RIME: A Scalable and Energy-Efficient Processing-In-Memory Architecture for Floating-Point Operations

Zhaojun Lu
lzj201377521@gmail.com
Huazhong University of Science and Technology
Wuhan, China

Md Tanvir Arafin
mdtanvir.arafin@morgan.edu
Morgan State University
Baltimore, Maryland

Gang Qu
gangqu@umd.edu
University of Maryland
College Park, Maryland

ABSTRACT

Processing in-memory (PIM) is an emerging technology poised to break the *memory-wall* in the conventional von Neumann architecture. PIM reduces data movement from the memory systems to the CPU by utilizing memory cells for logic computation. However, existing PIM designs do not support high precision computation (*e.g.*, floating-point operations) essential for critical data-intensive applications. Furthermore, PIM architectures require complex control module and costly peripheral circuits to harness the full potential of in-memory computation. These peripherals and control modules usually suffer from scalability and efficiency issues.

Hence, in this paper, we explore the analog properties of the resistive random access memory (RRAM) crossbar and propose a scalable RRAM-based in-memory floating-point computation architeture (RIME). RIME uses single-cycle NOR, NAND, and Minority logic to achieve floating-point operations. RIME features a centralized control module and a simplified peripheral circuit to eliminate data movement during parallel computation. An experimental 32-bit RIME multiplier demonstrates 4.8X speedup, 1.9X area-improvement, and 5.4X energy-efficiency than state-of-the-art RRAM-based PIM multipliers.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; **Emerging architectures**; • **Computer systems organization** → *Neural networks.*

KEYWORDS

Processing-in-Memory (PIM); Resistive Random Access Memory (RRAM); Floating-point Multiplier.

ACM Reference Format:
Zhaojun Lu, Md Tanvir Arafin, and Gang Qu. 2021. RIME: A Scalable and Energy-Efficient Processing-In-Memory Architecture for Floating-Point Operations. In *Proceedings of the Asia and South Pacific Design Automation Conference 2021 (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431524

1 INTRODUCTION

Recent progress in data-intensive precision computation have enabled significant breakthroughs in artificial intelligence (AI), machine learning (ML), and big-data centric computation [18]. Unfortunately, the conventional von Neumann architecture suffers from the *memory-wall* that severely limits efficiency and scalability for large-scale computation. For example, in precision computation, data movement between computing units and off-chip memory consumes orders of magnitude more energy than a floating-point operation [16].

Processing-in-memory (PIM) is a promising solution to reduce data movement during computation, and therefore, considered promising to break the *memory-wall* bottleneck in today's computer architecture. Research on existing main memory component (*i.e.,* dynamic random access memory (DRAMs)) has demonstrated progress in in-memory logic and arithmetic computations. Moreover, emerging non-volatile main memory solutions such as resistive random access memory (RRAM) have been exploited in recent literature (*e.g.,* ISAAC [15], PRIME [6] *etc.*) to support in-memory operations. RRAM-based designs demonstrate significant improvement in accelerating basic logic operations due to the high density, fast access, and low leakage of resistive memories.

However, non-volatile resistive PIM designs face several challenges. First, RRAM-based PIM modules seldom provide accurate floating-point computation. For example, the multiply-accumulate operations in [6, 15] are limited by the resolution of multi-level RRAMs, and thus minimally support precision computation. Second, PIM architectures that perform an accurate floating-point operation, such as [7], rely on the costly peripheral circuits and cumbersome external computing units [10]. Third, RRAM-based PIM architectures (*i.e.,* [10, 11]) often execute computations using single bit-wise operation (such as NAND or NOR) on a single bipolar memristor. Although this approach is acceptable for small 4/8-bit designs, it leads to a dramatic increase in latency for 32/64-bit arithmetic computation. Therefore, fundamental design optimization is required for accurate floating-point calculation in a non-volatile memory crossbar.

In this paper, we propose an RRAM-based in-memory floating-point processing architecture (RIME) to address the issues mentioned above. The primary contributions of the work are given below.

First, we present RRAM-compatible single-cycle computation techniques for 3-input Minority (Min_3), NAND, and NOR logic functions. We use these functions as fundamental building blocks for the RIME architecture. The supporting control module for RIME is also

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431524

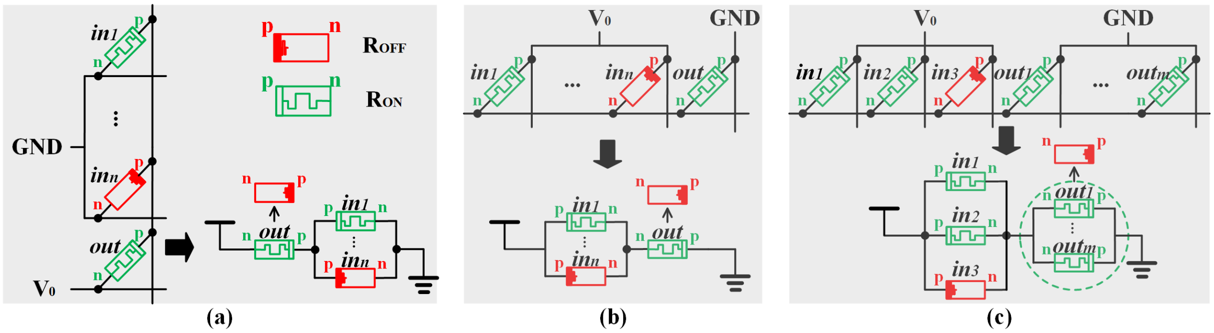

Figure 1: Basic logic operations for RIME (a) n-input NOR in a column (b) n-input NOR in a row (c) m-output Min$_3$ in a row.

designed using these logic circuits and integrated into the RRAM crossbar to reduce the overall complexity of the PIM subsystem.

Second, we provide a detailed implementation of a RIME based multiplier design in an RRAM crossbar. We demonstrate that RIME can support three degrees of parallelism: (1) multiple full-adders in RIME can operate in parallel in the same row of RRAMs; (2) a floating-point operation such as sign bit XORing, the exponent bit addition, and a fixed-point multiplication for the mantissa bits can be executed in parallel; and (3) multiple rows of RRAMs can operate in parallel because RIME utilizes different multipliers with a centralized control module.

Third, we also design and implement the control module and the peripheral circuits for a RIME module. Finally, we compare a 32-bit floating-point RIME multiplier with the state-of-the-art designs [3, 10, 11] in terms of latency, area, power efficiency. We find that a RIME multiplier outperforms current PIM multipliers by 4.8X speedup, 1.9X area efficiency, and 5.4X impermanent in energy consumption.

2 BACKGROUND AND RELATED WORK

An RRAM cell (also known as a memristor) is a two-terminal (*i.e., n* and *p*) bipolar device. The resistance of an RRAM cell is bounded by a high resistive state R_{OFF} (logic 0) and a low resistive state R_{ON} (logic 1). When a bias voltage $V_{pn} > |V_{on}|$ is applied, the memristor is SET from R_{OFF} to R_{ON}. On the other hand, when a reverse-bias voltage $V_{np} > |V_{off}|$ is applied, the memristor is RESET from R_{ON} to R_{OFF}. Here, V_{on} and V_{off} are defined as the voltage thresholds, and they are dependent on the device type and the voltage difference between n and p terminals (V_{np}).

In 2010, Borghetti *et al.* [4] proposed a memristor-based logic gate named IMPLY, that uses material implication operation within an RRAM crossbar. IMPLY performs logic operations by applying sequential voltage activation at different locations in the crossbar. The result is stored in one of the input RRAMs instead of dedicated output RRAM. IMPLY-based designs use costly circuit components and complex control modules [13]. These components make IMPLY gates impractical as a building block for an efficient PIM architecture. Later, Kvatinsky *et al.* presented memristor-aided logic (MAGIC), in which memristors serve as the input with previously stored data, and an additional memristor serves as the output [13]. Imani *et al.* [10] utilized the MAGIC-based NOR gates and proposed FloatPIM that supports floating-point PIM computation. Although NOR is a universal logic gate that can be used to implement addition and multiplication, the computational complexity and energy consumption are still too high for NOR-only-based PIM designs. On the other hand, Amarú *et al.* [1] proposed a majority-inverter graph (MIG) data structure for efficient logic optimization using the majority function together with negation. MIG is a highly flexible in-depth optimization that enables the design of high-speed logic circuits and field-programmable gate array (FPGA) implementations. However, MIG's standard application is not directed towards PIM operation since MIG requires reading the data from the memory array to the peripheral circuit, processing it, and writing it back to the memory. In this work, we describe the RIME architecture for floating-point PIM computation, that fundamentally improves the shortcomings of these exiting design.

3 RIME: LOGIC OPERATIONS

In this section, we describe the operating principles of the basic logic operations used in a RIME architecture. NOR operations in RRAM have been used in other PIM designs; however, for RIME, we propose additional structures for Min$_3$ and NAND operation, and incorporate them for in-memory processing. For our analysis, we use the VTEAM memristor model [14, 19] with $R_{off} = 10\ M\Omega$, $R_{on} = 10\ K\Omega$, $V_{off} = 0.3\ V$ and $V_{on} = -1.5\ V$. We present operation details for the logic gates below.

3.1 NOR

Fig. 1 (a) and Fig. 1 (b) illustrate n-input NOR logic in RRAM crossbar array. Note that NOT is a special case of NOR with $n = 1$. For logic operation, the output RRAM is SET to R_{ON} initially. To execute NOR in a column, as shown in Fig. 1 (a), ground (GND) is connected at the n terminals of the input RRAMs, and an execution voltage (V_0) is applied at the output RRAM's n terminal. In contrast, for row-based NOR operation (Fig. 1 (b)), V_0 is employed at the p terminals of the input RRAMs, and GND is connected at the output RRAM's p terminal. As long as there exists one input RRAM in the R_{ON} state, the output RRAM will be RESET from R_{ON} to R_{OFF}.

For row-implementation of n-input NOR, with all the n input RRAMs at R_{OFF} state, V_{np} of the output RRAM will be lower than V_{off}. For all the other input combinations, V_{np} of the output RRAM should be higher than V_{off} so that the circuit will behave as a NOR gate. In order to avoid the destructive operation of any input RRAM, V_{pn} of all the input RRAMs should be lower than $|V_{on}|$.

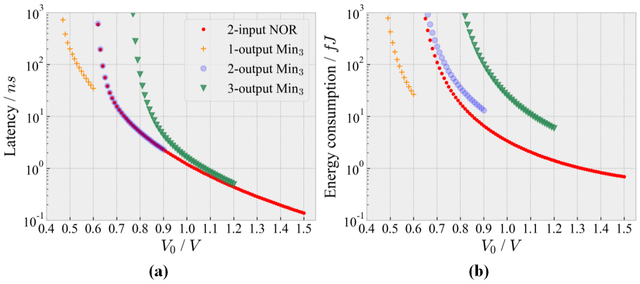

Figure 2: (a) Latency and (b) Energy consumption of basic logic gates in RIME

Since $R_{OFF} >> R_{ON}$, the constraint of V_0 for n-input NOR is given in Equation (1).

$$2 \cdot v_{off} < V_0 \leq |v_{on}| \tag{1}$$

The latency and energy consumption of n-input NOR gate under different V_0 are shown in Fig. 2. It should be noted that the value of n has a negligible impact on the results. As V_0 increases, the latency and energy consumption both exponentially decrease.

3.2 Min₃

A minority gate outputs logic 1 when less than half of the inputs are logic 1. Fig. 1 (c) illustrates m-output Min$_3$ in a row of RRAM crossbar array. To execute Min$_3$, first, the output RRAMs are SET to R_{ON}. Then, a bias voltage V_0 is applied at the p terminals of the input RRAMs, and p terminals of the output RRAMs are connected to GND. As long as more than one input RRAMs are R_{ON}, the m-output RRAMs will be RESET from R_{ON} to R_{OFF}. In contrast, When only one or no input RRAM is R_{ON}, V_{np} of the output RRAMs will be lower than V_{off}, which is insufficient for the RESET operation. Moreover, V_{pn} of all the input RRAMs should be lower than $|V_{on}|$ to avoid over-current. Therefore, the constraint on V_0 for m-output Min$_3$ is given in Equation (2). For example, if $m = 1$, $0.45\ V < V_0 \leq 0.6\ V$, and for $m = 2$, $0.6\ V < V_0 \leq 0.9\ V$.

$$(\frac{m}{2} + 1) \cdot v_{off} < V_0 \leq min\{(m+1) \cdot v_{off},\ |v_{on}|\} \tag{2}$$

The latency and energy consumption of m-output Min$_3$ ($m \in 1, 2, 3$) under different bias voltage V_0 are shown in Fig. 2. According to the experimental results, the 1-output Min$_3$ proposed in [8] has two fundamental drawbacks. First, operating points for NOR and Min$_3$ are different. Therefore, two distinct voltage source is required to integrate 1-output Min$_3$ and NOR in the same crossbar. This significantly increases the complexity of the control module and the peripheral circuits. Second, the minimum latency and energy consumption of the 1-output Min$_3$ are much higher than NOR at their common operating points. Therefore, we use a multiple-output Min$_3$ to address these issues for RIME. This solution comes with the cost of extra RRAMs. For example, note that we can move the operating point of Min$_3$ to $V_0 = 0.9\ V$ when $m = 2$. The latency and energy consumption also remains on the same scale (*i.e.,* 2.29 ns, 13.22 fJ and 2.27 ns, 6.59 fJ for a 2-output Min$_3$ and a NOR gate respectively) in this case.

3.3 NAND

For NAND operation in an RRAM crossbar row, first, the output RRAMs are SET to R_{ON}. Then, V_0 is applied at the p terminals of the input RRAMs, and p terminals of the output RRAMs are connected to GND. When all the n input RRAMs are at R_{ON}, V_{np} of the output RRAMs will be higher than V_{off} and the output RRAMs will be RESET from R_{ON} to R_{OFF}. As V_{pn} of all the input RRAMs should be lower than $|V_{on}|$, the constraint on V_0 for n-input/m-output NAND is given in Equation 3.

$$(\frac{m}{n} + 1) \cdot v_{off} < V_0 \leq min\{(\frac{m}{n-1} + 1) \cdot v_{off},\ |v_{on}|\} \tag{3}$$

The compatibility issues of Min$_3$ with NAND are even trickier. To keep the NAND practical in terms of latency and energy consumption, m should be larger than $n - 1$. For example, when $n = 2$ and $m = 2$, the curves of latency and energy consumption for NAND and 2-output Min$_3$ coincide.

Based on the discussions above, we use n-input NOR, the 2-output Min$_3$, and the 2-input/2-output NAND for RIME. Since both NAND and NOR are universal gates, any logic functions can be implemented using only NAND or NOR.

4 RIME-BASED FLOATING-POINT COMPUTATION

The IEEE-754 format for a 32-bit floating-point number is $X = (-1)^{Sign} \times (1.Fraction)_2 \times 2^{(Exponent-127)}$, where $Sign$ is 1-bit, $Fraction$ is 23-bit, and $Exponent$ is 8-bit. Floating-point operation, such as multiplication of two 32-bit floating-point operands, will require adders and fixed-point multipliers. Therefore, in this section, we design the sub-components of RIME: (1) a full-adder, (2) a fixed-point multiplier, (3) a floating-point multiplier, and (4) a control module. We use the basic logic gates developed in Section 3 to implement these modules. All of these sub-components are integrated into an RRAM crossbar to design RIME-derived floating-point computation units, as shown in Fig. 5.

4.1 Full-Adder

For designing a full-adder (FA), first, we need to develop an XOR gate. As shown in Equation (4), XOR logic can be executed within five clock-cycles using NAND and NOR.

$$A \oplus B = A' \cdot B + A \cdot B' = ((A' \cdot B)' \cdot (A \cdot B')')' \tag{4}$$

Fig. 3, illustrates a 1-bit RIME FA using Min$_3$ and NOR. A RIME FA needs only 11 RRAMs and five clock-cycles for an addition. We assume the inputs, A, B, and C_i are present in the memory before computation. The carry-bit C_o can be expressed with a Min$_3$ and a NOR gate, as shown in Equation (5). On the other hand, the sum S can be expressed with three Min$_3$ and three NOR as Equation (6). Fig. 3 and Table I illustrate how to map the FA into the RRAM

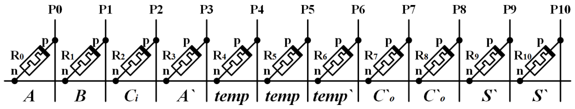

Figure 3: 1-bit RIME full-adder implemented in RRAM crossbar array.

Table 1: 1-Bit Full Adder using NOR and Min$_3$

Cycle #	Logic Operation
0	$R_0 = A, R_1 = B, R_2 = C_i$
1	$R_3 = NOR(R_0) = A'$
2	$\{R_4, R_5\} = Min_3(R_1, R_2, R_3) = temp$
3	$R_6 = NOR(R_5) = temp'$
4	$\{R_7, R_8\} = Min_3(R_0, R_1, R_2) = C_o'$
5	$\{R_9, R_{10}\} = Min_3(R_0, R_6, R_8) = S'$

crossbar array and obtain C_o' and S'. One of RIME architecture benefits is that for parallel computation, C_o' and S' can be shifted to target output RRAMs using NOR gates.This design significantly outperforms MAGIC adders [17] that uses NOR gates to execute 1-bit full adder using 15 RRAMs and 12 clock-cycles.

$$C_o = A \cdot B + B \cdot C_i + A \cdot C_i = Min_3'(A, B, C_i) \tag{5}$$

$$S = A \oplus B \oplus C_i = Min_3'(Min_3'(A', B, C_i), A, Min_3(A, B, C_i)) \tag{6}$$

4.2 Fixed-Point Multiplier

Shift-and-add and Wallace-tree are two standard algorithms for a fixed-point multiplier. The Wallace-tree algorithm has a higher speed but consumes more area [12]. For efficient implementation in a crossbar, a RIME multiplier adopts the Wallace-tree algorithm and splits the RRAM crossbar array for parallel computing. Fig. 4 shows the flow diagram for 4-bit fixed-point multiplier [12]. We have two 4-bit operands, $(a_3a_2a_1a_0)$ and $(b_3b_2b_1b_0)$, and $p_{ij} = a_ib_j$ is the partial product. In the first three stages (**S1** to **S3**), three FAs operate in parallel. In the last three stages (**S4** to **S6**), only one FA operates. We obtain this selective operation using two switches to split a row of RRAMs into three FAs so that the three FAs will not interfere when the switches are off.

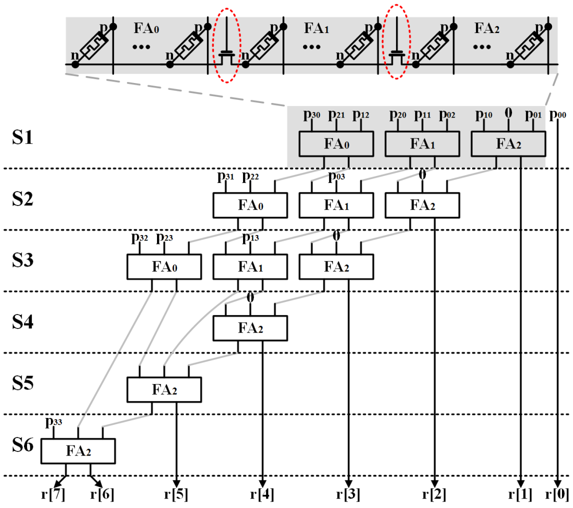
FA₀
FA₁
FA₂
S1
S2
S3
S4
S5
S6
p₃₀ p₂₁ p₁₂ p₂₀ p₁₁ p₀₂ p₁₀ 0 p₀₁ p₀₀
p₃₁ p₂₂ p₀₃ 0
p₃₂ p₂₃ p₁₃ 0
0
p₃₃
r[7] r[6] r[5] r[4] r[3] r[2] r[1] r[0]

Figure 4: Implementation of a 4-bit Wallace-tree multiplier in RIME.

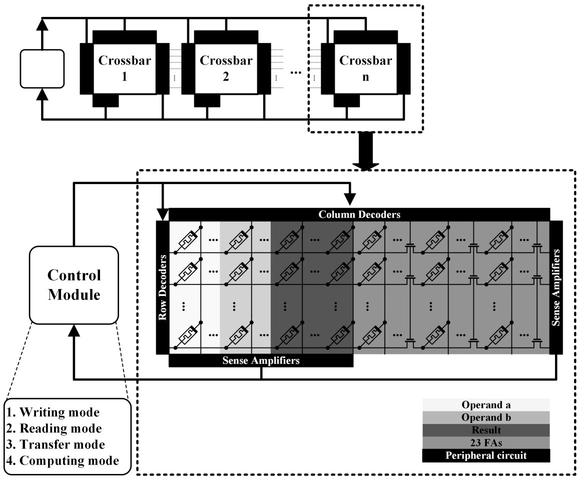

Figure 5: Implementation of a RIME computation unit in RRAM crossbars.

In each stage of the N-bit fixed-point multiplier, we first write the partial products into the target RRAMs of the $N-1$ FAs sequentially using $p_{ij} = NOR(a_i', b_j')$. Then, all the FAs operate in parallel and generate the C_o's and S's. Finally, we write the C_os into the target RRAMs of local FAs in parallel and write the Ss into the target RRAMs of neighboring FAs sequentially for the next stage. Therefore, the latency for the N-bit fixed-point multiplier will be $2 \cdot N^2 + 16 \cdot N - 19$ clock-cycles.

4.3 Floating-Point Multiplier

For a 32-bit floating-point multiplier, 23 FAs are required for the 24-bit fixed-point multiplication in the first 23 stages. In the last 23 stages, only one FA operates, and two idle FAs can be used for the 1-bit XOR and the 8-bit adder. Therefore, the total latency of the 32-bit floating-point multiplier is equal to the latency of the 24-bit fixed-point multiplier. We need $2 \cdot 33 = 66$ RRAMs to store the two operands, $1 + 10 + 48 = 59$ RRAMs to store the result, and $23 \cdot 11 = 253$ RRAMs for the 23 FAs. Therefore, a row of 378 RRAMs is required for the 32-bit floating-point multiplier in RIME.

4.4 Control Modules

The implementation of a complete RIME computation unit is shown in Fig. 5. RIME has n cascaded RRAM crossbar arrays controlled by a centralized control module (CM). This design is suitable for massively parallel computation. For example, a $1K \cdot 378$ RRAM crossbar array supports $1K$ multiplication in parallel. A RIME control module has four modes that provide essential functionalities, as discussed below.

Writing Mode. CM manages the column decoders and the row decoders in the writing mode. It can either write two floating-point numbers into a row within one clock-cycle column-parallel fashion, or write the same floating-point number into rows within multiple clock-cycles in a row-parallel manner.

Reading Mode. In a column-parallel way,CM turns on the switches at the right end of the RRAM crossbar array and controls the column decoders to read the values of the selected column of RRAMs within one clock-cycle. In a row-parallel design, CM controls the

Table 2: Area / μm^2 & energy consumption / pJ for a single 32-bit floating-point multiplier

	APIM [3]	FloatPIM [10]	RIME
RRAM array	2906 & 203	173 & 171	122 & 136
Peripheral Circuit	1326 & 1041	631 & 532	448 & 321
Control Module	8743 & 9082	8425 & 7621	4361 & 1090
Total	12975 & 10326	9229 & 8324	4931 & 1547

row decoders to read the operands and result in the selected row within one clock-cycle.

Transferring Mode. CM transfers the data in a row-parallel way by reading the values of the selected column of RRAMs and writing them into the target column of RRAMs in either local crossbar or the next crossbar.

Computing Mode. CM controls the column decoders and the row decoders to operate the selected floating-point multipliers (or other computation units) in parallel.

Regardless of the number of rows, in RIME designs, the latency for data writing, reading, and transferring depends on the bit-width of the floating-point number. Therefore, RIME can provide massively parallel and scalable computing units in the RRAM crossbar. Moreover, for data-intensive applications, RIME can be useful to provide fundamental functionalities such as matrix multiplication with floating-point precision and efficient communication between neighboring RRAM crossbar arrays.

5 PERFORMANCE EVALUATION

In this section, we discuss detailed implementation and experimental evaluation of the RIME architecture. We first compare the performance of RIME with the state-of-the-art architectures [3, 10, 11] for 32-bit floating-point multiplication in terms of latency, area, and energy consumption. We then demonstrate RIME's scalability to address the *memory-wall* issue. We use VTEAM memristor/RRAM model [14] with the parameters and simulation setup describe in [17] for the experiments. Circuit-level simulations are performed using HSpice with the FreePDK 45nm library. We use Verilog and Synopsys Design Compiler to implement and evaluate the control module proposed for RIME.

5.1 Comparison with MAGIC-based architecture

we choose two recently proposed MAGIC-based multipliers for comparison: APIM [11] and FloatPIM [10]. APIM serializes the full-adder as introduced in [17], and FloatPIM [10] adopts the algorithm proposed in [9] for fixed-point multiplication.

Latency. Fig. 6 compares the latency of N-bit fixed-point multipliers in APIM [11], FloatPIM [10], and RIME. We consider the algorithm to achieve full-precision without skipping the most significant bits. Since RIME executes the 1-bit XOR, the 8-bit adder, and the 24-bit fixed-point multiplier in parallel, the total number of cycles for 32-bit floating-point multiplier is $2 \cdot 24^2 + 16 \cdot 24 - 19 = 1517$ in RIME, while it is $6 + (12 \cdot 8 + 1) + (15 \cdot 24^2 - 11 \cdot 24 - 1) = 8478$ in APIM and $6 + (12 \cdot 8 + 1) + (13 \cdot 24^2 - 14 \cdot 24 - 1) = 7214$ in FloatPIM. Therefore, 32-bit RIME multipliers are 5.6X faster than APIM and 4.8X faster than FloatPIM.

Area. The total area of the 32-bit floating-point multiplier consists of three parts: the RRAM crossbar array, the peripheral circuit, and the control module. The numbers of RRAMs for a single 32-bit floating-point multiplier are 8450, 523, and 378 in APIM, FloatPIM, and RIME. The peripheral circuit in APIM is most complicated because it needs to read out the data from the RRAM crossbar array to the periphery, process it, and write back [9]. FloatPIM and RIME have similar peripheral circuits. RIME requires 22 extra switches for parallel computation but $523 - 378 = 145$ fewer decoders than FloatPIM. Since the control module controls the voltage at the n and p terminals of each RRAM cycle-by-cycle to implement the algorithm, the total numbers of cycles and the total numbers of RRAMs together determine the control module area. According to the results shown in Table II, RIME is 2.6X more area-efficient than APIM, and 1.9X more area-efficient than FloatPIM in implementing a single 32-bit floating-point multiplier.

Energy consumption. We also measure the energy consumption of the RRAM crossbar array, the peripheral circuit, and the control modules for our evaluation. The total number of the underlying logic operations determines the energy consumption of the RRAM crossbar array. The energy consumption of the peripheral circuit is driven primarily by the decoders. Finally, the latency and the complexity of the algorithm determine the energy consumption of the control module. According to the results shown in Table II, to execute a single 32-bit floating-point multiplication, RIME is 6.7X more energy-efficient than APIM and is 5.4X more energy-efficient than FloatPIM.

5.2 Comparison with MIG-based architecture

Bhattacharjee *et al.* [3] proposed a MIG-based architecture (ReVAMP) for in-memory computing. In each computation cycle, the instruction is read from the instruction register to determine the control inputs for the following components:(1) the source select multiplexer, (2) crossbar interconnect, and (3) the write circuit. The write circuits read the value of the target RRAMs, and the crossbar interconnect's output and apply the inputs to the row and column decoder of the RRAM crossbar array.

latency. ReVAMP needs 2338 160-bit instructions to execute a single 32-bit floating-point multiplier. We use an instruction memory with 374.4 Kb and 32-bit aligned access to store the instructions. Therefore, the total number of cycles is $2338 \cdot (160/32) = 11690$, which is 7.7X slower than a RIME multiplier.

Area. The total area of ReVAMP consists of three parts: the RRAM crossbar array, the peripheral circuit, and the instruction memory. The dimensions of the RRAM crossbar array for a single 32-bit floating-point multiplier are $81 \cdot 24 = 1944$, and the area is

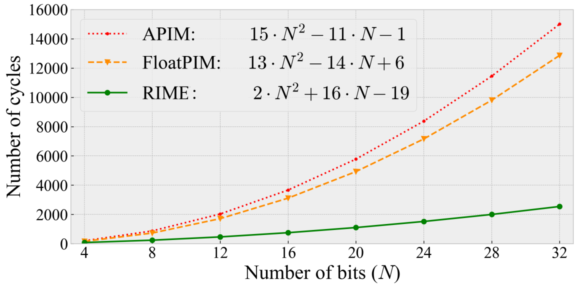

Figure 6: Latency of N-bit fixed-point multiplier.

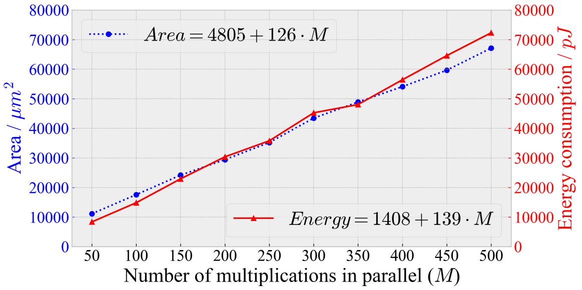

Figure 7: Area and energy consumption of M 32-bit floating-point multipliers in RIME.

668 μm^2. The peripheral circuit of ReVAMP is much more complicated than the MAGIC-based architecture because it has program counters, instruction fetch modules, instruction decode modules, *etc.*, that requires an area of 2405 μm^2. The area of the instruction memory with 374.4 Kb is about 9306 μm^2. Therefore, 32-bit RIME multipliers are 2.5X more area-efficient than ReVAMP.

Energy consumption. To execute a single 32-bit floating-point multiplication, ReVAMP consumes about 39250 pJ, which is 25X less energy-efficient than RIME.

5.3 Comparison with von Neumann architecture

Although the latency (3 ns), area (3523 μm^2), and energy consumption (7 pJ) of a von Neumann ASIC implementation of a single 32-bit floating-point multiplier [2] are less than RIME, it has two fundamental drawbacks. First, a chip can only support a limited number of computing units to execute in parallel due to the area constraint. Second, the data movement from the off-chip memory dramatically increases the latency and energy consumption. For example, the energy consumption of accessing two 32-bit operands from an on-chip memory is about 78 pJ, which is 200 times greater than external DRAM (10 nJ) [5].

In contrast, RIME supports scalability that allows thousands of 32-bit floating-point multipliers to execute in parallel and eliminates data movement for the ASIC implementation. According to the results in Fig. 2, the latency of logic operations is less than 2.5 ns. Thus the total latency for 32-bit multiplication is $1517 \cdot 2.5 = 3793$ ns. Considering the latency of data movement [5], the throughput of RIME with 500 32-bit floating-point multipliers is higher than the ASIC implementation. Fig. 7 gives the relationship between the number of 32-bit floating-point multipliers executing in parallel and the total area and energy consumption. Therefore, for data-parallel application, RIME easily outperforms conventional von Neumann designs.

6 CONCLUSION

In this paper, we proposed RIME, an RRAM-based PIM architecture for precision floating-point operation. Our design extends the basic logic operations in the RRAM crossbar array improves parallelism in computation by using several columns of switches.We also demonstrate the scalability of RIME for massively parallel precision operations. RIME floating-point computations achieve significant improvements in terms of speed, area, and energy consumption compared with state-of-the-art RRAM-based PIM architectures.

ACKNOWLEDGMENT

Zhaojun Lu and Gang Qu were supported in part by AFOSR MURI under award number FA9550-14-1-0351. Zhenglin Liu was partially supported by National Key Research & Development Program (2019YFB1310001) and the National Natural Science Foundation of China (Grant No. 61874047)

REFERENCES

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2014. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *Proceedings of the 51st Annual Design Automation Conference*. ACM, 1–6.

[2] Pasupuleti Anuhya and R Dhanabal. 2018. Asic Implementation of Efficient Floating Point Multiplier. In *2018 4th International Conference on Electrical Energy Systems (ICEES)*. IEEE, 138–141.

[3] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. 2017. ReVAMP: ReRAM based VLIW architecture for in-memory computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 782–787.

[4] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 2010. 'Memristive'switches enable 'stateful'logic operations via material implication. *Nature* 464, 7290 (2010), 873.

[5] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. 2016. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 568–580.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[7] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. 2018. Enabling scientific computing on memristive accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 367–382.

[8] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.

[9] Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky. 2018. Efficient algorithms for in-memory fixed point multiplication using magic. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[10] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 802–815.

[11] Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2017. Ultra-efficient processing in-memory for data intensive applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 6.

[12] G Ganesh Kumar and Subhendu K Sahoo. 2015. Implementation of a high speed multiplier for high-performance and low power applications. In *2015 19th International Symposium on VLSI Design and Test*. IEEE, 1–4.

[13] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[14] Shahar Kvatinsky, Misbah Ramadan, Eby G Friedman, and Avinoam Kolodny. 2015. VTEAM: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 8 (2015), 786–790.

[15] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[16] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 541–552.

[17] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.

[18] Xueyan Wang, Jianlei Yang, Yinglin Zhao, Yingjie Qi, Meichen Liu, Xingzhou Cheng, Xiaotao Jia, Xiaoming Chen, Gang Qu, and Weisheng Zhao. 2020. TCIM: Triangle Counting Acceleration With Processing-In-MRAM Architecture. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[19] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. 2013. Memristive devices for computing. *Nature nanotechnology* 8, 1 (2013), 13.