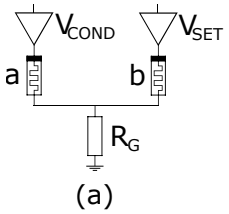
An Improved Serial IMPLY Adder Algorithm for Efficient Neural Network Applications

Fabian Seiler* and Nima TaheriNejad+*
*Technische Universität Wien (TU Wien), Austria, +Heidelberg University, Germany
fabian.seiler@tuwien.ac.at, nima.taherinejad@ziti.uni-heidelberg.de

***Abstract*—Memristive systems are one of the most promising candidates for a post-CMOS era. They are small, energy-efficient, and are ideal targets for In-Memory Computation (IMC) via stateful logic. As adders are critical building blocks for any computing systems, improving them is an essential design goal. With the rise of Artificial Intelligence (AI), providing memristive adders that are optimized for Neural Networks (NNs) is extremely important. For this, we propose a Material Implication (IMPLY)-based adder algorithm in the serial topology that can preserve the weights in memory, which was not addressed in the State-of-the-Art (SoA). Our approach is $20\% - 23\%$ faster and requires $1\% - 12\%$ less energy when the adder is used repeatedly. We propose a flowchart for IMPLY-based algorithms that can represent the state changes of individual memristors and apply it to our adder. We embed our adder in a shift-and-add multiplier and evaluate the potential gains on the 8-bit quantized ResNet18. Our approach is up to 17% more energy-efficient and requires up to 20% fewer cycles for the inference than SoA adder.**

I. Introduction

With the rise of demanding workloads such as Machine Learning (ML) an increasing amount of computational power is required. As in conventional computing methods, nearly two-thirds of the energy is consumed by data movement; a large emphasis now lies on In-Memory Computation (IMC) approaches [1]. As the computation takes place directly in memory, it may also provide a solution for the Von Neumann bottleneck [2]. As memristors are energy-efficient, have a small form factor, and inherently store data non-volatile, they are the ideal candidate for IMC [3]–[6]. In the realm of IMC the stateful logic Material Implication (IMPLY) is the most popular choice [3], [7]. As adders are the cornerstone of any modern computing system, a lot of focus lies on improving them [8]–[12]. State-of-the-Art (SoA) IMPLY-based adder disregard that inputs can be preserved for multiple successive operations. This leads to unnecessary reloading of inputs via inefficient `COPY` operations. As the computation of Neural Network (NN) inferences mainly consists of multiply–accumulate (MAC) operations, current approaches are not sufficient. In this work, we propose an input-preserving serial IMPLY adder algorithm that improves the speed and energy efficiency. This paper proceeds as follows: In Section II we cover related paper. We propose and simulate our adder algorithm in Section III and compare it to the SoA in Section IV. We evaluate a NN case study in Section V and conclude the paper in Section VI.



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

Fig. 1: IMPLY operation [3]: (a) Gate structure, (b) Truth table

II. Related Work

Memristors are novel two-terminal components that can store data in a non-volatile fashion with their resistive states. They were originally discovered by L. Chua [13] and physically realized by Strukov et al. [14]. Other advantages such as low power consumption, fast write time, and small dimensions, underline their potential as memory cells [9], [15], [16]. The memristor's minimum (R_{on}) and maximum (R_{off}) resistive values, which are commonly interpreted as logical '1' and '0' respectively, can be reached by applying a voltage [8], [17].

A wide range of stateful logic forms based on memristors, such as FELIX [18], SIXOR [19], MAGIC [20], and TMSL [21] have been presented. In this work, we focus on IMPLY [22], [23], since it boasts the most reliable operations [24]. This is an important analysis point for the non-ideal memristors, which is often disregarded. The basic structure to perform IMPLY operations is shown in Figure 1(a), where the initial resistive states of the memristors a and b represent the logical inputs. When the voltages V_{COND} and V_{SET} are applied, the operation $a \rightarrow b$ is executed. The result is stored in the b-memristor, overwriting its initial value and thereby losing this information. To perform IMPLY operations, the conditions $V_{COND} < V_C < V_{SET}$ and $R_{on} \ll R_G \ll R_{off}$ must be satisfied. V_C is the threshold voltage of the memristor [9], [15], [22], [23]. The corresponding truth table for IMPLY operations is shown in Figure 1(b).

As adders are the most basic building block in computing, there exists plenty of research on improving the performance of IMPLY-based adders [8], [11], [12], [25]–[28]. The topology of these adders can be divided into three categories: 1) serial [4], [15], 2) parallel [9], [23], and 3) hybrid forms, such as semi-serial [11] or semi-parallel [12]. Each topology has unique advantages in speed, area usage, or energy consumption, rendering them competitive with each other [26]. The serial topology uses fewer memristors than other archi-

tectures and requires no additional switches, making it easier to integrate within crossbar arrays [29]. Serial SoA IMPLY-based adder algorithms such as [8]–[10] focused only on logic minimization. They reuse all of the input memristors to store intermediate values, losing the initial states in the process. This works fine for a single ADD operation but is not very practical for more complex tasks that reuse inputs multiple times, as an inefficient COPY operation is required after each addition. This is especially important for tasks such as multiplication or MAC operations, which are very common in NNs. We propose an IMPLY-based adder in the serial topology that preserves one of the inputs, achieving a multiplication based on adders without requiring additional COPY operations. Our algorithm itself also is faster than SoA approaches, making it the ideal target for efficiently computing NN applications directly in memory.

III. Proposed Adder Algorithm

A. Design Methodology

As we are working in the serial IMPLY-based topology, only either an IMPLY or FALSE operation can be executed at each step. As they form the complete logic set $\{\rightarrow, \bot\}$ it is feasible to emulate any Boolean logic function [8], [30]. The optimization goal for our approach was to design adders that preserve one of the inputs (in our case the values stored in the a-memristors). When the adder is used repeatedly in complex structures such as multipliers an inefficient COPY operation is required to restore the overwritten inputs. With IMPLY logic, the COPY operation ($q' = p$) requires three additional steps for each input bit, which is not always parallelizable. The procedure is shown in Equation (1), where firstly a work-memristor and the q-memristor have to be reset, which can be done in one step with a high enough voltage [11]. After that, the inversion of p must be stored in w_0 which then implicates the q-memristor to store the original value.

$$q' = [p \rightarrow FALSE(w_0)] \rightarrow FALSE(q) \tag{1}$$

With our approach, adders can be used repeatedly to function as multipliers without losing the information of the a-input and storing the intermediate summation results in the b-memristors. With this, no COPY statements are required and the initial a-inputs are preserved, making it the ideal target to store weights for applications in NN.

B. Implementation

We analyzed the optimized IMPLY-based logical equations of a full adder, that were proposed in [8]. They used only two work memristors, which is the minimal amount to process two logical equations [31]. As the equations for Sum and C_{out}, which are shown in Equation (2) and Equation (3), are input symmetrical there must be intermediate states that are used multiple times. The methodology involves an initial computation of these states, which are then used to calculate the desired results. The order of operations is dependent on the sequential requirements of the given equations.

$$\begin{aligned} Sum &= \{(\overline{a} \rightarrow b) \rightarrow \left[(a \rightarrow \overline{b}) \rightarrow c\right]\} \\ &\quad \rightarrow \overline{\left[(a \rightarrow \overline{b}) \rightarrow \overline{\overline{a} \rightarrow b}\right] \rightarrow \overline{c}} \\ &= (X \rightarrow (Y \rightarrow c)) \rightarrow \overline{(Y \rightarrow \overline{X}) \rightarrow \overline{c}} \\ &= (X \rightarrow Z) \rightarrow \overline{(Y \rightarrow \overline{X}) \rightarrow \overline{c}} \end{aligned} \tag{2}$$

$$\begin{aligned} C_{out} &= \overline{(\overline{a} \rightarrow b) \rightarrow \overline{(a \rightarrow \overline{b}) \rightarrow c}} \\ &= \overline{X \rightarrow \overline{Y \rightarrow c}} \\ &= \overline{X \rightarrow \overline{Z}} \end{aligned} \tag{3}$$

We extracted three intermediate states that will be denoted X, Y, and Z that occur multiple times in both logical equations. This means that we require an additional work memristor to store the intermediate states and their deviations when the state of the a-memristor can not be overwritten. The corresponding logical equations are shown in Equation (4) - Equation (6). X and Y depict the only useful boolean functions that are possible to emulate with two IMPLY operations, which are OR and NAND. They represent the basic building blocks to emulate any boolean logic function efficiently in IMPLY logic. Z represents an intermediate step that can be used to calculate both XOR and MAJ, which are used in the equations for Sum and C_{out}, respectively.

$$X = \overline{a} \rightarrow b = a + b \tag{4}$$

$$Y = a \rightarrow \overline{b} = \overline{ab} \tag{5}$$

$$Z = Y \rightarrow c = ab + c \tag{6}$$

To illustrate this algorithm, we propose a flowchart that depicts the state changes and interactions between different memristors. The presented algorithm is shown in Figure 2, where each horizontal line corresponds to the state of a memristor over the whole process. This presentation method provides a clear overview of how an algorithm is processed, where the intermediate states are calculated, and how they are stored. For example, X is calculated in the third step when the inversion of a stored in the first work memristor implicates the b-memristor, overwriting the previous state. With our presented method, the a-memristor transfers its value two times to create X and Y but is not used anymore after the fifth step and keeps its initial state. The Sum is calculated in step 18 and stored in the b-memristor, while the C_{out} is saved in the c-memristor at the last step. A detailed procedure of the proposed algorithm and the equivalent logic, simplified with the intermediate states, is shown in Table I.

C. Circuit-level Simulation

To verify the functionality of the algorithm we simulated them on circuit-level using LT-SPICE. We used the Voltage-controlled ThrEshold Adaptive Memristor (VTEAM) model [32] implemented in SPICE [11], [33]. The model parameters are set similarly to Table II, which are fitted to a

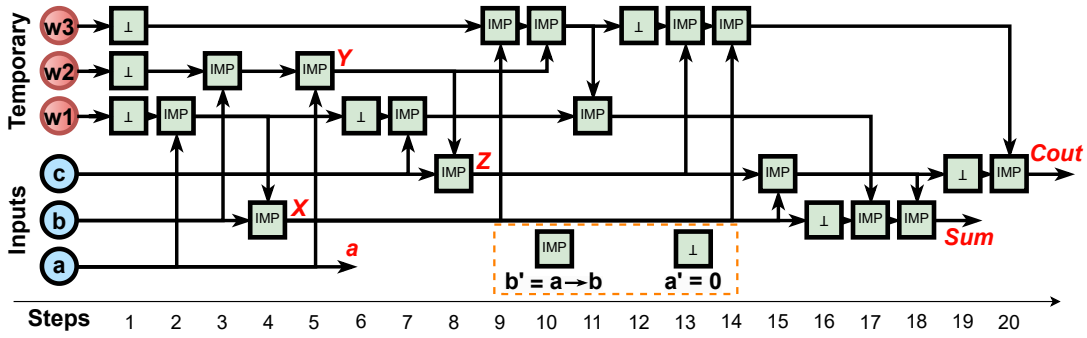


Fig. 2: Flowchart of the presented adder algorithm, where the current state of each memristor at every step is represented in the horizontal line. In steps 7, 8, and 9 the intermediate states X, Y, and Z are computed and stored in the corresponding memristors. These states are then combined as shown in Equation (2) and Equation (3). In step 18, Sum is stored in the b-memristor and C_{out} is saved in the c-memristor at step 20. The initial value of the a-memristor is unchanged.

TABLE I: Proposed Adder Algorithm

Steps	Operation	Equivalent Logic
1	$w_1 = w_2 = w_3 = 0$	False(w_1,w_2,w_3)
2	$w_1' = a \to w_1$	$w_1 = \overline{a}$
3	$w_2' = b \to w_1$	$w_2 = \overline{b}$
4	$b' = w_1' \to b$	$b' = \overline{a} \to b = X$
5	$w_2'' = a \to w_2$	$w_2'' = a \to \overline{b} = Y$
6	$w_1 = 0$	False(w_1)
7	$w_1' = c \to w_1$	$w_1' = \overline{c}$
8	$c' = w_2'' \to c$	$c' = Y \to c = Z$
9	$w_3' = b' \to w_3$	$w_3' = \overline{X}$
10	$w_3'' = w_2'' \to w_3'$	$w_3'' = Y \to \overline{X}$
11	$w_1'' = w_3'' \to w_1'$	$w_1'' = (Y \to \overline{X}) \to \overline{c}$
12	$w_3 = 0$	False(w_3)
13	$w_3' = c' \to w_3$	$w_3' = \overline{Z}$
14	$w_3'' = b \to w_3'$	$w_3'' = X \to \overline{Z}$
15	$c'' = b \to c'$	$c'' = X \to Z$
16	$b = 0$	False(b)
17	$b' = w_1'' \to b$	$b' = \overline{(Y \to \overline{X}) \to \overline{c}}$
18	$b'' = c'' \to b'$	$b'' = (X \to Z) \to \overline{(Y \to \overline{X}) \to \overline{c}} = Sum$
29	$c = 0$	False(c)
20	$c' = w_3'' \to c$	$c' = \overline{X \to \overline{Z}} = C_{out}$

TABLE II: VTEAM setup parameter

Parameter	v_{off}	v_{on}	α_{off}	α_{on}	R_{off}	R_{on}
Value	0.7V	-10mV	3	3	1 MΩ	10 kΩ
k_{on}	k_{off}	w_{off}	w_{on}	w_C	a_{off}	a_{on}
-0.5 nm/s	1cm/s	0 nm	3 nm	107 pm	3 nm	0 nm

TABLE III: IMPLY logic parameter

Parameter	V_{SET}	V_{RESET}	V_{COND}	R_G	t_{pulse}
Value	1V	-2V	900mV	40 kΩ	30 μs

discrete Knowm memristor [34]. This increases the practical relevance of our simulations and allows for an easier comparison, as it is a commonly used model [11], [12], [25]–[27]. We note here that similar to the difference between discrete and integrated CMOS devices, discrete memristors have increased energy consumption and slower operations. It is reasonable to assume that integrated memristors will provide a significant performance improvement. The IMPLY specific parameters we used are shown in Table III.

We simulated all eight different input combinations, which agree with our theoretical calculations. We calculated the average energy consumption by taking the mean of the input possibilities, which resulted in 5.3765nJ per bit. We note that the average energy consumption can substantially differ based on the model, technology, and IMPLY parameters used [11]. These results should rather be used to compare the efficiency of different algorithms. As real memristors show non-ideal behaviors [35] we follow the lead of [26], [28] and simulate the deviation of the resistive states R_{on} and R_{off} via the ATOMIC tool presented in [36]. Our experiments indicate that the results are correct (within the 33% threshold) for a deviation of up to ±30%. An example waveform with a deviation of ±20% is illustrated in Figure 3, to showcase the impact of the changed resistive states.

IV. Comparison with SoA

We compare to SoA adder algorithms in the serial topology on various circuit-level metrics and analyze the advantages gained through preserving the a-input. We do not compare to other topologies in this work as they are optimized for different design goals and an in-depth analysis would be beyond the scope of this work. We note here that in [37], an input-preserving adder was proposed in a deviation of the semi-parallel topology. We do not compare with this approach since their circuit may not be compatible with standard crossbar arrays. An overview of the comparison is shown in Table IV. In terms of energy consumption, our approach is equally efficient as [8] but worse by 12% than [9], [10]. The main advantage of the presented algorithm lies in the number of steps, which is often the limiting factor in the serial topology. Our algorithm is 9%-13% faster than the SoA. Our approach uses one additional memristor compared to [8], [9] to preserve the value of the a-memristor. The adder from [10] requires approximately a third more memristors than the other approaches.

The previous comparisons do not take into account the additional energy consumption and steps that are required to restore the data of input memristors via a `COPY` operation.

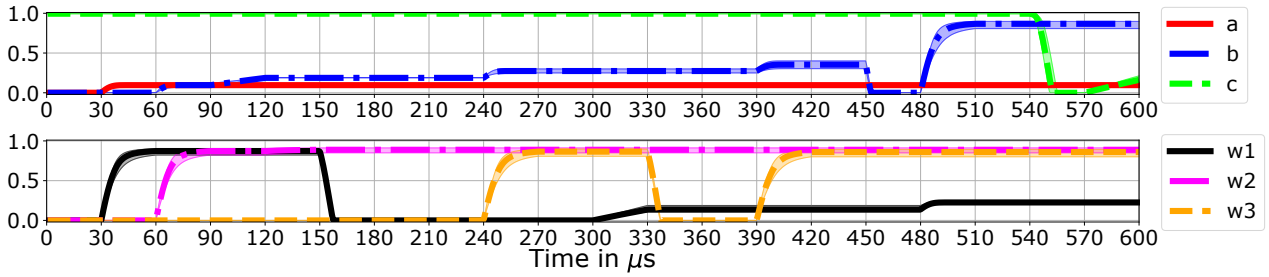


Fig. 3: Example waveforms of the proposed adder algorithm with the inputs abc="001". Sum is stored in the b-memristor between 510-540μs and C_{out} in the c-memristor between 570-600μs. The initial state of the a-memristor is preserved. The shaded area around the waveforms represents a deviation of $\pm 20\%$ of the resistive states.

TABLE IV: Circuit-Level comparison to SoA serial full adder

Full adder	**Energy consumption (nJ)**		**Improvement**	**No. of steps**		**Improvement**	**No. of memristors**		**Improvement**	**Input Preservation**		
	n	n=32-bit	n=32-bit	n	n=32-bit	n=32-bit	n	n=32-bit	n=32-bit	a	b	c_{in}
Serial [10]	4.7092n	150.6944	-12%	23n	736	13%	3n+3	99	31%	✘	✘	✘(Cout)
Serial [9]	4.7166n	150.0931	-12%	23n	736	13%	2n+3	67	-1%	✘(Sum)	✘	✘(Cout)
Serial [8]	5.3964n	172.6848	<1%	22n	704	9%	2n+3	67	-1%	✘	✘(Sum)	✘(Cout)
Proposed	5.3765n	172.0480	-	20n	640	-	2n+4	68	-	✓	✘(Sum)	✘(Cout)

We simulated the algorithms from [8]–[10] with the parameters from Table III to allow for a fair comparison.

We simulated the procedure from Equation (1) and calculated the average energy consumption the same way as explained in Section III-C, which resulted in 0.7147nJ per `COPY`. Each `COPY` requires three steps, which cannot be parallelized as the serial topology only consists of one computational section. To restore the a-input of a n-bit addition, $3n$ additional steps and $0.7147n$ nJ of energy are required. For a n=32-bit adder this would amount to 22.8704nJ and 96 steps. When these operations are considered as part of the addition algorithm, the presented algorithm is now 20%-23% faster and requires 1%-12% less energy than [8]–[10]. It is also the only adder algorithm that is able to make use of each input by either preserving the initial value or storing the Sum or C_{out} for further computational steps. This can be seen on the right of Table IV, where SoA algorithms unnecessarily waste the information of at least one input.

V. Application in Neural Networks

To showcase the advantages of our approach we embed our adder in a shift-and-add multiplier. The memristors are placed on a crossbar array, so we will use array indexing for the following explanations. Since the multiplication output requires twice the bit width of the n inputs, we initialize B with $2n$ memristors that are set to logical '0'. We define A as an array that encompasses all a-memristors in a decreasing order. The multiplication consists of n additions that can be represented via the recursive operation:

$$B_{k+1}[k+n+1:k] = [0, B_k[k+n:k+1]] + A \cdot b_k \quad (7)$$

where $k \in [0, n]$. The content of the memristors in A is repeatedly added to the shifted window of memristors in B when b_k is logical '1'. In NN applications the inputs of each layer are multiplied by weights and summed up, which can be expressed as `MAC` operations. At inference, the weights should either be preserved (our approach) or must be reloaded before the next inference. The same applies to Convolutional Neural Network (CNN), where individual kernel weights are multiplied by the inputs and then summed up. As we presented an integer-based adder and multiplier we want to apply our approach to NN that are quantized to 8-bit for integer-arithmetic-only inference [38]. This means no floating point operations are required. We require 16 memristors as accumulators and 16 memristors that are used to handle the carry-out for an 8×8-bit into 32-bit `MAC` operation. For full inference, only value clipping and shifting are further required [38], [39], which will be handled by the control logic outside the crossbar array. In this example, our approach would lead to $14\% - 17\%$ more energy efficient and $17\% - 20\%$ faster `MAC` operations when compared to conventional serial adder [8]–[10]. The inference of ResNet18 [40] requires roughly 3.6×10^9 `MAC` operations when we assume that every floating-point operation is equal to two `MACs`. When our adder is used, 1.4×10^{12} to 1.7×10^{12} cycles can be saved in total, when compared to [8]–[10]. We note here that ResNet18 is only used as a reference since it is commonly used and well-known. Our approach will yield higher gains when applied in bigger networks or when multiple inferences are computed consecutively.

VI. Conclusion

In this work, we presented an improved adder algorithm in the serial IMPLY topology that is faster than SoA adder while also preserving one input. This is visualized with a proposed flowchart. When the inefficient reloading of the input is taken into account our adder is $20\% - 23\%$ faster and $1\% - 12\%$ more energy efficient. On the quantized inference of ResNet18, our adder requires up to 20% fewer cycles and up to 17% less energy compared to SoA adder. Input-preserving adder algorithms in other topologies are domains of future work.

REFERENCES

[1] N. TaheriNejad. In-memory computing: Global energy consumption, carbon footprint, technology, and products status quo. pp. 1–6, 2024.

[2] H. A. D. Nguyen *et al.* A classification of memory-centric computing. *J. Emerg. Technol. Comput. Syst.*, 16(2), jan 2020.

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

[4] E. Lehtonen and M. Laiho. Stateful implication logic with memristors. *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 33–36, 2009.

[5] M. R. Alam *et al.* Exact stochastic computing multiplication in memristive memory. *IEEE Design Test*, pp. 1–8, 2021.

[6] M. R. Alam *et al.* Sorting in memristive memory. *ACM Journal on Emerging Technologies in Computing Systems*, pp. 1–22, 2022.

[7] C. Li *et al.* In-memory computing with memristor arrays. In *2018 IEEE International Memory Workshop (IMW)*, pp. 1–4, 2018.

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

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

[10] M. Teimoory *et al.* Optimized implementation of memristor-based full adder by material implication logic. In *2014 21st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 562–565, 2014.

[11] N. TaheriNejad *et al.* A semi-serial topology for compact and fast imply-based memristive full adders. In *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, 2019.

[12] S. Ganjeheizadeh Rohani *et al.* A semiparallel full-adder in imply logic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):297–301, 2020.

[13] L. O. Chua. Memristor—the missing circuit element. *IEEE Transactions on Circuit Theory*, CT-18(5):507–519, September 1971.

[14] D. B. Strukov *et al.* The missing memristor found. *Nature*, 453:80–83, May 2008.

[15] J. Borghetti *et al.* 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464:873–876, April 2010.

[16] E. Lehtonen and M. Laiho. Stateful implication logic with memristors. In *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, 2009.

[17] D. Radakovits *et al.* A memristive multiplier using semi-serial imply-based adder. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(5):1495–1506, 2020.

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

[19] N. TaheriNejad. Sixor: Single-cycle in-memristor xor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(5):925–935, 2021.

[20] S. Kvatinsky *et al.* Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[21] P. Huang *et al.* Reconfigurable nonvolatile logic operations in resistance switching crossbar array for large-scale circuits. *Advanced Materials*, 28(44):9758–9764, 2016.

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

[23] S. Kvatinsky *et al.* Memristor-based material implication (imply) logic: Design principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2014.

[24] D. Radakovits and N. Taherinejad. Behavioral leakage and inter-cycle variability emulator model for rerams (BELIEVER). *CoRR*, abs/2103.04179, 2021.

[25] F. Seiler and N. TaheriNejad. An imply-based semi-serial approximate in-memristor adder. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pp. 1–7, 2023.

[26] F. Seiler and N. TaheriNejad. Accelerated image processing through imply-based nocarry approximated adders. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14, 2024.

[27] S. E. Fatemieh *et al.* Fast and compact serial imply-based approximate full adders applied in image processing. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 13(1):175–188, 2023.

[28] 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.

[29] V. Lakshmi *et al.* A novel in-memory wallace tree multiplier architecture using majority logic. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(3):1148–1158, 2022.

[30] K. Bickerstaff and E. E. Swartzlander. Memristor-based arithmetic. In *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, pp. 1173–1177, 2010.

[31] E. Lehtonen et al. Two memristors suffice to compute all boolean functions. *Electronics letters*, 46(3):230, 2010.

[32] S. Kvatinsky *et al.* Vteam: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(8):786–790, 2015.

[33] D. Radakovits *et al.* Second (v2.0) ltspice implementation of vteam. Last accessed Aug 2024.

[34] Knowm sdc memristors, knowm.org/downloads/Knowm_Memristors.pdf. Last accessed Feb 2024.

[35] N. TaheriNejad and D. Radakovits. From behavioral design of memristive circuits and systems to physical implementations. *IEEE Circuit and Systems (CAS) Magazine*, 19(4):6–18, Fourthquarter 2019.

[36] F. Seiler and N. TaheriNejad. Atomic: Automatic tool for memristive imply-based circuit-level simulation and validation, 2024.

[37] X. Hu *et al.* A data non-destructive imply-based memristive semi-parallel full-adder for computing-in-memory systems. *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, 2021.

[38] B. Jacob *et al.* Quantization and training of neural networks for efficient integer-arithmetic-only inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2017.

[39] D. Schnöll *et al.* Fast, quantization aware dnn training for efficient hw implementation. In *2023 26th Euromicro Conference on Digital System Design (DSD)*, pp. 700–707, 2023.

[40] K. He *et al.* Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.