A Novel In-Memory Wallace Tree Multiplier Architecture using Majority Logic

Vijaya Lakshmi, John Reuben, *Member, IEEE*, Vikramkumar Pudi

***Abstract*—In-memory computing using emerging technologies such as resistive random-access memory (ReRAM) addresses the 'von Neumann bottleneck' and strengthens the present research impetus to overcome the memory wall. While many methods have been recently proposed to implement Boolean logic in memory, the latency of arithmetic circuits (adders and consequently multipliers) implemented as a sequence of such Boolean operations increases greatly with bit-width. Existing in-memory multipliers require $O(n^2)$ cycles which is inefficient both in terms of latency and energy. In this work, we tackle this exorbitant latency by adopting Wallace Tree multiplier architecture and optimizing the addition operation in each phase of the Wallace Tree. Majority logic primitive was used for addition since it is better than NAND/NOR/IMPLY primitives. Furthermore, high degree of gate-level parallelism is employed at the array level by executing multiple majority gates in the columns of the array. In this manner, an in-memory multiplier of $O(n.log(n))$ latency is achieved which outperforms all reported in-memory multipliers. Furthermore, the proposed multiplier can be implemented in a regular transistor-accessed memory array without any major modifications to its peripheral circuitry and is also energy-efficient.**

***Index Terms*—Memristor, resistive random-access memory (ReRAM), majority logic, 1Transistor-1Resistor (1T–1R), von Neumann bottleneck, in-memory computing, sense amplifier, processing-in-memory, parallel-prefix adder, Wallace Tree multiplier, read-out circuit.**

I. Introduction

The conventional von Neumann systems use separate units for storing data (memory) and processing data. The data movement between the memory and processing units is the major cause for the degraded performance of present day computing systems, often referred to as the "von Neumann bottleneck" or "memory wall." Memory wall will eventually result in enhanced processor speed being masked by the relatively slow improvements to DRAM speed [1]. "Computation energy" is dominated by "data movement energy" since the energy required for memory access grows exponentially along the memory hierarchy, *i.e.* from cache to off-chip DRAM [2]. A quantitative example is provided in [3] where it is pointed out that the DRAM access energy is 3556× the energy for 16-bit addition in a 45-nm CMOS technology. In the last two decades, several efforts were made to solve the memory wall problem by bringing the processor and memory unit closer together. Earlier efforts include 3-D stacking of DRAM dies over logic die (near-memory computing [4]) to reduce the latency and energy of data movement between processor and memory. More recently, significant efforts are being made to enhance the memory with computational capabilities *i.e.* the memory array which was conventionally used only to store data is augmented with capabilities to also execute Boolean logic operations. Research in this direction is being pursued in conventional memories like DRAM [5] and SRAM [6] and also emerging non-volatile memories like ReRAM [7] , PCM [8], STT-MRAM [9] and Ferroelectric FET [10]. Classified as "in-memory computing" or "processing-in-memory", all these works signify an important need to move computing to the location of the data.

Resistive RAM (ReRAM) is one of memory technologies from the memristor family which offers high density, non-volatility, good scalability, low WRITE energy and good resistance window [11]. The resistance of ReRAM cell can be switched reversibly between a low-resistance state (LRS) and a high-resistance state (HRS) by applying a suitable voltage across the cell. This change of resistance is due to the formation or rupture of a conductive filament between the two electrodes of the cell. ReRAM cells can be configured in a memory array using an access transistor for each cell, commonly referred to as a 1Transistor–1Resistor (1T–1R) array. The access transistor enables each cell of the array to be selected individually (thus avoiding sneak path currents) and also serves to limit the current through the ReRAM cell.

Boolean logic primitives like IMPLY (Material Implication), NOR, NAND, XOR, MAJORITY have been implemented in ReRAM array. Arithmetic circuits like adders and multipliers can be implemented as a sequence of Boolean operations. A detailed review of in-memory arithmetic using ReRAM technology is beyond the scope of this work. The reader is referred to the following surveys– classification based on state variable used for computation [12], logic primitives used for computation [13], cascading methodology [14] and latency for 1-bit addition [15]. Adders and multipliers are the building blocks of any computing unit and the efficiency of these circuits influence the efficiency of the entire computing unit. Multipliers are computationally intensive than adders and their computationally complexity grows quadratically with

Vijaya Lakshmi and Vikramkumar Pudi are with the Department of Electrical Engineering, Indian Institute of Technology Tirupati, Tirupati 517619, India. (e-mail:ee20d001@iittp.ac.in & vikram@iittp.ac.in)

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

Manuscript received August 2, 2021; revised October 28, 2021; accepted November 12, 2021.

Digital Object Identifier 10.1109/TCASI.2021.3129827

bit-width *i.e.* O(n^2). Consequently, existing in-memory multipliers require $\approx$ 1000 cycles for 32-bit multiplication. In this work, we tackle this exorbitant latency of in-memory multiplication.

The existing works on in-memory multiplication use Shift and Add Multiplier [16], Array Multiplier [17], Dadda Multiplier [18], Multiplier Using Semi-Serial IMPLY-based adder [19] and Carry-save-add-shift (CSAS) multiplier [20] architectures. In this work, Wallace Tree architecture is selected for our in-memory multiplier due to its reduced logical depth (less latency) [21], [22]. The phases of Wallace Tree architecture are partial product generation, stage by stage partial product reduction phase followed by a final phase of addition. Except the partial product generation phase (which is accomplished by AND operation), all phases require some form of addition. Conventionally, adders in CMOS technology were designed using AND, OR and XOR gates. But recent research in logic synthesis [23]–[26] indicates MAJORITY to be an efficient logic primitive for certain arithmetic-intensive circuits. However, to be able to perform multiplication in memory using majority gates, a viable method to execute a majority gate in the memory array is needed. Recently, such a method to implement a majority gate in a ReRAM array has been proposed where the majority gate is executed as a READ operation [7], [27].

A peculiar requirement for effective in-memory implementation is that the circuit must be as homogeneous as possible in terms of the logic primitives used *i.e.* a full adder expressed solely in terms of NOR gates is easier to implement in memory than the one expressed in terms of AND, OR and XOR gates. This is because, diverse logic primitives in a circuit require different voltages at the terminals of the memory array, thereby limiting parallelism and complicating the peripheral circuitry. Given this peculiar requirement, a full adder can be implemented with less logical depth (latency) using majority gates than using NAND/NOR/IMPLY gates (as elaborated later in Section III-A). For increasing bit-width (n-bit adder), parallel-prefix adders synthesized using majority gates were used to minimize latency. To summarize, wallace-tree multiplier architecture was chosen since it has optimized latency at architecture level [28], [29]. Furthermore, each addition stage of wallace-tree multiplier is synthesised in terms of majority gates and executed in memory. A significant reduction in latency compared to existing in-memory multipliers was achieved by combining the strengths of MAJORITY logic primitive and the Wallace Tree architecture.

The rest of this paper is organized as follows. In Section II, we briefly explain the Wallace Tree multiplication process and the reason for choosing it for in-memory multiplication. In Section III, we explain the motivation to choose majority gate as the logic primitive for in-memory multiplication and the efficient implementation of majority gate in-memory. The structure of the proposed in-memory computing system is briefly explained in Section IV. Section V presents the proposed 4×4 in-memory Wallace Tree multiplier and elaborates how it is implemented in a 1T-1R array followed by simulation results. In Section VI, we present the design of 8×8 in-memory Wallace Tree multiplier and observe how our multiplier performs with increasing bit width. In this manner, the latency and area for $n \times n$ multiplication are formulated. We compare the proposed in-memory multiplication technique with the existing works in Section VII, followed by conclusions in Section VIII.

II. Wallace Tree Multiplier

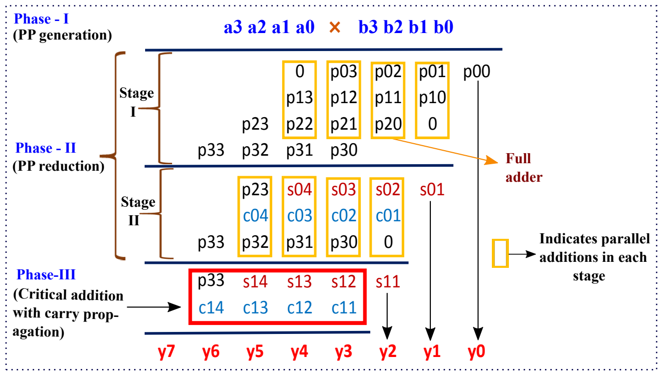

Fig. 1: 4×4 Wallace Tree multiplier. Partial products are computed as P_{ij}=$a_j.b_i$

Wallace Tree Multiplier is the most extensively used multiplier architecture in memory units and processors [22]. The Wallace Tree architecture for multiplication of two n-bit numbers encompasses three phases:

1) partial product generation by multiplying every bit of multiplier with multiplicand using n^2 AND gates
2) reducing the sum of partial products using full-adders
3) generation of multiplication result by the final 2-input addition (critical phase for latency)

The partial products generated in phase 1 are grouped in sets of 3 rows each and added using full adders. Additional partial products which are not a member of the group are transferred to the next stage and get added along with the sum and carry bits generated from the previous stage. The additions in these intermediate phases (partial product reduction) are performed in parallel with a maximum latency of a 1-bit full adder. This process continues until only two rows remain for addition in the final stage. The number of partial product reduction stages for $n \times n$ Wallace Tree multiplier follows $log_2\frac{n^2}{4}$ (*i.e.* from 4-bit to 32-bit multiplier, number of partial product reduction stages increases from 2 to 8). The final stage addition is critical in terms of latency as carry must be propagated quickly and therefore requires an adder with optimum latency. The latency of addition in the final stage determines the speed of the multiplier. A $n \times n$ Wallace Tree multiplier requires an adder of size $2(n{-}log_2 n)$ bit in its final phase. To optimize latency, a parallel-prefix adder (Ladner-Fischer) can be used in the final stage of Wallace Tree multiplier [28].

A $4{\times}4$ Wallace Tree multiplication process is shown in Fig. 1. To multiply two 4-bit numbers $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$, first all partial products (p_{00}-p_{33}) are generated using AND gates. Next the partial products are grouped and added to produce sums s_{01}-s_{04} and carries c_{01}-c_{04} as shown in Fig. 1. In stage II, the non grouped partial products (p_{23},$p_{30} - p_{32}$) of stage

I are added with the sums s_{01}-s_{04} and carries c_{01}-c_{04}. The parallel additions in each stage are shown with a rectangular yellow box and there exist four parallel additions in stages I and II. The results of stage II, sums s_{11}-s_{14} and carries c_{11}-c_{14} are grouped with left out partial product p_{33} and can be added using a conventional 4-bit adder in the final stage III. Using a 4-bit parallel-prefix (Ladner Fischer) adder in the final stage accelerates the addition process and gives 8-bit result with optimum latency. Similarly, different stages involved in the 8×8 Wallace Tree multiplication process are shown in Fig. 10. From Fig. 10, it is clear that the parallel additions in stage I, II, III and IV are 16, 16, 10 and 11 respectively. A 10-bit adder is required in the final stage which can be implemented using a parallel-prefix (Ladner Fischer) adder to minimize latency. The number of parallel additions in each stage increases as the bit-width of the Wallace Tree multiplier grows and they can be implemented in the columns of the memory array, as will be elaborated later. Finally, though Dadda multiplier has marginally less latency than the Wallace Tree multiplier [30], Wallace Tree multiplier architecture was preferred in this work. This is because partial products in Dadda multiplier are arranged such that results produced in one stage should replace the contents of the memory array for the next stage of addition. This overwriting of cells not only reduces the reliability of ReRAM (endurance) but also increases latency.

III. Majority Logic as a Logic Primitive for In-Memory Multiplication

A. Boolean Logic Primitives Implemented in Memory

In the past, different logic primitives like NAND, NOR, IMPLY, XOR have been implemented in memory [12]–[14]. Adders/multipliers can be implemented as a chain of Boolean logic operations. As stated in Section I, homogeneous logic primitive is preferred for in-memory implementation. It must be noted that, in literature, there are works which implement diverse logic primitives like AND, OR, XOR in the same memory array [31], [32]. But gate-level parallelism cannot be exploited using such logic primitives because different logic primitives require different voltages/currents to be applied at word line (WL) and bit line (BL). For an arithmetic intensive circuit like multiplier, gate-level parallelism is of paramount importance to minimize latency. Therefore majority logic primitive was preferred since a circuit synthesized in terms of a single logic primitive can enable gate-level parallelism.

As elaborated in Section II, Wallace Tree multiplier is implemented in three phases namely, partial product generation, full adders phase and a parallel-prefix adder in the last phase. Partial product, P_{ij}=$a_j.b_i$ = M($a_j, b_i, 0$) (M denotes majority) and can be implemented in one memory cycle. A 1-bit full adder implemented in memory using IMPLY [33], NOR [34] and MAJORITY are compared in Fig. 2. Using majority logic, full adder can be implemented in memory in 6 cycles (assuming majority and its complement are available at the sense amplifier output) which is significantly less compared to the adders using IMPLY/NOR. Furthermore, since majority gate is the fundamental logic primitive for many emerging nanotechnologies, recent research [23], [26] have

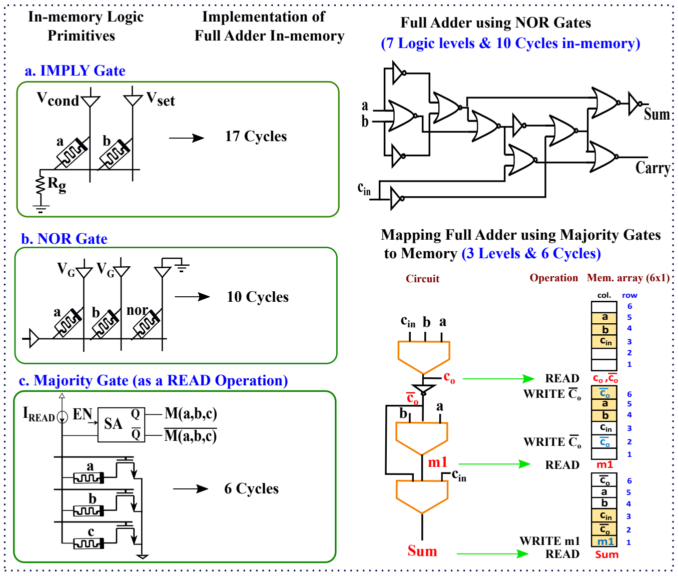
In-memory Logic Primitives
Implementation of Full Adder In-memory
Full Adder using NOR Gates
(7 Logic levels & 10 Cycles in-memory)
a. IMPLY Gate
Vcond
Vset
a
b
Rg
17 Cycles
Sum
Carry
cin
b. NOR Gate
VG
VG
a
b
nor
10 Cycles
Mapping Full Adder using Majority Gates to Memory (3 Levels & 6 Cycles)
Circuit
Operation
Mem. array (6x1)
c. Majority Gate (as a READ Operation)
IREAD
EN
SA
Q
M(a,b,c)
6 Cycles
cin b a
co
READ
WRITE C̄o
WRITE C̄o
READ
m1
WRITE m1
READ
Sum

Fig. 2: In-memory logic primitives and latency of a full adder using each logic primitive (IMPLY [33], NOR [34] and MAJORITY). A full adder implemented using majority gates incurs less latency and gates.

synthesized parallel-prefix adders solely using majority gates[1]. Research in majority logic synthesis revealed that arithmetic circuits synthesised using majority logic could achieve 14–33% reduction in logical depth when compared to conventional AND/NOT logic primitives [24], [35]. An 8-bit parallel-prefix adder synthesized in majority logic has 7 logic levels while the same adder synthesized using AND, OR, XOR gates has 8 logic levels [15]. In essence, majority logic primitive outperforms other logic primitives in terms of latency for both 1-bit addition and n-bit addition. As stated in Section II, the Wallace Tree multiplier is essentially composed of 1-bit and n-bit adders and all the three stages can be expressed in terms of majority gates and its inversion. Therefore, majority gate was chosen as the fundamental logic primitive in this work for in-memory multiplication.

B. In-Memory Implementation of a Majority Gate

In [7], [27], an efficient method to implement a majority gate in a transistor-accessed ReRAM array has been presented recently. In this method, majority is computed while reading from a 1T–1R array (Fig.3). It is type of non-stateful logic[2] *i.e.* inputs to the majority gate are resistances and the output is sensed as a voltage. 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 (Fig. 3). A sense Amplifier (SA) which can accurately sense the effective resistance $R_{eff} \approx R_a||R_b||R_c$ implements an 'in-memory' majority gate. R_a, R_b, R_c denotes the inputs of the majority gate with each of them being HRS (logic '1') or LRS (logic '0'). Consider the eight possibilities of (a,b,c),

[1]conventionally synthesized in terms of AND, OR and XOR gates
[2]In memristive logic, a logic family is said to be stateful if both the input and the output of a gate are represented as the resistance of the memristor

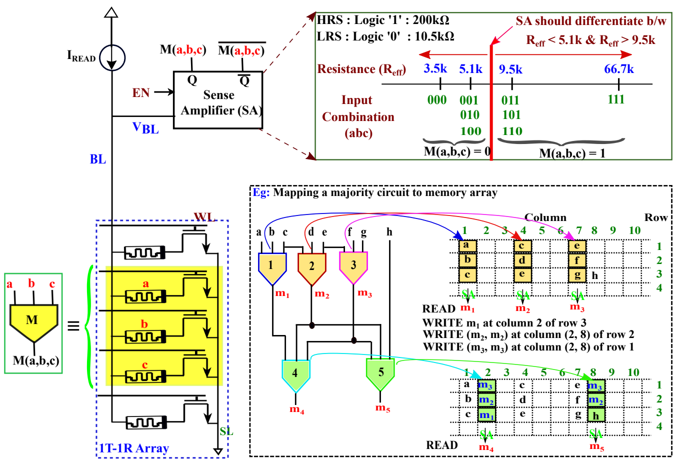

Fig. 3: Majority gate implemented in memory as a READ operation. During READ, if three rows are activated simultaneously, the ReRAM cells will be in parallel ($R_{eff} \approx R_a||R_b||R_c$) and a SA can be used to precisely sense R_{eff} [7].

the three binary inputs of the majority gate (Fig. 3, top). Let us consider a ReRAM technology with HRS of 200 kΩ and LRS of 10.5 kΩ [36]. Consequently, $R_{eff} \leq$ 5.1 kΩ if two or more ReRAM cells are in LRS and $R_{eff} \geq$ 9.5 kΩ if two or more ReRAM cells are in HRS. To output M(a,b,c), the SA should be able to differentiate between R^{001}_{eff} (two LRS and one HRS) and R^{011}_{eff} (two HRS and one LRS). A robust time-based SA is used to achieve this [7]. This gate enables parallelism (multiple majority gates can be executed in parallel in different columns). Hence it is well suited for our in-memory multiplier where multiple majority gates have to be executed in parallel. Multiple levels of logic can be cascaded by writing the data back to the memory, as illustrated in Fig. 3 (bottom). In fact, any arithmetic circuit can thus be executed as a sequence of READ (*i.e.* majority) and WRITE operations. Therefore, the proposed gate does not require any major modifications to the peripheral circuitry of a regular ReRAM memory. The gate is also energy-efficient (both reading and writing is energy-efficient in 1T–1R when compared to 1S–1R due to the absence of sneak paths).

IV. Structure of In-Memory Computing System

The in-memory computing system consists of a READ circuit (SA), WRITE circuit, circuits for row and column selection and the memory controller logic, as depicted in Fig. 4. Each column in the array has a dedicated sense amplifier. Since multiplication is performed using MAJORITY gates (Fig.7), which are executed as a series of READ and WRITE operations, we elaborate on the READ and WRITE circuits. We briefly describe the other parts of the peripheral circuitry and the reader is referred to [7] for more detailed description.

A. READ circuit (Sense Amplifier)

Fig. 5 depicts the design of the SA used to execute the majority gate during READ operation in memory. The circuit was designed in CMOS 45 nm node. A ReRAM device from IHP foundry has average resistance levels of HRS = 200 kΩ

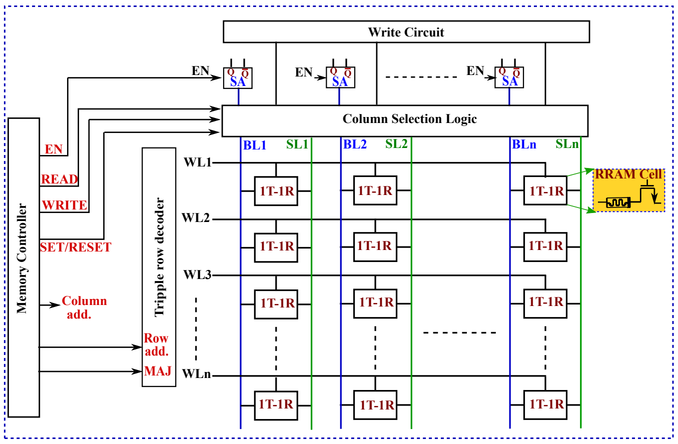

Fig. 4: Architecture of the in-memory computing system. Each column has a dedicated SA for READ operation (Fig. 5). Eight columns of the array share a WRITE circuit (Fig. 6). A triple-row decoder is used which uses MAJ as control signal to switch between majority operation (three rows selected) and normal READ/WRITE operation (single row is selected).

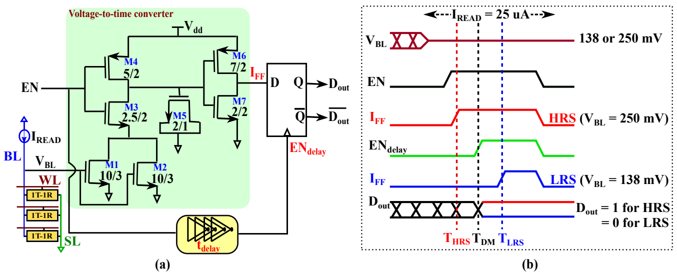

Fig. 5: (a) Schematic of the SA used for majority operation. The aspect ratio of transistors denoted $\frac{m}{n}$ implies a transistor of size $\frac{m \cdot W_{min}}{n \cdot L_{min}}$. (b) Wave forms of TBSA circuit showing the switching of output for HRS and LRS.

and LRS = 10.5 kΩ [36]. For such a device, the sensing window for majority gate = $R^{001}_{eff} - R^{011}_{eff}$ = 9.5 k - 5.1 kΩ = 4.4 kΩ (see Section III-B). To accurately sense this resistance window in the presence of ReRAM variations, a robust time-based sense amplifier (TBSA) proposed in [37] was chosen and adapted to our requirement in 45nm node. In time-based sensing, voltage to be sensed (Bit line voltage) is converted into a time delay and discriminated in time domain. The current-starved inverter (formed by transistors M_{1-4}) achieves voltage-to-time conversion by either allowing more current or less current depending on the voltage at gate of $M_{1,2}$. A small current I_{READ} = 25 μA injected into the ReRAM cell converts the cell's resistance into a voltage, which is fed to the voltage-to-time converter. Although the threshold voltage, $V_{th-M1,M2}$ is 420 mV, due to sub-threshold conduction of the transistor at 45 nm, transistors M_1 and M_2 start conducting even at 250 mV. To account for this, I_{READ} of 25 μA was used so that V_{BL} = 138 mV and 250 mV for R^{001}_{eff} and R^{011}_{eff} cases, respectively. For R^{011}_{eff} case, bit-line voltage $V_{BL} \geq 250mV$, which turns on $M_{1,2}$ sharply and introduces less delay in I_{FF} signal and I_{FF} rises at T_{HRS} (Fig.5-(b)). Whereas for R^{001}_{eff} case, bit-line voltage $V_{BL} \leq 250mV$ so that it limits the inverter current and more delay is introduced in I_{FF}

signal and I_{FF} rises at T_{LRS}, as shown in Fig. 5-(b). The delayed EN signal (denoted as EN_{delay}) is generated using a chain of inverters designed such that EN_{delay} rises at decision moment, T_{DM} ($T_{HRS} < T_{DM} < T_{LRS}$). At the rising edge of EN_{delay}, flip-flop input I_{FF} and its complement are available at the sense amplifier output as $M(a,b,c)$ and $\overline{M(a,b,c)}$. The SA will output $M(a,b,c)$ correctly for all the eight cases (000 to 111) if it is designed to differentiate between 001 case and 011 case. The correct functioning for all the eight cases was verified by simulation in Cadence Spectre. The sense amplifier was simulated with temperature variation (27±10°C) across process corners and the correct output was verified. The resilience of this SA to ReRAM variations and CMOS process variations were evaluated by extensive Monte Carlo simulations in [7].

B. WRITE Circuit

To accelerate in-memory computation, it is required to be able to write multiple cells of a row simultaneously. This was accomplished by using an op-amp as a driver, as shown in Fig. 6. Each 1T-1R cell requires $\approx 240\mu A$ to switch, and the op-amp is designed to drive eight 1T-1R cells simultaneously. To write into more than eight cells, multiple op-amps should be used. Fig. 6 illustrates the SET ($HRS \rightarrow LRS$) and RESET ($LRS \rightarrow HRS$) operation. The RESET operation in ReRAM is accomplished by applying a positive voltage to the SL, while BL is grounded, and SET operation is accomplished by applying a positive voltage to BL, while SL grounded. This is because Resistive RAM requires a voltage of opposite polarity to break the filament. The circuit of Fig. 6 was simulated in 45 nm technology to verify the WRITE circuit. V_{WRITE} of 1.2 V was used during SET and RESET operation and V_{WL} of 1.1 V was applied at the gate of the access transistor. Simultaneous writing into 8 cells of a row was verified by simulation. As will be explained in Section V-B, this aids in accelerating in-memory multiplication.

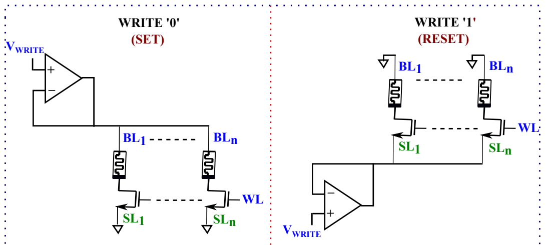

Fig. 6: The operational amplifier regulates the voltage while driving enough current to switch the cell. The op-amp is connected to BL/SL of the ReRAM cells and SL/BL is grounded for WRITE '0'/'1' operation.

C. Row Decoder, Column Selection Logic and Memory controller

Conventional decoder in memory array can select only one row at a time but we need to select three rows simultaneously during majority operation. Furthermore, single-row decoding is needed during normal memory READ/WRITE operation.

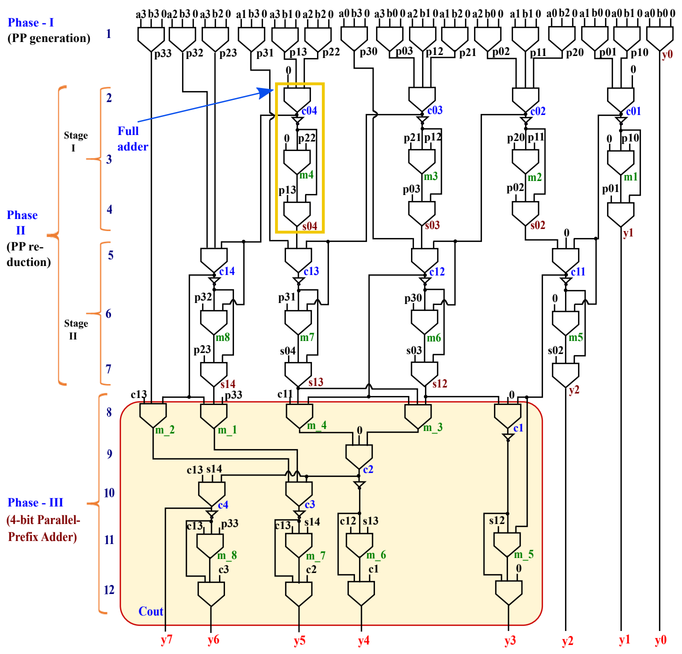

Fig. 7: The three phases of a 4×4 Wallace Tree multiplier (Fig. 1) expressed as 12 levels of majority logic gates. Levels 8-12 are the 4-bit parallel-prefix adder (Ladner Fischer) used to accelerate addition in Phase-III.

The triple-row decoder presented in [7] was used to accomplish this since it can switch between single-row and triple-row activation seamlessly. An additional control signal, MAJ is used, which when set to '1' selects three rows simultaneously (Majority operation) and when set to '0' selects a single row (normal memory READ/WRITE operation). Column Selection logic is responsible for selecting the appropriate columns (to be written to or read from) and connecting them to the READ or WRITE circuit. The memory controller coordinates the READ and WRITE operations by supplying control signals to the peripheral circuitry. More details of these circuits can be found in [7].

D. Energy for In-Memory Operations

The energy for multiplication can be computed by summing the energy for various logic operations. Multiplication is carried out as a sequence of READ (majority) and WRITE operations. Energy for a single READ operation is calculated as $E_{READ} = V_{DD}\int_0^{t_{READ}} I_{READ}\,dt + V_{DD}\int_0^{t_{READ}} I_{SA}\,dt$, where I_{READ} is the current fed to the 1T-1R cell (see Fig. 5), I_{SA} is the current ingested by the TBSA, and t_{READ} is the duration of READ cycle. In this work, t_{READ} was 20 ns and I_{READ} was 25 μA for our simulations in 45-nm CMOS process. The energy for a single majority operation, E_{READ} = 2.2 pJ/operation. The energy to write a single bit to 1T-1R array is $E_{WRITE} = V_{WRITE}\int_0^{t_{WRITE}} I_{WRITE}\,dt$, where t_{WRITE} was 50 ns in our simulation and E_{WRITE} = 8.2 pJ/bit.

V. Proposed 4×4 In-Memory Wallace Tree Multiplier

A. 4×4 *Wallace Tree Multiplier using Majority Logic*

The majority gate based implementation of a 4×4 Wallace Tree multiplier of Fig. 1 is shown in Fig. 7. All partial products (PPs) are generated in the first step using majority gates with '0' as the third input, $i.e.$ $p_{ab} = a \cdot b = M(a, b, 0)$. In the partial product reduction phase (Stages I and II), majority logic based full adders are used for addition and the addition process is carried out in parallel. The generated sums and carries are used as the inputs for addition in the next stage. A full adder needs three majority gates and can be implemented in memory in six cycles (3 READ and 3 WRITE cycles) (Fig. 2). The latency for addition in each stage except the last stage is equivalent to the latency of 1-bit full adder, $i.e.$ 6 cycles in-memory. As discussed in the Section II, final stage is the critical stage involving carry propagation. A majority based 4-bit parallel-prefix adder (Ladner Fischer) formulated using the procedure followed in [7] is used in the final phase to accelerate the addition. A 4×4 Wallace Tree multiplier circuit using majority logic given in Fig. 7 has 12 levels of majority gates where levels 8-12 is the 4-bit parallel-prefix adder.

B. *Mapping of the* 4×4 *Wallace Tree Multiplier to 1T-1R Array*

In this section, mapping of the 4×4 Wallace Tree multiplier structure of Fig. 7 to a 1T-1R array using majority logic and the sequence of operations are addressed. Since the majority gates are not stateful, the output of the majority gate (voltage) needs to be written to the array as inputs to the next logic level. The outputs of a logic level are written in contiguous rows of the array so that they can be the inputs to the majority gates in the next logic level. All the cells in the memory array are initialized to logic '0'(LRS). The two numbers to be multiplied ($a_3a_2a_1a_0$ and $b_3b_2b_1b_0$) are arranged in the memory array as shown in Fig. 8. For optimized latency, the multiplier is mapped in such a way that all the majority gates in a particular logic level are executed simultaneously in a READ operation. Furthermore, considering the limited endurance of ReRAM devices [38], we write each cell in the processing area only once during the entire operation of 4×4 multiplication. All the intermediate results of multiplication after each stage of computation are written in different memory locations and not overwritten on the same location. Since ReRAM requires voltage of opposite polarity to break the conductive filament, SET and RESET operations cannot be performed on multiple columns simultaneously. Therefore, writing multiple bits to any row is usually done in two steps (writing all 1's in the sequence in one step followed by writing all 0's in the sequence in next step). In our mapping, a single cycle is sufficient to write multiple bits in a row since all the cells are initialized to '0' $i.e.$ it is enough to write 1's in the desired locations. The writing of 1's is a RESET operation which is accomplished by using the op-amp driver circuit (see Section IV-B) to switch the multiple ReRAM cells at the locations where '1' has to be written.

The contents of the memory array during the execution of the twelve logic levels are shown in Fig. 8. The description of steps is given below:

1) Majority at col. (1 to 16);
2) Write (p33,p33,p32,p13,p31,p03,p30,p02,p01) at col. (2,3,4,5,8,9,11,12,15) of row 5;
3) Write (p23,p22,p12,p11,p10,y0) at col. (4,5,9,12,15,16) of row 6;
4) Write (p21,p20,1) at col. (9,12,16) of row 7;
5) Majority at col. (5,9,12,15);
6) Write (c04,$\overline{c04}$,c03,$\overline{c03}$,c02,$\overline{c02}$,c01,$\overline{c01}$) at col. (4,5,8,9,11,12,14,15) of row 4;
7) Write ($\overline{c04}$,$\overline{c03}$,$\overline{c02}$,$\overline{c01}$) at col. (5,9,12,15) of row 8;
8) Majority at col. (5,9,12,15);
9) Write (m4,m3,m2,m1) at col. (5,9,12,15) of row 3;
10) Majority at col. (5,9,12,15);
11) Write (s04,y1,s03,s02) at col. (8,10,11,14) of row 6;
12) Majority at col. (4,8,11,14);
13) Write (c14,c14,$\overline{c14}$,$\overline{c13}$,c12, $\overline{c12}$,c11,$\overline{c11}$) at col. (2,3,4,8,10,11,13,14) of row 3;
14) Write (c13,c11,c13,c12) at col. (2,6,7,13) of row 4;
15) Write ($\overline{c14}$,$\overline{c13}$,1,$\overline{c12}$,$\overline{c11}$) at col. (4,8,10,11,14) of row 7;
16) Majority at col. (4,8,11,14);
17) Write (m8,m7,m6,m5) at col. (4,8,11,14) of row 8;
18) Majority at col. (4,8,11,14);
19) Write (s14,s13) at col. (3,10) of row 4;
20) Write (s12,s14,s12,s13) at col. (6,7,10,13) of row 5;
21) Majority at col. (2,3,6,10,13);
22) Write (m_3,m_1) at col. (1,16) of row 5;
23) Write (m_4,m_2) at col. (1,16) of row 4;
24) Majority at col. (1);
25) Write c2 at col. (7,16) of row 3;
26) Majority at col. (7,16);
27) Write (c4,$\overline{c4}$,$\overline{c1}$,$\overline{c3}$,$\overline{c2}$) at col. (1,2,6,7,13) of row 6;
28) Write (1,c3,c2,c1) at col. (1,2,7,13) of row 7;
29) Majority at col. (1,2,6,7,13);
30) Write (m_8,m_5,m_7,m_6) at col. (2,6,7,13) of row 8;
31) Majority at col. (1,2,6,7,10,13,14,16);
32) Write (y7,y6,y5,y4,y3,y2,y1,y0) to the memory array.

C. *Simulation of* 4×4 *Wallace Tree Multiplier in 1T-1R Array*

Simulations were carried out using a 1T-1R array consisting of a 45nm NMOS transistor in series with a ReRAM device as the memory cell. The Stanford-PKU model which is a physics-based model was used to model the filamentary switching mechanism. The algorithm proposed in [39] was used to exactly fit the model to the characteristics of IHP's 1T–1R cell [36] (HRS= 200 kΩ and LRS=10.5 kΩ, V_{SET}= 0.8 V, V_{RESET} = -1.1 V). Following the mapping described in Section V-B, the functionality of a 4-bit in-memory multiplier was verified by executing a sequence of READ (Majority), and WRITE operations. The simulation was performed using Cadence Virtuoso tool using SAs and WRITE circuit designed in 45-nm CMOS technology. Multiple majority operations were performed in parallel by using a dedicated SA for each column in the 1T-1R array. As described in the Section

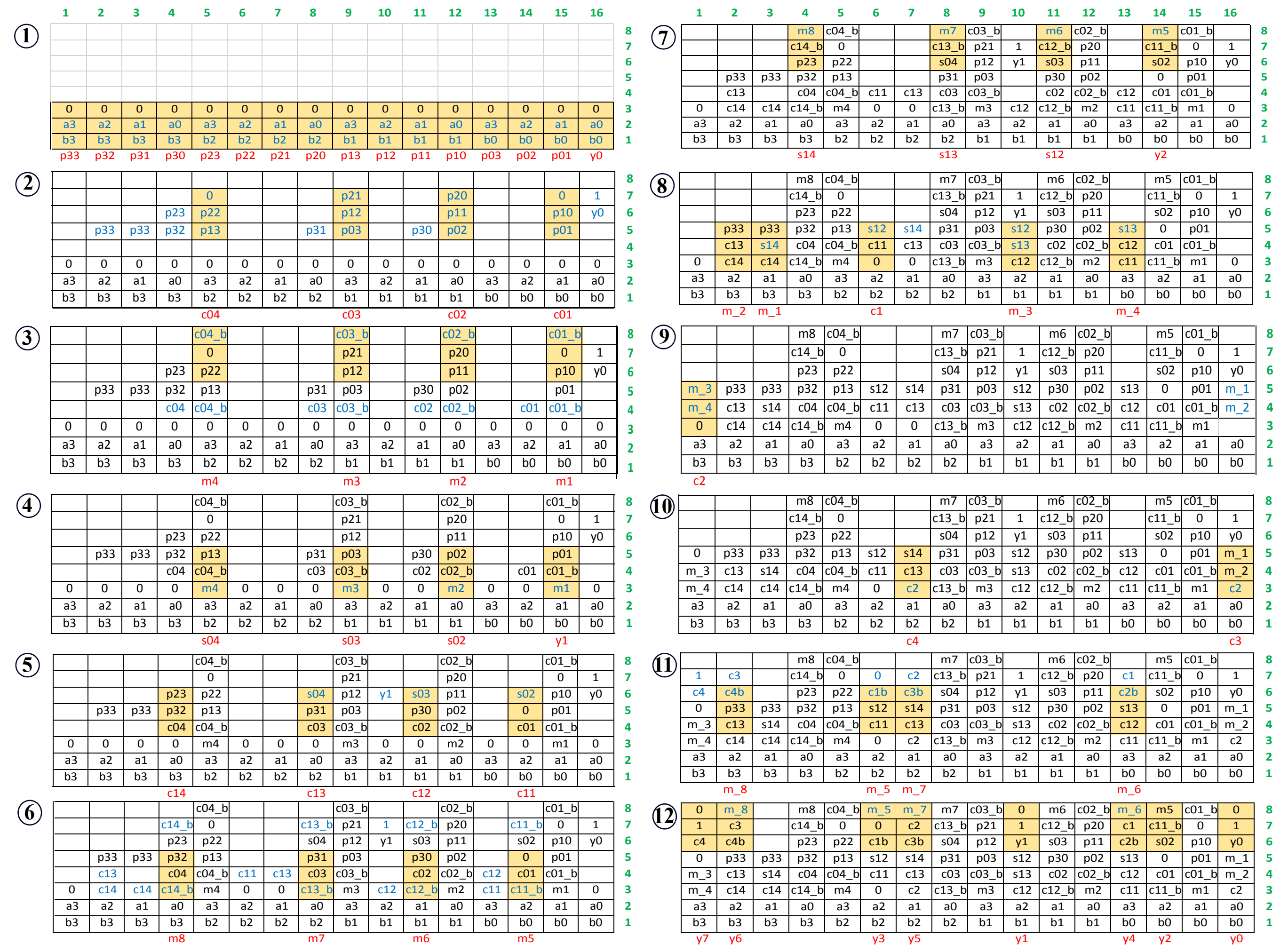

Fig. 8: Mapping of the 4×4 Wallace Tree multiplier (12 logic levels) of Fig. 7 to 1T-1R array. All the majority gates in a level are executed in parallel (shaded portions). Each majority logic level of Fig. 7 is mapped to 1T-1R array of size 8×16 (each rectangular box is assumed as a 1T-1R cell). In each step, the majority output is marked in red and the new contents written to the array in blue.

V-B, the result of multiplication is available at the 31st cycle. Simulation result of a 4×4 Wallace Tree Multiplier for the inputs A = 1010 and B = 0101 is shown in Fig.9.

VI. $n \times n$ In-Memory Wallace Tree Multiplier

A. 8×8 *In-memory Wallace Tree Multiplier*

In this section, we extend the proposed in-memory multiplier to design 8×8 and $n \times n$ Wallace Tree multipliers in memory. Specifically, we consider a 8×8 Wallace Tree multiplier using majority logic and calculate the latency to execute in memory. To accelerate the addition in the final stage of 8×8 Wallace Tree multiplier, a majority-based 10-bit parallel prefix (Ladner Fischer) adder is chosen. The majority-based circuit of 8×8 Wallace Tree multiplier is shown in Fig. 10. From the analysis of 4×4 and 8×8 multipliers, we formulate the latency required to execute $n \times n$ multiplier in memory. Comparing Fig. 7 and 10, it is clear that from 4×4 to 8×8 multipliers, the number of logic levels is increased from 12 to 19. This is the advantage of Wallace Tree architecture using majority logic and we will transfer this latency advantage to in-memory implementation as well. For 8×8 multiplier, two different ways of mapping were experimented: Latency optimized mapping and Area optimized mapping.

1) Latency optimized mapping: This method of mapping for 8×8 multiplier follows the same procedure that was used for 4×4 multiplier (see Section V-B). ReRAM cells are not reused during multiplication *i.e.* each cell is used only once during the entire multiplication process. This is endurance-friendly mapping considering the limited endurance of ReRAM devices [40]. Initially all the cells are initialized to logic '0'. When the new contents are to be written to the array during the multiplication process, only logic '1' is written to multiple cells in parallel using op-amp circuitry (see Section IV-B). This minimizes the number of write cycles there by reducing the latency of overall multiplication. However, to avoid overwriting, each intermediate result is written to a new location in the memory and, therefore, the area of the array increases marginally. Following this method, the proposed 8×8 multiplication in-memory requires a total of 58 cycles (39 WRITE and 19 READ cycles). The 19 logic levels of 8×8 multiplier require 59 memory cycles, as shown in Fig. 10. The number of ReRAM cells utilized for the 8×8

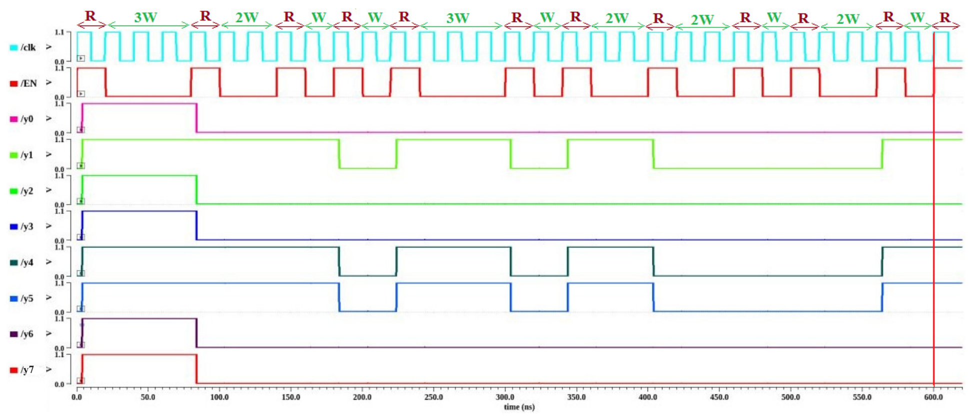

Fig. 9: Simulation result of a 4×4 Wallace Tree Multiplier for the inputs A = 1010 and B = 0101. EN is enable signal of SA which is high during the READ operation (R = READ & W = WRITE). Multiplication result is available at the 31st cycle (600n–620ns) marked with a red line and the result is $y_7y_6y_5y_4y_3y_2y_1y_0$ = 00110010.

multiplication in-memory is 560. Using this method, $nlog_2n$ latency is accomplished for $n \times n$ multiplication in memory. The proposed approach is one of the fastest implementations of multiplication in ReRAM array.

For 8×8 multiplier in this mapping, the number of majority operations is 283 and 556 bits are written. The total energy for majority operations is therefore 283 $\times$ 2.2 pJ/majority operation = 622 pJ. The total WRITE energy is 556 $\times$ 8.2 pJ/bit = 4559 pJ. The energy for 8×8 multiplications is 5181 pJ. The switching time, HRS, LRS and other parameters of ReRAM vary from device to device and energy value depends on HRS and LRS values and also the switching time. For accurate energy comparison, all the multipliers should be simulated using the same ReRAM device.

2) Area optimized mapping: In this method of mapping, some ReRAM cells which are used in the previous computation steps are re-used to write the intermediate values (resulting after READ operation) during the multiplication process. When the cells are re-used, there is no guarantee that the initial state is '0' unlike latency optimized mapping (Section VI-A1) where all cells are initialized to logic '0' and are not re-used. Here writing of logic '1' and logic '0' are performed separately when the intermediate results have to be written to the array. The simultaneous writing of logic '1' and '0' is not possible because it is required to connect op-amp circuitry to bit lines while writing '0' and source lines while writing '1' (Fig. 6). For example, to write "10101010" to a row, in area optimized mapping, first "_0_0_0_0" is written by a SET operation, and then, "1_1_1_1_" is written by a RESET operation[3]. This doubles the number of WRITE cycles compared to the latency-optimized mapping leading to an increase in overall latency of multiplier. The number of cells utilized is less in this mapping method due to overwriting of cells. Using this approach by reusing 40% of the cells, 8×8 multiplication in-memory requires 99 cycles (80 WRITE and 19 READ cycles) and 270 ReRAM cells. In this work, since our main focus is on latency optimization, we focused more on latency-optimized mapping for $n \times n$ in-memory multiplier.

[3]In latency-optimized method, writing "10101010," can be done in a single step by writing "1_1_1_1_" using parallel RESET operation in those four locations and there is no need to write other locations as all the memory cells are initialized to '0' (SET)

B. $n \times n$ *In-Memory Wallace Tree Multiplier*

The size of the 1T-1R array required for $n \times n$ In-Memory Wallace Tree Multiplier grows as $8 \times (n^2 + 6log_2\frac{n}{4})$. The number of partial product reduction stages for $n \times n$ Wallace Tree multiplier follows $log_2\frac{n^2}{4}$ (see Section II) and each stage has 3 levels of majority gates. Therefore, the number of levels of majority gates for partial product reduction phase in $n \times n$ in-memory multiplier is $3(log_2\frac{n^2}{4})$. Assuming the inputs to be multiplied are in the memory, the latency of in-memory computation can be formulated as follows:

- One read cycle is required to compute the partial products.
- $n \times n$ Wallace Tree multiplier has $log_2\frac{n^2}{4}$ partial product reduction stages (intermediate stages of addition) and each stage requires 6 cycles for addition using majority logic.
- $n \times n$ multiplier needs $2(n - log_2n)$ bit adder in the final stage. The number of cycles required for parallel-prefix addition in the last stage is $4\lfloor log_2 2(n-log_2n)\rfloor+6$ cycles (Cycles required for p-bit in-memory addition is $4\lfloor log_2p\rfloor + 6$, see ref [7]).
- The write cycles required to map the intermediate results to desired locations in memory are $(n-2)\lfloor log_2(n-2)\rfloor$+3.
- The total number of cycles required to compute $n \times n$ Wallace Tree multiplication in-memory is given by, Latency = $6log_2\frac{n^2}{4}+4\lfloor log_2 2(n-log_2n)\rfloor+(n-2)\lfloor log_2(n-2)\rfloor + 10$

The above equation is verified for n = 2, n = 4 and n = 8. The number of ReRAM cells utilized for the computation of 4×4 and 8×8 multiplication in-memory without overwriting the contents of cells is 128 and 560 respectively. For $n \times n$ multiplication in-memory, the number of ReRAM cells required is given by $8n^2 + 48log_2\frac{n}{4}$.

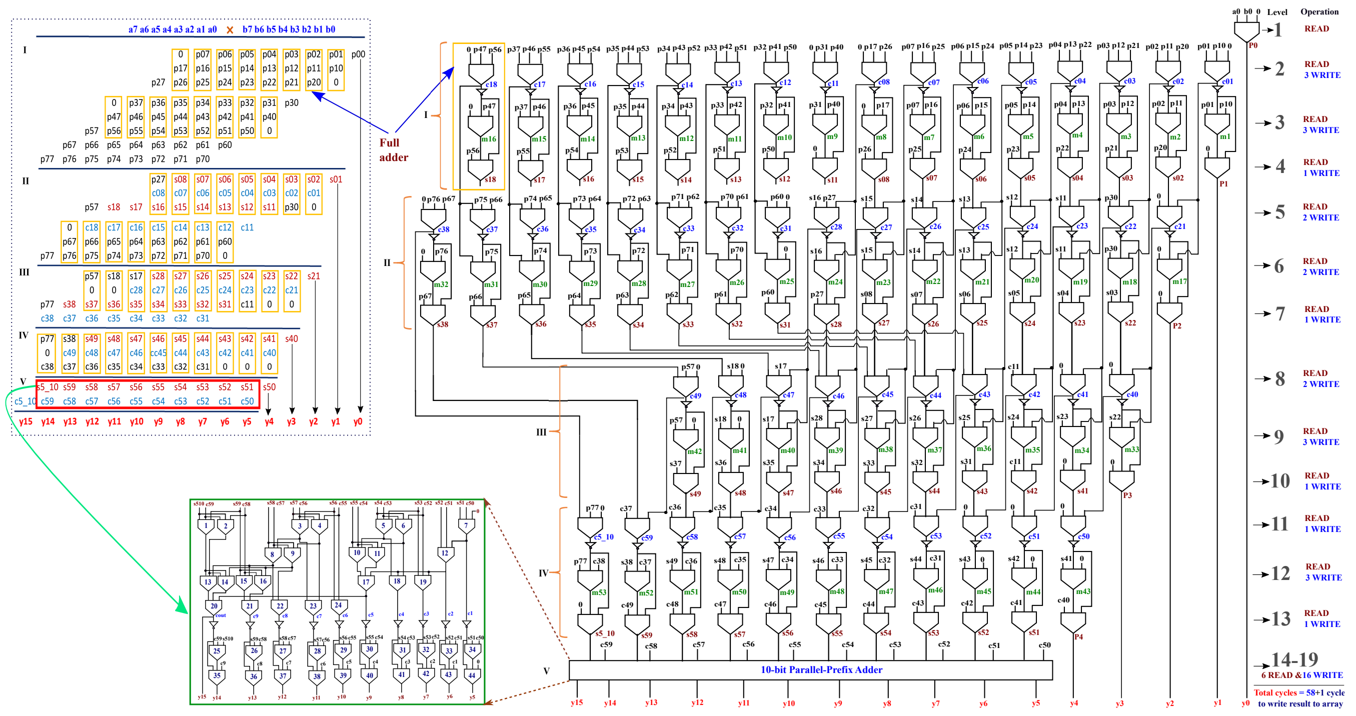
Full adder
10-bit Parallel-Prefix Adder
Level
Operation
READ
3 WRITE
1 WRITE
2 WRITE
6 READ &16 WRITE
Total cycles = 58+1 cycle to write result to array

Fig. 10: The five stages (I-V) of a general 8×8 Wallace Tree multiplier expressed as 19 levels of majority gates. READ and WRITE operations performed at each level are shown next to the level number. Levels 14-19 are the 10-bit parallel-prefix (Ladner Fischer) adder used to accelerate addition in the last stage to produce result. Total number of cycles required for 8×8 multiplication is 59, out of which 58 cycles are for the entire multiplication process and 1 cycle is to write result to the memory array.

TABLE I: Comparison of the proposed (8×8) In-memory multiplier with previous works in terms of Latency (# Cycles), Area (# Memristors & Modifications to peripheral circuitry)

Comparison of 8×8 Memristor Based Multipliers					
Multiplier	Logic Primitive	# Cycles	# Memristors	Modifications to Peripheral Circuitry	Array Compatibility
Shift and Add [16]	IMPLY Gate	296	57	63 Switches & 56 Drivers	No
Array [17]	IMPLY Gate	157	393	449 Switches & 449 Drivers	No
Dadda [18]	IMPLY Gate	106	385	482 Switches & 390 Drivers	No
Semi-Serial Adder Based [19]	IMPLY Gate	280	138	51 Switches	Moderate (switches, R_G)
Full Precision Fixed Point [41]	NOR Gate	726	155	NA*	Yes (1S-1R)
MultPIM [20]	Minority+NOT	139	105	NA*	Yes (1S-1R)
Proposed(Latency Optimized)	Majority Gate	59	560 (8×70 array)	Triple row decoder	Yes (1T-1R)
Proposed(Area Optimized)	Majority Gate	99	270	Triple row decoder	Yes (1T-1R)

* Source does not provide information about modifications to peripheral circuitry.

VII. Comparison with the Previous Works

In this section, we compare the proposed in-memory Wallace Tree multiplier with other in-memory multipliers in literature. A direct comparison with CMOS multipliers is not possible due to lack of accurate numbers for DRAM access latency/energy[4]. In literature, not all memristor-based multiplier works can be classified as in-memory multipliers. For example, the works [16]–[18] implement multipliers using memristors but not in a memory array. The work in [19] uses IMPLY logic to implement a multiplier in a memory array, but with significant modifications to the memory array (switches in certain parts of the array and work resistor R_G). In contrast, our work is true in-memory multiplication since the computation is performed as a sequence of READ/WRITE operations. Table I & II summarizes the metrics of latency, area (both array area and peripheral circuitry) and the degree of compatibility to a regular memory array. From Table I, it is clear that the proposed multiplier outperforms all existing multipliers in terms of latency. The multiplier proposed in this work (Latency optimized) achieves 44% less latency than the multiplier with least latency reported in literature (Dadda multiplier with 106 cycle latency). Further, IMPLY-based multipliers (Shift and Add, Array, Dadda & Semi-serial adder based) require additional switches and drivers (Table I). Multipliers based on NOR ([41]) and Minority+NOT ([20]) require an isolation

[4]The latency/energy of in-memory multiplier must be compared with the combined time/energy needed to fetch data from DRAM memory and be multiplied in CMOS-based processor.

TABLE II: Comparison of the proposed ($n \times n$) In-memory multiplier with previous works in terms of Latency (# Cycles), Area (# Memristors & Modifications to peripheral circuitry)

Comparison of $n \times n$ Memristor Based Multipliers				
Multiplier	# Cycles	# Memristors	Modifications to Peripheral Circuitry	Array Compatibility
Shift and Add [16]	$2n^2 + 21n$	$7n + 1$	$8n - 1$ Switches & $7n$ Drivers	No
Array [17]	$24n - 35$	$7n^2 - 8n + 9$	$8n^2 - 9n + 9$ Switches & Drivers	No
Dadda [18]	$NA*$	$NA*$	$NA*$	No
Semi-Serial Adder Based [19]	$\lceil log_2 n \rceil (10n + 2) + 4n + 2$	$2n^2 + n + 2$	$12\lceil \frac{n}{2} \rceil + \lceil \frac{n-1}{2} \rceil$ Switches	Moderate (switches, R_G)
Full Precision Fixed Point [41]	$13n^2 - 14n + 6$	$20n - 5$	$NA*$	Yes (1S-1R)
MultPIM [20]	$nlog_2(n) + 14n + 3$	$14n - 7$	$NA*$	Yes (1S-1R)
Proposed(Latency Optimized)	$6log_2 \frac{n^2}{4} + 4\lfloor log_2 2(n - log_2 n) \rfloor + (n-2)\lfloor log_2 (n-2) \rfloor + 10$	$8n^2 + 48log_2 \frac{n}{4}$	Triple row decoder	Yes(1T-1R)

* Source does not provide this information

voltage (to be applied at rows where NOR operation is not intended) and an execution voltage (for NOR operation). Thus, two additional voltage sources are needed and the peripheral circuitry must have the capability to multiplex atleast five voltage sources ($V_{READ}, V_{WRITE}, V_{GND}, V_{ISO}, V_{EXE}$) to each row/column. Therefore, existing multipliers need significant modifications to the peripheral circuitry, whereas the proposed multiplier requires only a minor modification to the row decoder (to select three consecutive rows during majority operation). The number of memristors required for the proposed multiplier is more owing to high degree of gate-level parallelism employed and endurance-friendly mapping. But memristor is a nanoscale device which occupies very less area and even if the number of memristors is increased, this does not contribute to significant increase in the total area (to compare two in-memory multipliers, both the array area and the increased peripheral circuitry area required to accommodate logic operations must be determined [7]). The proposed multiplier is energy efficient due to the absence of sneak path currents in 1T–1R configuration. In Tables I & II, the energy for different multipliers is not included since the energy of other multipliers is either not reported or reported for different ReRAM technology. As stated in Section IV-D, comparison of energy of different multipliers is inconclusive since the switching energy ($HRS \rightleftharpoons LRS$) varies across different ReRAM technologies. From Table I, proposed in-memory Wallace Tree multiplier is the fastest among the existing in-memory multipliers without significant increase in total area (memory array area + peripheral circuitry area) and also endurance-friendly.

The comparison of the proposed and the existing $n \times n$ memristor-based multipliers using the formulations from Section VI-B is given in Table II. Using the formulations in Table II to perform 64×64 (n = 64) multiplication in-memory, Shift and Add multiplier requires 9536 cycles, Array multiplier requires 1501 cycles, Multiplier using semi-serial adder requires 4110 cycles, Full precision fixed point multiplier requires 52358 cycles, MultPIM requires 1283 cycles and the proposed multiplier requires 404 cycles. From this example, it is clear that the proposed multiplier provides optimum latency for intensive multiplication and the latency grows as $O(n \cdot logn)$.

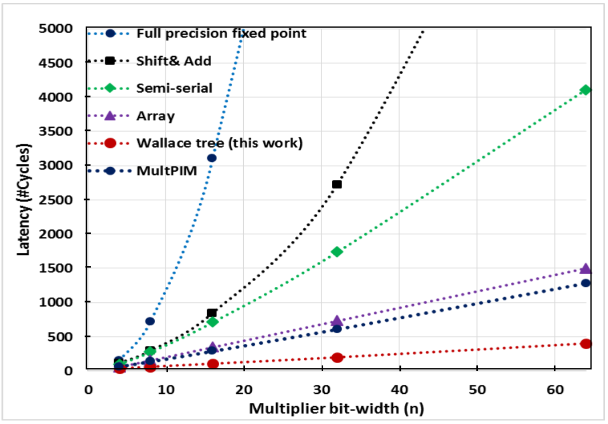

Fig. 11: Latency variation of in-memory multipliers with increase in bit-width. In-memory multipliers, Shift & Add, Array, Semi-serial adder based, Full precision fixed point and MultPIM require thousands of cycles for 64-bit multiplication, while the proposed in-memory multiplier requires only 404 cycles.

VIII. Conclusion

In this work, we have implemented a novel in-memory Wallace Tree multiplier using majority gates. Latency optimized in-memory multiplication is achieved by combining the strength of majority logic primitive and speed of Wallace Tree architecture. High degree of gate-level parallelism is employed since majority gates can be executed simultaneously in the columns of the array. In this manner, latency is optimized at each level– at the architecture level by choosing Wallace Tree, at the circuit level by choosing majority gates and in the memory array by parallel operations in the columns of the array. In addition to accelerating multiplication in 1T-1R array, the proposed multiplier is energy-efficient since energy associated with sneak currents is negligible in the 1T-1R array. Since the proposed multiplier performs parallel majority operations to minimize latency without overwriting the cells, it requires considerable area of the array but no major modifications to the peripheral circuitry of the memory array.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[2] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 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, 2016.

[4] S. F. Yitbarek, T. Yang, R. Das, and T. Austin, "Exploring specialized near-memory processing for data intensive operations," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1449–1452.

[5] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[6] 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.

[7] J. Reuben and S. Pechmann, "Accelerated addition in resistive ram array using parallel-friendly majority gates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 6, pp. 1108–1121, 2021.

[8] I. Giannopoulos, A. Sebastian, M. Le Gallo, V. Jonnalagadda, M. Sousa, M. Boon, and E. Eleftheriou, "8-bit precision in-memory multiplication with projected phase-change memory," in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 27.7.1–27.7.4.

[9] H. Zhang, W. Kang, K. Cao, B. Wu, Y. Zhang, and W. Zhao, "Spintronic processing unit in spin transfer torque magnetic random access memory," *IEEE Transactions on Electron Devices*, vol. 66, no. 4, pp. 2017–2022, 2019.

[10] X. Yin, X. Chen, M. Niemier, and X. S. Hu, "Ferroelectric fets-based nonvolatile logic-in-memory circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 159–172, 2019.

[11] S. Yu, "Resistive random access memory (rram)," *Synthesis lectures on emerging engineering technologies*, vol. 2, no. 5, pp. 1–79, 2016.

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

[13] Y. S. Kim, M. W. Son, and K. M. Kim, "Memristive stateful logic for edge boolean computers," *Advanced Intelligent Systems*, vol. 3, no. 7, p. 2000278, 2021.

[14] N. Xu, T. Park, K. J. Yoon, and C. S. Hwang, "In-memory stateful logic computing using memristors: Gate, calculation, and application," *physica status solidi (RRL) – Rapid Research Letters*, vol. 15, no. 9, p. 2100208, 2021.

[15] 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.

[16] L. Guckert and E. E. Swartzlander, "Optimized memristor-based multipliers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 2, pp. 373–385, 2017.

[17] L. E. Guckert *et al.*, "Memristor-based arithmetic units," Ph.D. dissertation, 2016.

[18] L. Guckert and E. E. Swartzlander, "Dadda multiplier designs using memristors," in *2017 IEEE International Conference on IC Design and Technology (ICICDT)*. IEEE, 2017, pp. 1–4.

[19] D. Radakovits, N. TaheriNejad, M. Cai, T. Delaroche, and S. Mirabbasi, "A memristive multiplier using semi-serial imply-based adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1495–1506, 2020.

[20] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Multpim: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2021.

[21] D. R. Gandhi and N. N. Shah, "Comparative analysis for hardware circuit architecture of wallace tree multiplier," in *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. IEEE, 2013, pp. 1–6.

[22] F. U. D. Farrukh, C. Zhang, Y. Jiang, Z. Zhang, Z. Wang, Z. Wang, and H. Jiang, "Power efficient tiny yolo cnn using reduced hardware resources based on booth multiplier and wallace tree adders," *IEEE Open Journal of Circuits and Systems*, vol. 1, pp. 76–87, 2020.

[23] 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, 2018.

[24] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2015.

[25] I. A. Young and D. E. Nikonov, "Principles and trends in quantum nano-electronics and nano-magnetics for beyond-cmos computing," in *2017 47th European Solid-State Device Research Conference (ESSDERC)*. IEEE, 2017, pp. 1–5.

[26] 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, 2017.

[27] 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.

[28] Y. devi Ykuntam, K. Pavani, and K. Saladi, "Design and analysis of high speed wallace tree multiplier using parallel prefix adders for vlsi circuit designs," in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2020, pp. 1–6.

[29] U. Kumar and A. Fam, "Enhanced wallace tree multiplier via a prefix adder," in *2020 IEEE Student Conference on Research and Development (SCOReD)*. IEEE, 2020, pp. 211–216.

[30] W. J. Townsend, E. E. Swartzlander Jr, and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in *Advanced signal processing algorithms, architectures, and implementations XIII*, vol. 5205. International Society for Optics and Photonics, 2003, pp. 552–560.

[31] Z.-R. Wang, Y. Li, Y.-T. Su, Y.-X. Zhou, L. Cheng, T.-C. Chang, K.-H. Xue, S. M. Sze, and X.-S. Miao, "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, 2018.

[32] L. Cheng, Y. Li, K.-S. Yin, S.-Y. Hu, Y.-T. Su, M.-M. Jin, Z.-R. Wang, T.-C. Chang, and X.-S. Miao, "Functional demonstration of a memristive arithmetic logic unit (memalu) for in-memory computing," *Advanced Functional Materials*, vol. 29, no. 49, p. 1905660, 2019.

[33] S. G. Rohani, N. Taherinejad, and D. Radakovits, "A semiparallel full-adder in imply logic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 297–301, 2019.

[34] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[35] E. Testa, M. Soeken, L. G. Amar, and G. De Micheli, "Logic synthesis for established and emerging computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 165–184, 2018.

[36] S. Pechmann, T. Mai, M. Völkel, M. K. Mahadevaiah, E. Perez, E. Perez-Bosch Quesada, M. Reichenbach, C. Wenger, and A. Hagelauer, "A versatile, voltage-pulse based read and programming circuit for multi-level rram cells," *Electronics*, vol. 10, no. 5, p. 530, 2021.

[37] Q.-K. 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, 2018.

[38] M.-F. Chang, C.-H. Chuang, M.-P. Chen, L.-F. Chen, H. Yamauchi, P.-F. Chiu, and S.-S. Sheu, "Endurance-aware circuit designs of nonvolatile logic and nonvolatile sram using resistive memory (memristor) device," in *17th Asia and South Pacific Design Automation Conference*. IEEE, 2012, pp. 329–334.

[39] 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.

[40] S.-Y. Hu, Y. Li, L. Cheng, Z.-R. Wang, T.-C. Chang, S. M. Sze, and X.-S. Miao, "Reconfigurable boolean logic in memristive crossbar: The principle and implementation," *IEEE Electron Device Letters*, vol. 40, no. 2, pp. 200–203, 2018.

[41] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, “Imaging: In-memory algorithms for image processing,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4258–4271, 2018.