3-5-2021

Memristor Crossbar Array Testing Using Sneak Paths

Rasika Dhananjay Joshi
Portland State University

Memristor Crossbar Array Testing Using Sneak Paths

by

Rasika Dhananjay Joshi

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
John M Acken, Chair
Marek Perkowski
Dan Hammerstrom
Steven Bleiler

Portland State University
2021

Abstract

Moore's law decline has paved the way to shift to new technologies at architectural and device levels. CMOS based technologies are facing many challenges with the growing demand for miniaturization. The growing heat dissipation is the major limitation for performance, energy efficiency and reliability with the increasing transistor count in integrated circuits. Manufacturing costs and process/memory performance gap have also grown steadily over the last several decades with the scaling down of the CMOS feature size. Memristor, a nanoscale device, has the potential to address the CMOS limitations because of its non-volatility, high density, low power operation, low cost per bit and CMOS compatibility.

The high density memristor crossbar structures are widely considered for performing memory operations, logic, stochastic and neuromorphic computations. However, these memristor based devices are prone to defects because of the non-deterministic nature of nano-scale fabrication. The motivation of my research is to develop an application independent methodology for testing memristor circuits for fault detection and fault diagnosis using a unique property of memristor crossbar circuits – sneak paths. Sneak paths are paths for current parallel to the intended path occurring in memristor crossbar architectures. This research characterizes sneak paths and sneak path currents as a function of size of the array, resistance values, input voltage and I/O switch vector. The equations I derived enable us to predict the sneak paths and sneak path currents for various array sizes to determine the constraints to resistive memristor circuits. The sneak path characterization work provides boundary conditions for applications that use memristor

crossbar arrays and provides insights into memristor crossbar testing. Using this characterization, a fault detection method is presented in the dissertation for fault detection of stuck-at low resistance and stuck-at high resistance faults using long sneak paths to result in shorter test vector sets. Long length sneak paths that enable fault detection with shorter test vector sets leads to improved test time. As the crossbar array size increases, the length of the longest possible sneak paths would also increase leading to improved test time compared to March testing. My fault diagnosis method using fault dictionary approach with improved test time is another highlight of this research. The results were demonstrated using LTspice simulations on resistive memristor crossbar circuits by varying resistance programming, IO switch-vectors, input voltage and size of the array.

The fault detection approach used for stuck-at LRS and stuck-at HRS fault detection is extended to test intermediate faults in memristor crossbar circuits. The method of selecting the detection limit for testing intermediate faults in crossbar circuits is presented in the dissertation using crossbar array simulations.

Dedication

To Aai, Baba, Shubhu, for their unconditional love and guidance throughout
To Sarvesh, for his constant support and encouragement
To the Almighty Lord Ganesh and Swami Samartha for giving me this opportunity

Acknowledgements

First and foremost, I would like to thank my advisor Dr. John M Acken for teaching me to be a researcher. His constant support and guidance throughout the study made it possible for me to reach this step in my academic career. His immense knowledge and patience have always inspired me and brought out the best in me. His continuous motivation and enthusiasm made this experience enjoyable and memorable for me. Thank you, Dr. Acken, for always believing in me.

I would like to thank my other committee members Dr. Perkowski, Dr. Hammerstrom and Dr. Bleiler for their valuable guidance and encouragement.

I would like to thank my husband Sarvesh without whom I would not have reached this step. His continuous support, encouragement and insightful discussions throughout the study were priceless. Thank you Sarvesh for being there for me always.

I would like to thank my parents for providing me all the educational opportunities, for always motivating me and for supporting me whenever I needed it. Thank you Aai, Baba and my brother Shubhu for your endless sacrifices and unconditional love for me. I would like to specially thank my mother for motivating me to pursue a Doctorate and without her I would not have reached this step. I would like to thank my in-laws as well for their continuous support and encouragement.

Last but not the least, I would like to thank Intel Corporation for supporting my study and for providing me with the opportunity to gain deeper understanding in my field of research. I am thankful to my managers and colleagues for their support and enthusiasm for my study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Background and Motivation

1.1 Introduction

CMOS technology is fast approaching its fundamental limitations with the growing demand for miniaturization. Excessive heat dissipation and increasing fabrication cost are primary concerns as the transistor density on the chip increases. The memory wall problem where the memory latency and bandwidth become insufficient for instruction and data transfers to the processor is also more prominent with the ever-increasing amount of data computations using conventional microelectronics technology. Conventional memory technologies such as Flash, DRAM, and SRAM are not able to keep up with the demand for scaling and low power. Memristor, an emerging nanoscale device, has the potential to address these issues in the near future.

In 1971, Leon Chua predicted the existence of a fourth fundamental element (the other three electrical elements are resistor, capacitor, and inductor) known as the memristor (short for memory resistor) [1]. Although he showed that such an element has interesting and useful circuit properties, no one presented a physical model or example of the memristor until 2008. R.S. William's team in Hewlett Packard Labs [2-3] then came up with a simple analytical example of memristance in thin film nanoscale devices. Memristors are one of the promising alternatives for next-generation memory technology due to their non-volatility, high density, low power operation, low cost per bit and CMOS compatibility. Memristor technology has become an attractive option for use in memory architectures, in-memory computing, logic, and neuromorphic applications. Memristor

devices find a broad range of applications in both analog and digital domains. Several research efforts have focused on expanding the memristor technology in the areas of design, test, memories, and memristor architectures for various applications. Crossbar structures are used for many of these applications for performing logic, memory, security, and stochastic computations.

1.2 Motivation for research

Nanoscale memristor devices are prone to defects due to the non-deterministic nature of nanoscale fabrication. It is necessary to test memristor devices for detecting memristor faults and to diagnose the location of such faults. Providing high quality and efficient test solutions is of great importance to enable the commercialization of memristor devices. The motivation behind my research is to generate a good quality testing methodology for memristor crossbar arrays that is application-independent. For example, the methodology will work for testing RRAM applications, for logic computations, neuromorphic applications and for user authentication systems etc.

1.3 Research Goals

My research focuses on analyzing the unique properties of memristor crossbar arrays specifically, sneak paths and sneak path currents for testing memristor circuits. Sneak paths are defined as current paths parallel to the target memristor path. My research work characterizes sneak path length and sneak path current as a function of the size of the array, memristor resistance values, input voltage and IO switch-vector. The sneak path characterization work provides boundary conditions for applications that use memristor

crossbar arrays and provides insights into memristor crossbar testing. A testing technique for memristor fault detection and fault diagnosis using sneak paths is proposed using the sneak path characterization work. The advantage of using a sneak path testing scheme is that multiple memristors can be tested at the same time by exploiting sneak path currents in crossbar arrays. Sneak path testing helps to reduce test time compared to the conventional March memory tests that target only one memristor device at a time, which consumes a lot of test time. My proposed testing technique addresses single stuck-at low resistance faults, single stuck-at high resistance faults and intermediate faults in memristor circuits. A new fault terminology, “intermediate faults” has been introduced that covers memristor resistances falling between low resistance and high resistance limits. The contributed test methodology aims to improve test time by proposing shorter tests by optimizing the set of IO test vectors and memristor resistance programming for a given size of the array. My research contribution includes the analysis of setting the right detection limit for detecting intermediate faults along with stuck-at low resistance and stuck-at high resistance faults.

1.4 Dissertation Structure

The dissertation is organized as follows. This chapter describes the introduction to memristor technology, the motivation behind the research and the research goals. Chapter 2 reviews memristor theory, crossbar arrays and their applications and memristor models. Chapter 3 describes sneak paths and sneak path currents in memristor circuits. This chapter also discusses my published sneak path characterization work in memristor crossbar circuits. Chapter 4 reviews test methodologies referenced in literature for testing

memristor circuits. The conclusions from these reviewed test methodologies are presented and my research objectives are discussed. Chapter 5 presents my published work for testing memristor faults in crossbar circuits using sneak paths for stuck-at low resistance and stuck-at high resistance faults. Chapter 6 extends the fault detection methodology used for stuck-at LRS and stuck-at HRS faults for testing intermediate faults in memristor circuits. It discusses my published work for analysis of setting the detection limit for intermediate fault detection in memristor crossbar circuits. Chapter 7 summarizes the contributions and conclusions of the dissertation. In addition, publications and future work are also discussed in this chapter.

Chapter 2

Introduction to Memristors and Memristor Crossbar Arrays

2.1 Memristor Introduction

The existence of the memristor was first theorized by Leon Chua in 1971 [1]. It was called the fourth missing element among the other three fundamental elements, namely resistor, capacitor, and inductor. These three two-terminal circuit elements already have established relationships between pairs of the four fundamental circuit variables, namely the current *i*, voltage *v* and charge *q*, and the flux-linkage φ. Chua noted that the number of equations connected to these pairs of circuit variables are six. Two of these relationships are defined by $dq = i$ and $d\varphi = v$. Three other relationships are defined by namely, resistor (the relationship between *v* and *i*), the inductor (the relationship between φ and *i)*, and the capacitor (the relationship between *q* and *v).* Chua invented the missing relationship between flux and charge as $d\varphi = Mdq$ where M is the memristance of the device as shown in Fig.1.

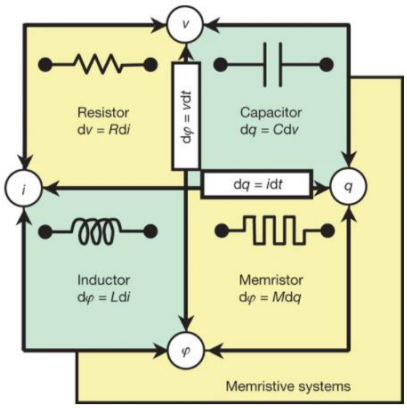


Fig. 1 Fourth missing element [2]

A memristor is a two-terminal passive resistive device whose resistances vary based on the history of voltages applied to it. In simple words, if a positive voltage is applied to the undoped end of this two-terminal passive device, the resistance decreases and if a negative voltage is applied, the resistance increases. The memristance (M, measured in Ohms) of the device is determined by the voltage V applied between the terminals as a function of time. The M of the device is expressed as shown in (1).

$$M = v(t)/I(t) \tag{1}$$

HP labs [2] developed memristors which consisted of 50-nm wide Titanium Oxide (TiO_2) thin film sandwiched between two platinum wires as seen in Fig. 2. This film consisted of two zones: First, un-doped low conductivity zone with an exact 2:1 ratio of oxygen to titanium. Second, doped high conductivity zone with oxygen deficient TiO_{2-x}. The memristor is modeled as two variable resistors connected in series. An internal state variable of the memristor denoted by "α" is equivalent to the ratio of the length of the doped region to the total width of the thin film.

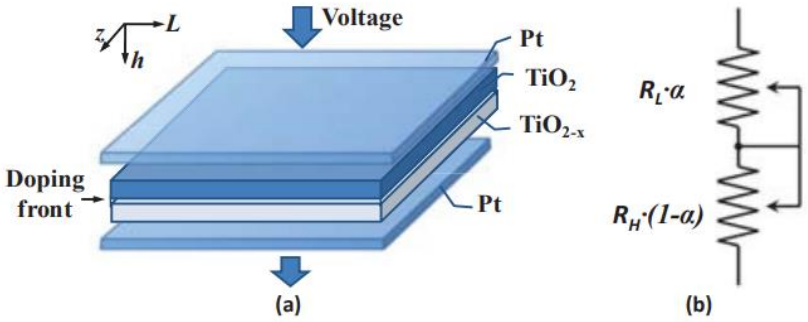


Fig. 2 a) TiO_2 thin film memristor structure b) equivalent circuit [4]

Applying a positive voltage ($v(t) > 0$) at the undoped end of the memristor lowers the resistance of the memristor due to the drifting operation of the oxygen vacancies into the un-doped region. Similarly, applying a negative voltage ($v(t) < 0$) increases the overall resistance of the memristor since now the oxygen vacancies drift in the opposite direction. Low resistance state (LRS) R_{on} occurs when $\alpha=0$ and high resistance state (HRS) R_{off} when $\alpha=1$. Thus, the total memristance M of the memristor is expressed in (2)

$$M(\alpha) = \alpha R_{on} + (1 - \alpha) R_{off} \tag{2}$$

The different memristance values exhibited by the memristor are used to represent different logic values. The memristor shows a non-linear behavior between the input voltage V and output current I. The hysteresis loop is shown in Fig. 3. The loops show the switching behavior of the device: it begins with a high resistance, and as the voltage increases, the current slowly increases. As charge flows through the device, the resistance drops, and the current increases more rapidly with increasing voltage until the maximum is reached. The result is an on-switching loop. When the voltage turns negative, the resistance of the device increases, resulting in an off-switching loop. Thus, the application of a positive bias voltage to the device leads to the switching of the resistance states from the High to the Low state, this switching is labeled as SET. A RESET switching corresponds to the exchange from the LRS to HRS state.

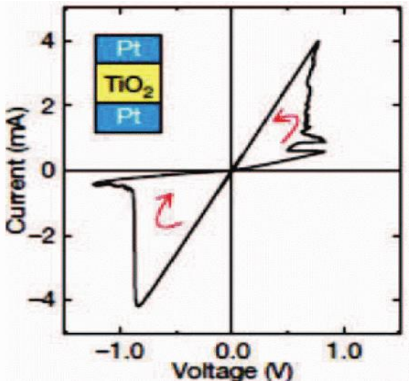


Fig. 3 Hysteresis Loop [5]

2.2 Memristor Write and Read Operations

The internal state variable of the memristor denoted by "$w(t)/D$" is equivalent to the ratio of the length of the doped region "w" to the total length of the TiO_2 film "D". The memristor can be defined at logic 0 when $0<w(t)/D<0.5$ and logic 1 when $0.5<w(t)/D<1.0$. The corresponding ideal output low and high levels are $w(t)/D=0$ and $w(t)/D=1$, respectively. In reality, to account for possible noise injections, a safety margin is left for each logic output: $0\leq w/D\leq O_L$, ($O_L=W_L/D<0.5$) for logic 0, and $O_H\leq w/D\leq 1.0$ ($O_H=W_H/D>0.5$) for logic 1. The region in between $O_L\leq w/D\leq O_H$ is an intermediate region that should be avoided for strict logic value read-write data integrity. Fig. 4 shows the situation where $O_L=0.4$ and $O_L=0.6$.

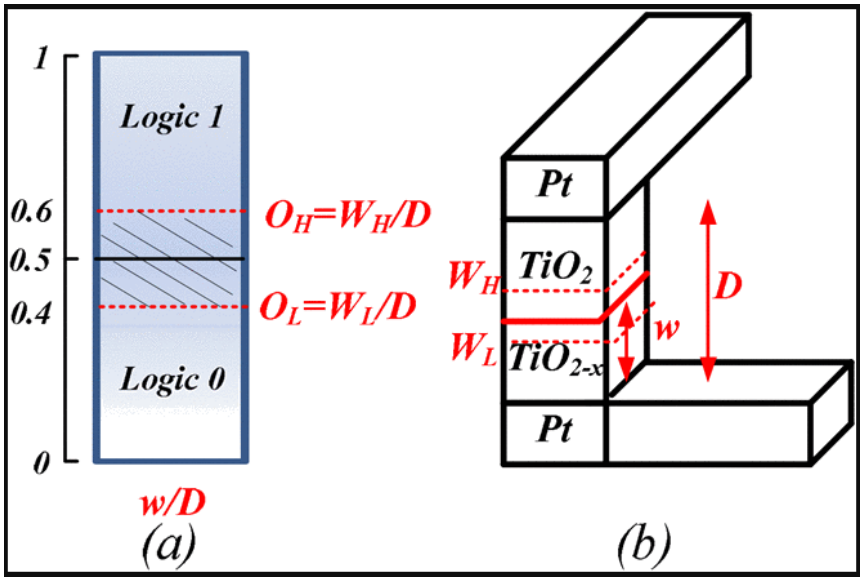


Fig. 4 (a) Memristor output levels, and (b) memristor 3D nano-structure [6]

1) Memory write operation:

A positive voltage is applied across the memristor for a fixed duration to write a logic 1. The duration of the pulse should be long enough to decrease the memristance from the logic 0 region to the logic 1 region. Similarly, to write a logic 0, a negative voltage is applied across the device long enough for the memristance to increase from the logic 1 region to the logic 0 region.

2) Memristor Read Operation:

Applying a voltage across the memristor causes the dopants to drift and change its memristance. To ensure that the resistance of the memristor is not changed during the read operation, a two-stage read operation is used [6]: Convert stage and sense amplifier stage. The convert stage is implemented by adding a series resistor to the memristor to convert the memristor state into a voltage signal since the current through the memristor carries the memristor state information. The second stage is to have a read pulse width limit so that

the memristance does not move beyond the safety margin. Fig. 5(c) shows the ideal read pattern is a negative pulse followed immediately by a positive pulse with the same magnitude and duration, creating a zero net change in memristance.

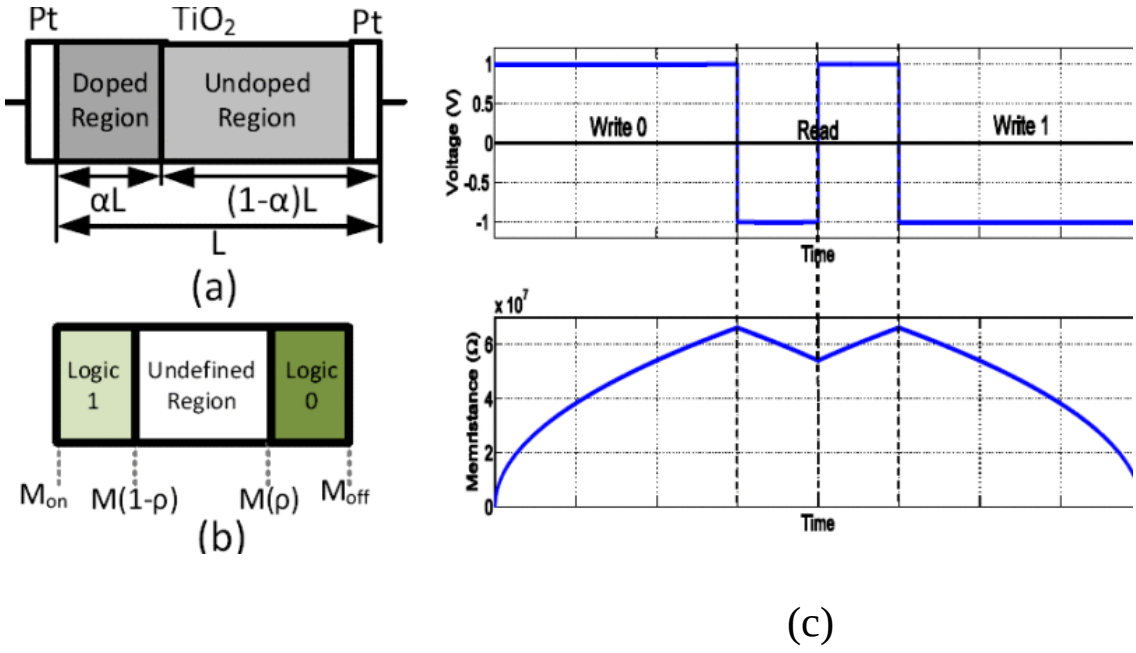


Fig. 5 a) Memristor model, (b) Memristance range for different logic levels, and (c) Variation of memristance due to voltage over time [7]

2.3 Crossbar Arrays

A crossbar array is a typical structure for many memristor implementations including memristor-based memories. Fig. 6 shows the schematic representation of a crossbar array with *m* wordlines (WLs) and *n* bitlines (BLs). It employs a memristor device at each intersection of horizontal and vertical metal wires without any selectors. A set of input voltages is applied on the word-lines (WLs) of the array and the output current is measured through each bit-line (BL). The device at the upper left corner (*Rj*) is the selected cell at

the intersection of the selected wordline and bitline. Unselected devices can be divided into three groups depending on whether they share an access line with ***Rj***. Devices sharing a line with ***Rj*** are also called "half-selected" devices. R*n* shares WL with *Rj* and *Rm* shares a BL with *Rj*. These are half-selected cells. *Rmn* shares no line with *Rj*; hence it is called as the unselected cell.

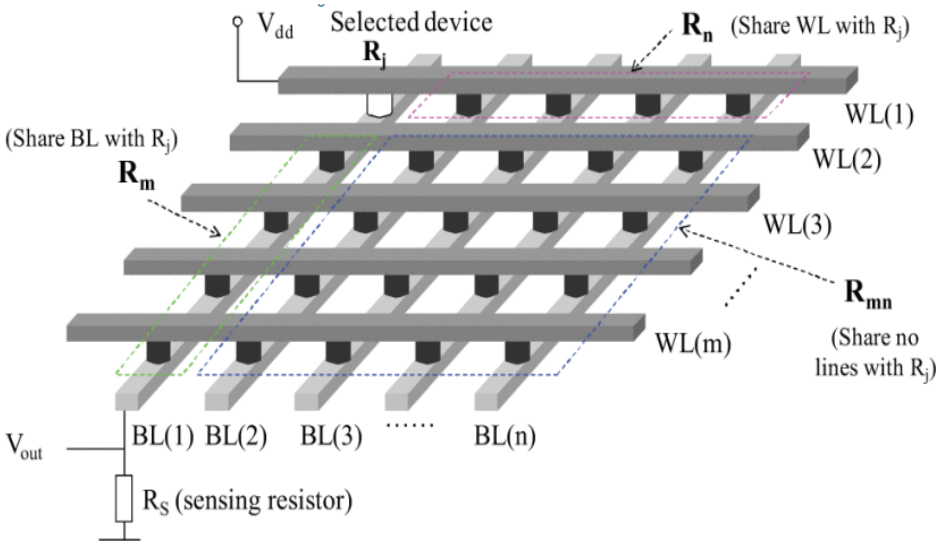


Fig. 6 Crossbar array with *m* WLs (horizontal line) and *n* BLs (vertical lines). *Rj* is selected cell. *Rn, Rm* are half-selected devices and *Rmn* is unselected device sharing no line with *Rj* [8].

2.3.1 Types of Memristor Crossbars

The generic structure of the memristor crossbar array is a 1M crossbar structure where the memristor devices are located at the intersection of each wordline and bitline of the array. The 1R-RRAM [9] resistive crossbar is an example of this structure that offers very high data density for data storage applications. The other crossbar structure commonly used is the 1T1R where a selector device, for example an access transistor is associated with the

memristive device. 1T1R [7] designs help eliminate sneak paths in the crossbar arrays but do not offer the same density as the 1R structure. Recently, a 2M-1M crossbar architecture has been proposed where each memristor cell has two access memristors and one target memristor [10]. 1D1R [11][12] structure is also used to suppress crosstalk by using external diodes. Rectifying memristors [13] have replaced the 1D1R structure due to its intrinsic diode-like behavior to suppress sneak paths. My research concentrates on resistive single memristive cell crossbar arrays (1M crossbar structure) to take advantage of sneak paths for testing memristor circuits.

2.3.2 Crossbar Applications

Researchers have made numerous efforts and initiatives to propose new crossbar architectures that offer high density, low energy consumption, low sneak path current effect and low wiring to outperform conventional memories. For example, the memristor-based memory cell can be utilized for high density memory and logic applications [10]. Another example is the multi-crossbar memristor architecture as an accelerator for matrix multiplications and handwriting recognition. This architecture achieves high speed and energy savings for 64x64 matrix multiplications [14]. Memristor crossbars have also been applied in user authentication systems [15], Resistive Random-Access Memories [RRAM] arrays [16], parallel computations [17], logic operations [18], neuromorphic systems [19] and Physically Unclonable Functions (PUFs) [20-21]. In summary, crossbar structures are used for many applications including logic, memory, stochastic computation, security PUFs, and neuromorphic applications.

Some of the crossbar array applications are discussed in this section.

1) 2M1M Crossbar Architecture: Memory [10]

This research in [7] presents a 2M1M crossbar architecture capable of memory and logic applications that provides a high area density in comparison with the state-of-the-art memristive memory architectures. It is a pure memristor-based memory cell and does not need CMOS transistors within the crossbar structure as seen from Fig. 7. The main advantages of this type of architecture are as follows:

- The read and write operations are done by the same memristor circuits without the need for additional circuitry within the memory fabric. Thus, the number of required elements is significantly reduced, simplifying the crossbar structure.
- The reading method does not need isolated access to the memristor node which reduces circuit wiring and leads to a very simple structure with less complexity.
- The proposed structure provides an effective gating mechanism by which memory elements can be partially isolated from the access line during the reading cycle, which considerably reduces the sneak path currents compared to its memory peers.
- The proposed memory structure provides acceptable speed and energy consumption in comparison with state of the art. Also, it has a higher density and less alternate current path effect comparing with its peer.

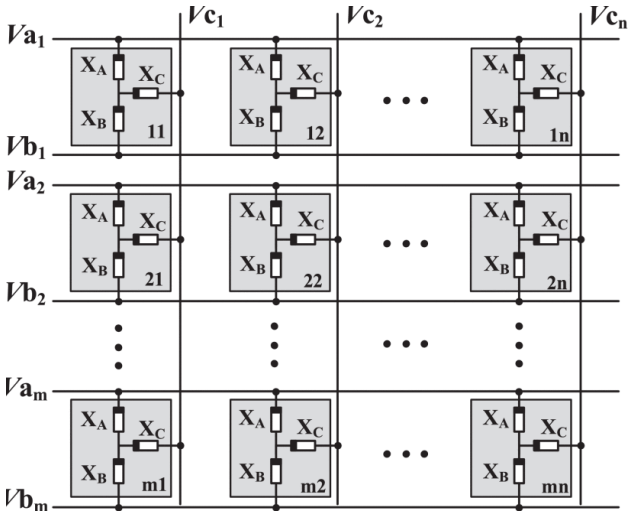


Fig. 7 Proposed 2M1M crossbar memory architecture [10]

2) Code Acceleration Using Memristor-Based Approximate Matrix Multiplier [14]

In this paper, the research focuses on building a memristor-based approximate accelerator to be used with general-purpose X86 processors for different applications such as matrix multiplication and handwriting recognition. Fig. 8 gives an overview of the memristor crossbar application for vector-matrix multiplication. V_1 is the input vector voltage to the columns of the crossbar, G is the matrix, V_o is the output voltage sensed by the trans-impedance amplifier with feedback resistor R_f. The high-level architecture of the proposed accelerator consists of multiple processing units that be used for performing independent computations through the extended instruction set architecture (ISA). These processing units consist of a memristor based crossbar, input-output buffers, and a logic circuit. To set up the accelerator, the program must initialize a processing unit which includes determining the size of the crossbar, configuring memristors' conductance, and determining the type of input numbers. The accelerator is compatible with signed complex number computations and with floating-point arithmetic. To validate the accelerator, it is

first utilized to multiply different matrices that vary in size and distribution. It is then used as an accelerator for accelerating the tiny-dnn, an open-source C++ implementation of deep learning neural networks. It provides more than 100× speedup and energy saving for 64 × 64 matrix multiplications.

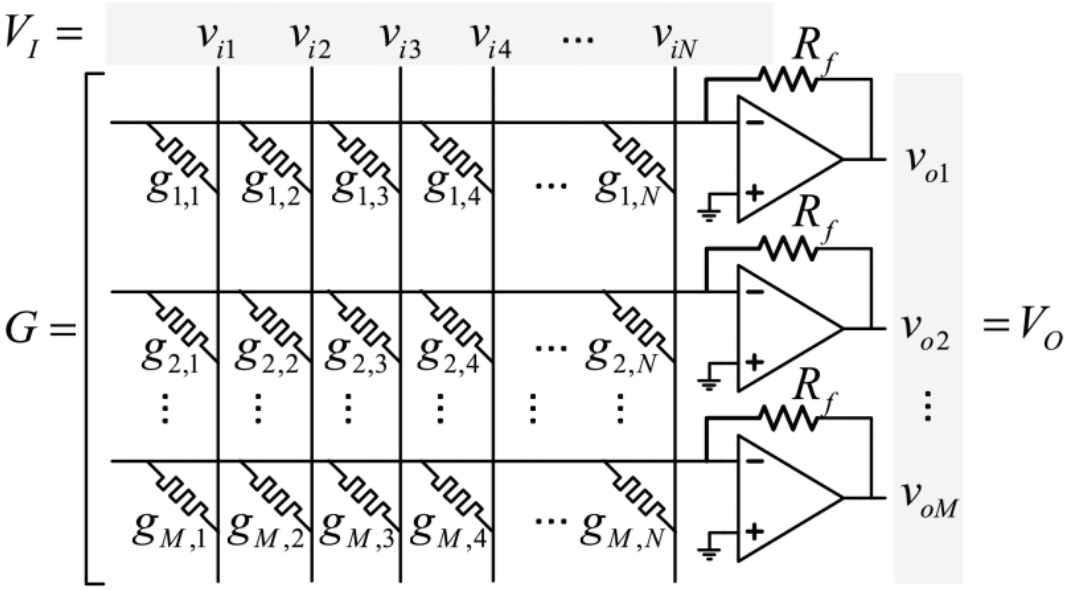


Fig. 8 Application of memristor crossbars for vector–matrix multiplication [14]

3) Automated synthesis of compact crossbars for sneak-path based in-memory computing [22]

The rise of data-intensive computational loads has exposed the processor-memory bottleneck in Von Neumann architectures. It has reinforced the need for in-memory computing using devices such as memristors. Boolean formula computing using sneak-paths in nanoscale memristor crossbars [23][24] suffers from the requirement to arrange memristors in dense nanoscale crossbars for ease of fabrication and the inability to produce compact crossbars for simple Boolean operations. The paper [22] is trying to answer two open questions using sneak paths in memristor crossbars for performing logical

computations: 1) The size estimation of the memristor crossbar that can compute a given Boolean formula using sneak paths 2) Synthesize compact crossbars for computing large Boolean formula using sneak paths. The authors demonstrate that the number of rows and columns required to calculate a Boolean formula is at most linear in the size of the Reduced Ordered Binary Decision Diagram (ROBDD) representing the Boolean function. The authors are the first to suggest the use of ROBDD for synthesizing compact memristor crossbars. They design sneak-path based memristor crossbars for circuits as large as 128-bit adders. For their experiments, they relied on HSPICE simulations.

4) Performance analysis of a memristive crossbar PUF design [25]

Physical unclonable functions (PUF) provide a unique hardware identifier where the intrinsic properties of the device are used to create a signature for security concerns including integrated circuit (IC) piracy, counterfeiting and secret key storage. A memristor crossbar based PUF circuit is described in this paper that utilizes variations in the write-time of the memristors as the primary entropy source. The main motivation to use memristor instead of CMOS for PUF designs is a lesser physical area and power dissipation. The proposed XBARPUF crossbar design schematic is shown in Fig. 9.

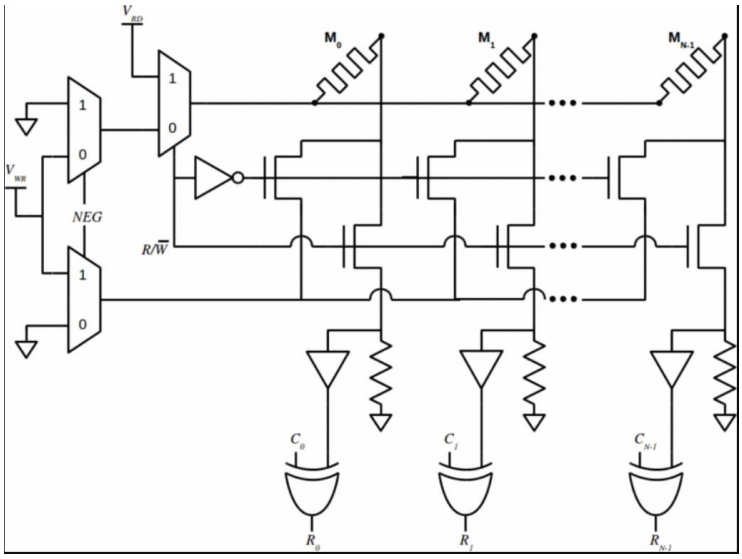


Fig. 9 Schematic of write-time memristive PUF circuit [25]

The amount of time taken for the memristor to SET during the write operation is the entropy source of the memristive PUF. The PUF circuit relies on the relative write-times of pairs of memristive circuits to generate the response. The write operation is governed by the challenge such that only one memristor in the pair is written at a time. This results in several unique combinations of altered memristors to select from while generating the signature. The sneak path currents in the crossbar design are also used for the response bit analysis. Results demonstrate strong statistical performance in terms of entropy, uniqueness, and uniformity [25].

5) ReVAMP: ReRAM based VLIW architecture for in-memory computing [26]

A general purpose computing platform has been proposed in this paper [26] that is based on Resistive RAM (ReRAM) crossbar array. This architecture supports VLIW (Very Long Instruction word) instructions to exploit parallelism in the memory array operations. The

ReRAM crossbar memory consists of 1S1R ReRAM devices arranged in a crossbar array fashion. Fig 10 shows the ReVAMP (ReRAM based VLIW architecture for in-Memory computing). It has two crossbar memories which are the instruction memory (IM) and Data storage and Computation Memory (DCM). It has a three-stage pipeline with instruction fetch (IF), instruction decode (ID) and execute (EX) stages. The instruction is fetched from the IM in the IF stage at the address held by the program counter. It is then loaded into the instruction register (IR) before the PC is updated. In the ID stage, the instruction is read to provide the inputs to the crossbar interconnect and write circuit.

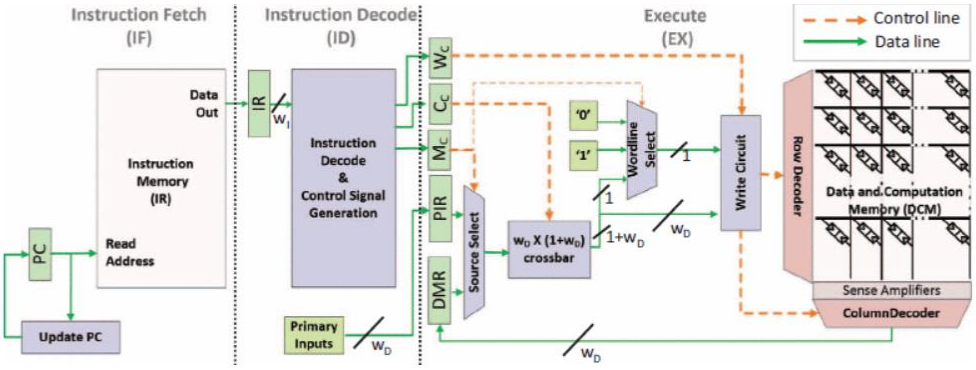


Fig. 10 ReVAMP Architecture [26]

The DMR (Data Memory Register) stores the data from the DCM. The primary input register (PIR) acts as a primary input data buffer. The crossbar interconnect consists of a set of multiplexers to select the number of wordline and bitline inputs as per the stored control signals. The write circuits in Fig.10 read the output of the crossbar-interconnect to determine the inputs to be applied to the row and column decoder of the DCM. The performance of the architecture is demonstrated in terms of delay, number of words and word utilization on the benchmark set.

2.4 Memristor Models

Several mathematical models of the memristors have been proposed to describe the behavior of memristors. This section will provide a brief description of different memristor models such as the linear ion drift model, the nonlinear ion drift model, and the ThrEshold Adaptive Memristor (TEAM) model.

1) Linear Ion Drift Model

The linear dopant drift model is widely utilized for memristor circuits and it provides a simple and useful approximation for memristor behavior [2]. Considering the TiO_2 memristor device as an example, the physical width D contains two regions, as shown in Fig 1(a). One of these regions has highly doped titanium dioxide with oxygen vacancies (TiO_{2-x}) and the other has undoped titanium dioxide (TiO_2). The device is modeled as two resistors connected in series and the region with the dopants has a higher conductance than the oxide region. The electric field generated through the applied bias is capable of drifting dopants based on the voltage polarity, therefore changing the resistance of the device. Assuming ohmic conductance, linear ion drift in a uniform field and ions having average ion mobility $\mu_{v,}$ equations (3) and (4) express the state variable and equivalent resistance

$$\frac{dw}{dt} = \mu_v \frac{R_{ON}}{D} i(t) \tag{3}$$

$$v(t) = \left(R_{ON} \frac{w(t)}{D} + R_{OFF} \left(1 - \frac{w(t)}{D}\right)\right) \cdot i(t) \tag{4}$$

where R_{ON} is the resistance when $w(t) = D$ and R_{OFF} is the resistance when $w(t) = 0$. On removing the bias, the dopants retain their place and the resistance of the device is preserved.

2) *Non-linear ion drift Model*

The behavior of the fabricated memristor device deviates significantly from the linear ion drift model and is very non-linear. Several non-linear ion drift models have been proposed, especially for logic computations [27-28]. Lehtonen [29] proposed a model based on the experimental results described in [30]. Equation (5) describes the relationship between current and voltage for this model.

$$i(t) = w(t)^n \beta \sinh\big(\alpha v(t)\big) + \chi[\exp\big(\Upsilon v(t)\big) - 1] \quad (5)$$

where α, β, Υ and χ are the experimental fitting parameters and *n* determines the influence of the state variable on the current. This model assumes asymmetric switching behavior and nonlinear dependence on voltage in the state variable differential equation as shown in (6),

$$\frac{dw}{dt} = \alpha \cdot f(w) \cdot v(t)^m \quad (6)$$

where α and *m* are constants, m is an odd constant and $f(w)$ is a window function. When the device is in ON state, the state variable w is close to 1 and $w = w(t)^n \beta \sinh\big(\alpha v(t)\big)$, describing a tunneling phenomenon. When device is in the off-state, the state variable w is close to 0 and $w = [\exp\big(\Upsilon v(t)\big) - 1]$.

3)Simmons tunnel barrier model

In [31], Pickett *at el*. presents a nonlinear memristive model of bipolar switching known as the Simmons tunnel barrier model. The model is derived from the experimental results of a dynamic testing protocol applied to a Pt-TiO_2-Pt memristor device. In this model, instead of two resistors in series like the HP model, a resistor is in series with the electron tunnel barrier as shown in Fig 11.

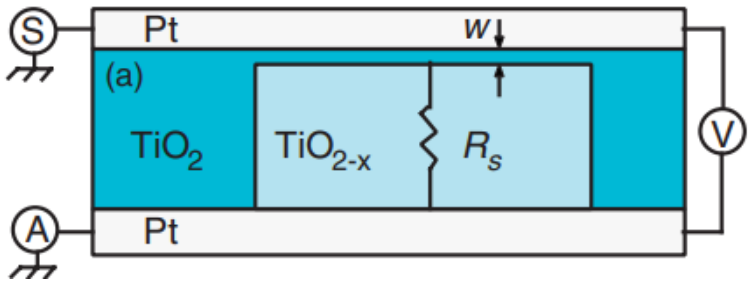


Fig. 11 Physical memristor structure based on the Simmon tunnel barrier model. *W* and R_s represent the tunneling barrier width and electroformed channel resistance respectively. *S*, *A*, and *V* represents the voltage source, ammeter, and voltmeter respectively [31].

The model exhibits nonlinear and asymmetric switching behavior due to the exponential dependence of the drift velocity of the ionized dopants on the applied current. In this model, the velocity of the oxygen vacancy drift can be explained by equation (7) for off-switching and (8) for on-switching.

$$\frac{dw}{dt} = f_{off} \sinh\left(\frac{i}{i_{off}}\right) \exp\left[-exp\left(\frac{w-a_{off}}{w_c} - \frac{|i|}{b}\right) - \frac{w}{w_c}\right] \tag{7}$$

$$\frac{dw}{dt} = f_{on} \sinh\left(\frac{i}{i_{on}}\right) \exp\left[-exp\left(\frac{w-a_{on}}{w_c} - \frac{|i|}{b}\right) - \frac{w}{w_c}\right] \tag{8}$$

where f_{off}, f_{on}, i_{off}, i_{on}, a_{on}, a_{off}, b and w_c are fitting parameters. f_{on} is an order of magnitude larger than f_{off}, and they both have effect on the magnitude of the change of $\frac{dw}{dt}$. i_{on} and i_{off} confine the current threshold effectively. a_{off} forces the upper bound and a_{on} forces the lower bound for $\frac{dw}{dt}$.

4)TEAM model

The TEAM model ThrEshold Adaptive Memristive Model [32] is a flexible and convenient model used for characterizing different memristive devices. In this model, a current threshold and tunable nonlinear dependence between current and derivative of the state variable has been suggested. The current-voltage relationship can be both polynomials as well as exponential. The derivative of the state variable for this model is expressed in (9).

$$\frac{dx(t)}{dt} = \begin{cases} k_{off} \cdot \left(\frac{i(t)}{i_{off}} - 1\right) \cdot f_{off}(x), 0 < i_{off} < i \\ 0, \qquad\qquad i_{on} < i < i_{off} \\ k_{on} \cdot \left(\frac{i(t)}{i_{on}} - 1\right) \cdot f_{off}(x), i < i_{on} < 0, \end{cases} \tag{9}$$

2.5 Summary of Chapter 2

In this chapter, memristor devices and memristor crossbar arrays were introduced. Some of the applications of crossbar arrays were also described. An application independent testing methodology is of great importance for testing memristor circuits used in these different crossbar applications. Finally, some of the memristor mathematical models were described in brief. These complex mathematical models are a function of voltage, time,

and frequency but all of them rely on the concept of R_{on} and $R_{off.}$ For my research, a simple resistive model is sufficient for testing purposes to represent whether a memristor is in a low resistance state or in a high resistance state.

Chapter 3

Sneak Path Characterization in Memristors

Note: Some of the contents of this chapter have been published below:

Rasika Joshi, John M Acken, "Sneak Path Characterization in Memristor Circuits", in *Journal of Electronics*, 2020. DOI: 10.1080/00207217.2020.1843716

Sneak path currents impact the performance of resistive crossbar array-based systems. It could have undesired effects on the reading and writing operations of the array based on the size of the array, memristor programming, input voltage and I/O switch vectors. Therefore, it is essential to characterize sneak paths and sneak path currents for understanding the constraints to the memristor crossbar operations. It will help to understand the design limitations when setting the size of a memristor array. A calculation model has been proposed for finding the length of different sneak paths for a given array size. These sneak paths have been analyzed based on the size of the array and the LRS/HRS memristor programming.

3.1 Introduction to Sneak Paths

Sneak paths are paths for current parallel to the primary current path occurring in memristor crossbar circuits. The bidirectional nature of memristors allows sneak paths in crossbar arrays. Sneak paths may corrupt the output current causing incorrect read and write operations in memory arrays. Fig. 12 shows a sneak path current example in a 3x3 crossbar circuit. The current flow highlighted in the bold blue line in Fig. 12 is the desired path of current flow through the selected cell at the intersection between the column and the row

of interest called as the primary current. Unfortunately, this ideal case is not the only path, and the current flows through an example sneak path highlighted in the dotted line in red, as shown in Fig 12. The sneak paths depend on the content of the memory and paths with lesser resistance and more memory content will sneak more current [33].

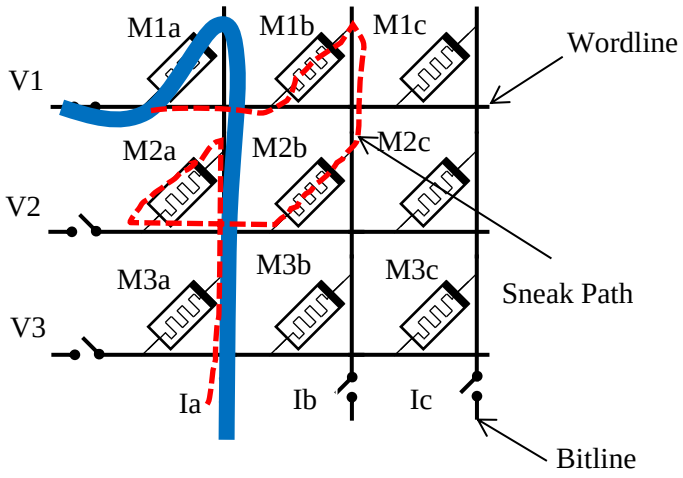


Fig. 12 Ideal case of current flow through a memristor cell and sneak path flow of current in a crossbar array.

3.2 Definition of IO switch-vector

The IO test vector set for a memristor crossbar array consists of the IO switch-vector settings for the rows (input) and columns (output). In a crossbar array of size *m*x*n*, the wordlines are the horizontal connections and the bitlines are the vertical connections. *m* is defined as the number of rows or wordlines, m_{open} is defined as the number of wordlines open, and m_{closed} as the number of wordlines closed. A wordline closed means that the input voltage source is connected to that wordline and a wordline open is not connected to a voltage source. When a wordline is closed, it is called a selected wordline. X_i is the switch state for the i^{th} row, where "1" is closed and "0" is open. *n* is defined as the number of

columns or bitlines, n_{open} as the number of bitlines open, and n_{closed} as the number of bitlines closed. A bitline closed means that the grounded current sensor on that column output is connected to that bitline and a bitline open is not connected to a grounded output current sensor. A bitline closed is called a selected bitline. Y_j is the switch state for the j^{th} column, where "1" is closed and "0" is open. In summary, the input state of $X_1X_2...X_i...X_m$ is combined with the output state of $Y_1Y_2...Y_j...Y_n$ to define the I/O switch-vector of $X_1X_2...X_mY_1Y_2...Y_n$.

3.3 Sneak Path Formula for number of sneak paths in crossbar arrays

The total number of sneak paths in a crossbar circuit is a function of the input conditions, array size and memristor programmed values. When all the memristors in the crossbar array are of equal resistance, all the sneak paths are three memristor long. The number of three memristor long sneak paths is expressed as n_{3mem} in (10):

$$n_{3mem} = m_{open} * m_{closed} * n_{open} * n_{closed} \quad (10)$$

When all the memristors in the crossbar array have equal resistance, the total number of three memristor long sneak paths in *m*x*n* circuit is the product of the bitline and wordline switches that are being switched on or off as shown in (10). $X_i, X_{i+1}, X_{i+2...}X_m$ are defined as the switches representing the wordlines to be switched on or off and $Y_j, Y_{j+1}, Y_{j+2....}Y_n$ are defined as the switches representing the bitlines of the crossbar array. For example, 3x3 circuit as shown in Fig. 12, the I/O switch vector is 100100. For a 2x2 circuit, considering the I/O switch-vector is $X_1X_2Y_1Y_2 =$ 1010, m_{open} =1, m_{closed} = 1, n_{open}= 1 and n_{closed} =1. The

total number of three memristor long sneak paths in this 2x2 array example is 1. For bigger memristor arrays, for example in an 8x8 circuit, the I/O switch-vector is $X_1X_2X_3X_4X_5X_6X_7X_8$ $Y_1Y_2Y_3Y_4Y_5Y_6Y_7Y_8$ = 1000000010000000, m_{open} = 7, m_{closed} = 1, n_{open} = 7 and n_{closed} = 1. The total number of possible three memristor long sneak paths for this 8x8 array example is 49. Some input/output combinations do not have sneak paths. When all the row switches or column switches are on, there would be no sneak paths. For there to be sneak paths, there should be $m > m_{open} \geq 1$ and $n > n_{open} \geq 1$ on the input and output respectively.

3.4 Analysis on Length of Sneak paths in crossbar arrays

The length of a sneak path is a function of input conditions, array size and memristor programmed values. HRS refers to the high resistance state and LRS refers to the Low resistance state of the memristor. A 3x3 crossbar array example with memristors labelled from *M1a, M1b, M1c* through *M3c* with all memristors having equal resistance values is considered. *V1, V2, V3* are the input voltages to the crossbar array and *Ia, Ib, Ic* are the output currents. In Fig. 13, the I/O switch-vector is $X_1X_2X_3$ = 100 and $Y_1Y_2Y_3$ = 100 for input voltages and output currents respectively. The primary current is the current through the selected cell or cells in the crossbar array. The selected cells are memristors at the intersection of the selected bitlines and the selected wordlines. $I_{primaryj}$ is the output current for selected cells on column j. Sneak path current is the current through the non-selected cells in the crossbar array (I_{sneakj}).

$$I_{outputj} = I_{primaryj} + I_{sneakj} \tag{11}$$

The output current is the sum of the primary current and the sneak path current as shown in (11). One metric for characterizing sneak path current is the relative magnitude of the primary current to the sneak path current and its effect on the output current.

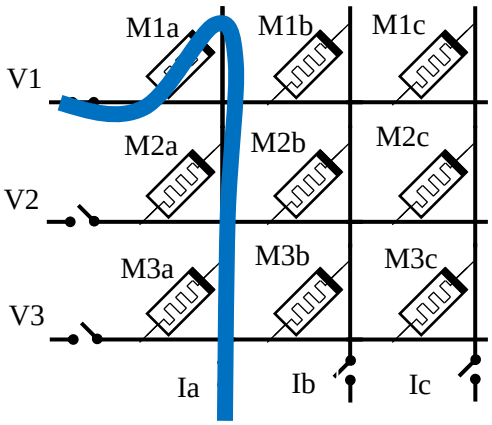


Fig. 13 3x3 Crossbar array with I/O switch-vector = 100100.

The circuit representation for this crossbar example is shown in Fig. 14. Notice that this circuit is not a mesh. The primary path is through selected cell *M1a*. The half-selected cells in this circuit are the ones sharing the line with *M1a*, namely *M1b, M1c, M2a and M3a*. The sneak paths through the electrical network are three memristor long as shown in Fig. 14, namely *M1b-M2b-M2a, M1b-M3b-M3a, M1c-M2c-M2a, and M1c-M3c-M3a*. The total number of possible I/O switch-vectors for an *m*x*n* crossbar circuit is expressed by (12), and the total number of I/O switch vectors that create sneak paths is shown in (13).

$$Total\ num\ I/O \text{ switch-vectors} = (2^m - 1) * (2^n - 1). \quad (12)$$

$$Total\ num\ sneak\ path\ I/O \text{ switch-vector} = (2^m - 2) * (2^n - 2). \quad (13)$$

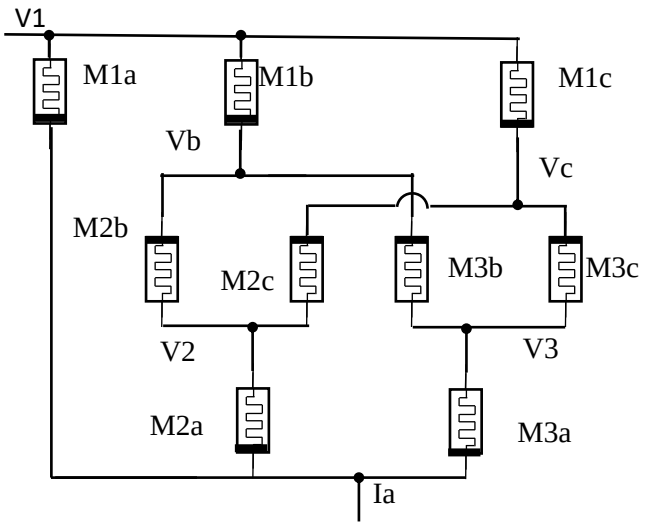
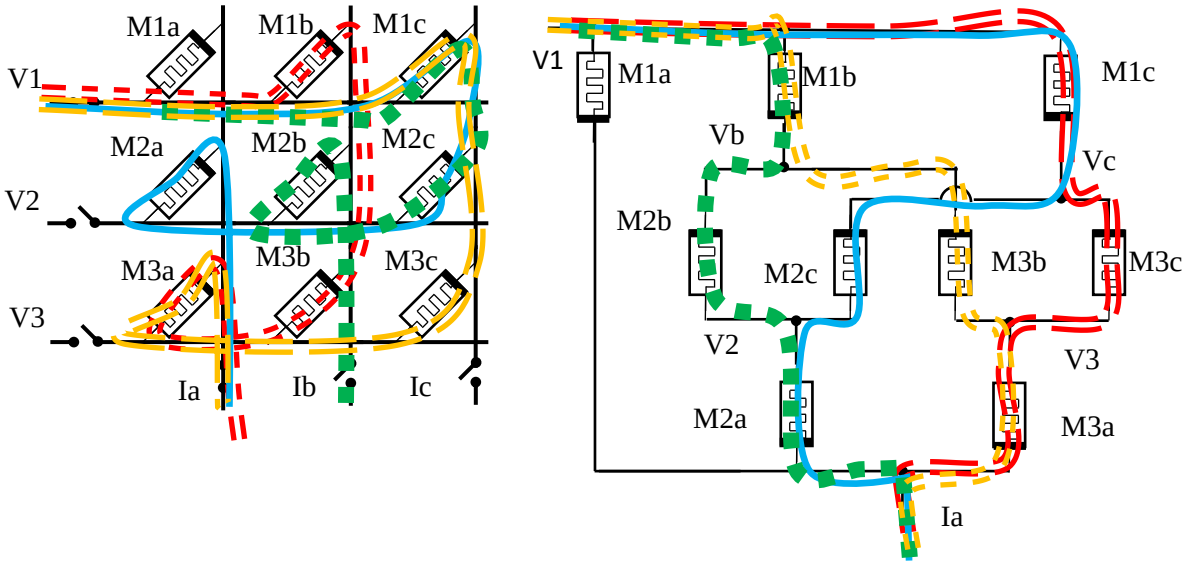


Fig. 14 Circuit diagram for 3x3 memristor array with I/O switch-vector = 100100.

The I/O switch-vectors where all wordlines and/or all bitlines set to floating condition are not being considered. For example, in a 4x4 crossbar circuit, the total number of possible functional I/O switch-vector cases are 225. Out of the 256 possible I/O switch vector combinations 31 are not functional because either all the inputs or all the outputs are disconnected. When all the inputs are 0 or all the outputs are 0, the crossbar array is disconnected and not functioning. There are 16 input switch vectors with all the outputs 0 plus 16 output switch vectors where all the inputs are 0, minus 1 for the repeated case of all zeroes on both input and output for a total of 31 non-functional I/O switch vectors. For the 4 x 4 crossbar circuit example, the total number of possible sneak path I/O switch vector cases is 196. As noted previously, if all the memristors have the same resistance values, then all the sneak paths are of length three. For example, consider the crossbar array shown in Fig. 15.



(a) Three memristor long paths in crossbar array

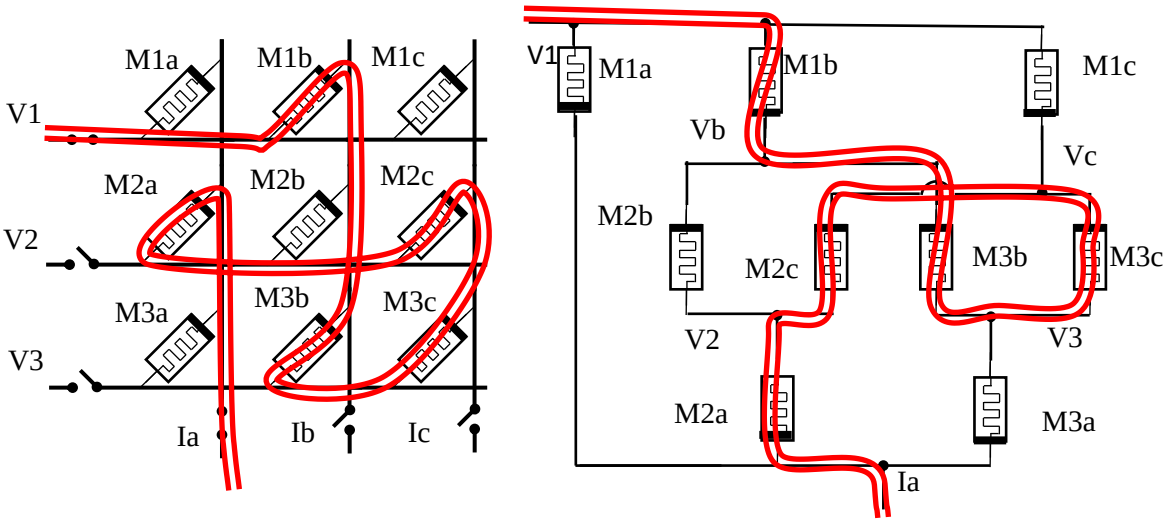
(b) Circuit equivalent showing three memristor long paths

	M1b-M2b-M2a
	M1b-M3b-M3a
	M1c-M2c-M2a
	M1c-M3c-M3a

Fig. 15 Sneak paths of length 3 in a 3x3 crossbar array with I/O switch vector = 100100

The three memristor long sneak paths are: *M1b-M2b-M2a*, *M1b-M3b-M3a*, *M1c-M2c-M2a*, and *M1c-M3c-M3a* as mentioned above. However, when the memristors are at different resistance values, some patterns can create longer sneak paths. The four possible five memristor long memristor sneak paths are: *M1c-M2c-M2b-M3b-M3a*, *M1c-M3c-M3b-M2b-M2a*, *M1b-M2b-M2c-M3c-M3a*, and *M1b-M3b-M3c-M2c-M2a*. One way to get the five-long path *M1c-M2c-M2b-M3b-M3a* is to have *M1b* and *M2a* in the HRS and the rest

in the LRS. Another programming to get the same long sneak path is *M1b*, *M2a*, and *M3b* in the HRS with the rest in LRS. To get the second example of the five memristor long sneak path, *M1c-M3c-M3b-M2b-M2a*, the memristors *M1b* and *M2c* are programmed to the HRS and the rest are LRS. To get the fourth example of path *M1b-M3b-M3c-M2c-M2a*, the memristors *M1c*, *M2b*, and *M3a* are in the HRS. This case is shown in Fig. 16 highlighted in red. There are many other patterns to get these and the other five memristor long sneak paths for a specific HRS/LRS programming pattern. Even with the five memristor long sneak paths there are still a total of four sneak paths. Specifically, three of the paths are three memristor long (*M1b-M2b-M2a*, *M1b-M3b-M3a*, *M1c-M2c-M2a*) and one of the paths is five memristor long (as shown in Fig. 16).



(a) Five memristor long sneak path in crossbar array

(b) Circuit equivalent showing five memristor long sneak path

Fig. 16 Sneak path M1b-M3b-M3c-M2c-M2a of length 5 in a 3x3 crossbar array with M1c=M2b=M3a=HRS and remaining memristors in LRS for I/O switch vector =100100.

As shown in Table 1 and (14), for this 3x3 array the total number of longest possible sneak paths is 4. The length of sneak paths is a function of the array size, the I/O switch-vector, and the programming of the individual memristors. The formulas for the different lengths of sneak paths for square array sizes (i.e. *m*=*n*) have been derived.

Table 1 Count of Possible Different Length Sneak Paths in Crossbar Circuits

Array size	**I/O switch-vector**	**3 long paths**	**5 long paths**	**7 long paths**	**9 long paths**
3x3	001 001	4	4	-	-
4x4	0001 0001	9	36	36	-
5x5	00001 00001	16	144	576	576
6x6	000001 000001	25	400	3600	14400
7x7	0000001 0000001	36	900	14400	129600
8x8	00000001 00000001	49	1764	44100	705600
9x9	000000001 000000001	64	3136	112896	2822400
100 x100	000……1 000……1	9801	94128804	8.86×10^{11}	8.16×10^{15}

From Table 1, the long length sneak paths for any array size are calculated using the formulas below. For three memristor long, five memristor long, seven memristor long and nine memristor long sneak paths, the possible number of sneak paths can be calculated as below:

$$n_{3mem} = (n-1)^2 \quad (14)$$

$$n_{5mem} = (n-1)^2 * (n-2)^2 \quad (15)$$

$$n_{7mem} = (n-1)^2 * (n-2)^2 * (n-3)^2 \quad (16)$$

$$n_{9mem} = (n-1)^2 * (n-2)^2 * (n-3)^2 * (n-4)^2 \quad (17)$$

When all the memristors are programmed to the same resistance value of LRS or HRS, all the sneak paths are three memristor long. Therefore, every three memristor path parallel to the target memristor is a sneak path. For a *n*x*n* array, that is $(n-1)^2$ paths. This is derived because there are $(n-1)$ parallel memristors on the selected bitline and $(n-1)$ parallel memristors on the selected wordline to the target memristor. There is a different unselected memristor connecting each selected bitline memristor to each selected wordline memristor. There are $(n-1)$ unselected bitlines and $(n-1)$ unselected wordlines hence $(n-1)^2$ different memristors each resulting in a unique sneak path. To achieve a five memristor long sneak path, two or three of the memristors need to be programmed as HRS and the remaining are programmed to LRS. The other two equations follow similar path with more memristors in HRS.

Considering a memristor crossbar circuit consisting of all low resistance programming or all high resistance programming, the following observations for the number and length of sneak paths have been made

(1) If $n_{closed} = n$ OR $m_{closed} = m$ for inputs and outputs switches then there will be no sneak paths.

(2) If there is at least one n_{open} in the input AND at least one m_{open} in the output, the length of the sneak path is always of three memristors. This only applies when all the memristors are of equal value.

(3) When the memristors are not of equal value, the length of the longest possible sneak path (L_{max}) is expressed by (18):

$$L_{max} = 2 * n - 1 \quad \text{for n} \leq \text{m} \qquad (18)$$

(4) The minimum number of n_{open} or m_{open} on inputs or outputs sets the path length. If n_{open} = 2 in the input AND m_{open} = 2 in the output, the longest possible sneak path will be of five memristor length. Similarly if n_{open} = m_{open} = 3 then a maximum possible length of seven memristors and so on. This applies when certain patterns of memristors are being programmed to high/low resistance value as discussed in Fig. 16.

(5) The lowest number of rows or columns sets the maximum length of the sneak paths. For example in 2x3, 2x2, and 3x2 arrays, the longest length sneak path is three memristors.

3.5 Analysis of Sneak Path Currents in Crossbar Arrays

The sneak path current significantly impacts the design space for a memristor array. The research addresses two questions about sneak path impact: 1) The effect of different parameters and conditions on the behaviour of sneak paths that in turn affect the memristor crossbar array performance; 2) the impact of sneak path current with respect to size of memristor array, memristor resistances, I/O switch-vector, high/low programming of the memristors. These effects set the boundaries and limits for the design space. A similar sneak path current analysis has been described in [34]. Tang Zhensen *et al*. [34] analyses the worst-case scenario for read operations that include the worst-case selected location and worst-case data pattern based on the effect of sneak paths and interconnection resistances. However, my research's characterization is for various cases (not just the worst case) and resistance values. In [8], the parameters for limiting the array size were first chosen such as the line resistance and non-linear device characteristics and then the sneak

path current was analysed. In contrast, the proposed sneak path current analysis helps to determine the boundary conditions for crossbar arrays. Also, the formula for sneak path calculation derived in [8] is based on equal values of memristors. In contrast, my proposed characterization includes various programming of memristor values and varying I/O switch-vectors. Cassuto paper [35][36] gives mathematical proofs for a sneak-path free readout and coding schemes to eliminate sneak paths. Their schemes are concentrating on eliminating sneak paths for read error-free column readouts in their application. Whereas, my research analyses the impact of different memristor parameters and operating conditions (such as I/O switch-vector and programming patterns for memristors in high resistance state (HRS) and low resistance state (LRS) on the behaviour of the sneak path currents, and in turn, the memristor output current for any given crossbar array application. For example, the size of the memristor array can be determined for a memristance range before the sneak path current interferes with the crossbar operation. For my research, memristor arrays with bidirectional memristors are being considered, and not rectifying memristors. The conditions/parameters looked at are the high/low programming of memristors, I/O switch-vectors (row and column selectors) – non-selected, selected and half-selected cells in the memristor crossbar circuit, square-non/square arrays, and ranges of memristor resistance.

Based on the results, curve fitting models for calculating the sneak path currents as a function of array size, memristor resistances, memristor programming, I/O switch-vectors and input voltage are determined. The characteristics of the complete relationship between memristor parameters (such as array size, high-low memristance ratios) and the sneak path current will provide a basis for design implementation trade-offs.

3.5.1 Sneak Path Current Calculation tool

The sneak paths were found using a python based sneak path calculator. The sneak path calculator gives the sneak paths for varying array sizes based on the sneak path algorithm discussed in theory. The calculator also generates a text file output directly fed into the LTspice simulator tool to simulate the output currents based on the resistance and input voltage values. Sneak path current analysis is based on these simulated currents. Here are the following steps to generate the LTSpice circuit using the python calculator:

(1) The number of wordlines and number of bitlines is taken from the user to create the I/O switch vector combination.

(2) The input voltage and the LRS/HRS value of the memristors are also taken as user inputs.

(3) The number of sneak paths is determined using the equation (14) through (17).

(4) The target memristor is identified from the I/O switch vector combination. All the memristors excluding the target memristor are used to create the sneak path circuit.

(5) Sneak paths are generated based on the model discussed for equation (14).

(6) The circuit node connections are assigned based on the sneak path information.

(7) The python generator outputs a file that is fed to the LTspice tool.

(8) The sneak path current is simulated in LTspice based on the user input voltage and resistance values.

3.6 Sneak Path Current Analysis w.r.t size of array and resistance programming

Crossbar array applications require quantitative analysis of array characteristics especially sneak path currents to provide boundary conditions for designing crossbar arrays. In the following sections, sneak path currents have been analysed with respect to different parameters such as the size of the array, resistance programming, input voltage and input/output conditions. My research presents equations based on simulation results for determining the sneak path current as a function of the memristor array parameters. The derived equations will help with the sneak path current prediction of any array size for understanding the constraints to the memristor crossbar operation.

3.6.1 Resistance Programming

Various technologies and models use different values of memristor resistance. Table 2 shows different published ranges of resistances for the low and high resistance states. For our initial data analysis, the low resistance value of 10KΩ and high resistance values of 1MΩ, 50KΩ and 500KΩ have been used. The design decisions based upon sneak path current are a function of the range of resistances, the ratio of the high-low resistances, and the ratio of memristor resistance to the line resistance. The sneak path current characterization includes the effects of different memristor resistance values.

Table 2 Low and High Resistance Values for Memristors

Paper	**HRS High Resistance**	**LRS Low Resistance**
Fault Modeling and Parallel Testing for 1T1M Memory Array [37]	1MΩ	10KΩ
A bridge technique for memristor state programming [38]	100KΩ	100Ω
A Test Method for Finding Boundary Currents of 1T1R Memristor Memories [39]	500KΩ	10K-160KΩ
Modeling Detection, and Diagnosis of Faults in Multilevel Memristor Memories [40]	200KΩ	100Ω
Sneak-Path Testing of Crossbar-Based Nonvolatile Random-Access Memories [41]	121KΩ	121Ω
Sneak Path Based Test for 3D-Stacked One Transistor N-RRAM array [42]	500KΩ	10KΩ
Design and Optimization of a Strong PUF Exploiting Sneak Paths in Resistive Cross-point Array [43]	10MΩ	100KΩ
Sneak-Path Based Test and Diagnosis for 1R RRAM Crossbar Using Voltage Bias Technique [9]	200KΩ	100Ω

Sneak path current is specifically affected by R_{HRS}/R_{LRS} ratio. As quoted in [10][44][45], the typical ratio of R_{HRS} to R_{LRS} is 10^2 - 10^3. Analysing Table 2 confirms their ratio. My research contribution uses R_{HRS}/R_{LRS} ratios from 2 to 100. The paper [46] evaluated R_{HRS}/R_{LRS} ratios from array sizes from 10 to 50. They found out by spice simulation that the sneak path current needs to be limited as a function of R_{HRS}/R_{LRS} ratio.

Another reference [32] recommends a high ratio between R_{HRS}/R_{LRS} to store distinct Boolean data in a memristive device.

The simulated sneak path currents for the different IO switch-vector combinations discussed in the sections below are of two types. The first type is called the *total sneak path current,* defined as the sneak path current measured for all the sneak paths in a crossbar array with single bitline output. The second type of sneak path current is called as *sneak path current per array bitline output* where the sneak path current is measured from a single output in an IO switch-vector having multiple bitline outputs.

3.6.2 Sneak Path Current for IO switch-vector $m_{closed} = n_{closed} = 1$

Considering the I/O switch-vectors set to one switched on input and one switched on output (i.e. $m_{closed} = 1$ and $n_{closed} = 1$), the following trend in the sneak path current values is observed in Fig. 11. The sneak path current measured for this IO switch-vector is the total sneak path current. The sneak path currents (I_{sneak}) is plotted on the Y axis and the size of side of the crossbar array (n) on the X axis. From the graph, the equation that is observed is $I_{sneak} = 49.8 * n - 73.9$ µA [$I_{sneak} = A*n + B$] where A = ~50µA and the offset B = ~ −74 µA. Here A is the function of the resistance in the crossbar circuit and the input voltage applied to the wordlines and is equal to (0.5/R) *V. The sneak path current relationship for Fig. 17 is shown in (19). The offset is derived from curve fitting.

$$I_{sneak} = \frac{0.5}{R} * V * n + offset \tag{19}$$

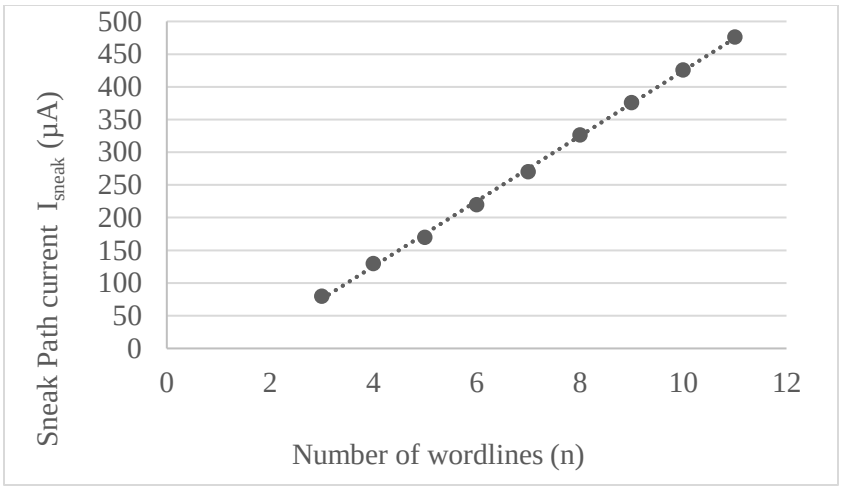


Fig. 17 Sneak Path current analysis for one input ON and one output ON [m_{closed} = n_{closed} =1] for LRS programming of 10KΩ where m=n.

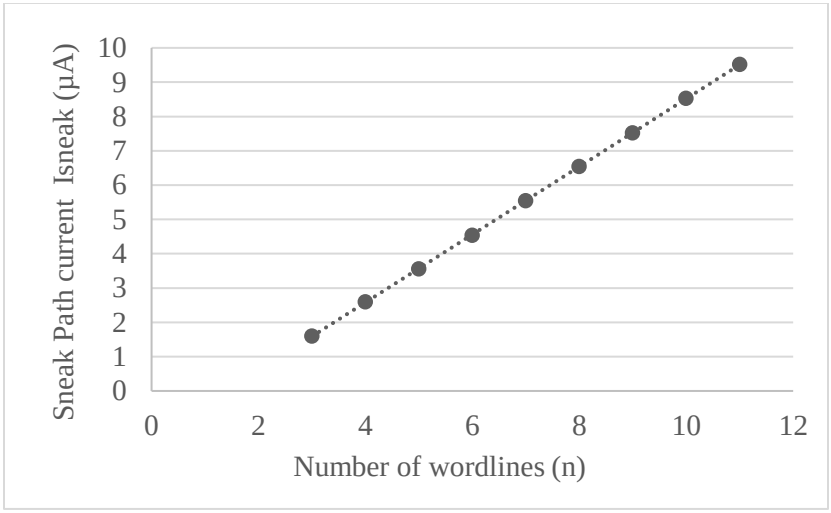


Fig. 18 Sneak Path current analysis for one input ON and one output ON for [m_{closed} = n_{closed} =1] HRS programming of 500KΩ where m=n.

This equation is followed for other memristance values of R. Fig. 18 shows the sneak path current values with HRS programming of 500KΩ. Here, the equation of the graph seen is I_{sneak} = 0.99 * n −1.37 µA where the slope A = 0.5/500KΩ = 1µA and offset B = −1.37 µA. A linear curve has been observed for these two graphs with 10KΩ and 500KΩ

resistance values with a slope of 0.5/R. Predictions can be made on sneak path current for larger array sizes and a variety of high/low resistance values using equation (19).

3.6.3 Sneak Path Current for IO switch-vector m_{closed} =m-1, n_{closed} =n-1

Considering another case of I/O switch-vector such as 011 011 for a 3x3 memristor array where $m_{closed} = m$–1=2 and $n_{closed} = n$–1=2, the sneak path current results for each array output can be observed for low resistance of 10KΩ and high resistance of 500KΩ in Fig. 19 and Fig. 20 respectively. For this IO switch-vector combination, the sneak path current measured is the *sneak path current per array bitline output* .The equation of the sneak path current for this IO switch-vector can be expressed as shown in (20).

$$I_{sneak} = C * n^{-0.85} \tag{20}$$

The scaling factor C = 64.5μA for LRS programming of 10KΩ and C = 1.29μA for HRS programming of 500KΩ. The impact of the offset value decreases with the increase in the size of the array. A power curve is observed for these two graphs with an exponential constant of -0.85 for these low and high resistance programming values.

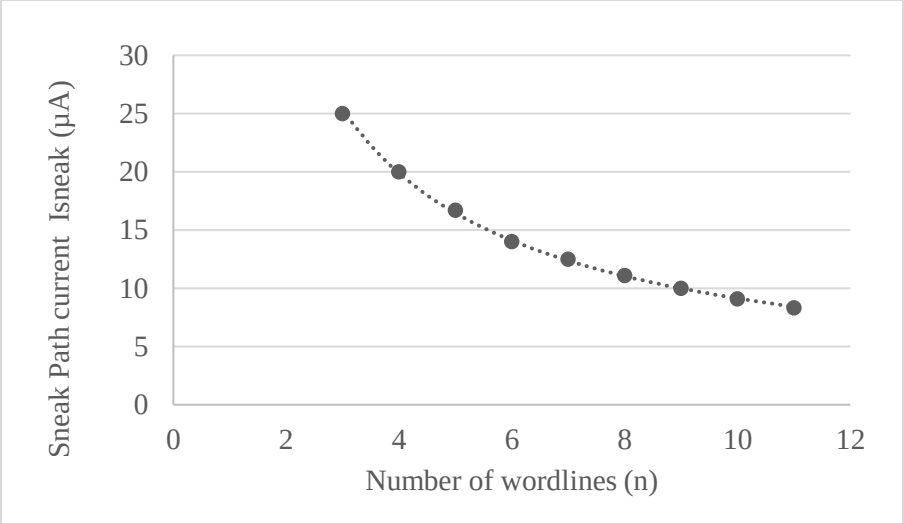


Fig. 19 Sneak Path current analysis for m–1 inputs ON and n–1 outputs ON [m_{closed} = m-1] for LRS programming of 10KΩ where m=n.

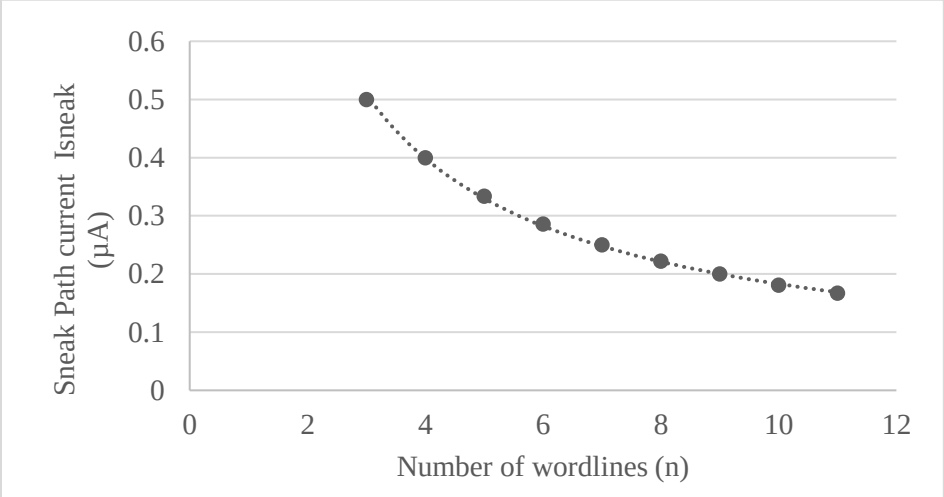


Fig. 20 Sneak Path current analysis for m–1 inputs ON and n–1 outputs ON [m_{closed} = m–1] for HRS programming of 500KΩ where m=n.

As shown in Fig. 19 and Fig. 20, the sneak path current may seem decreasing with the increase in the size of the array. However, one note is that this sneak path current is analysed per array output. For a 3x3 array, the sneak path current for 011 011 I/O switch-vector combination is 25µA per each array output with input voltage = 1V and all LRS =

10KΩ. There is a total of four sneak paths that are three memristor long for 011 011 crossbar circuit, namely *M2a-M1a-M1b, M2a-M1a-M1c, M3a-M1a-M1b, and M3a-M1a-M1c*. The current measurement through *M1a* is 50μA corresponding to a voltage of 0.5V which get divided with memristors *M1b* and *M1c* connected in series, each of them yielding a sneak path current output of 25μA. If the size of the array is increased from 3x3 to 4x4 (IO switch-vector combination is 0111 0111), now the voltage drop through *M1a* is 0.6V which gets divided through three memristors namely *M1b, M1c and M1d* connected in series. The sneak path current through each of the three bitline output is 20μA. As the size of the array increases, this voltage drop through *M1a* reduces the sneak path current output through each bitline output. The total sneak path current for the 3x3 crossbar circuit (011 011) with the two bitline outputs is 50μA and for 4x4 crossbar circuit (0111 0111), with the three bitline outputs it is 60μA. Using these 3x3 and 4x4 crossbar examples, it can be observed that as the total sneak path current increases, the sneak path current per bitline output decreases. The total sneak path current helps to drive designs decisions for estimating the size of the array and the individual sneak path current through each bitline output helps with setting the detection limit for testing memristor faults.

3.6.4 Sneak Path Current for IO switch-vector m_{closed} =1, n_{closed} =n-1

Simulations have been performed with varied array sizes by keeping the input test vector with one switched on input (m_{closed} =1) and the output test vector with all outputs switched on but one (n_{closed} = n–1). In this analysis, the input and output vectors have different switches as opposed to the previous examples. For example, the I/O switch-vector for 3x3 crossbar array for this analysis can be represented as $X_1X_2X_3$ = 001 and $Y_1Y_2Y_3$ = 011. A

logarithmic increase in the sneak path currents is observed as the size of the array increases as shown in Fig. 21.

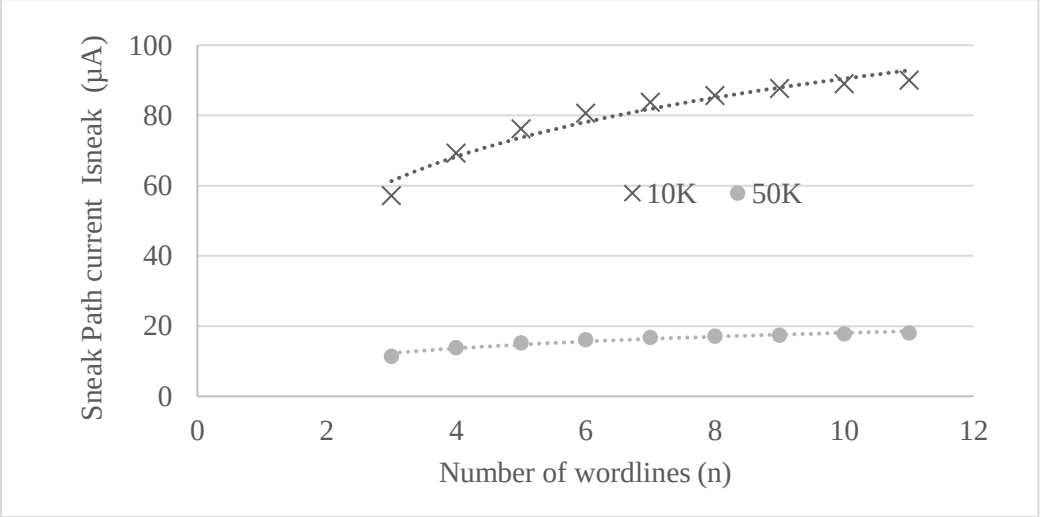


Fig. 21 Sneak Path current analysis for single input ON and all outputs ON except one [m_{closed} = 1 and n_{closed} = n–1] for HRS and LRS programming of 50KΩ and 10KΩ respectively.

Sneak paths currents for a given crossbar size have been analysed by increasing the number of switched on outputs (m_{closed} = 1, *2, 3…m*–1) and keeping a single switched on input vector (n_{closed} = 1). Fig. 22 shows a 6x6 crossbar array example for sneak path current analysis based on the input pattern on the X axis. From the plot, a peak in the sneak path currents is observed at $X_1X_2X_3X_4X_5X_6$ = 000001 and $Y_1Y_2Y_3Y_4Y_5Y_6$ = 000011.

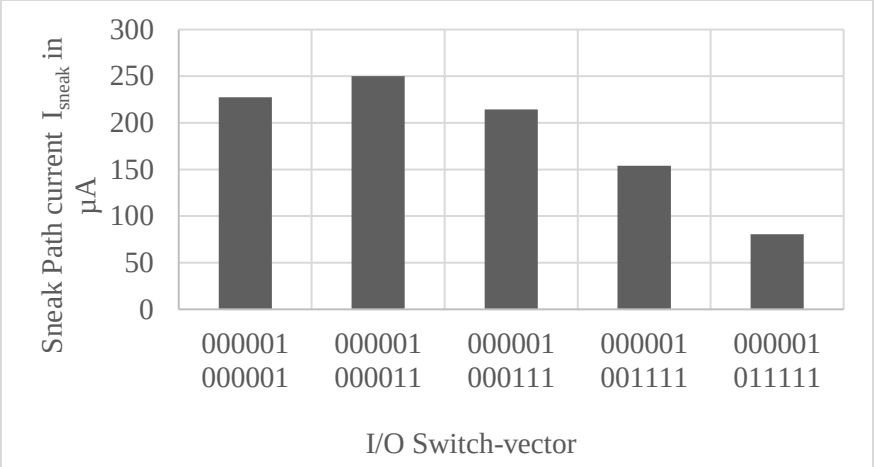
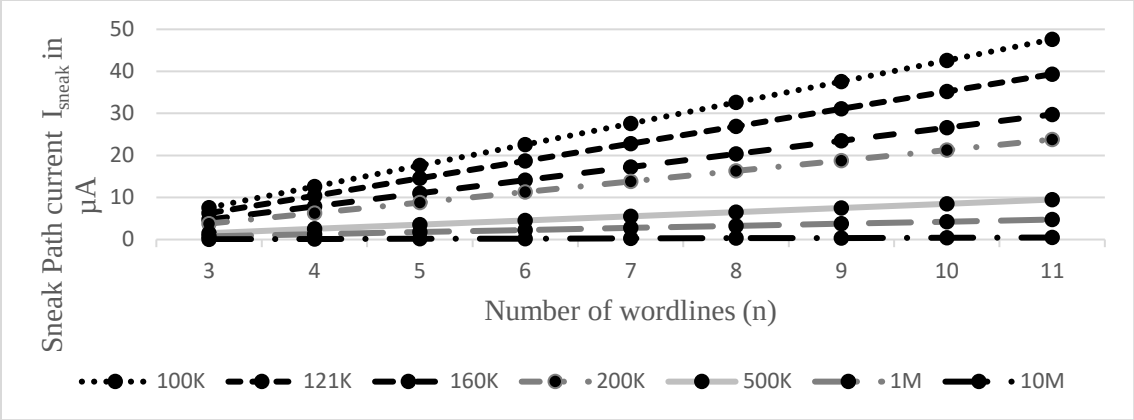


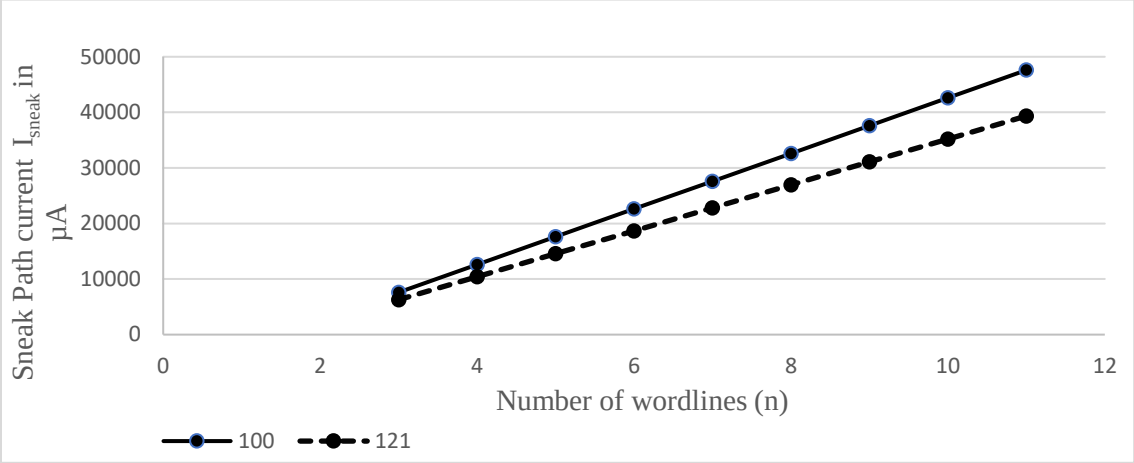
Fig. 22 Sneak Path current analysis with variation in I/O switch-vector (m_{closed} =1 and n_{closed} =1,2,3,4,5) for 6x6 crossbar array for 10K resistance programming.

3.6.5 Sneak Path current ranges

Fig. 23 shows the plot of the values for sneak path current based on the resistances in Table 2. For the sneak path current data analysis, low resistance value of 10KΩ and high resistance values of 1MΩ, 50KΩ, and 500KΩ is used. As quoted in [46], the reverse leakage current or the sneak path current ranges from 0µA to 20µA for ReRAM array sizes from 10x10 to 20x20. The authors in [46] indicate that the reverse current varies with the change in design parameters. The authors in [47] had a range of sneak path current ranging from 2.5µA to 20µA, with the number of array bits increasing from 10k to 10M. Our research contribution reports the sneak path current varying with the size of the array, memristor resistance, and IO-switch vector, as discussed in equations (12) and (13). In paper [44], the authors have given equivalent leakage circuit models for different 3D resistive RAM layers with complementary resistive cells. They have compared their reading margins and leakage resistance to that of one-layer crossbar memory.



(a)



(b)

Fig. 23 Sneak Path current analysis for one input ON and one output ON [n_{closed} = m_{closed} =1] for LRS and HRS programming of resistances in Table 2 where n=*m*.

3.6.6 Sneak Path current analysis as a function of Resistance

Sneak path currents of the crossbar circuits can be characterized by varying single memristance to a high resistance value while keeping other memristances to a low value and vice versa. Using the same crossbar circuit as in Fig. 15 with IO switch-vector=100100, LTspice simulation is performed with equal memristance values of 10KΩ for all the memristors and input voltage of 1V. The total simulated sneak path current value is 80µA. Consider memristors *M2a* and *M3b* carrying 40µA and 20µA current respectively are

switched to an HRS value of 1MΩ. On varying *M2a* to HRS the total sneak path current decreases to 50.5µA and on varying *M3b* to HRS the total sneak path current drops to 71.6µA from the original LRS sneak path current of 80µA. A similar experiment was done keeping all the memristor values as high resistances and switching one of the memristors in the unique sneak path to a low resistance value. The observed difference in the sneak path current was much higher in this case. The sneak path increased from 0.8µA to 1.14µA on switching one of the resistances to LRS. Similar experiments were also performed on larger array sizes of 4x4 and 5x5 memristor circuits. Switching a memristor to a low resistance value while others are programmed at higher values largely impacted the total output currents values compared to the impact of switching a memristor to a high resistance while others are programmed at low resistance. Sneak path currents vary based on the memristor programming and the location of the memristor in the crossbar array.

The following rules are used to select a single memristor to be set to a high resistance while all the remaining memristors are in a low resistance. The rules are also applicable to a single memristor set to a low resistance with the remaining memristors in high resistance.

- When $m_{closed} = m - 1$ AND $n_{closed} = n - 1$ vary the resistance of the middle memristor in the three memristor long sneak path for maximum impact to the sneak path current.
- When $m_{closed} = 1$ AND $n_{closed} = 1$, vary the resistance of the first or the last memristor in the three memristor long sneak path for maximum impact to the sneak path current.

- When $m_{closed} = m-1$ OR $n_{closed} = n-1$, vary the resistance of the last memristor in the three memristor long sneak path for maximum impact to the sneak path current.

3.6.7 Sneak path current analysis in comparison with the Primary current path

In Table 3, the sneak path current and the primary current for I/O switch-vectors of a 3x3 crossbar circuit are presented. For $X_1X_2X_3Y_1Y_2Y_3$ = 001001, the output current is the sum of the primary current of 100µA and sneak path current of 80µA when all memristors are programmed to a low resistance value of 10KΩ. As seen in Fig. 17, the sneak path current increases linearly with the size of the array for a given resistance programming and a given input voltage. However, the primary current stays constant even with the increase in the size of the array for a given resistance programming and input voltage. For example, for a 10x10 crossbar array, the sneak path current is 426µA with the primary current of 100µA. Similarly, for $X_1X_2X_3Y_1Y_2Y_3$ = 011011, the primary current will now be through two memristors each at every output, doubling the current from the previous case. In this case, the sneak path current will decrease exponentially with the array size as presented in Fig. 19. The sneak path current for a 10x10 array will be 9.09µA for this I/O switch-vector from Fig. 19 as opposed to the 3x3 crossbar value of 25µA in Table 3 for each bitline output. Based on the derived equations for a given I/O switch-vector, it can be seen how the sneak path current is a dominating factor of the output current. Predictions can be made for the sneak path current of various array sizes for a given I/O switch-vector, resistance programming and input voltage. Resistance programming plays a major role in affecting the sneak path currents. This sneak path analysis will help to drive design decisions such

as deciding the size of the array or the resistance programming to be used for a crossbar application.

For large array sizes such as 64x64, we can calculate the possible sneak path current values based on the input voltage and resistance programming, using the formula in (19). For this example, the sneak path current would be ~32µA for this array size, if all the memristors are programmed to 1MΩ with a primary current of ~1µA for the IO switch-vector m_{closed} =1 and n_{closed} =1. To avoid a huge impact of the sneak path on the total output current, design decisions could involve increasing the number of IO-switch vectors to reduce parallel paths and modifying the resistance programming of the sneak path circuit. The sneak path current can be reduced to ~1.2µA by considering the extreme case of increasing the number of IO switch-vectors with m_{closed} = m-1 and n_{closed} =n-1 for the same 64x64 array example.

For modifying the memristor programming to impact the sneak path current, one memristor is programmed to high resistance when all other memristors are at low resistance and vice-versa as seen in Table 3. This methodology was explained in Section 3.6.6. For $X_1X_2X_3Y_1Y_2Y_3$ = 011011, by switching one of the memristors to 1MΩ when all the other memristors in the crossbar array are programmed to 10KΩ, the sneak path current is reduced from an all LRS current of 28.6µA to 0.99µA.

Table 3 Primary Current and Sneak Path Current Comparison

	All HRS (1 MΩ)		All LRS (10KΩ)		1 Low (10KΩ) for all HRS		1 High (1 MΩ) for all LRS	
I/O switch vector	$I_{primary}$ (µA)	I_{sneak} (µA)	$I_{primary}$ (µA)	I_{sneak} (µA)	$I_{primary}$ (µA)	I_{sneak} (µA)	$I_{primary}$ (µA)	I_{sneak} (µA)
001 001	1	0.8	100	80	1	1.14	100	50.6
011 011	2	0.25	200	25	2	0.49	200	0.49
011 001	2	0.57	200	28.6	2	0.65	200	0.99

3.6.8 Line Resistance impact on sneak path current

The line resistance is relatively small compared to the memristor resistance. For example, in my 3x3 array simulation with line resistances of 2.5Ω and memristor resistance of 10KΩ, the sneak path current decreases by ~0.005μA from ~80μA sneak path current value without line resistance. Specifically, for the 3x3 array, the sneak path current without line resistance is 79.999μA and with line resistance is 79.995μA. For the 5x5 array, the sneak path current is 177.77μA without line resistance and 169.45μA with line resistance. If the size of the array is increased from 3x3 to 256x256, an increase in the sneak path current is observed based on the calculations. Although the line resistance impact increases with the size of the array, it does not significantly alter the analysis when compared without line resistance. Therefore, line resistances are not considered for this research.

3.7 Summary of Chapter 3

Sneak path currents are a limiting factor to resistive crossbar array operations. It is essential when designing a system using a crossbar array to consider sneak path current when deciding how large the array can be. A calculation model for finding the total number of sneak paths for crossbar arrays has been described. The model for finding the longest possible sneak path for any given memristor crossbar array has been demonstrated. In addition to that, curve fitting models for calculating the sneak path currents as a function of array size, memristor resistances, memristor programming, I/O switch-vectors and input voltage are presented. Linear, exponential, and logarithmic relationships between the sneak path current and the crossbar array size for different I/O switch-vectors are observed while

characterizing sneak path current. These models will help with the sneak path and sneak path current prediction of any array size for understanding the constraints to the memristor crossbar operation. Simulation results and sneak path analysis highlight the importance of selecting an I/O switch-vector and resistance programming for analysing sneak paths. The boundary conditions for these parameters are the deciding factors for various memristor crossbar applications and for memristor testing purposes.

Chapter 4

Review of Testing Resistive Memristor Crossbar Arrays

The memristor array is defect prone due to immature manufacturing defects. It is important to test these memristor device for faults and device characterization. In this chapter, the defect mechanisms in memristors are examined and the memristor fault models are discussed. This is followed by discussion on the test methodologies explored for testing memristor crossbar arrays in literature. The conclusions from the discussed methodologies are presented. Finally, the research questions and need for an efficient test methodology is described.

4.1 Faults in memristor circuits

There have been several published fault models for memristor circuits as shown in Table 4. Different types of physical defects such as variation in length, area, and doping give rise to memristor faults. Table 4 summarizes the defects in a memristor caused by parametric variations and the associated fault model for fault detection.

1) Stuck-at-LRS faults

Considering a TiO_2 memristor, excessive doping of the TiO_2 with oxygen vacancies causes the memristor to be fully doped. As a result, it remains stuck at 1 irrespective of the voltage applied across it. A *SA1* (stuck-at-1) or stuck-at LRS faults can also occur when the addressed column is shorted to the input voltage. The fault is represented as (0/1) where logic 0 is the expected output of a fault-free memristor, while logic 1 is the output when a *SA1* fault is present.

Table 4 Memristor Faults

Fault	Cause of Defect	References
SA0 or SA open	Under-doped/open defect	[7][9][40][48][41]
SAL – stuck at logic level	Open defect	[7][48]
SA1 or SA short	Fully doped /short to VDD	[7][9][41]
SW1	Under doped/Open defect	[7][40][48][41]
SW0	Excessively doped/open defect	[7][40][48][41]
Deep 0	Increase in Length or Decrease in Area	[7][40][48][41]
Deep 1	Decrease in Length or Increase in Area	[7][41]
Deep 1/0	Under-doped/change in L or A	[7][40][48][41]
UR	Excessively doped	[7][41]
Coupling	Short between rows/columns	[40][41]
Undefined state faults	Undefined logical state due to defect	[49]
Read destructive faults	Open defects	[49]
Unknown read fault	Open defects	[50]
Transition Faults	Open defects	[50][49]

2) Stuck-at-HRS or Stuck-open Faults

A defective memristor deprived of oxygen vacancies will manifest in a faulty memristor that is always at HRS irrespective of the applied voltage. A *SA0* (stuck-at-0) or stuck-at HRS fault may occur in a memristor when there is an “open” circuit in the row, column or at the cross point. This fault is represented by (1/0); logic 1 is the expected output of a fault-free memristor, while logic 0 is the output in the presence of a *SA0* fault.

3) Slow-write-1 (SW1)

A memristor can be defective with a Slow-write-1 fault due to a small decrease in dopant density. The write pulse might not have enough flux to change the value in the memristor memory from a HRS to an LRS. A wider than normal or higher than normal amplitude write pulse is needed to switch logic states. Slow-write 1 fault represents a slow transition from HRS to LRS. The fault is represented by $\langle 0w1/X_0 \rangle$, where X_0 is an undefined output when the memristor resistance could be in the undefined state or can be at HRS. The fault is activated by writing a logic 1 when the memristor is at logic 0 (represented by 0w1).

4) Slow-write-0 (SW0)

Like SW1, a transition from LRS to HRS will be slow when there is a small increase in the oxygen vacancies. The fault is denoted by $\langle 1w0/X_1 \rangle$, where X_1 is an undefined output when the memristor resistance could be in the undefined state or can be LRS. The fault is activated by writing a 0 when the memristor is at logic 1 (represented by 1w0). A slow write fault also occurs when there is an unintended series resistance within a crosspoint.

5) Deep-0

Deep-0 state occurs due to an increase in the length (L) or a decrease in the cross sectional area *(A)* of the memristor. This causes the upper and lower resistance limits of the memristor to shift. The upper and lower bounds of memristor resistance change to R_{off} $+\Delta$ and R_{on} $+\Delta$. The memristor is in a 'deep 0' state when its memristance > R_{off}. For a deep-0 faulty device, the duration of the write pulse is not long enough to switch the memristor device from deep 0 to logic 1. A *Deep-0* can be sensitized using a sequence of write operations represented as $\langle \{0w0, w1\}/X_0 \rangle$.

6) Deep-1

Deep-1 state occurs when there is a decrease in the length (L) or an increase in the cross-sectional area (A) of the memristor. This causes the upper and lower bounds of the memristance to decrease to $R_{on}-\Delta$ and $R_{off}-\Delta$. For a deep-1 faulty device, the duration of the write pulse is not long enough to switch the memristor device from deep 1 to logic 0. A *Deep-1* fault can be sensitized by a sequence of write operations denoted by {1w1, w0}. The Deep-1 fault is represented by $\langle \{1w1, w0\}/X_1\rangle$.

7) Deep-1/0

This type of fault demonstrates the characteristics of both *Deep-1* and *Deep-0* faults. The fault can be sensitized by testing for both Deep-1 and Deep-0 fault types. The cause of the defect can be dopant deficiency combined with either a decrease in length or an increase in cross-sectional area.

8) Unknown read (UR)

Unknown read faults occur due to open defects within the memristor device. They can occur due to a combination of parametric defects, such as an increased length, combined with excessive doping. The memristor output exhibits a range of memristance represented by $\rho\,(R_{off}-R_{on}) < M(\alpha) < (1-\rho)\,(R_{off}-R_{on})$, resulting in an undefined output, irrespective of the voltage applied across it. The fault is represented as $\langle -/X\rangle$.

4.2 Currently Published testing methodologies for Fault Detection

Most of the published memory testing techniques are based on the march algorithms [36, 49, 50]. March tests are exhaustive tests with long test times since they test one memory cell at a time. Some papers have integrated sneak-path based testing into the march testing

for improving test time [7, 42, 52]. The focus of the papers in [7, 42] has been to reduce the read test time using sneak paths over traditional march tests. However, write operations take longer test time than reads. The authors in [52] have proposed a DFT circuit for reducing write test time; however, it adds additional hardware overhead. [53] discusses the fault dictionary approach based on March test algorithms for RAM testing. Some of these fault detection test methodologies have been described in the following section.

1) Sneak Path Testing in Memristors [7]

This paper discusses the defect mechanisms and fault models for memristors faults such as stuck-at-0, stuck-at-1, slow-write-0, slow-write-1, deep-0, and deep-1 faults. The author proposes an efficient testing scheme that uses sneak paths for testing these types of faults in 1T-1M RRAM crossbars. The advantage of sneak-path based testing is multiple memristors can be tested in a single measurement unlike the march testing that tests one memristor at a time. With this improved testing methodology, a test time improvement of ~32% is observed compared with the march test.

Testing Methodology:

The paper uses 1T-1M crossbar architecture where transistors are used to eliminate sneak paths by controlling the flow of current through crossbar during normal mode. During test mode, sneak paths are used for testing memristor faults. A group of memristors that can be tested simultaneously is referred to as Region of Detection (RoD) and when faulty, can cause measurable change in the output current of column being accessed. The difference between the defect-free crossbar current and faulty current greater than the detection limit detects the fault. RoD for each fault type is determined to minimize test time and to

maximize the test area. Fig. 24 shows the example of a RoD for stuck-at fault detection. The memory element at the center of the RoD is under test and the other green cells represent other detectable faults in the RoD. To sensitize a stuck-at 0 fault in the RoD, all the memory locations are written logic 1 value.

			$I_{idealON}$			
			$I_{idealON}$			
		$I_{idealON}$	$I_{idealON}$	$I_{idealON}$		
$I_{idealON}$	$I_{idealON}$	$I_{idealON}$	SA0 Fault	$I_{idealON}$	$I_{idealON}$	$I_{idealON}$
		$I_{idealON}$	$I_{idealON}$	$I_{idealON}$		
			$I_{idealON}$			
			$I_{idealON}$			

(a) SA0- fault using RoD method of fault detection where $I_{idealON}$ is the fault free current in green for logic 1. The cell highlighted in red is the SA0 fault in the RoD.

			$I_{idealOFF}$			
			$I_{idealOFF}$			
		$I_{idealOFF}$	$I_{idealOFF}$	$I_{idealOFF}$		
$I_{idealOFF}$	$I_{idealOFF}$	$I_{idealOFF}$	SA1 Fault	$I_{idealOFF}$	$I_{idealOFF}$	$I_{idealOFF}$
		$I_{idealOFF}$	$I_{idealOFF}$	$I_{idealOFF}$		
			$I_{idealOFF}$			
			$I_{idealOFF}$			

(b) SA1 fault using RoD method of fault detection where $I_{idealOFF}$ is the fault free current in green for logic 0. The cell highlighted in red is the SA1 fault in the RoD.

Fig. 24 RoD current variation for stuck-at Fault detection [5] Redrawn

If the output current is less than the defect-free current $I_{idealON}$, then the fault is detected in the RoD. Similarly, a stuck-at 1 fault can be sensitized by writing a zero to the memory cells in the RoD. If a *SA1* fault exists in the RoD, the output current is greater than the

defect-free current $I_{idealOFF}$. Multiple memristors are tested in a single RoD using this method and number of memristor accesses are also reduced, thus providing advantage over simple march tests.

2) Sneak-Path based Test and Diagnosis using Voltage Bias [9]

The paper proposes to use voltage bias to manipulate sneak paths for fault detection and fault diagnosis in a 4x4 region of memristors at a time. The authors choose 1R RRAM crossbars for the study because of its high density and performance unlike the 1T1R RRAMs which require selector devices. The voltage bias programming method is used to control sneak paths in the 1R RRAM structure. The proposed test mechanism is motivated by two observations - i)voltage bias can be applied to wordlines and bitlines to mitigate the impact of sneak path ii) sneak paths can be used to give resistance information of multiple memristors that can detect faults by comparing faulty current with output currents. By applying distinct levels of voltage bias on each wordline and bitline, undesired sneak paths can be eliminated, and the useful ones can be used for fault detection by multiple memristor testing.

Test methodology

A 3x3 crossbar array example is shown in Fig. 25 to illustrate the proposed test mechanism. Fig. 25(a) shows the intended current path through M3 in blue and the parallel sneak path current through M2, M5 and M6 in red. Fig. 25(b) shows the elimination of the sneak path by applying a voltage bias V_x to the wordline and bitline of the array. By changing patterns on the voltage bias, the sneak paths through different memristors can be controlled. Fig. 25(c) shows two sneak paths through memristors M2, M5, M6, M8 and M9 highlighted in

red which helps to test multiple memristors in a single read operation with M3 as the intended memristor.

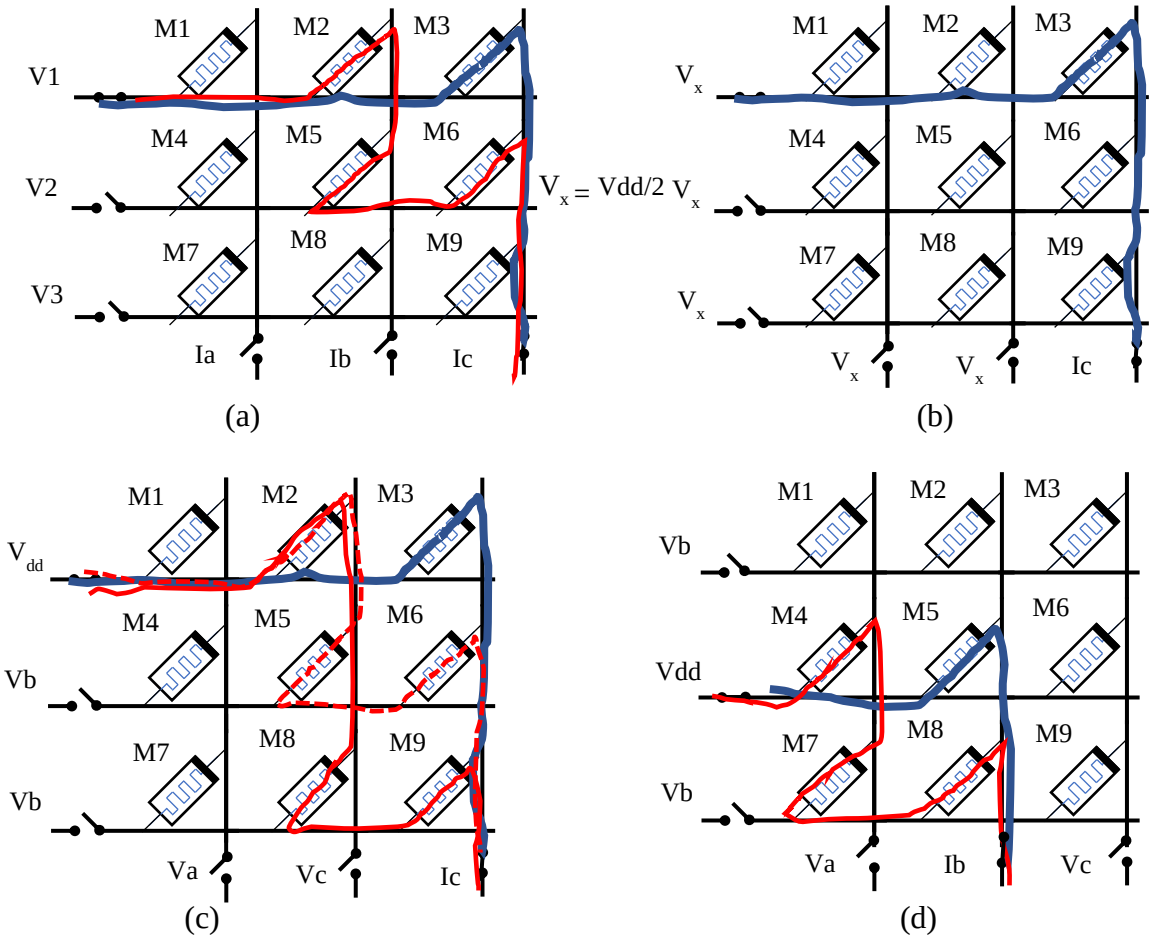


Fig. 25 Controlling sneak paths using voltage bias technique: (a) Example of sneak path through M2, M5 and M6 highlighted in red (b) sneak-path elimination with an uniform level of voltage bias V_x applied to wordline/bitlines; (c)Two sneak paths in red with intended memristor as M3 (d) Sneak path highlighted in red with intended memristor changed to M5 [9]. [Redrawn]

If the intended current path is changed to access memristor M5, other set of memristors can be tested using sneak path through M4, M7 and M8 as shown in Fig. 25(d).

3) *More efficient testing of metal-oxide memristor-based memories [52]*

The paper proposes March tests for testing metal-oxide memristor based memories using fast write operations. The authors focus on reducing the test application time and the test energy by proposing fast write operations. Fast march test algorithm is proposed for different fault models such as stuck-at faults, transition faults and shortened rows/column faults. The fast write method was modified to remove sneak paths from the test by grounding rows for reliable march testing. This paper uses a hybrid crossbar architecture consisting of combination of memristor and isolating transistor. The proposed Fast March Test (FMT) and the existing march test times have been compared for various fault detections. The proposed March test used a new fast write operation and reduced the test application time by 70% and the test energy by 40%.

Test methodology

To implement the fast write approach, DFT scheme is proposed to control the access times on the rows and columns during the write operation. The DFT circuit contains one timer to control the access time duration of the write operation called the W-Timer. W-Timer sets two different access times for write operation during normal mode and test mode. Fast write mode is selected during test mode.

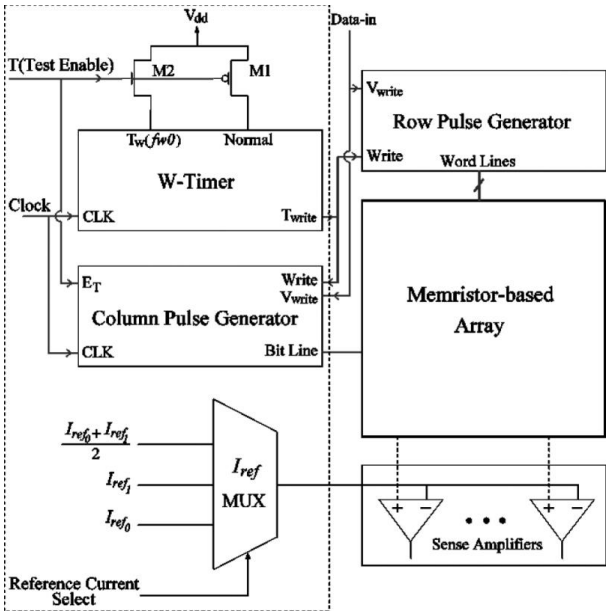


Fig. 26 Programmable DFT scheme [52]

The timer is shown on the left part of Fig. 26, and the associated selection hardware is shown above them.

4) DFT Schemes for Resistive Open defects in RRAMs [54]

Open defects may cause RRAM devices to enter an undefined state between logic 0 and logic 1 which can lead to test escapes and reliability issues. The regular march tests cannot guarantee high fault coverage for such type of defects. The paper motivates the need of special DFT to detect these unique faults in RRAMs. The paper proposes the use of two DFT schemes i) Short Write Time and ii) Low Write Voltage for fault detection. Simulation results show that defects causing the memristor device to enter undefined state can be detected with the DFT approach.

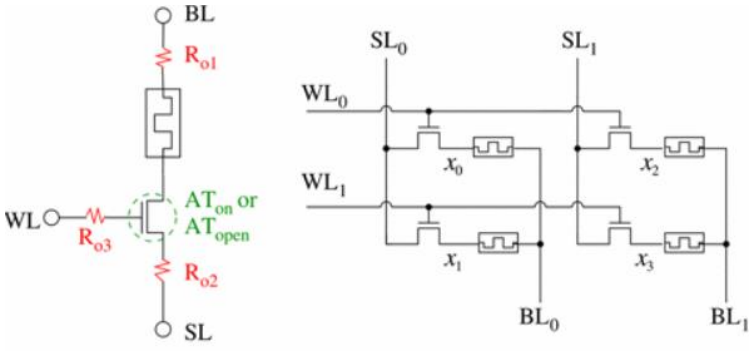
Test methodology

RRAM operations rely on the duration of access time and the supply voltage on the wordline and bitlines. The DFT schemes exploit these two properties for fault detection. The first DFT scheme referred to as Short Write Time (SWT) supplies the write voltage for a shorter period than the nominal write time. The second DFT scheme, referred to as Low Write Voltage (LWV), supplies a lower voltage than the nominal write voltage for the nominal time. The detection of these faults requires stressing the memristor device in such a way that: If the device has a defect and the output is in an undefined state, then the stressing has to shift the state of the device from the undefined state to a defined wrong state. The fault is then detected by performing a read operation after stressing the cell. If the device is fault-free, then it must remain in its correct defined state. Otherwise, the stress may cause overkill and yield loss. Simulations have been performed by injecting two open resistance faults using the write access time and the reduced supply voltage obtained values for the two DFT schemes. The simulation results show that the defects causing the RRAM cell to enter an undefined state are easily detected. However, both the DFT schemes could understress or overstress the RRAM under test leading to overkilling due to process variations.

5) Fault Modeling and Testing of 1T1R memories [50]

The paper proposes a testing methodology using march testing for 1T-1R 2x2 memristor memory structure as shown in Fig. 27. The paper proposes fault models based on electrical defects such as transistor suck-on, stuck-open faults and bridging faults. The paper also introduces to two new types of faults namely the write disturbance fault (WDF) and dynamic write disturbance fault (dWDF). The transistor stuck-on and bridge defects may

cause two-cell coupling faults where one cell is the aggressor and other is the victim. The resistance value of the bridge defect could disturb the write operation of the aggressor cell ultimately impacting the state of the victim cell. These two cells are said to have a WDF.



(a) (b)

Fig. 27 (a) Possible open, transistor stuck-on, transistor stuck-open defects in a 1T1R cell. (b) A 2×2 1T1R cell array [50].

The number of write operations on the aggressor cell a has an impact on the state of the victim cell. If the WDF is activated by more than two consecutive write operations in the aggressor, the new fault is called as dynamic disturbance fault (dWDF). A March test named as March-1T1R is proposed to cover the above defined faults in the 1T1R memristor array. The proposed March test requires (1+2a+2b)*N* write operations and *5N* read operations for an N-bit memristor memory, where a and b are the number of consecutive Write-1 and Write-0 operations for activating a dWDF.

6) On Defect Oriented Testing for Hybrid CMOS/Memristor Memory [55]

Hybrid CMOS/ memristor memory structures have the potential to replace the conventional non-volatile flash memory. Hybrid memories use the memristor as the storage element stacked on the top of the CMOS peripheral circuits creating three dimensional ICs. This paper discusses the defects in the hybrid memory system and a simulation model for defect injection and fault behavior is presented. The simulation results show that in addition to conventional semiconductor faults, there exist new unique faults due to open defects that require new test approaches (example, DFT techniques) to detect them. Fig. 28 shows the electrical circuit of a hybrid memory. The single memristor cell is divided into row group

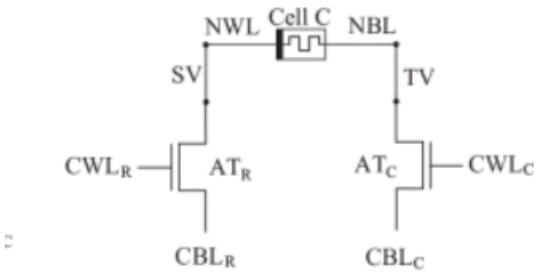


Fig. 28 Electrical equivalent circuit [55]

And column group, the row group consists of the CMOS wordline (CWL_R), CMOS bitline (CWB_R), access transistor (AT_R), short CNV (SV) and nano wordline (NWL). The column group consists of CMOS wordline (CWL_c), CMOS bitline (CBL_c), access transistor (AT_c). tall CNV (TV) and nanowire (NBL) as shown in Fig. 28. The Table 5 below gives the classification of defects in the three parts of hybrid memory which is the cell array, CMOS-to-nano vias and the peripheral circuits.

Table 5 Defect Classification in Hybrid memory [55]

Component	Classification	Location	Notation
Cell array	Open	Within cells	*OC*
		At nanowire bit lines (NBLs)	*OB*
		At nanowire word lines (NWLs)	*OW*
	Bridge	Between NBLs	*BB*
		Between NWLs	*BW*
		Between NBLs and NWLs	*BBW*
Via	Open	Within short CNVs	*OSV*
		Within tall CNVs	*0TV*
	Bridge	Between short CNVs	*BSV*
		Between tall CNVs	*BTV*
Peripheral circuits	Open	At gate, source and drain of transistor, interconnects	*OPC*
	Bridge	Among gate, source, drain and substrate of transistors, interconnects	*BPC*
	Short	At gate, source and drain of transistors, interconnects	*SPC*

Opens, bridges and shorts are the most commonly occurring faults in the memory cells and the CNVs. The defect injection and simulation are performed using the electrical SPICE memory model. The memory operation sequences for detecting faults are considered as:

- 0w1 – write 1 to a cell initialized to 0
- 1w0 – write 0 to a cell initialized to 1
- 1r1– read an expected value 1 from a cell
- 0r0 – read an expected value 0 from a cell

Using these sequences, the traditional memory fault models such as transition faults, stuck-at faults and incorrect read faults can be detected. However, special design for testability scheme is needed for unique faults such as:

- UWF_0 – cell set to an undefined state by write 0 operation
- UWF_1 – cell set to an undefined state by write 1 operation

7) *A Novel "Divide and Conquer" Testing Technique for Memristor based Lookup Table [56]*

The authors of this paper [56] propose an efficient approach for testing Memristor based Look up Table [MLUTs] formed by memristor array of LUTs. The main advantages of using this method are: 1) The ability to select any region of rows or columns in the memristor array using the memristor-based demultiplexer, 2) divide and conquer approach to effectively locate defective memristors in the MLUT can be applicable to other crossbar designs, 3)deterministic nature of the testing technique and 4) good scaling behavior. The TiO_2 based memristor device is used for the MLUT. The testing technique can be applied for fault detection of stuck-at 1, stuck-at 0 and -programmable defects [NPD]. A NPD1 defect is formed when the proportion of the doped region is slightly greater than the undoped region, similarly a NPD0 defect is formed when the undoped proportion is slightly greater than the undoped. In large-scale MLUT designs, it is very time consuming and tedious to test every crosspoint on each crossbar MULT exhaustively. A" Divide and Conquer" approach has been suggested by the authors to test multiple memristors in a single measurement using the demultiplexer. The defects can be detected by comparing the fault-free current and the crossbar output current in the first iteration. In the next iterations, the given region is split into halves and each of them are tested recursively for faults. An example of stuck-at fault detection using the divide and conquer approach is shown in Fig. 29 for an 8x4 MLUT.

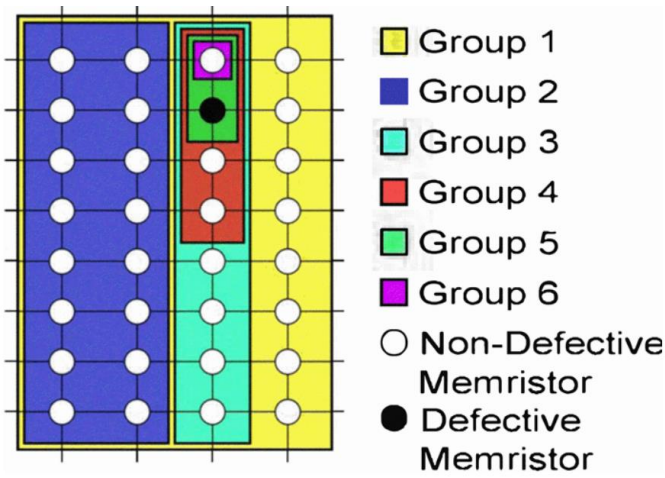


Fig. 29 Divide and Conquer approach [56]

I_{actual} or the crossbar output current is calculated by selecting all the rows and columns of the memristor. In the example in Fig. 29, based on the difference between fault-free current and I_{actual} current, a single defect can be identified. Following the iterative measurements, group 2 can be discarded from the search space since its defect free. From group 3, the defective memristor (black dot) can be detected in 4 measurements, discarding half of the search space in each iteration.

4.3 Fault Diagnosis

Faults can be either nonrecoverable or recoverable. For example, stuck-at faults and coupling faults are non-recoverable faults that can only be repaired by using redundant rows and columns. Slow-to-write faults, fast-write and deep faults can be recoverable faults. The faulty behavior can be recovered by controlling the duration of the write pulse or the voltage level of the write pulse to achieve the desired resistance to avoid the fault.

Some of the fault diagnosis methodologies referenced in literature are discussed in the below. To repair a fault, it is important to perform fault diagnosis to determine the fault location and the fault type.

4.3.1 Fault Diagnosis methodologies

Some of the test methodologies in literature for diagnosing faults in memristor circuits will be discussed in this section.

1) Sneak-Path based Test and Diagnosis using Voltage Bias [9]

The authors choose 1R RRAM crossbars for the study because of its high density and performance unlike the 1T1R RRAMs which require selector devices. The focus of the paper was fault diagnosis for single faults and multiple faults in a square RoD. By reconfiguring the voltage bias, sneak paths are controlled in the RoD. The output current of the RoD is compared with the fault free reference current to detect a fault.

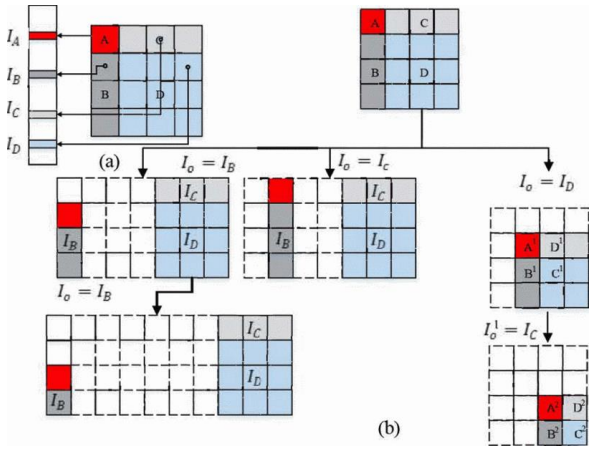


Fig. 30 Diagnosis process: (a) Example current in the RoD; (b) Diagnosis process for single fault in RoD [9]

In this method, a square RoD is partitioned into sub regions (A, B, C, D) with a set of reference currents associated with each subregion as shown in Fig. 30. The output current of each subregion is compared with the reference current to detect the faults in the RoD. The diagnosis algorithm for single fault is based on divide and conquer recursive process of the subregions. Multiple faults, though each one has small contribution to the output current, will cumulate their resistance variances to the output current through sneak-paths. The diagnosis process will have more measurements to diagnose multiple faults with the reference current being compared to the output current at every step.

2) Detection, Diagnosis and Repair of Faults in Memristor-based Memories [57]

This paper proposes an efficient testing technique for fault detection and fault diagnosis in memristor circuits using sneak paths. A hybrid diagnosis scheme that comprises of diagnosis by March test and sneak path testing is proposed to reduce test time. 1T-1M crossbar architecture is used to suppress the sneak paths in the normal mode using transistors and leverage sneak paths for testing in the test mode.

1. Diagnosis using March Sequence

A march test is defined by a sequence of operations applied to each memory cell before proceeding to the next cell. The order of proceeding to the next cell can be either in increasing address order (⇑) or decreasing address order (⇓). For an arbitrary addressing order, the symbol ⇳ is used. The memory operations are defined as ‘w0’ (write logic 0 to the memory cell), ‘r0’ (read logic 0 value from the memory cell.). Similarly, ‘w1’ (write logic 1) and ‘r1’ (read logic 1) are defined. The complete test is enclosed within curly brackets ‘{}’.

The March sequence to detect different type of faults such as stuck-at faults, slow-to-write, deep and coupling faults is described below.

{M1: ⇳(w0, w0, r0); M2: ⇑(r0, w1, r1); M3: ⇑(w1, r1); M4: ⇓(r1, w0, r0)}

SA1: sensitized and detected by M1.

SA0: sensitized by {w1} of M2 and M3. Detected by M3.

Deep-0/SW1: sensitized by {w0, w0} of M1 and {w1} of M2 and detected by M2.

Deep-1/SW0: sensitized by {w1} of M2, {w1} of M3, and {w0} of M4. Detected by M4.

Coupling: sensitized by {w0} of M1 and {w1} of M2 and detected by M2. Also sensitized by {w1} of M3 and {w0} of M4 and detected by M4.

Based on the detection sequences, the diagnostic sequence is shown in Table 6 to diagnose the fault type and fault location. Diagnostics is performed using various combinations of test sequence to determine type of fault.

Table 6 Test sequence and faults detected by each sequence for fault diagnosis [57]

Seq. No.	March Sequence	SA0	SA1	Deep-0	Deep-1	SW0	SW1	Coupling↑	Coupling↓
S1	⇳{w0,w0,r0}		√						
S2	⇳{w1,w1,r1}	√							
S3	⇳{w1,w0,r0}		√			√			
S4	⇳{w0,w1,r1}	√					√		
S5	⇳{w0, w0,w1,r1}	√		√			√		
S6	⇳{w1, w1,w0,r0}		√		√	√			
S7	⇳{w0}⇑{r0,w1}							√	
S8	⇳{w1}⇑{r1,w0}							√	
S9	⇳{w0}⇓{r0,w1}								√
S10	⇳{w1}⇓{r1,w0}								√
Diagnostic Sequence		**S2**	**S1**	**S5 and not S4**	**S6 and not S3**	**S3 and not S1**	**S3 and not S1**	**S7 or S8**	**S9 or S10**

2. Diagnosis using Sneak Paths

To minimize diagnostic time, sneak paths were used with the following sequence:

M1: ⇳(w0, w0); M2: ↑SA(r0); M3: ↑c(r0); M4: ⇳(w1); M5: ↑deep(r1); M6: ⇑(w1); M7: ↓SA(r1); M8: ↓c(r1); M9: ⇓(w0); M10: ↓deep(r0)}

SA1: sensitized by M1 and detected by M2. SA0: sensitized by M4, M6 and detected by M7.

Deep-0/SW1: sensitized by M1, M4 and detected by M5.

Deep-1/SW0: sensitized by M4, M6, M9; detected by M10.

Coupling: sensitized by M1, M3, M4 and detected by M5. Also sensitized by M6, M8, M9 and detected by M10. (21)

The equation (21) can diagnose the type of fault, but the only information about the fault location is that it is somewhere within the RoD. For diagnosing a single fault within the RoD, the iterative process of dividing the RoDs is utilized to pinpoint the location of the fault. In case of multiple defects in the RoD, the hybrid technique described below is used for fault diagnosis.

3. Diagnostics using Hybrid Technique

This technique combines the march and sneak-path diagnosis methods to reduce test time. The fault detection inside the RoD is performed using sneak paths. For single fault diagnosis in the RoD, the sneak path diagnosis technique is used. For multiple fault diagnosis in the RoD, march testing is performed on each memory cell to diagnose the fault

3)Diagnosis of Resistive Nonvolatile-8T SRAMs [53]

This paper proposes a two-phase diagnosis methodology for distinguishing between RAM faults and memristor faults of Resistive non-volatile-8T (Rnv8T) SRAMs. A Rnv8T SRAM cell consists of a 6T SRAM cell, two memristors (R_L and R_R) and two access memristors (M_2 and M_3) as shown in Fig. 31. This memory cell performs four functional operations such as read, write, store, and restore by activating wordlines and bitlines connected to the transistors and memristors in Fig. 31.

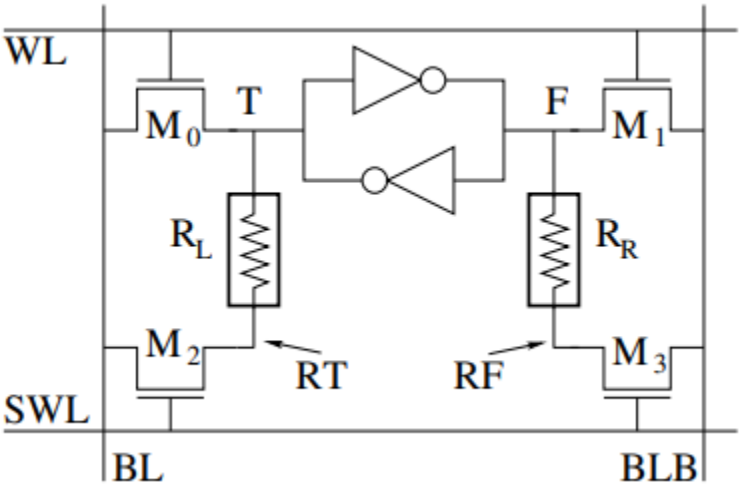


Fig. 31 Rnv8T SRAM cell structure [53]

In the first phase, the March 17-N algorithm is used to distinguish RAM faults such as stuck-at faults, state coupling faults and inversion coupling faults [58]. Once the RAM faults are identified, the proposed diagnosis algorithm March-MD is used for diagnosing memristor-related faults in the second phase of testing. Memristor-related faults include memristor stuck-at faults (MASF), slow store fault (SSF), store destructive fault (SDF) and memristor disturb read fault (MDRF). The MASF causes the SRAM cell to stuck-at a logic value of 0 or 1 after the restore operation. SSF refers to memristor not programmed to the

expected resistance value within the store operation time. Lastly, in presence of a MDRF fault, the SRAM cell returns unexpected or undefined state during read operation. The fault dictionary of March -MD is captured below for the memristor-related faults where $E_i = 0$ denotes the ith read operation that cannot detect the corresponding fault, If E_i = even or odd, then the corresponding fault can be detected at the even or odd addresses respectively.

Table 7 Fault dictionary of March-MD [53]

Fault Types	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
MSA0F	0	0	0	odd	0	0	0	odd
MSA1F	0	0	even	0	0	0	even	0
SS0F	0	0	even	0	0	0	0	0
SS1F	0	0	0	odd	0	0	0	0
SD0F	0	odd	0	odd	0	0	0	odd
SD1F	even	0	even	0	0	0	even	0
MDR0F	even	0	even	0	even	0	even	0
MDR0F	0	odd	0	odd	0	odd	0	odd
Fault Types	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}
MSA0F	0	0	even	0	0	0	even	0
MSA1F	0	0	0	odd	0	0	0	odd
SS0F	0	0	0	odd	0	0	0	0
SS1F	0	0	even	0	0	0	0	0
SD0F	even	0	even	0	0	0	even	0
SD1F	0	odd	0	odd	0	0	0	odd
MDR0F	0	odd	0	odd	0	odd	0	odd
MDR1F	even	0	even	0	even	0	even	0

4.4 Drawbacks of existing testing methodologies

In the paper [7] “Sneak Path Testing in Memristors” The test methodology cannot apply to 1R RRAM crossbar because the access transistors are important to control the sneak paths for fault diagnosis. It only considers single faults occurring in a ROD and fails to test and diagnose multiple faults. It incurs significant routing cost to switch each access transistor independently. In the paper “DFT Schemes for Resistive Open Defects in RRAMs” DFT schemes might overstress or understress the RRAM under test leading to overkilling. Redesign is an expensive solution to this problem. In “Fault Modeling and Testing of 1T1R

memories", the transistor selector devices degrade the performance and reduce the density of the RRAM crossbar. The memristor and the selectors are fabricated in heterogeneous technologies, resulting in high integration cost. In "Sneak-Path based test and diagnosis using voltage bias [9]", the focus of the paper is on fault diagnosis using the ROD concept. It is based on the divide and conquer recursive approach which is iterative and depends on the size of the array. In "More efficient testing of metal-oxide memristor-based memory" [52] Fast write march test has been proposed to reduce test application time during write operation. However, it has a DFT overhead for hardware changes.

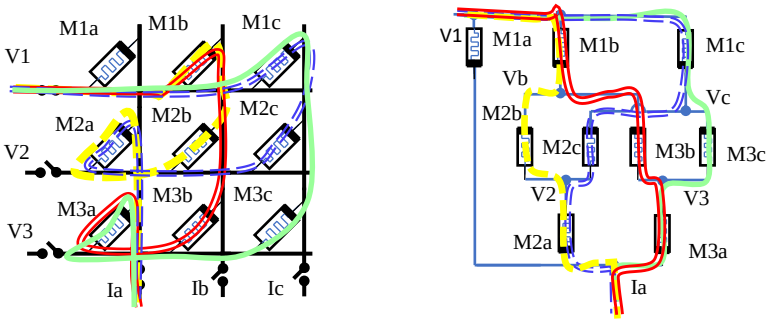
4.5 Research Goals for testing memristor circuits

For all the discussed methodologies, there has been limited focus on the fault coverage during fault detection. Since most of the methods are dependent on March testing that have 100 % fault coverage, there has been limited focus on improving fault coverage with shorter tests. All the methods are based on March exhaustive testing where the results have been analyzed for comparing test times and number of test operations. This leads to research goal of measuring fault coverage in memristor circuits.

4.5.1 Research Goal 1: Fault Coverage

My research goal is to develop a fault coverage for fault detection using sneak paths. Fault coverage calculation is required to evaluate different tests and to come up with a good quality test. It is possible to get better fault coverage using long length sneak paths for shorter tests. To develop an efficient test there is always a tradeoff between fault coverage and test time.

Example of Long Length Sneak Path:



(a) Three memristor long paths in crossbar array

(b) Circuit equivalent showing three long memristor paths

Fig. 32 Sneak paths of length 5 in a 3x3 crossbar array with M1c=M2b=M3a = HRS and all of the rest of the memristors in LRS and for IO switch-vector = 100100

Through our sneak path characterization work, the longest number of possible sneak paths in an array can be calculated. For example, in the 3x3 array below with IO switch-vector = 100100, the longest possible length of sneak path = 5. In this combination, it is possible to have four of three memristor long paths as shown below in Fig 32. The primary path is through selected cell *M1a*. The half-selected cells in this circuit are the ones sharing the line with *M1a* namely *M1b, M1c, M2a and M3a*. The sneak paths through electrical network is three memristor long as shown above namely *M1b-M2b-M2a, M1b-M3b-M3a, M1c-M2c-M3a, and M1c-M3c-M3a*

As noted previously, if all the memristors have the same resistance value then all the sneak paths are of length three. However, when the memristors are at different resistance values, some patterns can create longer sneak paths. The four possible five memristor long memristor sneak paths are: *M1c-M2c-M2b-M3b-M3a, M1c-M3c-M3b-*

M2b-M2a, *M1b-M2b-M2c-M3c-M3a*, and *M1b-M3b-M3c-M2c-M2a*. One way to get the five memristor long path *M1c-M2c-M2b-M3b-M3a* is to have *M1b* and *M2a* in the HRS and the rest in the LRS. Another programming to get the same long sneak path is *M1b*, *M2a*, and *M3b* in the HRS with the rest in LRS. To get the second example of the five memristor long sneak path, *M1c-M3c-M3b-M2b-M2a*, the memristors *M1b* and *M2c* are programmed to the HRS and the rest are LRS. To get the fourth example of path *M1b-M3b-M3c-M2c-M2a*, the memristors *M1c*, *M2b*, and *M3a* are in the HRS. This case is shown in Fig. 33.

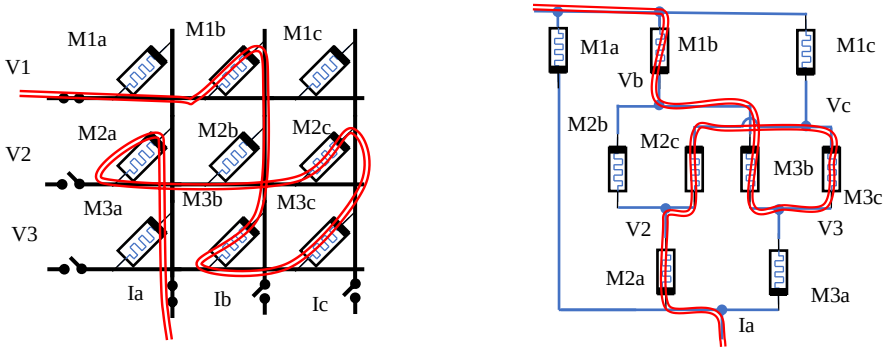


Fig. 33 Sneak paths of length 5 in a 3x3 crossbar array with M1c=M2b=M3a = HRS and all of the rest of the memristors in the LRS and for I/O switch-vector = 100100

There are many other patterns to get these and the other five memristor long sneak paths. Even with the five memristor long sneak paths, there are still total of four sneak paths. Specifically, three of the paths are three memristor long (*M1b-M2b-M2a*, *M1b-M3b-M3a*, *M1c-M2c-M2a*) and one of the paths is five memristor long (as shown in Fig. 33). As shown in Table 1 and (4), for this 3x3 array the longest possible sneak path is 5. The length of sneak paths is a function of the array size, the I/O switch-vector, and the programming of the individual memristors. It is possible to get more than three memristor long paths using different resistance programming seen from the example. This will lead to enhanced fault

detection with multiple faults. Sneak paths longer than 3 paths will lead to better fault coverage. My test methodology proposes to use these long length sneak paths for testing memristor faults since using long length sneak paths can detect more faults per test vector and therefore, it results in a shorter test vector set.

4.5.2 Research Goal 2: Fault Detection

My research will mainly target Stuck-at-Low Resistance State faults (SLRS) and Stuck-at-High Resistance State Faults (SHRS). A 3x3 crossbar array example described below shows fault detection using march testing.

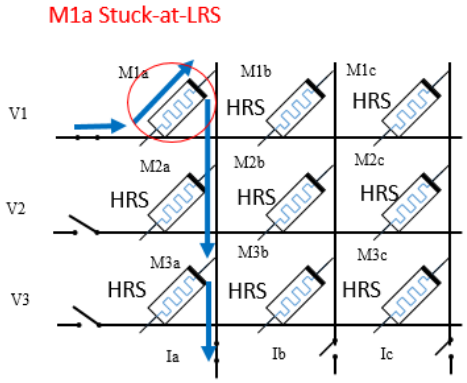


Fig. 34 Stuck-at LRS fault example for single step of march testing in a 3X3 crossbar array

In this example, *M1a* in the 3x3 memristor crossbar circuit is a stuck-at LRS fault with LRS = 10KΩ and HRS = 1MΩ .The condition to detect a Fault at *M1a* is $I_{output\ A} > I_{stuck\text{-}at\ LRS}$. From Table 3, $I_{output\ A[Reference]}$ = 1µA + 0.8µA which is the sum of the primary current and the sneak path current when all the memristors in the array are of 1MΩ each. However, since *M1a* is stuck-at LRS, the primary current will now be 100µA considering 10KΩ as the LRS. Since the $I_{Stuck\text{-}at\ LRS}$ is greater than the $I_{output\ A[Reference]}$ the fault will be

detected. This type of testing tests a single memristor at a time in the crossbar array. It marches through the array from one memristor to another, focusing on single memristor faults and is also referred to as March testing.

4.5.3 Fault Detection Using Sneak Paths

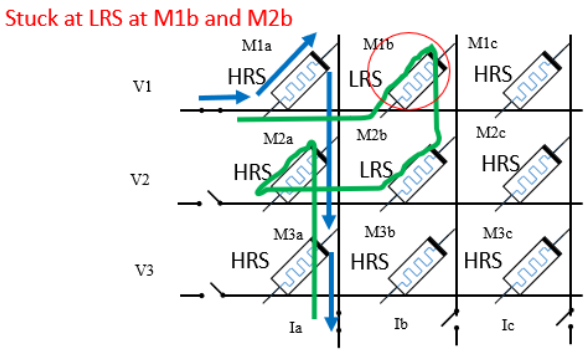


Fig. 35 Fault Detection Using Sneak Paths in a 3x3 crossbar array

For the 3x3 crossbar array example in Fig. 35, *M1b* is a stuck-at LRS fault in a All-HRS programmed array. The resistance values considered for this example are LRS = 10KΩ and HRS = 1MΩ. The condition to detect a fault in the circuit is $I_{output\ A} > I_{Stuck\text{-}at\ LRS}$. From Table 3, $I_{output\ A[Reference]}$ = 1μA + 0.8μA which is the sum of the primary current and the sneak path current for a fault-free circuit. However, since now *M1b* and *M2b* is stuck-at-LRS, the sneak path current will be 1.14μA as shown in Table 3 when one of the memristors is at LRS out of all the remaining HRS. Since now the $I_{Stuck\text{-}at\ LRS}$ is greater than the $I_{output\ A[Refernce]}$ the fault will be detected. This type of testing can test multiple memristors at a time since it utilizes sneak path current going through multiple memristors for fault detection.

4.5.4 Research Goal 3: Fault Diagnosis using Sneak Paths

In Section 4.3.1, two ROD methods were suggested by [7] and [9] for fault diagnosis in memristor circuits. It is an iterative process that involves many measurements to pinpoint the fault location. ROD is further divided with the output current being compared to the reference current. ROD changes with each fault type and it is possible to detect more than one fault type in a single ROD. My research goal was to study the scope of methods other than ROD for fault diagnosis. I found the fault dictionary could be a useful and promising fault diagnosis technique that can be used in memristor circuits. The fault detection technique using sneak paths is expanded to consider fault diagnosis for stuck-at LRS and stuck-at HRS faults.

4.5.5 Research Goal 4: Test Pattern Generation

My research goal is to generate a good quality test vector set or test patterns by characterizing sneak paths in memristor circuits as a function of input voltage, IO switch-vector, resistance programming and size of array.

4.6 Summary of Chapter 4

Memristor technology specific faults are discussed in this chapter. The behaviour of the fault and defect models are described in the literature review of memristor testing. Some of the fault detection and fault diagnosis methodologies for march and sneak-path based memristor testing are described followed by the observations of their limitations and the need for a better good quality test. Finally, my research goals using sneak path testing are discussed for testing memristor circuits.

Chapter 5

Sneak Path based testing in Memristor Circuits

Note: Some contents of this chapter have been approved for publication:
Rasika Joshi, John M Acken "Utilizing Sneak paths for Memristor Test time Improvement", in *IETE Journal of Research*, 2020.

My memristor test methodology contribution optimizes the IO-switch vectors and the memristor HRS/LRS programming for fault detection and fault diagnosis using sneak paths. The research focuses on the stuck-at low resistance and stuck-at high resistance faults, with a later extension to intermediate faults for fault detection and fault diagnosis analysis. Using the sneak path characterization work in Chapter 3, the research goal was to develop a method to evaluate a test for fault coverage of fault detection using sneak paths. This test methodology targets test time reduction using shorter test vector sets and tests multiple memristors at a time. Long length sneak paths are used for reducing write test time since write operations take longer test time than read. The following sections describe the fault detection and fault dictionary based diagnosis approach using a 3x3 crossbar array example. Fault coverage calculation is explained with the crossbar array example with long length sneak paths. The simulation results of this test methodology show an improved test time for fault detection and fault diagnosis with shorter test vector sets.

5.1 Stuck-at LRS and Stuck-at HRS faults

One cause of stuck-at LRS defect is excessive doping material due to which the memristor remains stuck at LRS irrespective of the voltage applied across it. If a memristor has a

stuck-at LRS fault, then it remains stuck at logic 1 when a negative voltage is applied to the memristor to turn it off. LRS is the faulty output in the presence of stuck-at LRS fault and HRS is the expected or the fault-free output. The cause of Stuck-at HRS defect is lack of doping material due to which the memristor remains stuck at HRS. If a memristor has a stuck-at HRS fault, then it remains stuck at logic 0 when a positive voltage is applied to the memristor to turn it on. HRS is the faulty output in the presence of stuck-at HRS fault and LRS is the expected or the fault-free output.

5.2 Fault Detection Approach

The fault detection methodology targets single stuck-at-LRS and single stuck-at-HRS faults in memristor crossbar arrays. The condition to detect a fault is based on the comparison between the device current (I_{CUT}) and the reference current ($I_{Reference}$). The sneak path current through the memristors falling on the selected wordline and bitline contribute to the total output current or I_{CUT}. $I_{Reference}$ is the output current of a defect-free crossbar. The difference between $I_{Reference}$ and I_{CUT} detects the fault if it is greater than the detection limit. For a stuck-at LRS fault, if the $I_{CUT} - I_{Reference}$ > Detection limit, the stuck-at LRS fault is detected for a given IO switch-vector. Similarly, for a stuck-at HRS fault, if the $I_{Reference} - I_{CUT}$ > Detection limit, the stuck-at HRS fault is detected for a given IO switch-vector. The flow chart in Fig. 36 and Fig 37 describes the methodology for fault detection for stuck-at HRS faults and stuck-at LRS faults, respectively. The advantage of using this method is the contribution of the sneak path current to the I_{CUT} for testing multiple memristors at a time over the test-time consuming march test. All the memristors in the selected wordline and bitline for a given IO switch-vector will be tested for fault

detection at the same time. This methodology is demonstrated using a 3x3 crossbar array example below.

5.2.1 Fault Detection Example Using Sneak Paths

This example is shown in Fig. 36. The condition to detect a stuck-at LRS fault in the circuit is $I_{CUT} - I_{Reference} >$ Detection limit. A fault will be detected if the bitline and wordline containing the fault are switched ON using the IO switch- vector of the memristor array.

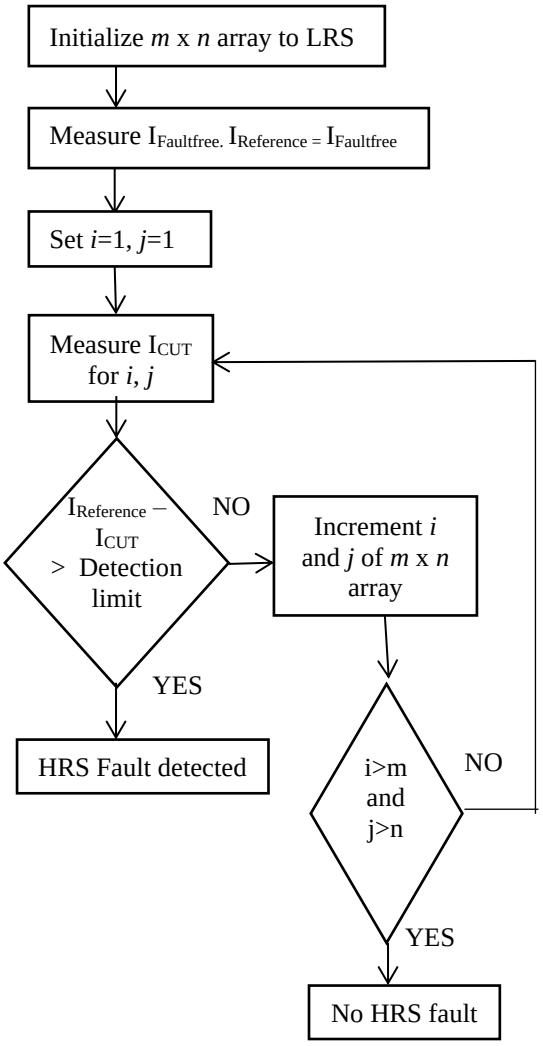


Fig. 36 Fault Detection for HRS Fault

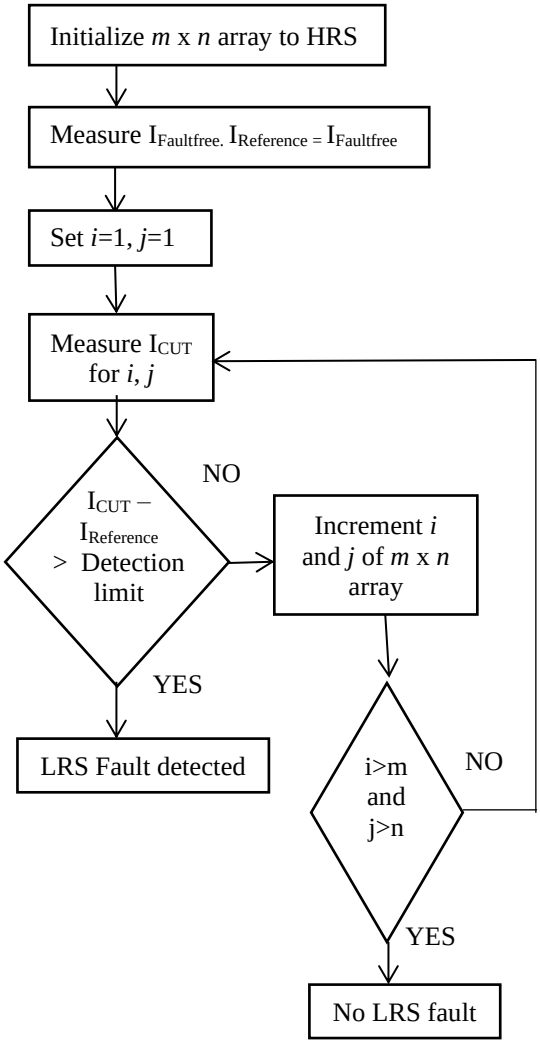
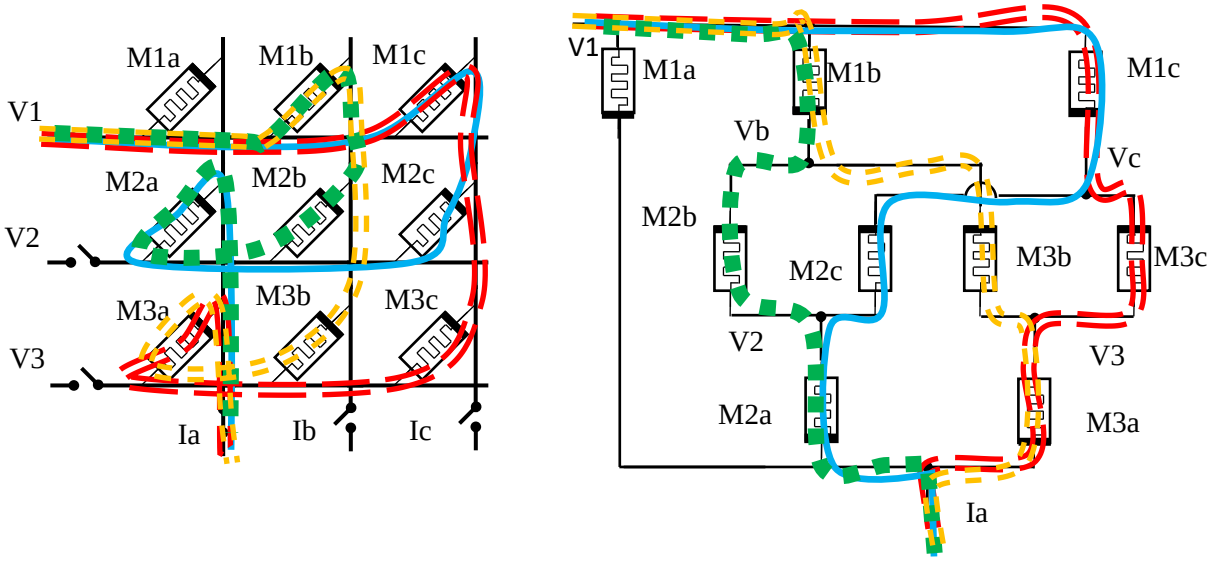


Fig. 37 Fault detection for LRS fault

For example, if IO switch-vector = 100100, all the memristors located on the wordline (*M1a, M1b, M1c*) and all the memristors on the bitline (*M1a, M2a, M3a*) would be tested and any single fault in these memristors would affect the output current and ultimately the $I_{CUT} - I_{Reference}$. The sneak path current through *M1b, M1c, M2a* and *M3a* would contribute to the output current. Three memristor long sneak paths through these memristors namely *M1b-M2b-M2a, M1b-M3b-3a, M1c-M2c-M2a, M1c-M3c-M3a* affect the output current

values. Single stuck-at LRS fault in *M1b, M1c, M2a* or *M3a* for the HRS programmed memristor array would be detected using this IO switch-vector. A similar example can be applied for a stuck-at HRS fault. In this case, the crossbar array is programmed to LRS and the sneak path current for the IO switch-vector under test would help to detect a fault based on the location of the fault and when $I_{Reference} - I_{CUT} >$ Detection limit.



(a) Three memristor long paths in crossbar array (b) Circuit equivalent showing three memristor long paths

	M1b-M2b-M2a
	M1b-M3b-M3a
	M1c-M2c-M2a
	M1c-M3c-M3a

Fig. 38 Sneak paths of length 3 in a 3x3 crossbar array with I/O switch vector = 100100

5.3 Fault Diagnosis Methodology Using Sneak Paths

A fault dictionary technique for diagnosing stuck-at LRS and stuck-at HRS faults in crossbar circuits is presented. Using our python-based sneak path calculator tool, we can find the three memristor long sneak paths for different array sizes. This sneak path

information is used in forming the fault dictionary. The fault dictionary is created based on the fault detection methodology discussed in section 5.2. The memristors that can be detected for a fault by applying the given IO switch-vector are marked with "Y" and the memristors that cannot be detected for a fault for that IO-switch-vector are marked with a "N". The fault diagnosed memristors are represented as bolded "***Y***" or "***N***". The methodology is based on the pass/fail analysis of each IO switch-vector applied to the circuit. Results show that a best-case scenario of fault diagnosis can be achieved in three IO switch-vectors for any crossbar array size if the pass/fail analysis works as below:

1) Apply the first IO switch-vector for i=1 and j=1 in the IO switch-vector for mxn array where "*i"* is the iterator for the "m" number of wordlines and "j" is the iterator for "n" number of bitlines.
2) If the first IO switch-vector fails, select the next vector that has "m" number of intersections with the first IO switch-vector. This step will eliminate m–1 or m of the possible $2m$–1 error locations.
3) Select the next vector with one number of intersections of Ys with the remaining of the first vector. This step will eliminate one remaining error locations.
4) Repeat the process until the fault location is diagnosed.

This methodology is presented in the flow chart in Fig. 39. Fault diagnosis can be achieved in a minimum of 3 vectors or a maximum of m+1 IO switch-vectors in this process.

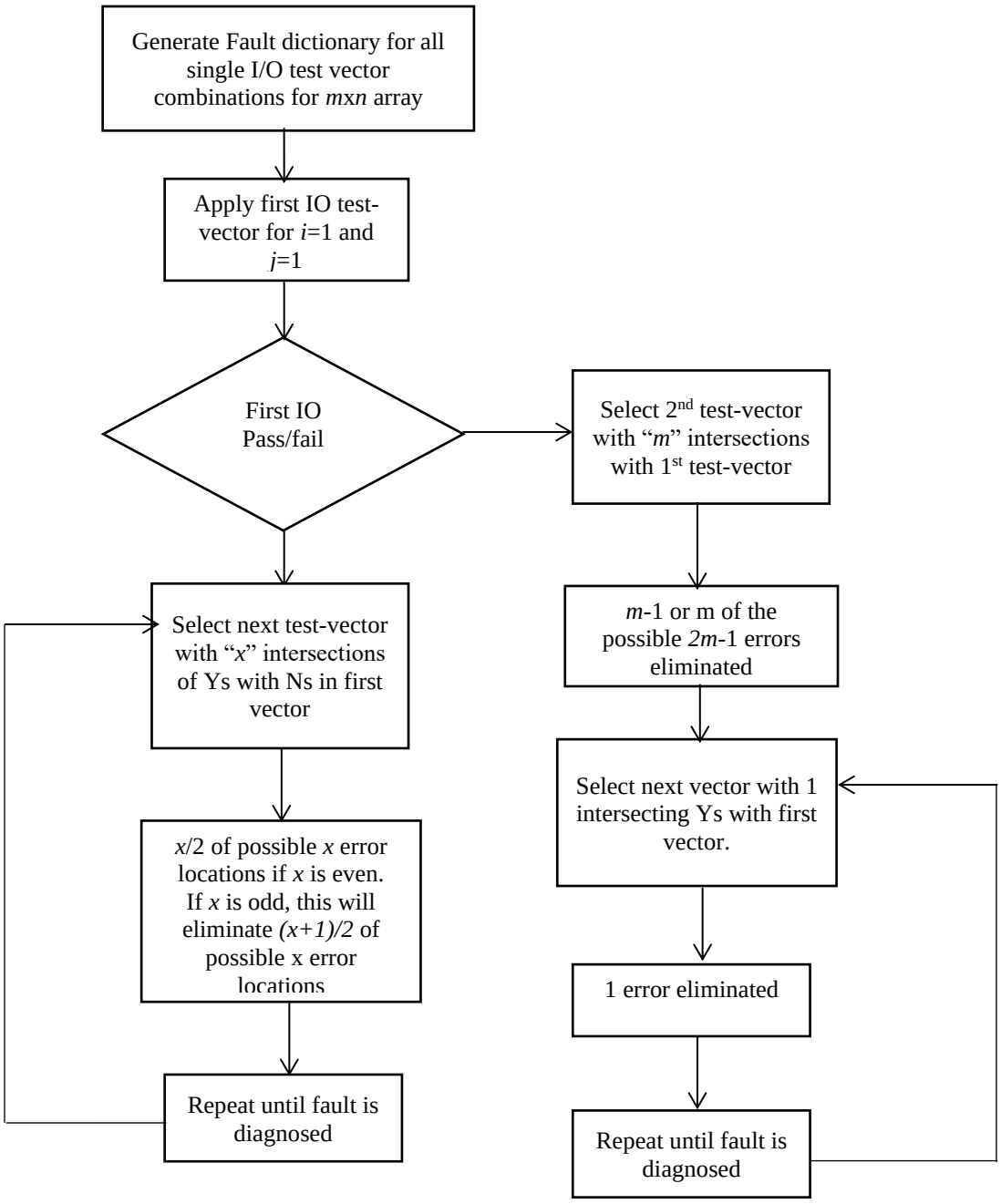


Fig. 39 Fault Diagnosis Methodology for LRS/HRS faults

This methodology is demonstrated using a 3x3 crossbar example in Table 3.

(1) When the first vector 100001 is applied, five faulty memristors are detected represented by Ys and four memristors are not detected for any fault represented by Ns. The result is a “fail” denoted by “F” for this IO switch-vector. The five memristors in the potential fault list are *M1a, M1b, M1c, M2c,* and *M3c*.

(2) The second vector 010001 was picked because it had three overlapping Ys with the first vector, and three was chosen because it is the width of the array. This IO switch-vector result is a “pass” denoted by “P” and the memristors *M1c, M2c* and *M3c* are removed from the potential fault list.

(3) The next vector 010010 is chosen with one overlapping Y with the remaining Ys in the first vector. This IO switch-vector fails with only one overlapping Y with the first vector on *M1b*. Hence, the fault is diagnosed at *M1b* as shown in Table 8.

The three test vectors and the pass/fail results for each of them are shown in Table 8.

Table 8 Diagnosis example when first test vector fails for 3x3 memristor array

		M1a	M1b	M1c	M2a	M2b	M2c	M3a	M3b	M3c	
F	100 001	Y	Y	Y	N	N	Y	N	N	Y	1st vector
	100 010	Y	***Y***	Y	N	Y	N	N	Y	N	
	100 100	Y	Y	Y	Y	N	N	Y	N	N	
P	010 001	N	N	Y	Y	Y	Y	N	N	Y	
F	010 010	N	***Y***	N	Y	Y	Y	N	Y	N	2nd vector
	010 100	Y	N	N	Y	Y	Y	Y	N	N	3rd vector
	001 001	N	N	Y	N	N	Y	Y	Y	Y	
	001 010	N	Y	N	N	Y	N	Y	Y	Y	
	001 100	Y	N	N	Y	N	N	Y	Y	Y	

Note: On the first vector 100001, the three light square Ys match with three Y's with the second vector 010001. These three light squares correspond to *M1c, M2c,* and *M3c*. Because the second vector 010001 passes, these three faults are eliminated as candidates. When the third vector 010010 is applied, it fails, the overlap is at *M1b,* and it is diagnosed. The fault diagnosed memristor *M1b* is represented as bold "***Y***" in Table 8.

The diagnosis algorithm will change if the first IO switch-vector when applied passes. The steps of the algorithm are captured below:

(1) Apply the first IO switch-vector for i=1 and j=1 in the IO switch-vector for mxn array where i is the iterator for the "m" number of wordlines and j is the iterator for "n" number of bitlines.

(2) Select an IO switch-vector which has x (number of possible memristors which might have errors) number of intersections of Ys with Ns in the first vector this will eliminate x/2 of possible x error locations if x is even. If x is odd, this will eliminate (x+1)/2 of possible x error locations.

(3) Repeat the process until the fault is diagnosed.

This methodology is presented in the flowchart in Fig. 39. The methodology is demonstrated using the example as captured for a 3x3 array.

(1) The first vector 100001 passes in the fault dictionary in Table 9. This result means memristors M1a, M1b, M1c, M2c and M3c do not have a fault. Now, the remaining memristors *M2a, M2b, M3a* and *M3b* are in the fault list as per the fault dictionary.

(2) On applying the second vector 100100, it passes for the two out of the four overlapping N and Ys with the first vector. This step removes *M2a* and *M3a* from the fault list since they are passing as per the fault dictionary.

(3) The third vector 010001 passes for the *M2b* overlap with the first vector. Hence, *M2b* is removed from the list and the fault is diagnosed at *M3b*

Table 9 Diagnosis example when first vector passes for 3x3 memristor array

		M1a	M1b	M1c	M2a	M2b	M2c	M3a	M3b	M3c	
P	100 001	Y	Y	Y	N	N	Y	N	N	Y	1st Vector
	100 010	Y	Y	Y	N	Y	N	N	Y	N	
P	100 100	Y	Y	Y	Y	N	N	Y	N	N	2nd vector
P	010 001	N	N	Y	Y	Y	Y	N	***N***	Y	
	010 010	N	Y	N	Y	Y	Y	N	Y	N	3rd vector
	010 100	Y	N	N	Y	Y	Y	Y	N	N	
	001 001	N	N	Y	N	N	Y	Y	Y	Y	
	001 010	N	Y	N	N	Y	N	Y	Y	Y	
	001 100	Y	N	N	Y	N	N	Y	Y	Y	

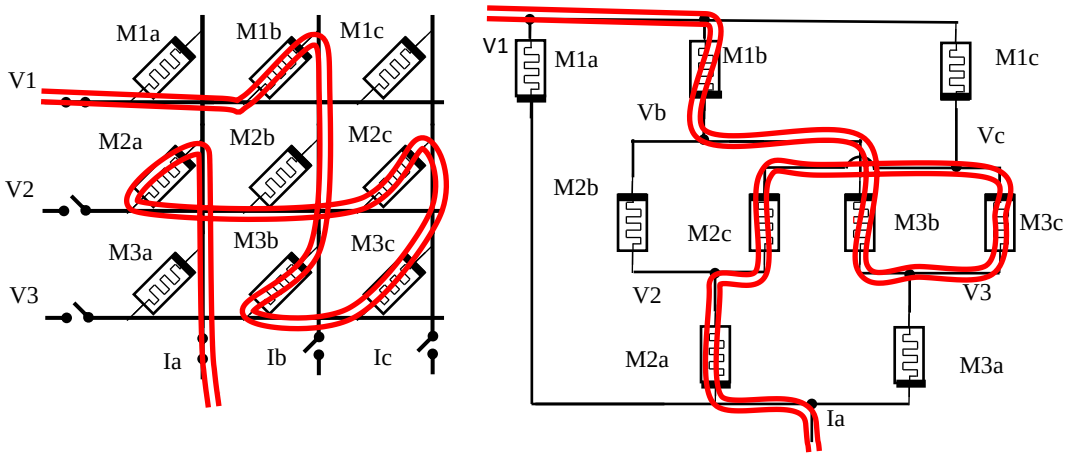
Note: In Table 8, the first applied test vector 100001 passes, leaving *M2a*, *M2b*, *M3a*, and *M3b* in the fault list. The second IO switch-vector 100100 passes for two Ys overlapping with the two Ns in the first vector. These two faults *M2a* and *M3a*, are eliminated as candidates. The third vector 010001 passes for Y overlap with N on *M2b* with fault remaining on *M3b*. The fault is diagnosed on *M3b*. The fault diagnosed memristor *M3b* is represented as bold "***N***" in Table 9.

5.4 Fault Coverage using sneak path testing

Fault coverage is a ratio of the total number of faults detected to the total faults possible in the memristor circuit, for a given test vector set. The number of test vectors (IO switch-vectors) in a test vector set can be reduced by utilizing long sneak paths. The sneak paths

longer than three memristors are referred to as long sneak paths. The following example for fault coverage uses five memristor long sneak paths. Using these long length sneak paths, any faulty memristor along the long sneak path is detected.

An IO switch-vector with one selected wordline and one selected bitline is considered as the first test vector. For example, consider the 3x3 memristor array with IO switch-vector =100100 in Fig. 37. Long length sneak paths are formed in crossbar arrays when memristors are at different resistance values. Four different five memristor long sneak paths are possible in this 3x3 crossbar circuit, namely *M1c-M2c-M2b-M3b-M3a, M1c-M3c-M3b-M2b-M2a, M1b-M2b-M2c-M3c-M3a,* and *M1b-M3b-M3c-M2c-M2a*. For the fault coverage analysis, two out of these four sneak paths are considered, which are *M1b-M3b-M3c-M2c-M2a* and *M1c-M2c-M2b-M3b-M3a*. One way to get the *M1b-M3b-M3c-M2c-M2a* path is to have *M1c*, *M2b* and *M3a* programmed to HRS and the rest to be in LRS as shown in Fig. 40.



(a) Five memristor long sneak path (b) Circuit equivalent showing five memristor long sneak path

Fig. 40 Sneak path M1b-M3b-M3c-M2c-M2a of length five in a 3x3 crossbar array with M1c=M2b=M3a=HRS and remaining memristors in LRS for I/O switch vector =100100.

Here, the input voltage is 1V with LRS =10KΩ and HRS= 1MΩ. The fault-free sneak path current in this scenario with the five memristor long sneak path is 21.1µA. Only single faults are considered during fault detection analysis. If either *M1b* or *M2a* has a stuck-at HRS fault, the sneak path current reduces from the original sneak path value of 21.1µA to 1.91µA. Similarly, if there is a fault on either *M2c* or *M3b*, the sneak path current reduces from the fault-free sneak path current value to 2.83µA. If there is a stuck-at HRS fault on M3c, the sneak path current reduces to 3.75µA. The difference between the fault-free current and the faulty current helps to detect the stuck-at HRS fault. For the second five memristor long sneak path, the *M1c-M2c-M2b-M3b-M3a*, the memristors *M1b, M2a* and *M3c* are programmed to HRS. Again, the fault-free sneak path current for the five memristor long sneak path is 21.1µA. If either *M1c* or *M3a* have a stuck-at HRS fault, the sneak path current reduces from the original fault free current to 1.91µA. Similarly, if there

is a fault on either *M2c* or *M3b*, the sneak path current reduces to 2.83µA. If there is a fault on *M2b*, the sneak path current reduces to 3.75µA. Thus, with these two five memristor long sneak paths, a 100% stuck-at HRS fault coverage is achieved for the 3x3 array as shown in Table 10.

Table 10 Five memristor long sneak paths in 3x3 memristor array

Sneak Path	Fault Detection on memristors	HRS programming
M1b-M3b-M3c-M2c-M2a	*M1b, M2a, M2c, M3b, M3c*	*M1c, M2b, M3a*
M1c-M2c-M2b-M3b-M3a	*M1c, M2b, M2c, M3a, M3b*	*M1b, M2a, M3c*

To get the two five memristor long sneak paths, resistance programming needs to have a specific pattern among memristors. For the above example, it is observed that with one test vector (IO switch-vector) and different resistance programming, a complete fault coverage set can be achieved with shortened test time.

For a single stuck-at LRS fault, three memristor long sneak paths are used to detect the fault. For the same example of 3x3 memristor array with IO switch-vector =100100 in Fig. 35, there are four possible sneak paths, namely *M1b-M2b-M2a*, *M1b-M3b-M3a*, *M1c-M2c-M2a*, and *M1c-M3c-M3a*. The total simulated sneak path current value is 0.8µA when the array is programmed in HRS =1MΩ. If either *M1b, M1c, M2a,* and *M3a* have a stuck-at LRS fault, the original sneak path current of 0.8µA increases to 1.14µA. The difference between the fault free sneak path current and the faulty current helps detect the LRS fault. Similarly, if memristors *M2b, M2c, M3b* and *M3c* have a stuck-at LRS fault, the original

sneak path current of 0.8µA increases to 0.87µA. However, these faults may not be detected if the detection limit is set to 0.2µA. The difference between faulty and fault free current in this case is less than 0.2µA. Another set of IO switch-vector can be used in this case such as 010100 and 001100 to detect LRS faults in *M2b*, *M2c* and *M3b, M3c* respectively for complete fault coverage.

Using long length sneak paths, we can test more faults per test vector leading to shorter test vector sets. Considering the 3x3 crossbar example that we looked at in Fig. 40, using the five long sneak path, we can test for 5 memristors at a time compared to March test that tests only one memristor element at a time. 5X improvement in test time can be achieved in this case. As size of the array increases, the length of the longest possible sneak path in the array will also increase based on our formula for L_{max} (longest possible sneak path) in (18). For example, for a 100x100 array, the test time improvement can be ~199X times better than march testing since L_{max} can be 199 memristors long.

5.5 Summary of Chapter 5

Sneak paths in memristor crossbar arrays can be utilized to reduce test time. A test methodology that uses sneak path current for both detection and diagnosis of single stuck-at LRS and stuck-at HRS faults in memristors has been described. The test methodology contribution optimizes the IO-switch vectors and the memristor HRS/LRS programming for testing memristors. The diagnosis methodology included a fault dictionary. These methodologies were demonstrated by applying them to a 3x3 memristor crossbar array. Results show that fault diagnosis can be achieved in three test vectors for the best case and in m+1 test vectors for $m > n$ for worst case in an mxn crossbar array. Finally, the fault

coverage calculation is also discussed using five memristor long sneak path and three memristor long sneak path crossbar examples for stuck-at HRS faults and stuck-at LRS faults respectively.

Chapter 6

Detection Limit for Intermediate Faults

Note: Some of the contents of this chapter have been accepted for publication below:
Rasika Joshi, John M Acken "Detection limit for Intermediate faults in Memristor circuits", *International Symposium on Quality Electronic Design* (ISQED' 21) April 7-8, 2021, California, USA.

This chapter introduces a new testing approach for the type of faults in memristor circuits called intermediate faults. My previously discussed test methodology in Chapter 5 using sneak paths can be extended to detect intermediate faults in crossbar circuits. The importance of setting the detection limit for intermediate fault detection is discussed using crossbar array examples. Simulation results present the detection limit for intermediate faults using five memristor long and three memristor long sneak paths in a crossbar array. A testing solution is described with a method to set the detection limits for intermediate fault detection in memristor crossbars.

6.1 Intermediate faults

There have been several published fault models for memristor circuits as shown in Table 1. Different types of physical defects such as variation in length, area, and doping give rise to memristor faults. The fault detection method discussed in Chapter 5 was used for detecting stuck-at LRS faults and stuck-at HRS faults in memristor circuits. Stuck-at LRS faults are caused due to excessive doping. Hence, it will be stuck-at logic 1 irrespective of the voltage applied to it. The faulty output state is at LRS and the fault-free output is at

HRS for a stuck-at LRS fault. Similarly, lack of doping could cause a memristor to be in a stuck-at HRS state. In this case, LRS is the expected output of a fault-free memristor, while HRS is the output in the presence of a stuck-at HRS fault. Table 11 lists the different memristor fault types and has a column named “memristor state” that describes the resistance value of the memristor due to the defect. The question marks in that column represent intermediate faults in memristor circuits.

Table 11 Memristor Faults

Fault	Cause of Defect	References	Memristor state
SA0 or SA open	Under-doped/open defect	[7][9][40][48][41]	HRS
SAL – stuck at logic level	Open defect	[7][48]	HRS
SA1 or SA short	Fully doped /short to VDD	[7][9][41]	LRS
SW0	Under doped/Open defect	[7][40][48][41]	?
SW1	Excessively doped/open defect	[7][40][48][41]	?
Deep 0	Increase in Length or Decrease in Area	[7][40][48][41]	?
Deep 1	Decrease in Length or Increase in Area	[7][41]	?
Deep 1/0	Under-doped/change in L or A	[7][40][48][41]	?
UR	Excessively doped	[7][41]	?
Undefined state faults	Undefined logical state due to defect	[49]	?
Unknown read fault	Open defects	[50]	?

Intermediate faults are those types of faults where the memristor resistance state lies between LRS and HRS. SW (slow-to-write) and Deep faults are discussed in [7,40,41,48]. An intermediate fault could be SW1/SW0 (Slow-to write 1) fault where the memristor state might be either in undefined state or could be logic 0/logic 1 respectively as discussed in [7,40,41,48]. It could also be a deep fault where the memristor state could have elevated M_{off} and M_{on} resistance values for a Deep-1 fault and lower M_{on} and M_{off} resistance values for a Deep-0 fault. It could be an undefined state fault where the logical state of the device is unknown and can lie between logic 0 and logic 1. It becomes important to have a proper detection limit to detect such faults since the resistance state of these faults could be either at logic 0 or logic 1.

6.2 Fault Detection Method for Intermediate Faults

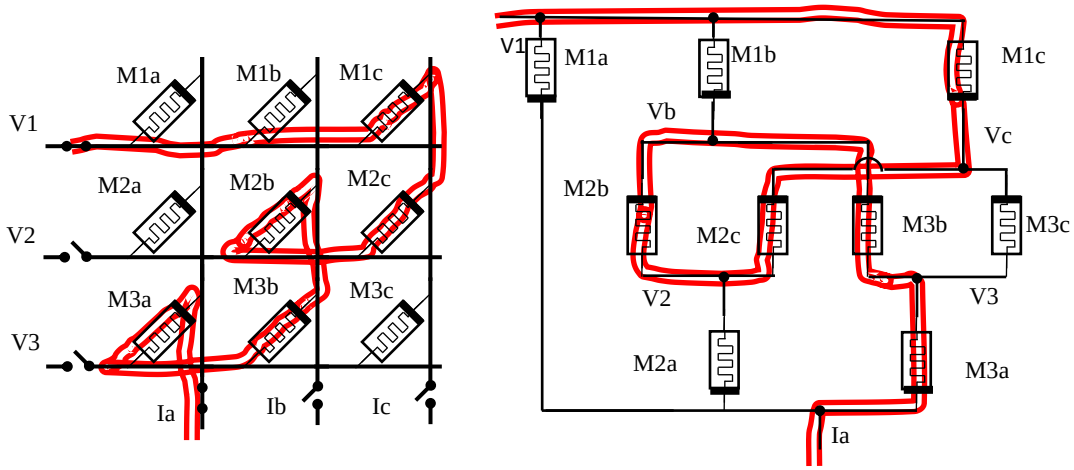
My previously discussed fault detection methodology targets single stuck-at-LRS and single stuck-at-HRS faults in memristor crossbar arrays. The condition to detect a fault is based on the comparison between the device current (I_{CUT}) (circuit under test) and the reference current ($I_{Reference}$). For a stuck-at LRS fault, if the $I_{CUT} - I_{Reference}$ > Detection limit, the stuck-at LRS fault is detected for a given IO switch-vector. Similarly, for a stuck-at HRS fault, if the $I_{Reference} - I_{CUT}$ > Detection limit, the stuck-at HRS fault is detected for a given IO switch-vector. The previous chapter (5) showed how to set the detection limit for detecting stuck-at LRS and stuck-at HRS faults using a 3x3 crossbar array as an example. The fault detection method in Fig. 36 and Fig. 37 can be extended to intermediate fault detection as well. However, the detection limit needs to be properly defined to detect such faults based on the memristor state. In the next section, the simulation results and the

proposed detection limits are presented to detect intermediate faults in a 3x3 crossbar array example.

6.2.1 Fault detection example for Intermediate faults

The sneak paths longer than three memristors are referred to as long sneak paths. The following example for fault detection uses five memristor long sneak paths Using these long length sneak paths, any faulty memristor along the long sneak path is detected. A 3x3 memristor array with IO switch-vector =100 100 shown in Fig. 41 is considered as an example for the intermediate fault detection. It's possible to create a long length sneak path when the memristors are at different resistance value. In this example, four of five memristor long sneak paths are possible namely: *M1c-M2c-M2b-M3b-M3a, M1c-M3c-M3b-M2b-M2a, M1b-M2b-M2c-M3c-M3a, and M1b-M3b-M3c-M2c-M2a*. The five memristor long sneak path *M1b-M3b-M3c-M2c-M2a* can be achieved by keeping memristors M1c, M2b and M3a programmed to HRS while other memristors at LRS. Similarly, five memristor long sneak path *M1c-M2c-M2b-M3b-M3a* is possible when memristors *M1b, M2a* and *M3c* are programmed to HRS and rest are in LRS. This path *M1c-M2c-M2b-M3b-M3a* is shown in Fig. 41. For this circuit, input voltage = 1V, LRS =10KΩ and HRS = 1MΩ are the voltage and resistance value assignments. The fault free sneak path current for the five memristor long sneak path is 21.1µA for this circuit. This research analysis only considers single faults for fault detection. For the path *M1c-M2c-M2b-M3b-M3a*, if there is a stuck-at HRS fault on either *M1c* or *M3a*, the new sneak path current would be 1.91µA. Similarly, if there is a stuck-at HRS fault on either *M2c* or *M3b*, the faulty sneak path current would be 2.833µA. The sneak path reduces to 3.75µA if there

is a fault on *M2b*. The sneak path current is analyzed in the presence of different intermediate faults. Table 12 captures the different output sneak path currents in presence of intermediate resistance faults between 10KΩ and 1MΩ which are the LRS and HRS values of the memristor respectively.



(a) Five memristor long sneak path

(b) Circuit equivalent showing five memristor long sneak path

Fig. 41 Sneak path M1c-M2c-M2b-M3b-M3a of length five in a 3x3 crossbar array with M1b=M2a=M3c=HRS and remaining memristors in LRS for I/O switch vector =100100.

The sneak path currents are simulated for $R_{intermediate}$ = 500KΩ, 200KΩ, 100KΩ, 50KΩ and 20KΩ resistance values in Table 12.

Table 12 Sneak Path current analysis for Intermediate faults in a 3x3 crossbar array

$R_{intermediate}$	$I_{Faultycurrent}$ M1c	$I_{Faultycurrent}$ M2b	$I_{Faultycurrent}$ M2c	$I_{Faultycurrent}$ M3a	$I_{Faultycurrent}$ M3b
20kΩ	17.6µA	17.98µA	17.79µA	17.6µA	17.79µA
50kΩ	11.9µA	12.8µA	12.5µA	11.9µA	12.5µA
100kΩ	7.96µA	9.29µA	8.63µA	7.96µA	8.63µA
200kΩ	5.03µA	6.62µA	5.82µA	5.03µA	5.82µA
500kΩ	2.77µA	4.55µA	3.66µA	2.77µA	3.66µA

$I_{Reference}$ = 21.1µA and I_{CUT} = $I_{Faultycurrent}$ for each of the memristors along the five memristor long sneak path as shown in Table 11. Noise margins are technology dependent. Any fabrication process is going to have different amounts of variations or noise margin. For this example, a noise margin at ±10% is considered. The difference between $I_{Reference}$ and I_{CUT} needs to be greater than the detection limit to detect the HRS fault. Fig. 39 shows the difference between the fault free current with noise variation added and the 50KΩ faulty sneak path current with noise variation added is ~8µA. We choose half of this value which is ~4µA as the detection limit to help detect memristor faults with resistance values closer to HRS. The Detection limit is represented in Fig. 42 by a black box. From Table 11, the detection limit of 4µA can help to detect all the $R_{intermediate}$ faults > 20KΩ. However, the 20KΩ intermediate fault cannot be detected since the difference between $I_{Reference}$ and the I_{CUT} is ~3µA. With R_{LRS} =10KΩ, the 20KΩ intermediate resistance is closer to the LRS value and can be detected as a stuck-at LRS fault. Such faults cannot be detected using this method and another approach is needed to detect stuck-at LRS faults. The method to detect stuck-at LRS faults at intermediate resistance values is described below.

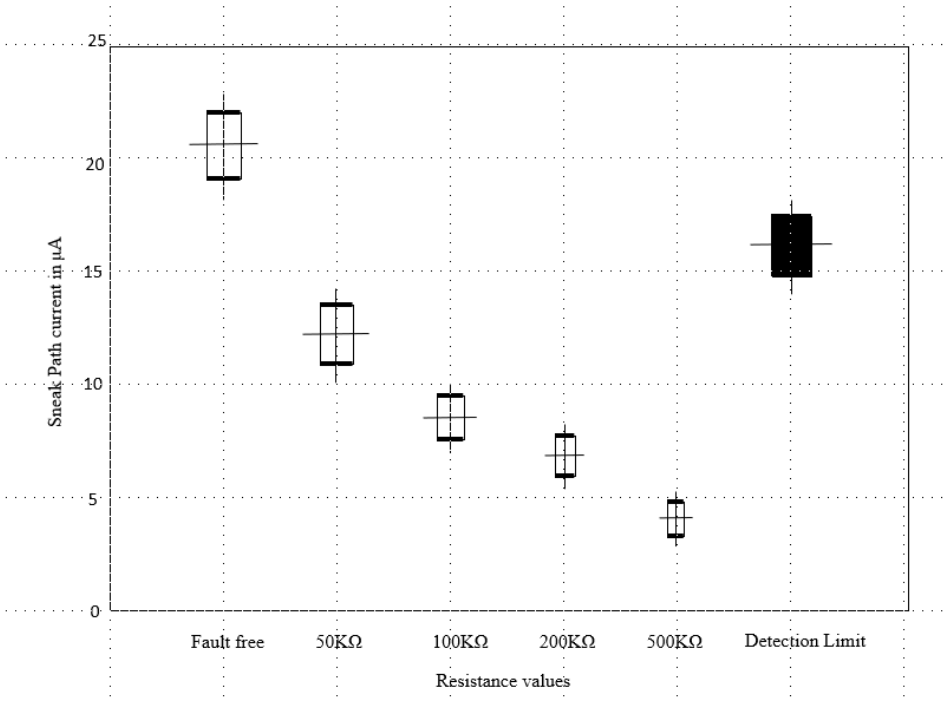


Fig. 42 Sneak Path current for fault free and intermediate faults in a 3x3 crossbar array

For a single stuck-at LRS fault, three memristor long sneak paths are used to detect the fault. For the same example of 3x3 memristor array with IO switch-vector =100100 in Fig. 43, there are four possible sneak paths namely *M1b-M2b-M2a, M1b-M3b-M3a, M1c-M2c-M2a,* and *M1c-M3c-M3a*. The total simulated fault-free sneak path current value is 0.8µA when the array is programmed in HRS =1MΩ. If either *M1b, M1c, M2a,* and *M3a* have a stuck-at LRS fault, the sneak path current of 0.8µA increases to 1.14µA. The difference between the fault free sneak path current and the faulty current helps to detect the LRS fault. For this example, when $R_{intermediate}$ = 20KΩ, the faulty sneak path increases to 1.13µA. The detection limit is chosen as 0.16µA since it is half of the worst-case difference between the fault free sneak current and the faulty sneak path current. As mentioned before, if the $I_{CUT} - I_{Reference}$ > Detection limit, the stuck-at LRS fault is detected for a given IO switch-

vector. For $R_{intermediate}$ = 20KΩ, the difference between the reference current and the faulty sneak path current is ~0.32µA and the stuck-at LRS fault can be detected. Similarly, if the remaining memristors M2b, M2c, M3b and M3c have a stuck-at LRS fault, the original sneak path current of 0.8µA increases to 0.87µA. These faults may not be detected since the difference between the faulty and fault free sneak path currents is less than 0.16µA. Different IO switch-vectors need to be used such as 010100 and 001100 to detect LRS faults in M2b, M2c and M3b, M3c respectively for complete fault coverage.

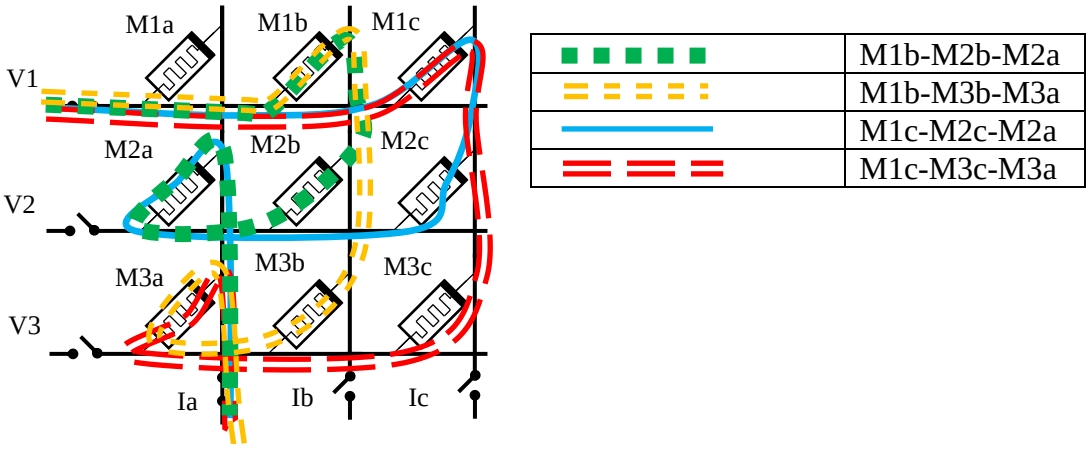


Fig. 43 Three memristor long sneak paths in 3x3 crossbar array with IO switch vector =100100 with all memristors in HRS.

6.2.2 Current resolution for Fault detection measurement

Resistance programming (R_{LRS}/R_{HRS}) and the tolerance variation plays an important role in the current resolution for fault detection measurement, especially for bigger crossbar arrays (greater than 10x10). For the same 3x3 crossbar array example with the five long sneak path as discussed in section 6.2.1, if LRS = 1KΩ (instead of 10KΩ) and HRS = 1MΩ, the fault free current would be ~200µA. The stuck-at HRS fault of 1MΩ on the five memristor long sneak path would not be detected in this case since the faulty output current

would be ~5µA which would not satisfy the detection limit criteria, considering the tolerance margin of ±10%. The R_{LRS}/R_{HRS} ratio helps to set the detection limit for fault detection. For the reference current or the fault-free current measurement, LRS/HRS resistance programming need to be set in a way such that the difference between the faulty current and fault-free current falls outside the tolerance margin. For large arrays, such as 14x14, the long length sneak path of 27 memristors would have a fault-free sneak path current of ~10µA if the LRS memristors are set to 10KΩ and HRS memristors are set to 1MΩ. A stuck-at HRS fault along the long length sneak path would not contribute to the variation in the sneak path current in this case. The difference in fault-free current and faulty current for this case is extremely small and cannot be detected. However, if we set LRS = 100Ω and HRS = 1MΩ, the stuck-at HRS fault along the long sneak path can be detected with the fault-free current at ~13µA and the faulty current at ~6µA respectively. This shows importance of resistance programming in achieving desired current resolution for fault detection for bigger crossbar arrays.

As discussed in Chapter 3 section 3.6.7, choosing the right IO switch-vector also has an impact on the sneak path current. As the crossbar array size increases, a combination of following three approaches can be utilized to achieve fault detection in various array sizes:

1) R_{LRS}/R_{HRS} programming
2) IO switch-vector combinations
3) Optimizing sneak path lengths

6.3 Summary of Chapter 6

Just as memristors have variations in resistance values, so do faulty memristors have variations in values. The term for the faulty values between stuck-at HRS and stuck-at LRS values is intermediate faults. A testing solution has been described to detect intermediate faults in memristor circuits based on a published fault detection method using sneak paths. A different method needs to be used for intermediate memristors closer to HRS and another approach is used for detecting intermediate faults closer to LRS. A method to set detection limits for intermediate fault detection is demonstrated using 3x3 crossbar array simulations. The fault detection scheme can be used for detecting intermediate faults along with stuck-at low resistance and stuck-at high resistance faults.

Chapter 7

Summary, Conclusions, Achievements and Future Work

7.1 Summary and Conclusion

The focus of my dissertation was to develop a test methodology to test memristor crossbar circuits independent of application. The methodology was driven by my characterization of sneak paths length and sneak path currents in the crossbar circuits. Sneak paths are characterized as a function of the size of array, resistance values, input voltage and IO switch-vector. Formulas have been derived to calculate the number of sneak paths in various array sizes. The conditions which determine the length of the sneak paths are described. The equations I derived for different input/output conditions help predict sneak paths and sneak path currents for various array sizes. This work characterizing sneak paths provide boundary conditions for designing crossbar arrays for various applications and provides insights into memristor crossbar testing.

An efficient testing methodology is required since memristor devices are prone to defects due to the immature manufacturing process and fabrication techniques. Using the sneak path characterization, I have developed a method to evaluate a test for fault coverage using a shorter test vector set. The advantage of using sneak path based testing is that multiple memristors can be tested in a single measurement unlike the March testing that tests one memristor in a single measurement. My research focuses on fault detection and fault diagnosis test methodology for stuck-at low resistance and stuck-at high resistance faults in memristor crossbar circuits. The objective of the fault detection method is to improve test time by using long length sneak paths with shorter test vector set. For larger

crossbar arrays such as the 100x100 array, the length of the longest possible sneak path can be 199 memristors long, leading to 199X test time improvement compared to March test. This testing approach is extended to intermediate fault testing. The procedure for setting the detection limit for intermediate faults has been analyzed using crossbar array simulations. The importance of setting the detection limit for intermediate fault detection is discussed using crossbar array examples. Simulation results were used to establish the detection limit for intermediate faults using five memristor long and three memristor long sneak paths in a crossbar array. A testing solution is described with a method to set the detection limits for intermediate fault detection in memristor crossbars.

For all the test methodologies referenced in literature, there has been limited focus on the fault coverage during fault detection since most of the testing methods are based on March testing that has 100% fault coverage. My first research goal achieved was a test method for fault detection in memristor crossbar circuits for stuck-at low resistance and stuck-at high resistance faults by using shorter test vector sets and LRS/HRS resistance programming. Secondly, a fault diagnosis technique using sneak paths was developed. Fault dictionary testing proved to be a productive technique with improved test time for finding the location of the faulty memristor by looking at the intersection of applied test vectors on the crossbar array. The results were demonstrated using LTSpice simulations on crossbar array examples. I have achieved the end goal of developing a test pattern for fault detection and fault diagnosis by optimizing the resistance programming, IO switch-vectors, input voltage and the size of the array for stuck-at fault and intermediate fault detection.

Crossbar arrays are used for various applications such as memory operations, neuromorphic, security and stochastic. The major research contribution is to have an application independent testing methodology for testing resistive memristor crossbars. This would mean that the testing approach works for RRAM application as well as security, neuromorphic, logic and stochastic applications since the test methodology utilizes sneak paths in these resistive circuits. By optimizing the memristor crossbar array parameters, the sneak paths and sneak path currents are efficiently used for memristor crossbar array testing.

7.2 Achievements and Publications

1) Characterized sneak path length and sneak path currents in memristor crossbar arrays for design decisions.

Journal Publication published: Joshi, R., & Acken, J. M. (2020). Sneak Path Characterization in Memristor Crossbar Circuits. *International Journal of Electronics*, 1–18. https://doi.org/10.1080/00207217.2020.1843716

2) Developed fault testing and fault diagnosis methodology utilizing sneak paths in memristor crossbar arrays with improved test time

Journal Publication approved for publication: Rasika Joshi, John M Acken "Utilizing Sneak paths for Memristor Test time Improvement", in *IETE Journal of Research*, 2020 doi 10.1080/03772063.2021.1883483

3) Detection limit for intermediate fault detection by extending stuck-at LRS/stuck-at HRS fault detection methodology in memristor crossbar arrays.

Conference publication accepted: Rasika Joshi, John M Acken “Detection limit for Intermediate faults in Memristor circuits”, *International Symposium on Quality Electronic Design* (ISQED’ 21) April 7-8, 2021, California, USA.

7.3 Future Work

The future work could further improve the test generation technique for fault detection and fault diagnosis for test time by optimizing the test vector set applied to memristor crossbar arrays. The sneak path characterization work and the fault testing methodology for fault detection and diagnosis can be used to develop EDA (Electronic Design Automation) tools for designing and testing memristor circuits.

References

[1] L. Chua, "Memristor-The missing circuit element," in *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507-519, September 1971.

[2] Strukov, D. B., Snider, G. S., Stewart, D. R., & Williams, R. S. (2008). The missing memristor found. *Nature*, *453*(7191), 80.

[3] Williams, R. (2008). How We Found The Missing Memristor. *IEEE Spectrum*, *45*(12), 28–35. https://doi.org/10.1109/mspec.2008.4687366

[4] Z. Li et al., "An overview on memristor crossbar based neuromorphic circuit and architecture," 2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Daejeon, 2015, pp. 52-56.

[5] K. Mbarek, F. O. Rziga, S. Ghedira and K. Besbes, "Characterization, and modeling of memristor devices," *2017 International Conference on Engineering & MIS (ICEMIS)*, Monastir, 2017, pp. 1-5.

[6] Y. Ho, G. M. Huang, and P. Li, "Nonvolatile memristor memory: device characteristics and design implications," International Conference on Computer-Aided Design, pp. 485 - 490, November 2009.

[7] S. Kannan, J. Rajendran, R. Karri and O. Sinanoglu, "Sneak-path Testing of Memristor-based Memories," *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, Pune, 2013, pp. 386-391.

[8] A. Chen, "A Comprehensive Crossbar Array Model With Solutions for Line Resistance and Nonlinear Device Characteristics," in *IEEE Transactions on Electron Devices*, vol. 60, no. 4, pp. 1318-1326, April 2013.

[9] Li, T., Bi, X., Jing, N., Liang, X., & Jiang, L. (2017). Sneak-Path Based Test and Diagnosis for 1R RRAM Crossbar Using Voltage Bias Technique. Proceedings of the 54th Annual Design Automation Conference 2017. https://doi.org/10.1145/3061639.3062318

[10] M. Teimoori, A. Amirsoleimani, A. Ahmadi and M. Ahmadi, "A 2M1M Crossbar Architecture: Memory," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2608-2618, Dec. 2018.

[11] Kim, K. M., Zhang, J., Graves, C., Yang, J. J., Choi, B. J., Hwang, C. S., & Williams, R. S. (2016). Low-Power, Rectifying, and Forming-Free Memristor with an

Asymmetric Programing Voltage for a High-Density Crossbar Application. *Nano letters*, *16*(11), 6724-6732.

[12] Golubović, D. S., Miranda, A. H., Akil, N., Van Schaijk, R. T. F., & Van Duuren, M. J. (2007). Vertical poly-Si select pn-diodes for emerging resistive non-volatile memories. *Microelectronic engineering*, *84*(12), 2921-2926.

[13] Kim, K. H., Hyun Jo, S., Gaba, S., & Lu, W. (2010). Nanoscale resistive memory with intrinsic diode characteristics and long endurance. *Applied Physics Letters*, *96*(5), 053106.

[14] M. Nourazar, V. Rashtchi, A. Azarpeyvand and F. Merrikh-Bayat, "Code Acceleration Using Memristor-Based Approximate Matrix Multiplier: Application to Convolutional Neural Networks," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2684-2695, Dec. 2018.

[15] M. T. Arafin and G. Qu, "Memristors for Secret Sharing-Based Lightweight Authentication," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2671-2683, Dec. 2018.

[16] A. Grossi *et al.*, "Experimental Investigation of 4-kb RRAM Arrays Programming Conditions Suitable for TCAM," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2599-2607, Dec. 2018.

[17] K. C. Rahman, D. Hammerstrom, Y. Li, H. Castagnaro and M. A. Perkowski, "Methodology and Design of a Massively Parallel Memristive Stateful IMPLY Logic-Based Reconfigurable Architecture," in *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 675-686, July 2016, doi: 10.1109/TNANO.2016.257272

[18] M. J. Aljafar, M. A. Perkowski, J. M. Acken and R. Tan, "A Time-Efficient CMOS-Memristive Programmable Circuit Realizing Logic Functions in Generalized AND–XOR Structures," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 23-36, Jan. 2018, doi: 10.1109/TVLSI.2017.2750074.

[19] S. N. Truong, "Single Crossbar Array of Memristors With Bipolar Inputs for Neuromorphic Image Recognition," in *IEEE Access*, vol. 8, pp. 69327-69332, 2020

[20] G. S. Rose, N. McDonald, L. Yan, B. Wysocki and K. Xu, "Foundations of memristor based PUF architectures," *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Brooklyn, NY, 2013, pp. 52-57.

[21] A. Mazady, M. T. Rahman, D. Forte and M. Anwar, "Memristor PUF—A Security Primitive: Theory and Experiment," in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 5, no. 2, pp. 222-229, June 2015.

[22] D. Chakraborty and S. K. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, 2017, pp. 770-77.

[23] A. Velasquez and S. K. Jha, "Parallel computing using memristive crossbar networks: Nullifying the processor-memory bottleneck", *9th International Design & Test Symposium (IDT) 2014*, pp. 147-152, 2014.

[24] A. Velasquez, "Automated synthesis of crossbars for nanoscale computing using formal methods", *Nanoscale Architectures (NANOARCH) 2015 IEEE/ACM International Symposium on*, pp. 130-136, 2015.

[25] G. S. Rose and C. A. Meade, "Performance analysis of a memristive crossbar PUF design," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, CA, 2015, pp. 1-6, doi: 10.1145/2744769.2744892.

[26] D. Bhattacharjee, R. Devadoss and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW architecture for in-memory computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, 2017, pp. 782-787, doi: 10.23919/DATE.2017.7927095.

[27] Biolek, Z., Biolek, D., & Biolkova, V. (2009). SPICE Model of Memristor with Nonlinear Dopant Drift. *Radioengineering*, *18*(2).

[28] Prodromakis, T., Peh, B. P., Papavassiliou, C., & Toumazou, C. (2011). A versatile memristor model with nonlinear dopant kinetics. *IEEE transactions on electron devices*, *58*(9), 3099-3105.

[29] Lehtonen, E., & Laiho, M. (2010, February). CNN using memristors for neighborhood connections. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on* (pp. 1-4). IEEE.

[30] Yang, J. J., Pickett, M. D., Li, X., Ohlberg, D. A., Stewart, D. R., & Williams, R. S. (2008). Memristive switching mechanism for metal/oxide/metal nanodevices. *Nature nanotechnology*, *3*(7), 429-433.

[31] Pickett, M. D., Strukov, D. B., Borghetti, J. L., Yang, J. J., Snider, G. S., Stewart, D. R., & Williams, R. S. (2009). Switching dynamics in titanium dioxide memristive devices. *Journal of Applied Physics*, *106*(7), 074508.

[32] Kvatinsky, S., Friedman, E. G., Kolodny, A., & Weiser, U. C. (2013). TEAM: Threshold adaptive memristor model. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *60*(1), 211-221.

[33] Zidan, M. A., Fahmy, H. A. H., Hussain, M. M., & Salama, K. N. (2013). Memristor-based memory: The sneak paths problem and solutions. *Microelectronics Journal*, *44*(2), 176–183. https://doi.org/10.1016/j.mejo.2012.10.001

[34] Tang, Z., Wang, Y., Chi, Y., & Fang, L. (2018). Comprehensive Sensing Current Analysis and Its Guideline for the Worst-Case Scenario of RRAM Read Operation. *Electronics*, *7*(10), 224. https://doi.org/10.3390/electronics7100224

[35] Cassuto, Y., Kvatinsky, S., & Yaakobi, E. (2013). Sneak-path constraints in memristor crossbar arrays. *2013 IEEE International Symposium on Information Theory*. https://doi.org/10.1109/isit.2013.6620207

[36] Cassuto, Y., Kvatinsky, S., & Yaakobi, E. (2016). Information-Theoretic Sneak-Path Mitigation in Memristor Crossbar Arrays. *IEEE Transactions on Information Theory*, *62*(9), 4801–4813. https://doi.org/10.1109/tit.2016.2594798

[37] Sun, L., Zheng, N., Zhang, T., & Mazumder, P. (2018). Fault Modeling and Parallel Testing for 1T1M Memory Array. *IEEE Transactions on Nanotechnology*, *17*(3), 437–451. https://doi.org/10.1109/tnano.2018.2806938

[38] Tarkhan, M., Maymandi-Nejad, M., Klidbary, S. H., & Shouraki, S. B. (2019). A bridge technique for memristor state programming. *International Journal of Electronics*, *107*(6), 1015–1030. https://doi.org/10.1080/00207217.2019.1692371

[39] Lin, T.-Y., Chen, Y.-X., Li, J.-F., Lo, C.-Y., Kwai, D.-M., & Chou, Y.-F. (2016). A Test Method for Finding Boundary Currents of 1T1R Memristor Memories. *2016 IEEE 25th Asian Test Symposium (ATS)*. https://doi.org/10.1109/ats.2016.44

[40] Kannan, S., Karimi, N., Karri, R., & Sinanoglu, O. (2015). Modeling, Detection, and Diagnosis of Faults in Multilevel Memristor Memories. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 34(5), 822–834. https://doi.org/10.1109/tcad.2015.2394434

[41] Kannan, S., Rajendran, J., Karri, R., & Sinanoglu, O. (2013). Sneak-Path Testing of Crossbar-Based Nonvolatile Random Access Memories. IEEE Transactions on Nanotechnology, 12(3), 413–426. https://doi.org/10.1109/tnano.2013.2253329

[42] Zhang, Q., Cui, X., Xu, X., Wang, X., Ma, Z., & Zhou, S. (2016). Sneak-path based test for 3D stacked one-transistor-N-RRAM array. 2016 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC). https://doi.org/10.1109/edssc.2016.7785249

[43] Liu, R., Chen, P.-Y., & Yu, S. (2017). Design and optimization of a strong PUF exploiting sneak paths in resistive cross-point array. 2017 IEEE International Symposium on Circuits and Systems (ISCAS). https://doi.org/10.1109/iscas.2017.8050792

[44] Karakulak, E., Mutlu, R., Ucar, E. (2015) "Sneak path current equivalent circuits and reading margin analysis of complementary resistive switches based 3D stacking crossbar memories." Informacije MIDEM 44.3 2015: 235-241

[45] J. Zhou, K. Kim and W. Lu, "Crossbar RRAM Arrays: Selector Device Requirements During Read Operation," in *IEEE Transactions on Electron Devices*, vol. 61, no. 5, pp. 1369-1376, May 2014

[46] Youn, Y., Sim, J.-Y., Park, H.-J., & Kim, B. (2015). An approximate condition to avoid reverse leakage current in ReRAM crossbar design. 2015 International SoC Design Conference (ISOCC). https://doi.org/10.1109/isocc.2015.7401656

[47] Sun, W., & Shin, H. (2018). Analysis of read margin of crossbar array according to selector and resistor variation. 2018 International Conference on Electronics, Information, and Communication (ICEIC). https://doi.org/10.23919/elinfocom.2018.8330651

[48] S. Kannan, R. Karri and O. Sinanoglu, "Sneak path testing and fault modeling for multilevel memristor-based memories," *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Asheville, NC, 2013, pp. 215-220.

[49] S. Hamdioui, H. Aziza and G. C. Sirakoulis, "Memristor based memories: Technology, design and test," *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, 2014, pp. 1-7.

[50] Y. Chen and J. Li, "Fault modeling and testing of 1T1R memristor memories," *2015 IEEE 33rd VLSI Test Symposium (VTS)*, Napa, CA, 2015, pp. 1-6.

[51] Y. Luo, X. Cui, M. Luo and Q. Lin, "A high fault coverage march test for 1T1R memristor array," *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, Hsinchu, 2017, pp. 1-2.

[52] S. N. Mozaffari, S. Tragoudas and T. Haniotakis, "More Efficient Testing of Metal-Oxide Memristor–Based Memory," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 1018-1029, June 2017.

[53] Y. Li, J. Li, C. Hsu and C. Sun, "Diagnosis of Resistive Nonvolatile-8T SRAMs," *2018 International SoC Design Conference (ISOCC)*, Daegu, Korea (South), 2018, pp. 23-24.

[54] N. Z. Haron and S. Hamdioui, "DfT schemes for resistive open defects in RRAMs," 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2012, pp. 799-804.

[55] N. Z. Haron and S. Hamdioui, "On Defect Oriented Testing for Hybrid CMOS/Memristor Memory," *2011 Asian Test Symposium*, New Delhi, 2011, pp. 353-358, doi: 10.1109/ATS.2011.66.

[56] V. A. Hongal, R. Kotikalapudi, Y. Kim and M. Choi, "A novel " divide and conquer " testing technique for memristor based lookup table," *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Seoul, 2011, pp. 1-4, doi: 10.1109/MWSCAS.2011.6026406.

[57] S. Kannan, N. Karimi, R. Karri and O. Sinanoglu, "Detection, diagnosis, and repair of faults in memristor-based memories," *2014 IEEE 32nd VLSI Test Symposium (VTS)*, Napa, CA, 2014, pp. 1-6, doi: 10.1109/VTS.2014.6818762.

[58] J.-F. Li, K.-L. Cheng, C.-T. Huang and C.-W. Wu, "March-based RAM diagnosis algorithms for stuck-at and coupling faults", *Proc. Int'l Test Conf. (ITC)*, pp. 758-767, Oct. 2001.

Appendix: Source code Listing

```
"""Python tool to calculate the sneak path length, number of sneak paths
and sneak path information for a given IO switch-vector. Builds a LTspice
compatible circuit for simulation."""
#author: Rasika Joshi

from typing import List

f = open('output.txt','a')
"""output file to print the target memristor, number of sneak paths,
length of sneak paths and the sneak path for the given IO switch-vector"""

# add code to check for string inputs as well
while True: # The loop keeps running until user enters positive integer value
    v = input("Number of inputs to memristor crossbar grid\n") #number of rows or input voltages

    if v == "": #check for blanks
        print('please enter positive integer value\n')
        continue
    elif v.isalpha(): #check for alpha
        print('please enter positive integer value\n')
        continue
    elif int(float(v))== 0:
        print('please enter positive integer value\n')
        continue
    else:
        if float(v) != abs(int(float(v))):  #check for floating point
            print('please enter positive integer value\n')
            continue
        else:
            break

while True:
    i = input("Number of outputs to memristor crossbar grid\n") # number of columns or output
current

    if i == "":
        print('please enter positive integer value\n')
        continue
    elif i.isalpha():
        print('please enter positive integer value\n')
        continue
    elif int(float(i)) == 0:
        print('please enter positive integer value\n')
        continue
    else:
        if float(i) != abs(int(float(i))):
            print('please enter positive integer value\n')
            continue
```

```
        else:
            break

f.write('\n')
f.write('The memristor crossbar dimensions are ' + v + 'x' + i + '\n' )

v = int(v) # v is converted from string to integer
i = int(i) # i is converted from string to integer

# calculating number of cases
mc = ((2 ** v) - 1) * ((2 ** i) - 1) ## zero case removed from input and output

f.write('The number of IO switch-vectors are ' + str(mc) + '\n')

# calculating total primary path plus sneak path cases.

ms = ((2 ** v) - 2) * ((2 ** i) - 2)

f.write('The number of sneak path IO switch-vectors are ' + str(ms) + '\n' + '\n')

#list of notations for bitline outputs
listi = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
         'W', 'X', 'Y', 'Z'];
#list of notations for wordline inputs
listv = [];
counter = 1
while (counter <= v):           # if v =3, listv = [1,2,3]
    listv.append(counter);      # loop to add the elements to list v correspond to size of v
    counter = counter + 1

#defining all combinations of IO switch-vector for given size of array
import itertools
inp = list(map(list, itertools.product([0,1], repeat = v)))
#inp = [[1,0,0]] #if manual input is needed, please comment above line and uncomment this line
to add manually
out = list(map(list, itertools.product([0,1], repeat = i)))
#out = [[1,0,0]] #if manual input is needed, please comment above line and uncomment this line
to add manually

#Removing all zeros or all ones combinations from IO switch-vector
for x in inp:
    co = 0 #dummy variable
    dum = 0 #dummy variable
    for y in x:
        # This condition checks for all zeroes in Inp combinations
        if y == 0:
            co = co + 1
            if co == v:
                inp.remove(x)
        # This condition checks for all ones in Inp combinations
```

```
        if y == 1:
            dum = dum + 1
            if dum == v:
                inp.remove(x)

for x in out:
    co1 = 0 #dummy variable
    dum1 = 0 #dummy variable
    for y in x:
        # This condition checks for all zeroes in Out combinations
        if y == 0:
            co1 = co1 + 1
            if co1 == i:
                out.remove(x)
        # This condition checks for all ones in Out combinations
        if y == 1:
            dum1 = dum1 + 1
            if dum1 == i:
                out.remove(x)

loop = 1 #initiate dummy variable. To be used later to create LTspice.cir files for all IO
combinations
n = [0 for x in range(0,mc)]
#blank list of all zeroes. initiated with max capacity of mc elements to input LTspice file
combinations later

for xi in inp:
    for xo in out:

        inpstr = ''.join(str(e) for e in xi)
        outstr = ''.join(str(e) for e in xo)
        f.write("\nThe input combination is ")
        f.write(inpstr)
        f.write("\nThe output combination is ")
        f.write(outstr)

        #Opening LTspice file for each combination of IO switch-vector and making the file
writable
        n[loop] = 'LTspice' + inpstr + outstr + '.cir'
        j = open(n[loop],'w')

        ilet = []
        vlet = []
        #ilet is for memristor bitlines (output will be ilet = ['A', 'B', 'C',....]
        for c1 in range(0, i):
            ilet.append(listi[c1])
        #vlet is for memristor wordlines (output will be vlet = [1, 2, 3,....]
        for c2 in range(0, v):
            vlet.append(c2 + 1)
```

```
    # Procedure to find the primary memristor by intersection of 1s in the IO switch-vectors
    primary_memristors = []
    count = 0
    count1 = 0
    for nu in xi:
        count1 = 0
        if nu == 1:
            for nu1 in xo:
                if nu1 == 1:
                    primary_memristors.append('M' + str(listv[count]) + str(listi[count1]))
                    # Append to the list of primary memristors if more than 1
                    count1 = count1 + 1
                else:
                    count1 = count1 +1
            count = count + 1
        else:
            count = count + 1
    f.write('\n\nPrimary memristors in the crossbar array are\n\n')
    for x in primary_memristors:
        f.write(x + ' ')

    crossbar = []
    for x in range(0, v):
        for y in range(0, i):
            crossbar.append('M' + str(vlet[x]) + str(ilet[y]))

    #f.write('/n/n' + str(crossbar))
    # Uncomment this line to print the crossbar array elements

    # Dictrmap is a dictionary to map every element in the crossbar to R1, R2, R3.. to follow
LTSPICE conventions
    dictrmap = {}
    u = 0
    for e in crossbar:
        dictrmap[e] = 'R' + str(u + 1)
        u = u + 1
    # primi and primv are two arrays to store the split characters of the primary memristor. For
example memristor
    #M1A is split into 1 and "A" in primv and primi respectively
    primi = []
    primv = []
    for x in primary_memristors:
        prim = str(x)
        if len(x) == 3:
            primv.append(prim[1])
            primi.append(prim[2])
        else:
            primv.append(prim[1:3])
            primi.append(prim[3])
```

```
    # Remove primary memristor characters from ilet and vlet. For example, if primary
memristor is M1A, 1 and "A"
    # will be removed from vlet and ilet respectively.
    for x in primv:
        for y in vlet:
            if int(x) == int(y):
                vlet.remove(int(y))
    for x in primi:
        for y in ilet:
            if str(x) == str(y):
                ilet.remove(str(y))

    # primiu and primvu are used to create sets from primi and primv respectively
    primiu = set(primi)
    primvu = set(primv)
    #List of all the possible First memristors of the sneak path
    shortpathstart = []
    for x in ilet:
        for y in primvu:
            p = 'M' + str(y) + str(x)
            shortpathstart.append(p)
    #List of all the possible last memristors of the sneak path
    shortpathend = []
    for x in primiu:
        for y in vlet:
            p = 'M' + str(y) + str(x)
            shortpathend.append(p)
    #List of all the possible middle memristors of the sneak path
    shortpathm = list(set(crossbar).difference(primary_memristors))
    shortpathm1 = list(set(shortpathm).difference(shortpathstart))
    shortpathmid = list(set(shortpathm1).difference(shortpathend))

    minofinpout = min(i,v)
    highestsneakpathlength = ((2*minofinpout)-1)
    f.write('\n\nThe longest possible sneak path is ' + str(highestsneakpathlength) + " memristors
long")
    ## Code to obtain the sneak paths for 3 memristor long sneak paths
    path = []
    number = 3
    if number <= highestsneakpathlength:
        f.write("\n\nThe sneak paths are as follows: \n")
        for x in shortpathstart:
            if len(x) == 3: #number of chracters in the first memristor of the sneak path is 3
                for y in shortpathend:
                    if len(y)==3: # number of characters in the last memristor of the sneak path is 3
                        for z in shortpathmid:
                            if len(z) == 3: # number of characters in the middle memristor of the sneak
path are 3
                                if z[1] == y[1] and z[2] == x[2]:
                                    path.append(str(x) + ', ' + str(z) + ', ' + str(y))
```

```
                                # generate sneak path using the above condition where the middle
memristor in
                                #the sneak path will have the same bitline as the first memristor and
same
                                #wordline as the last memristor
                            else: # number of characters in the middle memristor of the sneak path are 4
                                if z[1:3] == y[1] and z[3] == x[2]:
                                    path.append(str(x) + ', ' + str(z) + ', ' + str(y))

                        else: # number of characters in the last memristor of the sneak path are 4
                            for z in shortpathmid:
                                if len(z) == 3:
                                    if z[1] == y[1:3] and z[2] == x[2]:
                                        path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                                else:
                                    if z[1:3] == y[1:3] and z[3] == x[2]:
                                        path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                    else: # number of characters in the first memristor of the sneak path are 4
                        for y in shortpathend:
                            if len(y)==3:
                                for z in shortpathmid:
                                    if len(z) == 3:
                                        if z[1] == y[1] and z[2] == x[3]:
                                            path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                                    else:
                                        if z[1:3] == y[1] and z[3] == x[3]:
                                            path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                            else:
                                for z in shortpathmid:
                                    if len(z) == 3:
                                        if z[1] == y[1:3] and z[2] == x[3]:
                                            path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                                    else:
                                        if z[1:3] == y[1:3] and z[3] == x[3]:
                                            path.append(str(x) + ', ' + str(z) + ', ' + str(y))
                # Write to output file
                for x12 in path:
                    f.write("\n" + x12)
                noofsneakpath3 = len(path)
                f.write("\n\nTotal number of sneak paths with 3 memristor length are " +
str(noofsneakpath3))
                f.write(' \n')
                f.write(' \n')

            ## Code to obtain the sneak paths for 5 memristor long sneak paths
            number = 5
            if number <= highestsneakpathlength:
                path5 = [str(x) + ', ' + str(z1) + ', ' + str(z2) + ', ' + str(z3) + ', ' + str(y) for x in
shortpathstart
                         for y in
```

```
                shortpathend for z1 in shortpathmid if z1[2] == x[2] for z2 in shortpathmid if z2 !=
z1 and
                (z1[1]) == (z2[1]) and z2[2] != z1[2] for z3 in shortpathmid if
                z3 != z2 and z3 != z1 and z2[2] == z3[2]
                and z3[1] == y[1] and z3[1] != z2[1] and z3[1] != z1[1]]

        for x12 in path5:
            f.write("\n" + x12)

        noofsneakpath5 = len(path5)
        f.write("\n\nTotal number of sneak paths with 5 menristor length are " +
str(noofsneakpath5) + "\n")
        f.write(' \n')
        f.write(' \n')
    number = 7
    if number <= highestsneakpathlength:
        path7 = [str(x) + ', ' + str(z1) + ', ' + str(z2) + ', ' + str(z3) + ', ' + str(z4) + ', ' + str(
            z5) + ', ' + str(y)
                for x in shortpathstart for y in shortpathend for z1 in shortpathmid if z1[2] == x[2]
                for z2 in shortpathmid if z2 != z1 and z1[1] == z2[1] and z2[2] != z1[2] for z3 in
shortpathmid
                if z3 != z2 and z3 != z1 and z2[2] == z3[2] and z3[1] != z2[1] and z3[1] != z1[1] for
z4 in
                shortpathmid
                if z4 != z3 and z4 != z2 and z4 != z1 and z3[1] == z4[1] and z4[2] != z3[2] and z4[2]
!= z2[2]
                and z4[2] != z1[2] for z5 in shortpathmid if z5 != z4 and z5 != z3 and z5 != z2 and
z5 != z1
                and z4[2] == z5[2] and z5[1] == y[1] and z5[1] != z4[1] and z5[1] != z3[1] and
z5[1] != z2[1]
                and z5[1] != z1[1]]

        for x12 in path7:
            f.write("\n" + x12)

        noofsneakpath7 = len(path7)
        f.write("\n\nTotal number of sneak paths with 7 menristor length are " +
str(noofsneakpath7) + "\n")
        f.write(' \n')
        f.write(' \n')

    number = 9
    if number <= highestsneakpathlength:
        path9 = [
            str(x) + ', ' + str(z1) + ', ' + str(z2) + ', ' + str(z3) + ', ' + str(z4) + ', ' + str(z5) + ', ' +
            str(z6) + ', ' + str(z7) + ', ' + str(y) for x in shortpathstart for y in shortpathend for z1 in
            shortpathmid if z1[2] == x[2] for z2 in shortpathmid if z2 != z1 and z1[1] == z2[1] and
z2[2] != z1[2]
            for z3 in shortpathmid if z3 != z2 and z3 != z1 and z2[2] == z3[2] and z3[1] != z2[1]
and z3[1] != z1[1]
```

```
            for z4 in shortpathmid if z4 != z3 and z4 != z2 and z4 != z1 and z3[1] == z4[1] and
z4[2] != z3[2]
            and z4[2] != z2[2] and z4[2] != z1[2] for z5 in shortpathmid if z5 != z4 and z5 != z3
and z5 != z2
            and z5 != z1 and z4[2] == z5[2] and z5[1] != z4[1] and z5[1] != z3[1] and z5[1] !=
z2[1] and z5[1] !=
            z1[1]
            for z6 in shortpathmid if
            z6 != z5 and z6 != z4 and z6 != z3 and z6 != z2 and z6 != z1 and z5[1] == z6[1]
            and z6[2] != z5[2] and z6[2] != z4[2] and z6[2] != z3[2] and z6[2] != z2[2] and z6[2] !=
z1[2]
            for z7 in shortpathmid if z7 != z6 and z7 != z5 and z7 != z4 and z7 != z3 and z7 != z2
and z7 != z1
            and z6[2] == z7[2] and z7[1] == y[1] and z7[1] != z6[1] and z7[1] != z5[1] and z7[1] !=
z4[1] and
            z7[1] != z3[1] and z7[1] != z2[1] and z7[1] != z1[1]]

         for x12 in path9:
            f.write("\n" + x12)

         noofsneakpath9 = len(path9)
         f.write("\n\nTotal number of sneak paths with 9 menristor length are " +
str(noofsneakpath9) + "\n")
         f.write(' \n')
         f.write(' \n')

      number = 11
      if number <= highestsneakpathlength:
         path11 = [
            str(x) + ', ' + str(z1) + ', ' + str(z2) + ', ' + str(z3) + ', ' + str(z4) + ', ' + str(z5) + ', ' +
            str(z6) + ', ' + str(z7) + ', ' + str(z8) + ', ' + str(z9) + ', ' + str(y) for x in shortpathstart
            for y in shortpathend for z1 in shortpathmid if z1[2] == x[2] for z2 in shortpathmid if z2
!= z1
            and z1[1] == z2[1] and z2[2] != z1[2] for z3 in shortpathmid if z3 != z2 and z3 != z1
and z2[2] == z3[2]
            and z3[1] != z2[1] and z3[1] != z1[1] for z4 in shortpathmid if z4 != z3 and z4 != z2
and z4 != z1
            and z3[1] == z4[1] and z4[2] != z3[2] and z4[2] != z2[2] and z4[2] != z1[2] for z5 in
shortpathmid
            if
            z5 != z4 and z5 != z3 and z5 != z2 and z5 != z1 and z4[2] == z5[2] and z5[1] != z4[1]
and z5[1] != z3[1]
            and z5[1] != z2[1] and z5[1] != z1[1] for z6 in shortpathmid if z6 != z5 and z6 != z4
and z6 != z3
            and z6 != z2 and z6 != z1 and z5[1] == z6[1] and z6[2] != z5[2] and z6[2] != z4[2] and
z6[2] != z3[2]
            and z6[2] != z2[2] and z6[2] != z1[2] for z7 in shortpathmid if z7 != z6 and z7 != z5
and z7 != z4
            and z7 != z3 and z7 != z2 and z7 != z1 and z6[2] == z7[2] and z7[1] != z6[1]
```

```
            and z7[1] != z5[1] and z7[1] != z4[1] and z7[1] != z3[1] and z7[1] != z2[1] and z7[1] !=
z1[1]
            for z8 in shortpathmid if z8 != z7 and z8 != z6 and z8 != z5 and z8 != z4 and z8 != z3
and z8 != z2
            and z8 != z1 and z8[1] == z7[1] and z8[2] != z7[2] and z8[2] != z6[2] and z8[2] !=
z5[2]
            and z8[2] != z4[2] and z8[2] != z3[2] and z8[2] != z2[2] and z8[2] != z1[2]
            for z9 in shortpathmid if
            z9 != z8 and z9 != z7 and z9 != z6 and z9 != z5 and z9 != z4 and z9 != z3 and z9 != z2
            and z9 != z1 and z9[2] == z8[2] and z9[1] == y[1] and z9[1] != z8[1] and z9[1] != z7[1]
and z9[1] != z6[
               1]
            and z9[1] != z5[1] and z9[1] != z4[1] and z9[1] != z3[1] and z9[1] != z2[1] and z9[1] !=
z1[1]]

         for x12 in path11:
            f.write("\n" + x12)

         noofsneakpath11 = len(path11)
         f.write("\n\nTotal number of sneak paths with 11 menristor length are " +
str(noofsneakpath11) + "\n")
         f.write(' \n')
         f.write(' \n')


#########################################################################################
#######################################
# LT spice circuit file generation

      dictstart = {}
      dictstop = {}
      dictstart1 = {}
      dictstop1 = {}
      count = 1
      node = 1


      path1 = []
      tsp = ((2**i)-2)*((2**v)-2)
      strdum = (i*v) + 1
      # dictstart1 is dictionary for number of Rx which can be utilized as ground resistance
      # this will have a huge node value assigned to not have them coincide with any existing
values in dictrmap
      for b in range(1,(tsp+1)):
         dictstart1[('R' + str(strdum))] = 1000000000000 + b
         #dictstop1[('R' + str(strdum))] = 0
         strdum = strdum + 1
      # Utilizes same code as for sneak path generation. Path1 has all elements stored individually
instead of 3 long paths
      for x in shortpathstart:
```

```
if len(x) == 3:
    for y in shortpathend:
        if len(y) == 3:
            for z in shortpathmid:
                if len(z) == 3:
                    if z[1] == y[1] and z[2] == x[2]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
                else:
                    if z[1:3] == y[1] and z[3] == x[2]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
        else:
            for z in shortpathmid:
                if len(z) == 3:
                    if z[1] == y[1:3] and z[2] == x[2]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
                else:
                    if z[1:3] == y[1:3] and z[3] == x[2]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
else:
    for y in shortpathend:
        if len(y) == 3:
            for z in shortpathmid:
                if len(z) == 3:
                    if z[1] == y[1] and z[2] == x[3]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
                else:
                    if z[1:3] == y[1] and z[3] == x[3]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
        else:
            for z in shortpathmid:
                if len(z) == 3:
                    if z[1] == y[1:3] and z[2] == x[3]:
                        path1.append(str(x))
                        path1.append(str(z))
                        path1.append(str(y))
                else:
                    if z[1:3] == y[1:3] and z[3] == x[3]:
                        path1.append(str(x))
```

```
                        path1.append(str(z))
                        path1.append(str(y))

    dictr = {}
    pathi = []
    for x12 in path1:
        if len(x12) == 3:
            y = x12[0:3]
        else:
            y = x12[0:4]
        pathi.append(y)

    pathi1 = list(set(pathi))
    # dictr is dictionary for all memristors used in sneak path in terms of LTspice Rx variables
    for u in pathi1:
        dictr[u] = dictrmap[u]

    count = 1
    node = 1
    v = len(xi) #inp voltage
    vol = []
    # vol is list of number of voltages in terms of V1, V2, V3....
    for u in range(1, (v + 1)):
        vol.append('V' + str(u))

    for x123 in xi:
        if x123 == 1:
            nodestart = node
            nodestop = 0
            dictstart[vol[(count - 1)]] = nodestart # maps voltage to start nodes
            dictstop[vol[(count - 1)]] = nodestop # maps voltage to end nodes

            cou = (i*v) + 1
            for x12 in path:

                if x12[1:3].isalnum(): #checks to see if memristor is 3 character long
                    if int(x12[1]) == count:
                        if len(x12) == 13:
                            # below code determines start and stop nodes for first memristor in path
                            if dictr[x12[0:3]] not in dictstart: # condition to ensure elements already in
dictstart are not repeated
                                if vol[(count - 1)] in dictstart:
                                    # Below assigns start node of voltage to nodestart.
                                    # This will eventually be assigned to first memristor corresponding to
Voltage
                                    nodestart = dictstart[vol[(count - 1)]]
                                    nodestop = node + 1
                                    x121 = dictr[x12[0:3]] #Rx equivalent of first memristor in sneak path
                                    dictstart[x121] = nodestart
                                    dictstop[x121] = nodestop
```

```
                        #if second memristor in path already has nodes assigned, below
condition checks
                        #those nodes and assigns start node of second memristor in path to stop
node of
                        #first memristor in path
                        if dictr[x12[5:8]] in dictstop:
                            nodestart = dictstart[vol[(count - 1)]]
                            nodestop = dictstart[dictr[(x12[5:8])]]
                            x121 = dictr[x12[0:3]]
                            dictstart[x121] = nodestart
                            dictstop[x121] = nodestop
                    # below code determines start and stop nodes for second memristor in path
                    if dictr[x12[5:8]] not in dictstart: # condition to ensure elements already in
dictstart are not repeated
                        if dictr[x12[0:3]] in dictstart:
                            x122 = dictr[x12[5:8]] #Rx equivalent of first memristor in sneak path
                            nodestart = dictstop[dictr[(x12[0:3])]]
                            nodestop = node + 2
                            dictstart[x122] = nodestart
                            dictstop[x122] = nodestop
                            # if third memristor in path already has nodes assigned, below condition
checks
                            # those nodes and assigns start node of third memristor in path to stop
node of
                            # second memristor in path
                            if dictr[x12[10:13]] in dictstop:
                                nodestart = dictstop[(dictr[x12[0:3]])]
                                nodestop = dictstart[(dictr[x12[10:13]])]
                                x122 = dictr[x12[5:8]]
                                dictstart[x122] = nodestart
                                dictstop[x122] = nodestop
                    # below code determines start and stop nodes for second memristor in path
                    if dictr[x12[10:13]] not in dictstart:
                        if dictr[x12[5:8]] in dictstart:
                            x123 = dictr[x12[10:13]]
                            nodestart = dictstop[dictr[(x12[5:8])]]
                            nodestop = dictstart1[('R'+ str(cou))] #circuit ends with a ground node
                            dictstart[x123] = nodestart
                            dictstop[x123] = nodestop
                            dictstart[('R'+ str(cou))] = dictstart1[('R'+ str(cou))]
                            dictstop[('R'+ str(cou))] = 0 #ground node


                elif len(x12)== 14:
                    if x12[1:3].isnumeric():
                        if dictr[x12[0:4]] not in dictstart:
                            if vol[(count - 1)] in dictstart:
                                nodestart = dictstart[vol[(count - 1)]]
                                nodestop = node + 1
                                x121 = dictr[x12[0:4]]
```

```
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[6:9]] in dictstop:
               nodestart = dictstart[vol[(count - 1)]]
               nodestop = dictstart[dictr[(x12[6:9])]]
               x121 = dictr[x12[0:4]]
               dictstart[x121] = nodestart
               dictstop[x121] = nodestop
      if dictr[x12[6:9]] not in dictstart:
         if dictr[x12[0:4]] in dictstart:
            x122 = dictr[x12[6:9]]
            nodestart = dictstop[dictr[(x12[0:4])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[11:14]] in dictstop:
               nodestart = dictstop[(dictr[x12[0:4]])]
               nodestop = dictstart[(dictr[x12[11:14]])]
               x122 = dictr[x12[6:9]]
               dictstart[x122] = nodestart
               dictstop[x122] = nodestop

      if dictr[x12[11:14]] not in dictstart:
         if dictr[x12[6:9]] in dictstart:
            x123 = dictr[x12[11:14]]
            nodestart = dictstop[dictr[(x12[6:9])]]
            nodestop = dictstart1[('R'+ str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


   elif x12[6:8].isnumeric():
      if dictr[x12[0:3]] not in dictstart:
         if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:3]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[5:9]] in dictstop:
               nodestart = dictstart[vol[(count - 1)]]
               nodestop = dictstart[dictr[(x12[5:9])]]
               x121 = dictr[x12[0:3]]
               dictstart[x121] = nodestart
               dictstop[x121] = nodestop
      if dictr[x12[5:9]] not in dictstart:
         if dictr[x12[0:3]] in dictstart:
            x122 = dictr[x12[5:9]]
```

```
            nodestart = dictstop[dictr[(x12[0:3])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[11:14]] in dictstop:
               nodestart = dictstop[(dictr[x12[0:3]])]
               nodestop = dictstart[(dictr[x12[11:14]])]
               x122 = dictr[x12[5:9]]
               dictstart[x122] = nodestart
               dictstop[x122] = nodestop

      if dictr[x12[11:14]] not in dictstart:
         if dictr[x12[5:9]] in dictstart:
            x123 = dictr[x12[11:14]]
            nodestart = dictstop[dictr[(x12[5:9])]]
            nodestop = dictstart1[('R'+ str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

   elif x12[11:13].isnumeric():
      if dictr[x12[0:3]] not in dictstart:
         if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:3]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[5:8]] in dictstop:
               nodestart = dictstart[vol[(count - 1)]]
               nodestop = dictstart[dictr[(x12[5:8])]]
               x121 = dictr[x12[0:3]]
               dictstart[x121] = nodestart
               dictstop[x121] = nodestop
      if dictr[x12[5:8]] not in dictstart:
         if dictr[x12[0:3]] in dictstart:
            x122 = dictr[x12[5:8]]
            nodestart = dictstop[dictr[(x12[0:3])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[10:14]] in dictstop:
               nodestart = dictstop[(dictr[x12[0:3]])]
               nodestop = dictstart[(dictr[x12[10:14]])]
               x122 = dictr[x12[5:8]]
               dictstart[x122] = nodestart
               dictstop[x122] = nodestop

      if dictr[x12[10:14]] not in dictstart:
```

```
            if dictr[x12[5:8]] in dictstart:
                x123 = dictr[x12[10:14]]
                nodestart = dictstop[dictr[(x12[5:8])]]
                nodestop = dictstart1[('R'+ str(cou))]
                dictstart[x123] = nodestart
                dictstop[x123] = nodestop
                dictstop[('R' + str(cou))] = 0
                dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

    elif len(x12) == 15:
        if x12[1:3].isnumeric() and x12[7:9].isnumeric():
            if dictr[x12[0:4]] not in dictstart:
                if vol[(count - 1)] in dictstart:
                    nodestart = dictstart[vol[(count - 1)]]
                    nodestop = node + 1
                    x121 = dictr[x12[0:4]]
                    dictstart[x121] = nodestart
                    dictstop[x121] = nodestop
                    if dictr[x12[6:10]] in dictstop:
                        nodestart = dictstart[vol[(count - 1)]]
                        nodestop = dictstart[dictr[(x12[6:10])]]
                        x121 = dictr[x12[0:4]]
                        dictstart[x121] = nodestart
                        dictstop[x121] = nodestop
            if dictr[x12[6:10]] not in dictstart:
                if dictr[x12[0:4]] in dictstart:
                    x122 = dictr[x12[6:10]]
                    nodestart = dictstop[dictr[(x12[0:4])]]
                    nodestop = node + 2
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop
                    if dictr[x12[12:15]] in dictstop:
                        nodestart = dictstop[(dictr[x12[0:4]])]
                        nodestop = dictstart[(dictr[x12[12:15]])]
                        x122 = dictr[x12[6:10]]
                        dictstart[x122] = nodestart
                        dictstop[x122] = nodestop

            if dictr[x12[12:15]] not in dictstart:
                if dictr[x12[6:10]] in dictstart:
                    x123 = dictr[x12[12:15]]
                    nodestart = dictstop[dictr[(x12[6:10])]]
                    nodestop = dictstart1[('R'+ str(cou))]
                    dictstart[x123] = nodestart
                    dictstop[x123] = nodestop
                    dictstop[('R' + str(cou))] = 0
                    dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


        elif x12[1:3].isnumeric() and x12[12:14].isnumeric():
```

```
    if dictr[x12[0:4]] not in dictstart:
        if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:4]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[6:9]] in dictstop:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = dictstart[dictr[(x12[6:9])]]
                x121 = dictr[x12[0:4]]
                dictstart[x121] = nodestart
                dictstop[x121] = nodestop
    if dictr[x12[6:9]] not in dictstart:
        if dictr[x12[0:4]] in dictstart:
            x122 = dictr[x12[6:9]]
            nodestart = dictstop[dictr[(x12[0:4])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[11:15]] in dictstop:
                nodestart = dictstop[(dictr[x12[0:4]])]
                nodestop = dictstart[(dictr[x12[11:15]])]
                x122 = dictr[x12[6:9]]
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop

    if dictr[x12[11:15]] not in dictstart:
        if dictr[x12[6:9]] in dictstart:
            x123 = dictr[x12[11:15]]
            nodestart = dictstop[dictr[(x12[6:9])]]
            nodestop = dictstart1[('R'+ str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

elif x12[6:8].isnumeric() and x12[12:14].isnumeric():
    if dictr[x12[0:3]] not in dictstart:
        if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:3]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[5:9]] in dictstop:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = dictstart[dictr[(x12[5:9])]]
                x121 = dictr[x12[0:3]]
                dictstart[x121] = nodestart
```

```
                dictstop[x121] = nodestop
        if dictr[x12[5:9]] not in dictstart:
            if dictr[x12[0:3]] in dictstart:
                x122 = dictr[x12[5:9]]
                nodestart = dictstop[dictr[(x12[0:3])]]
                nodestop = node + 2
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop
                if dictr[x12[11:15]] in dictstop:
                    nodestart = dictstop[(dictr[x12[0:3]])]
                    nodestop = dictstart[(dictr[x12[11:15]])]
                    x122 = dictr[x12[5:9]]
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop

        if dictr[x12[11:15]] not in dictstart:
            if dictr[x12[5:9]] in dictstart:
                x123 = dictr[x12[11:15]]
                nodestart = dictstop[dictr[(x12[5:9])]]
                nodestop = dictstart1[('R'+ str(cou))]
                dictstart[x123] = nodestart
                dictstop[x123] = nodestop
                dictstop[('R' + str(cou))] = 0
                dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


elif len(x12) == 16:
    if dictr[x12[0:4]] not in dictstart:
        if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:4]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[6:10]] in dictstop:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = dictstart[dictr[(x12[6:10])]]
                x121 = dictr[x12[0:4]]
                dictstart[x121] = nodestart
                dictstop[x121] = nodestop
    if dictr[x12[6:10]] not in dictstart:
        if dictr[x12[0:4]] in dictstart:
            x122 = dictr[x12[6:10]]
            nodestart = dictstop[dictr[(x12[0:4])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[12:16]] in dictstop:
                nodestart = dictstop[(dictr[x12[0:4]])]
                nodestop = dictstart[(dictr[x12[12:16]])]
```

```
                    x122 = dictr[x12[6:10]]
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop

            if dictr[x12[12:16]] not in dictstart:
                if dictr[x12[6:10]] in dictstart:
                    x123 = dictr[x12[12:16]]
                    nodestart = dictstop[dictr[(x12[6:10])]]
                    nodestop = dictstart1[('R'+ str(cou))]
                    dictstart[x123] = nodestart
                    dictstop[x123] = nodestop
                    dictstop[('R' + str(cou))] = 0
                    dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

if x12[1:3].isnumeric():
    if int(x12[1:3]) == count:
        if len(x12) == 13:
            if dictr[x12[0:3]] not in dictstart:
                if vol[(count - 1)] in dictstart:
                    nodestart = dictstart[vol[(count - 1)]]
                    nodestop = node + 1
                    x121 = dictr[x12[0:3]]
                    dictstart[x121] = nodestart
                    dictstop[x121] = nodestop
                    if dictr[x12[5:8]] in dictstop:
                        nodestart = dictstart[vol[(count - 1)]]
                        nodestop = dictstart[dictr[(x12[5:8])]]
                        x121 = dictr[x12[0:3]]
                        dictstart[x121] = nodestart
                        dictstop[x121] = nodestop
            if dictr[x12[5:8]] not in dictstart:
                if dictr[x12[0:3]] in dictstart:
                    x122 = dictr[x12[5:8]]
                    nodestart = dictstop[dictr[(x12[0:3])]]
                    nodestop = node + 2
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop
                    if dictr[x12[10:13]] in dictstop:
                        nodestart = dictstop[(dictr[x12[0:3]])]
                        nodestop = dictstart[(dictr[x12[10:13]])]
                        x122 = dictr[x12[5:8]]
                        dictstart[x122] = nodestart
                        dictstop[x122] = nodestop

            if dictr[x12[10:13]] not in dictstart:
                if dictr[x12[5:8]] in dictstart:
                    x123 = dictr[x12[10:13]]
                    nodestart = dictstop[dictr[(x12[5:8])]]
                    nodestop = dictstart1[('R' + str(cou))]
                    dictstart[x123] = nodestart
```

```
            dictstop[x123] = nodestop
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]
            dictstop[('R' + str(cou))] = 0


elif len(x12) == 14:
    if x12[1:3].isnumeric():
        if dictr[x12[0:4]] not in dictstart:
            if vol[(count - 1)] in dictstart:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = node + 1
                x121 = dictr[x12[0:4]]
                dictstart[x121] = nodestart
                dictstop[x121] = nodestop
                if dictr[x12[6:9]] in dictstop:
                    nodestart = dictstart[vol[(count - 1)]]
                    nodestop = dictstart[dictr[(x12[6:9])]]
                    x121 = dictr[x12[0:4]]
                    dictstart[x121] = nodestart
                    dictstop[x121] = nodestop
        if dictr[x12[6:9]] not in dictstart:
            if dictr[x12[0:4]] in dictstart:
                x122 = dictr[x12[6:9]]
                nodestart = dictstop[dictr[(x12[0:4])]]
                nodestop = node + 2
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop
                if dictr[x12[11:14]] in dictstop:
                    nodestart = dictstop[(dictr[x12[0:4]])]
                    nodestop = dictstart[(dictr[x12[11:14]])]
                    x122 = dictr[x12[6:9]]
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop

        if dictr[x12[11:14]] not in dictstart:
            if dictr[x12[6:9]] in dictstart:
                x123 = dictr[x12[11:14]]
                nodestart = dictstop[dictr[(x12[6:9])]]
                nodestop = dictstart1[('R' + str(cou))]
                dictstart[x123] = nodestart
                dictstop[x123] = nodestop
                dictstop[('R' + str(cou))] = 0
                dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


    elif x12[6:8].isnumeric():
        if dictr[x12[0:3]] not in dictstart:
            if vol[(count - 1)] in dictstart:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = node + 1
```

```
            x121 = dictr[x12[0:3]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[5:9]] in dictstop:
               nodestart = dictstart[vol[(count - 1)]]
               nodestop = dictstart[dictr[(x12[5:9])]]
               x121 = dictr[x12[0:3]]
               dictstart[x121] = nodestart
               dictstop[x121] = nodestop
      if dictr[x12[5:9]] not in dictstart:
         if dictr[x12[0:3]] in dictstart:
            x122 = dictr[x12[5:9]]
            nodestart = dictstop[dictr[(x12[0:3])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[11:14]] in dictstop:
               nodestart = dictstop[(dictr[x12[0:3]])]
               nodestop = dictstart[(dictr[x12[11:14]])]
               x122 = dictr[x12[5:9]]
               dictstart[x122] = nodestart
               dictstop[x122] = nodestop

      if dictr[x12[11:14]] not in dictstart:
         if dictr[x12[5:9]] in dictstart:
            x123 = dictr[x12[11:14]]
            nodestart = dictstop[dictr[(x12[5:9])]]
            nodestop = dictstart1[('R' + str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

   elif x12[11:13].isnumeric():
      if dictr[x12[0:3]] not in dictstart:
         if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:3]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[5:8]] in dictstop:
               nodestart = dictstart[vol[(count - 1)]]
               nodestop = dictstart[dictr[(x12[5:8])]]
               x121 = dictr[x12[0:3]]
               dictstart[x121] = nodestart
               dictstop[x121] = nodestop
      if dictr[x12[5:8]] not in dictstart:
         if dictr[x12[0:3]] in dictstart:
            x122 = dictr[x12[5:8]]
```

```
                nodestart = dictstop[dictr[(x12[0:3])]]
                nodestop = node + 2
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop
                if dictr[x12[10:14]] in dictstop:
                    nodestart = dictstop[(dictr[x12[0:3]])]
                    nodestop = dictstart[(dictr[x12[10:14]])]
                    x122 = dictr[x12[5:8]]
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop

            if dictr[x12[10:14]] not in dictstart:
                if dictr[x12[5:8]] in dictstart:
                    x123 = dictr[x12[10:14]]
                    nodestart = dictstop[dictr[(x12[5:8])]]
                    nodestop = dictstart1[('R' + str(cou))]
                    dictstart[x123] = nodestart
                    dictstop[x123] = nodestop
                    dictstop[('R' + str(cou))] = 0
                    dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

    elif len(x12) == 15:
        if x12[1:3].isnumeric() and x12[7:9].isnumeric():
            if dictr[x12[0:4]] not in dictstart:
                if vol[(count - 1)] in dictstart:
                    nodestart = dictstart[vol[(count - 1)]]
                    nodestop = node + 1
                    x121 = dictr[x12[0:4]]
                    dictstart[x121] = nodestart
                    dictstop[x121] = nodestop
                    if dictr[x12[6:10]] in dictstop:
                        nodestart = dictstart[vol[(count - 1)]]
                        nodestop = dictstart[dictr[(x12[6:10])]]
                        x121 = dictr[x12[0:4]]
                        dictstart[x121] = nodestart
                        dictstop[x121] = nodestop
            if dictr[x12[6:10]] not in dictstart:
                if dictr[x12[0:4]] in dictstart:
                    x122 = dictr[x12[6:10]]
                    nodestart = dictstop[dictr[(x12[0:4])]]
                    nodestop = node + 2
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop
                    if dictr[x12[12:15]] in dictstop:
                        nodestart = dictstop[(dictr[x12[0:4]])]
                        nodestop = dictstart[(dictr[x12[12:15]])]
                        x122 = dictr[x12[6:10]]
                        dictstart[x122] = nodestart
                        dictstop[x122] = nodestop
```

```
    if dictr[x12[12:15]] not in dictstart:
        if dictr[x12[6:10]] in dictstart:
            x123 = dictr[x12[12:15]]
            nodestart = dictstop[dictr[(x12[6:10])]]
            nodestop = dictstart1[('R' + str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


elif x12[1:3].isnumeric() and x12[12:14].isnumeric():
    if dictr[x12[0:4]] not in dictstart:
        if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:4]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[6:9]] in dictstop:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = dictstart[dictr[(x12[6:9])]]
                x121 = dictr[x12[0:4]]
                dictstart[x121] = nodestart
                dictstop[x121] = nodestop
    if dictr[x12[6:9]] not in dictstart:
        if dictr[x12[0:4]] in dictstart:
            x122 = dictr[x12[6:9]]
            nodestart = dictstop[dictr[(x12[0:4])]]
            nodestop = node + 2
            dictstart[x122] = nodestart
            dictstop[x122] = nodestop
            if dictr[x12[11:15]] in dictstop:
                nodestart = dictstop[(dictr[x12[0:4]])]
                nodestop = dictstart[(dictr[x12[11:15]])]
                x122 = dictr[x12[6:9]]
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop

    if dictr[x12[11:15]] not in dictstart:
        if dictr[x12[6:9]] in dictstart:
            x123 = dictr[x12[11:15]]
            nodestart = dictstop[dictr[(x12[6:9])]]
            nodestop = dictstart1[('R' + str(cou))]
            dictstart[x123] = nodestart
            dictstop[x123] = nodestop
            dictstop[('R' + str(cou))] = 0
            dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

elif x12[6:8].isnumeric() and x12[12:14].isnumeric():
```

```
        if dictr[x12[0:3]] not in dictstart:
            if vol[(count - 1)] in dictstart:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = node + 1
                x121 = dictr[x12[0:3]]
                dictstart[x121] = nodestart
                dictstop[x121] = nodestop
                if dictr[x12[5:9]] in dictstop:
                    nodestart = dictstart[vol[(count - 1)]]
                    nodestop = dictstart[dictr[(x12[5:9])]]
                    x121 = dictr[x12[0:3]]
                    dictstart[x121] = nodestart
                    dictstop[x121] = nodestop
        if dictr[x12[5:9]] not in dictstart:
            if dictr[x12[0:3]] in dictstart:
                x122 = dictr[x12[5:9]]
                nodestart = dictstop[dictr[(x12[0:3])]]
                nodestop = node + 2
                dictstart[x122] = nodestart
                dictstop[x122] = nodestop
                if dictr[x12[11:15]] in dictstop:
                    nodestart = dictstop[(dictr[x12[0:3]])]
                    nodestop = dictstart[(dictr[x12[11:15]])]
                    x122 = dictr[x12[5:9]]
                    dictstart[x122] = nodestart
                    dictstop[x122] = nodestop

        if dictr[x12[11:15]] not in dictstart:
            if dictr[x12[5:9]] in dictstart:
                x123 = dictr[x12[11:15]]
                nodestart = dictstop[dictr[(x12[5:9])]]
                nodestop = dictstart1[('R' + str(cou))]
                dictstart[x123] = nodestart
                dictstop[x123] = nodestop
                dictstop[('R' + str(cou))] = 0
                dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]


elif len(x12) == 16:
    if dictr[x12[0:4]] not in dictstart:
        if vol[(count - 1)] in dictstart:
            nodestart = dictstart[vol[(count - 1)]]
            nodestop = node + 1
            x121 = dictr[x12[0:4]]
            dictstart[x121] = nodestart
            dictstop[x121] = nodestop
            if dictr[x12[6:10]] in dictstop:
                nodestart = dictstart[vol[(count - 1)]]
                nodestop = dictstart[dictr[(x12[6:10])]]
                x121 = dictr[x12[0:4]]
```

```
                        dictstart[x121] = nodestart
                        dictstop[x121] = nodestop
                if dictr[x12[6:10]] not in dictstart:
                    if dictr[x12[0:4]] in dictstart:
                        x122 = dictr[x12[6:10]]
                        nodestart = dictstop[dictr[(x12[0:4])]]
                        nodestop = node + 2
                        dictstart[x122] = nodestart
                        dictstop[x122] = nodestop
                        if dictr[x12[12:16]] in dictstop:
                            nodestart = dictstop[(dictr[x12[0:4]])]
                            nodestop = dictstart[(dictr[x12[12:16]])]
                            x122 = dictr[x12[6:10]]
                            dictstart[x122] = nodestart
                            dictstop[x122] = nodestop

                if dictr[x12[12:16]] not in dictstart:
                    if dictr[x12[6:10]] in dictstart:
                        x123 = dictr[x12[12:16]]
                        nodestart = dictstop[dictr[(x12[6:10])]]
                        nodestop = dictstart1[('R' + str(cou))]
                        dictstart[x123] = nodestart
                        dictstop[x123] = nodestop
                        dictstop[('R' + str(cou))] = 0
                        dictstart[('R' + str(cou))] = dictstart1[('R' + str(cou))]

            node = node + 10 # node increments for every new sneak path
            cou = cou + 1 #variable to increment ground node for every sneak path
        count = count + 1 #node to increment voltage numbers

    # Final output to LTspice file
    iv = i*v
    for k1 in dictstart:
        for k2 in dictstop:
            if k1 == k2:
                # Condition to put voltage nodes at start in LTspice file
                if str(k1)[0] == 'V':
                    final = str(k1) + ' ' + str(dictstart[k1]) + ' ' + str(dictstop[k2]) + ' ' + str(1) + ';'
                # Condition to add ground nodes with resistance 1
                elif int(k1[1:3]) > iv:
                    final = str(k1) + ' ' + str(dictstart[k1]) + ' ' + str(dictstop[k2]) + ' ' + str(1) + ';'
                elif int(k1[1:4]) > iv:
                    final = str(k1) + ' ' + str(dictstart[k1]) + ' ' + str(dictstop[k2]) + ' ' + str(1) + ';'
                # Condition to add memristor resistance and correct nodes for circuit connections
                else:
                    final = str(k1) + ' ' + str(dictstart[k1]) + ' ' + str(dictstop[k2]) + ' ' + str(10000) + ';'

                #print above values to LTspice circuit file
                print(final)
                j.write("\n")
```

```
                j.write(final)

        j.write('\n\n.op')
        j.close()
        loop = loop + 1



################################################################################
#####################################

f.close()
j.close()
```