Accelerated Addition in Resistive RAM Array Using Parallel-Friendly Majority Gates

John Reuben, *Member, IEEE*, Stefan Pechmann, *Graduate Student Member, IEEE*

***Abstract*—To overcome the 'von Neumann bottleneck', methods to compute in memory are being researched in many emerging memory technologies including Resistive RAMs (ReRAMs). Majority logic is efficient for synthesizing arithmetic circuits when compared to NAND/NOR/IMPLY logic. In this work, we propose a method to implement a majority gate in a transistor-accessed ReRAM array during READ operation. Together with NOT gate, which is also implemented in-memory, the proposed gate forms a functionally complete Boolean logic, capable of implementing any digital logic. Computing is simplified to a sequence of READ and WRITE operations and does not require any major modifications to the peripheral circuitry of the array. While many methods have been proposed recently to implement Boolean logic in memory, the latency of in-memory adders implemented as a sequence of such Boolean operations is exorbitant (*O(n)*). Parallel-prefix adders use prefix computation to accelerate addition in conventional CMOS-based adders. By exploiting the parallel-friendly nature of the proposed majority gate and the regular structure of the memory array, it is demonstrated how parallel-prefix adders can be implemented in memory in *O(log(n))* latency. The proposed in-memory addition technique incurs a latency of *4·log(n)+6* for n-bit addition and is energy-efficient due to absence of sneak-currents in 1Transistor–1Resistor configuration.**

***Index Terms*—Resistive RAM (ReRAM), Non-volatile Memory (NVM), majority logic, majority gate, memristor, 1Transistor-1Resistor (1T–1R), von Neumann bottleneck, in-memory computing, sense amplifier, processing-in-memory, parallel-prefix adder, logic-in-memory, compute-in-memory, read-out circuit**

I. Introduction

THE movement of data between processing and memory units is the major cause for the degraded performance (both energy and latency-wise) of contemporary computing systems, often referred to as the 'von Neumann bottleneck' or 'memory wall'. 'Computation energy' is dominated by 'data movement energy' since the energy for memory access grows exponentially along the memory hierarchy (from cache to off-chip DRAM) [2]. As a quantitative example, [3] points out that the energy for DRAM access is 3556× the energy for 16-bit addition in 45 nm CMOS technology. Similarly, DRAM access latency is ≈ 100 ns [4], while latency of 32-bit adder is 4 ns in CMOS technology [5], implying that data movement latency forms a significant portion of computation latency in conventional von-Neumann computing model. Consequently, there had been many efforts in the last 10-15 years to combat the memory wall by bringing the processor and memory unit closer together. For example, 3D stacking of DRAM dies over logic die (enabled by Through-Silicon-Via technology) was used to reduce the energy and latency of data movement between processor and memory, in what was called near-memory computing. Going a step further, there have been efforts to move computing not just near memory, but to the memory itself $i.e.$ the memory array.

ReRAMs are two terminal devices (usually a Metal-Insulator-Metal structure) capable of storing data as resistance. When subject to voltage stress, the resistance can be switched between a Low Resistance State (LRS) and a High Resistance State (HRS). The change of resistance is due to the formation or rupture of a conductive filament, depending on the direction of the current flow through the structure. The word 'memristor' is also used by researchers to denote such a device, because it is essentially a resistor with memory. However, it must be noted that the word memristor can refer to a broader class of devices which have the capability to change their resistance in response to voltage/current stress ($e.g.$ Phase Change Memory (PCM), Spin Transfer Torque-Magnetic RAM (STT-MRAM)).

Connecting such ReRAM devices in a certain manner, or by applying certain voltage patterns, or by modifying the sensing circuitry, basic Boolean gates (NOR, NAND, XOR, IMPLY logic) have been demonstrated in ReRAM arrays [6]–[13]. The motivation for such efforts is to perform Boolean operations on data stored in the memory array, without moving them out to a separate processing circuit, thus mitigating the von Neumann bottleneck. Although such Boolean operations are being explored in DRAM [14], [15] and SRAM [16], [17], emerging NVMs like ReRAM bring high density on-chip, which SRAM lacks. Reviews of in-memory computing approaches in emerging NVMs are presented in [18], [19]. To construct a memory array using such devices, two configurations are common: 1Transistor–1Resistor (1T–1R) and 1Selector–1Resistor (1S–1R). The 1T–1R configuration uses a transistor as an access device for each cell, isolating the accessed cell from its neighbours in the array. The 1S–1R configuration uses a two-terminal device called a 'selector' which is fabricated in series with the memristive device. The 1S–1R is area-efficient, but suffers from current leakage (sneak–path problem) due to the inability to access a particular

This is an extended version of the paper presented in IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), Manchester, United Kingdom, 2020 [1]

John Reuben is with Chair of Computer Architecture, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 91058 Erlangen, Germany. (email:johnreubenp@gmail.com / johnreuben.prabahar@fau.de)

Stefan Pechmann is with Chair of Communications Electronics, Universität Bayreuth, 95447 Bayreuth, Germany. (e-mail:stefan.pechmann@uni-bayreuth.de).

Manuscript received October 2, 2020; revised January 23, 2021 and February 19, 2021; accepted March 21, 2021.

Digital Object Identifier 10.1109/TVLSI.2021.3068470

cell without interfering with its neighbours [20].

Majority logic, a type of Boolean logic, is defined to be true if more than half of the n inputs are true, where n is odd. Hence, a majority gate is a democratic gate and can be expressed in terms of Boolean AND/OR as $MAJ(a,b,c) = a \cdot b + b \cdot c + a \cdot c$, where a, b, c are Boolean variables. Although majority logic was known since 1960, there has been a revival in using it for computation in many emerging nanotechnologies (spin waves, magnetic Quantum-Dot cellular automata, nano magnetic logic, Single Electron Tunneling [21]–[23]). Recent research [22]–[25] has confirmed that majority logic is to be preferred not only because a particular nanotechnology can realize it, but also because of its ability to implement arithmetic-intensive circuits with less gates. In this paper, we propose a majority gate whose structure is conducive for parallel-processing in the memory array. When three rows of a 1T–1R array are activated simultaneously, the resistance of the ReRAM cells in a column will be in parallel during the READ operation. A Sense Amplifier (SA) which can accurately sense the effective resistance implements an 'in-memory' majority gate. Then, a method to implement a NOT gate in memory is proposed. MAJORITY together with NOT is functionally complete and any Boolean logic can be expressed in terms of them.

The peripheral circuitry around the array facilitates in-memory computing. Therefore, we design the peripheral circuitry and the modification required in it to accommodate the execution of MAJORITY and NOT gates. Latency of in-memory adders is a compelling problem $i.e.$ numerous cycles of Boolean operations are required to implement any arithmetic circuit in memory. If the latency of in-memory arithmetic is not optimized carefully, it may take longer to compute in memory than the combined time to fetch data from memory and compute in a CMOS-based processor. Adders are the fundamental unit of any computing system. While many works have been reported to implement adders in memory, the issue of latency has not been carefully studied and optimized. Consequently, many in-memory adders have a $O(n)$ latency for n-bit addition and practical adders of 32-bit/64-bit in memory require hundreds of cycles. In this work, we focus on adders and tackle this exorbitant latency of in-memory adders. We couple the strength of majority logic with the array structure of the memory to implement fast parallel-prefix (PP) adders in memory array.

Our main contributions can be summarized as follows:

1) We propose a method to implement a majority gate in memory which is conducive for parallel-processing
2) We present the complete peripheral circuitry around the array to perform in-memory arithmetic
3) We consider an 8-bit PP adder synthesized in majority logic and implement it in the memory array using the proposed gates in $O(log(n))$ latency
4) We extend the 8-bit in-memory adder to 16-bit and thus formulate the latency, energy and area of the array required for n-bit in-memory addition

The rest of the paper is organized as follows. In Section II, we justify why we chose majority logic as the logic primitive to minimize the latency of adders in memory. Section III presents the principle of reading majority from a 1T–1R array and the detailed sensing methodology. In Section IV, we present the framework to compute in the memory using the proposed majority gate by elaborating the peripheral circuitry of the memory array. Section V briefly introduces parallel-prefix technique and discusses how an 8-bit parallel-prefix adder can be implemented in a 1T–1R array using the proposed majority and NOT gates. We extend the 8–bit in-memory adder to 16–bit adder to ascertain how our adder performs with increasing bit-width. In this manner, the latency, area and energy for n-bit addition are formulated as analytic expressions in Section VI. We compare the proposed in-memory addition technique with the state-of-the-art in Section VII, followed by conclusions in Section VIII.

II. Motivation for choosing Majority Logic as the logic primitive for in-memory computing

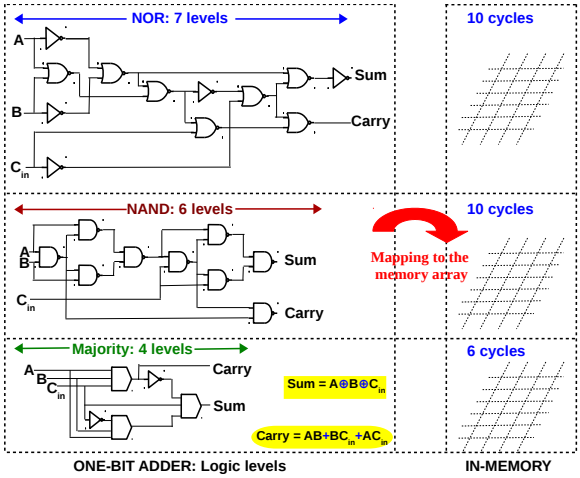

Fig. 1: In-memory implementation does not favor heterogeneity of logic primitives [26]. Arithmetic circuits are synthesized in terms of a particular logic primitive which can be implemented with minimal modifications to the conventional memory array. In view of this, approaches to implement MAJORITY gate and NOT (to make it functionally complete) are to be favored for latency-optimized in-memory computing.

In the past, different logic primitives like NAND, NOR, IMPLY, XOR have been used by researchers to implement Boolean logic in memory. Arithmetic circuits can be implemented as a sequence of these Boolean operations. An important requirement is that the logic primitive must be 'functionally complete'. NOR is 'functionally complete' since any Boolean logic can be expressed in terms of NOR gates. Similarly, NAND, IMPLY+FALSE [27] and MAJORITY+NOT [24] are also functionally complete. Since a conventional memory array supports only READ and WRITE operations, these logic primitives necessitate some enhancements to the conventional array. This enhancement can be some modification to the array structure, or the peripheral circuitry or a combination of these. With these modifications, the memory array is enhanced to be able to execute a logic primitive along with READ and WRITE operations. Consequently, for in-memory implementation, an arithmetic circuit is synthesized

in terms of that particular logic primitive and mapped to the memory array. Therefore, unlike CMOS implementation which accommodated heterogeneity of logic gates (NAND, NOR, XOR), in-memory implementation usually accommodates homogeneity of gates (only NOR or only NAND). A 1-bit full adder implemented in memory using NOR [28], NAND [29] and MAJORITY [30] are compared in Fig. 1. It is evident that majority logic could achieve 1-bit adder functionality with less logical depth (latency) than NAND/NOR. Furthermore, notice that k-levels of logic gets expanded to $k + x$ cycles in memory (interconnections between logic levels contribute to x additional cycles, [26]). Therefore, for latency-optimized in-memory implementation, it is important to choose a logic primitive which minimizes k, the number of logic levels.

Recent research in logic synthesis confirms the aforementioned trend (observed in 1-bit full adder) for other circuits as well. In [24], the authors present Majority-Inverter Graph (MIG), a new logic manipulation structure consisting of three-input majority nodes and regular/inverted edges. Logic functions are represented by MIGs and further optimized using both algebraic and Boolean methods, summarized in [23], [24]. A selection of circuits from both IWLS'05 benchmarks and HDL arithmetic benchmarks were considered and synthesis results obtained with MIG optimization tool were compared to And-Inverter Graphs (AIG) optimized by ABC in terms of logical levels. For IWLS'05 benchmarks, an average 14% reduction in logical depth and for arithmetic HDL benchmarks, about 33% reduction in logical depth were obtained by MIG compared to AIG [23], [24]. This motivated the authors to pursue majority logic as the fundamental logic primitive to minimize the latency of in-memory adders.

III. Majority Gate in 1T–1R Array

A. Majority gate: Operating principle

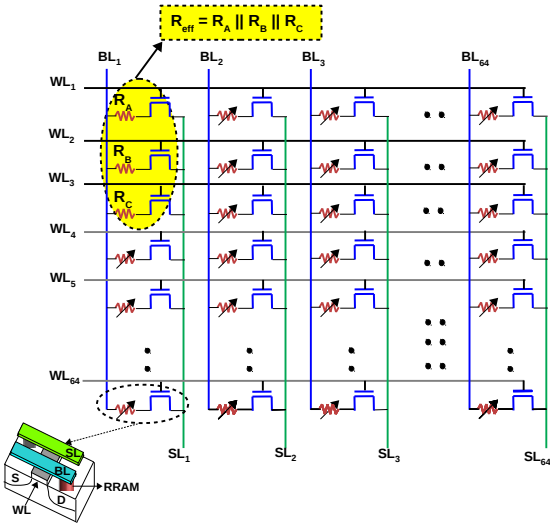

Fig. 2: When three rows are activated (WL_{1-3}) simultaneously in a 1T-1R array, the resistances of the three ReRAM devices are in parallel. An 'in-memory' majority gate can be implemented by accurately sensing the effective resistance R_{eff}.

Consider an array of ReRAM cells arranged in a 1T-1R configuration, as depicted in Fig. 2. Each cell can be individually read/written to by activating the corresponding wordline (WL) and applying appropriate voltage across the cell (BL and SL). To read from a cell, the corresponding WL is activated, a small current is injected into the cell and the voltage across the cell is sensed in a voltage-mode SA $i.e.$ the BL voltage is sensed while the SL is grounded. Now, if three rows are activated simultaneously during read operation (Rows 1 to 3 in Fig. 2), the resistances in column 1 are in parallel (neglecting the parasitic resistance of BL and SL). During read, the access transistor will be in linear region, and hence the transistor's resistance will be ([31])

$$r_t = \frac{1}{\mu_n C_{ox}(\frac{W}{L})(V_{GS} - V_t)} = 544 \quad \Omega. \tag{1}$$

The effective resistance between BL and SL will therefore be

$$R_{eff} = (R_A + r_t)||(R_B + r_t)||(R_C + r_t) \approx (R_A||R_B||R_C), \tag{2}$$

if the drain-to-source resistance of transistor (r_t) is small compared to LRS. Table I lists the truth table of 3-input majority gate ($M_3(A, B, C)$) and the effective resistance for all the eight possibilities. To verify the proposed gate on a real ReRAM device, we choose the 1T-1R cell from IHP[1]. The 1T–1R structure consists of a NMOS transistor having a (W/L) of (150 nm/130 nm). The drain of the access-transistor is connected in series to the ReRAM. The ReRAM is a $TiN/Hf_{1-x}Al_xO_y/Ti/TiN$ stack integrated between Metal2 and Metal3 in the BEOL of the CMOS process. The cells have a mean LRS and HRS of 10 kΩ and 133.3 kΩ, respectively. Therefore, the R_{eff} is $\geq$ 8.7 kΩ when two or more cells are in HRS (shaded grey in Table I) and $\leq$ 4.8 kΩ when two or more cells are in LRS. Consequently, a majority gate can be implemented during a READ operation by precisely sensing R_{eff}. This manner of computing majority enables parallelism since multiple gates can be executed in the columns of the array. The proposed gate is energy-efficient (both reading and writing is energy-efficient in 1T–1R when compared to 1S–1R arrays due to the absence of sneak paths [32]). As can be deciphered from Table I, the crucial aspect of the proposed gate is to be able to differentiate between R_{eff}^{001} (two LRS and one HRS) and R_{eff}^{011} (two HRS and one LRS). Let's denote the resistance to be differentiated as sensing window. Sensing window for majority = 8.7 kΩ – 4.8 kΩ = 3.9 kΩ for IHP's cell (resistance window = 13.3).

B. Sensing methodology

The methodology to reliably translate R_{eff} into a CMOS-compatible voltage is the crucial aspect of the proposed majority gate. R_{eff}^{001} is 4.8 kΩ and R_{eff}^{011} is 8.7 kΩ, and differentiating such a resistance window ($\approx$ 3.9 kΩ) needs a robust SA. It must be noted that this will be exacerbated by the variability exhibited by the ReRAM devices. To meet this requirement, a time-based SA recently proposed in [33] was chosen and adapted to our requirement. Different from conventional sensing schemes (voltage-mode and current-mode),

[1]Innovations for High Performance Microelectronics– Leibniz-Institut für innovative Mikroelektronik, Germany

TABLE I: Precisely sensing R_{eff} results in majority: Logic '0' is LRS (10 kΩ) and logic '1' is HRS (133.3 kΩ)

A	B	C	$M_3(A,B,C)$	R_{eff}	R_{eff}
0	0	0	**0**	$\frac{LRS}{3}$	3.3 kΩ
0	0	1	**0**	$\frac{HRS \cdot LRS}{LRS+2 \cdot HRS}$	4.8 kΩ
0	1	0	**0**	$\frac{HRS \cdot LRS}{LRS+2 \cdot HRS}$	4.8 kΩ
0	1	1	**1**	$\frac{HRS \cdot LRS}{HRS+2 \cdot LRS}$	8.7 kΩ
1	0	0	**0**	$\frac{HRS \cdot LRS}{LRS+2 \cdot HRS}$	4.8 kΩ
1	0	1	**1**	$\frac{HRS \cdot LRS}{HRS+2 \cdot LRS}$	8.7 kΩ
1	1	0	**1**	$\frac{HRS \cdot LRS}{HRS+2 \cdot LRS}$	8.7 kΩ
1	1	1	**1**	$\frac{HRS}{3}$	44.4 kΩ

the time-based sensing scheme converts the BL voltage (to be sensed) into a time delay and discriminates in time-domain. This sensing scheme was originally proposed to read data from STT-MRAM [33], which has a resistance margin of a few kΩ. Therefore, it is well suited for the proposed majority gate. Furthermore, this time-based sensing achieves two to three orders of magnitude improvement in sensing (BER) compared to conventional schemes, in addition to being reference-less [33]. 'Read-Disturb' in ReRAM refers to the perturbation of the conductive filament during the data reading phase resulting in an unintended change of the stored memory state [34]. Experimental works [35], [36] suggest that read-disturb is severe only when the READ voltage across the ReRAM cell is $>$ 0.3 V. To avoid read-disturb, we designed the time-based SA such that it can sense R_{eff} correctly for an $I_{READ} \leq 30$ μA ($I_{READ} > 30$ μA could have provided a better sensing margin, but would have made the SA susceptible to read-disturb).

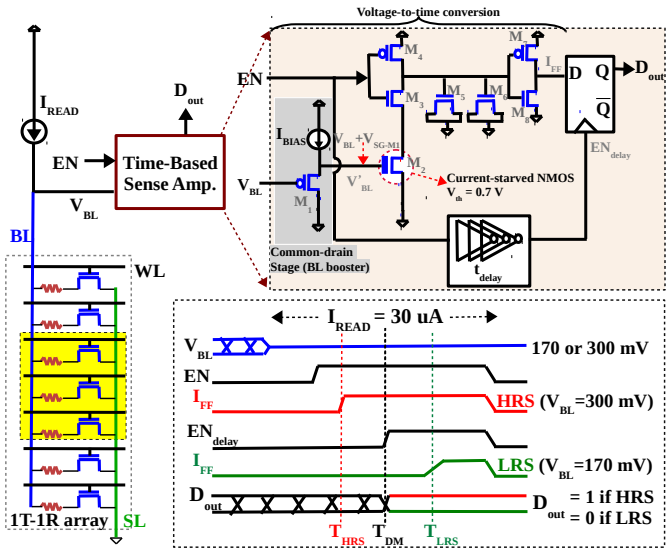

Fig. 3: A small current I_{READ} injected into the cell converts the resistance to a voltage which is fed to the time-based SA. A current-starved inverter transforms this voltage into a proportional delay which is sensed as a CMOS-compatible voltage by the D-FF [33].

The time-based sensing circuit is essentially a voltage-to-time converter followed by a time-domain comparator (D-flip flop). Voltage-to-time conversion is achieved in three stages– a common-drain stage, a current-starved inverter (M_{2-4}) and a pulse-shaping inverter (M_{5-8}), as depicted in Fig. 3. During READ, a current I_{READ} is injected into the 1T-1R cell (corresponding three WLs are activated and SL is grounded). Depending on the effective resistance R_{eff}, the BL reaches a voltage ($V_{BL} \approx I_{READ} \times R_{eff}$). Read-disturb phenomenon [34]–[36] imposes a constraint that voltage across cell must be $\leq$ 0.3 V during READ. Hence a common-drain stage is used to boost the BL voltage to a level suitable for the following stage. After BL voltage is boosted to $V_{BL} + V_{SG-M_1}$, it is used to modulate the gate voltage of M_2 (M_{2-4} forms a current-starved inverter). For best sensing results, the voltage at the gate of M_2 should be $\geq V_{th-M2}$ for R_{eff}^{011} case so that it turns on sharply and introduces less delay in EN signal. For R_{eff}^{001} case, the voltage at the gate of M_2 should be $< V_{th-M2}$ so that it significantly limits the inverter current, and introduces more delay in the EN signal. A high-threshold NMOS transistor from IHP's process[2] was used as M_2 to achieve this functionality effectively. In this manner, V_{BL} is converted to a delay in the EN signal which is further shaped by M_7–M_8 inverter to have a sharp rising edge at the input of the D-flip flop (I_{FF}). At the rising edge of EN_{delay}, I_{FF} is available at the output of the SA.

The time-based sensing circuit of Fig. 3 was designed in IHP's 130 nm CMOS process, and simulated to verify the functioning of the majority gate. I_{READ} of 30 μA was used resulting in a V_{BL} of a few hundred mV (150–300 mV), as shown in Fig. 3. M_1 biased by I_{BIAS} of 1 μA works as a common-drain stage and shifts the BL voltage by the necessary source-gate voltage of M_1 to conduct I_{BIAS}. The amount of voltage by which V_{BL} is shifted can be tuned by tuning I_{BIAS} and the W/L-ratio of M_1 and it was tuned to be 0.45 V *i.e.* $V_{BL}^{'}$ = V_{BL}+ 0.45 V in Fig. 3. When EN goes high, the current-starved inverter introduces a delay proportional to $V_{BL}^{'}$ *i.e.* a higher $V_{BL}^{'}$ incurs less delay. In this manner, a V_{BL} of 300 mV and 170 mV ($V_{BL}^{'}$ = 0.75 V and 0.62 V) results in the I_{FF} rising high at T_{HRS} and T_{LRS}, respectively. t_{delay} is a chain of inverters designed to delay the EN signal such that T_{DM} (Decision Moment) is between T_{HRS} and T_{LRS}. When EN_{delay} goes high at T_{DM}, it latches the signal at I_{FF} and hence the D_{out} is high for high resistance (R_{eff}^{011} = 8.7 kΩ) and low for low resistance (R_{eff}^{001} = 4.8 kΩ). It must be noted that for R_{eff}^{111} = 44.4 kΩ, I_{FF} goes high before T_{HRS} and, for R_{eff}^{000} = 3.3 kΩ, I_{FF} goes high after T_{LRS}. Therefore, once designed to differentiate between R_{eff}^{011} and R_{eff}^{001}, the time-based SA will output $M_3(A,B,C)$ correctly for all the eight cases. Furthermore, the same SA can be used to read a single bit by using a smaller I_{READ} (and activating a single WL during normal read operation). Hence the proposed gate does not necessitate any modification to the read-out circuit of the regular memory array.

[2]The high-threshold NM0S variant of IHP 130 nm process has a V_{th} of 0.7 V as opposed to the regular NMOS from the same process which has a V_{th} of 0.45 V. Since M_2 is the crucial factor in deciding the delay (and consequently the SA's output), using a high-threshold transistor additionally achieves better immunity against CMOS process variations.

C. Robustness of the majority gate against ReRAM variations

ReRAM cells exhibit variability in their programmed resistive states cycle-to-cycle and device-to-device [37] and the majority gate needs to be evaluated in the presence of these variations. The variability in resistance states is normalized by the mean and expressed by co-efficient of variation $\frac{\sigma}{\mu}$, where σ is the standard deviation from μ, the mean resistance of the state. Furthermore, many experimental works have revealed that variability is larger at HRS than at LRS [38]–[40] due to stochastic nature of the filament rupture[3]. Since the ReRAM is not switched while computing majority, the 1T–1R cell was modeled as a transistor in series with a resistor[4] and ReRAM variability was incorporated as a Gaussian distribution in that resistor. Since exact variability of IHP's devices were not available, we considered a variability, (σ_{LRS}/μ_{LRS}) of 12.6 % and (σ_{HRS}/μ_{HRS}) of 20.9 %, which is the statistically reported variability for a similar HfO_x device [38]. 50000 Monte Carlo simulations were performed where the resistances R_A, R_B, R_C were Gaussian distributed to reflect the ReRAM variability $i.e.$ $R_{eff}^{001} = LRS||LRS||HRS$ and $R_{eff}^{011} = LRS||HRS||HRS$ are calculated from the Gaussian distributed LRS/HRS with aforementioned μ and σ. The impact of process variations was analysed using the statistical model files for the CMOS transistors provided by the foundry. With combined effects of ReRAM variability and process variability (in transistors of SA and access-transistor of ReRAM), the Bit Error Rate (BER) was found to be $5{\times}10^{-4}$. Sample wave-forms at the input of the D-flip flop are plotted in Fig. 4-(a). The time-based SA achieves clear distinction between R_{eff}^{001} and R_{eff}^{011} in the presence of ReRAM variations and CMOS process variations. Fig. 4-(b) depicts the probability density function (PDF) of R_{eff}^{011} and R_{eff}^{001} during MC runs. It can be observed that, although σ_{LRS} and σ_{HRS} are 1.26 kΩ and 28 kΩ respectively, $\sigma_{R_{eff}^{001}}$ and $\sigma_{R_{eff}^{011}}$ are reduced to 0.4 kΩ and 0.96 kΩ, respectively. This is because R_{eff} is dominated by LRS (equivalent resistance of three resistance in parallel is lower than the least of the three resistance). Hence, R_{eff} is affected more by variation at LRS than at HRS. LRS variation is well controlled by the compliance current in 1T–1R configuration [31]. Therefore, our majority gate has reasonably good immunity to ReRAM variations.

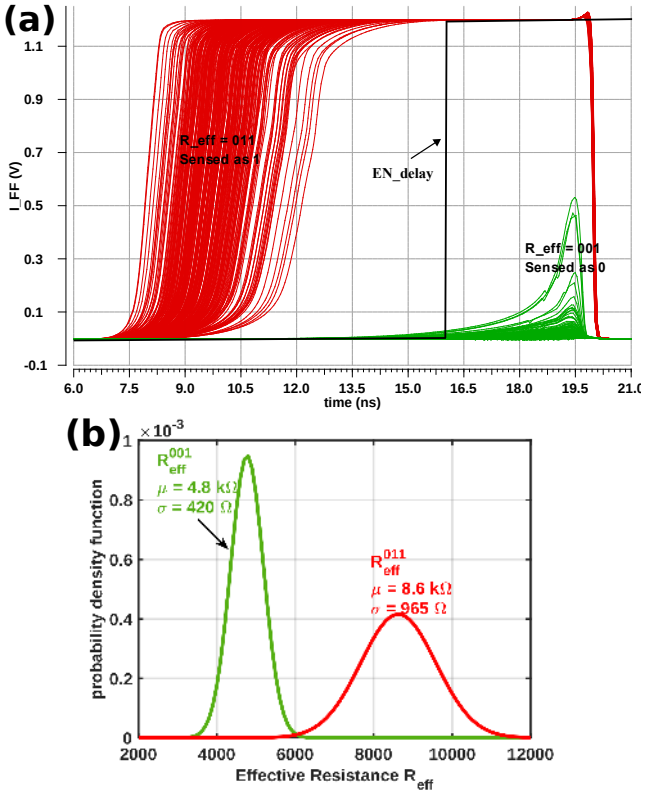

Fig. 4: (a) Sample MC simulation wave-forms of the time-based SA. At 16 ns, the EN_{delay} (clock of the D flip-flop) goes high deciding the output. MC simulations include both CMOS process variations (in the SA and the access-transistor of ReRAM) and ReRAM variations. (b) PDF of R_{eff}^{011} and R_{eff}^{001} over 50000 random runs indicates a tight distribution.

IV. Framework to compute in 1T–1R array

A. Functional completeness

As shown in Fig. 5-(a), NOT operation can be implemented in a 1T–1R array by activating a single row and latching $\overline{Q}$ from the output of the time-based SA during READ (D-Flip flop of Fig. 3 outputs Q and $\overline{Q}$). This is accomplished by using a control signal INV which is low during READ and majority operation (Q is latched) and goes high only during NOT operation ($\overline{Q}$ is latched). During majority operation, the SA has to distinguish between 4.8 kΩ and 8.7 kΩ, while during READ and NOT operations, the SA has to distinguish between 10 kΩ (LRS) and 133 kΩ (HRS). Hence I_{READ} is scaled down to 2.5 μA to maintain low BL voltages while reading a single bit. Majority together with NOT is functionally complete $i.e.$ any Boolean logic can be expressed in terms of majority and NOT gates [24]. As stated in Section II, MIG is a new logic manipulation structure consisting of three-input majority nodes and regular/inverted edges. Fig. 5-(b) is the MIG of a 1-bit full adder obtained by MIGhty (MIG synthesis tool) and, any Boolean logic can be synthesised in terms of majority and NOT gates in a similar manner. Since both majority and NOT gates are implemented as READ, multiple levels of gates can be cascaded by writing the read data back to the array. In essence, 'computing' is simplified to a sequence of READ and WRITE operations, orchestrated by the memory controller, as depicted in Fig. 5-(c). A memristive logic family formulates a functionally complete Boolean logic using a memristive device (ReRAM/PCM/STT-MRAM) as the computing device. The proposed method of implementing a majority and NOT gate in a 1T–1R array forms a new memristive logic family.

B. Triple-row decoder design

A conventional decoder for a 1T–1R array can select one row at a time, while the proposed majority gate needs three

[3]Few studies have compared the actual variation in HRS and LRS. For e.g, in a HfO_x device, the reported variation is 16% at LRS and 36% at HRS. For TiO_x device, it is 14% at LRS and 26% at HRS [40]

[4]The ReRAM cell was modelled as a resistor to verify the functioning of a single majority gate, it was later modelled by Stanford-PKU model to verify 8-bit addition in memory, Section V-C

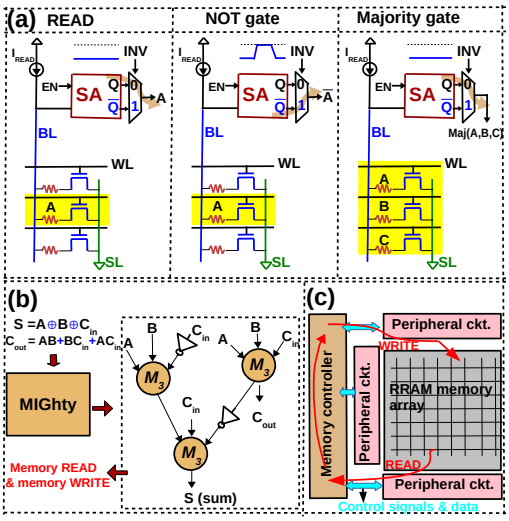

Fig. 5: (a) NOT operation implemented with a 2:1 Mux at the output of the time-based SA; all logic operations are essentially READ operations (b) 1-bit full adder expressed as Majority-Inverter-Graph, where M_3 represents 3-input majority operation (c) With majority/NOT gate computed as READ, multiple levels of logic can be executed by writing the data back to the memory, simplifying computing to READ and WRITE operations.

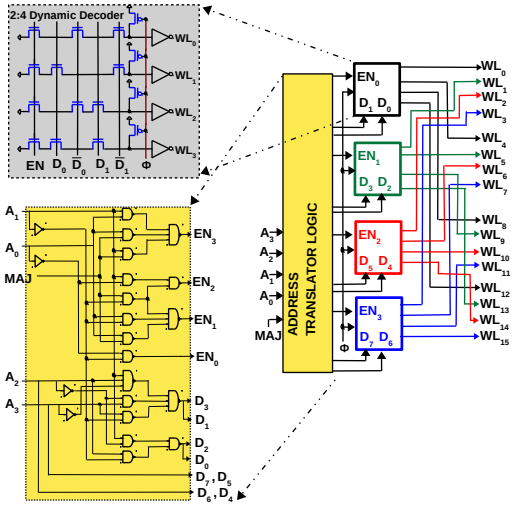

Fig. 6: Triple-row decoding is achieved by interleaving multiple single-row decoders. When control signal MAJ is logic '0' (READ/WRITE/NOT), WL_i corresponding to row address $A_3A_2A_1A_0$ is selected. When MAJ is logic '1' (majority), WL_i, WL_{i+1}, WL_{i+2} are selected.

rows to be selected simultaneously. Moreover, the row-decoder must be versatile to switch between single-row activation and triple-row activation seamlessly. This is because, as stated in the previous section, one must be able to read/write a single bit of the array (READ/WRITE/NOT) as well as read three bits in a column (majority). To this end, we propose a robust row decoder which is designed by interleaving multiple single-row decoders. As depicted in Fig. 6, a 4:16 triple-row decoder can be designed by interleaving four 2:4 dynamic $NAND$ decoders[5]. Since single-row decoding must co-exist with triple-row decoding, an address translator circuit is used to switch between the two modes using MAJ as a control signal. For example, to select a single row WL_5, the address is $A_3A_2A_1A_0$ = '0101' and MAJ = '0'. For these inputs, the address translator outputs $EN_3EN_2EN_1EN_0$ = '0010' and $D_7D_6D_5D_4D_3D_2D_1D_0$ = 'XXXX01XX' (green decoder in Fig. 6 is enabled and it's second row is selected, thereby activating WL_5). But, for the same row address $A_3A_2A_1A_0$ = '0101' and MAJ = '1', the address translator outputs $EN_3EN_2EN_1EN_0$ = '1110' and $D_7D_6D_5D_4D_3D_2D_1D_0$ = '010101XX' (blue, red and green decoders are enabled and second row of each of them is selected, thereby activating WL_5, WL_6 and WL_7). The address translator inputs MAJ and $A_3A_2A_1A_0$ and generates $D_7D_6D_5D_4D_3D_2D_1D_0$ and $EN_3EN_2EN_1EN_0$ to achieve this desired functionality for all the 16 cases. With the address translator logic (88 transistors), the triple-row decoder requires 200 transistors, while a regular 4:16 dynamic decoder (only single row activation) requires 136 transistors, a 47% increase in the row-decoder area. The address translator does not add any significant latency to the decoding process. The decoder was designed in 130 nm IHP process and its functionality was verified and decoding latency was found to be 496 ps.

C. Area of time-based Sense Amplifier

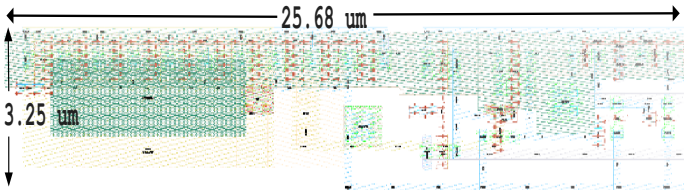

Fig. 7: Layout of time-based SA.

The main drawback of ReRAM based in-memory adders is their latency. The motivation for pioneering a parallel-friendly gate was to exploit it to accelerate addition by executing gates in parallel. To evaluate the number of gates that can be executed in parallel, we evaluated the area of the time-based SA. The time-based SA of [33] could sense the BL voltage without an op-amp, and, this was an important reason for adopting it for our majority gate (conventional SAs use operational amplifiers, which consume huge silicon area and power). The layout of the time-based SA of Fig. 3 is drawn in Fig. 7 and occupies an area of 25.68 × 3.25 = 83.5 μm^2. It must be noted that this area estimate does not include the area of the delay element since it is shared by all the SA in the array (t_{delay} in Fig. 3 is implemented as series of inverters with MOS capacitive load between them). From [41], the layout of a single 1T–1R cell occupies 450 nm × 450 nm = 0.2 μm^2 in 130 nm (12.4 F^2). If the SA is stacked along its height of 3.25 μm, eight columns of the array can share a SA. This means that the number of majority gates that can be executed

[5]a dynamic decoder uses a precharge signal ϕ, which when low, all WL are driven to '0'. When ϕ goes high, WL_i corresponding to D_1D_0 goes high, provided EN is '1'

in parallel in an array is the number of columns divided by a factor of 8 $i.e.$ 32 gates can be executed simultaneously in a 256×256 array, 8 gates in a 64×64 array $etc.$

D. WRITE circuit

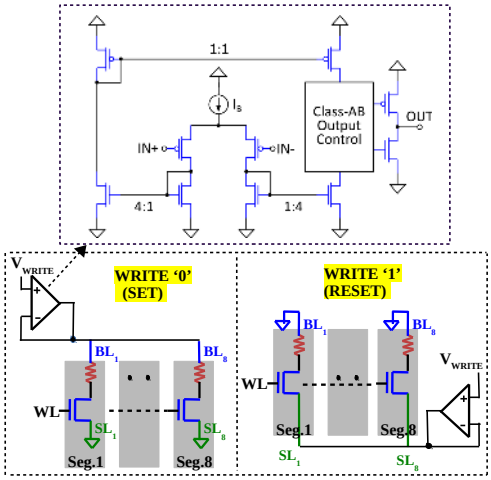

Fig. 8: Write circuit: Op-amp regulates the voltage while driving enough current to write into the cell.

To write to the cells, an operational amplifier is used as shown in Fig. 8. The amplifier consists of PMOS transistors as input devices, followed by a voltage gain stage using a symmetrical operational trans-conductance amplifier (OTA) configuration. The output stage is a fast class-AB output stage. It has a DC voltage gain of 61.1 dB and a phase margin of 66°. Each cell requires ≈ 300 μA to switch and the operational amplifier is designed to deliver the required current to program eight 1T–1R cells simultaneously (≈ 2.4 mA). In ReRAM, SET operation (the filament is created) is accomplished by applying a positive voltage to the BL while SL is grounded and RESET operation (the filament is ruptured) is accomplished by applying a positive voltage to the SL while BL is grounded. This is because a voltage of opposite polarity is needed across the ReRAM cell to break the filament. As shown in Fig. 8, the op-amp must be connected to BL for SET operation and SL for $RESET$ operation, which is accomplished by the SET/RESET switch (see Fig. 9). To verify the WRITE circuit, the operational amplifier was designed in IHP's 130 nm technology and the 1T–1R cell was modeled by fitting the Stanford-PKU model to characteristics of IHP's ReRAM [31]. A voltage pulse (V_{WRITE} = 1.2 V) of 100 ns duration was used and simultaneous writing of 8 1T–1R cells was verified by simulation.

E. In-memory computing system

The memory controller of a regular memory (be it DRAM-based or NVM-based) is responsible for orchestrating the READ and WRITE operation by issuing the control signals to the peripheral circuitry of the array. In addition, the memory controller must be augmented with additional capability to execute majority and NOT operation. Since both majority and NOT operations are READ operations in this logic family, the

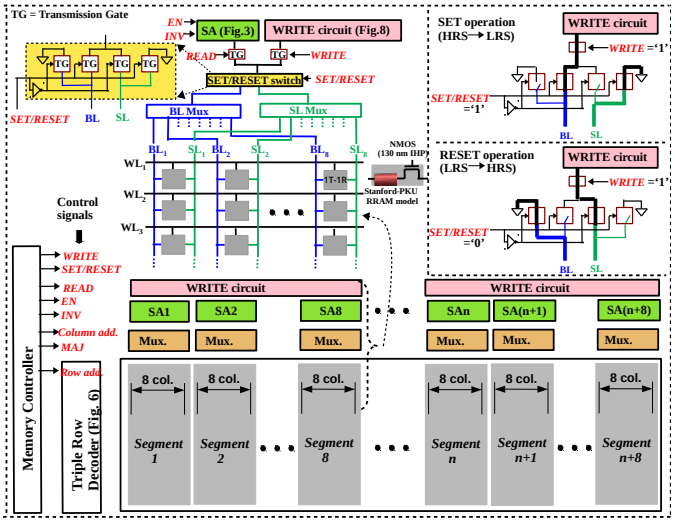

Fig. 9: The memory array augmented with computing capability. Since the area occupied by the SA corresponds to 8 columns, the array is partitioned into segments (8 columns form a segment) and each segment has a dedicated SA. 8 segments share a WRITE circuit.

controller does not require any major alterations. To execute a majority operation, an additional control signal called MAJ is needed, which is set to logic '1' during majority operation[6] and, the address of the first row (out of three rows in which majority is to be performed) is placed on the row decoder. It must be noted that majority operation is executed on three contiguous bits of data in a column and the triple row decoder presented in section IV-B will not only select the row corresponding to the address placed on the row decoder, but also the next two rows if MAJ is '1'. The column address is placed on the column decoder to select column(s) in which majority is to be executed and the SA is activated (using the EN signal) to get the output. The NOT operation is the same as the READ operation with the only exception being the controller issues the control signal INV, which goes high to invert the read data at the output of the SA (Fig. 5-(a)).

The complete in-memory computing system consists of the triple-row decoder, the read-out circuit (SA) and the WRITE circuit, collectively forming the peripheral circuitry around the 1T–1R array, as illustrated in Fig. 9. Since the area occupied by the SA corresponds to 8 columns, the array is partitioned into segments (8 columns form a segment) and each segment has a dedicated SA. As described in Section IV-D, the WRITE circuit can simultaneously write 8 cells and therefore 8 segments (64 columns) share a WRITE circuit. The BL and SL multiplexer will select one out of the 8 BL/SL of a segment and connect it to the SA (READ, NOT, Majority operation) or the WRITE circuit (SET, RESET operation). The polarity of the voltage applied across the cell is positive for SET and negative for RESET. The $SET/RESET$ signal is used to accomplish this change in voltage polarity using four transmission gates (Fig. 9). The control signals are depicted in red in Fig. 9. The control signals activated during memory

[6]this signal acts as an additional input to the row decoder, Fig. 6

and logic operations are summarized in Table II.

TABLE II: Control signals for memory and logic operations

Operation	$READ$	$WRITE$	$SET/RESET\ EN$	INV	MAJ
READ	1	0	1	1	0	0
NOT	1	0	1	1	1	0
Majority	1	0	1	1	0	1
SET $i.e.$ WRITE '0'	0	1	1	0	0	0
RESET $i.e.$ WRITE '1'	0	1	0	0	0	0

F. Energy for in-memory operations

To assess the energy required for computation, we first calculate the energy required for each logic operation. We calculate the energy for a single majority operation, as

$$E_{MAJ} = V_{DD} \int_0^{t_{READ}} I_{READ} \cdot dt + V_{DD} \int_0^{t_{READ}} I_{SA} \cdot dt \quad (3)$$

where I_{READ} is the current injected into the 1T–1R cell (see Fig. 3), I_{SA} is the current consumed by the time-based SA (including I_{BIAS}) and t_{READ} is the READ cycle duration. It must be noted that in Eq. 3, t_{READ} was 15 ns and I_{READ} was 30 μA in our simulations in IHP's 130 nm CMOS process. The energy for a single majority operation, E_{MAJ} = 0.63 pJ. The energy for the NOT operation is the same as the energy to read a single bit, and it was calculated to be E_{NOT} = 0.13 pJ/bit. E_{NOT} is smaller than E_{MAJ} because I_{READ} was smaller (2.5 μA) for NOT and READ, where a single bit is read. The energy to write a bit, $E_{WRITE} = V_{cell} \cdot \int_0^{t_{WRITE}} I_{WRITE}.dt$, where t_{WRITE} was 100 ns in our simulation (although switching time is $\leq$ 10 ns for these devices, t_{WRITE} was set to 100 ns to account for worst-case scenarios). E_{WRITE} was calculated to be 12 pJ/bit.

V. Eight-bit in-memory Parallel-prefix adder

A. Parallel-prefix adder using majority logic

Parallel-prefix (PP) adders are a family of adders originally proposed to overcome the latency incurred by the rippling of carry in CMOS-based adders. The regular structure of the memory array and the proposed parallel-friendly majority gate can be combined to implement PP adders in the memory array. PP adders have a 'carry-generate block' followed by a 'sum-generate block' (Fig. 10). The 'carry-generate block' can generate the carry 'ahead' by prefix computation, and is known to reduce the latency to $O(log(n))$, for n-bit adders [25]. Kogge-Stone, Ladner-Fischer, Brent-Kung and the like, are examples of PP adders. According to the taxonomy of PP adders [42], these adders essentially trade-off between logical depth, fan-out and wiring tracks. Since majority gate is the basic building block for many emerging nanotechnologies, prior works [22], [25] have formulated such PP adders in terms of majority gates. For this work, we chose Ladner-Fischer since it has optimised logical depth and minimum number of majority gates for n-bit adder [25]. It must be noted that for in-memory implementation, logic depth will translate to memory cycles and number of gates will influence the area of the array required for computing. It was important to choose Ladner-Fischer to minimize in-memory latency and area (elaborated in section V-B). The carry-generate and sum-generate blocks for an eight-bit adder in majority logic are derived from [22], [25] (Fig. 10). For an eight-bit adder, the logical depth is six levels of majority gates and one level of NOT gates, and at most eight gates are needed simultaneously in each level.

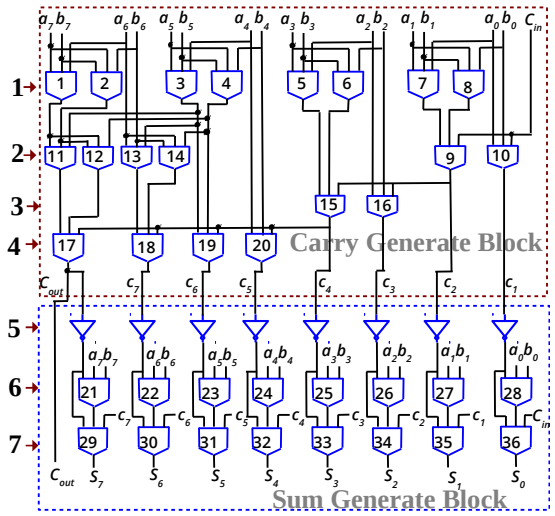

Fig. 10: Eight-bit PP adder (Ladner-Fischer) expressed as 7 levels of majority and NOT gates [22], [25]. Majority gates 1–20 constitute carry generate block and 21–36 constitute sum generate block.

B. Mapping of the eight-bit PP adder to 1T–1R array

In this section, we map the eight-bit Ladner-Fischer adder structure of Fig. 10 to a 1T–1R array using the proposed logic family, and elaborate the sequence of operations. Since the proposed gates are not stateful[7], the output of the majority gate (voltage) needs to be written to the array as inputs to the next logic level. Furthermore, the row-decoder places a restriction that only three consecutive rows can be selected. Therefore, outputs of a logic level must be written such that they are stored in consecutive rows for the subsequent majority operation. We assume a 6×80 processing area (to store the intermediate results of the computation), which is initialized to logic '0', $i.e.$, all cells are in LRS. Further, we assume that the two numbers to be added ($a_7a_6a_5a_4a_3a_2a_1a_0$, $b_7b_6b_5b_4b_3b_2b_1b_0$ and C_{in}) are arranged in the processing area as depicted in Fig. 11. To minimize latency, we map the adder in a way such that all the majority gates in a logic level are executed simultaneously in a READ operation. Furthermore, in view of the limited endurance of ReRAM devices[8], we map the gates in such a way that each bit in the 6×80 array is switched once during the entire duration of 8-bit addition

[7]In memristive logic, a logic family is said to be stateful if both the input and output of a computation are represented as resistance of the ReRAM/memristor [18]

[8]Endurance denotes the number of times the device can be switched between two stable states, while maintaining enough resistance ratio between them. Experimentally reported endurance vary from 10^6–10^{12} depending on whether it is HfO_x, SiO_x, TaO_x

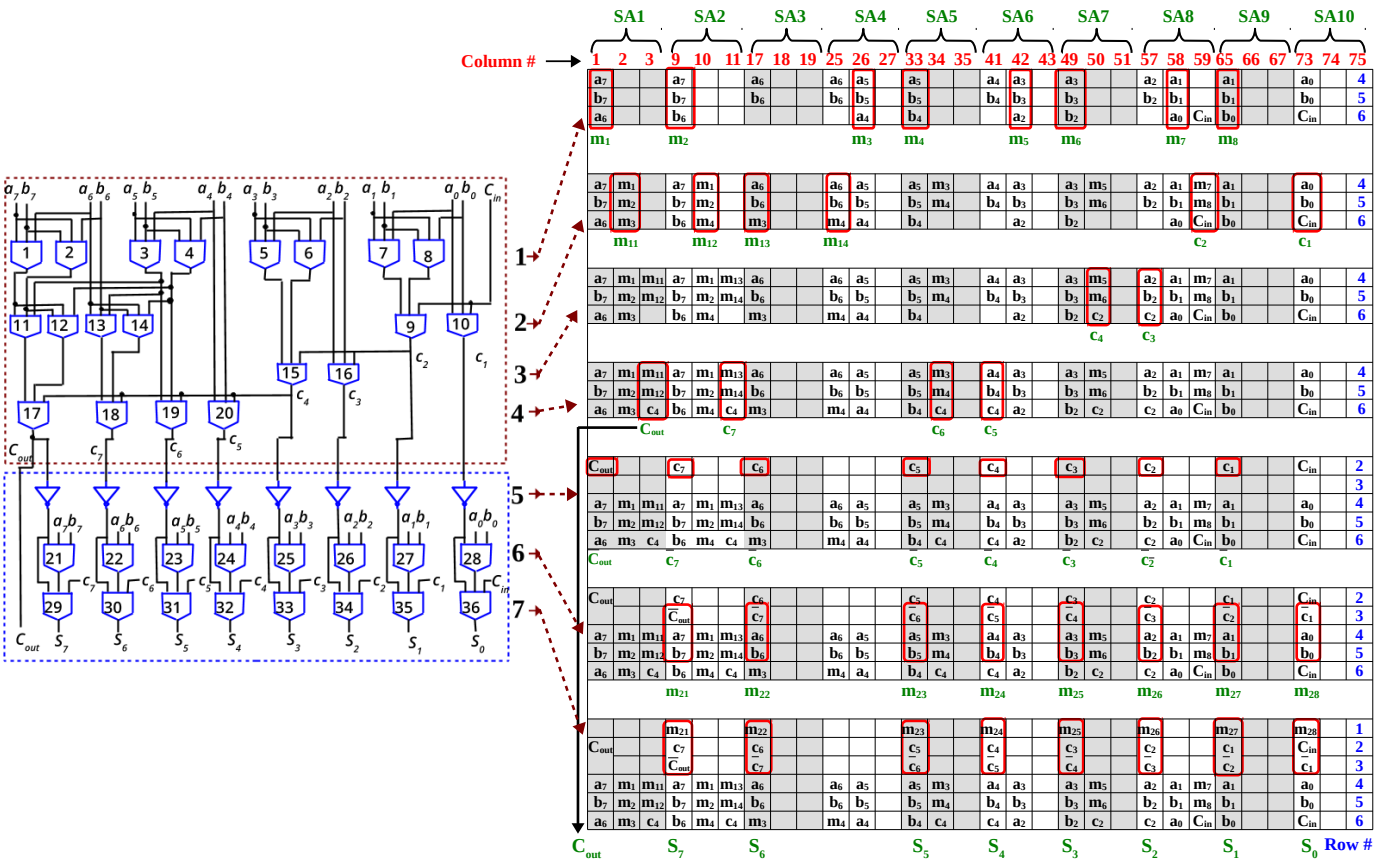

Fig. 11: Mapping of the eight-bit LF adder of Fig. 10 to 1T–1R array. All the majority gates in a level are executed in parallel (red boxes). m_i represent the output of the i^{th} majority gate and c_i is the carry generated during parallel-prefix addition (denoted green since it is read as a 'voltage' and then written into the array).

(intermediate results of addition namely m_i and carry c_i are written in different locations and not overwritten on the same location, see Fig. 11). If the endurance of the device is 10^8, in principle, 10^8 additions can be reliably performed in the 6×80 processing area. Another constraint of ReRAMs is that SET and RESET cannot be performed on multiple columns simultaneously since voltage of opposite polarity is needed to break the filament (Fig. 8). Therefore, writing multiple bits to a row is usually done in two steps, $i.e$, to write '10101010', first '_0_0_0_0' is written by a SET operation and then '1_1_1_1_' is written by a RESET operation. In our mapping, multiple bits can be written to a row in a single cycle since the 6×80 processing area is initialized to '0' $i.e.$ to write 10101010, only '1_1_1_1_' is written by switching the four ReRAM cells at the locations corresponding to '1' using the op-amp as the driver (Fig. 8). The contents of the array during the execution of the seven logic levels are depicted in Fig. 11. The steps are

1) Majority at col. (1,9,26,33,42,49,58,65).
2) Write ($m_1m_1m_3m_5m_7$) at col.(2,10,34,50,59) of row 4.
3) Write ($m_2m_2m_4m_6m_8$) at col.(2,10,34,50,59) of row 5.
4) Write ($m_3m_4m_3m_4$) at col. (2,10,17,25) of row 6.
5) Majority at col. (2,10,17,25,59,73).
6) Write ($m_{11}m_{13}$) at col. (3,11) of row 4.
7) Write ($m_{12}m_{14}$) at col. (3,11) of row 5.
8) Write (c_2c_2) at col. (50,57) of row 6.
9) Majority at col. (50,57).
10) Write ($c_4c_4c_4c_4$) at col. (3,11,34,41) of row 6.
11) Majority at col. (3,11,34,41).
12) Write ($C_{out}c_7c_6c_5c_4c_3c_2c_1$) at col. (1,9,17,33,41,49,57,65) of row 2.
13) NOT at col. (1,9,17,33,41,49,57,65).
14) Write ($\overline{C_{out}}\overline{c_7}\overline{c_6}\overline{c_5}\overline{c_4}\overline{c_3}\overline{c_2}\overline{c_1}$) at col. (9,17,33,41,49,57,65,73) of row 3.
15) Majority at col. (9,17,33,41,49,57,65,73).
16) Write ($m_{21}m_{22}m_{23}m_{24}m_{25}m_{26}m_{27}m_{28}$) at col. (9,17,33,41,49,57,65,73) of row 1.
17) Majority at col. (9,17,33,41,49,57,65,73), row 1-3.
18) Write ($C_{out}S_7S_6 \cdot\cdot S_1S_0$) to the memory array.

In the above mapping, it must be noted that each bit-wise majority operation is a READ operation and it must be followed by WRITE to be used as an input to the next logic level. Although WRITE operations increase latency, they can be used to our advantage while mapping by writing the data to the precise location in the array where it is needed in the next logic level. For example, in the mapping of majority gates to the array, ($m_1m_1m_3$..) are written to particular columns in row 4 (step 2) and ($m_2m_2m_4$..) are written to the same columns in row 5 (step 3) so that they are aligned for the majority operation in step 5. As enumerated above, two eight-bit numbers can be added by a sequence of READ and WRITE operations, requiring a total of 18 steps (6 Majority, 1 NOT and 11 WRITE cycles). The proposed approach is one of the

fastest implementation of eight-bit adder in ReRAM array, with only one other work [7] reporting a lower latency (Table III). Therefore, reading out in itself is not a disadvantage[9] as long as the total latency/energy is conserved $i.e.$ even by repeatedly reading and writing, majority-based adder can achieve better latency than NAND/NOR-based adder. Finally, from the endurance viewpoint, the number of such 8-bit additions that can be performed can be increased by wear-leveling techniques which distribute the WRITE operations evenly across the array $e.g.$ if a 30×80 processing area is available, consecutive addition operations can be mapped to different parts of the 30×80, thereby allowing 5×10^8 additions assuming a device with 10^8 endurance.

C. Simulation of eight-bit PP adder in 1T–1R array

To verify in-memory addition, we choose the 1T–1R cells from SG13S process of IHP. The IV-curve of the three-terminal unit (Stanford-PKU ReRAM model in series with IHP's 130 nm NMOS model) was fitted to the characteristics of IHP's three-terminal 1T–1R cell (TE, Gate and Source). The detailed fitting procedure is presented in earlier works [31], [37]. In this manner, the access transistor's resistance (which will appear in series with the memory cell's resistance) is also taken into account during simulation to faithfully reproduce the response while reading and writing into the cells. The 1T–1R array was simulated together with the SAs and WRITE circuit in 130 nm CMOS technology. Following the mapping elaborated in the previous section, the correct functioning of an 8-bit adder was verified by performing a sequence of READ ($i.e$ Majority, NOT) and WRITE operations. Parameters of the ReRAM model used in simulation are presented in the Appendix.

TABLE III: Comparison of eight-bit adders in ReRAM array

Primitive	Array	Latency	Area	Comment/Ref
IMPLY	1S-1R	58 steps	72 cells	Each step is IMPLY operation [27]
IMPLY+OR	1S-1R	54 steps	88 cells	Each step is IMPLY/OR operation [43]
NOR	1S-1R	38 steps	19×22	Each step has one or more NOR operations [44]
Majority	1S-1R	48* steps	8×3	Each step is majority (Fig. 14 (a)) or READ [44]
OR/AND	1S-1R	37 steps	64 cells	Each step has one or more OR/AND operation [45]
ORNOR	1S-1R	31 steps	54 cells	Each step has one or more ORNOR/IMPLY [46]
Majority+NOT	1T-1R	18 steps	6×80	Each step is majority/NOT or WRITE (this work)
XOR**	1T-1R	16 steps	three 1×8	Each step is XOR/READ [7]

* Latency is calculated as 24 RM_3 (Resistive Majority) instructions in [44], where each RM_3 consists of a READ followed by majority of Fig. 14 (a)
** XOR gate of [7] is not parallel-friendly and consequently multiple gates cannot be executed in parallel in the array (to circumvent this, [7] has used multiple arrays). Furthermore, XOR is not functionally complete and has to be used in conjunction with other gates to implement other arithmetic circuits. In contrast, majority+NOT is functionally complete and can be implemented with minimal peripheral overhead in the proposed method.

[9]Writing the data to the exact location where it is needed in the next logic level is necessary even in stateful logic families where the output of the gate is not read-out in a sense amplifier. $e.g.$ COPY operations in [28]

The proposed method naturally enables parallel-prefix addition by 'reading' majority simultaneously from columns of data. Therefore, the number of steps for eight-bit addition in a ReRAM array is shortened to 18 steps, as summarized in Table III. For our eight-bit adder, the energy consumption, calculated from simulations, was 708 pJ (36 majority, 8 NOT and 57 WRITE operations). In the Table III, we have not compared the energy for computation since they are either not reported [7] or reported for another ReRAM technology [44]. Depending on the ReRAM technology in which the adder is implemented/simulated, the energy will differ (switching energy depends on HRS, LRS and switching times which varies from few ns to even 1 μs). Therefore, it would be unfair to compare the energy of computation across different ReRAM technologies. However, the latency can be a good measure of energy comparison since, for each logic primitive, we mention what is the operation performed in each step. It is reasonable to expect the proposed adder to require a large array area (6×80) since it executes multiple gates in parallel.

VI. N-bit In-memory Parallel-Prefix Adders

In this section, we extend the proposed method of executing majority gates in parallel to design n-bit PP adders in memory. Specifically, we consider a 16-bit PP adder in majority logic and determine the latency to execute in memory. From the analysis of 8-bit and 16-bit adders, we formulate the latency required to execute n-bit adder in memory. Our end goal is to prove that the proposed majority gate will enable the execution of n-bit PP adders in logarithmic time complexity in memory.

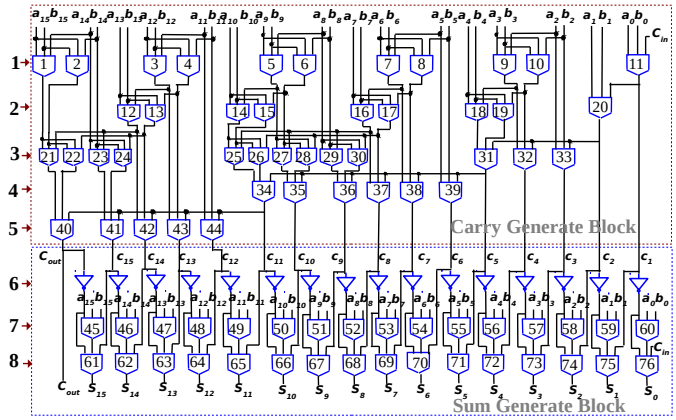

Fig. 12: 16-bit Ladner-Fischer adder expressed in terms of majority and NOT gates [22], [25].

Consider the 16-bit PP adder (Ladner-Fischer type) depicted in Fig. 12. As stated, prior works [22], [25] have formulated PP adders in majority logic, which we have adopted here for our in-memory implementation. Comparing Fig. 10 and Fig. 12, we can observe that from 8-bit to 16-bit, the number of logic levels increased from 7 to 8. This is the greatest advantage of PP computation and we will transfer this latency advantage to in-memory implementation as well. More specifically, the 'carry generate' block incurs one extra level of logic to compute the carry when we go from 8-bit to 16-bit (sum generate block remains 3 logic levels, compare Figs. 10 and

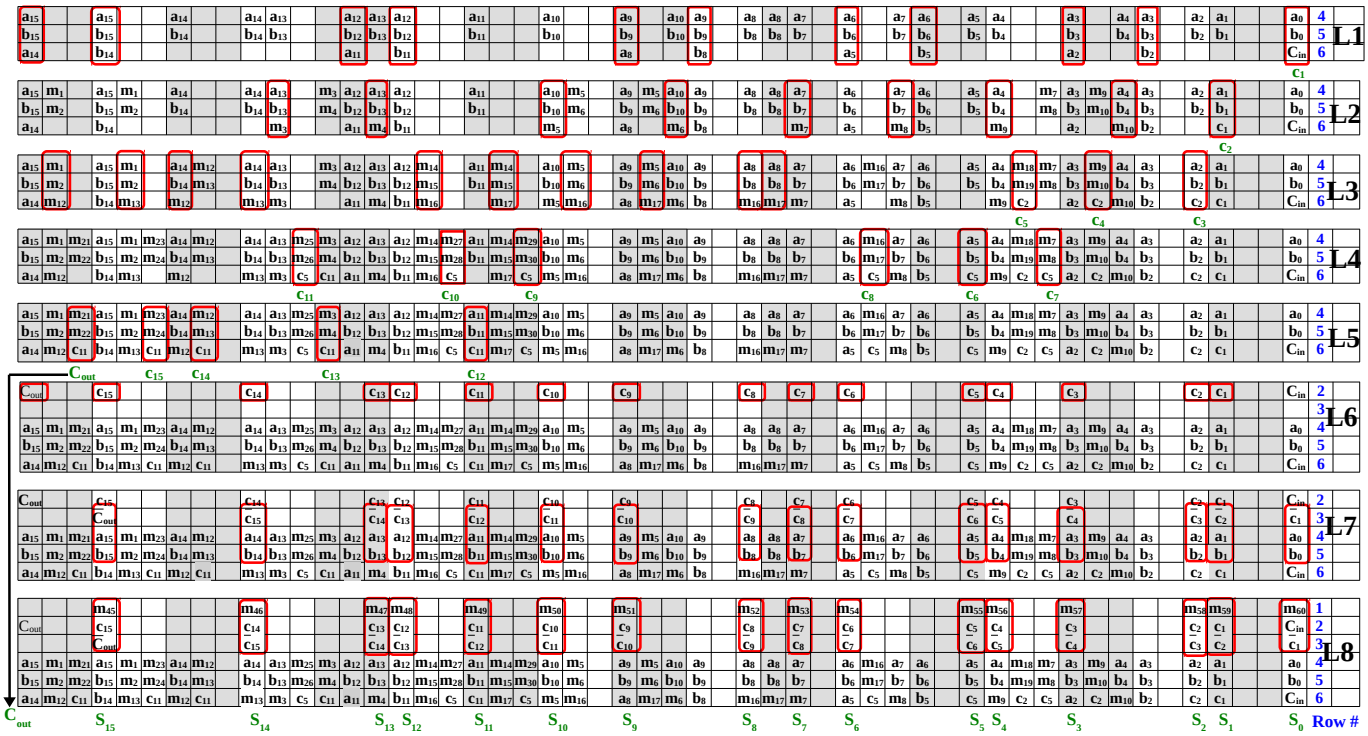

Fig. 13: Mapping of the 16-bit PP adder to the memory array using the proposed majority gates; L1–L8 denotes the mapping of the 8 logic levels of Fig. 12 and c_i denotes the carry generated during the prefix addition. $C_{out}S_{15}S_{14}\cdots S_0$ is the 17-bit sum after 22 steps.

12). In general, for n-bit addition, the number of logic levels, l is given by

$$l = log_2 n + 4 \tag{4}$$

for PP adder of the type of Ladner-Fischer expressed in majority logic [25]. Next, we map this 16-bit adder onto the 1T–1R array following the same procedure used for mapping the 8-bit adder (Section V-B). In the following steps, m_i denotes the output of the i^{th} majority gate and c_i the carry generated during prefix computation. The steps are (corresponding to Fig. 13):

1) Majority at row 4-6
2) Write ($m_1 m_1 m_3 m_5 m_5 m_7 m_9$) at row 4
3) Write ($m_2 m_2 m_4 m_6 m_6 m_8 m_{10}$) at row 5
4) Write ($m_3 m_4 m_5 m_6 m_7 m_8 m_9 m_{10} c_1$) at row 6
5) Majority at row 4-6
6) Write ($m_{12} m_{14} m_{14} m_{16} m_{18}$) at row 4
7) Write ($m_{13} m_{15} m_{15} m_{17} m_{19}$) at row 5
8) Write ($m_{12} m_{13} m_{12} m_{13} m_{16} m_{17} m_{16} m_{17} m_{16} m_{17} c_2 c_2 c_2$) at row 6
9) Majority at row 4-6
10) Write ($m_{21} m_{23} m_{25} m_{27} m_{29}$) at row 4
11) Write ($m_{22} m_{24} m_{26} m_{28} m_{30}$) at row 5
12) Write ($c_5 c_5 c_5 c_5 c_5 c_5$) at row 6
13) Majority at row 4-6
14) Write ($c_{11} c_{11} c_{11} c_{11} c_{11}$) at row 6
15) Majority at row 4-6
16) Write ($C_{out} c_{15} c_{14} \cdots c_2 c_1$) at row 2
17) NOT at row 2
18) Write ($\overline{C_{out}}\,\overline{c_{15} c_{14}} \cdots \overline{c_2 c_1}$) at row 3
19) Majority at row 3-5
20) Write ($m_{45} m_{46} \cdots m_{59} m_{60}$) at row 1
21) Majority at row 1-3
22) Write ($C_{out} S_{15} S_{14} \cdots S_1 S_0$) to the memory array.

For in-memory addition, the 7 logic levels of 8-bit adder gets translated to 18 steps. As enumerated above, the 8 logic levels of 16-bit adder requires 22 steps. The number of in-memory steps is always higher than the number of logic levels because the interconnections between logic levels become additional WRITE operations. A careful comparison of the in-memory steps for 8-bit and 16-bit adder reveals that each logic level gets translated to at least two steps, $i.e.$ $2l$ in-memory steps. In addition, the first few logic levels of the carry-generate block require two more WRITE steps. This phenomenon is true for (l-5) out of the l levels. Therefore, the number of steps required for l logic levels of a n-bit adder can be calculated as follows.

$$\begin{aligned} steps_{in-memory} &= (2l) + 2(l-5) \\ &= 4l - 10 \\ &= 4(log_2 n + 4) - 10 \\ &= 4 \cdot log_2 n + 6 \end{aligned} \tag{5}$$

The estimation of the array area required is straightforward–only 6 rows are required independent of adder size. The number of segments required is n+2. Since each segment has 8 columns, the size of the array required for n-bit addition is 6×(8n+16).

As stated, the energy for in-memory addition depends on the characteristics of the memristive device (which varies in each work). Therefore, we formulate the energy consumption of our in-memory adder in analytic expressions. This will aid

others to do qualitative comparison with our adder, in future. From Fig. 10 and Fig. 12, we can observe that the number of majority gates grows from 36 to 76 from 8-bit to 16-bit, while the number of NOT gates remains the same (n NOT gates for n-bit adder). In general, n-bit Ladner-Fischer PP adder will require $(n \cdot log_2 n + n + 2)$ majority gates [25]. For our in-memory implementation, each gate's input becomes a WRITE operation while each gate's output becomes a READ operation $i.e.$ each majority gate becomes three WRITE operations and one READ operation while each NOT gate becomes a WRITE and a READ operation. The total energy for an n-bit adder is therefore

$$Energy_{n-bit} = E_{WRITE} \cdot \big(3n \cdot log_2 n + 4n + 6\big) + E_{MAJ} \cdot \big(n \cdot log_2 n + n + 2\big) + E_{NOT} \cdot n \quad (6)$$

where E_{WRITE} is the energy required to write a single bit and E_{MAJ} and E_{NOT} are the READ energy for majority and NOT operations per bit. From the simulation results in section IV-F, $E_{WRITE} \approx 20 \times E_{MAJ}$ implying that the energy consumption of our adder will be dominated by the energy to WRITE to the array. Although our adder requires numerous WRITE operations, writing to a 1T–1R array can be achieved in an energy-efficient manner due to the absence of sneak currents. Sneak currents contribute to energy leakage and constitute a portion of the energy consumed in 1S–1R based adders [44], [46], while the percentage of such energy leakage is negligible in 1T–1R arrays. Overall, the proposed adder is energy-efficient, thanks to the 1T–1R configuration.

VII. Comparison with related works

A. Majority gate implementation in Resistive RAM (1S–1R)

In this section, we compare our work with two other in-memory majority gate implementation in literature. The first method to implement majority gate is proposed in prior works [47]–[49]. In this method, a majority gate is implemented in a 1S–1R array by applying the two inputs of the majority gate as voltages across ReRAM's terminals, and the initial state of the ReRAM (which is also the third input) switches to evaluate majority (see Fig. 14(a)). This manner of computation complicates the row/column decoders of the memory array, which were conventionally used to select rows/columns. Thus the peripheral circuitry will get complicated, $i.e.$, the row/column decoders have to be significantly modified to do row selection (during memory operation) and to apply the inputs (during majority operation). In contrast, in our implementation, the row/column decoders retain their functionality as in a conventional memory array, with a minor modification (triple-row decoding capability). Furthermore, our gate is conducive for parallel-processing since multiple gates can be mapped to the same set of rows, while multiple majority gates have to be mapped diagonally in [47]–[49] (Fig. 14-(c)). This parallel-friendly nature of our majority gates resulted in in-memory parallel-prefix adders with $O(log_2(n))$ latency. The second implementation is the in-memory minority gate (inverse of majority gate) proposed in [50]. The minority gate is realized by exploiting voltage division between three memristors (which store the inputs) and an output memristor. All the four memristors are located in a row/column of the array. Analysis of the functioning of such a gate with variations (in memristor's switching voltages and resistive states) is not discussed in [50].

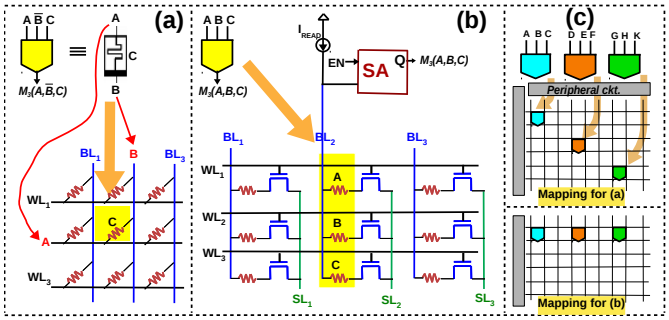

Fig. 14: (a) In-memory majority gate of previous works [47]–[49] (b) Proposed parallel-friendly gate (c) When multiple gates have to be executed in parallel, the majority gates of previous works [47]–[49] have to be mapped diagonally because two gates cannot be executed in the same row/column. This manner of computation complicates both the peripheral circuitry and memory controller (inputs of the gates influence row/column decoding).

B. Comparison with other in-memory adders

In this section, we compare our proposed in-memory addition technique with other adders/logic primitives and provide insights regarding latency, area and energy. Table IV summarizes the latency and array area required by different n-bit in-memory adders. From Table IV, it is evident that only PP configuration can shorten the latency drastically since for non-PP structures, latency grows as $O(n)$. Furthermore, even with PP structure, majority logic-based implementation performs better than OR/AND implementation [45]. As plotted in Fig. 15, the proposed adder is one of the fastest in-memory adders since its latency has a logarithmic dependence on n. This was possible because of the parallel-friendly nature of the proposed majority gate which enabled efficient in-memory implementation. Regarding the array area, the proposed adder requires a huge area which is to be expected since parallel-prefix adders minimize latency at the cost of hardware resources (more gates in parallel). Considering Figs. 11 and 13, one can observe that only 18% of the cells in the 6×80 (6×144 for 16-bit) area are actually used for computation. Due to sharing of SA, the remaining cells are blocked $i.e.$ this area cannot be used as a regular memory during computing. Even with this huge area requirement, n-bit adders can be realized in an array of reasonable size (32-bit adder requires 6×272).

Conventionally, in the field of VLSI, two adders are compared in terms of latency, energy and silicon area the circuit occupies. In CMOS-based adders, the circuit is dedicated to perform addition, but in in-memory computing, we are 're-purposing' the memory array to perform addition. Hence, the array area an in-memory adder requires is not a significant limiting factor since the array is already available (used as a regular memory) and a portion of it is used for computing. As stated, in all related works, computing capability is augmented to the memory array by some modifications to the peripheral

TABLE IV: Comparison of n-bit in-memory adders

Primitive	Adder Type	Latency	Array Area	Ref.
IMPLY	Ripple carry	$5n$+18	$9n$	[27]
IMPLY+OR	Ripple carry	$6n$+6	$11n$	[43]
NOR	Ripple carry	$10n$+3	$13n$-3	[51]
NOR	Look-Ahead	$5n$+8	13(n+1)	[52]
OR+AND**	Ripple carry	$6n$+1	$4n$	[53]
ORNOR	Parallel-clocking	$2n$+15	$6n$+6	[46]
RIMP/NIMP*	Pre-calculation	$2n$+4	$2n$+2	[54]
OR+AND	Parallel-prefix	$8log_2(n)$+13	(5+$log_2(n)$)n	[45]
Majority+NOT	Parallel-prefix	$4log_2(n)$+6	6(8n+16)	This work

*RIMP/NIMP stands reverse implication and inverse implication in a Complementary Resistive Switch (CRS) based adder. **Memristor overwrite logic
Note: XOR-based adder of Table III is not compared here since it is not extended to n-bit in [7]

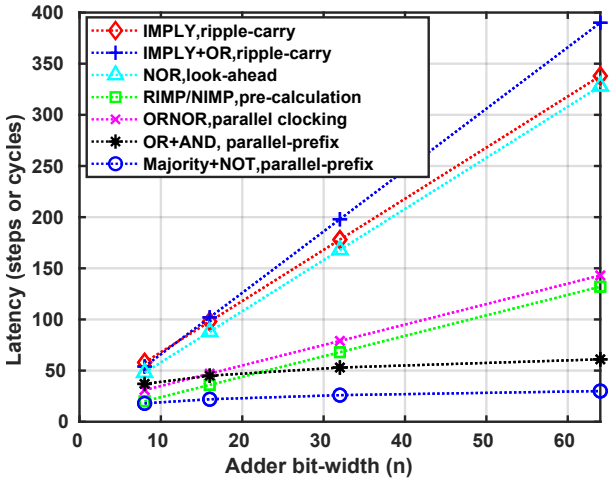

Fig. 15: In-memory adders with $O(n)$ latency require hundreds of cycles for 64-bit addition while the proposed in-memory adder will require only 30 cycles.

circuitry of the array. The increase in the peripheral circuit area required to perform logic operations is a significant factor to be considered since this increase in silicon area is solely to make the array 'computable'. Therefore, a holistic comparison between in-memory adders should consider both the increase in the peripheral circuitry area and the array area (occupied during addition), with the former being the significant factor. In this work, the triple-row decoder is the only change (Section IV-B) required while all the other parts of the peripheral circuitry do not change since computing is performed using normal memory operations (READ and WRITE).

C. Strengths and weakness of the proposed in-memory adder

The proposed in-memory adder achieves $O(log(n))$ latency, which is it's major strength. Due to the sharing of the SAs, the area of the array required during addition grows as $\approx 48n$, which is it's major weakness. Regarding the peripheral circuitry area, our in-memory adder necessitates a 47% increase in row decoder area to accommodate triple-row decoding. The energy consumption of the proposed adder is dominated by the energy to write to the cell, which can be achieved energy-efficiently in a 1T–1R configuration due to the absence of sneak paths. Although the focus of this work was addition, this work has great potential for accelerating in-memory multiplication. For example, a shift-and-add multiplier multiplies two n-bit numbers by shifting and adding the partial products. Since n-bit addition can be performed in $O(log(n))$, n-bit multiplication can be performed in memory in $O(n.log(n))$.

VIII. Conclusion

In this work, we have proposed a new memristive logic family based on majority logic. The majority gate can be implemented in a 1T–1R array without necessitating any major change in the peripheral circuit (except the row decoder which needs to be modified to activate three rows simultaneously). Majority logic can be combined with parallel-prefix techniques to design fast adders, and the proposed gate can be used to implement them in memory arrays, with $O(log_2(n))$ latency. In addition to accelerating computation in the array, the proposed adder is energy-efficient since sneak currents and its associated energy leakage is negligible in 1T–1R array. While the proposed adder does not require major modifications to the peripheral circuitry of ReRAM, it requires a considerable area of the memory array since it performs parallel-processing to minimize latency of computation.

Appendix
Stanford-PKU ReRAM model parameters

E_a = 0.6	a_0 =2.5×10^{-10}	t_{ox} = 6×10^{-9}	T_0 = 298 K
R_{TH} = 1500	I_0 = 7×10^{-4}	g_0 =0.318×10^{-9}	V_0 = 0.35
v_0 = 0.4	γ_0 = 20	β = 0.4	δ_g^0 = 0.005
T_{CRIT} =450	T_{SMTH} =500	gap_{max} =1.8×10^{-9}	gap_{min} =0.85×10^{-9}

References

[1] J. Reuben and S. Pechmann, "A parallel-friendly majority gate to accelerate in-memory computation," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 93–100.
[2] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.
[3] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, April 2017.
[4] H. Wong and S. Salahuddin, "Memory leads the way to better computing," *Nature Nanotechnology*, vol. 10, pp. 191–194, 2015.
[5] T. Xiao *et al.*, "Energy and performance benchmarking of a domain wall-magnetic tunnel junction multibit adder," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, no. 2, pp. 188–196, 2019.
[6] B. Chen, F. Cai, J. Zhou, W. Ma, P. Sheridan, and W. D. Lu, "Efficient in-memory computing architecture based on crossbar arrays," in *2015 IEEE International Electron Devices Meeting (IEDM)*, Dec 2015, pp. 17.5.1–17.5.4.
[7] Z. Wang *et al.*, "Efficient implementation of boolean and full-adder functions with 1t1r rrams for beyond von neumann in-memory computing," *IEEE Transactions on Electron Devices*, vol. 65, no. 10, pp. 4659–4666, Oct 2018.
[8] S. Hu *et al.*, "Reconfigurable boolean logic in memristive crossbar: The principle and implementation," *IEEE Electron Device Letters*, vol. 40, no. 2, pp. 200–203, Feb 2019.
[9] B. C. Jang *et al.*, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Advanced Functional Materials*, vol. 28, no. 2, p. 1704725, 2018.

[10] W. Chen *et al.*, "A 16mb dual-mode reram macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme," in *2017 IEEE International Electron Devices Meeting (IEDM)*, Dec 2017, pp. 28.2.1–28.2.4.

[11] C. Xue *et al.*, "A 22nm 2mb reram compute-in-memory macro with 121-28tops/w for multibit mac computing for tiny ai edge devices," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 244–246.

[12] P. Jain *et al.*, "A 3.6mb 10.1mb/mm2 embedded non-volatile reram macro in 22nm finfet technology with adaptive forming/set/reset schemes yielding down to 0.5v with sensing time of 5ns at 0.7v," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 212–214.

[13] W.-H. Chen, C. Dou, and K.-X. L. et al, "Cmos-integrated memristive non-volatile computing-in-memory for ai edge processors," *Nature Electronics*, vol. 2, no. 9, pp. 420–428, 2019.

[14] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Compute dram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019, p. 100–113.

[15] M. F. Ali, A. Jaiswal, and K. Roy, "In-memory low-cost bit-serial addition using commodity dram technology," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 155–165, 2020.

[16] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy, "X-sram: Enabling in-memory boolean computations in cmos static random access memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4219–4232, 2018.

[17] J. Wang *et al.*, "A compute sram with bit-serial integer/floating-point operations for programmable in-memory vector acceleration," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 224–226.

[18] J. Reuben *et al.*, "Memristive logic: A framework for evaluation and comparison," in *Power And Timing Modeling, Optimization and Simulation (PATMOS)*, September 2017, pp. 1–8.

[19] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature Electronics*, vol. 1, pp. 333 – 343, 2018.

[20] N. Talati, R. Ben-Hur, N. Wald, A. Haj-Ali, J. Reuben, and S. Kvatinsky, *mMPU—A Real Processing-in-Memory Architecture to Combat the von Neumann Bottleneck*. Singapore: Springer, 2020, pp. 191–213.

[21] L. Amarú, P. Gaillardon, and G. De Micheli, "Majority-based synthesis for nanotechnologies," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 499–502.

[22] G. Jaberipur, B. Parhami, and D. Abedi, "Adapting computer arithmetic structures to sustainable supercomputing in low-power, majority-logic nanotechnologies," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, pp. 262–273, Oct 2018.

[23] E. Testa, M. Soeken, L. G. Amaru, and G. De Micheli, "Logic synthesis for established and emerging computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 165–184, 2019.

[24] L. Amarú, P. E. Gaillardon, and G. D. 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, May 2016.

[25] V. Pudi, K. Sridharan, and F. Lombardi, "Majority logic formulations for parallel adder designs at reduced delay and circuit complexity," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1824–1830, Oct 2017.

[26] J. Reuben, "Rediscovering majority logic in the post-cmos era: A perspective from in-memory computing," *Journal of Low Power Electronics and Applications*, vol. 10, no. 3, 2020.

[27] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[28] R. Ben-Hur *et al.*, "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*, pp. 1–1, 2019.

[29] P. Huang *et al.*, "Reconfigurable nonvolatile logic operations in resistance switching crossbar array for large-scale circuits," *Advanced Materials*, vol. 28, no. 44, pp. 9758–9764, 2016.

[30] J. Reuben, "Binary addition in resistance switching memory array by sensing majority," *Micromachines*, vol. 11, no. 5, 2020.

[31] J. Reuben, D. Fey, and C. Wenger, "A modeling methodology for resistive ram based on stanford-pku model with extended multilevel capability," *IEEE Transactions on Nanotechnology*, vol. 18, pp. 647–656, 2019.

[32] A. Chen, "Memory selector devices and crossbar array design: a modeling-based assessment," *Journal of Computational Electronics*, Sep 2017.

[33] Q. Trinh, S. Ruocco, and M. Alioto, "Time-based sensing for reference-less and robust read in stt-mram memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 10, pp. 3338–3348, Oct 2018.

[34] N. Raghavan, "Statistics of disturb events in oxram devices — a phenomenological model," in *2017 IEEE International Reliability Physics Symposium (IRPS)*, 2017, pp. PM–3.1–PM–3.7.

[35] M. Chang *et al.*, "A high-speed 7.2-ns read-write random access 4-mb embedded resistive ram (reram) macro using process-variation-tolerant current-mode read schemes," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 3, pp. 878–891, 2013.

[36] W. Shim, Y. Luo, J. Seo, and S. Yu, "Investigation of read disturb and bipolar read scheme on multilevel rram-based deep learning inference engine," *IEEE Transactions on Electron Devices*, vol. 67, no. 6, pp. 2318–2323, 2020.

[37] J. Reuben, M. Biglari, and D. Fey, "Incorporating variability of resistive ram in circuit simulations using the stanford–pku model," *IEEE Transactions on Nanotechnology*, vol. 19, pp. 508–518, 2020.

[38] A. Prakash and H. Hwang, "Multilevel cell storage and resistance variability in resistive random access memory," *Physical Sciences Reviews*, vol. 1, no. 6, pp. –, 2016.

[39] G. C. Adam, A. Khiat, and T. Prodromakis, "Challenges hindering memristive neuromorphic hardware from going mainstream," *Nature Communications*, vol. 9, 2018.

[40] V. Parmar and M. Suri, *Exploiting Variability in Resistive Memory Devices for Cognitive Systems*. New Delhi: Springer India, 2017, pp. 175–195, doi: 10.1007/978-81-322-3703-7_9.

[41] A. Levisse, B. Giraud, J. . Noel, M. Moreau, and J. . Portal, "Rram crossbar arrays for storage class memory applications: Throughput and density considerations," in *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, Nov 2018, pp. 1–6.

[42] D. Harris, "A taxonomy of parallel prefix networks," in *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers*, Nov 2003, pp. 2213–2217.

[43] L. Cheng *et al.*, "Functional demonstration of a memristive arithmetic logic unit (memalu) for in-memory computing," *Advanced Functional Materials*, vol. 29, no. 49, p. 1905660, 2019.

[44] J. Reuben *et al.*, *A Taxonomy and Evaluation Framework for Memristive Logic*. Cham: Springer International Publishing, 2019, pp. 1065–1099.

[45] A. Siemon, S. Menzel, D. Bhattacharjee, R. Waser, A. Chattopadhyay, and E. Linn, "Sklansky tree adder realization in 1s1r resistive switching memory architecture," *The European Physical Journal Special Topics*, vol. 228, no. 10, pp. 2269–2285, 2019.

[46] A. Siemon *et al.*, "Stateful three-input logic with memristive switches," *Scientific Reports*, vol. 9, no. 1, p. 14618, 2019.

[47] P. Gaillardon *et al.*, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 427–432.

[48] S. Shirinzadeh, M. Soeken, P. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1422–1435, July 2018, doi: 10.1109/TCAD.2017.2750064.

[49] D. Bhattacharjee, A. Easwaran, and A. Chattopadhyay, "Area-constrained technology mapping for in-memory computing using reram devices," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 69–74.

[50] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 55:1–55:7.

[51] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[52] Y. S. Kim *et al.*, "Stateful in-memory logic system and its practical implementation in a taox-based bipolar-type memristive crossbar array," *Advanced Intelligent Systems*, vol. 2, no. 3, p. 1900156, 2020.

[53] K. Alhaj Ali *et al.*, "Memristive computational memory using memristor overwrite logic (mol)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2370–2382, 2020.

[54] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 5, no. 1, pp. 64–74, 2015.