Testing Computation-in-Memory Architectures Based on Emerging Memories

Hamdioui, Said; Fieback, M.; Nagarajan, S.; Taouil, Mottaqiallah

DOI
10.1109/ITC44170.2019.9000117

Publication date
2019

Document Version
Accepted author manuscript

Published in
2019 IEEE International Test Conference (ITC)

Testing Computation-in-Memory Architectures Based on Emerging Memories

Said Hamdioui Moritz Fieback Surya Nagarajan Mottaqiallah Taouil
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628CD, Delft, The Netherlands
Email: S.Hamdioui@tudelft.nl

***Abstract*—Today's computing architectures and device technologies are becoming incapable of meeting the increasingly stringent demands on energy and performance posed by evolving applications. Therefore, alternative novel post-CMOS computing architectures are being explored. Some of these are Computation-in-Memory (CIM) architectures based on memristive devices; they integrate the processing units and the storage in the same physical location (i.e., the memory based on memristive devices). Due to their advanced manufacturing processes, use of new materials, and dual functionality, testing such chips requires specific schemes and therefore special attention. This paper describes the need for testing CIM architectures, proposes a systematic test approach, and shows the strong dependency of the test solutions on the nature of the architecture. All of these will be demonstrated using a design that is designed for computation-in-memory bit-wise logical operations.**

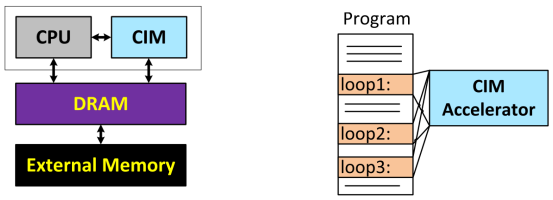
I. Introduction

In the past decades, the world has seen a phenomenal increase in computing performance, resulting in smaller, faster, and more energy efficient computers. However, today's computer architectures as well as the CMOS technology used to manufacture them are facing major challenges such as memory wall, power wall, leakage wall, and cost wall [1, 2]; these make them economically not attractive for many evolving applications which are extremely demanding, e.g., in terms of MOPs/Watt. Therefore, continuing with delivering sustainable benefits in the foreseeable future requires the exploration of alternative (unconventional) computing architectures that leverage novel post-CMOS device technologies such as memristive devices (e.g., resistive RAM (RRAM), phase change memory (PCM), spin-transfer-torque magnetic RAM (STT-MRAM)). One of these is a Computation-in-Memory (CIM) architecture based on memristive devices [3, 4]; it is based on integrating the processing units and the memory in the *same physical location*. As a consequence, it signicantly reduces the memory accesses and data movements while supporting massive parallelism, potentially resulting in orders of magnitude improvement in terms of energy and computing efciency [5, 6]. Many companies (e.g., IBM, ARM), research institutes (e.g., IMEC), and universities are investigating and demonstrating such an architecture [5]. There are still many issues that have to be solved in order to get this computer technology mature enough; examples are: endurance of the memristive devices, variability, complexity of the control units within the CIM core, etc [7, 8]. In addition, and like all other ICs, these CIM dies need to be tested for manufacturing defects, in order to guarantee sufcient outgoing product quality to the customer. The manufacturing process of memristive-based CIM cores involves additional steps and makes use of new materials [9], which may lead to new failure mechanisms. In addition, a CIM die acts both as a memory as well as a computing unit; and hence, it has to be tested for both functionalities.

To the best of our knowledge, this is the first paper to discuss the test needs for memristive device-based CIM architectures. Nevertheless, there is some published work on emerging memories (relying on memristive devices) upon which CIM architectures are based. Most of this work is based on modeling defects as linear resistors, injecting them in the memory netlist in order to perform circuit simulation and derive fault models, and thereafter test and design-for-testability (DfT) solutions [10–16]. However, recent work has demonstrated that using resistors to model defects in, for example, RRAM and STT-MRAM is not accurate enough due to the non-linearity of these devices [17, 18]. Inaccurate defect modeling may lead to non-realistic fault models, and hence, low-quality tests; this has enabled the development of Device-Aware-Test approach [19].

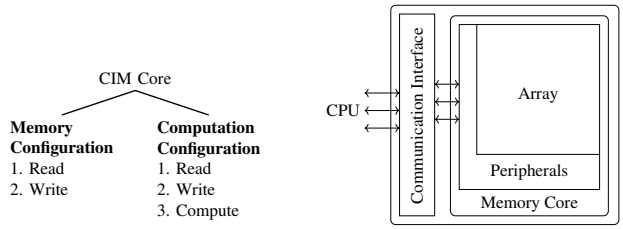
This paper addresses the test aspects of CIM architectures based on emerging memristive devices. It briefly discusses the feasibility of functional and structural testing. In addition, it provides a systematic and structural approach for testing, and it highlights the need for testing the CIM die for its two different functional configurations, once as a memory and once as a computing unit. The paper also shows the dependency of the test solutions on the nature of the CIM architecture itself by demonstrating this test approach for a CIM design that performs bit-wise logic operations.

The remainder of this paper is structured as follows. Section II presents background information on CIM, including a classification of different CIM architectures. Section III presents the proposed structural test approach for CIM architectures. Sections IV, V and VI apply this approach for a case study based on bit-wise logical CIM implementation. Section VII presents a discussion and conclusion.



(a) CIM core (b) CIM program

Fig. 1: CIM accelerator [5]



(a) CIM configurations (b) CIM block diagram

Fig. 2: CIM configurations and block diagram

II. Computation-in-Memory

In this section, we briefly present the concept of CIM architectures and classify them. Then, an implementation example for each class is given; one of them will be used as case study in this paper. However, in order to better understand these implementations, the working principles of an RRAM (used as a memristive device) will be be introduced first.

A. CIM Concept and Classification

The CIM architecture is based on integrating the processing units and the storage in the same physical memory location. A realistic implementation that many researchers are prototyping is shown in Fig. 1a [5, 20, 21]; the CIM core may consist of very dense memristive crossbar array and CMOS peripheral circuitry. The CIM die takes over the memory-intensive computation parts from the processor, thus significantly speeding up the execution and reducing the energy consumption by eliminating large amounts of data transfers. Fig. 1b illustrates a program that could be executed efciently on a CIM architecture; multiple loops can be executed within the CIM core while the other parts of the program can be executed on the conventional core. Each time a loop is invoked, the CPU sends a macro-instruction to the CIM core which decodes and executes it locally, and returns the final results.

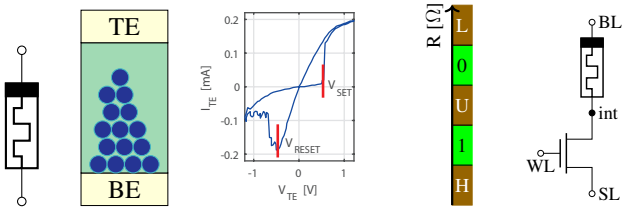
As already mentioned, computing in the CIM core takes actually place within the memory. Hence, the CIM core can operate in two different configurations: *memory* and *computation* configuration. Fig. 2a shows these configurations, as well as the operations that each configuration requires. Since the computation configuration also uses read and write operations, it is a superset of the memory configuration. Fig. 2b shows a block diagram of a CIM die. In addition to the memory core, it consists also of a communication interface. It is worth noting that computations in the CIM core take place *within* the memory core. Because a memory core consists of a *memory array* and *peripheral circuits*, and depending on *where* the result of the computation is produced, CIM architectures can be divided into two classes [22]:

- *CIM-Array (CIM-A):* In CIM-A, the computation result is produced within the array. Examples of such architectures are PLiM [23], ReVAMP [24], CIM device [25], etc. The CIM-A core typically requires a significant redesign of the memory array to support computing, as conventional memory cell layouts are typically optimized for storage functionality only.
- *CIM-Periphery (CIM-P):* In CIM-P, the computation result is produced within the peripheral circuitry. Examples of such architectures are PRIME [26], Pinatubo [27], CIM-Accelerator [28], etc. This architecture focuses on special circuits in the peripherals to realize, for example, bit-wise logic operations [27, 29], matrix-vector multiplication [6, 30], etc. Even though the computational results are produced in the peripheral circuits, the memory array could be a significant component in the computations. For example, to perform bit wise logic operations, multiple rows in the array need to be simultaneously activated.

As CIM performs operations within the memory core, at least part of the operands should be stored in the memory array. In other words, the operator being executed within the memory needs to have *all operands* stored in the array (as *resistive*) or *only* part of the operands is stored in the array and the other part is received via the memory port(s) (hence their logic values are *hybrid*, i.e., resistive and voltage). This results in four sub-classes: CIM-Ar, CIM-Ah, CIM-Pr and CIM-Ph; the additional letters 'r' and 'h' denote the nature of the inputs (operands), namely resistive and hybrid, respectively. An example of CIMP-Ah and CIM-Pr will be discussed in Subsection C and D.

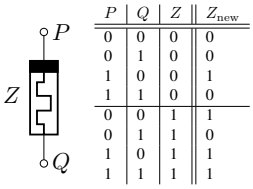
B. RRAM Device Technology

RRAM devices are one of the most popular memristive devices; they are non-volatile, two-terminal, non-linear devices that can switch their resistance [7, 31, 32]. The symbol to denote an RRAM device is shown in Fig. 3a, while the structure of the device is shown in Fig. 3b; the structure consists of a metallic oxide (green) that is stacked between two electrodes (yellow, top (TE) and bottom electrode (BE)) [7, 32]. When a voltage higher than the set threshold ($V_{\mathrm{TE}}>V_{\mathrm{SET}}$) is applied, some of the bonds between the metal and oxygen ions break. The oxygen ions are attracted to the positively charged electrode, leaving behind a chain of vacancies (blue circles). This chain, called the *conductive filament* (CF), can conduct a current. Even if the bias voltage is removed, the CF will remain intact, making this device *non-volatile*. On the contrary, when a negative voltage lower than the reset threshold ($V_{\mathrm{TE}}<V_{\mathrm{RESET}}$) is applied, some of the ions move back into the oxide, thus reducing the size of the CF. The shape of the CF determines the resistance of the device; larger CFs have a lower resistance. Fig. 3c shows the RRAM switching behavior with a current-voltage graph.



(a) Symbol (b) CF (c) I-V curve (d) Resistance (e) 1T-1R

Fig. 3: RRAM device



P	Q	Z	Z_{new}
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

Fig. 4: Majority logic

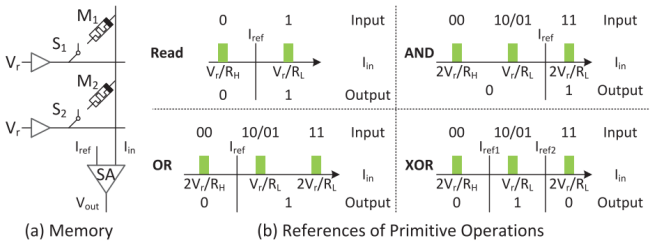
For memory applications, we distinguish two resistive states: the low resistive state (LRS, SET state, or logical '1'), and high resistive state (HRS, RESET state, or logical '0'). As this resistance is continuous and slightly varies per write cycle [7, 32], ranges that correspond to these two states are defined. Fig. 3d shows these specs for the two logic ranges ('0' and '1'), as well ranges outside the defined specs that an RRAM device can enter due to defects or extreme process variations [7]; these are 'L', 'U', and 'H' ranges corresponding respectively to extreme low logic state (resistance beyond the spec), undefined state, and extreme high logic state (resistance below the spec); the states 'L', 'U', and 'H' have been seen in defective RRAMs [19]. Fig. 3e presents a typical 1T-1R cell; here, BL, WL, and SL indicate bit line, word line, and select line respectively, while int is the internal node of the cell.

C. CIM-Ah: Majority Logic

The majority logic gate [3] shown in Fig. 4 is an implementation example of the CIM-Ah class using a memristive device Z. It has three inputs: P and Q supplied as voltages from the *peripherals*, and Z stored in the *array*; The output Z_{new} is produced after a majority operation is performed. The output is '1' if the majority of the inputs P, $\overline{Q}$, and Z are '1', as described in the truth table. Here $\overline{Q}$ denotes the negation of Q. The state of Z can only change to another state for a limited amount of input combinations P and Q, as shown in the truth table of the figure. This function can be used to develop other logic functions, like imply or inversion.

D. CIM-Pr: Scouting Logic

Scouting logic [29] pictured in Fig. 5a is an implementation example of CIM-Pr executing bit-wise logic OR, AND, and XOR. The operands are initially programmed in the *memory cells* M1 and M2. The operation is performed by selecting the two cells simultaneously (by applying a *read voltage* V_{r}) and comparing the resulting current (I_{in}) to a reference current (I_{ref}) using a dedicated sense amplifier (SA); the reference to be used depends on the operation to be performed as shown in Fig. 5b.



(a) Memory (b) References of Primitive Operations

Fig. 5: Scouting logic operations [29]

III. Test Methodology for CIM

This section presents a testing methodology for CIM cores. However, the difference between functional versus structural testing for such cores is first discussed.

A. Functional and Structural Testing

Tests for electronics can be classified into two categories: *functional* and *structural* tests [33]. Functional tests aim at checking the *proper operation* of the device-under-test, while structural tests aim at checking if the device is *manufactured correctly*. The question is which of these two approaches could be used for CIM die testing.

Functional tests apply a range of input stimuli to a device and observe if the corresponding output responses are correct, as defined by the device operation. To illustrate this for a CIM core, assume a functional test is used to test the CIM core being able to perform bit-wise logic operations of two operands. If we assume that the memory within the CIM core has r-bit row addresses, then there are $2^r \cdot (2^r - 1)$ possible combinations of selecting two operands. Even if extremely high test frequencies of 10 GHz are used, testing for all these combinations would take more than 58 years per chip for an address size of r=32. Besides being extremely time consuming, detection of all faults is still not guaranteed. For instance, the above case does not consider different values for the operands.

Structural tests, in contrary to functional tests, verify if a device is *manufactured correctly*; i.e., the device is free of manufacturing defects such as broken connections. It assumes that if the device is manufactured correctly, the device should functionally work properly. These tests rely on *fault models* that describe the faulty behavior of the device in the presence of a defect. This makes it feasible to define how these faults are sensitized and measure whether a test detects them or not. Therefore, *accurate* fault models are the key enabler for high quality structural test solutions. Note that structural tests do not require all input combinations to be tested, but merely those that *sensitize* the targeted faults models. Therefore, structural tests are both faster and achieve a higher and measurable fault coverage [33]. As a result, structural testing is more widely adopted. Thus, a high-quality CIM test should be a *structural* test, and will require accurate fault models reflecting the real defect behavior of CIM dies. Nevertheless, functional tests could be used to increase the fault coverage of faults that cannot be detected with structural testing, as is recognized in the community [34].

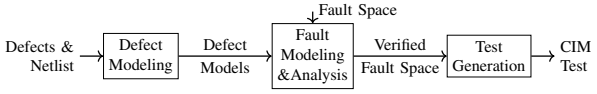


Fig. 6: Structural test approach

B. CIM Test Approach

As already mentioned, a CIM core operates in two configurations: memory configuration and computation configuration. Note that at least part of the CIM hardware used in the computation configuration is not used in the memory configuration. Hence, CIM cores cannot be tested as regular memories. CIM cores have to be tested for both configurations. As the computation configuration makes use of the memory, the latter has to be tested first. Testing CIM cores has to be performed as follows:

1) **Memory Configuration Test:** In this case, the memory functionality is tested; i.e., only the hardware that is required to perform memory operations is enabled and tested. Obviously, common memory test solutions applicable to the type of memory can be used (e.g., RRAM, STT-MRAM, PCM). Note that testing CIM in this configuration is an independent step, and does not test all hardware involved in the computation configuration.
2) **Computation Configuration Test:** In this case, the hardware responsible for all the computing functionalities is tested. This hardware strongly depends on the CIM architecture and the computing features it enables. For example, testing a CIM die with (analog) vector matrix multiplication features could be different than testing for logic bit-wise operations.

Test development for any IC follows three known steps illustrated in Fig. 6. First, the *defects* must be understood and adequately modeled. The resulting *defect models* are injected into the electrical netlist of the design. Second, this netlist is simulated and the faulty behaviors are observed and compiled into *fault models*. Ideally, before the fault analysis, the complete *fault space* should be defined (when applicable). During the *fault analysis*, the fault space is verified by injecting every defect in the netlist, which results in a set of realistic faults for that specific design or layout. Third, test solutions for the realistic faults are *generated*. Applying the above test development approach to CIM would mean applying it two times; once for each CIM configuration (i.e., memory and computation).

Test development for CIM as memory: the memory core of CIM can be any kind of memory such as conventional ones (SRAM, DRAM) as well as emerging ones (RRAM, PCM, STT-MRAM). Although testing of SRAM and DRAM is very mature, testing of emerging memories is still under investigation. They may need radically new approaches in defect modeling; a defective non-linear device (e.g., an RRAM device) cannot be accurately modeled with a linear resistor in series or in parallel with a perfect device [18, 19].

Test development for CIM as computing unit: As already mentioned, testing CIM in this configuration is strongly dependent on the design of the architecture. Defining what to test for implies the identification of the modified or new blocks integrated with the memory core to realize the computing functionality. To illustrate this, we will briefly analyze two examples of CIM architectures: CIM-Ah and CIM-Pr as discussed in Section II.

CIM-Ah Majority Logic: realizing such functionality within e.g. RRAM crossbar will need the modification of the following memory components: a) Memory array, b) BL and SL drivers, and c) Control logic. CIM-A architectures require always a redesign of the memory cells, as the conventional memory cell dimensions and their embedding in the bit and word line structure do not allow them to be used for logic. A conventional memory cell is namely heavily optimized in terms of processing stack and layout. Therefore, any modifications of the array require a new cell design and characterization process for the new control voltages, currents, etc. In addition, modifications in the periphery are needed to support the changes in the cell. In case of CIM-Ah Majority Logic, the write drivers and the control circuitry have to be redesigned to support the required functionality; e.g., the control logic needs to assure that the output of the sense amplifier can be fed back into the array via the drivers for operations on data from multiple cells. Therefore, testing CIM-Ah Majority Logic requires the guarantee of testing the memory array, BL and SL drivers, and the control logic. Note that the memory array is tested both in the memory configuration as well as in computation configuration; an access to the memory during computation could lead to an erroneous bit flip of the cell.

CIM-Pr Scouting Logic: As Fig. 5a shows, realizing such functionality, for example within RRAM crossbars, will need the modification of the following memory components: a) Memory array, b) Word line decoders, c) Sense amplifiers, and d) Control logic. Even though the computational results are produced in the peripheral circuits, the memory array for CIM-P is a substantial component in the computation. As the peripheral circuits are modified, the currents and voltages applied to the memory array are typically different than in the conventional memory. Obviously, the majority of the changes take place in the peripheral circuits and minimal to medium changes are required in the memory array. CIM-Pr Scouting Logic activates two or more (but not many) rows of a memory array simultaneously (similar to multi-port memories) during computations. Hence, in addition to a customized sense amplifier to perform the logic operation, this architecture also requires modifications in the address decoder to activate several rows at the same time. Note, however, that modifications in the cell array could be minimal as the total read current is still small. Therefore, testing CIM-Pr Scouting Logic requires to test the memory array, sense amplifiers, the decoders, and the control logic. Note also here that the memory array is tested both in the Memory Configuration as well as in Computation configuration; e.g., simultaneous access of the memory array during computing may lead to a fault in a cell.

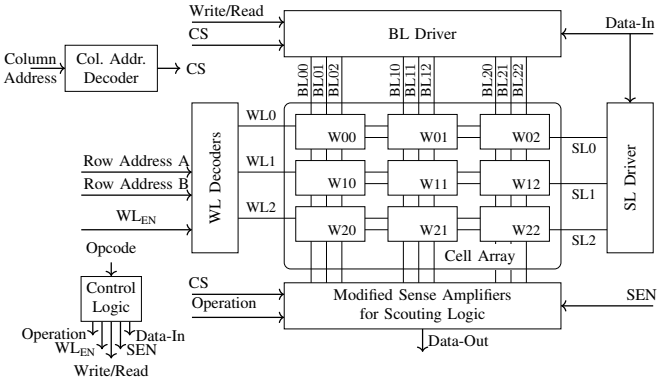
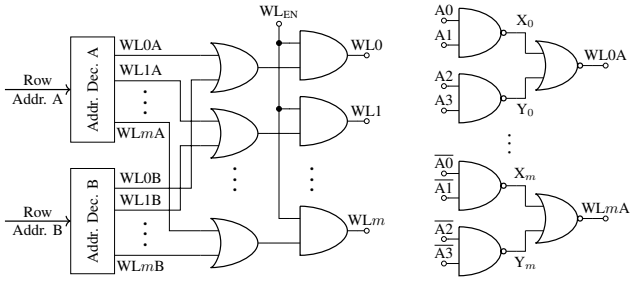


Fig. 7: Simulation Architecture



(a) WL Decoder (b) Address Decoder

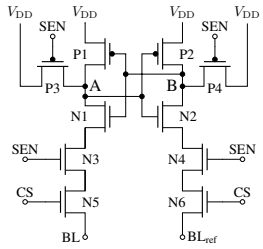
Fig. 8: WL circuitry

IV. CIM-Pr Architecture

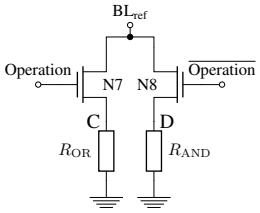
This section describes the implementation of CIM-Pr architecture based on Scouting Logic [29], which will be used to demonstrate the proposed test approach in the next two sections. This architecture is shown in Fig. 7; it is based on a regular RRAM design. Note that the majority of the building blocks (subcircuits) remain unmodified; these consist of the column address decoder, the BL driver, and the SL driver. The column address decoder decodes the column address and drives the corresponding column select (*CS*) line. The BL driver drives the BL corresponding to the *CS* line with the data in *Data-In*. To prevent the decoder from disturbing read operations, its output is fed through a tri-state buffer that is controlled by *Write/Read*. The SL driver controls the SLs based on *Data-In*; the SL is '0' when setting (w1) and reading, and '1' when resetting (w0) the cells.

However, in order to perform Scouting logic (i.e., bit-wise logic operations on two operands), some sub-circuits needed to be redesigned; these consist of: a) the WL decoder (that should be able to select two wordlines simultaneously), b) the SAs (that need to support appropriate logic functions; see Fig. 5), c) the control circuitry (to provide appropriate control signals based on the *Opcode*), and d) memory array. The latter is typically optimized for storage and would undergo some minimal modifications to allow for specific drive voltages and read currents. As the modified subcircuits will need special attention during testing, they will be briefly explained next; we focus on WL decoders and SAs.

- **WL Decoders:** The WL decoder, shown in Fig. 8a, decodes two *Row Addresses* A and B and drives the corresponding WLs (i.e., address m drives WLm) for the selection of the appropriate two words for a logic operation. Besides logic gates, the WL decoder consists of two identical address decoders. Fig. 8b illustrates such a 4-bit decoder; for each input combination (e.g., $A_3A_2A_1A_0$=1111) one WL is selected. Each two selected WLs per input combination (e.g., $A_3A_2A_1A_0$=$B_3B_2B_1B_0$=1111) are ORed, and the resulting signal is ANDed with the *WL$_{EN}$* signal to control the timing.



(a) Sense Amplifier [35]



(b) Operation reference

Fig. 9: Sensing Circuitry

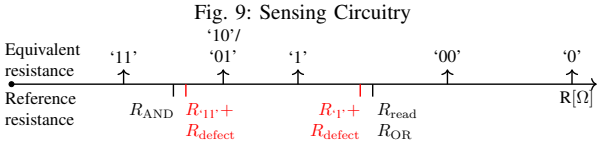


Fig. 10: Scouting logic relative resistance and references

- **Sense Amplifier:** One possible modified SA design for Scouting logic is shown in Fig. 9a, which is based on [35]. The two nodes A and B are precharged when no operation takes place, i.e., *SEN*='0'. Once the SA is enabled via the *SEN* and *CS* signals, the two nodes will be discharged via BL and BL$_{ref}$. The time it takes to discharge the nodes depends on the connected resistances to these nodes. For example, if R_{BL}<R_{BLref}, BL will discharge faster. After some time, the cross-coupled inverters begin to charge node B, allowing for even faster discharging of node A and the capturing of the operation outcome; we use node B in our design. To enable OR and AND bit-wise logic operations, the SA needs to have two corresponding reference currents, I_{OR} and I_{AND} (see Fig. 5). These are implemented using two different resistors, R_{OR} and R_{AND}, as shown in Fig. 9b. The *Operation* signal is used to select a reference; $\overline{Operation}$ is its logic complement. Fig. 10 shows the relative resistance of these references with respect to the equivalent resistance of the two cells being selected for the operation. In the memory configuration, the equivalent resistance is equal to the resistance of the cell being read, while in the computation configuration, it is equal to the parallel resistance of two cells being accessed. Note that R_{read}=R_{OR}.

V. CIM-Pr Memory Configuration Test

This section illustrates the test approach for the CIM-Pr core in the memory configuration; it includes defect modeling, fault analysis, and test generation; see Fig. 6.

A. Defect Modeling

The manufacturing process of a CIM core consists of three production phases: the front-end-of-line (FEOL), the back-

end-of-line (BEOL), and CF forming. To accurately estimate the impact of manufacturing defects on the circuit behavior, these defects need to be understood and modeled such that they can be used during circuit simulation for fault analysis. Two classes of defect models exist; they are discussed next.

Linear resistor as defect model: During the FEOL phase, transistors are fabricated on the wafer. Here standard transistor defects may occur that are related to line edge roughness, random dopant fluctuations, gate material granularity, etc. [36]. These defects may result, e.g., in reduced driving capabilities. They can have an impact on the peripheral circuitry as well as on the memory array, e.g., the SA becomes biased towards one logical value. After the FEOL phase, the lower metal layers are deposited in the BEOL phase. Lithographic issues or misalignment may cause defects here, resulting in shorts or opens in the wiring [37]. These defects again affect both the peripherals and the memory array. For example, the address decoder may wrongfully access multiple cells at the same time. These defects have been always modeled as linear resistors [11, 13] that act as a short or an open between two nodes.

Device-Aware defect models: The RRAM device is fabricated between two metal layers. Defects that may occur can be related to the electrode [38] and the oxide structure [39], which do affect the memory array. After this step, the remaining metal layers are deposited. To create a CF in the RRAM device, a forming step is required; this step strongly depends on the forming current (I_{form}) and may cause defects like over-forming or non-forming [17]. Although it can be convincing for modeling opens and shorts in interconnects, using linear resistors for defect modeling has never been validated for any device. It has recently been demonstrated that this assumption is inaccurate for emerging technologies such as (RRAM) [17] and (STT-MRAM) [40]; the results showed that the traditional approach may even lead to wrong fault models. Hence, it is incapable of delivering high-quality test solutions. This has resulted in the development of *device-aware defect modeling* approach [17, 19, 40]; it aims at accurately modeling physical defects, by incorporating the way the defect impacts the technology parameters (e.g., length, width) and thereafter the electrical parameters (e.g., the critical switching current) of the device [40]. This results in an electrical model of the defective device (e.g. RRAM device). This model can be then used to replace a defect free model at the circuit level to investigate its impact on the memory behavior. Note that in case of Device-Aware defect modeling, each defect may result in a different electrical model of the device.

B. Fault Modeling

Fault modeling is ideally based on two steps: 1) fault space definition, and 2) fault space validation using defect injection and circuit simulation. The fault space identifies *all possible* faults that can take place; i.e., any deviation from the correct functional behavior of a memory. This can be done analytically as the space of the potential memory operations is defined. However, the space is huge and constraints should be made in order to limit the space to a reasonable sized one. Once the space is identified, the fault analysis can take place; stimuli sensitizing each of the faults should be developed and applied to an appropriate memory simulation model while the defective device is replaced with its model. This should be repeated for all possible defects. Next, we will illustrate the above, first for the memory array and thereafter for the key peripheral circuits (i.e., address decoder and sense amplifier).

Fault Modeling for memory array

Fault Space: Memory array faults can be described by *Fault Primitive (FPs)* [41]. A fault is noted in the $\langle \mathrm{S/F/R} \rangle$ notation. In this notation, S denotes the sensitizing sequence for the fault, i.e., $\mathrm{S} = x_0 O_1 x_1 \ldots O_i x_i \ldots O_n x_n$. Here, x_i denotes the cell state, i.e., $x_i \in \{0, 1\}$, O_i denotes the operation that takes place, i.e., $O_i \in \{\mathrm{r}, \mathrm{w}\}$, where r and w indicate a read and write operation, respectively, and n is the number of operations. F denotes the value that is stored in the cell after S is performed, i.e., $\mathrm{F} \in \{\mathrm{H}, 1, \mathrm{U}, 0, \mathrm{L}\}$, where 'U' denotes the undefined state [41], 'H' the extreme logical 1 state, and 'L' the extreme logical 0 state, as demonstrated by measurements performed on defective RRAM and STT-MRAM devices [12, 18]. Finally, R (read output) describes the output of a read operation if the last operation in S is a read operation. $\mathrm{R} \in \{0, 1, ?, -\}$, where ? denotes a random read value (e.g., the sensing current is very close to sense amplifier reference current), and '$-$' denotes that R is not applicable, i.e., when the last operation in S is a write operation.

Given the above S, F, and R, the fault space for the memory array can be defined, like it was done in [19] for *static* single-cell faults. More complex faults such as those involving more than one operation (i.e., *dynamic* faults) or those involving multiple cells (e.g., *coupling* faults) can be defined in a similar manner by extending the FP notation [41].

Fault Analysis: some work on RRAM fault analysis is presented in [12, 13, 42] where the defects were modeled as a linear resistor (LR), and other work in [17, 19] where the authors used Device-Aware (DA) defect modeling. We only illustrate the results for the forming defect as presented in [19]. Table I lists the results of the static single-cell fault analysis; the FPs sensitized when assuming LR (both as a series and parallel resistor) and DA models for the forming defect are shown. The results are obtained by simulating different sizes of the defect. The table clearly highlights the difference between the two approaches. The unique DA faults (7 out of 8 of the realistic faults) cannot be sensitized with LR approach. Moreover, the LR model approach triggers 8 unique faults which are not realistic for forming defects, hence leading to a waste of test time. Note that only 1 common fault is observed by both approaches.

A complete fault analysis should consider each potential defect in the memory array, model it using the DA approach, and thereafter perform defect injection and circuit simulation.

Fault Modeling for some peripheral circuits

Address Decoder: Address decoder faults (AFs) in semiconductor memories are well studied. These faults can be

TABLE I: Validated faults using LR and DA models.

Range	FPs	DA	LR series	LR parallel
5 µA	⟨1w0/L/−⟩	Yes	No	No
[5; 13] µA	⟨1/U/−⟩, ⟨1w1/U/−⟩, ⟨1r1/U/1⟩	Yes	No	No
(13; 34] µA	⟨0/L/−⟩, ⟨0r0/L/0⟩, ⟨0w1/L/−⟩	Yes	No	No
(13; 34] µA; [4k; 40k] Ω	⟨0w0/L/−⟩	Yes	Yes	No
[12k; 16k) Ω	⟨0w1/U/−⟩	No	Yes	No
[16k; ∞) Ω	⟨0w1/0/−⟩	No	Yes	No
[1.6k; 5k] Ω	⟨1w0/U/−⟩	No	Yes	No
(5k; ∞) Ω	⟨1w0/1/−⟩	No	Yes	No
[8k; ∞) Ω	⟨1r1/1/0⟩	No	Yes	No
[0; 12k] Ω	⟨0w1/0/−⟩, ⟨0r0/0/1⟩	No	No	Yes
[0; 3k] Ω	⟨1w0/1/−⟩	No	No	Yes
(3k; 20k] Ω	⟨1w0/U/−⟩	No	No	Yes
[0; 6] Ω	⟨1w1/H/−⟩	No	No	Yes
[0; 1] Ω	⟨1r1/H/1⟩	No	No	Yes

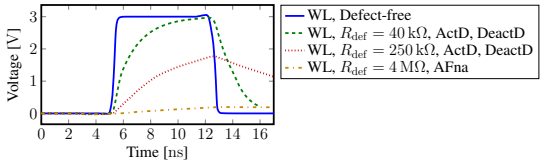


Fig. 11: WL decoder faults

static or dynamic. Static AFs are mainly caused by completely broken interconnects (e.g., wordline) or low ohmic bridges between connections and consist of four possible faults [43]: 1) *No-access* (AFna): an address does not access its cell, 2) *Multiple cells* (AFmc): an address uniquely accesses multiple cells, 3) *Multiple addresses* (AFma): a cell is uniquely accessed by multiple addresses, and 4) *Other cells* (AFoc): an address additionally accesses other cells. On the other hand, dynamic or delay address decoder faults (ADFs) are caused by partial opens and shorts; they consist of two possible faults [44]: 1) Activation delay (ActD): the activation, e.g., of a wordline, is delayed, and 2) Deactivation delay (DeActD): the deactivation, e.g., of a wordline, is delayed. These faults may lead to erroneously addressing of multiple cells at the same time, or to shortening the cell access time which may cause a e.g., a write operation to fail.

Fault analysis for address decoders has been studied also very well by assuming a linear resistor as defect model [44, 45]. For example, Fig. 11 illustrates how an open defect in a WL can cause AFna or ADFs, depending on the defect size.

Sense Amplifier: Sense amplifier faults in semiconductor memories have been well studied [45, 46]. They can be divided into static and dynamic faults. Static faults are assumed to be caused by complete opens, low ohmic shorts to V_{DD} or GND, or low ohmic bridges [43]; they consist of the traditional *SA Stuck-at fault* (SASF), an SASF means that the SA always outputs the same value, independent of its inputs. Dynamic faults are caused by partial opens and shorts and consist of two faults: 1) *Unbalanced SA fault* (USAF) [46]: the SA has a continuous tendency to switch to a certain value under equal input conditions, rather than being balanced, and 2) *Slow SA fault* (SSAF) [45]: the SA is too slow to switch, which may result in incorrect read values.

Fault analysis for SAs has been performed by assuming that any defect can be modeled as a linear resistor. Fig. 12 illustrates the faults that may occur when performing a r0 operation in an SA in the presence of an open defect (R_{def}) between transistors N2 and N4 of Fig. 9a. This defect leads to a slower discharge of node B, as the path to GND now has

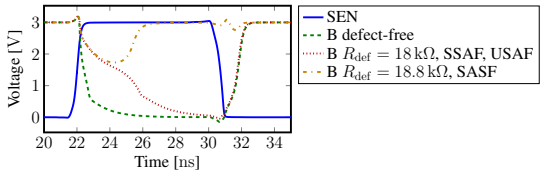


Fig. 12: SA faults

a higher resistance. It can be seen that when $R_{\text{def}} < 18.8\,\text{k}\Omega$, the defect causes an unbalance in the SA and thus is slowing down the sensing operation causing USAFs and SSAFs. When $R_{\text{def}} \geq 18.8\,\text{k}\Omega$, the SA will always switch to the wrong value, thus leading to an SASF.

C. Test Development

The output of the fault modeling (i.e., a set of fault models) is crucial for the development of efficient and high-quality test solutions. Faults can be classified in two categories [19]: strong and weak faults. Strong faults cause functional errors in the memory operation, and can be sensitized (and may be detected) by a known sensitizing sequence. On the contrary, weak faults do not result in any functional error; instead, weak faults are parametric faults, e.g., reduced bit line swing. These faults also need to be detected, as they may pose a reliability risk, e.g., increased in-field failure rate. Moreover, depending on the effort needed to detect them, faults can be divided into easy-to-detect (ETD) and hard-to-detect (HTD) faults. The detection of ETD faults can be *guaranteed* by applying write and read operations, e.g., by using a March test [43]. However, March tests *cannot* guarantee the detection of HTD faults, although they may detect them. Guaranteeing their detection may require additional effort; e.g., the use of a special Design-for-testabilty (DfT) circuitry. An example of an ETD fault is $\langle 1\text{r}1/0/0\rangle$, and an example of an HTD fault is $\langle 1\text{r}1/\text{U}/?\rangle$;

In order to develop appropriate test solutions for the CIM core in its memory configuration, first the obtained faults from fault modeling should be analyzed and classified into ETD and HTD faults and thereafter test solutions should be developed. In the rest of this section, we will illustrate the above for the previously discussed faults for the three components.

Memory Array: Let us consider the results shown in Table I for the forming defect when using Device-Aware fault modeling. The defect can sensitize in total 8 FPs, which can be grouped into 4 fault classes, where a fault class is a set of FPs sensitized by the same single defect with a certain range/size. Inspecting the table reveals that only $\langle 0\text{w}1/\text{L}/-\rangle$ is an ETD fault, while the rest is HTD faults. Detecting $\langle 0\text{w}1/\text{L}/-\rangle$ can be easily done by a March element $\Updownarrow (\text{w}0, \text{w}1, \text{r}1)$.

HTD faults in the memory array are typically related to the cell being in a forbidden state (i.e., 'H', 'U', or 'L') [19]. As already mentioned, March tests may detect some of these faults; repeating tests targeting HTD faults with different memory backgrounds and different address sequence [44, 45] will increase the detection probability. For example, the FP $\langle 1/\text{U}/-\rangle$ may be detected with a March element $\Updownarrow (\text{w}1, \text{r}1)$. However, detection is not guaranteed. Therefore, using DfT is a common practice to further increase the chance of detecting

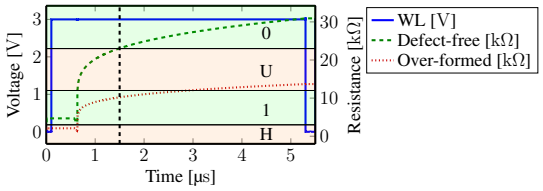


Fig. 13: Defect-free and over-formed cell

HTD faults. For example, the scheme in [13] uses shortened write times and reduced write voltages to detect cells that are in the 'U' state. This scheme can be modified to also detect cells suffering from an over-formed defect; i.e., cells whose state is 'H' instead of '1' after forming, as illustrated in Fig. 13. The figure shows a RESET operation on two cells that are initially in '1' and 'H' (i.e., a $1\mathrm{w}0$ operation) for the defect-free and over-formed cell, respectively. From the figure it follows that the defect-free cell switches quicker to the correct resistance range than the over-formed cell. The DfT is now used to shorten the write operation time to $1.5\,\mu\mathrm{s}$ (indicated by the black dashed line in the figure). The defect-free cell will have switched to '0', while the over-formed cell is still in '1'. A subsequent read operation on both cells will reveal this, and thus the defective cell can be detected.

Address Decoder: The four static AFs (AFna, AFmc, AFma, AFoc) belong to the ETD faults, while the two ADFs (ActD, DeActD) belong to the HTD faults. It has been shown that the detection of static AFs can be guaranteed by a March test that contains the following two March elements [43, 45]: $\Uparrow (\mathrm{r}x, \ldots, \mathrm{w}\bar{x})$ and $\Downarrow (\mathrm{r}\bar{x}, \ldots, \mathrm{w}x)$; here, $x \in \{0,1\}$ and $\bar{x}$ denotes the negation of x. The ADFs, however, *may* be detected by March tests; the detection probability strongly depends on the delay [44]. Their detection requires: 1) Sensitizing Address Transitions, and 2) Sensitizing Operation Sequences. Sensitizing address transition(s) can be caused by an address pair or an address triplet. For example, a Sensitizing Address Pair consists of a sequence of two addresses A_f and A_g which have to be applied in sequence because ADFs are sensitized by address transitions. These transitions are generated using an Addressing Method such as Address Complement, The H1 Addressing Method (H1 stands for hamming distance is 1), etc. [44]. On the other hand, the Sensitizing Operation Sequence should be generated and applied to each of the generated address pairs (A_f, A_g); this sequence consists of two operations ($Ox_\mathrm{f}; Oy_\mathrm{g}$), one operation applied to A_f and the other to A_g. O denotes a read or write operation ($O \in \{\mathrm{r}, \mathrm{w}\}$) with expected or written data $x, y \in \{0,1\}$. The operation on A_g has to be performed with the *complement* of the data value applied to A_f in order to detect e.g., ActD; because of the fault, Ox_f may fail. It is worth noting that each of the decoders should be tested individually; hence the test for address decoders need to be repeated twice for our CIM core in memory configuration.

Sense Amplifier: the static SASF belongs to the ETD faults and its detection can be easily guaranteed by any March test consisting of the two March elements (or a single March element combining both of them): $\Updownarrow (..., \mathrm{r}x, ...)$ and $\Updownarrow (..., \mathrm{r}\bar{x}, ...)$, with $x \in \{0,1\}$ [45]. Detecting dynamic faults (USAF, SSAF), which belong to the HTD faults may be done with March tests, although special DfT can do a better job. The sensitization and detection of SSAF requires the application of *back-to-back* operations to the memory using 1) *different* data values (0 and 1) and 2) *fast-row* addressing (i.e., each address increment or decrement causes an adjacent physical row to be accessed) [45]; back-to-back operations indicate that the two operations take place after each other without any delay. For example, a test consisting of the following March element (using fast-row addressing) may detect SSAFs: $\Updownarrow (\mathrm{r}x, ..., \mathrm{w}\bar{x})$; the read and write are back-to-back and use different data. E.g., the operation $\mathrm{w}0$ brings the bit lines in the worst case state for the following $\mathrm{r}1$ operation, applied to the next cell in the same column. Special DfT which can be used to complement March tests, can work better in detecting such faults. For example, the DfT proposed in [47] to detect HTD faults in SRAMs can be used here; it is based on monitoring the bit line swing at the input of the SA.

VI. CIM-Pr Computation Configuration Test

This section presents the test approach for the CIM-Pr core in the computation configuration. We follow again the approach that was presented in Section III. Note that the CIM-Pr under consideration is based on scouting logic.

A. Defect Modeling

Obviously, the same defect models apply for this configuration as those discussed in the previous section; they are Linear resistor and Device-Aware defect modeling. Linear resistors are suitable for interconnect defects and have been also shown to do a good job for transistor defects, while Device-Aware defect modeling is suitable for the RRAM defects.

B. Fault Modeling

Next fault modeling will be applied first to the memory array, then to the address decoders and sense amplifiers.

Memory Array: defining the fault space of memory array in the computation configuration is still an open question and can be strongly array design and architecture dependent [48, 49]. The memory array in the computing configuration acts as a special case of dual port memory; it allows for simultaneous access of two cells/locations in the same column. Hence, this may give rise to new faults. For example, accessing two cells simultaneously may unintentionally flip the state of one of them. Defining the fault space will need also the extension of the FP notation $\langle \mathrm{S/F/R} \rangle$. We can build on the notation developed for dual-port memory faults [50]; we denote a FP due to the simultaneously access as $\langle \mathrm{S_1 : S_2 \ / \ F_1 : F_2 \ / \ R} \rangle_{\mathrm{OP}}$, where $\mathrm{S_1}$ and $\mathrm{S_1}$ specify the sensitizing operations, ':' denotes the fact that $\mathrm{S_1}$ and $\mathrm{S_1}$ are applied *simultaneously*, $\mathrm{F_1}$ and $\mathrm{F_2}$ describe the value of the accessed cells after the sensitizing operations, R gives the read value, and OP specifies the operation performed (e.g., AND, OR). For example, $\langle 1\mathrm{r}1_1 : 1\mathrm{r}1_2 \ / \ 1_1 : 1_2 \ / \ 0 \rangle_{\mathrm{AND}}$ describes an AND operation on two cells containing '1' that results in

a wrong output '0'. To illustrate that such a fault is realistic, consider an open defect (R_{defect}) in the bit line that increases its resistance slightly. When the AND operation takes place, the equivalent resistance (see Fig. 10) $R_{\text{eq}} = R_{\text{`11'}} + R_{\text{defect}}$ can become higher than $R_{\text{AND}}$ and thus results in a wrong read output. In the memory configuration, however, no fault occurs as $R_{\text{`1'}} + R_{\text{defect}} < R_{\text{read}}$. Hence, this fault only occurs in the computation configuration. Defining the complete fault space and validating it, is still an open question.

Address Decoder: The Scouting logic computation configuration requires both address decoders to act simultaneously to select the appropriate word lines. This configuration may give rise to unique address decoder faults, and is quite similar to dual-port memories [51]; also here two addresses should be selected simultaneously. Hence, the same fault space and fault models can apply. Such faults are called *port interference faults* and are due to potential interference/bridges between the two decoders (between wires of the two different decoders). They differ from single AFs in the sense that they only occur when two decoders are accessed simultaneously, and not when operating sequentially. E.g., one of the decoder erroneously select an additional world line when the inputs of both decoders have defined value. Consider Fig. 8a and assume the two addresses $\mathrm{A_1A_2A_1A_0}$='1111' and $\mathrm{B_3B_2B_1B_0}$='1110' are selected in a 4-bit WL decoders; these will drive WL0A and WL1B simultaneously. If now a low ohmic bridge defect exist between the node $\mathrm{Y_1}$ of the decoder circuit driving WLB1 and the node $\mathrm{X_2}$ of the decoder circuit driving WLA2 (see Fig. 8a), then the simultaneous selection of WL0A and WL1B will result in erroneous selection of WL2A, i.e., WL0, WL1, and WL2 will be activated.

Sense Amplifier: The modified SA in the computation configuration may suffer from similar faults as the SA in the memory configuration. These faults (consisting of SASF, USAF and SSAF) can take place in each of the computing configurations of the SA including OR, AND, and XOR; note that the modified SA uses different reference currents to perform the different logic operations. The validation of such faults using fault analysis is still an open question.

C. Test Development

Tests for the computation configuration focus on: 1) testing the hardware that was not used during memory configuration test and, 2) on testing of unique faults that may be sensitized due to simultaneous access of the memory array (due to the selection of the operands of the logic operation). The test development approach in the computation configuration is similar to that of the memory configuration. Next we will illustrate the approach for (some of) the faults discussed in previous subsection.

Memory Array: Defining the complete fault space and validating it is still an open question. Nevertheless, we will illustrate how to develop an appropriate test for such faults. Let's consider the fault $\langle 1\mathrm{r}1_1 : 1\mathrm{r}1_2 \ / \ 1_1 : 1_2 \ / \ 0 \rangle_{\text{AND}}$ discussed in the previous subsection. This is an ETD fault as it produces a wrong output 0 instead of 1. If we assume that this fault only takes place when two accessed cells/operands (in the same column) are physically adjacent, then such a fault can be detected by a March test containing e.g., the following two March elements: $\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-2} (..., \mathrm{r}1_{r,c} : \mathrm{r}1_{r+1,c}, ...))$. Note that a nested addressing is used; R and C denote the number of rows and columns of the array, respectively. For each column c, cells at row r and $r+1$ are simultaneously accessed by an $\mathrm{r}1$ operation. Note that before such operations are performed, the cells have to be initialized with an appropriate data-background [45] (i.e., the pattern of 1's and 0's as seen in the memory array). For example, a solid 1 background (1111... /1111... /1111...) satisfies this requirement.

Address Decoder: Tests developed for dual-port memory address decoder faults (i.e., port interference faults) [51], can be easily adapted and used for testing the unique address decoder faults in computation configuration. Such tests have a time complexity (in the worst case) of $\mathcal{O}\left(R^2\right)$ where R is the number of array rows.

Sense Amplifier: An SA in each of the computation configuration (e.g., AND, OR) can suffer from the same faults as an SA in the memory configuration; these faults consist of SASF, USAF and SSAF. However, testing such faults will require special attention. For example, to detect the ETD fault SASF in the AND mode, a March test should contain the two March elements (or a single March element combining both of them): $\Updownarrow_{c=0}^{C-1} (..., \mathrm{r}0_i : \mathrm{r}x_j, ...)$ and $\Updownarrow_{c=0}^{C-1} (..., \mathrm{r}1_i : \mathrm{r}1_j, ...)$, where $x \in \{0, 1\}$ and (i, j) two addresses indicating any two cells/operands in the same column. The fault SASF1 will be detected by the parallel operations $\mathrm{r}0_i : \mathrm{r}x_j$ as this will return 1 instead of 0, while SASF0 will be detected by the parallel operations $\mathrm{r}1_i : \mathrm{r}1_j$ as this will return 0 instead of 1. Note that actually performing each of the two parallel operations once is enough for the detection of SSAF, and there is no need to repeat them for different address combinations (i, j). Next, we show how we can detect the SSAF in the AND configuration. As already mentioned in Section V, this fault is HTD and may be detected by a March test when applying *back-to-back* operations resulting in *different* data values (0 and 1) and using *fast-row* addressing. For example, a test consisting of the following March element (using fast-row addressing) may detect SSAFs of the SA in the AND configuration: $\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-2} (\mathrm{r}1_{r,c} : \mathrm{r}1_{r+1,c}, \mathrm{w}0_{r,c}, \mathrm{r}0_{r,c} : \mathrm{r}1_{r+1,c}))$; the two parallel operations are back-to-back and result in different data output. For example, the operation $\mathrm{r}0_{r,c} : \mathrm{r}x_{r+1,c}$ results in 0 bringing the SA in the worst case state for the following $\mathrm{r}1_{r,c} : \mathrm{r}1_{r+1,c}$ operation that has to result in 1, applied to the next cells in the same column. $\mathrm{w}0_{r,c}$ is just a write operation. Here also special DfT can be developed to complement March tests, and even do a better job in detecting such faults.

VII. Discussion and conclusion

This work highlighted the structural testing of CIM dies. Although our case study was based on Scouting logic, the approach is applicable to any CIM design.

Testing CIM dies solely as a memory is not enough, as each computation configuration needs to be tested as well, where

the focus is on testing 1) the partial and completely non-tested hardware during the memory test phase (this hardware consists of the modified or newly added components to the memory), 2) the unique faults that could take place due to simultaneous memory access (e.g., when executing a logic operation).

Although a lot of memory fault models and test solutions can be reused for CIM in the computation configuration, many new solutions are needed. These are strongly CIM architecture dependent. For example, the test solutions for CIM based on Scouting logic will differ from analog vector matrix multiplication with ADCs. Clearly there are still many open questions to be worked out such as:

- Fault modeling: defining the fault spaces for the different CIM architectures in the computation configuration and validating them using realistic design.
- Test solutions and optimization: developing appropriate test solutions (test algorithms, DfT, BIST solutions, etc) for the different architectures; optimizing the test approach by exploring the combination of the test solutions for the memory configuration and the computation configuration, especially for production test.

ACKNOWLEDGMENT

This research on CIM architecture is supported by EC Horizon 2020 Research and Innovation Program through MNEMOSENE project under Grant 780215.

REFERENCES

[1] D. A. Patterson, "Future of Computer Architecture," in *BEARS*, 2006.
[2] S. Hamdioui *et al.*, "Memristor for Computing: Myth or Reality?" In *DATE*, 2017.
[3] E. Linn *et al.*, "Beyond von Neumann-logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, 2012.
[4] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE*, 2015.
[5] S. Hamdioui *et al.*, "Applications of Computation-In-Memory Architectures based on Memristive Devices," in *DATE*, 2019.
[6] D. Fujiki *et al.*, "In-Memory Data Parallel Processor," in *ASPLOS*, vol. 53, 2018.
[7] H.-S. P. Wong *et al.*, "Metal-Oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, 2012.
[8] H. A. Du Nguyen *et al.*, "On the Implementation of Computation-in-Memory Parallel Adder," *IEEE TVLSIS*, vol. 25, no. 8, 2017.
[9] Y. Chen *et al.*, "Recent Technology Advances of Emerging Memories," *IEEE Des. Test*, vol. 34, no. 3, 2017.
[10] S. Hamdioui *et al.*, "Test and Reliability of Emerging Non-volatile Memories," in *ATS*, 2017.
[11] S. Kannan *et al.*, "Sneak-Path Testing of Crossbar-Based Nonvolatile Random Access Memories," *IEEE TN*, vol. 12, no. 3, 2013.
[12] C. Y. Chen *et al.*, "RRAM defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE TC*, vol. 64, no. 1, 2015.
[13] S. Hamdioui *et al.*, "Testing Open Defects in Memristor-Based Memories," *IEEE TC*, vol. 64, no. 1, 2015.
[14] Chin-Lung Su *et al.*, "MRAM defect analysis and fault modeling," in *ITC*, 2004.
[15] J. Azevedo *et al.*, "A Complete Resistive-Open Defect Analysis for Thermally Assisted Switching MRAMs," *IEEE TVLSIS*, vol. 22, no. 11, 2014.
[16] X. Pan *et al.*, "Modeling and test for parasitic resistance and capacitance defects in PCM," in *NVMTS*, 2012.
[17] M. Fieback *et al.*, "Testing Resistive Memories: Where are We and What is Missing?" In *ITC*, 2018.
[18] L. Wu *et al.*, "Pinhole Defect Characterization and Fault Modeling for STT-MRAM Testing," in *ETS*, 2019.
[19] M. Fieback *et al.*, "Device-Aware Test: A New test Approach Towards DPPB Level," in *ITC*, 2019.
[20] A. Sebastian *et al.*, "Temporal correlation detection using computational phase-change memory," *Nat. Comm.*, vol. 8, no. 1, 2017.
[21] B. Chen *et al.*, "Efficient in-memory computing architecture based on crossbar arrays," in *IEDM*, 2015.
[22] M. A. Lebdeh *et al.*, "Memristive Device Based Circuits for Computation-in-Memory Architectures," in *ISCAS*, 2019.
[23] P.-E. Gaillardon *et al.*, "The Programmable Logic-in-Memory (PLiM) computer," in *DATE*, 2016.
[24] D. Bhattacharjee *et al.*, "Revamp: Reram based vliw architecture for in-memory computing," in *DATE*, 2017.
[25] S. Hamdioui *et al.*, "Computing device for big data applications using memristors," in *US Patent 9,824,753*, 2017.
[26] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Comp. Arch. News*, vol. 44, 2016.
[27] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016.
[28] H. A. D. Nguyen *et al.*, "Memristive devices for computing: Beyond cmos and beyond von neumann," in *VLSI-SoC*, 2017.
[29] L. Xie *et al.*, "Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing," in *ISVLSI*, 2017.
[30] M. Le Gallo *et al.*, "Mixed-precision in-memory computing," *Nat. Elec.*, vol. 1, no. 4, 2018.
[31] L. Chua, "Memristor-The missing circuit element," *IEEE TCT*, vol. 18, no. 5, 1971.
[32] S. Yu *et al.*, "Emerging Memory Technologies: Recent Trends and Prospects," *IEEE SSC*, vol. 8, no. 2, 2016.
[33] M. L. Bushnell *et al.*, *Essentials of Electronic Testing for Digital Memory & Mixed-Signal VLSI Circuits*. Springer Science+Business Media, 2000.
[34] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *VLSI-DAT*, 2018.
[35] W. Zhao *et al.*, "Synchronous Non-Volatile Logic Gate Design Based on Resistive Switching Memories," *IEEE TCSI*, vol. 61, no. 2, 2014.
[36] K. J. Kuhn *et al.*, "Process Technology Variation," *IEEE TED*, vol. 58, no. 8, 2011.
[37] E. I. Vatajelu *et al.*, "Challenges and Solutions in Emerging Memory Testing," *IEEE TETC*, 2017.
[38] H. Y. Lee *et al.*, "Evidence and solution of over-RESET problem for HfOX based resistive memory with sub-ns switching speed and high endurance," in *IEDM*, 2010.
[39] M. Lanza *et al.*, "Grain boundaries as preferential sites for resistive switching in the HfO2 resistive random access memory structures," *Appl. Phys. Lett.*, vol. 100, no. 12, 2012.
[40] L. Wu *et al.*, "Electrical Modeling of STT-MRAM Defects," in *ITC*, 2018.
[41] S. Hamdioui *et al.*, "An experimental analysis of spot defects in SRAMs: realistic fault models and tests," in *ATS*, 2000.
[42] Y.-X. Chen *et al.*, "Fault modeling and testing of 1T1R memristor memories," in *VTS*, 2015.
[43] A. J. van de Goor, *Testing Semiconductor Memories - Theory and Practice*. John Wiley & Sons, 1991.
[44] S. Hamdioui *et al.*, "Opens and Delay Faults in CMOS RAM Address Decoders," *IEEE TC*, vol. 55, no. 12, 2006.
[45] A. van de Goor *et al.*, "Detecting faults in the peripheral circuits and an evaluation of SRAM tests," in *ITC*, 2004.
[46] K. Zarrineh *et al.*, "Defect analysis and realistic fault model extensions for static random access memories," in *IWMTDT*, 2000.
[47] G. C. Medeiros *et al.*, "DFT Scheme for Hard-to-Detect Faults in FinFET SRAMs," in *ETS*, 2019.
[48] D. Niu *et al.*, "Low power memristor-based ReRAM design with error correcting code," in *ASP-DAC*, 2012.
[49] B. Zhao *et al.*, "Common-source-line array: An area efficient memory architecture for bipolar nonvolatile devices," *ACM TODAES*, vol. 18, no. 4, 2013.
[50] S. Hamdioui *et al.*, "Efficient tests for realistic faults in dual-port srams," *IEEE TC*, vol. 51, no. 5, 2002.
[51] S. Hamdioui *et al.*, "Address decoder faults and their tests for two-port memories," in *IWMTDT*, 1998.