mMPU – a Real Processing–in–Memory Architecture to Combat the von Neumann Bottleneck

Nishil Talati, Rotem Ben Hur, Nimrod Wald, Ameer Haj Ali, John Reuben, and Shahar Kvatinsky

Abstract Data transfer between processing and memory units in modern computing systems is their main performance and energy-efficiency bottleneck, commonly known as the *von Neumann bottleneck*. Prior research attempts to alleviate the problem by moving the computing units closer to the memory that have had limited success since data transfer is still required. In this chapter, we present mMPU - memristive Memory Processing Unit, which relies on a memristive memory to perform computation using the memory cells and therefore, directly tackles the von Neumann bottleneck. In mMPU, the operation is controlled by a modified controller and peripheral circuit without changing the structure of the memory cells and arrays. As the basic logic element, we present Memristor Aided loGIC (MAGIC), a technique to compute logical functions using memristors within the memory array. We further show how to extend basic MAGIC primitives to execute any arbitrary Boolean function and demonstrate the micro-architecture of the memory. This process is required to enable data computing using MAGIC. Finally, we show how to build the computing system using mMPU, which performs computation using MAGIC to enable a real processing-in-memory machine.

1 Introduction

Contemporary general-purpose computing systems use von Neumann architecture, or an ameliorated version of it, which separates the processing units (or CPUs) from

Nishil Talati · Rotem Ben Hur · Nimrod Wald · Ameer Haj Ali · John Reuben · Shahar Kvatinsky
Technion – Israel Institute of Technology, Haifa 3200003, Israel
e-mail: nishil.t@campus.technion.ac.il
e-mail: rotembenhur@campus.technion.ac.il
e-mail: nimrodw@campus.technion.ac.il
e-mail: ameerh@campus.technion.ac.il
e-mail: johnreuben@technion.ac.il
e-mail: shahar@ee.technion.ac.il

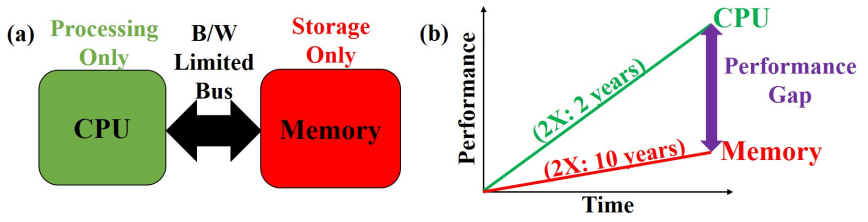


Fig. 1 (a) Abstract model of von Neumann architecture, where two separate units (CPU and memory) are dedicated for data processing and data storage. These elements are connected through a bandwidth(B/W)-limited bus for data transfer [35]. (b) Performance scaling of CPU and memory with respect to time.

the memory system. Due to this separation, data has to travel between the processor and memory through a bandwidth-limited bus, which causes a massive overhead of performance and energy. This is called the von Neumann bottleneck. For years, researchers have been attempting to devise possible replacements for this computation model. Furthermore, with the scaling in the size of the transistor, the performances of both CPUs and memory have scaled; however, the performance of the CPU doubles every two years, while the performance of the memory doubles every ten years, as shown in Fig. 1(b). This is the reason for today's large performance gap between between CPU and memory. As a result, the processor has to wait for multiple clock cycles in order to receive data from the memory, which is known as the *memory wall*.

Some previous approaches to alleviate the von Neumann bottleneck [16, 33, 15, 36] try to move the processing units (PUs) closer to the memory. While doing so, these methods use the DRAM technology for the memory system. Although DRAM is a mature and commercial memory technology, conventional DRAM cells, which are used to store data, are incapable of processing data, and as a consequence, data must still be transferred to closely placed PUs. Hence, these approaches only alleviate the von Neumann bottleneck to a limited extent. An attractive way to completely solve the von Neumann bottleneck is to give computation capabilities directly to the memory cells, thereby eliminating the need for transferring data.

Emerging memory technologies, such as Resistive RAM (RAM), Phase-Change Memory (PCM), Spin-Transfer Torque Magnetoresistive RAM (STT-RAM), *etc.*, are considered to be potential candidates for replacing the conventional memory technologies, *i.e.,* DRAM and Flash. Unlike conventional memory technologies that represent data in terms of presence/absence of charge, emerging memories store the logical value in terms of difference in the value of the cell resistance. Hence, we collectively call them memristors (*i.e.,* memory + resistors) [25]. Apart from data storage, the variable resistance property can also be exploited to employ the memristor cells directly for data processing, which has the potential to resolve the von Neumann bottleneck completely.

A memristor is a two-terminal passive circuit element with variable resistance that can be controlled by applying voltage across it. The resistance of the memristor is confined between minimum and maximum resistance values, commonly represented as a low-resistance state (LRS or R_{ON}) and a high-resistance state (HRS or R_{OFF}). The execution of various logical functions is carried out by assembling memristors with/without other components in different circuit connections and by applying different voltages across them [24, 26, 27, 43, 46, 17, 34, 30, 29].

In this chapter, we present the memristive Memory Processing Unit (mMPU), which directly tackles the von Neumann bottleneck by giving the processing capabilities to the memristive memory elements. We first present Memristor-Aided loGIC (MAGIC), which is a technique to execute logical operations. Specifically, we present MAGIC NOR, which is a technique to perform computation within the memristive memory array, by adding a voltage level to the regular memory operation, and without changing the memristive memory crossbar architecture. The inputs and outputs of the MAGIC gate are the resistance values of the memristors. Hence, it can be used to process data already stored within the memory without reading the inputs, and the output is inherently stored at the desired location inside the memory, obviating the need for a write operation. Furthermore, the MAGIC NOR execution is non-destructive in terms of inputs. Hence, logic execution within the memristive memory enables a true processing-in-memory (PiM) architecture.

We further show how to extend MAGIC execution from a single gate to multiple gates in parallel to the implementation of a Single-Instruction Multiple-Data (SIMD) machine. We describe the microarchitecture of the mMPU that is required to enable true PiM. Specifically, we show the design of an mMPU controller that receives the regular read/write as well as processing commands from the CPU. The write instruction is executed by applying voltage across the memristors through wordlines/bitlines and the read instruction is executed by applying voltage and measuring the current through the memristor using a sense amplifier. Processing instructions are broken down by the mMPU controller into a sequence of MAGIC NOR operations, which can be performed using the memristors. We also present SIMPLE MAGIC, which can synthesize any arbitrary Boolean function into a sequence of MAGIC operations, which can be used within the mMPU controller. Finally, we show the implications of the system integration of mMPU in two different modes – (a) mMPU as an accelerator, and (b) mMPU as a processing unit that is also the system memory. Data-intensive and massively parallel applications, such as deep learning and image processing, which suffer the most from the von Neumann bottleneck, can be efficiently executed on the mMPU.

2 PiM: Prior Art and Its Impact

Early efforts in investigating PiM date back to the ’90s. Some famous proposals include a configurable PiM chip that can operate as a conventional memory or as a Single Instruction Multiple Data (SIMD) processor for data processing [16]. The

authors of Active pages [33] have proposed placing the CPU and configurable logic elements next to the DRAM subarrays to speed up the processing. In Computational RAM [15], the sense amplifiers of the random access memory are connected directly to the SIMD pipelines. The Berkeley IRAM project [36, 35] advocated widening the bandwidth between CPU and memory by designing them on the same die.

Early adaptation of PiM failed to gain widespread adoption because of four major challenges [6]. The first challenge was inadequate implementation of technology. Although prior proposals tried to integrate the memory and CPU on the same die, the incompatible fabrication technologies of DRAM and CPU made it difficult to incorporate these approaches in practical computing systems. The second was the processor architecture that can use the high bandwidth enabled by proximity to memory. Early PiM research required custom architectures, requiring considerable design efforts and significant advancement in the developer community. The third challenge was the development of interfaces that allowed PiM computing units as well as external processing units to access memory. Early efforts required the design and adoption of custom memory interfaces. The fourth challenge was the programming models. Early approaches had to develop the programming abstractions from the bottom up.

Today, the aforementioned challenges are being overcome by modern age with the advancement in technologies and methodology involved in building computers. For example, thhe first challenge has be overcome by the emergence of 3D die stacking, enabling heterogeneous integration of logic and memory, and emerging memory technologies, facilitating 3D fabrication of memory arrays on top of CMOS substrates [1]. The evolution of various other processing platforms, *e.g.*, GPGPUs, custom accelerators *etc.*, have solved the second problem by efficiently utilizing the high bandwidth offered by the memory within the thermal constraints of the memory modules [14]. Recent die-stacked memory interface standards (such as High Bandwidth Memory [2]) and off-chip memory interfaces that expose load-store semantics (such as Hybrid Memory Cube [3]) meet nearly all the memory interface requirements of PiM, which surmounts the third challenge. Recent frameworks such as Heterogeneous System Architecture [7] and the associated software tools for accelerators have addressed the fourth challenge to widespread adoptation of PiM.

Although the advancement in technologies solve most of the aforementioned problems, the current state-of-the-art technologies and future PiM proposals should address the new set of issues such as workload heterogeneity (different algorithms present various memory layouts, access patterns, and involve computations with different degrees of parallelism and complexity) and fabrication challenges in memory that can enable PiM.

One current state-of-the-art PiM concept is Micron's Automata Processor (AP) [13], as shown in Fig. 2. The AP natively implements the non-deterministic finite automata (NFA) paradigm in hardware. Thus, the AP is an accelerator designed specifically for symbolic pattern matching. In this architecture, the input symbol is provided to multiple memory arrays by decoding it, instead of the row address. Automata operations are invoked through a routing matrix structure exploiting the

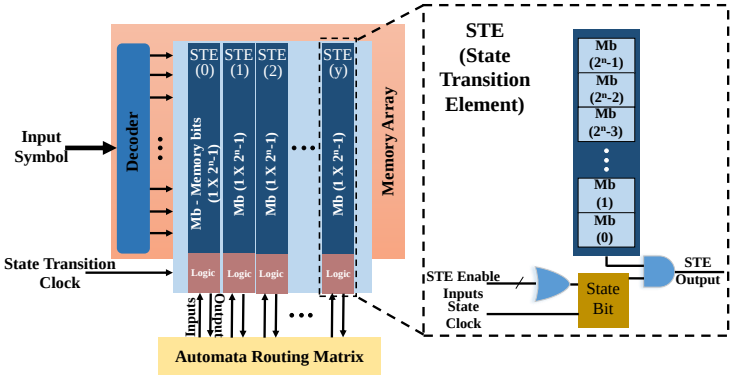


Fig. 2 Modern PiM architecture – Micron's Automata Processor (AP) [13], which exploits the inherent bit-parallelism in DRAM for symbolic pattern matching by performing multiple operations on a single data and by that reducing the number of memory accesses.

inherent bit-parallelism of traditional DRAM, enabling Multiple Instruction Single Data (MISD) architecture. This architecture provides the flexibility to program independent automata on a single silicon device [40]. Apart from the AP, several other recent proposals for PiM enable the transition from DRAM to resistance-based emerging non-volatile memory technologies (NVRAM). These approaches include the accelerators for enhancing artificial neural networks [8, 11], DDR3-compatible interface with dual in-line memory modules (DIMM), capable of performing content addressable searches [18], associative computing [19, 45], *etc*.

All of the previous approaches for addressing the von Neumann bottleneck using PiM have relied on reducing the distance between the processing and the conventional memory system, *i.e.,* DRAM. Although DRAM has been exploited to its best capabilities, these approaches still suffer from a fundamental problem – the need to transfer data between the CPU and the memory. Because DRAM cells are incapable of performing logical operations, systems with DRAM as a memory require a separate resource to perform computation. Emerging memristive technologies, such as Resistive Random Access Memory (RRAM or ReRAM) [41, 42, 28], enable a new approach, where the computation of logical functions is done directly using the memory cells, without any need to instantiate additional CMOS blocks for processing. In this chapter, the von Neumann bottleneck is solved by giving computational capabilities directly to the memristive memory cells. Thus, the proposed approach is fundamentally different than all the previously proposed techniques in PiM and tackles the data movement issue directly.

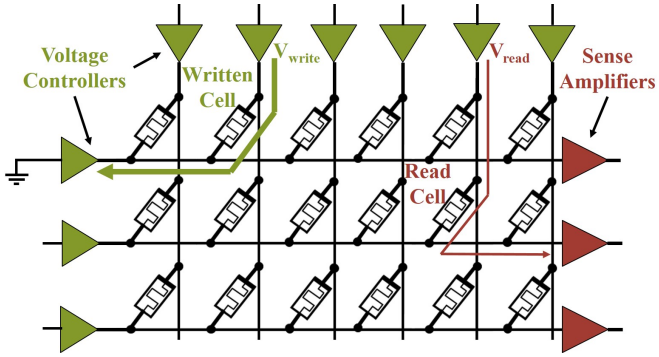


Fig. 3 Crossbar structure of memristive memory array. Voltage controllers and sense amplifiers are used to perform read, write, and logic operations. Example of a write operation by applying V_{write} across the target memristors, and a read operation by applying V_{read} across the memristor and measuring the current using a sense amplifier. Note that reads and write operations are performed in time-multiplexed fashion.

3 Computation with Memristors

In this section, we first describe the operation of the memristor crossbar array as memory. Then, we present Memristor Aided loGIC (MAGIC), a logic family that enables the performing of logical operations within the memristive memory. We further show how to integrate the MAGIC circuit within the memristive memory array without requiring major modifications in the crossbar structure and techniques to perform vector operations using MAGIC.

3.1 Memristive Memory

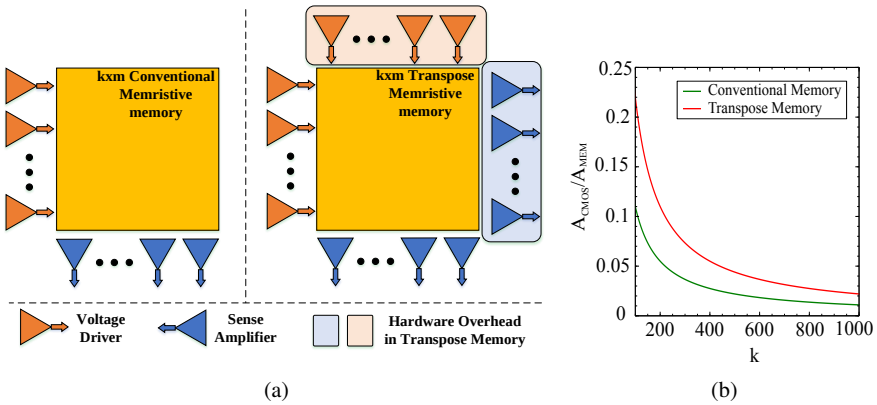
The memristor stores the logical value in terms of its resistance, in contrast to conventional memories, which use a charge to represent data. This resistance is controlled by applying voltage across the memristor. Memristors can be fabricated between two metals, which act as the top and the bottom electrodes of a switching dielectric material. Hence, memristors can be fabricated in the metal layers as part of a standard CMOS Back End of Line (BEOL) process. Memristive memory generally utilizes a crossbar structure, which enables an extremely dense memory array with memory cell area of $4F^2$, where F is the technology feature size. Fig. 3 shows one such design of a memristive memory crossbar array. Voltage drivers, row/column decoders, and sense amplifiers are used as a part of the peripheral circuit to support write and read operations, similar to other memory technologies. To perform a write operation, a write voltage V_{write}, higher than the threshold voltage (v_{on} and v_{off}, which switches the memristor to LRS and HRS, respectively), is applied across the target memristor through the wordlines and bitlines. For a memristor with asym-

metric switching characteristics (*i.e.,* $v_{on} \neq v_{off}$), two different write voltages are applied for writing logic 1 (*i.e.,* V_{SET}) and 0 (*i.e.,* V_{RESET}). Since during the write operation, the voltage is applied through wordlines and bitlines, even the memristors adjacent to the target memristors are partially influenced by this voltage, which may disturb the state of the unselected memristor; this is known as the write disturb problem [4]. Half-select voltages (typically $V_{write}/2$ or $V_{write}/3$ [5]) are applied to isolate the non-target memristors.

Read operations are performed by applying a voltage V_{read}, with a magnitude lower than the threshold voltage for switching, and measuring the current passing through the device using a sense amplifier (SA), as shown in Fig. 3. A primary challenge for the read operation for memristive memory is the sneak path current phenomenon [47, 9, 31, 38], which is due to the resistive nature of the memory cells: the read voltage also creates additional current paths, different than the desired path, and this additional current flow adds resistance in parallel to the selected memristor, which depends on the stored data in the unselected memristors. There are several ways to overcome this challenge [47, 9, 20], including modification of the memory cell structure (*i.e.,* using a diode/transistor/selector in series with the memristor) and using different biasing schemes for the unselected lines (*i.e.,* ground/half-select biasing schemes).

Although the memristive memory crossbar structure is symmetrical, accessing memory cells in a conventional memristive memory array is possible only from one direction. Access from the other direction is blocked since only specific voltages can be applied in each row/column, and the decoding and sensing circuits are connected to a single edge of the array. To enable the access to memory cells from all sides, voltage controllers and sense amplifiers can be added on both sides of the memristive memory crossbar, constituting a memory called *transpose memory* [39]. Additional peripheral circuitry would provide more flexibility to the memory array and would capabilities to the memory system. Fig. 4(a) illustrates the difference in peripheral circuitry between $k \times m$ conventional and transpose memory crossbars. Fig. 4(b) shows the comparison of the ratio of total area utilized at CMOS and memristive layer for different values of array sizes (*i.e.,* $k \times k$). The comparison shows that the ratio is almost equal (which implies the area utilization) for large array sizes (*i.e.,* $k \geq 100$). Note that this is a general comparison irrespective of the memristor technology used, *i.e.,* without considering the maximum allowed array size.

All operations (read, write, and half-selecting cells) are performed in transpose memory by application of similar voltages as in conventional memory, with the added freedom of applying these voltages from both horizontal and vertical directions. Furthermore, as described later in Section 3.2, transpose memory offers the additional feature of transposing the logic execution in the columns of the array, whereas in conventional memory, this is only possible over a memory row.



(a) (b)

Fig. 4 (a) Comparison of additional supporting CMOS circuitry to facilitate logic implementation at nanocrossbar layer for $k \times m$ conventional and transpose memories, and (b) Ratio between CMOS area (A_{CMOS}) and memristor area (A_{MEM}) for different array sizes (*i..e.,* different k for $k \times k$ arrays) for conventional and transpose memory crossbars. The area utilization at nanocrossbar layer improves for larger arrays.

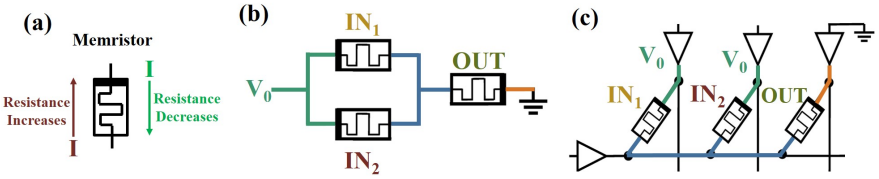


Fig. 5 (a) Desired switching characteristic of a memristor, schematic of a (b) two-input MAGIC NOR gate and a (c) MAGIC NOR gate within a memristive memory array. IN_1 and IN_2 are the input memristors and OUT is the output memristor. A single voltage V_0 is applied to perform the NOR operation [24].

3.2 MAGIC – Memristor Aided loGIC

MAGIC is a stateful logic family [37], compatible for computation within the memristive memory [24]. In MAGIC, n input memristors and a single output memristor are used to execute n-input Boolean functions (*e.g.,* NOR, NAND, OR, AND, and NOT). Some MAGIC gates, such as NOR and NOT, can be implemented within the memristive memory crossbar array, not requiring any modification of the crossbar or the memory cells. An additional voltage level is required, apart from read and write voltages, in order to support the MAGIC execution within the memory. Fig. 5(b) shows the schematic of a two-input MAGIC NOR gate, where IN_1 and IN_2 are the inputs of the NOR gate, and OUT is the output. The input memristors and the output memristor are always connected in the reverse polarity as shown in Fig. 5(b).

To execute the MAGIC NOR operation, the output memristor is initialized to R_{ON}. A voltage V_0, higher than the threshold voltage for switching, is applied to the

input memristors, and the output memristor is grounded from the other terminal as shown in Fig. 5(c). Due to the resistive nature of memristors, the voltage is divided between the input and output memristors. Consequently, the output switches from R_{ON} to R_{OFF}, only if both the inputs are logic 1, *i.e.,* the voltage across the output memristor is high. The value of the MAGIC execution voltage V_0 has to be within a certain interval to ensure that the MAGIC gate works as expected. The value of V_0 should be high enough to switch the output memristor during the MAGIC execution, when all the inputs are logic 1, which sets the lower bound on V_0. Furthermore, the value of V_0 should be sufficiently low to prevent switching of the input memristors. This sets the higher bound on V_0. Hence, the constraints on an n-input MAGIC NOR gate execution voltage V_0 should be

$$\frac{v_{off}}{R_{ON}} \cdot \left\{ R_{ON} + \left(\frac{R_{OFF}}{n-1} \right) || R_{ON} \right\} < V_0, \tag{1}$$

$$V_0 < min\left[v_{off} \cdot \left(1 + \frac{R_{OFF}}{nR_{ON}} \right), |v_{on}| \cdot \left(1 + \frac{nR_{ON}}{R_{OFF}} \right) \right], \tag{2}$$

which ensures that the gate executes a NOR operation, and the input data is never destroyed. Note that the aforementioned constraint is constructed neglecting the parasitic effects of wires. In a more realistic scenario, where a unit interconnect resistance of r_w is considered between two adjacent wordlines/bitlines, (1) becomes

$$\frac{v_{off}}{R_{ON}} \cdot \left\{ R'_{ON} + \left(\frac{R'_{OFF}}{n-1} \right) || R'_{ON} \right\} < V_0, \tag{3}$$

$$V_0 < min\left[v_{off} \cdot \frac{\left(\frac{R'_{OFF}}{n} + R'_{ON} \right)}{R_{ON}}, |v_{on}| \cdot \frac{(R'_{OFF} + nR'_{ON})}{R_{OFF}} \right]. \tag{4}$$

where R'_{ON} and R'_{OFF} denote the effective resistances and are equal, respectively, to $(R_{ON} + iR_w)$ and $(R_{OFF} + iR_w)$. Note that these expressions are similar to (1, 2).

It is possible to further extend the execution of a MAGIC NOR operation from a memory row to a memory column in the transpose memory [39]. Fig. 6(a) shows the schematic of a MAGIC NOR gate on a memory column. In this case, the MAGIC execution voltage (V_0) is applied to the output memristor, and the parallel combination of the input memristors is grounded from the side, which is not connected to the output memristor. This is the only difference between them, and the range of V_0 is the same as in the previous case of NOR logic execution, which is non-destructive in terms of its inputs. The steps involved in MAGIC execution over both rows and columns are summarized in Table 1.

The parallelism of MAGIC within crossbar arrays is limited; two independent MAGIC NOR gates cannot be executed simultaneously in the same row, as illustrated in Fig. 6(b). If V_0 is applied to two different sets of input memristors ($\{IN_1^1, IN_2^1\}$ and $\{IN_1^2, IN_2^2\}$), and output memristors ($\{OUT^1, OUT^2\}$) are grounded, the equivalent circuit becomes as shown in Fig. 6(c). Due to the con-

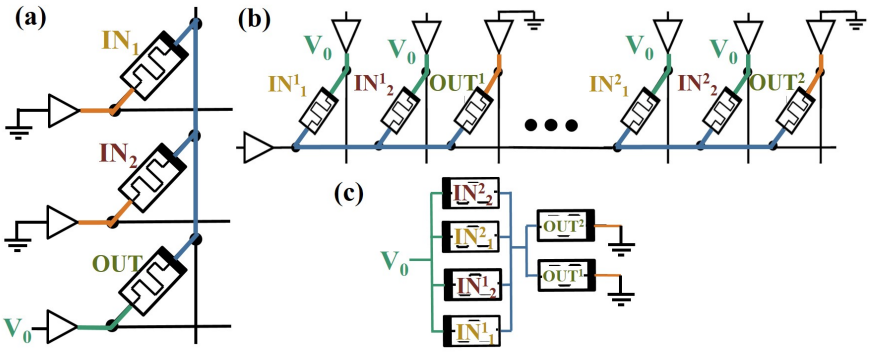


Fig. 6 (a) MAGIC NOR execution over a memristive memory column. (b) Attempt to execute two distinct MAGIC NOR operations over the same row simultaneously, and (c) its equivalent circuit schematic, demonstrating the wrong operation.

Step #	Operation	Application of Voltages
1	Initialize output memristor at R_{ON}	$out \leftarrow V_{WRITE}$
2	Apply V_0 to the input (output) memristor(s), and ground to the output (input) memristor(s) for execution over a memory row (column)	$in1, in2, ... \leftarrow V_0$ (GND) and $out \leftarrow GND(V_0)$

Table 1 Steps involved in MAGIC NOR execution across a row (column) of a memristive memory.

nection pattern between the input and the output memristors, two output memristors are actually connected in parallel, leaving the equivalent resistance at the output as $R_{ON}/2$, rather than R_{ON}, resulting in a wrong operation.

3.3 Vector Operation using MAGIC

While the MAGIC execution voltages are applied to wordlines or bitlines (for transpose MAGIC operation), the influence of these voltages is spread throughout the whole data line, and not limited to the particular memory row/column. As shown in Fig. 7, if V_0 is applied to the first two columns, and the third column is grounded, all the memristors situated in the first column perform the MAGIC NOR operation with its neighboring cell on the second column and produce the output on the corresponding cell in the third column. This situation can be exploited to perform vector operations [39]. Note that the latency to perform this vector operation is independent of the size of the vector, as long as the entire vector can fit inside an array, and the voltage drivers can provide the required currents for proper behavior.

If the vector operation is restricted to few rows in the array, it is possible to isolate a particular row from the MAGIC execution. This is achieved using isolation voltages, which are similar to half-select voltages for write operations. While in write operations, half of the voltage is applied (*i.e.*, $V_{write}/2$) to prevent the unwanted

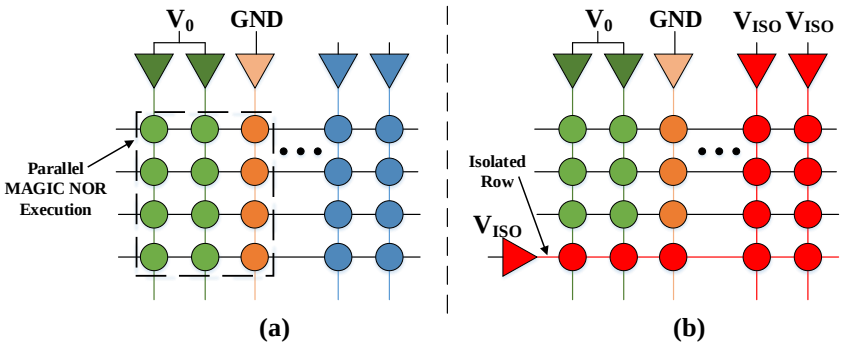


Fig. 7 (a) Intrinsic parallel MAGIC NOR execution over for data present in all the rows, and (b) isolation of a row using an isolation voltage applied to that row (*i.e.,* V_{ISO}) to prevent execution of MAGIC NOR.

logic operations, applying $V_0/2$ in a MAGIC NOR operation would disturb the input memristors. Hence, we propose ranges of voltages that can be applied to isolate rows/columns, thus preventing them from executing a MAGIC NOR operation as shown in Fig. 7(b). When a MAGIC operation is performed over the rows, V_{ISO} must fulfill

$$0 < |V_{ISO}| < |v_{off}| < \frac{V_0}{2}, \tag{5}$$

and when a MAGIC operation is performed over columns, V_{ISO} should carry out

$$V_0 - |v_{off}| < |V_{ISO}| < |v_{on}|, \tag{6}$$

where v_{on} and v_{off} are the SET and RESET switching thresholds for the memristor, and V_0 is the MAGIC execution voltage. The voltage levels that should be supported by the peripheral circuit in order to perform conventional memory operations and execute MAGIC logic within the memristive memory are listed in Table 2. Fig. 8 shows the design of the peripheral circuit needed to support these operations and the voltage levels inside the memristive memory. Analog multiplexers, as shown in Fig. 8(b), can be designed to assert different voltage levels to support write and MAGIC operations, and a sense amplifier can be used to perform read operations.

3.3.1 Limitations on the Performance of Vector Operations using MAGIC

While MAGIC NOR operations can be performed in every row (column) in parallel, the length of the SIMD that can be implemented within a memristive crossbar is restricted by the size of the array. The size of the array is further dependent on various circuit and technological parameters. The circuit parameters crucial for deciding

Operation	Voltages Applied
Write	$V_{write} = V_{SET}$ and V_{RESET} for writing logic 1 and 0
Read	V_{read}
Ground	GND
Half-select	$V_{write}/2$
MAGIC Execution	V_0
MAGIC Isolation	V_{ISO}

Table 2 Voltage levels supported by the peripheral circuit to perform conventional memory operations and execute MAGIC NOR gates within the memory.

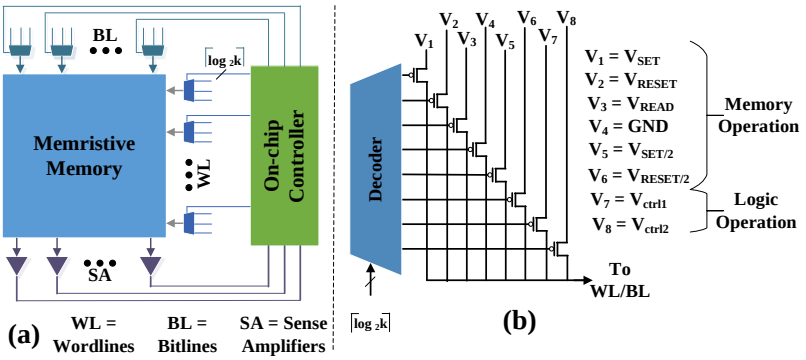


Fig. 8 (a) Peripheral circuit around memory. (b) Structure of an analog mux.

the size of the array are the MAGIC execution voltage V_0, and the technological parameters include memristive properties (R_{ON}, R_{OFF}, v_{on}, and v_{off}) and parasitic effects of the CMOS process (*i.e.,* interconnect resistance and capacitance). To be able to support MAGIC NOR operations in all the rows (columns) of the crossbar, the MAGIC execution must be supported in the worst case configuration at the row (column) farthest from the voltage drivers, since the voltage across it would be the lowest. Worst case configuration occurs when all the resistance values in the array are R_{ON} and it is required to execute MAGIC over all the rows (columns). This is because lower memristor resistance would require higher current to be drawn from the drivers, and as a consequence, the IR drop across the parasitic resistances would be high, lowering the voltage drop across the farthest memristor. Hence, given fixed V_0 and other technological parameters, a finite number of MAGIC NOR operations will be supported, which will limit the size of the memristive crossbar.

Furthermore, to support the execution of multiple MAGIC NOR operations in parallel, the voltage drivers would require a large current inside the array, which has two consequences. First, to supply a current large enough to support several MAGIC NOR operations, the drivers must also be large, which will increase the area of the chip. Second, since V_0 has a higher voltage level than write voltage, performing many MAGIC NOR operations in parallel will increase the energy consumption.

Hence, while the goal is parallel execution of MAGIC NOR gates, this parallelism will be limited by the area and power budget of the chip from the point–of–view of the peripheral circuit.

4 mMPU Microarchitecture

The primary difference between a memristive memory and an mMPU is their control mechanism. In addition to supporting regular memory operations (*i.e.,* read and write), the mMPU controller also handles logic operations within the memory, and in practice its implementation determines the performance of the mMPU. We now present the modifications that must be made to the on-chip controller of the mMPU [21]. We further show SIMPLE MAGIC [23], an automatic synthesis tool we have developed that receives any arbitrary Boolean function as input and proposes an optimal (in terms of latency, energy, or area) sequence of MAGIC NOR gates to implement that function using the mMPU.

4.1 mMPU Controller

The mMPU controller is responsible for generating the control signals for the memory to perform read, write, and logical operations within the mMPU. As shown in Fig. 9, the CPU sends the instruction to the mMPU controller. This instruction is received by a CPU-in block, where it is decoded. Then, this instruction is broadcast to the arithmetic, read, and write blocks, and a block suitable for the instruction type is selected using the memory out mux. For example, if the CPU sends an arithmetic instruction, the control sequence from the arithmetic block would be selected to be sent to the memristive memory.

Whereas reads and writes in the mMPU are performed in a conventional way [21], across the memristor over the target wordlines and bitlines, executing logical instructions is more complicated since they require a sequence of logical steps. The arithmetic block is a sophisticated finite state machine, the role of which is to efficiently break the instruction down into a series of MAGIC operations, and to select the memristive cells to perform the operations within the memory array. For example, the CPU sends an instruction to add two numbers (*i.e.,* ADD) within the memory. The instruction is received by the CPU-in block, which identifies the instruction as ADD and generates the memory out mux select signal. Then, the instruction is sent to the arithmetic block, where an appropriate, pre-synthesized execution sequence is selected for this instruction. This execution sequence is then executed on the memristive memory. The mMPU controller pipelines this operational sequence to the memory, changing the applied voltages on each memory clock cycle. Efficient pipelining maximizes the processing efficiency in terms of speed and energy.

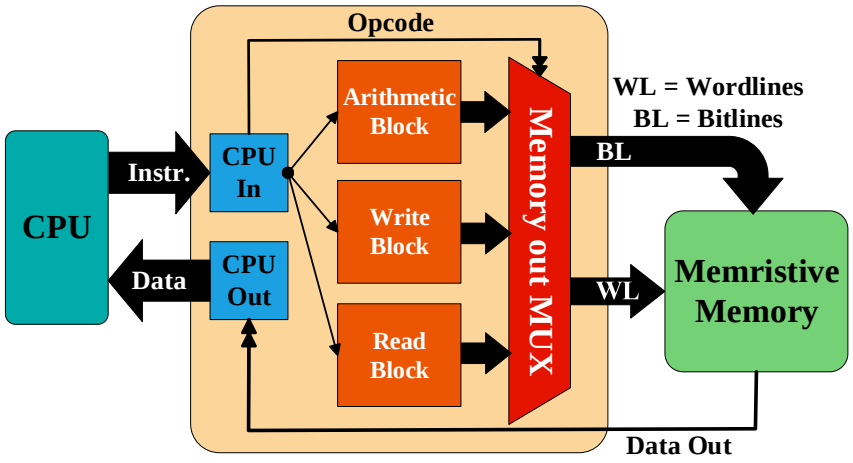


Fig. 9 Detailed block diagram of the mMPU controller, where an arithmetic block is added to support computation within the memristive memory [21].

To optimize the throughput of the arithmetic instruction execution, different considerations should be taken into account [22], as detailed below.

4.1.1 Algorithms for Processing-in-Memory

To enable efficient data processing using the mMPU, novel algorithms (*e.g.,* algorithms based solely on MAGIC NOR operations) need to be developed. Exploiting the parallelism offered by the mMPU as described in Section 3 is essential to optimize these algorithms in terms of energy, performance, and area. For example, multiplying K-binary matrices, each of which is of size $M \times N$, requires $5NK - 5K + 2M + 1$ steps when optimizing the algorithm for MAGIC NOR execution within the mMPU [21]. This algorithm has a quadratic time complexity of $\mathcal{O}(NK)$, while in standard von Neumann architecture, a cubic time complexity of $\mathcal{O}(NKM)$ is required. This instance exemplifies the potential performance benefits of processing data within the memory. Hence, design of a correct algorithm is the key for efficient processing using the mMPU.

4.1.2 Processing Area

Logic execution within the mMPU requires utilization of memory cells for computation. This utilization must maintain the integrity of the data stored in the memristive memory. For example, while calculating complex Boolean functions, several MAGIC NOR/NOT operations must be performed, and the intermediate values of

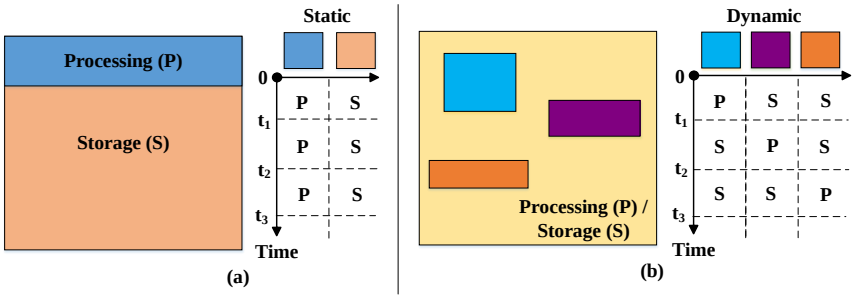


Fig. 10 (a) Static processing area, where a portion of the memory space is dedicated for processing (in blue), (b) dynamic processing area, where a portion of memory space, variable in location and size, is allocated for processing or storage (in blue, purple, and orange), and allocation of processing (P) and storage (S) areas with respect to time. The tables next to the figures denote the time multiplexing of processing and storage space for both the schemes. Symbols S and P mean storage and processing, respectively.

these operations are also stored within the memristors, which we call functional memristors [22, 39]. The functional memristors must be separated from the memristors where valid data is stored, and the Operating System (OS) has to make sure that no data is destroyed. One straightforward solution to this problem is to allocate a fixed amount of memory space for processing; this is known as the *static processing area* [21] as shown in Fig. 10(a). A more complicated solution is to dynamically allocate the processing area based on the availability of the memory cells and required amount of functional memory space for processing; this is known as the *dynamic processing area*, as shown in Fig. 10(b).

Fig. 10 shows the difference between static and dynamic processing areas. It also shows how the dynamic technique time-multiplexes the different portions of the available memory for processing and storage, while the static technique uses the dedicated areas for processing and storage. While the dynamic processing area scheme efficiently allocates the memory space without any wastage, it requires a costly memory management. In contrast, the static processing area scheme does not require any memory management since the area is committed at design time, but it suffers from lower memory utilization.

4.2 Automatic Logic Synthesis using SIMPLE MAGIC

The state machine of the mMPU controller is designed to execute the sequence of required NOR and NOT operations within mMPU. Wisely exploiting the parallelism capabilities described in Section 3 to execute numerous NOR operations simultaneously on different rows or columns may significantly improve the computation performance. To maximize the efficiency of the computations performed

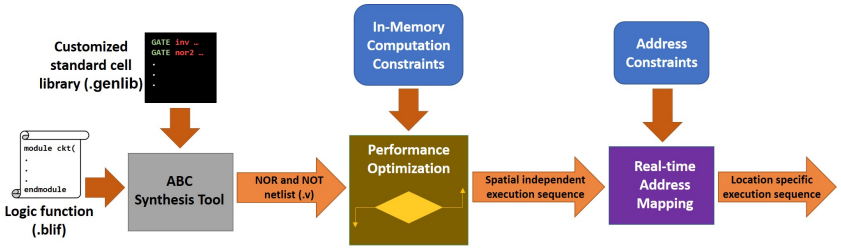


Fig. 11 The desired logic function is synthesized using ABC [32] for NOR and NOT gates and then optimized specifically for MAGIC within memory, generating a general mapping and a sequence of operations. The general execution is mapped to specific cells in real-time, based on the temporary state of the mMPU and its available cells [23].

by the mMPU, the controller has to be designed to perform an optimized NOR and NOT sequence that is optimized in terms of either latency, energy, area, or a combination of the three. The optimized algorithm is determined automatically using SIMPLE MAGIC [23], a tool we recently developed. SIMPLE receives any logic function, and performs the following flow, as illustrated in Fig. 11:

1. The function is converted into a netlist of NOR and NOT gates using a modified ABC synthesis tool [32].
2. The netlist is mapped into a memristive memory, by solving an optimization problem, using the z3 SMT solver [12]. Thus, for every gate j, the variables of the problem are:

   - The coordinated wordline and bitline of the inputs A_j, B_j and output E_j of the gate:
   $$\Big(\{R_{A_j}, C_{A_j}\}, \{R_{B_j}, C_{B_j}\}, \{R_{E_j}, C_{E_j}\}\Big).$$
   - The number of the clock cycle in which the gate is executed is T_j.

   The mapping is done while taking into account the following constraints of in-memory processing:

   - Inputs and outputs of each MAGIC gate have to be mapped to a legal memory cell (when the size of the memory is $ROW_{num} \times COL_{num}$):
   $$\forall x_j \in \{A_j, B_j, E_j\} : (0 < C_{x_j} \leq Col_{num}) \cap (0 < R_{x_i} \leq Row_{num}). \quad (7)$$
   - The execution time of each gate is positive:
   $$\forall gate\ j : T_j > 0. \quad (8)$$
   - Outputs of different gates have to be mapped to different memory cells:
   $$\forall E_k, E_j : (C_{E_j} \neq C_{E_k}) \cup (R_{E_j} \neq R_{E_k}). \quad (9)$$

- Inputs and output of each MAGIC NOR gate have to be mapped to the same column or the same row (as described in Section 3.2):

$$\forall gate\ j : [(C_{A_j} = C_{B_j} = C_{E_j}) \cap (R_{A_j} \neq R_{B_j} \neq R_{E_j})] \cup [(C_{A_j} \neq C_{B_j} \neq C_{E_j}) \cap (R_{A_j} = R_{B_j} = R_{E_j})]. \tag{10}$$

- To perform several MAGIC gates in parallel, the inputs and outputs have to be aligned (as shown in Fig. 7):

$$\begin{gathered}\forall gate\ j,k : T_j \neq T_k \cup \\ \{\{[(C_{A_j} = C_{A_k} \cap C_{B_j} = C_{B_k}) \cup (C_{A_j} = C_{B_k} \cap C_{B_j} = C_{A_k})] \cap (C_{E_j} = C_{E_k})\} \cap \\ (R_{A_j} = R_{B_j} = R_{E_j} \cap R_{A_k} = R_{B_k} = R_{E_k})\} \cup \\ \{\{[(R_{A_j} = R_{A_k} \cap R_{B_j} = R_{B_k}) \cup (R_{A_j} = R_{B_k} \cap R_{B_j} = R_{A_k})] \cap (R_{E_j} = R_{E_k})\} \cap \\ (C_{A_j} = C_{B_j} = C_{E_j} \cap C_{A_k} = C_{B_k} = C_{E_k})\}.\end{gathered} \tag{11}$$

- A MAGIC gate can be executed only when its inputs were produced previously and each input has to be located in the same memory cell as the output of the gate connected to it.

$$\begin{gathered}\forall E_h, x_j \in \{A_j, B_j\}\ that\ are\ connected : \\ [(C_{E_h} = C_{x_j}) \cap (R_{E_h} = R_{x_j})] \cap (T_h < T_j).\end{gathered} \tag{12}$$

The optimization problem can be solved for minimizing the latency, area, energy, or a combination of them. For example, the optimization function for minimizing latency is:

$$Latency_{best\_mapping} = min\{\max_j T_j\},\ where\ 0 < j \leq \#gates. \tag{13}$$

3. The mapping is reshuffled in real-time, according to the occupancy of the memory at the moment the computation is done.

Automation of the process promises optimal results and reduces the time required to design the mMPU controller. The first two steps are performed to design the state machine of the arithmetic block of the mMPU controller, and the third step is performed by the mMPU controller during run-time. Fig. 12 presents the performance speedup of SIMPLE of 1.9× on average as compared to a NOT and NOR netlist prior to optimization with SIMPLE (also before synthesizing the netlist with ABC). Additionally, SIMPLE yields performance speedup of 1.94× compared to previous work [10]. Two major factors contribute to the performance benefit of SIMPLE. SIMPLE tries to exploit the intrinsic parallelism offered by MAGIC NOR execution within the memristive memory. Furthermore, while exploiting this parallelism, SIMPLE rearranges the netlist in such a way that the copy operations of data within the array are not required between the successive steps of execution. Current and future improvements of SIMPLE may further increase performance.

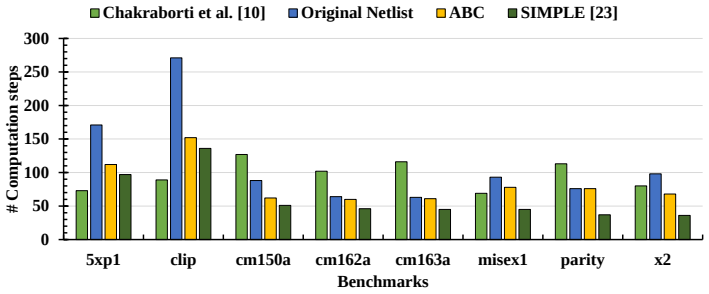


Fig. 12 Performance comparison of SIMPLE [23] (dark green) with other synthesis approaches, which include Chakraborti *et al.* [10] (green), the original netlist without synthesis (blue), and the netlist synthesized with ABC (yellow).

5 System Design using mMPU

Introducing an mMPU to a computing machine requires that new aspects of system design be considered. First, the appropriate computation model for exploiting mMPU capabilities must be chosen. Using the mMPU as a standalone accelerator, as shown in Fig. 13(a), allows us to exploit the existing knowledge about accelerator operation. In this usage model, the mMPU address space is separated from that of the main memory. Any data that is to be processed within the mMPU needs to be transferred (via direct R/W operations or DMA transactions) from its original location in the main memory to a dedicated processing location within the mMPU. Once the processing is completed, the result needs to be copied back to a location reserved for it in the main memory for later use.

Another optional computation model is to incorporate the mMPU address space as a part of the (or as the entire) main memory address space, as shown in Fig. 13(b). Combined with careful data allocation, this usage model may avoid most of the data transfers and further speedup computation. This enhancement, however, comes at the cost of more complicated control (discussed later in this section), and with the need to reserve parts of the available memory space (otherwise used to store data) for intermediate results of the computation.

Data coherency also must be addressed. Using the mMPU allows data to be modified in its location within the main memory and without modifying any instances of the same data down the memory hierarchy (*i.e.,* in caches). Therefore, maintaining data coherency requires an added capability to invalidate data in caches if the data was changed by the mMPU. When the mMPU is used as an accelerator, data that is processed needs to be locked against changes (by using an atomic operation or some other means) to avoid it being changed while the mMPU is processing. The concepts of data redundancy and memory reliability also need to be addressed in order for a system containing the mMPU to be seamlessly compatible with existing SW and data correction mechanisms.

A programming model must be suited for each usage model for efficient utilization of the mMPU. Because the rest of the system should be as oblivious to the

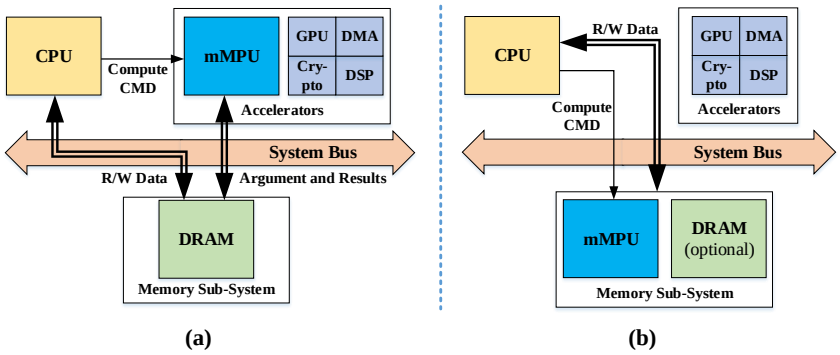


Fig. 13 Illustration of the possible mMPU usage models. When using the mMPU as (a) an accelerator, data to be processed is copied from the main memory to the mMPU and computing commands are sent from the CPU. When using the mMPU as (b) a part of the main memory, the data meant for processing is stored beforehand in the mMPU address space, allowing the commencement of processing with a single command from the CPU.

mMPU as possible, standard interfaces should be adopted, and the mMPU should be designed so that minimal changes to the rest of the system are required. Furthermore, apart from using mMPU for data processing, it can also be selectively used as the system memory, making it compatible with the von Neumann computing model. Rather than being burdened with challenging optimization tasks as in the case of conventional architectures, for the general use case the programmer only has to determine the desired operation, the addresses of the inputs and outputs, and the size of the inputs. Such an accelerator is addressed with software support, *i.e.,* additional libraries with specific functions that the mMPU will support, such as CUDA [44] in NVIDIA GPUs. In this case the CPU will offload the code to the mMPU directly without the need to modify the ISA or the current conventional systems.

Two approaches are proposed for utilizing the mMPU as a memory capable of computing. The first requires extending the ISA with additional commands that the mMPU supports. These commands will be successively dispatched by the CPU to the mMPU so that computation tasks are performed on specified locations in the memory (*i.e.,* addresses). In the second approach, the mMPU will have a reserved address, which when written to, will initiate the equivalent command. Thus, an instruction for in-memory computing contains a write operation to a reserved address that is mapped to a dedicated register within the mMPU controller. The instruction must contain all the relevant information for execution, such as the required operation, operands and result location, and size. An example of such an instruction is shown in Figure 14.

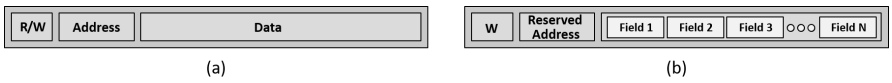


Fig. 14 Examples of the structure of an mMPU instruction. In a conventional memory access instruction (a) the instruction is composed of a direction (Read/Write) bit, an address field and a data field. An instruction for in-memory computing (b) is always in the Write direction, and written to an address which is reserved by the controller for computing instructions. The rest of the bits are used to transmit any information needed for the execution of the command and may specify the operation to be carried out, the input/output location and size, *etc.*

6 Conclusions

Data transfer between processing and memory units is the major performance and energy-efficiency bottleneck of modern computing systems, commonly known as the von Neumann bottleneck. Whereas prior art has tried to reduce the distance between processing and memory units to solve this problem, we propose the mMPU, an entirely different solution that can tackle the von Neumann bottleneck even more efficiently. In the mMPU, we rely on employing memristive memory cells directly for processing, which largely eliminates the necessity for data transfer. We also present MAGIC, a technique to execute logical operations within the memristive memory crossbar without any modification of the memory structure. We further show how to extend execution of a single MAGIC gate to a parallel execution of several MAGIC gates within the memory crossbar. We present our recent works on the mMPU microarchitecture design, which includes the mMPU controller and an automatic logic synthesis tool. Finally, we describe implications of the system integration of the mMPU while using it in two different ways, *i.e.,* an accelerator mode and in a main memory mode. Applications that will benefit the most from this new architecture include includes deep learning, image processing, DNA sequencing, and matrix multiplication, which have a high degree of intrinsic parallelism and large amounts of data.

References

1. HSA Foundation: Harmonizing the Industry Around Heterogeneous Computing. URL http://www.hsafoundation.com/
2. JEDEC Solid State Technology Association: High Bandwidth Memory (HBM) DRAM. URL http://www.jedec.org/standards-documents/results/jesd235
3. Hybrid Memory Cube Consortium: Hybrid Memory Cube Specification 1.0 (2013)
4. Li et al., H.: Write disturb analyses on half-selected cells of cross-point rram arrays. In: Proc. IEEE Int. Rel. Physics Symp., pp. MY.3.1–MY.3.4 (2014)
5. Chen et al., Y.C.: An access-transistor-free (0T/1R) non-volatile resistance random access memory (RRAM) using a novel threshold switching, self-rectifying chalcogenide device. In: IEEE Int. IEDM '03 Tech. Dig. Electron Devices Meeting, pp. 37.4.1–37.4.4 (2003)
6. Balasubramonian, R., Grot, B.: Near-Data Processing. IEEE Micro **36**(1), 4–5 (2016). DOI 10.1109/MM.2016.1

7. Black, B.: Die Stacking is Happening! Proceedings of the International Symposium on Microarchitecture **2013**
8. Bojnordi, M.N., Ipek, E.: Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 1–13 (2016). DOI 10.1109/HPCA.2016.7446049
9. Cassuto, Y., Kvatinsky, S., Yaakobi, E.: Sneak-path constraints in memristor crossbar arrays. In: Proc. IEEE Int. Symp. Inform. Theory (ISIT), pp. 156–160 (2013)
10. Chakraborti, S., Chowdhary, P.V., Datta, K., Sengupta, I.: Bdd based synthesis of boolean functions using memristors. In: 2014 9th International Design and Test Symposium (IDT), pp. 136–141 (2014). DOI 10.1109/IDT.2014.7038601
11. Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., Xie, Y.: PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 27–39 (2016). DOI 10.1109/ISCA.2016.13
12. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 337–340 (2008)
13. Dlugosch, P., Brown, D., Glendenning, P., Leventhal, M., Noyes, H.: An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. IEEE Transactions on Parallel and Distributed Systems **25**(12), 3088–3098 (2014). DOI 10.1109/TPDS.2014.8
14. Eckert, Y., Jayasena, N., Loh, G.H.: Thermal Feasibility of Die-Stacked Processing in Memory. Proceedings of the 2nd Workshop Near-Data Processing (2014)
15. Elliott, D.G., Stumm, M., Snelgrove, W.M., Cojocaru, C., Mckenzie, R.: Computational RAM: implementing processors in memory. IEEE Design Test of Computers **16**(1), 32–41 (1999). DOI 10.1109/54.748803
16. Gokhale, M., Holmes, B., Iobst, K.: Processing in memory: the Terasys massively parallel PIM array. Computer **28**(4), 23–31 (1995). DOI 10.1109/2.375174
17. Guckert, L., Swartzlander, E.E.: MAD gates: Memristor logic design using driver circuitry. IEEE Trans. Circuits Syst. II, Exp. Briefs **64**(2), 171–175 (2017). DOI 10.1109/TCSII.2016.2551554
18. Guo, Q., Guo, X., Bai, Y., Ipek, E.: A resistive TCAM accelerator for data-intensive computing. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 339–350. ACM (2011)
19. Guo, Q., Guo, X., Patel, R., Ipek, E., Friedman, E.G.: AC-DIMM: Associative Computing with STT-MRAM. ACM SIGARCH Computer Architecture News **41**(3), 189–200 (2013)
20. Huang, J.J., Tseng, Y.M., Luo, W.C., Hsu, C.W., Hou, T.H.: One selector one resistor (1s1r) crossbar array for high-density flexible memory applications. pp. 31.7.1–31.7.4. IEEE (2011)
21. Hur, R.B., Kvatinsky, S.: Memristive memory processing unit (MPU) controller for in-memory processing. In: 2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE), pp. 1–5 (2016). DOI 10.1109/ICSEE.2016.7806045
22. Hur, R.B., Talati, N., Kvatinsky, S.: Algorithmic Considerations in Memristive Memory Processing Units (MPU). In: CNNA 2016; 15th International Workshop on Cellular Nanoscale Networks and their Applications, pp. 1–2 (2016)
23. Hur, R.B., Wald, N., Talati, N., Kvatinsky, S.: SIMPLE MAGIC: Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC. Proceeding of the IEEE International Conference on Circuits Aided Design, November 2017
24. Kvatinsky, S., Belousov, D., Liman, S., Satat, G., Wald, N., Friedman, E.G., Kolodny, A., Weiser, U.C.: MAGIC – Memristor-Aided Logic. IEEE Transactions on Circuits and Systems II: Express Briefs **61**(11), 895–899 (2014). DOI 10.1109/TCSII.2014.2357292
25. Kvatinsky, S., Friedman, E.G., Kolodny, A., Weiser, U.C.: The Desired Memristor for Circuit Designers. IEEE Circuits and Systems Magazine **13**(2), 17–22 (2013). DOI 10.1109/MCAS.2013.2256257
26. Kvatinsky, S., Satat, G., Wald, N., Friedman, E.G., Kolodny, A., Weiser, U.C.: Memristor-based material implication (imply) logic: Design principles and methodologies. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **22**(10), 2054–2066 (2014). DOI 10.1109/TVLSI.2013.2282132

27. Kvatinsky, S., Wald, N., Satat, G., Kolodny, A., Weiser, U.C., Friedman, E.G.: MRL–Memristor Ratioed Logic. In: 2012 13th International Workshop on Cellular Nanoscale Networks and their Applications, pp. 1–6 (2012). DOI 10.1109/CNNA.2012.6331426
28. Lee, J., Jo, M., jun Seong, D., Shin, J., Hwang, H.: Materials and process aspect of cross-point RRAM (invited). Microelectronic Engineering **88**(7), 1113 – 1118 (2011)
29. Levy, Y., Bruck, J., Cassuto, Y., Friedman, E.G., Kolodny, A., Yaakobi, E., Kvatinsky, S.: Logic operations in memory using a memristive akers array. Microelectronics Journal **45**(11), 1429 – 1437 (2014)
30. Li, S., Xu, C., Zou, Q., Zhao, J., Lu, Y., Xie, Y.: Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: Design Automation Conference (DAC), pp. 1–6 (2016). DOI 10.1145/2897937.2898064
31. Lynch, W.: Worst-case analysis of a resistor memory matrix. IEEE Trans. Comput. **C-18**(10), 940–942 (1969)
32. Mishchenko, A.: ABC: A System for Sequential Synthesis and Verification (2012). URL http://www.eecs.berkeley.edu/ alanmi/abc/
33. Oskin, M., Chong, F.T., Sherwood, T.: Active Pages: A Computation Model for Intelligent Memory. SIGARCH Comput. Archit. News **26**(3), 192–203 (1998). DOI 10.1145/279361.279387. URL http://doi.acm.org/10.1145/279361.279387
34. Papandroulidakis, G., Vourkas, I., Vasileiadis, N., Sirakoulis, G.C.: Boolean logic operations and computing circuits based on memristors. IEEE Trans. Circuits Syst. II, Exp. Briefs **61**(12), 972–976 (2014). DOI 10.1109/TCSII.2014.2357351
35. Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K.: A Case for Intelligent RAM. IEEE Micro **17**(2), 34–44 (1997). DOI 10.1109/40.592312. URL http://dx.doi.org/10.1109/40.592312
36. Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K.: Intelligent RAM (IRAM): chips that remember and compute. In: 1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers, pp. 224–225 (1997). DOI 10.1109/ISSCC.1997.585348
37. Reuben, J., Ben-Hur, R., Wald, N., Talati, N., Ali, A.H., Gaillardon, P.E., Kvatinsky, S.: Memristive Logic: A Framework for Evaluation and Comparison. International Symposium on Power and Timing Modeling, Optimization, and Simulation (PATMOS) 2017 (in press)
38. Shin, S., Kim, K., Kang, S.M.: Analysis of passive memristive devices array: Data-dependent statistical model and self-adaptable sense resistance for RRAMs. Proc. IEEE **100**(6), 2021–2032 (2012)
39. Talati, N., Gupta, S., Mane, P., Kvatinsky, S.: Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC). IEEE Transactions on Nanotechnology **15**(4), 635–650 (2016). DOI 10.1109/TNANO.2016.2570248
40. Wang, K., Qi, Y., Fox, J.J., Stan, M.R., Skadron, K.: Association Rule Mining with the Micron Automata Processor. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 689–699 (2015). DOI 10.1109/IPDPS.2015.101
41. Wong, H.S.P., Lee, H.Y., Yu, S., Chen, Y.S., Wu, Y., Chen, P.S., Lee, B., Chen, F.T., Tsai, M.J.: Metal Oxide RRAM. Proceedings of the IEEE **100**(6), 1951–1970 (2012). DOI 10.1109/JPROC.2012.2190369
42. Woods, W., Taha, M.M.A., Tran, S.J.D., Brger, J., Teuscher, C.: Memristor panic: A survey of different device models in crossbar architectures. In: Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH15), pp. 106–111 (2015). DOI 10.1109/NANOARCH.2015.7180595
43. Xie, L., Nguyen, H.A.D., Taouil, M., Hamdioui, S., Bertels, K.: Fast boolean logic mapped on memristor crossbar. In: International Conference on Computer Design, pp. 335–342 (2015). DOI 10.1109/ICCD.2015.7357122
44. Yang, C.T., Huang, C.L., Lin, C.F.: Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. Computer Physics Communications **182**(1), 266 – 269 (2011)
45. Yavits, L., Kvatinsky, S., Morad, A., Ginosar, R.: Resistive Associative Processor. IEEE Computer Architecture Letters **14**(2), 148–151 (2015). DOI 10.1109/LCA.2014.2374597

46. Zha, Y., Li, J.: Reconfigurable in-memory computing with resistive memory crossbar. In: International Conference on Computer-Aided Design, pp. 1–8 (2016). DOI 10.1145/2966986.2967069
47. Zidan, M.A., Fahmy, H.A.H., Hussain, M.M., Salama, K.N.: Memristor-based memory: The sneak paths problem and solutions. Microelectronics Journal **44**(2), 176 – 183 (2013)