Unlocking Sneak Path Analysis in Memristor Based Logic Design Styles

Kamalika Datta*, Saeideh Shirinzadeh*†, Phrangboklang Lyngton Thangkhiew‡ Indranil Sengupta§¶, Rolf Drechsler*‖
*German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
† Fraunhofer Institute for Systems and Innovation Research(ISI), Karlsruhe, Germany
‡Department of Computer Science and Engineering, Indian Institute of Information Technology Guwahati, India
§Department of Computer Science and Engineering, JIS University, India
¶Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India
‖Institute of Computer Science, University of Bremen, Germany
E-mail: kamalika.datta@dfki.de, saeideh.shirinzadeh@dfki.de, phrangboklang.thangkhiew@iiitg.ac.in, indranil.sengupta@jisuniversity.ac.in, drechsler@uni-bremen.de

***Abstract*—Memristors or Resistive Random Access Memory (RRAM) are emerging non-volatile memory devices that can be used for both storage and computing. In this type of memory the information is stored in memory cells in the form of resistance. One of the very important challenges in memristive crossbars is the existence of *Sneak Paths*, which result in erroneous reading of memory cells. Most of the logic in-memory techniques have emphasized on improving the logic design perspective, but have given minor importance to the sneak path issue. In this paper we show the effect of sneak paths on crossbars of various sizes, and then try to analyze the logic design approaches like MAGIC and MAJORITY with respect to their immunity to sneak paths. Experimental result shows that with some extra overhead we can eliminate the sneak path effect in various logic design methods.**

***Index Terms*—RRAM, memristor crossbar, sneak path, in-memory computing**

I. Introduction

According to the International Technology Road-map for Semiconductors (ITRS), contemporary memory technologies like SRAM, DRAM, Flash, etc. will not be able to sustain the design challenges arising from the scaling down of transistors. Various alternative solutions like memristors are being investigated by researchers. Memristor is a passive circuit element that is capable of changing its resistance depending on the voltage applied across it [1]. The resistance change is non-volatile in nature, which makes applications like resistive memory and in-memory computing possible.

Although the device properties of memristors were discussed many years before, but the popularity of such device started with the physical realization of memristors in 2008 [2]. Since its fabrication in 2008, research has taken place for designing memory system using memristors as well as for performing logic operations [3]–[5]. Due to their fast resistive switching properties and low footprint for fabrication, it is possible to fabricate high-density resistive memory systems using memristor crossbars. The crossbar structure typically consists of a set of horizontal and vertical nanowires, with the memristor devices fabricated at every junction [5]. By applying suitable voltages across the rows (also called *word lines*) and columns (also called *bit lines*) of the crossbar, selected cells can be set to either low resistive state (LRS) or high resistive state (HRS).

One vital issue with memristor crossbars is the presence of sneak paths during cell readout operations [6]. A sneak path is defined as a parallel current conduction path, which is separate from the direct path used to sense the state of a crossbar cell. The presence of such undesirable parallel paths can result in erroneous reading of a cell, since a cell in HRS can be wrongly sensed as being in LRS due to high sneak path current.

A number of approaches have been proposed to characterize and mitigate the sneak path problem in crossbars. There have been efforts to formally characterize the conditions under which sneak paths can cause erroneous readout [7]. While mapping logic operations to crossbar cells, there will be no sneak path current if none of the necessary conditions as mentioned in [7] are met. In several works relating to in-memory computing on crossbars [8], some rows can be disabled by feeding them with an isolation voltage V_{iso} of suitable magnitude. In an alternate approach, individual crossbar cells can be enabled or disabled by providing some access device with every cell. This leads to one-diode one-resistor (1D1R) and one-transistor one-resistor (1T1R) structures that can solve the sneak path problem, but at the cost of additional area overhead due to the larger sizes of the access devices [9]. As an alternative [10], every crossbar cell can contain two back-to-back memristors in complementary states. This arrangement can minimize the sneak path problem with nominal hardware overheads; however, the complexity of the read circuitry increases. Also if we consider the various logic design styles [8], [11]–[17] in literature, we see that most of them use 0T1R crossbar structure without any access devices, and are hence vulnerable to sneak paths. Hence it is of utmost importance to analyze sneak path effects in those methods.

The main contributions of this paper are as follows:

- We critically analyze the impact of sneak paths in cross-

bars, and how they impact the correct reading of resistive states of the devices.
- We analyze sneak paths of various lengths say k where k is an odd number, whereby k number of memristor devices in LRS appear in series and sneak some current through the column being sensed.
- We analyze MAGIC and MAJORITY based logic design styles and the impact of sneak paths on these methods. A limited size crossbar with three columns is found to eliminate sneak path issues.

The rest of the paper is organized as follows: Section II provides a brief literature survey of memristors, and some prior works on analyzing sneak paths in crossbars. Section III discusses sneak path analysis on different size crossbars. In Section IV we analyze the existing in-memory computing techniques like MAGIC and MAJORITY and how sneak paths affect them. In Section V we provide the experimental results, followed by concluding remarks in Section VI.

II. Preliminaries and Related Works

In this section, we discuss about the preliminaries of memristors, and also some previous works that have analyzed sneak paths in crossbar.

A. Memristors

A memristor is a two-terminal passive circuit element that exhibits non-volatile switching of its resistance in response to the voltage applied across it [1]. In 2008 a research group from HP labs reported the fabrication of a memristor device [2]. They used a metal-insulator-metal structure, where a Titanium Oxide (TiO_2) slab is created between Platinum (Pt) electrodes. In general, the device has a region with oxygen vacancies (TiO_{2-x}) referred to as the *doped region*, and one with no oxygen vacancies (TiO_2) referred to as the *undoped region*. The doped region is highly conductive while the undoped region is highly resistive. Depending on the voltage applied across the terminals, the boundary between the two regions can be made to move, which changes the overall resistance of the device. When the voltage across the device is withdrawn, the state of the device remains unchanged, which makes it an ideal candidate for non-volatile memories. Memristors are programmed to be in one of two states, *low resistance* (logic 1) and *high resistance* (logic 0) for logic and memory operations.

Fig. 1(a) depicts the schematic symbol of a two-terminal memristor. We can fabricate memristors in a compact fashion as a crossbar, as shown in Fig. 1(b), where each device is fabricated at the junction of a distinct row and column wire.

Memristive crossbars have been explored for both resistive storage and in-memory computing applications. However, the *sneak path* issue can potentially limit the crossbar from operating with full benefits. We briefly discuss the various works in the literature that either analyze sneak paths or discuss methods to mitigate them.

B. Related Works on Sneak Path Analysis

Many works have been reported in the literature where analysis of sneak paths have been performed [6], [7], [10], [18], [19].

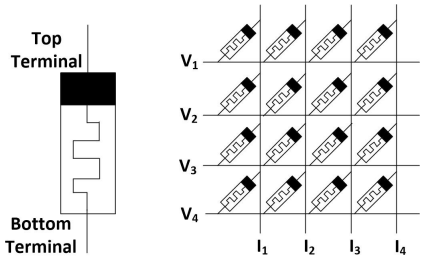


Fig. 1. (a) Schematic of a two-terminal memristor, (b) A 4×4 crossbar.

Linn et al. introduced a complementary resistive switch as a solution to sneak path reads in crossbars [10]. Here two anti-serial memristors are placed together to form a single cell.

Zidan et al. analyzed sneak paths in a crossbar with particular emphasis on how aspect ratio of the memory array affects the overall sneak path issue [6]. They also suggested that higher number of cells in LRS leads to more sneak paths. Finally they proposed a method for gating the memory cell with a three-terminal memristor device. This method also increases the crossbar size with the gating elements.

Cassuto et al. stated the necessary conditions to maintain sneak-path-free readouts in crossbars [7]. They put emphasis on the requirement of isolated-zero-rectangle free array, which is an important concept from logic design point of view.

Gul et al. have used a one-diode one-resistor (1D1R) structure, with Schottky diodes as access devices [18]. They fabricated the device and investigated anti-crosstalk characteristics.

As an alternative choice, one can also use one-transistor one-resistor (1T1R) structure [9]. Although both 1T1R and 1D1R structures mitigate the sneak path effect on a crossbar, they increase the total area of the crossbar. Joshi et al. analyzed and proposed methods for detecting sneak path currents in memristive crossbar based on crossbar array size, I/O switch vectors, memristor resistance and memristor programming [19].

We can see that several solutions have been suggested for mitigating sneak paths in crossbar arrays. But if we want to utilize the full benefit of a crossbar without access devices (i.e.. 0T1R), then we must map the circuits in such a way that sneak paths do not occur. This is not an easy task from the point of view of logic design. In this paper we analyze MAGIC and MAJORITY based logic design styles for sneak paths. As opposed to other methods in literature not only we analyze sneak paths but also try to provide some perspective for logic synthesis. We specifically identify the issues of these methods where sneak path can affect the overall correctness.

III. Sneak Paths and Current Analysis in Memristor Crossbars

In a memristive crossbar, when we want to read the state of a cell (i.e., HRS or LRS), we apply a read voltage along the row and measure the current along the column. However, it may so happen that due to parallel current conduction paths in

the crossbar, the current being sensed is affected. This can lead to erroneous cell reading, whereby a cell in HRS is wrongly interpreted as being in LRS.

Consider a 5×5 crossbar (see Fig. 2(a)) where we want to read a HRS cell highlighted in orange, in row-3 and column-3, denoted as (R3, C3). The green line shows the desired current path from V_3 to I_3. But due to the LRS states of other cells in the crossbar, there can be other current flowing paths (as shown in Fig. 2(b) and 2(c)). Thus, the HRS state of the cell may erroneously be read as LRS. In general, sneak paths refer to some undesired parallel path in a crossbar through which currents can flow in addition to the desired path.

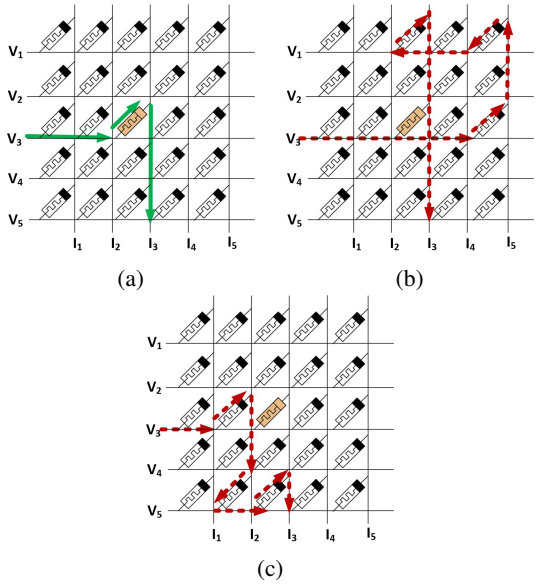


Fig. 2. A 5×5 memristive crossbar, where $V_3 = V_{read}$ is the volatge applied in row 3, and current sensed from column 3: (a) The crossbar without sneak paths, (b) Presence of a sneak path, (c) Presence of another sneak path.

It may be noted that if the cell at (R3, C3) in Fig. 2(a) is in LRS, and we want to read its state, the presence of sneak paths will further increase the current and hence the cell will be read correctly, even if there is an effect of sneak path. The effect of sneak path is prevalent in those cells which are in HRS state. We can say that in a crossbar with higher number of memristors in LRS, the chance of having sneak path effect is also higher.

A. Sneak Path of Various Lengths

In a memristor crossbar sneak paths of various lengths can exist. Consider sneak path of length k (where k is odd). In this case the current flowing from the voltage source to the current sensor will cover k memristors in series. As the value of k increases, the magnitude of sneak path current will be smaller. However, there may be multiple sneak paths appearing simultaneously, thereby the total current will get added up. Consider the crossbar of Fig. 3(a) where the cell at (R3,C4) is being read (dotted sky-blue square). Fig. 3(b), 3(c) and 3(d) show sneak paths of lengths 3, 5 and 7 respectively when some of the cells (denoted by dotted red circles) are in LRS.

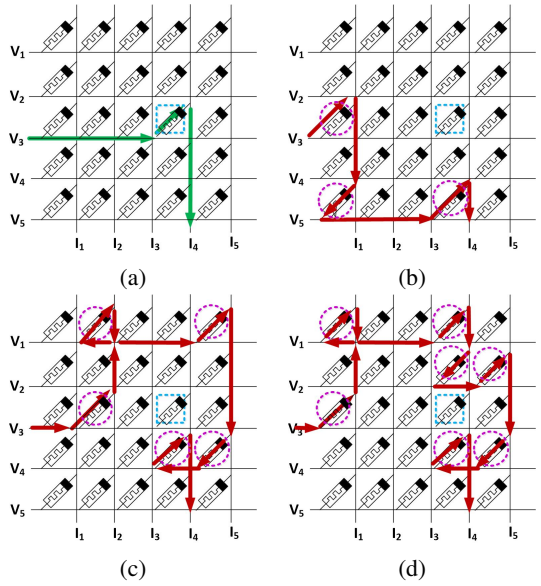


Fig. 3. (a) A 5×5 crossbar where the cell (R3,C4) is being read, (b) A sneak path of length 3, (c) A sneak path of length 5, (d) A sneak path of length 7

Proposition 1: A sneak path that can result in erroneous readout of a cell in HRS must be of odd length.

Proof: A memristor on the crossbar in LRS state provides a current conduction path between the row and column to which it is connected. A sneak path consists of a chain of memristors along a current flowing path, each of which is in LRS.

Every LRS memristor along the sneak path will switch the current from a row to a column, or from a column to a row. To read the state of a cell M_{ij}, we apply a voltage in row R_i and measure the current in column C_j. In other words, any sneak path that may lead to erroneous read operation of cell M_{ij} must switch the current initially entering the crossbar through R_i finally to column C_j. It is clear that only an odd number of memristors in the path can switch the current from a row to a column, because every pair of consecutive memristors in the path shall switch current from a row to a column, and again back to a row. Hence the result.

■

Each of the memristors (in LRS) along a sneak path will switch the current from a row to a column, or from a column to a row. Since the voltage is applied along a row, and the current is sensed along a column, there must be odd number of such current switching along a sneak path. For a crossbar of size $n \times n$, the number of distinct sneak path lengths will be $(n-1)$, and of sizes $k = 3, 5, 7, \ldots, 2n-1$. Thus for a 5×5 crossbar, we can have $5-1 = 4$ number of distinct sneak path lengths (with $k = 3, 5, 7, 9$). Similarly, for a 6×6 crossbar, we can have sneak paths of lengths $k = 3, 5, 7, 9, 11$. The total number of sneak paths for a $n \times n$ array can be calculated using the formula:

$$SP_{Total} = \prod_{i=1}^{\frac{k-1}{2}} (n-i)^2 \tag{1}$$

for $k = 3, 5, 7, \ldots, 2n-1$.

For a 6×6 crossbar, we can have maximum sneak path length of $6 - 1 = 5$. Hence according to eqn.(1), possible

values of k can be 3, 5, 7, 9 and 11. Total number of sneak paths of length 3, 5, 7, 9 and 11 can be estimated as [19]:

$$\begin{aligned} SP_3^{6\times6} &= (6-1)^2 \\ SP_5^{6\times6} &= (6-1)^2.(6-2)^2 \\ SP_7^{6\times6} &= (6-1)^2.(6-2)^2.(6-3)^2 \\ SP_9^{6\times6} &= (6-1)^2.(6-2)^2.(6-3)^2.(6-4)^2 \\ SP_{11}^{6\times6} &= (6-1)^2.(6-2)^2.(6-3)^2.(6-4)^2.(6-5)^2 \end{aligned}$$

where $SP_k^{m\times n}$ denotes the total number of sneak paths in an $m \times n$ crossbar of length k.

Thus, the total number of possible sneak paths of all sizes is given by

$$\begin{aligned} SP_{Total}^{6\times6} &= 25 + (25*16) + (25*16*9) + \\ &\quad (25*16*9*4) + (25*16*9*4*1) \\ &= 32825 \end{aligned}$$

It may be observed that even in a small 6×6 crossbar, there can be 32825 possible sneak paths of various lengths. The effect of sneak current will be greater for larger crossbars when higher number of cells are in LRS state.

B. Current Analysis in Memristor Crossbar

In this section we analyze the currents in the crossbar both in the presence and absence of sneak paths. We use SPICE simulation to estimate the currents. We have made certain assumptions while performing the current analysis. The memristors in the crossbar are assumed to be ideal (i.e. without any parameter variations) and two-valued, with a memristor being in either LRS or HRS. We have used the resistance values corresponding to LRS and HRS as 500Ω and $5M\Omega$ respectively, with a HRS-to-LRS ratio as 10^4. This is consistent with the parameters provided in recent publications (e.g. in [20]). For reading the state of a cell at location (i, j), we apply a voltage of $0.5V$ to row i of the crossbar, and sense the current in column j by connecting a 10Ω resistance to ground and then measure the voltage across it. In general, voltages can be applied to any arbitrary number of rows, and current sensed in any arbitrary number of columns. We have developed a software tool in C that reads the size of the crossbar, HRS and LRS resistance values, positions of the crossbar cells in LRS, and the rows where the read voltages of $0.5V$ have been applied. The tool directly generates a SPICE netlist file where the memristors are modeled as resistances, which is then simulated using LTSpice to get the measured current values.

Example 1: Consider a 8×8 crossbar as shown in Fig. 4, where the voltages along the rows are denoted as $(V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8)$, and the currents along the columns as $(I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8)$. If we want to read the state of the cell in the fourth row and fifth column, we have to apply the read voltage to V_4 and sense the current I_5. In the absence of sneak paths, if the cell to be read is in HRS, we get a low current value 0.42μA (due to the presence of multiple parallel paths in HRS), and if the cell is in LRS, we get a higher current value 0.98mA.

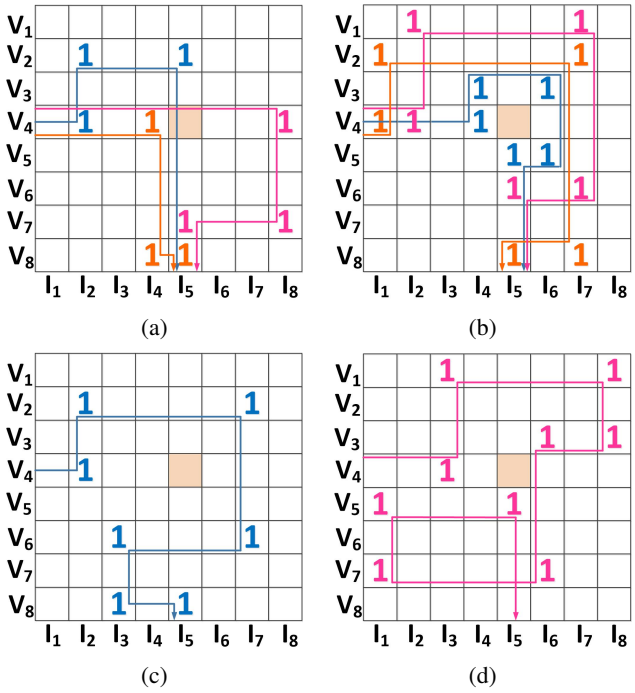


Fig. 4. A 8×8 crossbar where the cell (R4, C5) is being read: (a) Three sneak paths of length 3, (b) Three sneak paths of length 5, (c) One sneak path of length 7, (d) One sneak path of length 9.

Let us now consider the scenario where the cell at $(R4, C5)$ is in HRS. The following scenarios are depicted:

a) The cells marked in Fig. 4(a) in LRS contribute to three sneak paths of length 3: (R4-C2, R2-C2, R2-C5), (R4-C4, R8-C4, R8-C5), and (R4-C8, R7-C8, R7-C5).
b) The cells marked in Fig. 4(b) in LRS contribute to three sneak paths of length 5: (R4-C4, R3-C4, R3-C6, R5-C6, R5-C5), (R4-C2, R1-C2, R1-C7, R6-C7, R6-C5), and (R4-C1, R2-C1, R2-C7, R8-C7, R8-C5).
c) The cells marked in Fig. 4(c) in LRS contribute to one sneakpath of length 7: (R4-C2, R2-C2, R2-C7, R6-C7, R6-C3, R8-C3, R8-C5).
d) The cells marked in Fig. 4(d) in LRS contribute to one sneakpath of length 9: (R4-C3, R1-C3, R1-C8, R3-C8, R3-C6, R7-C6, R7-C1, R5-C1, R5-C5).

It may be noted that for a sneak path of length k, the current flows through k conducting memristors in series. The higher the value of k is, the smaller is the sneak path current. The current values in the presence of sneak paths of various lengths will be as follows:

- Normal read of a cell in LRS (with no sneak path):

$$I_{norm} = \frac{V}{R_{LRS}} \tag{2}$$

- Current in the presence of sneak path of length k:

$$I_{sp} = \frac{V}{k \times R_{LRS}} \tag{3}$$

Here, V denotes the read voltage, and R_{LRS} denotes the resistance of a memristor in LRS state.

This characteristic of sneak path may be taken into consideration when designing in-memory logic synthesis methods using memristors. It is worth mentioning that if we want to realize sneak path free designs, we need to compromise with respect to either area or computational steps. In the next section we discuss the logic synthesis and mapping techniques that are proposed in literature and we further analyze the effect of sneak paths on these techniques.

IV. Analyzing Logic Design Styles for Sneak Paths

Research on realizing functions on memristive crossbars has taken a great leap [4], [8], [9], [11]–[13], [15], [21]–[27]. In this section we shall discuss select in-memory logic design techniques using memristor crossbar. We shall further analyze these methods and identify the ones that are more susceptible to sneak paths.

A. MAGIC-based computing

Memristor Aided LoGIC (MAGIC) [8] is a computing method used to carry out logic operations directly on memristor crossbars. It is a stateful logic design style that is found to be efficient as compared to other memristor-based logic design styles like IMPLY [21]. One of the major benefits of MAGIC is that the inputs and outputs are stored in separate memristors as opposed to IMPLY. Also, MAGIC does not require any extra hardware like resistors, which makes it more suitable for crossbar realization.

1) Logic Design using MAGIC: Several prior works exist for realizing logic gate operations using MAGIC [13], [15], [17], [27]–[29]. The basic idea is to represent a function in terms of NOR/NOT netlist and then map the gate operations to the crossbar (in either row-wise or column-wise fashion). The gate mapping techniques can be broadly divided into *Serial Mapping* and *Parallel Mapping* [15].

Fig. 5(a) shows the NOR/NOT gate realization of a full adder, which consists of 12 gates across 7 levels. The gates can be mapped to the crossbar using single-row mapping as shown in Fig. 5(b). First the inputs are initialized in columns C_1, C_2, C_3 with the input values A, B, C_{in} respectively. Then all the gates in the netlist (12 gates in total), are mapped to a single row in columns from C_4 to C_{15}. The gate operations are then carried out in level-wise order by applying the necessary control signals to the crossbar columns.

This method will never give rise to sneak paths as only a single row is enabled during computation. Although we have shown row-wise operations in Fig. 5(b), it is also possible to map the gates in a single column and then carry out column-wise operations. One drawback of the single-row (single-column) method is that for larger functions, the number of time steps required will be high due to the sequential nature of execution.

Fig. 5(c) shows how the gates can be mapped across multiple rows of the crossbar [13]. Most of the methods in literature [13], [14], [27], [29] have used multiple-row mapping, with parallel gate evaluation across the rows. The same can be done for multiple-column mapping as well. This

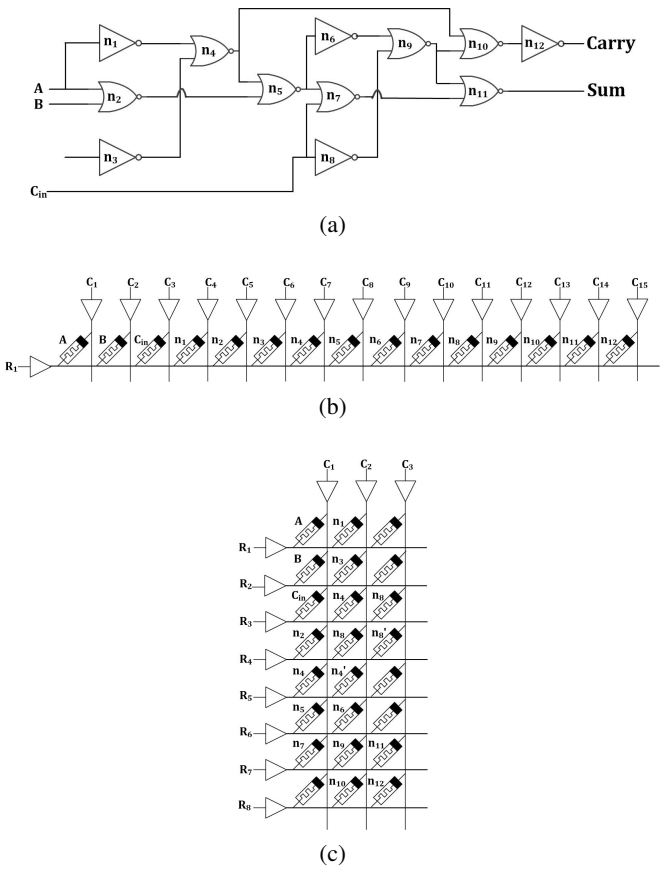


Fig. 5. Realization of a full adder: (a) NOR/NOT gate netlist, (b) Single-row mapping, (c) Multiple-row mapping.

requires proper alignment of the input and output memristors, and may necessitate copying the states of memristors for necessary alignments [13]. In general, a copy operation can be implemented by a read followed by a write; however, the correctness of the read operation becomes susceptible to sneak paths. However, in MAGIC we can implement the copy operation using two back-to-back NOT operations, which does not involve any reads and is hence resistant to sneak paths. The only time sneak paths can play a role is while reading the final outputs.

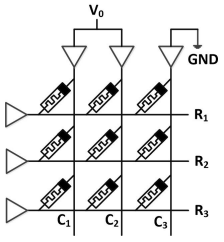


Fig. 6. Mapping of full adder in multiple crossbar rows [15].

Another gate mapping method based on MAGIC [15] uses a crossbar of fixed size to map the gates. For the full adder netlist of Fig. 5(a), we see that the maximum number of gates in a level is $n = 3$. Thus we need to evaluate the n gates in parallel, for which we require a crossbar of size $n \times 3$. Fig. 6

shows the corresponding mapping using the method of [15]. Due to the restriction imposed on the size of the crossbar, there will not be any sneak paths during read operation as discussed in *Lemma 1*.

Lemma 1: There will not be any sneak path during MAGIC NOR gate evaluation using row-wise parallel mapping.

Proof: The output of a NOR gate will be 0 if one or more of the inputs are 1; otherwise, the output will be 1. The crossbar can be represented as a binary state matrix, where the elements represent the memristor states. It is known that sneak paths in a crossbar can result in erroneous read only when there is the presence of an (imaginary) rectangle in the state matrix with three of the corners in state 1 and one of the corner in state 0 [7], where the cell in state 0 is being read.

Fig. 7(a) shows the mapping of n 2-input NOR gates in an $n \times 3$ crossbar for parallel evaluation, with the first two columns denoting inputs and the third column denoting output. Considering the cell marked '0' in the output column, reading the cell may become erroneous due to sneak paths if, say, a rectangle exists with the corner cells $a = b = c = 1$. However, the output of a NOR gate (here, c) will be 1, if all the gate inputs (here, b) are at 0. This contradicts the assumption, and hence the condition $a = b = c = 1$ can never be satisfied. The same logic can be extended to an $n \times m$ crossbar for parallel evaluation of n NOR gates with $(m-1)$ inputs each as shown in Fig. 7(b), for any values of n and m. ■

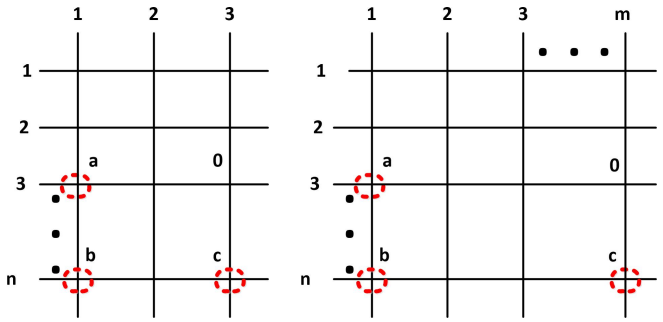


Fig. 7. Sneak path in MAGIC computation: (a) n 2-input NOR gates, (b) n $(m-1)$-input NOR gates.

B. Majority-based computing

The output of the 3-variable majority function evaluates to 1 if at least two of the inputs are at 1; otherwise it evaluates to 0. It can be represented as [30]:

$$MAJ(A, B, C) = AB + BC + CA \quad (4)$$

The set {MAJ, INV, 0, 1} is functionally complete, and can be used to implement any Boolean function.

The MAJ operation can be implemented using a single memristor as shown in Fig. 8(a), where the inputs A and B (as B', in complemented form) are applied as voltages across the terminals TE and BE [23]. A voltage $+V$ is applied for logic value 1, while $-V$ is applied for logic value 0. It is assumed that the memristor will be set (to 1) if $V_{TE,BE} = +2V$, and will be reset (to 0) if $V_{TE,BE} = -2V$. The third input C, and also the output, is mapped to the resistive state of the memristor. In other words, computation is carried out as a combination of stateful and stateless logic styles.

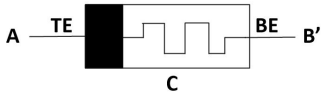
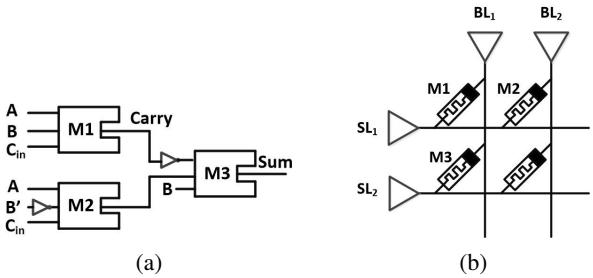


Fig. 8. Memristor realization of $MAJ(A, B, C) = AB + BC + CA$.

Example 2: The MAJ/NOT netlist for a full adder is shown in Fig. 9(a), which consists of three majority gates $M1$, $M2$ and $M3$, and two inverters. One possible crossbar mapping for carrying out the evaluation is shown in Fig. 9(b), with the required control signals depicted in Fig. 9(c).

It can be seen that the evaluation requires three write operations in steps T1 and T2 (with two parallel writes in step T1). During the MAJ operations in steps T3, T5 and T7, the controller applies the required voltages to the SL and BL lines. Lastly, the controller reads the state of memristor in steps T4, T6 and T8. During the read operations, there can be errors due to sneak paths. Although sneak path may not be an issue for such a small crossbar but as many intermediate read operations are required in MAJ based mapping, this method is vulnerable to sneak paths.



Step	Micro-operations	Operation
T1	$M1 = C_{in}$, $M2 = C_{in}$	Initialization
T2	$M3 = B$	Initialization
T3	$BL_1 = V(A), SL_1 = V(B')$	M1 = MAJ(A,B,C)
T4	Read $M1$	Read operation
T5	$BL_2 = V(A), SL_1 = V(B)$	M2 = MAJ(A,B′,C)
T6	Read $M2$	Read operation
T7	$BL_1 = V(M1'), SL_2 = V(M2')$	M3 = MAJ(M1′,M2,B)
T8	Read $M3$	Read operation

(c)

Fig. 9. Full adder realization: (a) MAJ/NOT netlist, (b) Crossbar mapping, (c) Required micro-operations.

To summarize, the MAGIC based approach require more memristors to carry out the operations (i.e. $m + 1$ for an m-input NOR gate), but does not suffer from sneak path issues during gate operations. Also, the final read operation can be made resilient to sneak paths for the $n \times m$ organization of the crossbar. In contrast, the MAJ based approach requires less number of memristors but suffers from sneak path issues.

V. Experimental Results

The gate mapping methods, both for MAJ gate decomposition and MAGIC NOR/NOT gate decomposition, have been implemented in C and run on a Linux-based machine with Core-i7 processor, 3 GHz clock, and 8GB memory. We have carried out experiments both on IWLS and ISCAS-85 benchmark functions.

For MAJ gate decomposition, functional specifications for the IWLS and ISCAS-85 benchmarks are converted into equivalent MAJ/NOT gate netlists using CirKit library [31]. The netlists are then topologically sorted and levelized. All the MAJ gates in a level can be evaluated in parallel on a $k \times k$ crossbar, where k denotes the maximum number of MAJ gates in a level. The number of cycles required to evaluate all the gates in a level is 3 (or 4), depending on whether all the inputs of the MAJ gates are uncomplemented (or complemented). It may be noted that read operations will be required at every level, which can be carried out in a single cycle. In other words, the number of MAJ gate levels and the number of read cycles required are the same. These read operations become vulnerable in the presence of sneak paths, which depends on the state of the crossbar as discussed in Section III.

Let N denote the total number of MAJ gates in the initial netlist. We assume the gates are topologically ordered in k levels $\{L_1, L_2, \ldots, L_k\}$. Some of the MAJ gates may be having complemented inputs. If n_i denotes the number of MAJ gates in level L_i, then

$$N = \sum_{i=1}^{k} n_i \tag{5}$$

Also, let $n_{i,NOT}$ denote the number of complemented inputs in the MAJ gates in level L_i. For evaluating the gates in each level we need 3 (or 4) cycles. Parallel evaluation of gates are possible and hence if there are *m* gates in a level we can evaluate all the m gates in 3 (or 4) cycles. For parallel evaluation we need to make sure that we map each gates in different rows/columns unless they have a shared operand. In this case, two or more resistive majority gates can be mapped to a single row for parallel evaluation.

In a similar way, for the MAGIC based approach, the IWLS and ISCAS-85 functional benchmarks are transformed into equivalent NOR/NOT gate netlists using the ABC tool. The gate netlists are similarly topologically sorted and levelized, and the gates mapped to the crossbar in level-wise fashion for evaluation [15]. It has been shown that for k gates mapped to a $k \times 3$ crossbar, all the gates can be evaluated in parallel without any sneak path effects [15]. The number of cycles required to evaluate all the gates in a level is 6.

Table I shows the results of gate mapping using the MAJ-based and MAGIC-based approaches. The first three columns of the table show the names of the benchmarks, number of primary inputs (#PI), and number of primary outputs (#PO) respectively. The next six columns show the results for the MAJ-based approach, in terms of the number of MAJ gates (#MAJ), number of MAJ gate levels (#Lev), number of cycles required to evaluate (C_{MAJ}), the number of read operations (#Read), the crossbar size (Csize), and the number of possible sneak paths of length 3 (#SP3). The next three columns show the results for the MAGIC-based approach, containing the number of NOR/NOT gates (#Gates), number of gate levels (#Lev), and number of cycles required for evaluation (C_{MAG}). The last column show the cycle overhead (% Overhead) of MAGIC-based method over MAJ-based method, calculated as a percentage using the formula $(C_{MAG}/C_{MAJ}) * 100$.

From the table it is evident that the number of cycles required to evaluate a function in the MAGIC-based approach is much higher compared to that for the MAJ-based approach. However, evaluation of gates across levels in the MAJ-based approach is vulnerable to sneak paths. As the size of the crossbar increases, the number of sneak paths also increases very rapidly (number of sneak paths of length 3 is shown for illustration), and hence there is high chance of erroneously reading a cell value. To summarize, we can say that the MAGIC-based approach may not be suitable for applications where lower latency is required. Although MAJ-based approach offers much lower latency, for practical realization we need to design sneak path aware mapping technique.

VI. Conclusion

In this paper we analyze sneak paths of various lengths in a crossbar array. We analyze two logic design styles based on MAJ and MAGIC for sneak paths. We observe that some of the MAGIC based logic design methods are resilient towards sneak paths. Although MAJ-based mapping is more effective in terms of number of cycles, it is susceptible to sneak paths. We also compare the cycle overhead of MAGIC-based method with respect to MAJ-based method. As a future work, algorithms for sneak path aware MAJ-gate mapping can be developed.

References

[1] L. Chua, "Memristor – the missing circuit element," *IEEE Trans. on Circuit Theory*, vol. CT-18, no. 5, pp. 507–519, 1971.

[2] D. Strukov, G.S.Snider, D. Stewart, and R. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[3] Y. Ho, G. M. Huang, and P. Li, "Dynamical properties and design analysis for nonvolatile memristor memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, pp. 724–736, 2011.

[4] E. Lehtonen, J. Poikonen, and M. Laiho, "Two memristors suffice to compute all boolean functions," *Electronic Letters*, vol. 46, no. 3, 2010.

[5] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, pp. 13–24, 2013.

[6] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics Journal*, vol. 44, pp. 176–183, 2013.

[7] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *Intl. Symp. on Information Theory*, pp. 156–160, 2013.

[8] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[9] F. Lalchhandama, M. Sahani, V. M. Srinivas, I. Sengupta, and K. Datta, "In-memory computing on resistive ram systems using majority operation," *Journal of Circuits, Systems and Computers*, vol. 31, no. 4, 2022.

[10] E. Linn, R. Rosezin, C. Kugeler, and R. Waser, "Complementary resistive switches for passive nanocrossbar memories.," *Nature Materials*, vol. 9, no. 5, pp. 403–406, 2010.

[11] S. Chakraborti, P. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of boolean functions using memristors," in *Proc. Intl. Design and Test Symp. (IDT)*, pp. 136–141, 2014.

TABLE I
EXPERIMENTAL RESULTS ON IWLS AND ISCAS-85 BENCHMARKS

Benchmark			MAJ Approach						MAGIC Approach			
Name	#PI	#PO	#MAJ	#Lev	C_{MAJ}	#Read	Csize	#SP3	#Gates	#Lev	C_{MAG}	% Overhead
9sym_d	9	1	61	6	23	6	12×12	121	403	23	138	600.0
con1f1	7	1	9	3	11	3	5×5	16	14	7	42	381.8
con2f2	7	1	10	4	15	4	5×5	16	15	7	42	280.0
exam1_d	3	1	7	2	8	2	2×2	1	11	7	42	525.5
exam3_d	4	1	11	2	7	2	6×6	25	16	9	54	771.4
max46_d	9	1	132	12	42	12	26×26	625	218	19	114	271.4
newill_d	8	1	21	5	19	5	9×9	64	35	13	78	410.5
newtag_d	8	1	10	3	11	3	6×6	25	20	7	42	381.8
rd53f1	5	1	11	4	14	4	5×5	16	22	8	48	342.8
rd53f2	5	1	20	5	18	5	6×6	25	26	10	60	333.3
rd73f1	7	1	41	5	18	5	9×9	64	119	16	96	533.3
rd73f2	7	1	19	3	12	3	6×6	25	31	10	60	500.0
rd84f1	8	1	80	5	19	5	17×17	256	205	21	126	663.1
rd84f2	8	1	22	4	16	4	8×8	49	36	10	60	375.0
sao2f1	10	1	36	4	15	4	15×15	196	59	14	84	560.0
sao2f2	10	1	45	4	15	4	17×17	256	66	12	72	480.0
sao2f3	10	1	31	4	15	4	14×14	169	44	13	78	520.0
sao2f4	10	1	33	4	15	4	13×13	144	115	16	96	640.0
sym10_d	10	1	80	6	23	6	16×16	225	640	28	168	730.43
t481_d	16	1	26	5	19	5	8×8	49	517	29	174	915.8
xor5_d	5	1	13	6	23	6	4×4	9	20	9	54	234.8
c432	36	7	95	23	84	23	9×9	64	250	37	222	264.3
c499	41	32	292	21	68	21	30×30	841	605	29	174	276.2
c880	60	26	259	17	52	17	29×29	784	520	30	180	346.1
c1355	41	32	292	21	69	21	30×30	841	605	26	156	226.1
c1908	33	25	296	28	102	28	28×28	729	582	40	240	235.3
c2670	233	140	437	19	63	19	82×82	6561	1059	30	180	285.7
c3540	50	22	824	33	106	33	86×86	7225	1421	54	324	305.7
c5315	178	123	1073	24	83	24	235×235	54756	1975	50	300	361.4
c6288	32	32	1867	116	348	116	255×255	64516	2717	120	720	206.9
c7552	207	108	1067	33	119	33	138×138	18769	2302	43	258	216.8

[12] P. L. Thangkhiew, R. Gharpinde, V. C. Paturi, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *11th Intl. Design and Test Symp. (IDT)*, pp. 142–147, 2016.

[13] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta, "A scalable in-memory logic synthesis approach using memristor crossbar," *IEEE Trans. on VLSI Systems*, vol. 26, pp. 355–366, 2018.

[14] P. L. Thangkhiew and K. Datta, "Scalable in-memory mapping of boolean functions in memristive crossbar array using simulated annealing," *Journal of Systems Architecture*, vol. 89, pp. 49–59, 2018.

[15] P. L. Thangkhiew, R. Gharpinde, and K. Datta, "Efficient mapping of boolean functions to memristor crossbar using magic nor gates," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 8, pp. 2466–2476, 2018.

[16] D. N. Yadav, P. L. Thangkhiew, and K. Datta, "Look-ahead mapping of boolean functions in memristive crossbar array," *Integration*, vol. 64, pp. 152–162, 2019.

[17] P. L. Thangkhiew, A. Zulehner, R. Wille, K. Datta, and I. Sengupta, "An efficient memristor crossbar architecture for mapping Boolean functions using Binary Decision Diagrams (BDD)," *Integration*, vol. 71, pp. 125–133, 2020.

[18] F. Gul, "Addressing the sneak-path problem in crossbar RRAM devices using memristor-based one schottky diode-one resistor array," *Results in Physics*, vol. 12, 2019.

[19] R. Joshi and J. Acken, "Sneak path characterization in memristor crossbar circuits," *International Journal of Electronics*, vol. 108, no. 8, pp. 1255–1272, 2021.

[20] F. Zahoori, T. Z. A. Zulkifli, and F. A. Khanday, "Resistive random access memory (rram): an overview of materials, switching mechanism, performance, multilevel cell (mlc) storage, modeling, and applications," *Nanoscale Research Letters*, vol. 15, no. 90, pp. 1–26, 2020.

[21] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, pp. 873–876, 2010.

[22] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference*, pp. 117:1–117:6, 2016.

[23] L. Amaru, P.-E. Gaillardon, and G. D. Micheli, "Majority inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conf.*, pp. 1–6, 2014.

[24] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 948–953, 2016.

[25] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, G. De Micheli, and R. Drechsler, "Endurance management for resistive logic-in-memory computing architectures," in *Design, Automation & Test in Europe*, 2017.

[26] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 7, pp. 1422–1435, 2018.

[27] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, "A staircase structure for scalable and efficient synthesis of memristor-aided logic," in *Asia and South Pacific Design Automation Conference*, p. 237–242, 2019.

[28] N. Talati, S. D. Gupta, P. S. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Trans. on Nanotechnology*, vol. 15, pp. 635–650, 2016.

[29] R. Hur, R. Ronen, A. Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2434–244, 2020.

[30] P.-E. Gaillardon, L. Amaru, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, "The programmable logic-in-memory (PLiM) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, p. 427–432, 2016.

[31] M. Soeken *et al.*, "The EPFL logic synthesis libraries," November 2019. arXiv:1805.05121v2.