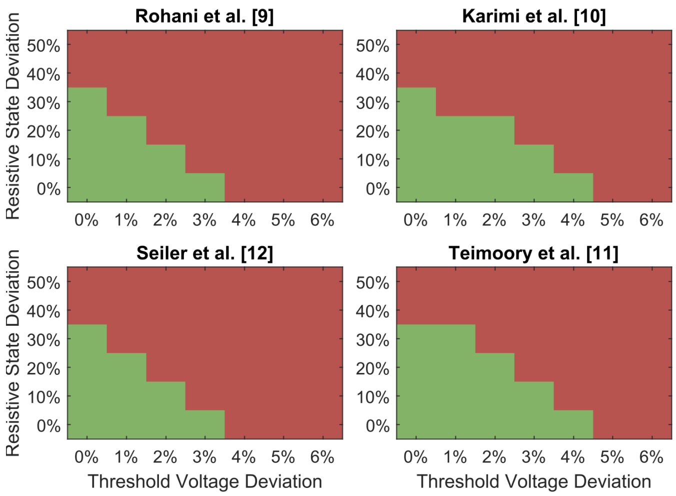


Fig. 4: Maximum deviation ranges that lead to valid outputs of SoA IMPLY algorithms. Valid areas are colored in green.

IV. Experimental Results

A. Experimental Setup

In this section, we evaluate SoA IMPLY-based serial adder algorithms [9]–[12] using ATOMIC. All algorithms are validated and lead to the same circuit-level metrics as specified in the original paper, so we focus on the novel non-ideality experiments instead. We compare their stability while varying the maximum deviation of both sets of tested parameters and evaluate all combinations as explained in Section III-C2. We evaluate a grid of deviation combinations and check where the algorithms are valid for all potential inputs. The VTEAM and IMPLY-specific parameters were chosen as presented in [12]. The resistive states R_{on} & R_{off} are varied up to $\pm 50\%$ in 10% steps, which is motivated by the experiments from [4], [7]. The voltage thresholds v_{on} & v_{off} are evaluated up to only 6% in 1% steps, as the conducted experiments revealed that IMPLY operations are highly sensitive to these changes.

B. Discussion

The results of the aforementioned experiments on the serial algorithms [9]–[12] are illustrated in Figure 4 where the maximum deviation of the threshold voltages and the resistive states are shown on the x-axis and y-axis, respectively. The green areas indicate that all input combinations lead to valid outputs. If the area is colored red, at least one waveform with maximum deviations $(d_{state}, d_{threshold})$ lead to an invalid or incorrect output. The evaluated algorithms all lead to very similar resulting valid ranges, indicating that these results are dependent on the memristor model and IMPLY parameters. The algorithm from Teimoory et al. [11] exhibits the highest stability compared to the other approaches, followed by Karimi et al. [10]. We can see that all algorithms are more sensitive to a deviation of the threshold voltages than to the resistive states by roughly a decade. This stems from the fact that even a small variation of the threshold may tremendously alter the hysteresis and, therefore, the behavior of the memristors. While changes of the resistive states may lead to decreased accuracy of the IMPLY operations, it still leads to good results if the condition $R_{on} \ll R_G \ll R_{off}$ is still satisfied. The most critical variation is therefore $R_{on} + d\%$ and $R_{off} - d\%$ as the difference between the logical states decreases. Our experiments indicate that for all algorithms, the input case "abc$_{\text{in}}$=000" is the most problematic since it always leads to the first invalid Sum output. As this output is calculated as $Sum = \overline{Sum} \rightarrow 0$ in all algorithms, a degradation of the logic state of the memristor where $\overline{Sum}$ is stored may lead to worse results This state degradation is cumulative of previously applied operations, where our experiments indicate that the case $1 \rightarrow 0$ has the highest impact on the states.

In summary, with ATOMIC, we have simulated and analyzed 4 SoA serial adder algorithms in roughly 4 hours each and generated a detailed report showing behavioral deviations. A comparable manual process would have taken multiple days/weeks for each algorithm.

V. Conclusion

In this work, we present ATOMIC, an Automatic Tool for Memristive IMPLY-based Circuit-level Simulation and Validation to drastically reduce development time. We implemented a process for non-ideality experiments to tackle the lack of available analysis tools in this area, which we use to compare the applicability of SoA IMPLY adder algorithms, reducing the simulation time from multiple days/weeks to a few hours. Future work may include compatibility with other logic forms and further automated experimentation.

References

[1] S. Kvatinsky *et al.*, "Memristor-based imply logic design procedure," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 142–147.

[2] ——, "MAGIC; memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, Nov 2014.

[3] D. Radakovits and N. Taherinejad, "Behavioral leakage and inter-cycle variability emulator model for rerams (BELIEVER)," *CoRR*, vol. abs/2103.04179, 2021. [Online]. Available: https://arxiv.org/abs/2103.04179

[4] F. Seiler and N. TaheriNejad, "Efficient image processing via memristive-based approximate in-memory computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[5] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, pp. 873–6, 04 2010.

[6] 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.

[7] F. Seiler and N. TaheriNejad, "Accelerated image processing through imply-based nocarry approximated adders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[8] F. Seiler *et al.*, "Atomic: Automatic tool for memristive imply-based circuit-level simulation and validation," 2024. [Online]. Available: https://arxiv.org/abs/2410.15893

[9] S. G. Rohani *et al.*, "An improved algorithm for imply logic based memristive full-adder," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2017, pp. 1–4.

[10] A. Karimi and A. Rezai, "Novel design for a memristor-based full adder using a new imply logic approach," *Journal of Computational Electronics*, vol. 17, 09 2018.

[11] M. Teimoory *et al.*, "Optimized implementation of memristor-based full adder by material implication logic," in *ICECS2014*, 2014, pp. 562–565.

[12] F. Seiler and N. TaheriNejad, "An improved serial imply adder algorithm for efficient neural network applications," in *2025 IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)*, 2025, pp. 1–5.