# SPIMulator: A Spintronic Processing-in-memory Simulator for Racetracks

PAVIA BERA, University of South Florida, Tampa, USA
STEPHEN CAHOON, University of Pittsburgh, Pittsburgh, USA
SANJUKTA BHANJA, University of South Florida, Tampa, USA
ALEX JONES, University of Pittsburgh, Pittsburgh, USA

In-memory processing is becoming a popular method to alleviate the memory bottleneck of the Von Neumann computing model. With the goal of improving both latency and energy cost associated with such in-memory processing, emerging non-volatile memory technologies, such as Spintronic magnetic memory, are of particular interest, as they can provide a near-SRAM read/write performance and eliminate nearly all static energy without experiencing any endurance limitations. Spintronic Racetrack Memory (RM) further addresses density concerns of spin-transfer torque memory (STT-MRAM). Moreover, it has recently been demonstrated that portions of RM nanowires can function as a polymorphic gate, which can be leveraged to implement multi-operand bulk bitwise operations. With more complex control, they can also be leveraged to build arithmetic integer and floating point processing in memory (PIM) primitives. This article proposes SPIMulator, a Spintronic PIM sim*ulator* that can simulate the storage and PIM architecture of executing PIM commands in Racetrack memory. SPIMulator functionally models the polymorphic gate properties recently proposed for Racetrack memory, which allows *transverse access* that determines the number of "1"s in a segment of each Racetrack nanowire. From this simulation, SPIMulator can report real-time performance statistics such as cycle count and energy. Thus, SPIMulator simulates the multi-operand bit-wise logic operations recently proposed and can be easily extended to implement new PIM operations as they are developed. Due to the functional nature of SPIMulator, it can serve as a programming environment that allows development of PIM-based codes for verification of new acceleration algorithms. We demonstrate the value of SPIMulator through the modeling and estimations of performance and energy consumption of a variety of example applications, including the Advanced Encryption Standard (AES) for encryption primarily based on logical and look-up operations; multiplication of matrices, a frequent requirement in scientific, signal processing, and machine learning algorithms; and bitmap indices, a common search table employed for database lookups.

CCS Concepts: • **Hardware → Functional verification;**

Additional Key Words and Phrases: In-memory computing, domain wall memory, spintronic memory, processing in memory

**ACM Reference Format:**
Pavia Bera, Stephen Cahoon, Sanjukta Bhanja, and Alex Jones. 2024. SPIMulator: A Spintronic Processing-in-memory Simulator for Racetracks. *ACM Trans. Embedd. Comput. Syst.* 23, 6, Article 94 (September 2024), 27 pages. https://doi.org/10.1145/3645112

Authors' addresses: P. Bera and S. Bhanja, University of South Florida, 4202 E Fowler Ave, Tampa, FL 33620, USA; e-mails: paviabera@usf.com, bhanja@usf.edu; S. Cahoon and A. Jones, University of Pittsburgh, 1238 Benedum Hall, Pittsburgh, PA 15261, USA; e-mails: stc127@pitt.edu, akjones@pitt.edu.

ACM 1539-9087/2024/09-ART94
https://doi.org/10.1145/3645112

## 1 INTRODUCTION

Among the most important challenges of system design is speed and efficient data access. Emerging **non-volatile memories (NVMs)** have been widely studied, particularly over the past decade, to improve the energy and density properties of conventional memories, allowing increasingly dense low-power memories to be placed closer to the processor. Among these, **Spin-Transfer Torque Magnetic memory (STT-MRAM)** and **Phase-change memory (PCM)** [36] have gained commercial traction, the former from Everspin and the latter from Intel and Micron in their Optane product. STT-MRAM, **resistive memory (ReRAM)** [3], and **ferroelectric memory (FRAM)** have also had industrial development from companies such as Samsung, HP, and TI, respectively. To increase the density further, **multi-level cells (MLCs)** have become popular to extend beyond the one-cell, one-access device methodology by storing multi-bit symbols in a single device by storing different voltage or resistance levels. This has improved the density but creates challenges with access speed and fidelity.

Another particular concern for these devices is that many of them have limited endurance. While Flash has endurance as low as $10^5$ write cycles, PCM and resistive memories exhibit $10^8$–$10^9$ and $10^{11}$–$10^{12}$ write cycles, respectively [26]. STT-MRAM is an attractive alternative due to its near-SRAM performance, CMOS compatibility, low static power, and good endurance [6]. Unfortunately, STT-MRAM has insufficient density for storage class applications.

More fundamentally, even these increasingly dense technologies do not directly address the core problem of the "memory wall" [12, 27]. The memory wall is a side-effect of the Von Neumann architecture, which separates the processor from a distinct memory. The speed differential between the processor and memory has created a significant bottleneck, as dataset sizes grow to exceed what can fit within caches. Some paradigms have changed from working sets to operating on streaming or infrequently reused data. **Processing-in-memory (PIM)** [10, 11, 21, 28, 31, 32] has been proposed to reduce demands on the memory bus to relieve the memory wall bottleneck. This can allow continued scaling. PIM solutions have been demonstrated for DRAM [21, 28], PCM [10], ReRAM [23], and STT-MRAM [22] but are limited to two operand operations although multi-operand extensions for NVMs have been discussed [10].

A recent exciting alternative memory is the spintronic **"Racetrack" memory (RM)**. RM extends the free magnetic layer of the STT-MRAM **magneto tunnel junction (MTJ)** into a nanowire separated by manufactured notches. This allows the magnetization to be segregated into domains, separated by domain walls. As such, it is sometimes referred to as **domain-wall memory (DWM)**.[1] Thus, data must be shifted to an access port to be read or written. It serves as a lateral multi-level cell, but individual bits can be accessed. RM, originally proposed and demonstrated by IBM [16, 17], retains the static energy benefits of STT-MRAM with a 10× higher density [33]. RM has a theoretical area per bit as small as $2F^2$, where $F$ is the technology feature size [1]. Moreover, RM avoids endurance challenges by providing $\geq 10^{16}$ write cycles [2], which far exceeds other technologies.

However, the notion of data shifting requires that RM architectures be designed fundamentally differently than traditional single or even multi-level cell devices. RM access times depend on both traditional delays as well as variable shift delays. A variety of solutions have been proposed to mitigate shifting overheads [2, 9, 24, 29, 30]; however, shift minimization remains an important challenge. Thus, it is very difficult to adopt existing memory simulation tools for DRAM or other NVMs to model RM because of this shift delay. For shift minimization, several custom simulators [24, 29, 30] were designed, and one simulator, RTSim [7], was proposed to study large-scale main memories built from RM [9].

[1]We refer to RM and DWM interchangeably to refer to the same structure.

Moreover, the behavior of fault tolerance and PIM for these devices is also fundamentally different from existing tools. For instance, the landscape changed recently when a technique to transversely access a portion of the RM nanowire was proposed. This opened up interesting properties for the device, such as protection against misalignment and pinning while shifting [14]. It also has interesting properties for PIM, allowing multi-operand bulk bitwise processing, which has been shown to benefit high-dimensional computing, multi-operand addition, and two-operand operations for integer and fixed-point data [13], and floating-point data [15], which open up myriad opportunities for improving the memory wall bottleneck.

Thus, we present SPIMulator, a **S**pintronic **P**rocessing-**I**n-**M**emory **S**imulator for cycle-level functional simulation of RM with PIM capabilities. We demonstrate how the PIM capabilities from prior work [13, 15, 31] can be written and implemented in SPIMulator to validate bulk-bitwise operations and arithmetic operations. We demonstrate increasingly complex algorithms to show how RM using PIM can execute these algorithms to introduce SPIMulator's capabilities. Unfortunately, RTSim is insufficient for building the **SPIM**ulator because the data is not retained. SPIMulator allows a programming interface to RM PIM processing to develop and validate processing on actual data. Knowledge about the delays and energy consumption of elements of the memory pipeline can also provide estimates of performance and energy consumption. Through pipelining it is possible to extend the tool to model both **single-instruction-multiple-data (SIMD)** and tightly pipeline execution models. SPIMulator has been validated for performance and power estimates against cycle-level simulators. We do not claim that the **SPIM**ulator is a fully cycle-accurate simulator. However, it can provide fast feedback to PIM program designers and is easily extensible to implement new extensions to the PIM instruction set.

In particular, in this article, we make the following contributions: We

- Present the features and functions of **SPIM**ulator to describe how to articulate PIM algorithms using RM.
- Describe how **SPIM**ulator models shifting faults in RM and demonstrate methods to detect single-bit and multi-bit errors by implementing parity checking, error correction, and shifting fault recovery.
- Demonstrate how **SPIM**ulator can implement multiplication, arithmetic reduction, and bitmap indices.
- Demonstrate **SPIM**ulator's capabilities using each step of the 128-bit AES encryption algorithm.
- Provide an experimental analysis by reporting the performance and energy consumption of these algorithms as tested with **SPIM**ulator.

## 2 BACKGROUND

In this section, we provide a background on RM, its behavior and methods for access, and how it can be used to build memory arrays. We then discuss recent advances, including the access modes that allow individual bit-wise and multi-bit access and how this has been used to support multiple applications, including implementing the polymorphic gate that serves as the foundation for PIM. We then describe details of the PIM approach and memory architecture to support PIM, including modifications to the sensing circuit leveraging **transverse read (TR)** [20] to realize multi-operand PIM. This includes the algorithms to achieve multi-operand addition and two-operand multiplication.

### Racetrack Memory Fundamentals

Racetrack Memory is a spintronic NVM formed from ferromagnetic nanowires. An example nanowire is shown in Figure 1. These nanowires consist of domains, shown in blue in the

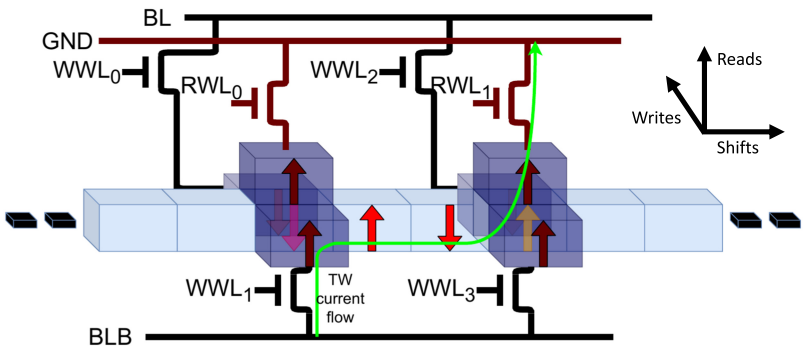



Fig. 1. Racetrack memory nanowire example using perpendicular magneto anisotropy (PMA) (Racetrack memory 2.0) [2] with two access ports that each supported standard reads through an MTJ as well as shift writing [25]. The access transistors update to correct an error and to support transverse accesses shown using the green line [25]. During a transverse read, current traverses the nanowire with resistance and resulting voltage determined by the number of 1's (number of parallel and antiparallel domains) [20]. For a transverse write, data is shifted via $WWL_1$ into the nanowire, and the value at the rightmost head is shifted to GND via $RWL_1$ [13]. Note that transverse writes can be conducted to either end of the nanowire.

figure, separated by **domain wall memory (DWM)** formed by explicitly engineered notches. Each domain is represented by a direction, either perpendicular (+Z/-Z) or parallel (+X/-X) magnetic anisotropy (i.e., magnetization direction). The red arrows in the figure show example perpendicular anisotropy in the example. Data is read by comparing the direction (anti-parallel or parallel) to a fixed reference, producing a binary value of 0 or 1. For a domain to be read or written it must be aligned with a head that is shared among several domains [35]. Domains are shifted by applying a small current across the nanowire, which moves the correct domain into the domain aligned with the head. This small current needs to be precise to avoid shifting fault [14, 34], which is when the incorrect domain becomes aligned with the head due to over- or under-shifting.

Memories constructed from RM can be organized in a hierarchical fashion similar to DRAM using traditional Ranks, Banks, Subarrays, and Tiles, as shown with an overview in Figure 2(a) with the bank expanded in Figure 2(c) and the subarray shown in Figure 2(d). A traditional tile might be 512 rows of 512-bits. However, RM nanowires cannot generally be realized in 512-domain lengths. Thus, these memories subdivide tiles into **Domain Block Clusters (DBCs)**, as shown in Figure 2(b). DBCs are collections of nanowires that are shifted together forming the width of a row. The DBCs in this case are bundled into groups of 512 nanowires but with a length of 32 readable domains, each as shown in Figure 2(b). The nanowires of a DBC are shifted in unison, which allows each read or write to access 512 bits at a time. In the example, each tile is composed of 16 DBCs. Thus, rows may target different DBCs, but accessing a particular row in a DBC may require shifting to align that location of the access port.

The nanowire in the figure has two access ports shown in dark blue. Traditionally, this allows reduced shifting, because data could be accessed at either access port. At each port, above the nanowire is a fixed magnetic layer separated from the magnetically free domain of the nanowire through an insulator, forming a **magneto tunnel junction (MTJ)** used in STT-MRAM. If the magnetically free domain is aligned with the fixed layer, then it has a lower tunneling resistance (e.g., a logic "0") and if anti-parallel, then it has a higher resistance (e.g., a logic "1") and can be read by a current applied across the MTJ. While it is possible to change the magnetization using the MTJ with a stronger current, a faster and lower energy solution was proposed to place

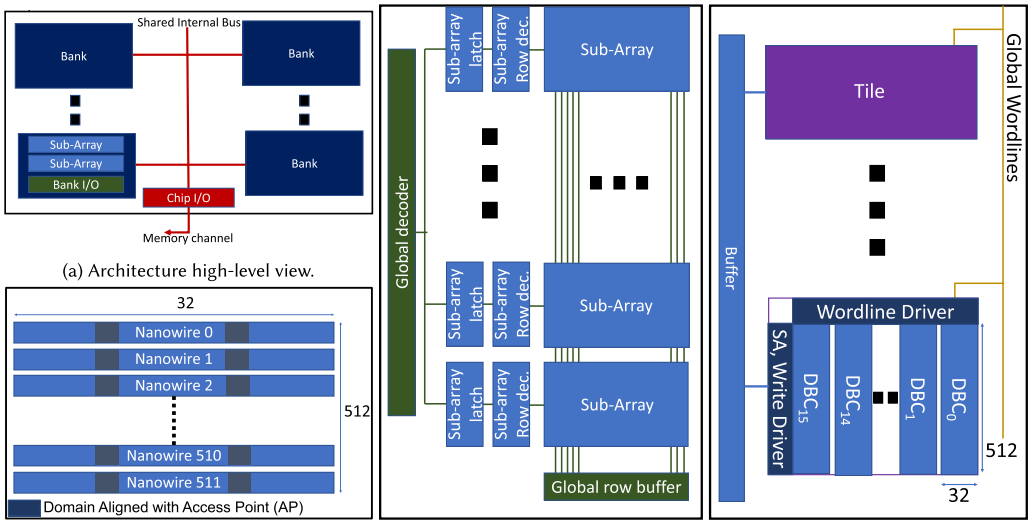



Fig. 2. Racetrack memory main memory architecture [8, 13].

a fixed parallel and anti-parallel region orthogonal to the nanowire to allow an orthogonal shift rather than a current write [25].

Beyond traditional reading and writing, RM can now leverage a recently proposed transverse reading technique [20] that counts the number of ones between two "heads" (access points) or between a head and an extremity by measuring the resistivity between these two locations. This value of resistivity is only unique to the number of ones and not their respective locations. An example of this current path is shown in green in the figure. Unfortunately, the sense margin for these segments decreases as the number of domains in the segment increases [20]. Because of the increasing sense margin, there is a maximum distance at which this can be performed, known as the transverse read distance or $TR_d$. If using two heads, then placing them at the maximum $TR_d$ allows for the most coverage of data, which will be useful for PIM operations discussed later. A second method for counting the number of "1"s in an RM nanowire segment has been proposed using a multi-domain MTJ [4, 14], which has potentially better scalability than TR.

CORUSCANT proposes another operation on DWM known as a **transverse write (TW)** [13]. A transverse write is an operation that allows for a shift and write to be done between the heads at the same time. As shown in Figure 1, the desired write done via $WWL_1$ is pushed into the nanowire at access port zero $AP_0$ and a shift is done between the heads. This results in the value noted in yellow at the opposing head being pushed out to ground via $RWL_1$ at $AP_1$, leaving the newly written value and all but one of the previous values between the heads.

The memories used for PIM computation require each nanowire in a PIM-enabled DBC to have an equal number of heads (at least 2) to enable these TR and TW operations. By enabling tranverse reads, each DWM nanowire can function as a *polymorphic* gate that is optimized for computing arbitrary logic functions, sum, and carry logic output. This design approach implements multi-operand bulk-bitwise operations that can outperform recently proposed main-memory PIM architectures [35]. Also, this method of counting ones allows for the implementation of multi-operand addition in memory. Using a carry-save-inspired approach and logical shifts, multiplication can

also be implemented. By using large reductions, the number of additions for multiplication is reduced to one.

## 3 SPIMULATOR

Processing-in-memory has received great traction from the architectural community, as it can help address the memory wall problem. Also, it can tackle data latency and excess energy consumption related to the overuse of the memory bus due to data movement. **Processing-in-memory (PIM)** can reduce the stress on the memory bus by handling the operations directly in memory rather than transferring data to the CPU to perform the computations. Emerging nonvolatile memories such as **Domain-wall memory (DWM)** an extension of **Spin transfer torque magnetic random access memory (STT-MRAM)** have shown great potential for memory processing. An example of a DWM-based in-memory computing device leverages transverse reading to provide *multi-operand bulk-bitwise logic* [13]. However, since calculating PIM operations is not traditional, it can be difficult to design algorithms in memory and verify their correctness. This is where a tool that simulates such behavior is useful, allowing users to test their designs and tweak their algorithms as needed.

**SPIM**ulator is a DWM-based PIM simulator capable of simulating the PIM architecture on a Domain Block Cluster level. **SPIM**ulator is a complete solution that gives the users the reconfigurability of access port location and TR distance but also provides a versatile main memory design through integration with different tools. This versatility of the main memory design and the PIM block can allow users to explore different memory designs [7] for different algorithms. This allows the users to streamline the memory size according to their algorithmic needs, saving space and energy costs. The tool also visualizes each command given on an instruction list, allowing the users to determine the correctness of any given algorithm and giving them insight into how the PIM logic functions. Cycle and energy count estimation is also shown at the end of the execution of the instruction set, which gives the users the ability to customize their design of the algorithm for better efficiency.

This work demonstrates simulation altering one tile comprising 16 DBCs, as represented in Figure 4. There is a great deal of importance in simulating one tile, as we can represent a simple state machine using one tile. The use of multiple small tiles can prove to be beneficial in the long run, as it can reduce latency through parallel and pipelined execution. Although we have demonstrated only one tile, **SPIM**ulator can be conceptualized with multiple tiles and written with modified versions of other simulators, e.g., Reference [7]. We plan on demonstrating the scalability of the main memory and running parallel algorithms in our future work. Although some existing simulators are capable of modelling the main memory architecture [7, 18, 19], **SPIM**ulator is the only one with PIM capabilities and *multi-operand bulk-bitwise logic* while retaining all other functionality, e.g., read and write. This allows the implementation of a wide variety of algorithms in memory while having the ability to configure or integrate the memory space with other tools in accordance with the implementation needs of the algorithms.

*The **SPIM**ulator tool is available open-source and can be found at*https://github.com/Pitt-JonesLab/DWMsimulator

The expected developer design flow would follow the steps outlined in Figure 3. The most crucial step is step 1, which is to construct the script for the execution of the algorithm. Some significant constraints and regulations are necessary to run the script successfully. In the following section, the constraints and regulations are discussed in more detail in Sections 3.1 and 3.3. In Section 3.1, the DBCs and nanowires addressing are explained. Section 3.3 explains the PIM instructions and different write operations.

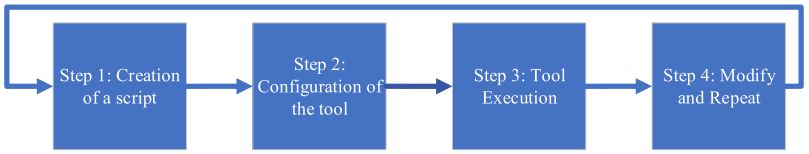



Fig. 3. SPIMULATOR design flow.

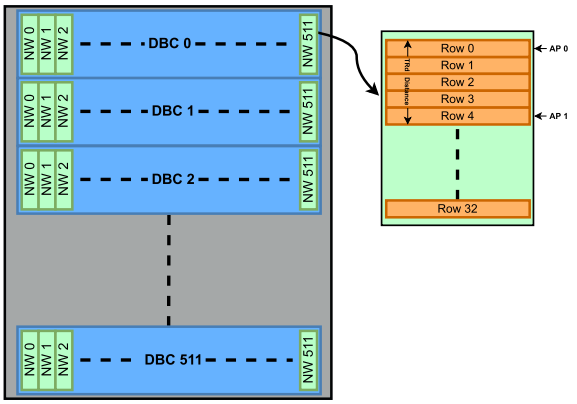



Fig. 4. DBC architecture.

### 3.1 Memory Layout and Addressing

**SPIM**ulator defaults to the model from prior work [13] with one PIM-enabled tile constructed from 16 DBCs and a local row buffer to perform reads and writes to memory.[2] Per DBC, given there are 512 nanowires and 32-bit length tracks, this requires 16 DBCs (Figure 4). Each DBC is assumed to possess at least two access ports **AP0** and **AP1** at the head and tail of the transverse access region, separated by $TR_d$ domains. **SPIM**ulator will access data by shifting and aligning the access ports **AP0 and AP1** with the data automatically to incur the minimum number of shifts. Thus, for a traditional read/write access, if the tail, i.e., **AP1**, is close to the operation location, then the tail end will move so as not to have more shifts than necessary. The same will be true for the head, i.e.,**AP0**. If **AP0** is closer to the operation location, then to minimize shifts, the **SPIM**ulator will align the head with the operation location. Conventionally, the access points remain fixed in DWM architectures, and the data moves to align with the access port. To accomplish this, the device requires unused buffering domains on the exterior of the nanowire. These are *overhead* domains required to keep the access port fixed. However, we can change the frame of reference such that the access point is shifting in relation to the data for the ease of display in **SPIM**ulator.

For executing PIM programs, **SPIM**ulator mimics the translation layer at the memory controller to convert these instructions in basic memory operations. Aside from logical and arithmetical operations, like prior work [13], the PIM block is capable of logical shifts and transverse reads and writes, which can also be translated into proper operation from the PIM instructions. To preserve the expandability and generality of the **SPIM**ulator, we have upheld the following nomenclature for the instructions shown in Equation (1), below. Each PIM cpim instruction consists of a source address, *"src,"* indicating which DBC and nanowire position to align with the nearest access port

[2]In principle, there is no reason not to simulate multiple PIM enabled tiles per subarray. However, this configuration was selected due to the peripheral circuitry overhead [13].

before issuing the operation *"op"* instruction. For PIM operations, the operation occurs *between* the access points, which is the **Transverse Read distance (TR$_d$)**. **SPIM**ulator also uses one (or more) new instructions [21] that map the operations to be issued by the memory controller. *"op"* is the operation selected from the PIM block. A more detailed explanation is discussed in Section 3.3. After the *"op"* is computed, the result is stored in the local buffer until a *"write_op"* operation is executed at the destination address *"dst"* where the result will be stored. To integrate our write operations with the cpim instruction, we assume the instruction to include the *"write_op."* Here, *"write_op"* is a three-bit value that represents a write operation in the same DBC address range where the cpim *"op"* is executed. It is also important to note that we have an intelligent alignment of access ports. That means there will be a shift operation before *cpim "op"* is executed automatically to align the access ports to the *"src"* or *"dst"* address within the DBC. The energy and the cycle cost for such shifts are also automatically incorporated with the costs of *cpim* instructions.

$$cpim\ \ dst\ \ src\ \ op\ \ blocksize\ \ write\_op \tag{1}$$

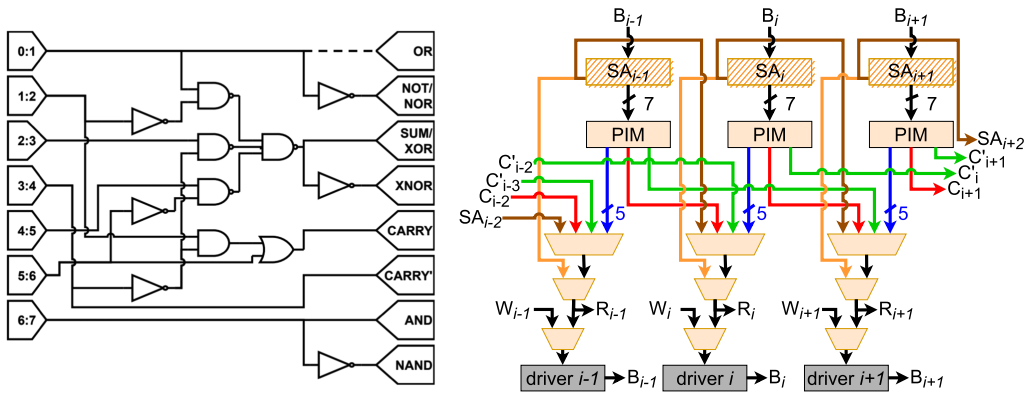
**SPIM**ulator's addressing is conducted using an address targeting an organization like Figure 4. The example presumes a single tile, 16 DBCs per tile, and 32 rows per DBC. After identifying the tile, addresses are stripped to the lower nine bits starting from "0," the first row for the first DBC (DBC 0), continuing to "32," the first row for the second DBC (DBC 1), and so on. Extrapolating, we have a total of "512" addresses, and the last row of the last DBC (DBC 15) will be "511." Since virtual addresses are used by the tool, the memory controller translates these virtual addresses to physical addresses.

### 3.2 Execution and Latency

Each instruction requires a specific time to execute. The tool schedules the busy (wait) time required for each instruction based on the number of core reads and writes. For instance, multiplication requires multiple sub-reads and writes to compute the final result. Each of this memory access requires time to access the row ($t_{RAS}$), a column access delay ($t_{RCD}$), time to shift to the correct column ($t_{RP}$), and time to access the column ($t_{CAS}$). Writes also require time to stabilize before they can be re-accessed ($t_{WR}$). Using Equation (2) [13] for example, writes use $t_{RAS}$ + $t_{RCD}$ + $t_{RP}$ + $t_{CAS}$ + $t_{WR}$ cycles.

$$DWM[cycles] = t_{RAS}/t_{RCD}/t_{RP}/t_{CAS}/t_{WR} = 9/4/2S/4/4 \tag{2}$$

To compute the latency and energy values for an algorithm, each command is split into its base components. These components are as follows: *Write, Transverse Write, Read, Transverse Read, Shifts,* and *Stores*. Each affects the latency and energy differently. Reads represent the activation of a row and column in memory to read a value into the row buffer. Writes are similar to reads in that they activate a row and column in memory, but instead, they write a value from the row buffer into memory and require additional time to stabilize. Transverse reads and writes are similar in latency to their previously mentioned counterparts but require different energy amounts due to shifting (T Write) or reading through PIM logic (T Read). As explained previously, shifts are required to align addresses with the access points. Finally, stores are just traditional memory accessed by the CPU. Since stores are required to do a write as part of their functionality, the additional time spent for these writes on the bus is also accounted for and does not add any additional cost in energy. Most PIM instructions require a TR to obtain a result. For tiles/DBCs with more than one access port to conduct TR, the **sensing circuitry (SA)** is generally modified, and the output of these SA is a seven-level sensing circuit, which is the input for the PIM logic as shown in Figure 5(a). The PIM logic operator is determined by selecting the output through multiplexer, as shown in the modified rowbuffer that contains the PIM logic as shown in Figure 5(b). Note this rowbuffer is




(a) PIM logic for up to seven operands (b) Local buffer extension for PIM and logical shifting

Fig. 5. PIM hardware logic tile overview [13].

also modified to support logical shifting. This process is simulated by scheduling the delays and energy consumption from Equation (2) and the extended peripheral circuitry using event-driven simulation.

### 3.3 Instructions

The instruction set for **SPIM**ulator can be divided into two categories of PIM operations: basic memory and logical/arithmetic instructions. In this section, we will explain each set of instructions with examples. For logical/arithmetic instructions, we have implemented basic gates ~(AND, OR, NOT, XOR, XNOR, NAND, and NOR) and addition and multiplication operations ~(SUM and MULT). We will demonstrate each logical/arithmetic instruction and their behavior in the PIM block. The PIM capabilities have been assumed to be present in all of the DBCs in the memory architecture. Figure 4 shows two access points **AP0** and **AP1**, which are spaced according to the user-assigned **Transverse Read distance (TR$_d$)**. For the examples followed, we have assigned a conservative TR$_d$ of 7, which can be scaled up to 32, as stated in Reference [20]. **SPIM**ulator will presume that some portion of the virtual memory space will be reserved for PIM operation, and the operating system can manage this space when conducting virtual to physical address translation, as is often the case with memory-mapped I/O. Thus, the user can schedule PIM operations in memory and align with tile and DBC boundaries.

**SPIM**ulator is equipped with Basic Memory Operations such as {WRITE, COPY, STORE, Logical SHIFT Left, Logical SHIFT Right, READ}. Due to our intelligent alignment of the access ports, either the head **AP0** or the tail **AP1** aligns itself with the given address at *"dst."* This helps minimize the cycle counts and energy consumption. For logical/arithmetical operations *"op,"* the address in the *"src"* is always accessed by **AP0** to ensure that the correct operands are being used. Once the TR$_d$ head has aligned with the given *"dst"* and *"src,"* the given *"op"* is executed between the TR$_d$ distance. The *"op"* is executed among all of the *"blockwise,"* i.e., 512 nanowires. The result from the *"op"* is temporarily stored in the Local buffer (the information transfer bus among the DBCs) until the *"write_op"* is given. *"write_op"* ~["0," "1," "2," "3," "4," "5," "6"] is a "3-bit" identifier that represents the seven different types of possible write. Table 1 lists all basic memory instructions with different write types and Table 2 lists all the Logical and Arthritic operations instruction sets.

(A) **Write operations:**

(i) *Write: "0"* is one of the types of write (*"write_op"*) operation that will overwrite at the destination *"dst"* after the TR$_d$ head or tail aligns with the *"dst"* address. The data from the

Table 1. Basic Memory and Write Operation Instruction Sets

| CPIM | dst | src | op | Blksize | write_op | Description |
|---|---|---|---|---|---|---|
| cpim | $0–$511 | data | STORE | 512 | "0" | Write the data into the destination addr |
| cpim | $0–$511 (AP0) | data | STORE | 512 | "1" | Shifts content down within the $TR_d$ while transverse writing data at AP0 |
| cpim | $0–$511 | data | STORE | 512 | "2" | shifts content up within the $TR_d$ while transverse writing data at AP1 |
| cpim | $0–$511 | data | STORE | 512 | "3" | shifts content towards the bottom extremity (last row) and transverse writes data at AP0 |
| cpim | $0–$511 | data | STORE | 512 | "4" | shifts content up towards the top extremity and transverse writes data at AP1 |
| cpim | $0–$511 | data | STORE | 512 | "5" | shifts content up towards the top extremity and transverse writes data at AP0 |
| cpim | $0–$511 | data | STORE | 512 | "6" | shifts content down towards the bottom extremity and transverse writes data at AP1 |
| cpim | $0–$511 | $0–$511 | COPY | 512 | "0–6" | Reads data from **src** addr and writes at **dst** addr following any 0–6 options |
| cpim | $0–$511 | $0-$511 | SHL [1,8,32] | 512 | "0–6" | Logical data shift by 1/8/32 bits left and appends 1/8/32 bits of zeros at the right extremity |
| cpim | $0–$511 | $0–$511 | SHR [1,8,32] | 512 | "0–6" | Logical data shift by 1/8/32 bits right and appends 1/8/32 bits of zeros at the left extremity |

local buffer overwrites the data at the *"dst"* address and thus any previous information at *"dst"* will be lost. This instruction is to be used conjoined with other operational instructions such as *"AND," "SHL," "ADD,"* and so on. The result after performing any operations at *"src"* is stored in the local buffer temporarily and then overwritten at *"dst"* according to *"0."*

(ii) *Transverse write at AP0: "1"* is a special type of write operation that represents transverse write at **AP0**. This means that the data present within the $TR_d$ is shifted "down" towards **AP1** by one row and then the shifted data is written at *"dst"* address (see Figure 4). In doing so, the data is written at the address of **AP0** and data between **AP0** and row prior to **AP1** are shifted to the next address (similar to a technique used in wear leveling). The data at **AP1** is lost. This instruction is issued when there is a need to shift place data while keeping the head at the same address or to clear out old data.

(iii) *Transverse write at AP1: "2"* is a special type of write operation that represents transverse write at **AP1**. Similar to the above command, except data is shifted into **AP1** and the data between the access ports are shifted into the previous row's address and the data at **AP0** gets lost.

Table 2. Logical and Arithmetic Operation Instruction Sets

| CPIM | dst | src | op | Blksize | write_op | Description |
|---|---|---|---|---|---|---|
| cpim | $0–$511 | $0–$511 | and | 512 | "0–6" | and op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | or | 512 | "0–6" | or op within the $TR_d$ distance |
| cpim | $0–$511 | $0-$511 | nand | 512 | "0–6" | nand op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | nor | 512 | "0–6" | nor op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | xor | 512 | "0–6" | xor op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | xnor | 512 | "0–6" | xnor op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | not | 512 | "0–6" | not op within the $TR_d$ distance |
| cpim | $0–$511 | $0–$511 | carry | 512 | "0–6" | carry op counts number of 1's within $TR_d$ distance and outputs 1 for count = 2,3,6,7 |
| cpim | $0–$511 | $0–$511 | carryprime $C'$ | 512 | "0–6" | carryprime $C'$ op counts number of 1's within $TR_d$ distance and outputs 1 for count = 4,5,6,7 |
| cpim | $0–$511 | $0–$511 | add | 512 | "0–6" | add op for operands within the $TR_d$ - 2 distance |
| cpim | $0–$511 | $0–$511 | mult | 512 | "0–6" | 8-bit product |

(iv) *Transverse write at AP0 shift to bottom extremity: "3"* represents transverse write at **AP0**, but instead of deleting the data at **AP1**, it pushes it down by one row beyond the TR distance. This means that the entire data starting from **AP0** is shifted down by one address until the end of the nanowire. In doing so, only the last row of data at the end of the nanowire (i.e., modulus 32+31) will be lost.

(v) *Transverse write at AP1 shift to top extremity: "4"* represents transverse write at **AP1** and pushes the data at the track beyond the TR distance (above **AP0**) towards the top. This means all addresses beyond the $TR_d$ will be shifted up by one, with the top row (i.e., modulus 32) of the DBC being logically deleted.

(vi) *Transverse write at AP0 shift to top extremity: "5"* represents transverse write at **AP0** and pushes the data up the track (above AP0) by one row. This will result in all data above **AP0** being shifted up by one, and the data at the top address is lost.

(vii) *Transverse write at AP1 shift to bottom extremity: "6"* represents transverse write at **AP1** and pushes the data towards the bottom of the track below the $TR_d$. This means that the data below **AP1** will be shifted downwards. In doing so, only the data at the bottom address is deleted.

(B) **Basic Memory Operations:**

(i) *"STORE":* When issued with this instruction, the "data" is loaded from the CPU to the local buffer and then is written according to the *write_op* "(0–6)" after the $TR_d$ **AP0** or **AP1** aligns with the destination *"dst"* address. The peculiarity of this instruction is that the CPU data to be loaded can be supplied with the instruction. For example, "*cpim $96 0xA24B791CEF6 STORE 512 0* " implies that Data=0xA24B791CEF6 will be written at address $96 after the closest of the AP's aligns with $96.

(ii) *"COPY":* When issued with this instruction, the closest of the access points will first align with the source *"src"* address, and then the "data" is read from the source *"src"* address to the local buffer. The read data in the local buffer is then written according to the *write_op* "(0–6)" after the $TR_d$ **AP0** or **AP1** aligns with the destination *"dst"* address. The peculiarity of this instruction is that it reads the *"src"* first and then writes the read data. For example, "*cpim $96 $12 COPY 512 0*" implies that source address *"src"* $12 will be read and transferred to the local buffer. Then it will be written at *"dst"* address $96 after the

closest of the APs aligns with the source address \$12 and the destination address \$96, respectively.

(iii) *"SHL"* (logical shift left) represents the left shift of each bit at the *"src."* This instruction comes in three forms: SHL1, SHL8, and SHL32. When issuing any of the above, **SPIM**ulator's first step is to read at *"src"* address by using the intelligent alignment of the $TR_d$ access port and then the number of shifts required. The data is then read to the row buffer and shifted by the instructed number. Data at the left extremity is lost, and *0's* is added to the right extremity. This operation is conducted in the source address, and the result is stored temporarily in the local buffer until the write instruction is executed. The write instruction is always accompanied with *"SHL"* instruction. An example of this instruction "*cpim \$32 \$0 SHL1 512 0* " will first align either of the AP's nearest to \$0 and data will be left shifted by 1-bit. The shifted data will be in the local buffer before it is overwritten into the destination address \$32.

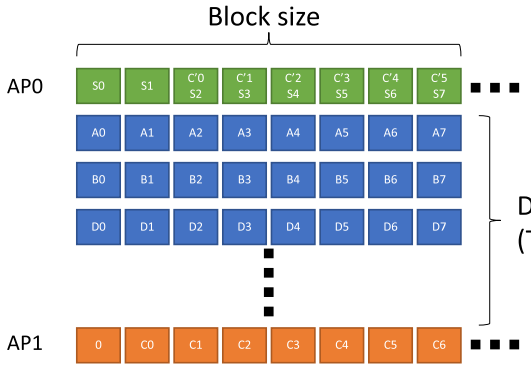
(iv) *"SHR"* (logical shift right) represents the right shift of each bit at the *"src"* address. This instruction comes in three forms: SHR1, SHR8, and SHR32. When issuing any of the above, **SPIM**ulator first reads the *"src"* address by using the intelligent alignment of the $TR_d$ access port and then determines the amount to be shifted. The data is then read to the local buffer and shifted by the desired amount. Data at the right extremity is lost, and *0's* is added to the left extremity. This operation is conducted in the source address, and the result is stored temporarily in the local buffer until the write instruction is executed. The write instruction is always accompanied by *"SHR"* instruction. An example for this instruction is "*cpim \$32 \$0 SHR8 512 0*," which will first align either of the APs nearest to \$0, and data will be right shifted by 8 bits. The shifted data will be in the local buffer before it is overwritten into the destination address \$32.

(v) *"R AP0 and R AP1"* represents a simple read instruction at the *src* location after aligning either **AP0** or **AP1** access port, which is the head or tail of $TR_d$ with *src*. Although this instruction is not mentioned in the cpim instruction set, it is important to have these options to issue a simple read command when necessary. An example of this instruction is "*read \$12 AP0*," which will align the access port AP0 with source address \$12 and then copy the data into the local buffer. These instructions can be helpful when data is required to be moved out of the memory to the CPU.

(C) **Logical and Arithmetic Operations:**

(i) *{"AND," "OR," "XOR," "XNOR," "NAND," "NOR," "NOT"}* represents the logical operation "and," "or," "xor," "xnor," "nand," "nor," or "not" when issued with the cpim instruction. The operation occurs among the operands between the TR distance after aligning the access port **AP0** with the source address *"src."* The operation result is temporarily stored in the local buffer until it is written according to the instruction's last three bits "(0–6)" to the destination *"dst"* address. An example of logical operations is: "*cpim \$32 \$0 and 511 0*" in which the *"and"* operation is executed from source address *"src"* location to the TR size. The result is then stored in the local buffer and overwritten at the destination address *"dst."*

(ii) *"CARRY"* and *"CARRYPRIME"* are special instructions that are required for the *"ADD"* and *"MULT"* operations. They are normally not called by the users, as they are not part of the arithmetic operations. But users can still have the option to call these instructions in **SPIM**ulator if they want to create custom functions that will require *"CARRY"* and *"CARRYPRIME"* calculations. Both *"CARRY"* and *"CARRYPRIME"* operations count the number of ones within the $TR_d$ distance and will output 1 to the local buffer. *"CARRY"* outputs 1 when the number of 1 within the $TR_d$ is any of the following: 2,3,6,7.




| No. of 1's between AP0 and AP1 | Output |
|---|---|
| 2,3,6,7 | CARRY = 1 |
| 4,5,6,7 | CARRYPRIME = 1 |

Fig. 6. Addition using TR.

*"CARRYPRIME"* outputs 1 when the number of 1 within the $TR_d$ is any of the following: 4,5,6,7. An example of the carry instruction is "*cpim $32 $0 carry 511 '1'*" which issues a carry operation at source address $0 and transverse writes at destination address at $0. Example of carryprime "*cpim $32 $0 carryprime 511 '0'*," which issues a carryprime operation at source address $0 and overwrites at destination address at $0. The output from these instructions can be written according to the instruction's last *"3-bits,"* i.e., "(0–6)."

(iii) *"ADD"* represents the arithmetic "addition" operation $A + B$. We will show an example of an addition operation for five operands in Figure 6 for $TR_d = 7$. It is important to note that we can perform addition for $(TR_d - 2)$ bits. Therefore, for our example, we can perform a 5-bit addition for $TR_d = 7$. In step 1, a TR of $dwm_0$ (first nanowire) is conducted. $S_0$, which is XOR of $a_0 \ldots e_0$, (5-bit number) is computed by the PIM block, which is the blue bits between the **AP0** and **AP1** access ports. Simultaneously, carry $C_0$, is computed and sent to the right to the driver (**AP1**) for $dwm_1$ (second nanowire) shown in orange, and carry prime $C'_0$ is sent to the left of the driver **AP0** for $dwm_2$ (third nanowire), shown in green in Figure 6. In step 2, a similar set of steps occurs, except the operations include $C_0$ in addition to $a_1 \ldots e_1$. Then, in step 3, TR is conducted over $C'_0, a_2 \ldots e_2, C_1$, seven total elements. We will continue to compute $S, C$, and $C'$ for every nanowire until the third last nanowire (509) is reached. In the general case, for step k+1 (i.e., $dwm_k$), TR is conducted over $C'_{k-2}$, $a_k \ldots e_k$, $C_{k-1}$ with $S_k$ written to $port_L$ of $dwm_k$, $C_k$ written to $port_R$ of $dwm_{k+1}$ and $C'_k$ written to $port_L$ of $dwm_{k+2}$. Figure 6 is an example of addition for $(TR_d - 2)$ numbers of operands placed in between the access points.

(iv) *"MULT"* represents the arithmetic "multiplication" operation $A * B$. A foundational method to compute $A * B$ is to sum A, B times; e.g., for $B = 3$, $A * 3$ can be computed as $A + A + A$. Thus, we can perform multiplication by making several additions. Even with a 5 operand add, this method can quickly require many steps. Consider 9*A*: This can be computed by computing 5*A* in one step and then computing $5A + A + A + A + A$ in a second step. This method could be improved by generating 5*A* in one additional step, then replicating 5*A* and summing to compute 25*A*, and so on, but this clearly scales poorly. One method to accelerate this process is to shift the copies of A to quickly achieve the precise partial products that, when summed, produce the desired product. We have reserved our last DBC $\mathbf{DBC}_{15}$ for the purpose of shifting copies of A and executing *carry, carry prime,* and *add* operations on the shifted copies of A. This is the reduction process, as shown in Figure 7. One example of "multiplication" is shown in Figure 8. Here, we multiply $0 \times$ FE and $0 \times 1$F to generate $0 \times 1{,}588$.

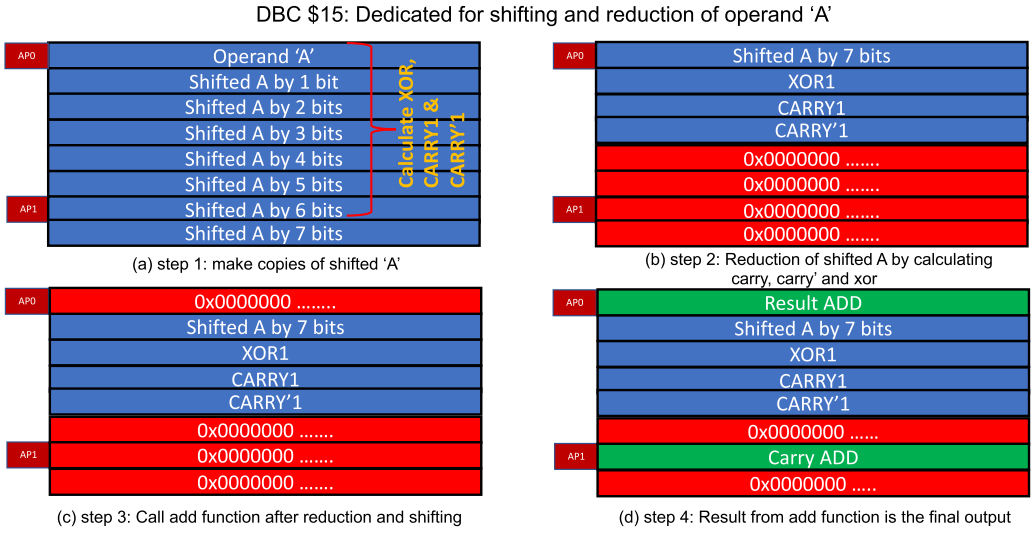



Fig. 7. Multiplication.

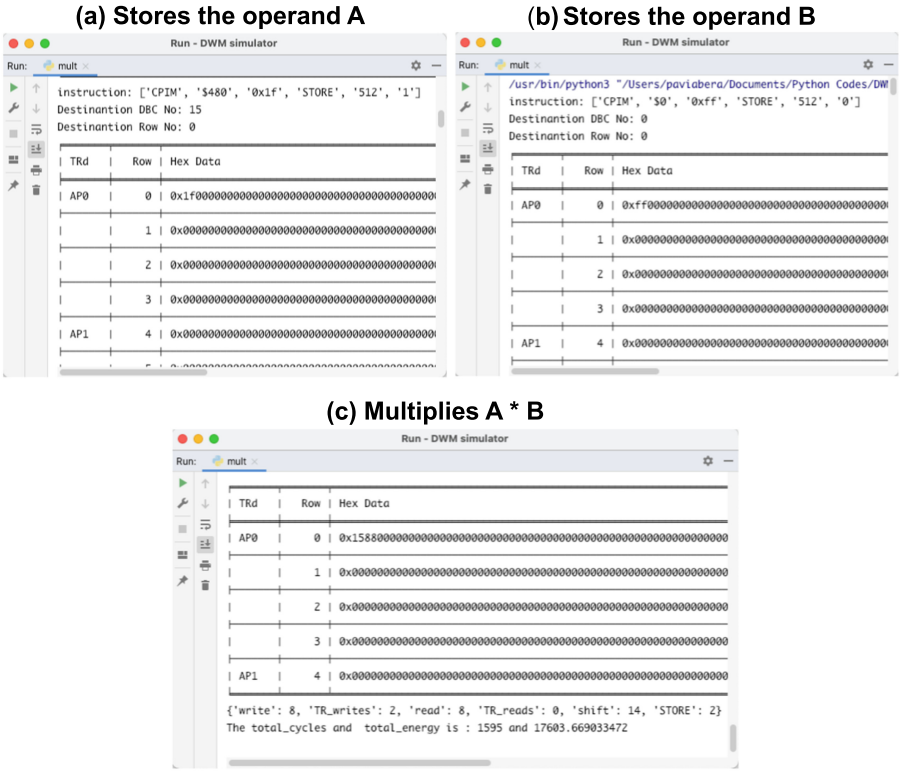



Fig. 8. Snapshot of **SPIM**ulator tool while performing multiplication of two 8-bit operands.

### 3.4 Fault Modeling:

Recalling that DWM retains the high endurance and other positive qualities of STT-MRAM, such as low energy consumption, it increases density by requiring the device to shift data with the nanowire to be aligned with the access points. However, DWM is susceptible to shifting faults, which are shifts in the position of domain walls within the memory cells. Shifting faults in DWM refer to the unintended movement of magnetization between domain walls within the memory cells, such as shifting too many to too few domains during a shift operation. Several factors contribute to the occurrence of shifting faults in DWM. Thermal effects, including temperature variations, fluctuations in the power network, and fluctuations in the material's magnetic properties, can impact domain-wall motion. Additionally, external magnetic fields, whether intentional or environmental, can influence domain-wall motion. Moreover, material defects, such as impurities or irregularities, can act as nucleation sites for domain wall movement. Manufacturing variations in cell dimensions or magnetic properties can also contribute to shifting faults and cause domain walls to remain pinned.

Shifting faults in DWM are persistent and can result in incorrect accesses and data corruption or loss. When reading or writing data, the memory controller accurately positions domain walls to determine the stored information. Corrective shifts and even scrubbing may be required to restore the data if there is incorrect domain-wall movement.

Moreover, frequent shifting faults can degrade both the reliability and the performance of DWM devices due to the overhead of fault-tolerance actions, which affects the overall speed and efficiency of these systems. Thus, addressing shifting faults is crucial to ensure the reliable operation of DWM devices. Reference [14] presents promising new fault-tolerance schemes for DWMs, such that they can detect and correct a wide range of shifting faults without significantly increasing the power consumption or area overhead of DWMs. PIETT detected and corrected shifting faults at runtime by employing "Parallel Independent Error and Transverse access Tapes." **SPIM**ulator implements a PIETT-inspired fault modeling scheme specifically designed to address shifting faults, misalignment, and pinning errors in **Domain-Wall Memories (DWM)**. This integration of a fault-handling mechanism underscores SPIMulator's commitment to enhancing user experience by ensuring the reliability and accuracy of simulations. Also, this approach aligns with the practical usability of SPIMulator and adds value to the overall simulation framework.

(i) *Parity checking:* Parity checking is a simple and efficient way to detect if a single-bit flip error has occurred. A parity bit must be added to each DWM cell/nanowire. The parity bit is calculated by XORing the bits in the cell, which can be accomplished through TRs. Checking the parity bit can also be done to determine a single-bit error has occurred. More sophisticated codes of these parity bits can be stored to reduce storage, such as the DECC process proposed in PIETT [14].

(ii) *Corrective Shifts:* By encoding the overhead bits, it is possible to detect over- and undershifting in the system [14]. In this instance the position can be corrected through "corrective shifts." **SPIM**ulator allows user-inserted instructions or stochastic misalignment faults to be injected into the simulation to add performance and energy overheads for these corrective shifts.

(iii) *Error correction:* To protect the data stored in the memory, error-correcting codes may be employed to correct single- or multiple-bit errors. **SPIM**ultaor allows the extension of the DBCs to include extra nanowires to store parity bits to enable user selected protection modes for protecting memory accesses using Hamming codes, BCH codes, Reed-Solomon codes, among others. If these errors occur due to a pinning fault, i.e., when part of the nanowire shifts a different number of positions than the rest, then this can permit scrubbing. Need for scrubbing can be detected through pinning fault detection circuits as described in PIETT [14].

Table 3. Performance Parameters and Instruction Counts

| Algorithm | Write | TW | Read | TR | Shifts | Store | Energy (pJ) | Cycle |
|---|---|---|---|---|---|---|---|---|
| Bitmap | 15 | 2 | 4 | 3 | 26 | 10 | 8,354.26 | 683 |
| Dot Product | 380 | 48 | 188 | 120 | 295 | 16 | 265,270.34 | 26,426 |
| AES | 267 | 101 | 294 | 122 | 1,767 | 4 | 900,482.85 | 76,608 |

An example of a manually inserted shifting fault overheads is: "*cpim $32 $0 CS 511 '0'*," in which the *"CS"* operation is executed from source address *"src"* location to the TR size for all 512 nanowires. The result of the shifting fault operation is then stored in the local buffer and overwritten at the destination address *"dst."* The operation *"CS"* stands for corrective shifts. Such operations insert overhead but do not actually shift the nanowires.

## 4 RESULTS

In this section, we will verify the mutability and versatility of the **SPIM**ulator by demonstrating the computing in memory (PIM components) through end-to-end execution of the following algorithms: `Bitmap indices`, `Dot product`, and **AES (advanced encryption standard block cipher)**. These three case studies were selected as they are frequently used processes in data indexing, signal processing, machine learning, and data privacy applications. They also showcase the diverse algorithmic in-memory execution capabilities of the **SPIM**ulator. We have also conducted quantitative analysis in our three case studies with variable $TR_d \in \{4, 5, 7\}$ for verification of user-assigned TR distance.

We evaluate the robustness of our PIM block in **SPIM**ultor by calculating the *writes*, *reads*, and *shifts* required to execute the algorithms, end-to-end, in memory. It is important to note that the *writes*, *reads*, and *shifts* counts in Table 3 may not reflect the minimal scenario for the algorithm implementation. **SPIM**ulator demonstrates the workability of the algorithm, however, it is possible the algorithm can be further optimized using approaches such as pipelining or subarray/bank-level parallelism that can be introduced.

### 4.1 Case 1: Bitmap Indices

Bitmap indices is a search algorithm where data is stored in a tabular format, and queries are executed via multi-operand bit-wise operations. This type of indexing is used for large databases with low cardinality columns that are used frequently in searches. A bitmap index is a binary number whose length equals the number of variables entered into the table. Figure 9(a) is an example where gender has a binary value, males as 0 and females as 1, and it also stores their last logged-in information. Each person is represented by a column, and the corresponding week logged in will be represented by 1. So, in Figure 9(a), **Person 1** is a male and logged in the current week. The case study uses data from Figure 9(b), which is the hex equivalent generated in Figure 9(a).

To arrange data in memory, it is best to allow each nanowire to represent an entry in the table (Person) and each row to represent a condition. Using bitmap indexing, searching the list will require performing different logical operations on specific rows to meet the search criteria. For example, searching how many males have logged in within the past two weeks will require following logical operations.

— "STORE" table in Figure 9(b) in DBC 0.
— "COPY" 0th, 1st, and 2nd weeks of data to DBC 1.
— "ORing" together 0th, 1st, and 2nd weeks will give all people that have logged in within two weeks.
— Save "OR" results in DBC 3.

| Person | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Gender | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 Weeks | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 Week | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 Weeks | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 3 Weeks | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 Weeks | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 Weeks | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

(a) Bitmap indices of Data

| Gender | 0x2D |
|---|---|
| 0 Weeks | 0x81 |
| 1 Week | 0x22 |
| 2 Weeks | 0x08 |
| 3 Weeks | 0x10 |
| 4 Weeks | 0x40 |
| 5 Weeks | 0x04 |

(b) Hex equivalent of Table 9(a)

Fig. 9. Case 1: Bitmap Indices. Figure 9(a) shows a set of data representing the gender and weeks since last logged in. Figure 9(b) is the hex equivalent of Table 9(a), which is stored into the memory of **SPIM**ulator.

.

— "COPY" Gender row to DBC 2.
— "NOTing" Gender to calculate $\overline{Gender}$ to make *male* a required search condition.
— Save $\overline{Gender}$ in DBC 3.
— "STORE" 0XFF, i.e., 1's in DBC 3 for AND operation as default DBC state is 0.
— "ANDing" the result of OR and $\overline{Gender}$ to generate each entry that meets the criteria of the number of males logged in within the past two weeks.
— Save "AND" result in DBC 2.

Figure 10 is the instruction set for searching how many males have logged in within the past two weeks. Figure 11 illustrates the execution by the **SPIM**ulator in four steps. The instructions for this search operation have been designed to minimize shift operations by utilizing the different DBCs, thus reducing the overhead cost from shifts. Using a unique DBC for each operation, the energy and cycles used for shifts can be significantly reduced. For example, in Figure 11, the bitmap indices are stored in DBC 0, the "OR" operations are done in DBC 1, the "NOT" operations are done in DBC 2, and the "AND" operations are done in DBC 3. This is because the TR access port will not be required to align with the data every time a logical operation takes place. Note that for operations like "AND," rows between access ports not containing operands must be padded with 1's "STORED" to produce the right result. The default state of every row in the memory is zeroed. Therefore, we have stored 0xFF in between the TR access ports before the "AND" instruction. Also, the "NOR" operation is our "NOT" equivalent in **SPIM**ulator.

To illustrate how reliability functions in **SPIM**ulator note lines 4–5 of Figure 10. This shows how the instruction flow can be modified if a fault occurs in the system. In this example, a shifting fault in one of the nanowires occurs as the shifter is trying to shift from address 0x2D ($45) to 0x22 $34. Thus, it includes an overhead for corrective shifts to correct this fault.

### 4.2 Case 2: Multiplication of Matrices

Matrix multiplication is one of the most widely used mathematical operations in the scientific community. Especially after the proliferation of machine learning such as deep learning, matrix multiplication has become an integral part of the basic computation. Therefore, being able to optimize time and computation speed is of great significance. In-memory matrix multiplication computation can help alleviate resource consumption and accelerate many machine-learning algorithms. Equation (3) shows how to obtain the result for the multiplication of two $2 \times 2$ matrices.

$$MatrixMult = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{bmatrix} \tag{3}$$

In this case study, the $2 \times 2$ matrix multiply will be computed entirely in memory. The results of this case will validate the addition and multiplication capabilities of **SPIM**ulator. Figure 12 is the instruction set for two $2 \times 2$ matrix multiplication. We have "STORED" $\left(\begin{smallmatrix} 0xFF & 0x0F \\ 0xAB & 0x1A \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 0x1F & 0x11 \\ 0xF1 & 0x01 \end{smallmatrix}\right)$

```
# Step 1: Storing Bitmap Indices
CPIM $12 0x2D STORE 512 0 #gender
CPIM $13 0x81 STORE 512 0 #week 0
# during simulation a shift fault occurs shifting from 0x2D to 0x22
CPIM $45 $34 CS 511 0 # auto inserted instruction to employ corrective shifts
CPIM $14 0x22 STORE 512 0 #week 1
CPIM $15 0x08 STORE 512 0 #week 2
CPIM $16 0x10 STORE 512 0 #week 3
CPIM $17 0x40 STORE 512 0 #week 4
CPIM $18 0x04 STORE 512 0 #week 5
# Step 2: Copy to DBC 1 and OR the results
# DBC 1 is reset to 0's to enable fewer operand ORs
CPIM $32 $13 COPY 512 0
CPIM $32 $14 COPY 512 1
CPIM $32 $15 COPY 512 1
CPIM $96 $32 OR 512 0 #save OR result in DBC 3
# DBC 3 is preset 1's to enable fewer operand ANDs
CPIM $97 0xFF STORE 512 0
CPIM $98 0xFF STORE 512 0
CPIM $98 0xFF STORE 512 0
# Bitmap
# Search condition (Example: how many males have logged in within the past two weeks?)
# Steps 3 & 4: Copy to DBC 3, NOT (~NOR) extract males Gender AND with logins
CPIM $64 $12 COPY 512 0
CPIM $100 $64 NOR 512 0 # save NOR/NOT result in DBC3
CPIM $64 $96 and 512 0 # save AND result in DBC2
```

Fig. 10. Bitmap indices CPI assembly example.

and then computed the products of $a*w$ ($0xFF*0x1F$) and $b*y$ ($0x0F*0xF1$). Then they are added together $(a*w)+(b*y)$ to obtain the first element in the result matrix. These steps are repeated to compute the other three elements $[(a*x)+(b*z)]$, $[(c*w)+(d*y)]$, and $[(c*x)+(d*z)]$) to obtain the final result. Figure 13 shows the final four elements of Equation (3) computed by the **SPIM**ulator.

### 4.3 Case 3: Advance Encryption Standard

**Advance Encryption Standard (AES)** is a standard encryption method standardized in 2001 by **NIST (U.S National Institute of Standards and Technology)**. AES is a block cipher algorithm that can use a **128, 192, or 256** bit size cipher key, depending on the required security concerns to encrypt 128 bits of input data at a time [5]. Depending on the key size, 10, 12, or 14 rounds of permutation, combination and substitution are performed to encrypt the input data. These processes are known as SubByte, ShiftRows, MixColumns, and AddRoundKey, as shown in Figure 14. In this case study, we have designed a way to perform AES using the PIM architecture of Reference [13] and the instruction count was also analyzed in Table 3. It is important to note that these values do not reflect the best performance capabilities of the **SPIM**ulator.

Before we explain the SubByte, ShiftRows, MixColumns, and AddRoundKey, it is important to determine how the data will be placed and operated upon in the Domain wall memory. Traditionally, AES data is shown in matrix format, and the above-mentioned steps are performed sequentially for the required number of rounds. However, since **SPIM**ulator performs operations vertically, data is stored across different nanowires for ease of functioning, as shown in Figure 15. This has a positive side effect of reduced shifts required for access of new rows for the data. With data placement explained, the next sections will discuss how each process of AES is performed for the first round of operations.

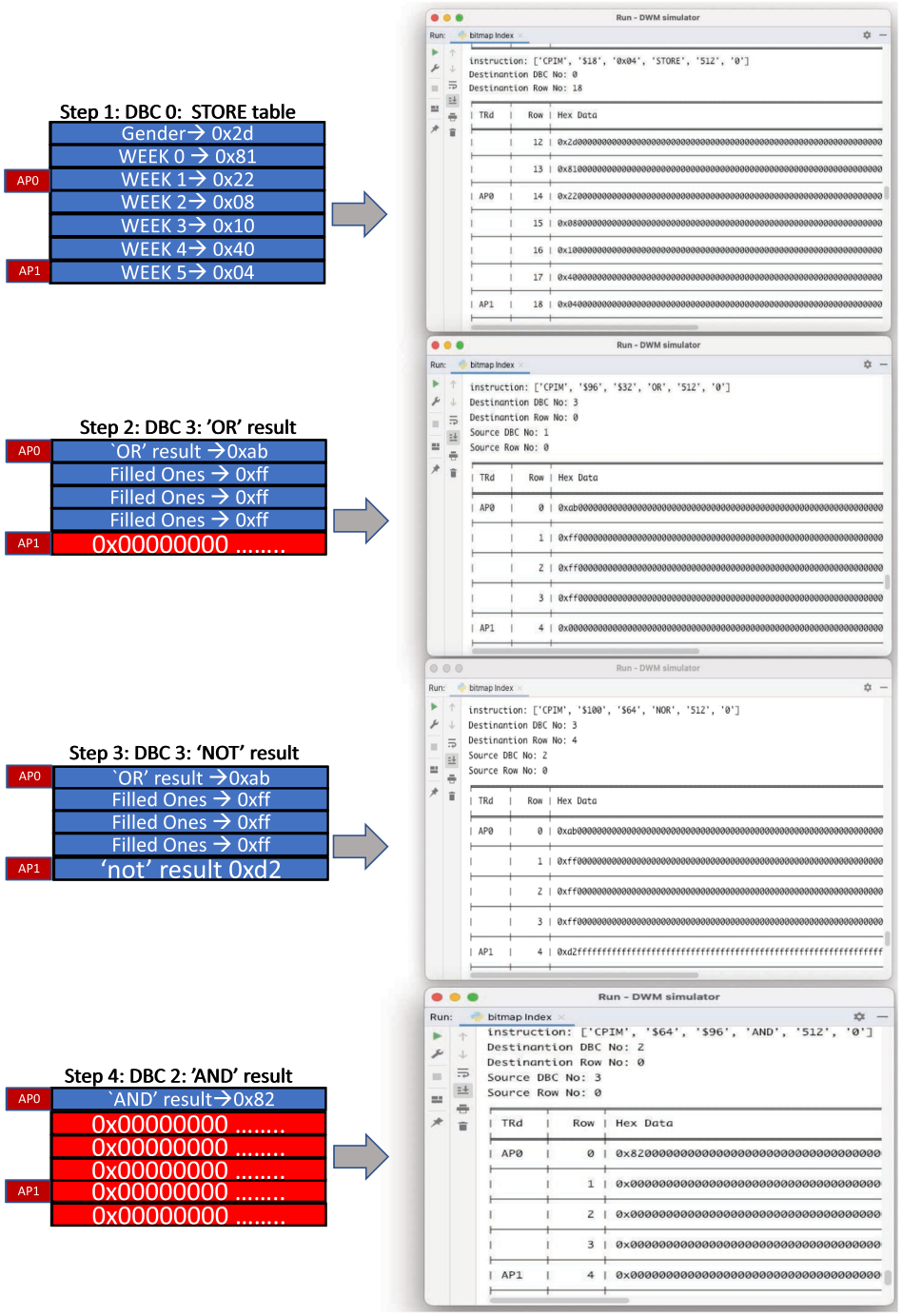



Fig. 11. Logic operational steps to search for males who logged in within the past two weeks. **Step 1:** stores the data in hex format, **Step 2:** OR's the 0th, 1st, and 2nd weeks and stores it in DBC 3, **Step 3:** NOTs the gender and stores in DBC 3, **Step 4:** ANDs DBC 3 and stores result in DBC 2.

```
# Use last DBC $480 to $511 to shift the 'A' operand
CPIM $0 0xFF STORE 512 0 # a
CPIM $480 0X1F STORE 512 0 # w
CPIM $32 $0 MULT 8 0 # a*w
CPIM $0 0x0F STORE 512 0 # b
CPIM $480 0xF1 STORE 512 0 # y
CPIM $33 $0 MULT 8 0 # b*y
CPIM $64 $32 ADD 8 0 # a*w + b*y

CPIM $0 0xFF STORE 512 0 # a
CPIM $480 0x11 STORE 512 0 # x
CPIM $32 $0 MULT 8 0 # a*x
CPIM $0 0x0F STORE 512 0 # b
CPIM $480 0x01 STORE 512 0 # z
CPIM $33 $0 MULT 8 0 # b*z
CPIM $65 $32 ADD 8 0 # a*x + b*z

CPIM $0 0xAB STORE 512 0 # c
CPIM $480 0x1F STORE 512 0 # w
CPIM $32 $0 MULT 8 0 # a*w
CPIM $0 0x1A STORE 512 0 # d
CPIM $480 0xF1 STORE 512 0 # y
CPIM $33 $0 MULT 8 0 # d*y
CPIM $66 $32 ADD 8 0 # c*w + d*y

CPIM $0 0xFF STORE 512 0 # c
CPIM $480 0x1F STORE 512 0 # x
CPIM $32 $0 MULT 8 0 # c*x
CPIM $0 0x0F STORE 512 0 # d
CPIM $480 0xF1 STORE 512 0 # z
CPIM $33 $0 MULT 8 0 # d*z
CPIM $66 $32 ADD 8 0 # c*x + d*z
```

Fig. 12. Instruction flow for $2 \times 2$ matrices dot product.

**SubByte** is the first process of AES where a lookup table known as the S-box (Table 4) is implemented. In this process, the input data is transformed into new values using the S-box lookup table. This process introduces non-linearity to the encryption algorithm. The S-box is a multiplicative inverse table derived from the Galois field. This look-up table has been stored in memory, and then the data is substituted from this table. This is done by storing the S-box in memory and using the *"COPY"* instruction to substitute the values, as shown in Figure 16. When performing the SubByte, each byte in a row is used as an address. This will be done repeatedly until all bytes have been substituted, at which point the results will be put together via an OR operation. The amount of space required for this can be cut in half by instead having alternating copies of two adjacent SubBytes stored in the addresses. Accessing the desired SubByte can be done by either copying the row as is or logically shifting it right by eight. But for simplicity reasons, the S-box in this case study is stored in 256 regular rows across consecutive DBCs. Due to the **SPIM**ulators continuous addressing technique, it is easy to substitute the required bytes from the stored s-box.

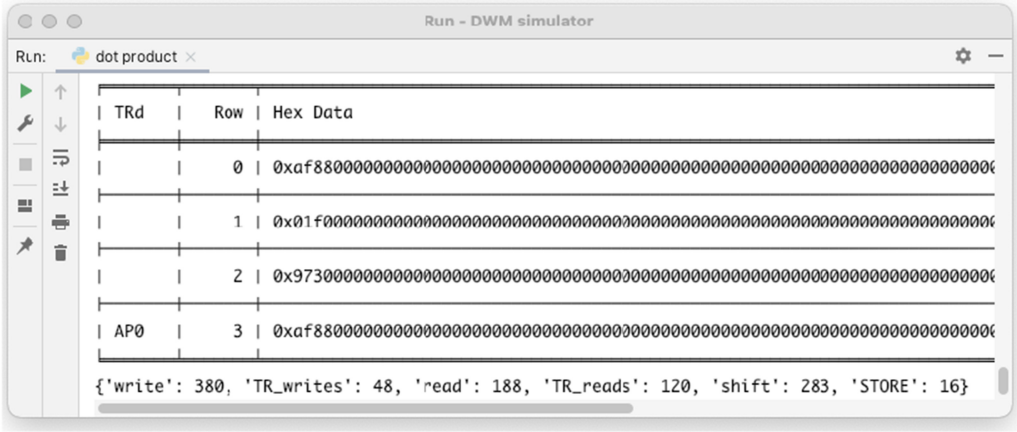



Fig. 13. **SPIM**ulator output of $\begin{pmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{pmatrix}$.

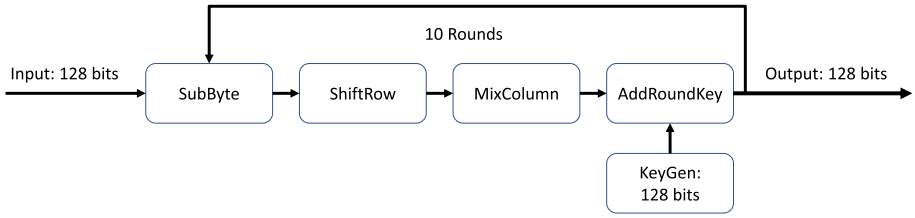



Fig. 14. AES flow chart.

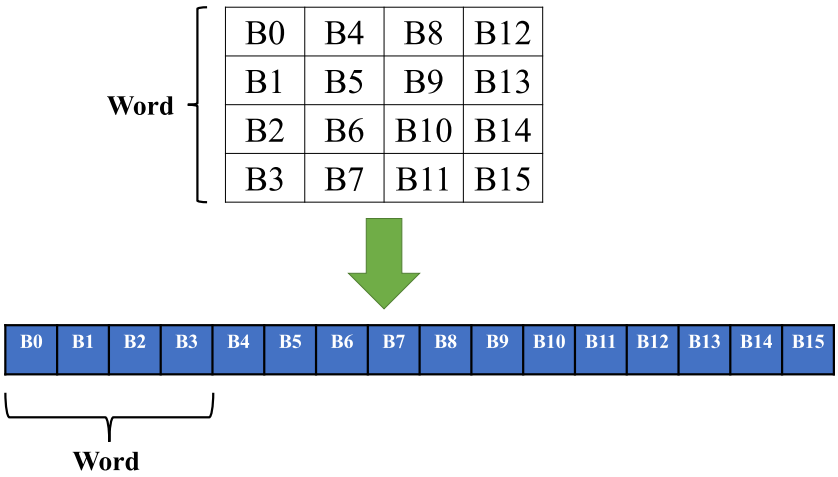
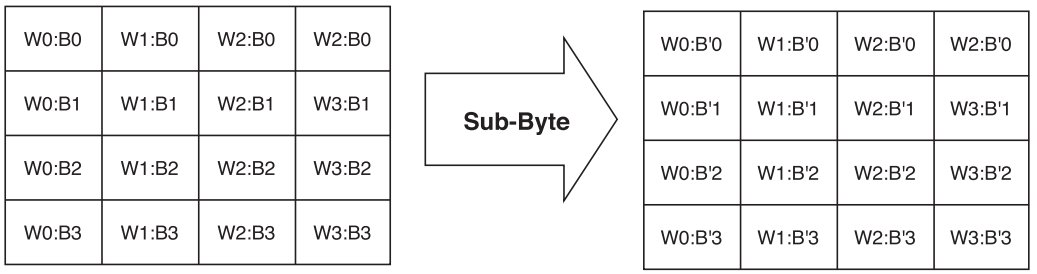



Fig. 15. Data placement in DBC.

**ShiftRows** is the next process of the AES algorithm and is a relatively simple step to calculate. In terms of the matrix, it is a circular shift of each row by 0, 1, 2, and 3 columns, respectively. Using Figure 15 as a reference, this can be translated into circular shifting each byte in words by 0, 32, 64, and 96 bits based on its position. This step can be done in memory via logical shifts, as discussed in Section 3.3. For SPIMulator, these steps have to be broken into pieces that are all OR'd together:




| | |
|---|---|
| ROW 0 | S_BOX (0000) = 0x63 |
| ROW 1 | S_BOX (0001) = 0x7c |
| ROW 2 | S_BOX (0002) = 0x77 |
| ROW 3 | S_BOX (0003) = 0x7b |
| ----- | |
| ----- | |
| ----- | |
| ROW 256 | S_BOX (0ff0) = 0x16 |

Fig. 16. S-box in memory.

Table 4. AES SubByte Table (S-box)

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0a | 0b | 0c | 0d | 0e | 0f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| 10 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | c0 |
| 20 | b7 | fd | 93 | 26 | 36 | 3f | f7 | cc | 34 | a5 | e5 | f1 | 71 | d8 | 31 | 15 |
| 30 | 04 | c7 | 23 | c3 | 18 | 96 | 05 | 9a | 07 | 12 | 80 | e2 | eb | 27 | b2 | 75 |
| 40 | 09 | 83 | 2c | 1a | 1b | 6e | 5a | a0 | 52 | 3b | d6 | b3 | 29 | e3 | 2f | 84 |
| 50 | 53 | d1 | 00 | ed | 20 | fc | b1 | 5b | 6a | cb | be | 39 | 4a | 4c | 58 | cf |
| 60 | d0 | ef | aa | fb | 43 | 4d | 33 | 85 | 45 | f9 | 02 | 7f | 50 | 3c | 9f | a8 |
| 70 | 51 | a3 | 40 | 8f | 92 | 9d | 38 | f5 | bc | b6 | da | 21 | 10 | ff | f3 | d2 |
| 80 | cd | 0c | 13 | ec | 5f | 97 | 44 | 17 | c4 | a7 | 7e | 3d | 64 | 5d | 19 | 73 |
| 90 | 60 | 81 | 4f | dc | 22 | 2a | 90 | 88 | 46 | ee | b8 | 14 | de | 5e | 0b | db |
| a0 | e0 | 32 | 3a | 0a | 49 | 06 | 24 | 5c | c2 | d3 | ac | 62 | 91 | 95 | e4 | 79 |
| b0 | e7 | c8 | 37 | 6d | 8d | d5 | 4e | a9 | 6c | 56 | f4 | ea | 65 | 7a | ae | 08 |
| c0 | ba | 78 | 25 | 2e | 1c | a6 | b4 | c6 | e8 | dd | 74 | 1f | 4b | bd | 8b | 8a |
| d0 | 70 | 3e | b5 | 66 | 48 | 03 | f6 | 0e | 61 | 35 | 57 | b9 | 86 | c1 | 1d | 9e |
| e0 | e1 | f8 | 98 | 11 | 69 | d9 | 8e | 94 | 9b | 1e | 87 | e9 | ce | 55 | 28 | df |
| f0 | 8c | a1 | 89 | 0d | bf | e6 | 42 | 68 | 41 | 99 | 2d | 0f | b0 | 54 | bb | 16 |

a row of the first bytes, two rows of the second bytes shifted left and right by 32 bits, two rows of the third bytes shifted left and right by 64 bits, and two rows of the last bytes shifted left and right by 96. Each of these rows will need to be masked to place to values in the correct position. An example of this is shown in Figure 17. Since the first byte of each word is not shifted, they start in the correct place and are highlighted in green. The whole line is then shifted right by 4 bytes at a time (1 word) and masked to orient a portion of the words. The remaining words are placed by then shifting the line left 4 bytes and masking off the excess. The final correct result should be obtained by OR'ing the individual masked pieces together.

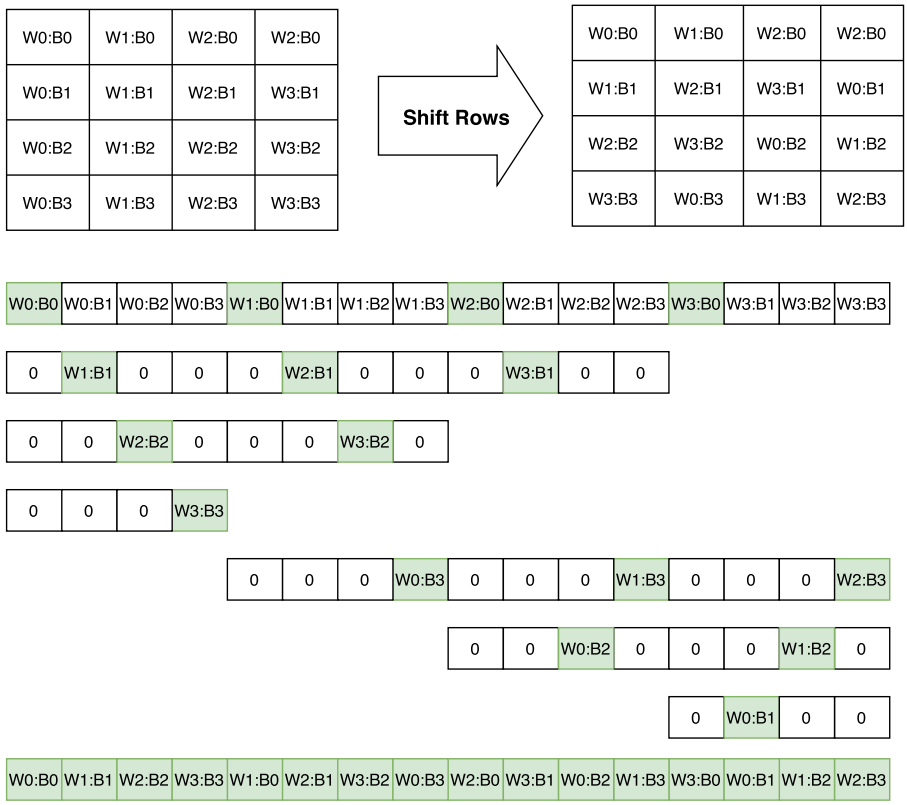



Fig. 17. Shift rows demonstration in memory: Each green box represents when a byte is in its desired position in memory. When the first row is masked to obtain the first byte of each word, the respective shifted and masked values can all be OR'd together to get the final result.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$
$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$
$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$
$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

Fig. 18. MixColumn from FIPS 197 [5].

**MixColumns** is the combinational step after ShiftRows, and it involves mixing data around via matrix multiplication so each input affects each output. Since this operation is done in a Galois Field, traditional multiplication is not required. Figure 18 from FIPS 197 [5] gives further detail about this operation.

Notice that what would usually be an addition in matrix multiplication is, in fact, an XOR operation. For multiplication, if the number is less than 127 (no 1 at the most significant bit), then multiplication by two is a logical shift and multiplication by 3 is a logical shift and an XOR with the operand. If any operand is greater than 127 prior to multiplying, then an additional XOR with the polynomial of the Galois field 0x1B. For the tool, the overflow is handled by creating the 0x1B constant via the first bit of each byte, as shown in Figure 19. If the MSB is 1 (B0), then the 0x1B constant is created, otherwise it will simply create 0x00. By using this overflow handler and shifts,

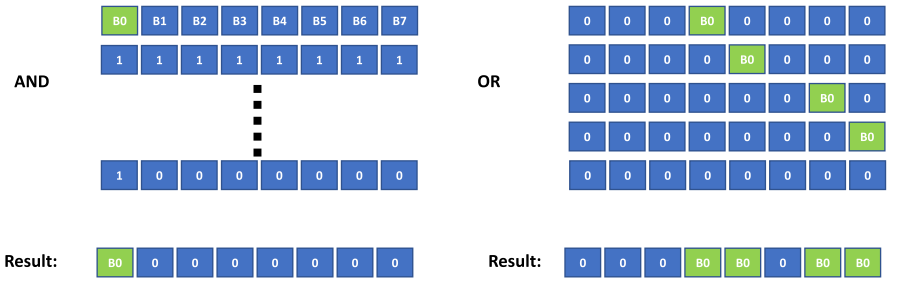



Fig. 19. Overflow demonstration.

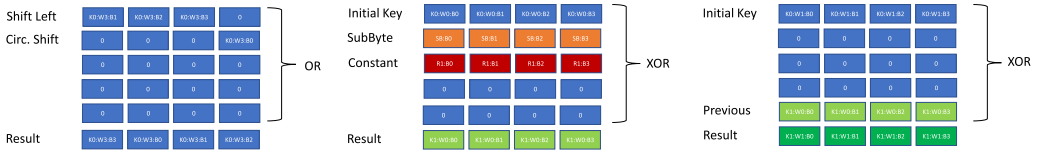



Fig. 20. Key generation in memory.

multiplication by both 2 and 3 can be handled, and the final result is obtained by XORing each word with its respective values.

**AddRoundKey** is the last step in the algorithm. For this step, a round key must first be generated. Starting with an initial round key, the next round key is made with word (4 bytes) at a time. The first word is the most involved, as it requires the XOR of a round constant, the first word of the old key, and a final value. This final value is generated by circular shifting the last word of the current key by 1 byte and then performing SubByte upon it. An equation for the first word is as follows:

$$k_{new}[n] = SubByte(k_{old}[4] >> 8) \oplus r_i \oplus k_{old}[0],$$

where $k_{new}$ is the new round key, $k_{old}$ is the previous round key, and $r_i$ is the round constant. Both the methods for circular shifting and SubByte were shown to be executable in memory, so this step can be done in the tool. The final three words are generated by performing XOR with the prior word and the word in the old key's position, as seen in the following equation:

$$k_{new}[n] = k_{old}[n] \oplus k_{new}[n-1].$$

After generating the key, executing AddRoundKey can be done with an XOR of the key with the data. In memory, this can be executed as shown in Figure 20. Before doing the SubByte, the data is shifted and two copies are made, one with the first half of the circular shift and one with their remaining bit. These pieces are OR'd together then substituted. After this, the initial key and round constant are XOR'd with the SubByte result to get the first new word. Each additional word can be obtained by logical shifting the previous result in to align with the next word and performing XOR. Each calculated word can be combined via an OR to get the final answer. The generated key is verified and the instruction set for AES with the **SPIM**ulator can be found in the github repository at https://github.com/Pitt-JonesLab/DWMsimulator

### 4.4 SPIMulator with Other Racetrack Memory PIM Proposals

In this section, we demonstrate how **SPIM**ulator can be customized and leveraged to implement other DWM PIM approaches, such as the XOR operation implemented using **giant magnetoresistance (GMR)** [31]. **SPIM**ulator's framework provides a flexible platform for customizing the

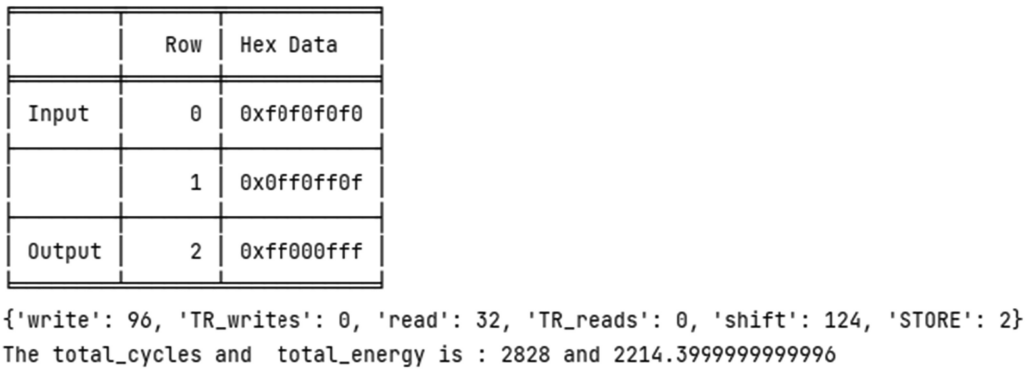



Fig. 21. XOR operation from Reference [31].

behavior of domain-wall memory in a processing-in-memory setting. In accordance with the principles presented in Reference [31], we first adapt **SPIM**ulator's memory architecture to emulate their non-volatile domain-wall memory. **SPIM**ulator allows for the definition of the key parameters and characteristics of domain-wall memory organization, such as the width of the memory and the number of nanowires. For example, the display is modified to show two neighboring nanowires as well as a third output nanowire. To validate the effectiveness of the XOR operation implemented within our customized **SPIM**ulator, we analyze the performance metrics, such as cycle count and energy consumption, and compare them to traditional computational approaches. Figure 21 illustrates the implementation of XOR after **SPIM**ulator was tailored based on the principles of non-volatile domain-wall memory PIM.

## 5 CONCLUSION

In summary, **SPIM**ulator is a simulator for Spintronic **Processing in Memory (PIM)** that models PIM architecture in Racetrack memory. The simulator provides real-time performance estimates such as cycle count and energy consumption and is capable of simulating the polymorphic gate properties of Racetrack memory. In this article, **SPIM**ulator has been used to model, validate, and estimate the performances of various applications including encryption, dot product, and database search algorithms. This includes functional simulation to ensure the algorithm is programmed correctly. **SPIM**ulator can also assess the potential benefits of using Spintronic PIM technology to alleviate the memory bottleneck of the Von Neumann computing model and to improve latency and energy cost associated with in-memory processing. We also demonstrate how additional fault tolerance instructions can be introduced to correct faults both through manual fault injection and faults stochastically generated by the simulator to mimic the impact of reliability and error correction schemes. In future work, it would be desirable to integrate a memory tool like RTSim with **SPIM**ulator to implement detailed cycle-level simulation and massively parallel (e.g., subarray-level) implementation and estimation.

## REFERENCES

[1] C. Augustine, A. Rachowdhur, B. Behin-Aein, S. Srinivasan, J. Tschanz, Vivek K. De, and K. Ro. 2011. Numerical analysis of domain wall propagation for dense memory arrays. In *IEEE International Electron Devices Meeting (IEDM'11)*. IEEE, 17–6.

[2] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillon, and S. S. P. Parkin. 2020. Magnetic racetrack memory: From physics to the cusp of applications within a decade. *Proc. IEEE* 108, 8 (2020), 1303–1321. DOI: https://doi.org/10.1109/JPROC.2020.2975719

[3] Y.-C. Chen, H. Li, and W. Zhang. 2012. A RRAM-based memory system and applications. In *the Non-volatile Memories Workshop*.

[4] Prayash Dutta, Albert Lee, Kang L. Wang, Alex K. Jones, and Sanjukta Bhanja. 2023. A multi-domain magneto tunnel junction for racetrack nanowire strips. *IEEE Trans. Nanotechnol.* 22 (2023), 581–583. DOI: https://doi.org/10.1109/TNANO.2023.3298920

[5] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. 2001. Advanced Encryption Standard (AES). DOI: https://doi.org/10.6028/NIST.FIPS.197

[6] Yiming Huai. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bull.* 18, 6 (2008), 33–40.

[7] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, and Jeronimo Castrillon. 2019. RTSim: A cycle-accurate simulator for racetrack memories. *IEEE Comput. Archit. Lett.* 18, 1 (Jan. 2019), 43–46. DOI: https://doi.org/10.1109/LCA.2019.2899306

[8] Asif Ali Khan, Sebastien Ollivier, Fazal Hameed, Jeronimo Castrillon, and Alex K. Jones. 2023. DownShift: Tuning shift reduction with reliability for racetrack memories. *IEEE Trans. Comput.* 72, 9 (2023), 1–14. DOI: https://doi.org/10.1109/TC.2023.3257509

[9] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart S. Parkin, and J. Castrillon. 2019. ShiftsReduce: Minimizing shifts in racetrack memory 4.0. *ACM Trans. Archit. Code Optim.* 16, 4, Article 56 (Dec. 2019), 23 pages. DOI: https://doi.org/10.1145/3372489

[10] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *53rd Annual Design Automation Conference*. 1–6.

[11] Bicheng Liu, Shouzhen Gu, Mingsong Chen, Wang Kang, Jingtong Hu, Qingfeng Zhuge, and Edwin H.-M. Sha. 2017. An efficient racetrack memory-based processing-in-memory architecture for convolutional neural networks. In *IEEE International Symposium on Parallel and Distributed Processing with Applications and IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC'17)*. IEEE, 383–390.

[12] Sally A. McKee. 2004. Reflections on the memory wall. In *1st Conference on Computing Frontiers (CF'04)*. Association for Computing Machinery, New York, NY, 162. DOI: https://doi.org/10.1145/977091.977115

[13] Sebastien Ollivier, Stephen Longofono, Prayash Dutta, Jingtong Hu, Sanjukta Bhanja, and Alex K. Jones. 2022. CORUSCANT: Fast efficient processing-in-racetrack memories. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*. 784–798. DOI: https://doi.org/10.1109/MICRO56248.2022.00060

[14] Sebastien Ollivier, Stephen Longofono, Prayash Dutta, Jingtong Hu, Sanjukta Bhanja, and Alex K. Jones. 2023. Toward comprehensive shifting fault tolerance for domain-wall memories with PIETT. *IEEE Trans. Comput.* 72, 4 (2023), 1095–1109. DOI: https://doi.org/10.1109/TC.2022.3188206

[15] Sébastien Ollivier, Xinyi Zhang, Yue Tang, Chayanika Choudhuri, Jingtong Hu, and Alex K. Jones. 2022. POD-RACING: Bulk-bitwise to floating-point compute in racetrack memory for machine learning at the edge. *IEEE Micro* 42 (2022). DOI: 10.1109/MM.2022.3195761

[16] Stuart Parkin and See-Hun Yang. 2015. Memory on the racetrack. *Nature Nanotechnol.* 10, 3 (2015), 195–198.

[17] Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. 2008. Magnetic domain-wall racetrack memory. *Science* 320, 5874 (Apr. 2008), 190–194.

[18] M. Poremba and Y. Xie. 2012. NVMain: An architectural-level main memory simulator for emerging non-volatile memories. In *IEEE Computer Society Annual Symposium on VLSI*. 392–397.

[19] M. Poremba, T. Zhang, and Y. Xie. 2015. NVMain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Comput. Archit. Lett.* 14, 2 (July 2015), 140–143.

[20] Kawsher Roxy, Sébastien Ollivier, Arifa Hoque, Stephen Longofono, Alex K. Jones, and Sanjukta Bhanja. 2020. A novel transverse read technique for domain-wall "racetrack" memories. *IEEE Trans. Nanotechnol.* 19 (2020), 648–652. DOI: https://doi.org/10.1109/TNANO.2020.3014091

[21] Vivek Seshadri, Donghuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. IEEE, 273–287.

[22] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsk. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Trans. Nanotechnol.* 15, 4 (2016), 635–650.

[23] Minh S. Q. Truong, Eric Chen, Deanyone Su, Liting Shen, Alexander Glass, L. Richard Carley, James A. Bain, and Saugata Ghose. 2021. RACER: Bit-pipelined processing using resistive memory. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Association for Computing Machinery, New York, NY, 100–116. DOI: https://doi.org/10.1145/3466752.3480071

[24] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2012. TapeCache: A high density, energy efficient cache based on domain wall memory. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, New York, NY, 185–190. DOI: https://doi.org/10.1145/2333660.2333707

[25] Rangharajan Venkatesan, Mrigank Sharad, Kaushik Roy, and Anand Raghunathan. 2013. DWM-TAPESTRI—An energy efficient all-spin cache using domain wall shift based writes. In *Conference on Design, Automation and Test in Europe*. EDA Consortium, 1825–1830.

[26] Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Comput. Sci. Eng.* 17, 2 (2015), 73–82. DOI: https://doi.org/10.1109/MCSE.2015.4

[27] Oreste Villa, Daniel R. Johnson, Mike O'connor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. 2014. Scaling the power wall: A path to exascale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 830–841. DOI: https://doi.org/10.1109/SC.2014.73

[28] Xin Xin, Youtao Zhang, and Jun Yang. 2020. ELP2IM: Efficient and low power bitwise operation processing in DRAM. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 303–314.

[29] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones. 2016. FusedCache: A naturally inclusive, racetrack memory, dual-level private cache. *IEEE Trans. Multi-Scale Comput. Syst.* 2, 2 (Apr. 2016), 69–82. DOI: https://doi.org/10.1109/TMSCS.2016.2536020

[30] Haifeng Xu, Yong Li, R. Melhem, and A. K. Jones. 2015. Multilane racetrack caches: Improving efficiency through compression and independent shifting. In *20th Asia and South Pacific Design Automation Conference*. 417–422. DOI: https://doi.org/10.1109/ASPDAC.2015.7059042

[31] Hao Yu, Yuhao Wang, Shuai Chen, Wei Fei, Chuliang Weng, Junfeng Zhao, and Zhulin Wei. 2014. Energy efficient in-memory machine learning for data intensive image-processing by non-volatile domain-wall memory. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. 191–196. DOI: https://doi.org/10.1109/ASPDAC.2014.6742888

[32] Masoud Zabihi, Zamshed Iqbal Chowdhur, Zhengang Zhao, Ula R. Karpuzcu, Jian-Ping Wang, and Sachin S. Sapatnekar. 2018. In-memory processing on the spintronic CRAM: From hardware design to application mapping. *IEEE Trans. Comput.* 68, 8 (2018), 1159–1173.

[33] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, , and W. Zhao. 2015. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Design Automation Conference*.

[34] Chao Zhang, Guangyu Sun, Xian Zhang, Weiqi Zhang, Weisheng Zhao, Tao Wang, Yun Liang, Yongpan Liu, Yu Wang, and Jiwu Shu. 2015. Hi-fi playback: Tolerating position errors in shift operations of racetrack memory. In *International Symposium on Computer Architecture (ISCA'15)*. 694–706. DOI: https://doi.org/10.1145/2749469.2750388

[35] Y. Zhang, W. Zhao, J. Klein, D. Ravelsona, and C. Chappert. 2012. Ultra-high density content addressable memory based on current induced domain wall motion in magnetic track. *IEEE Trans. Mag.* 48, 11 (Nov. 2012), 3219–3222. DOI: https://doi.org/10.1109/TMAG.2012.2198876

[36] Ping Zhou, Bo Zhao, Jun Ang, and Outao Zhang. 2009. A durable and energ efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News*, Vol. 37-3. ACM, 14–23.

Received 10 February 2023; revised 8 January 2024; accepted 21 January 2024