2019

Automated Synthesis of Unconventional Computing Systems

Amad Ul Hassen
University of Central Florida

AUTOMATED SYNTHESIS OF UNCONVENTIONAL COMPUTING SYSTEMS

by

AMAD UL HASSEN
MSc Computer Science, University of Central Florida, 2016
MSc Electrical Engineering, University of Engineering & Technology Lahore, 2013
BSc Electrical Engineering, University of Engineering & Technology, Lahore, 2008

A Dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2019

Major Professor: Sumit Kumar Jha

ABSTRACT

Despite decades of advancements, modern computing systems which are based on the von Neumann architecture still carry its shortcomings. Moore's law, which had substantially masked the effects of the inherent memory-processor bottleneck of the von Neumann architecture, has slowed down due to transistor dimensions nearing atomic sizes. On the other hand, modern computational requirements, driven by machine learning, pattern recognition, artificial intelligence, data mining, and IoT, are growing at the fastest pace ever. By their inherent nature, these applications are particularly affected by communication-bottlenecks, because processing them requires a large number of simple operations involving data retrieval and storage. The need to address the problems associated with conventional computing systems at the fundamental level has given rise to several unconventional computing paradigms. In this dissertation, we have made advancements for automated syntheses of two types of unconventional computing paradigms: in-memory computing and stochastic computing. In-memory computing circumvents the problem of limited communication bandwidth by unifying processing and storage at the same physical locations. The advent of nanoelectronic devices in the last decade has made in-memory computing an energy-, area-, and cost-effective alternative to conventional computing. We have used Binary Decision Diagrams (BDDs) for in-memory computing on memristor crossbars. Specifically, we have used Free-BDDs, a special class of binary decision diagrams, for synthesizing crossbars for flow-based in-memory computing. Stochastic computing is a re-emerging discipline with several times smaller area/power requirements as compared to conventional computing systems. It is especially suited for fault-tolerant applications like image processing, artificial intelligence, pattern recognition, etc. We have proposed a decision procedures-based iterative algorithm to synthesize Linear Finite State Machines (LFSM) for stochastically computing non-linear functions such as polynomials, exponentials, and hyperbolic functions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Modern computers are based on the von Neumann architecture. The earliest description of the von Neumann architecture comes from *The First Draft of a Report on EDVAC* [8]. The EDVAC (Electronic Discrete Variable Automatic Computer) was envisioned to have a Central Processing Unit (CPU) –comprised of an arithmetic logic unit (ALU) and a control unit (CU)– data and program memories, and input/output peripherals. Program instructions were stored in a memory unit, just like other data in memory. Data and address buses were provided to move data between different parts of the computer. Thanks to the exponential decrease in transistor size in the coming decades, computational capabilities of CPUs kept increasing exponentially. The size of memory also increased due to the same reason. However, the communication capacity of the shared bus, connecting CPU with memory units, didn't increase at the same pace. Consequently, an increase in the computational capabilities of CPU or the size of memory did not translate into a similar increase in the throughput of computing systems. The limited speed of the shared communication-bus adversely affects system performance due to the time wasted in transferring data between CPU and memory. This phenomenon is known as *von Neumann-bottleneck*, *memory-processing bottleneck*, or *communication bottleneck*.

For decades, exponential growth in computing capabilities due to Moore's law was sufficient to mask the inherent shortcomings of the von Neumann architecture. However, in recent years, not only has Moore's law slowed down due to transistor size reaching atomic scale, but fabrication technology is also facing problems on other fronts such as, increasing leakage currents due to quantum tunneling, process variations at such a small scale, and vulnerability of small-scale electronics to radiation flips even at sea levels.

The breakdown of Dennard scaling in the mid 2000s due to increasing dynamic power meant that

the trend of increasing clock frequencies cannot continue indefinitely. This ushered the era of multicore scaling, where multiple CPUs fabricated on the same die work in parallel to improve system performance. Although multicore scaling has improved the computing throughput of general purpose computers, this trend is obstructed by the inability to dissipate all the dynamic power generated by a fully powered CPU. This phenomenon is known as dark silicon, because all of the circuitry in a die cannot be powered-on at the same time without violating the thermal constraints.

Recent advances in machine learning (ML), artificial intelligence (AI), and data mining, and their widespread applications have changed both the nature and size of computing. It has become increasingly clear that general purpose architecture cannot efficiently fulfill the demands of these specialized compute-intensive applications. For instance, Graphics Processing Units (GPU) have long eclipsed CPUs not only for multimedia, but also for AI and ML applications. Researchers are also developing AI and ML specific computing architectures such as Tensor Processing Unit (TPU), Deep Learning Accelerator (DLA), Vision Processing Unit (VPU), and Neural Network Processor (NNP). These Application Specific Integrated Circuits (ASIC) are also called AI accelerators. It is important to mention that despite its deficiencies, ASICs are not intended to replace general purpose computer. These new computing architectures such as GPUs, NPUs, and other ASICs are intended to augment the general purpose computer to increase overall computing efficiency.

Besides advances on architectural fronts, recent advances in material science and nanotechnology have propelled several new nanoelectronic devices to the forefront. Some recently discovered devices are memristors, magnetic tunnel junctions (MTJ), 3d transistors, spin-torque-transfer (STT) memories, phase change memories (PCM), etc. These newly discovered devices are collectively referred to as emerging devices, and the novel architectures employing these devices are referred to as emerging computing architectures. Small size, low to no standby power, non-volatility, and high read/write speeds are distinct features of these emerging devices as compared to traditional

Complementary Metal Oxide Semiconductor (CMOS) transistors. Circuits built from these emerging nanodevices devices are not only faster and more power efficient, they are also more compact than the traditional CMOS-based circuits. On the other hand, the problems associated with these devices are high variability, low reliability, smaller lifespan (number of read/write cycles) and differing read and write times. These devices are still in the research phase, and these parameters are likely to improve further in coming years. Nonetheless, the aforementioned properties of these devices make them excellent candidates to augment or improve existing computing systems for better power, area, delay, and computing efficiency.

Emerging ReRAM devices for Novel Architectures

The von Neumann bottleneck arises from the need to move data between CPU and memory through a shared bus. Programs which require a high number of data transfers are particularly affected by this bottleneck. In-memory computing eliminates this problem by unifying data storage and data processing at the same physical location. Resistive or magnetic memories are used as elementary cells in architectures implementing in-memory computing. Since the cells in these memories can be directly accessed using address lines, resistive and magnetic memories are also called resistive random access memories (ReRAM or RRAM) and magnetic random access memories (MRAM) respectively. In chapter 3, we have used crossbars architecture for in-memory computing of Boolean functions.

ReRAM devices were successfully realized by the early 2000s [9–11]. Snider had explored crossbars for implementing simple Boolean functions [12]. He assumed that each cell in his crossbars was composed of a hysteretic resistor which could be programmed to a high-resistance-state (1GΩ) or a low-resistance-state (1MΩ).

Despite some early works on ReRAM devices, their applications for in-memory computing were extensively researched only after 2008 when HP Labs claimed that they have realized Chua's memristor [2]. The advent of nanoelectronic devices in the upcoming years further brightened the prospects of bringing memory and computing closer to each other. The small size, higher fabrication density, low latency, and low standby power particularly favor these nanodevices for in-memory computing. Most of these devices store information in the form of resistance. Resistive memories don't need constant power to retain information. Therefore, these memories are non-volatile and consume less energy. This is in contrast to conventional DRAM or SRAM, which stores information in the form of electric charge. Since HP's claim about the realization of memristors, several other devices with similar behavior have been discovered, such as polymeric memristor, ferroelectric memristor, layered memristor, carbon nanotube memristor, and spin based memristor. In the next section, we introduce memristor and its VI-characteristics before going into the details of the HP memristor.

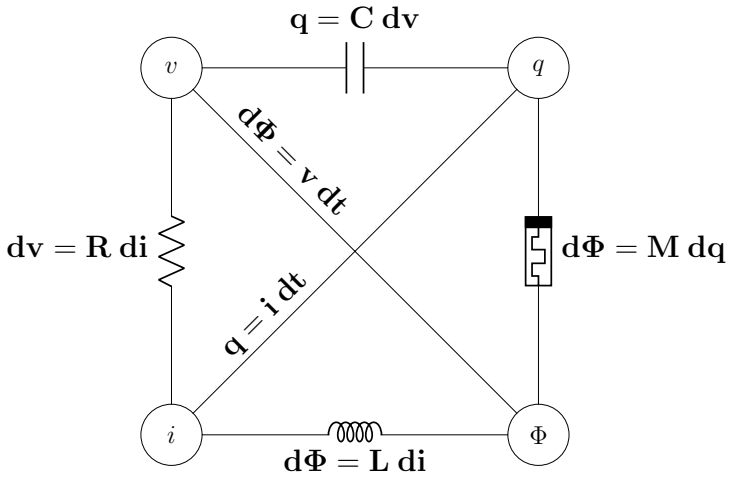


Figure 1.1: Symmetry diagram to predict the existence of memristor [1].

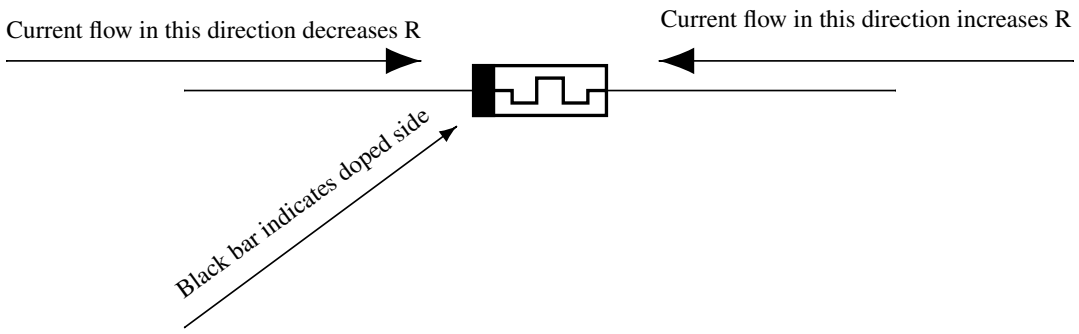


Figure 1.2: Memristor symbol. The resistance of a memristor decreases when current flows into it, the resistance increases when current flows out of it.

Memristor

Memristor is a two terminal passive device. It describes the relationship between magnetic flux and electric charge. Leon Chua used argument of symmetry (Fig. 1.1) to postulate the existence of memristor in 1971 [1]. Based on his analysis, he argued that the resistance of a memristor would depend on the charge (integral of how much electric current) that has flowed through it. The resistance of a memristor is a measure of how much current has passed through it, thus giving it the name Memristor; short for "memory" and "resistor". Fig. 1.2 shows the memristor's symbol.

Memristor remained a theoretical concept until 2008 when it was realized at HP labs [2]. The first memristor was made up of titanium oxide, which was sandwiched between platinum electrodes. The channel between the platinum electrodes consisted of doped and undoped layers of titanium dioxide (TiO_2) as shown in Fig. 1.3. The undoped layer consisted of pure TiO_2 and the doped layer had 0.5 percent less oxygen. Holes created due to absence of oxygen make the doped layer (TiO_2^-) a better conductor than the undoped TiO_2. The boundary between the doped and the undoped layers acts as a state variable. When the polarity of the external voltage is such that the current flows from the doped layer towards the undoped layer, the resistance of the memristor decreases.

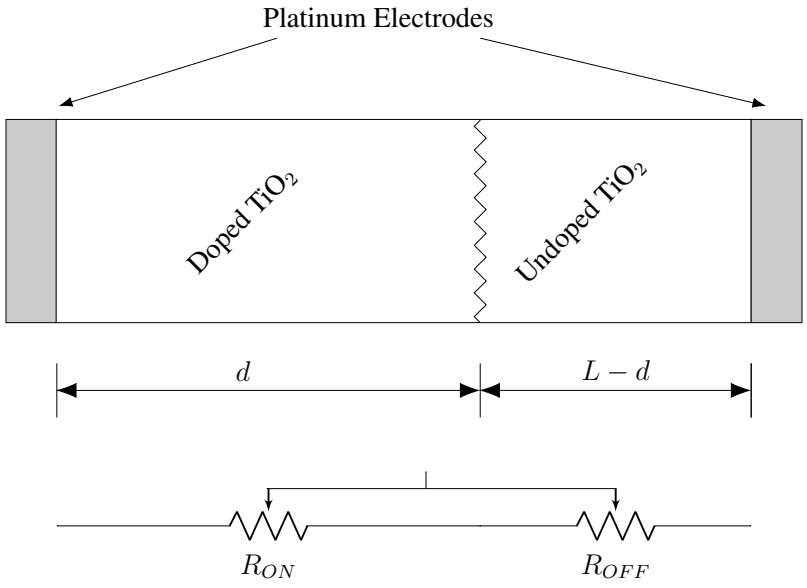


Figure 1.3: Titanium dioxide memristor and its resistive model [2].

This is because the boundary separating the doped and undoped layers moves toward the right side in Fig. 1.3, thus increasing the relative length of the conductive doped layer. When the current flows in the opposite direction, the relative length of the undoped layer increases, thus increases the overall resistance of the memristor. In terms of channel length, the total resistance of the memristor is given by equation 1.1,

$$R_{memristor} = R_{ON} \times \frac{d}{L} + R_{OFF} \times \frac{L-d}{L} \tag{1.1}$$

here d is the length of the doped layer, L is the total length of the memristive channel, R_{ON} is the resistance when the entire length of channel consists of the doped layer, and R_{OFF} is the resistance when the entire channel consists of the undoped layer. Now that we know R_{ON} (minimum) and R_{OFF} (maximum) resistances of a memristor, we define ON and OFF memristors in definitions 1 and 2.

Definition 1 (ON Memristor)**.** *An ON memristor is a memristor with minimum resistance (R_{ON}), and it is said to be in the low-resistance-state (LRS) or ON state.*

Definition 2 (OFF Memristor)**.** *An OFF memristor is a memristor with maximum resistance (R_{OFF}), and it is said to be in the high-resistance-state (HRS) or OFF state.*

In stateful logic, an ON memristor encodes binary 1 or $true$, while an OFF memristor encodes binary 0 or $false$.

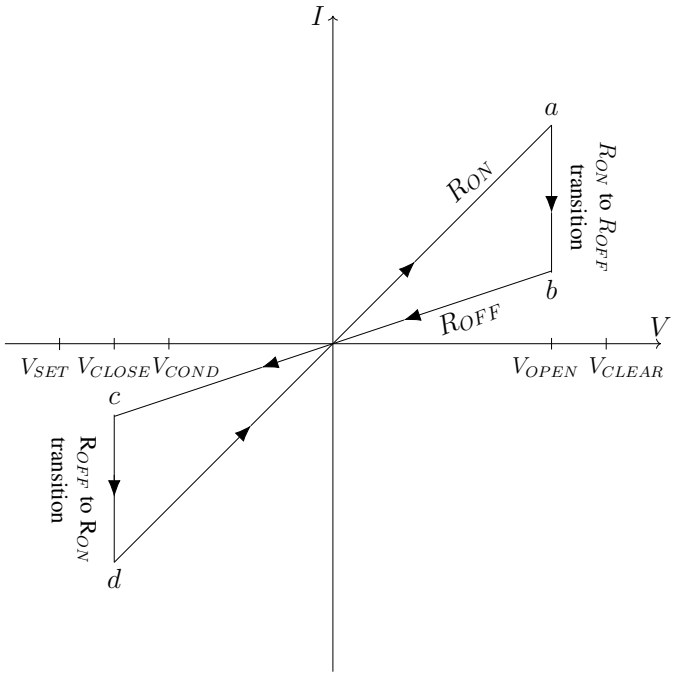


Figure 1.4: Demonstration of hysteresis in a hypothetical memristor model.

Fig. 1.4 shows VI plot of an ideal memristor. The hysteresis formed in this VI plot is characteristic to memristive devices [13]. In such VI plots, the line with the higher slope (**ad** in Fig. 1.4) roughly corresponds to high-resistance-state (HRS) and the line with the smaller slope (**bc** in Fig. 1.4) roughly corresponds to the low-resistance-state (LRS). On the V-axis, V_{CLOSE} and V_{OPEN} are the voltage values at which a memristor transitions from the high-to-low and low-to-high resis-

tance states respectively. V_{COND} is the voltage necessary for performing IMPLY operations using memristive circuits 2.4a. V_{CLEAR} and V_{SET} are the voltage values for a successful transition to the high-resistance and low-resistance states in a memristive circuit for computing IMPLY logic. Please notice that $|V_{CLEAR}| > |V_{OPEN}|$, $|V_{SET}| > |V_{CLOSE}|$, and $|V_{COND}| < |V_{CLOSE}|$.

In recent years, there is a considerable debate over the use of the term "memristor" for ReRAM devices [14]. Whether ReRAM devices can also be referred to as memristors is irrelevant to our work. Our work is applicable to all ReRAM devices as long as there is considerable difference between the the maximum and the minimum values of resistance of the device. For the rest of the document, we will use the terms memristor and ReRAM interchangeably.

In-Memory Computing

In-memory computing is also referred to as in-memory processing, computing-in-memory, logic-in-memory, or processing-in-memory. As the name suggests, in-memory computing intertwines storage and computing such that the stored information can be processed without retrieving it. Flow-based computing is a special type of in-memory computing which employs flow/sneak paths between crossbar nanowires for computing Boolean functions. Next, we describe *memristor crossbars*, *sneak paths*, and *in-memory computing using sneak paths*.

Memristor Crossbars

Nanoscale memristors are naturally assembled in the form of uniform two-dimensional arrays or crossbars. Memristive crossbars may be the architecture of choice for in-memory computing as nanoscale memristors can be packed together in a crossbar with high density. An $n \times m$ crossbar consists of n horizontal nanowires and m vertical nanowires. Each horizontal nanowire is con-

nected with all vertical nanowires through m distinct memristors. Similarly, each vertical nanowire is connected with all of the n horizontal nanowires through n different memristors. If a memristor is ON, the horizontal and the vertical nanowires connected to its terminal will be shorted; for an OFF memristor, the corresponding nanowires will not be connected. Fig. 1.5 shows a 4×4 crossbar. Crossbar nanowires are conductive metals and can be considered as nodes, while memristors are passive elements which can be programmed to desired resistance values. As a result, such crossbar circuit has $(n + m)$ nodes and nm two terminal memristors. As we increase the size of a crossbar, the number of nodes increases linearly, while the number of memristive connections between nanowires increases quadratically. At moderate and large scales, the number of memristive connections far outnumbers the number of nodes, thus giving rise to the phenomenon of sneak paths. Although, these sneak paths are problematic for storage applications, flow based in-memory computing employs these naturally abundant sneak paths for computing Boolean functions.

Sneak Paths in Crossbars

Thanks to passivity of memristor and homogeneity of crossbar structure, sneak paths occur naturally in memristor crossbars. Sneak paths are trails of low-resistance-paths between two nanowires which are not directly connected with each other through an ON memristor. Fig. 1.5 shows an example of a crossbar containing two sneak paths. In this crossbar, ON memristors are colored blue while OFF memristors are colored black. The memristor labeled as W is in OFF state, while memristors labeled as X, Y, and Z are in ON state. Please notice that an attempt to read the state of W via r_3 and c_2 will interpret it to be in the ON state. This erroneous read is caused by the low-resistance-path which appears parallel to W. The red line in Fig. 1.5 highlights this low-resistance-path, which passes through the ON memristors (X, Y, and Z). Such a low resistance path is called a sneak path. Similarly, the black line in Fig. 1.5 shows another sneak path between the bottom and the topmost nanowires, which are not directly connected with each other.

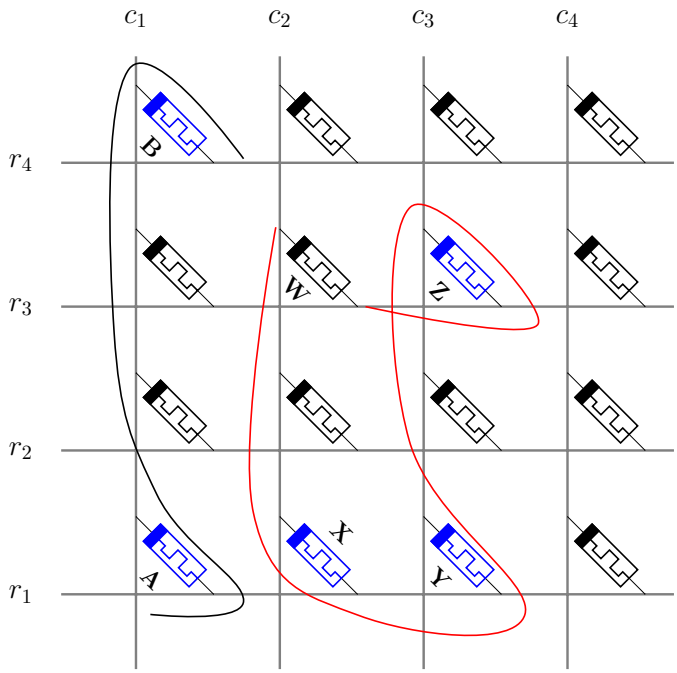


Figure 1.5: Depiction of two sneak paths in a 4×4 crossbar. Black and blue memristors are in the OFF and ON states respectively. Red line shows a sneak path between third row and second column. Although W is OFF, this sneak path, which passes through the memristors X,Y,and Z will make it appear to be ON. Black line shows a sneak path between bottom and the topmost nanowires.

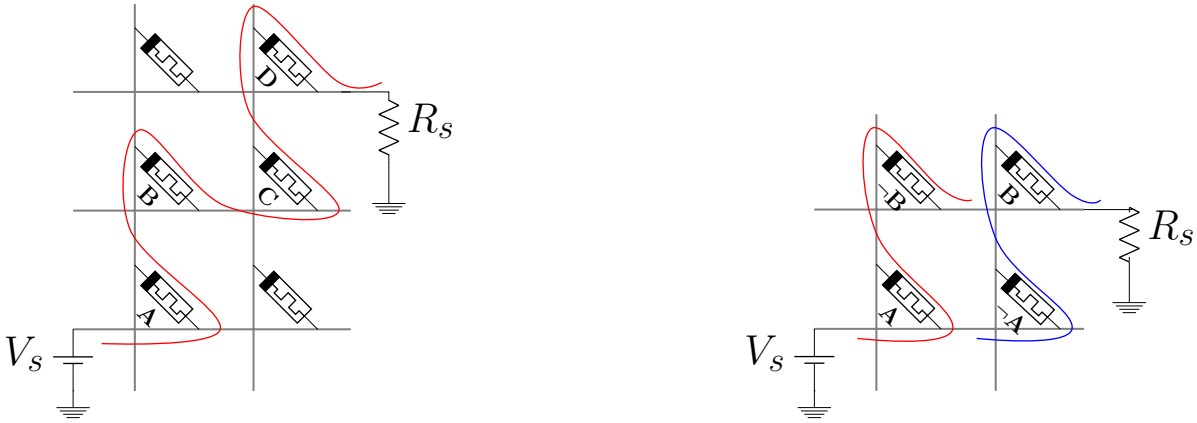
Although sneak paths are bane for memory applications of memristor crossbars [15], they are boon when crossbars are used for in-memory computing [16] [17]. If managed properly, sneak paths can be used for in-memory computing on crossbars. We have formally defined sneak path-based in-memory computing in the Chapter 3.

Flow-based In-Memory Computing

Memristors serve a dual purpose in flow/sneak path-based in-memory computing: storage (binary inputs are stored by configuring memristors to low-resistance (ON) or high-resistance (OFF) states) and computing (the resistance of memristors assists flow-based computing by controlling

the amount current flowing through the nanowires connected to their terminals). In flow-based computing of a Boolean formula on a crossbar, the crossbar memristors are configured such that there is a flow of current from an input nanowire to an output nanowire through the sneak paths in the crossbar if and only if the Boolean formula evaluates to $true$, and there is no such sneak path when the Boolean formula is $false$.

Figure 1.6a illustrates the concept of flow/sneak path-based in-memory computing for a simple example of 4-input AND gate on a 3×2 crossbar. Individual memristors are labeled as input literals (A, B, C, D), unlabeled memristors are always in non-conductive or high-resistance-state 0' and memristors labeled as '1' are always in conductive or low-resistance-state. If the first input A is $true$, the memristor labeled as A will be in conductive state allowing the current to flow from the bottom row to the first column, if the second input B is also true, the current will sneak into second row, similarly if C and D are also true, the current will eventually reach the topmost nanowire through the sneak path represented by the red line in figure 1.6a. Similarly, Fig. 1.6(b) shows a crossbar and its two sneak paths that implement a 2-input XOR gate.



(a) Crossbar for 4-bit AND

(b) Crossbar for 2-bit XOR

Figure 1.6: (a) Crossbar design for flow-based in-memory computing of a 4-input AND gate on a 3×2 crossbar. The sneak path is highlighting the flow of current from the bottom nanowire to topmost nanowire. (b) Design for in-memory computing of a 2-input XOR gate on a 2×2 crossbar.

After configuring the crossbar, a voltage source is applied at the bottom nanowire and current is sensed in the topmost nanowire. The flow of current in the topmost nanowire symbolizes that function is $true$, no flow means function is $false$.

Stochastic Computing

Transistor size, which has been decreasing exponentially over the last few decades, has reached atomic range. Now the transistor has already become so small that any further miniaturization comes with its own side effects such reduced noise margins, quantum tunneling, process variation, and radiation-induced single-event upsets (SEU) even at sea level. Such uncertain behavior has made fault-tolerant computing paradigms more relevant than ever. Stochastic computing is one such paradigm that is inherently error-tolerant due to its probabilistic nature.

Stochastic computing leverages laws of probability to perform computations. It performs computations on input Bernoulli stream/s and generates output Bernoulli stream/s. A Bernoulli stream is a binary stream where each bit is independently and identically distributed (iid) according to some distribution. The relationship between the probability of '1' in the input and output streams characterizes the computation performed. The circuit performing stochastic operations is also called a Stochastic Circuit (SC) or a Stochastic Computational Element (SCE).

Stochastic circuits have simpler and more compact realizations than their deterministic counterparts. Elementary operations such as multiplication and addition can be implemented stochastically using a few logic gates [18]. For example, scaled addition can be implemented using an OR gate and multiplication, which is one of the most important operations using a modern computer, can be performed using a single AND gate. Due to their simplicity and small size, stochastic circuits implementing such operations are energy-efficient and massively parallelizable [19].

By their probabilistic nature, stochastic circuits have some degree of uncertainty associated with them. Therefore, stochastic computing is more suitable for error-tolerant applications such as multimedia, machine learning and pattern recognition, data mining, etc. These applications happen to be among the most computation-hungry tasks for conventional computing machines. Fortunately, these applications require a large number of simple arithmetic operations, which have compact and parallelizable realizations in stochastic computing. The precision of stochastic circuits can be increased by increasing the length of stochastic streams. Conversely, shorter stochastic streams decrease both the accuracy and the computational delay. This allows us to reduce power consumption and computational delay by sacrificing some accuracy without the need to change underlying hardware.

Stochastic computing is more robust than conventional computing due to high error tolerance [20]. Error tolerance of stochastic circuits comes from the fact that each bit in a stochastic stream has the same significance irrespective of its position in the stream. Thus, the effects of single bit errors in stochastic circuits are not as serious as in deterministic circuits, where an error in the most significant bit may alter the output significantly. Moreover, the error tolerance of stochastic circuits can be increased by employing longer streams. For example, doubling the length of a stochastic stream decreases the significance of individual bits by the same factor. Stochastic circuits also lend themselves to much higher clocks than conventional circuits due to small circuit delay [19]. Recently, stochastic computing has been used for image scaling and thresholding [21] [22].

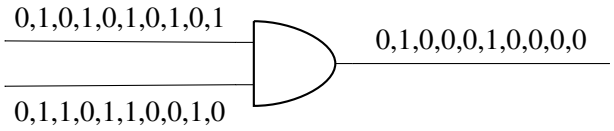


Figure 1.7: Two-input AND gate can act as a stochastic multiplier. Probability of '1' in the output stochastic stream is approximately equal to the product of probabilities of '1' in input stochastic streams. I.e $p_{out}(1) \approx p_1(1) \times p_2(1)$.

Input and Outputs for Stochastic Circuits

Inputs and outputs of a stochastic circuit are comprised of stochastic streams of binary numbers (0 or 1). Given a stochastic stream S_x, it represents a stochastic number which is equal to the probability of '1' in it. For example, if the stochastic stream S_x contains N_1 1's and N_0 0's, then S_x represents the number $x = \frac{N_1}{N_0+N_1}$, which is same as probability of 1 in S_x. Since stochastic numbers are also a probabilistic representation, they fall in the $[0, 1]$ range. Thus, real-world inputs must be scaled to non-negative numbers less than 1 before they can be converted into stochastic streams.

When real-world input is converted into a stochastic stream, the transformation is such that there is a linear mapping between input value and probability/relative frequency of '1' in the corresponding Bernoulli stream. Brown and Card have discussed the generation of such sequences using linear feedback shift registers (LFSR) or cellular automata (CA) [19]. Output of an SCE is a Bernoulli stream of logical ones and zeros. An integrator used at the output of an SCE will convert output Bernoulli stream into corresponding probability value. A counter can also achieve the same purpose with the additional advantage that the stream is converted into the equivalent binary representation.

The order of 1's and 0's in a stochastic stream has no significance. For example, 00011100 and 01010100 represent the same number, i-e $\frac{3}{8}$. As described earlier, uniform significance of individual bits in stochastic streams makes stochastic computing error tolerant. However, this increased robustness comes at the cost of decreased representational power. For example, a stochastic stream of length N can encode only N+1 stochastic numbers. Thus, to increase the precision of stochastic circuits, input length needs to be increased linearly, while for conventional circuits, input length needs to be increased only logarithmically.

Simple elementary arithmetic computations (such as multiplication, scaled addition) can be performed stochastically using very small combinational circuits comprised of logic gates or multiplexers [20]. More complex computations such as 'function approximation' can also be performed easily using simple sequential circuits modeled by Finite State Machine (FSM). In Chapter 4, we have used decision procedures to synthesize FSMs for stochastic computation for more complex functions such as polynomials, exponentials and tanh.

CHAPTER 2: LITERATURE REVIEW

The inability of the conventional computer to efficiently process modern computing requirements driven by machine learning, artificial intelligence, data mining, and multimedia has renewed interest in unconventional computing systems in recent years. In-memory computing and stochastic computing are two such paradigms of interest. By their nature, these applications require a large number of data-retrieval and data-storage operations. Therefore, they are particularly adversely affected by the Memory Wall, which is caused by limited bus speed of the von Neumann architecture. In-memory computing employs recent nanoelectronic ReRAM devices to bring storage and processing to physically same location, thus eliminating the problem of the Memory Wall. Stochastic computing performs computations by leveraging the laws of probability. Stochastic computing offers two major advantages over conventional computing. First, stochastic circuits are naturally more error-resilient than their conventional counterparts. This is especially significant due to decreasing noise margins in ever shrinking integrated circuits. Secondly, stochastic circuits are simpler and more compact than their exact implementations, resulting in better area and power efficiency. The simplicity of stochastic circuits also makes them easier to parallelize.

In this chapter, we briefly discuss the following seminal works on unconventional computing.

- In-Memory Computing
  - Matrix Multiplication
  - Memristor Ratioed Logic (MRL)
  - Memristor-Aided Logic (MAGIC)
  - IMPLY Family
    - IMPLY-based NAND

    - * IMPLY-based 2-to-1 MUX
  - Majority-Inverter Graph (MIG)
    - * MIG-IMPLY
    - * MIG-MAJ
  - Flow/Sneak Path based Computing
- Stochastic Computing

Matrix Multiplication

K.T. Hung introduced systolic architectures in 1979 to implement concurrent computations for application specific VLSI algorithms [23]. Systolic arrays have been used to speed up matrix multiplication. A memristor crossbar can be classified as a 2D systolic architecture, where each memristor can possibly serve as a simple processing unit. Fig. 2.1 shows a schematic diagram of a 1T1M crossbar for vector-matrix multiplication [24]. In this crossbar, a transistor and a memristor is placed on each intersection of horizontal and vertical nanowires as shown in Fig. 2.1.

Let's assume that we want to multiply an $n \times 1$ vector $\mathbf{v}$ with an $n \times n$ matrix $\mathbf{M}$ using an $n \times n$ crossbar as shown in Fig. 2.1. Let m_{ij} represent the entry in the ith row and the jth column of the matrix $\mathbf{M}$. Similarly, let r_{ij} represent the resistance of the memristor connected between the ith row and the jth column of the crossbar. First, the matrix $\mathbf{M}$ is loaded onto the crossbar, such that for entry $m_{ij} \in \mathbf{M}$ the conductance of the corresponding memristor (between the ith row and the jth column) is $g_{ij} = \frac{1}{r_{ij}} = m_{ij}$. After the crossbar is configured according to the matrix $\mathbf{M}$, the input vector $\mathbf{v}$ is applied to the horizontal nanowires such that the voltage applied to the ith row is equal to the ith entry of $\mathbf{v}$ as shown in Fig. 2.1. To read outputs, each column of the crossbar is connected with a separate op-amp based analog adder as shown in Fig. 2.1. This adder

accumulates $v_i g_{ij}$ for all the memristors connected with the jth column, which is same as the dot product of the vector $\mathbf{v}$ and the conductance vector $\mathbf{g_j}$ for jth column. Formally, the output of jth adder, y_j, can be represented by the following equation,

$$y_j = -r_o \sum_{i=1}^{n} \frac{v_i}{r_{ij}} = -r_o \sum_{i=1}^{n} v_i g_{ij} = -r_o \mathbf{v^T g_j}, \tag{2.1}$$

where v_i is the voltage of the ith row, r_{ij} and g_{ij} are the resistance and conductance values of the memristor between the ith row and jth column, and r_o is the feedback resistance of the op-amps in the adder circuits.

Once the matrix $\mathbf{M}$ is loaded onto the crossbar and the input vector $\mathbf{v}$ is applied to the horizontal nanowires, each entry of the output vector $\mathbf{y} = [y_1, y_2, \ldots, y_n]$ is computed simultaneously as shown in Fig. 2.1. The output $\mathbf{y}$ is given by

$$\mathbf{y}^T = -r_o \mathbf{v}^T \mathbf{G} = -r_o \mathbf{v}^T \mathbf{M}, \tag{2.2}$$

here, $\mathbf{v}$ is the vector representing voltages applied at the horizontal/input nanowires. $\mathbf{G}$ is the matrix representing conductances of the memristors in the crossbar. The negative sign in equation 2.2 appears because the adders in Fig.2.1 use amplifiers in inverting configuration.

On a single processor, the complexity of vector-matrix multiplication is $O(n^2)$, due to n^2 multiplications and $n^2 - n$ additions. When a crossbar is used to compute vector-matrix product, these multiplications and additions are performed simultaneously on analog circuits, thus reducing the overall computational complexity of vector-matrix multiplication from $O(n^2)$ to $O(1)$. Matrix-matrix multiplication requires n cycles of the vector-matrix multiplication, thus the complexity of matrix-matrix multiplication is $O(n)$.

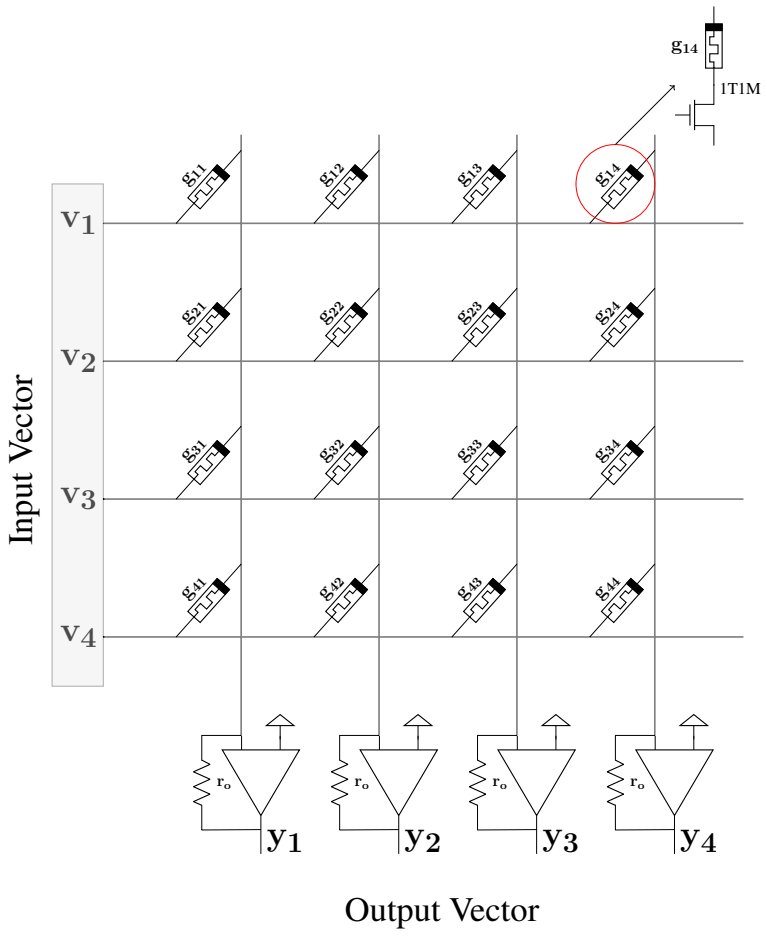


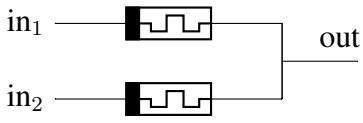
Figure 2.1: Schematic diagram of vector-matrix multiplication on a crossbar.

Matrix multiplication has widespread applicability in machine learning, multimedia, pattern recognition, etc. Velasquez et al. have used crossbars with rectifying memristors (1 diode, 1 memristor) for multiplication of Boolean matrices [25]. Hu et al. have proposed a conversion algorithm to map the matrix values as conductance on real memristors in a crossbar [24]. Shafiee et al. have used crossbar based matrix multiplication for designing neural network accelerator [26]. Zhang et al. have transformed matrices to minimize the effects of stuck-at faults in memristor crossbars [27].

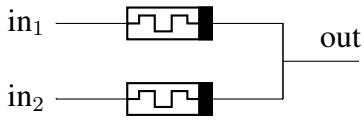
Memristor Ratioed Logic (MRL)

Kvatinsky et al. proposed Memristor Ratioed Logic (MRL) to compute logic using memristive circuits [28]. MRL circuits are hybrid of memristive and CMOS components. MRL employs polarity of memristive devices for computing elementary logic functions (AND, OR). A CMOS-inverter is connected at the output of AND and OR, making them NAND and NOR gates, which can be used to implement any Boolean functions.

The resistance of a memristor decreases when the current flows in one direction (into the device), while it increases when the current flows in the opposite direction (out of the device). Fig. 2.2a and 2.2b show arrangements of memristors for computing two-input AND and OR gates respectively. As shown in Fig. 2.2, the common terminal of the memristors serves as output node, while the other terminals serve as input nodes.



(a) Realization of 2-input OR using MRL.



(b) Realization of 2-input AND using MRL.

Figure 2.2: MRL realizations of AND and OR gates.

Let's first consider the OR-gate shown in Fig. 2.2a. Let V_{High} and V_{Gnd} denote the input voltages corresponding to logic 1 ($true$) and 0 ($false$) respectively. In Fig. 2.2a, when both inputs are either $true$ or $false$, the output will be the same as inputs, because $V_{in_1} = V_{in_2} = V_{out}$. When $V_{in_1} = V_{High}$ and $V_{in_2} = V_{Gnd}$, the current will flow *into* the upper memristor and decrease its resistance. At the same time, the resistance of the lower memristor will increase because the same amount of current will flow out of the lower memristor and reach the ground connected at V_{in_2}. If the input voltages are applied for a sufficient duration, the resistance of the upper

memristor will saturate at R_{ON}, while the resistance of the lower memristor will saturate at R_{OFF}. If $R_{OFF} >> R_{ON}$, the voltage at the output terminal will be $V_{out} = \frac{R_{OFF}}{R_{OFF}+R_{ON}} V_{High} \approx V_{High}$. If the inputs are reversed ($V_{in_1} = V_{Gnd}$ and $V_{in_2} = V_{High}$), the same reasoning will again result in $V_{out} \approx V_{High}$.

For AND gate, the polarity of the memristors is reversed as shown in Fig. 2.2b. Here too, when both inputs are equal, the same voltage will appear at the output terminal because $V_{out} = V_{in_1} = V_{in_2}$. However, when $V_{in_1} \neq V_{in_2}$, the flow of current will be such that the memristor connected with V_{High} saturates to R_{OFF} and the memristor connected with V_{Gnd} saturates to R_{ON}. If $R_{ON} << R_{OFF}$, the output voltage will be $V_{out} = \frac{R_{ON}}{R_{OFF}+R_{ON}} V_{High} \approx 0$, which is in accordance with AND gate.

Memristor-Aided Logic (MAGIC)

Kvatinsky et al. proposed Memristor-Aided Logic (MAGIC) to compute logic using memristors [3]. MAGIC uses resistance to represent logical state, therefore inputs, outputs, and intermediate results are in the form of resistance as opposed to voltage. Logic 0 (Low) is represented by an OFF memristor with R_{OFF} resistance and logic 1 (High) is represented by an ON memristor with R_{ON} resistance.

MAGIC computation is a two step process. During the initialization step, input memristors are configured to the ON or OFF states depending upon the inputs, while the out-memristor is initialized to a default state. During the computation step, the voltage V_s is applied as shown in Fig. 2.3. At the end of computation, the state of the out-memristor is configured to reflect the outcome of computation. For example, Fig. 2.3a shows a circuit for MAGIC implementation of a 2-input NOR function. When both inputs in_1 and in_2 are 0, the corresponding memristors (in_1 and in_2) will be

OFF, and there won't be sufficient current to switch the out-memristor to the OFF-state. When at least one of the input memristors is ON due to $true$ input, sufficient current will flow out of the out-memristor to switch its state from ON to OFF, which is in accordance with the functionality of the NOR gate. For MAGIC NAND shown in Fig. 2.3b, sufficient current will be available to switch the state of the out-memristor only when both of input memristors are in the ON state. MAGIC achieves inversion by reversing the polarity of the out-memristor. Fig. 2.3d and 2.3e show MAGIC circuits for OR and AND gates respectively. It's important to mention that the out-memristor is initialized to logic 1 (ON) for NAND and NOR gates, while the initial state is logic 0 (OFF) for MAGIC AND and OR gate.

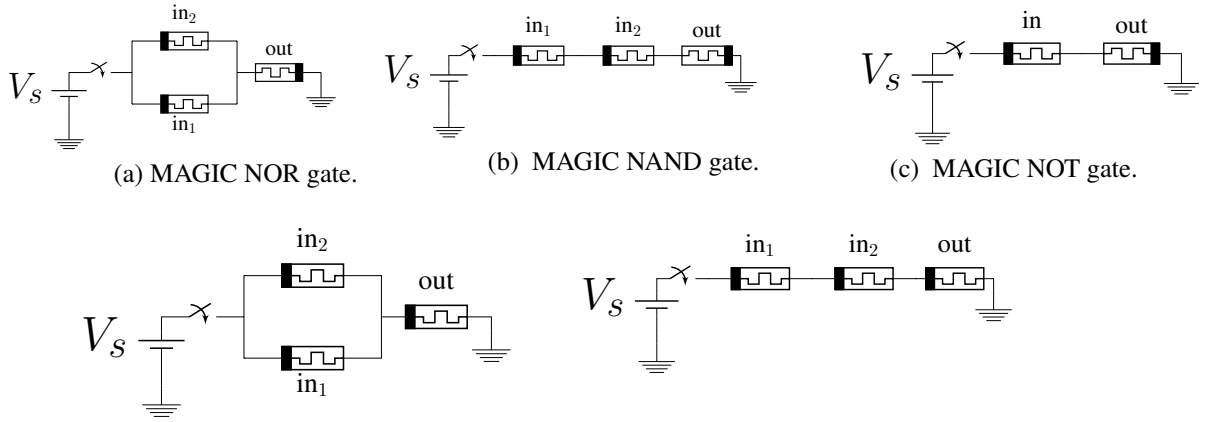
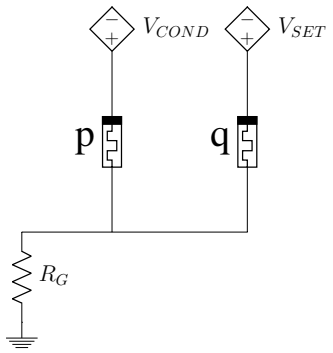


Figure 2.3: MAGIC gates from Kvatinsky et al. [3].

IMPLY Family

Material implication (IMPLY) is a natural logical operation for purely memristive circuits [29]. Fig. 2.4a shows a circuit for computing material implication using two memristors. This circuit computes the function $f_{IMPLY}(p, q) = p \rightarrow q = \neg p + q$. The truth table for this function is given

in Fig. 2.4b. Such computational setup is also known as stateful logic, because the memristors not only act as computational units, they also store inputs, intermediate results, and outputs as resistance.



(a) IMPLY circuit using memristors [30].

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

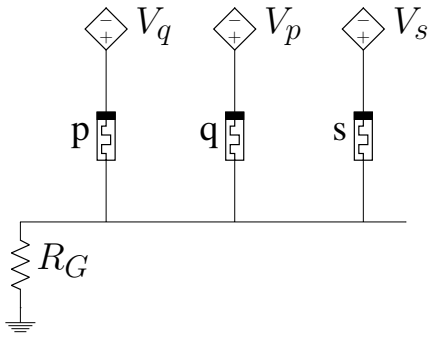
(b) Truth table for $p \rightarrow q$.

Figure 2.4: IMPLY gate.

Figure 2.4a shows a circuit for computing $p \rightarrow q$. This circuit consists of two memristors (p-memristor and q-memristor) and one resistor (R_G). V_{COND} is applied to the p-memristors and V_{SET} is applied to the q-memristor. The resistor R_G controls the eventual voltage across the p-memristor and q-memristor as we explain later. Let I_p, I_q and I_g represent the amounts of current flowing through the p-memristor, q-memristor, and R_G respectively. In the initialization phase, the states of the p and q memristors are configured to reflect the corresponding inputs (p and q). After the computation phase is complete, the state of the q-memristor is configured according to $p \rightarrow q$. Sometimes the p-memristor is referred to as the input memristor and the q-memristor is referred to as the work memristor. In complex memristive circuits, individual IMPLY operations are also referred to as micro-operations.

When $p = 0$, the p-memristor will be in the high-resistance-state, thus making I_p smaller. Consequently, the voltage across R_G, $V_G = I_g R_G = (I_p + I_q)R_G$, will also be smaller. By design, the value of R_G is chosen such that the voltage across the q-memristor $V_q = V_{SET} - V_G$ is larger

than V_{CLOSE} whenever the p-memristor is OFF. Therefore, irrespective of the initial state of the q-memristor, its final state will be ON whenever the p-memristor is OFF. This corresponds to the first two entries of the truth table in Fig. 2.4b. When $p = 1$, the p-memristor will be in the low-resistance-state, resulting in higher I_p. Therefore, the voltage across R_G, $V_G = (I_p + I_q)R_G$, will also be larger. Since $V_q = V_{SET} - V_G$, the larger value of V_G will result in smaller voltage across the q-memristor such that $V_q < V_{CLOSE}$. Thus, the q-memristor will retain its state whenever $p = 1$. This corresponds to the last two entries of the truth table in Fig. 2.4b.



(a) IMPLY-based NAND gate.

Initialization:
p = p, q = q, s = 0
Computation:
Op1 : $p \rightarrow s$
Op2: $q \rightarrow s$

(b) Micro-operations for a NAND gate.

Figure 2.5: IMPLY-based NAND gate.

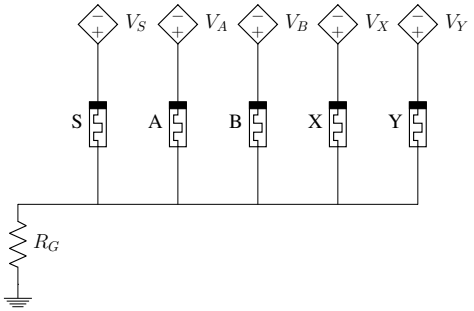
Memristive NAND

Memristive IMPLY and $false$ constitute a functionally complete set; they can compute any Boolean function. Two imply operations are sufficient to compute a two-input NAND function, NAND(p,q) = $q \rightarrow (p \rightarrow 0) = \neg p + \neg q$. Fig. 2.5 shows a memristive circuit for a NAND gate, which is also a universal gate. This circuit has three memristors p, q, and s and one resistor R_G. It computes the NAND function by performing two micro-operations: $p \rightarrow 0 = \neg p$ and $q \rightarrow \neg p = \neg p + \neg q$. For the first micro-operation, $p \rightarrow s$, the p and s memristors are initialized to p and 0 respectively. At the end of computation, the s-memristor contains the value $s = \neg p$. This value of s is used in the

second micro-operation $q \rightarrow s$. At the end of the second micro-operation, the state of s reflects the result of $\neg p + \neg q$, which is NAND of p and q.

IMPLY-based 2-to-1 Multiplexer

Multiplexer (MUX) is widely used in logic synthesis. Because of its universality and application in large scale synthesis, its memristive realizations have been explored in several works [31], [32]. Chakraborti et al. have compared its three memristive implementations differing in number of memristors and micro-operations. Fig. 2.6a shows their implementation of a 2-to-1 MUX, which requires 5 memristors and 6 micro-operations (including initialization). In this figure, the memristors A and B represent the input lines of the MUX, S is the memristor for the select line, and X and Y are the work memristors. Fig.2.6b shows the sequence of six micro-operations for computing the functionality of the 2-to-1 MUX. At the end of computation, the result is stored in the Y-memristor.



(a) IMPLY-based 2-to-1 MUX.

Op1: S= s, A = a, B = b,
X= 0, Y = 0
Op2 : $s \rightarrow x$
Op3: $b \rightarrow x$
Op4: $x \rightarrow y$
Op5 : $a \rightarrow s$
Op6 : $s \rightarrow y$

(b) Micro-operations for the 2-to-1 MUX in (a).

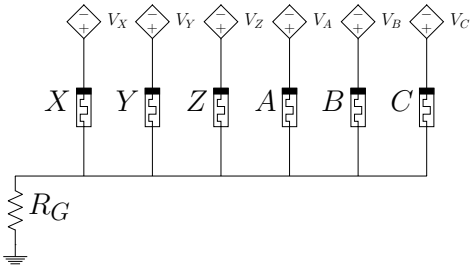
Figure 2.6: IMPLY-based 2-to-1 MUX from Chakraborti et al. [4].

Majority-Inverter Graph (MIG)

Majority-Inverter-Graph (MIG) is a graphical data structure for optimizing circuits of Boolean functions [33]. Each MIG node has three inputs and it implements Boolean majority. The functionality of an MIG node can be described as $M(x, y, z) = xy + yz + zx$, where x, y, and z are three inputs. By fixing one of the inputs of an MIG-node to 1 or 0, its functionality can be reduced to a 2-input OR or AND gate respectively. For example, $M(x, y, 0) = xy$ and $M(x, y, 1) = x + y$. Because of greater representational power of an MIG node, MIGs can be more compact than the And-Inverter Graphs (AIG) and OR-Inverter Graphs (OIG). MIGs based optimizations are especially useful for optimizing the depth/delay. Next, we described two approaches proposed by Shirinzadeh for in-memory computing on MIGs [5].

IMPLY-based Majority-Inverter Graph (MIG-IMP)

Shirinzadeh et al. have proposed an IMPLY-based realization of the majority function [5]. Fig. 2.7 shows their memristive realization for an MIG node. This circuit needs 6 memristors and one resistor R_G. The memristors labeled as X, Y, and Z are the input memristors, while A, B, and C represent the work memristors which are used for storing or reusing the intermediate results of micro-operations. Fig. 2.7b shows the sequence of 10 mirco-operations for computing the majority function. They have replaced each node of the MIG with its memristive realization. Additionally, an extra memristor performing $p \rightarrow 0$ can account for the complemented output of a majority node.



(a) Circuit for one MIG-IMP node in [5].

Op1: X = x, Y = y, Z = z, A=B=C = 0	Op6: $y \rightarrow c$
Op2: $x \rightarrow a$	Op7: $z \rightarrow c$
Op3: $y \rightarrow b$	Op8: $a = 0$
Op4: $a \rightarrow y$	Op9: $b \rightarrow a$
Op5: $x \rightarrow b$	Op10: $c \rightarrow a$

(b) Micro-ops. for an MIG-IMP node.

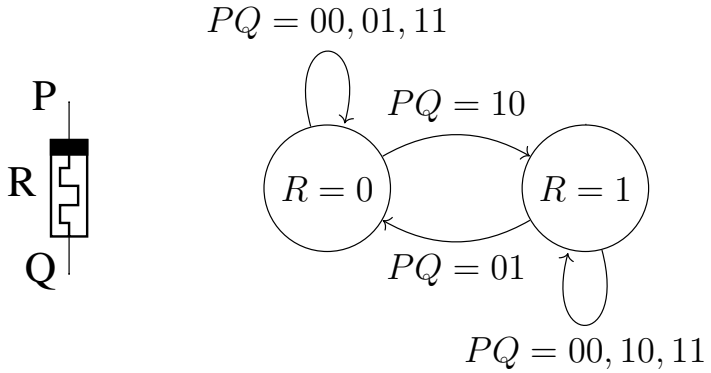
Figure 2.7: IMPLY-based implementation of the majority node in [5].

MIG-MAJ

Shirinzadeh et al. have shown how a memristor can be used to compute the majority operation [5]. Let R represent the current state of the memristor R, such that low-resistance-state is represented by 1 and high-resistance-state is represented by 0. Let P and Q represent logical inputs applied to the P and Q terminals of the memristor, such that P = 1, Q = 0 corresponds to V_{SET}, P = 0, Q = 1 corresponds to V_{CLEAR}, and P = Q corresponds to V_{COND}. Let R_{Next} represent the state after the application of P and Q across its terminals, then the resulting state diagram and state transition tables are shown in Fig. 2.8. The relationship between the current state R and next state R_{Next} of the memristor can be represented as follows:

$$R_{Next} = (P\neg Q)\neg R + (P + \neg Q)R = MAJ(P, \neg Q, R). \tag{2.3}$$

Thus the state transition of a memristor happens according to the majority operation of P, $\neg Q$, and R.



(a) State diagram for realizing MAJ operation with one memristor.

P	Q	R	R_{Next}	P	Q	R	R_{Next}
0	0	0	0	0	0	1	1
0	1	0	0	0	1	1	0
1	0	0	1	1	0	1	1
1	1	0	0	1	1	1	1

(b) State transition table for MAJ operation.

Figure 2.8: Memristive MAJ node and its state transition table [5].

IMPLY-based Binary Decision Diagram

Chakraborti et al. have used Binary Decision Diagrams (BDD) to design purely memristive circuits for Boolean functions [4]. A BDD is a graphical representation of Boolean functions, consisting of one root node, several intermediate nodes, and two terminal nodes. A BDD node can be realized by a 2-to-1 multiplexer (MUX). Fig. 2.6 shows their implementation of a 2-to-1 MUX, which requires 5 memristors and 6 micro-operations shown in Fig. 2.6b. They synthesize an reduced ordered BDD (ROBDD) for the target function and replace each node of the ROBDD with their memristive MUX. For example, an ROBDD for the 4-bit function $f = x_1 \neg x_2 + x_3 x_4$ has 4 nodes.

This function is computed in 24 micro-operations and requires 20 memristors [4].

Flow-based In-Memory Computing on Memristor Crossbars

Sneak paths are naturally pervasive in memristor crossbars. A sneak path is a trail of memristors in low-resistance-state between two crossbar nanowires that are not directly connected through an ON memristor. Flow-based computing relies on these sneak paths for in-memory computing. During computation, the individual memristors of a crossbar are configured such that a sneak path exists between the bottom and the topmost nanowires only when the target function is $true$, and no sneak path exists between them when the target function is $false$. The existence of a sneak path is verified by applying a small voltage at the bottom nanowire and measuring how much current is flowing out of the top nanowire. The flow of significant amount of current in the topmost nanowire symbolizes that the function is $true$, no flow means the function is $false$. Figs. 1.6a and 1.6b illustrate this concept for a 4-input AND gate and 2-input XOR gate respectively.

Our work is on flow-based in-memory computing on memristive crossbar. Next, we explain previous works on the synthesis of crossbars for flow based in-memory computing.

Mapping Negation Normal Form on Crossbars

Any Boolean formula can be represented in negation normal form (NNF). An NNF representation of a Boolean formula contains only three operators: conjunction, disjunction, and negation. Additionally, the negation operator can be applied only to atomic variables; it cannot be applied to compound expression. Velasquez et al. have proposed two simple rules for structural implementation of conjunction and disjunction operations on crossbars [16], while the negated variables are directly mapped onto crossbars memristors. They have shown that the repetitive use of these

two simple rules can synthesize a crossbar circuit for any Boolean formula. Although it is computationally inexpensive to map an NNF on a crossbar, the downside of their approach is that the synthesized crossbars can become exponentially large. This is because the NNF of a function can itself have an exponentially large representation.

Formal Methods for Flow-based In-Memory Computing

Velasquez et al. have used formal methods to synthesize crossbars for in-memory computing Boolean formulae [17]. The use of decision procedures has resulted in compact crossbars for 1-bit adder. But the problem with this approach is its computational requirements. Despite the compactness of synthesized crossbars, it is not scalable for large or moderately sized functions because of its exponential computational complexity in terms of the number of memristors in a crossbar.

Binary Decision Diagrams for Flow-based In-Memory Computing

A Binary Decision Diagram (BDD) can be used for graphical representation of a Boolean function f. A BDD is a directed acyclic graph (DAG) with one root node, two terminal nodes (0,1), and possibly several non-terminal/intermediate nodes. Whenever the function f is $true$ for some input, a path will connect the root node of its BDD with the terminal-1. This is similar to flow-paths in flow-based computing on crossbars. In flow-based computing, a sneak path exists between the bottom and the topmost nanowires of a crossbar whenever the function f is $true$. This similitude of computing paths in crossbars and BDD makes the latter ideal for flow-based synthesis of the former. Hassen and Dwaipayan et al. have used reduced ordered BDDs (ROBDDs) for synthesizing crossbars for flow-based in-memory computing [6] [34].

Stochastic Computing

Stochastic computing has been around since the 1950s. Von Neumann introduced stochastic computing in his pioneering paper "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components" [35]. Shannon et al. described how to build high reliability circuits from arbitrarily unreliable relays [36]. They studied different redundancy schemes for improving the overall reliability of the circuits built from unreliable components. In 1967, Gaines defined stochastic computational element (SCE) as a simple circuit which performs computations on input Bernoulli stream(s) and generates an output Bernoulli stream [18]. He has discussed single-line unipolar, single-line bipolar, and two-line bipolar representations of signals for stochastic streams. He described SCEs for multiplication, scaled addition, squaring, square root, inversion, integrator, and divider [18]. Poppelbauem et al. have also explored essentially the same ideas for computing systems based on random pulse sequences (RPS). They have discussed schemes for generating RPS by thresholding noise diodes. A clock signal and D-flip flops suffice to convert an asynchronous RPS into a synchronous RPS [37].

After initial investigations of stochastic computing in the 1950s and 1960s, the interest in stochastic computing waned due to success of the digital computer built using integrated circuits. In the recent years, stochastic computing is gaining attention again due to efficient hardware implementations of artificial spiking neural networks and stochastic decoder for error control codes, [38, 39]. Furthermore, the reliability issues associated with the miniaturization of integrated circuits and the shortcomings of the von Neumann architecture in the post 2000 years have increased the significance stochastic computing. Brown et al. gave a comprehensive overview of several SCEs for their potential application in pulsed neural networks [19]. They have also discussed the conversion of digital inputs into stochastic streams and vice versa. An integrator used at the output of an SCE can convert the output Bernoulli stream into corresponding probability value. They have also pre-

sented implementations of Gaines' functions, addition, multiplication, division, squaring, etc, for stochastic streams with unipolar and bipolar symbols [18].

Qian et al. have employed the Bernstein polynomials for computing polynomial functions stochastically [40]. Their approach can synthesize stochastic circuits for polynomials that have Bernstein coefficients in the unit range. Alaghi et al. have used spectral transformation of Boolean functions for designing stochastic circuits [41]. They represent a Boolean function in the form of a multilinear polynomial before taking its inverse Fourier transform. Their approach works for arbitrary polynomials as well, provided the coefficients of the inverse Fourier transform of the polynomial are in the range [-1, 1].

State machines are used for designing stochastic circuits for more complex tasks. Gaines et al. used sequential circuits for stochastic computing [18]. They proposed ADaptive DIgital Element (ADDIE), which is a saturating counter. It cannot be incremented beyond a maximum value or decremented below a minimum value. Brown et al. proposed finite state machines (FSM) for synthesizing stochastic circuits for tanh, linear-gain, and exponential functions. Li et al. synthesized finite state machines by optimizing the square loss function for tanh, polynomial, and exponentiation functions [7]. We have synthesized these functions using decision procedures. Our approach reduced the worst-case error by up to 65 percent.

CHAPTER 3: FBDD-BASED SYNTHESIS OF CROSSBARS FOR FLOW BASED COMPUTING [1]

Introduction

John von Neumann's "First Draft" defining a computer architecture for the EDVAC system [8] has survived for seven decades due to an exponential decrease in feature sizes over this period. The slowdown of Moore's law, the end of Dennard scaling and the rise of big data have led to a renewed interest in More-than-Moore devices [42] and novel computer architectures [43], including in-memory computing systems [44]. The ability to compute without moving data across the von Neumann barrier between the processor and the memory reduces both the energy and the time needed to perform the computations.

A two-dimensional crossbar of nanoscale memristors forms a desirable fabric for in-memory computing as memristors can serve as non-volatile storage devices and the values stored in the memristors can control the flow of current through sneak paths in the nanoscale crossbar. We can perform arbitrary Boolean computations on a nanoscale crossbar using the flow of current through sneak paths in the crossbar [17, 34, 45, 46]. The critical step in this design process is the mapping of memristors in a crossbar to the variables in the Boolean formula being computed.

Reduced Ordered Binary Decision Diagrams (ROBDDs) have been successfully used to design nanoscale memristor crossbars capable of implementing Boolean formulae using flow-based computing [6, 34]. However, there exist Boolean formulae such that the size of their most succinct ROBDD representations with the best variable ordering is exponential in the number of variables.

[1] Related Publication: A. Ul Hassen, D. Chakraborty and S. K. Jha, "Free Binary Decision Diagram-Based Synthesis of Compact Crossbars for In-Memory Computing", in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 5, pp. 622-626, May 2018.

In this chapter, we cover the following:

1. How a bipartite variant of a Free Binary Decision Diagram (FBDD) can be used to synthesize a nanoscale crossbar that implements flow-based computing for a given Boolean formula.
2. The efficacy of FBDDs by synthesizing a nanoscale memristor crossbar for the middle bit of a 4-bit multiplier that takes 69.9% less area than a crossbar designed using ROBDDs [6].

A 4-bit multiplier designed using FBDD-based approach needs 8.4% less area than the ROBDD-based approach [6], while a multiplier designed using the best of both approaches needs 42.8% less area than the approach based only on ROBDDs.

Flow-based Computing using Sneak Paths

A nanoscale memristor crossbar of n rows and m columns has nm memristors. The plurality of memristive connections among the horizontal and vertical nanowires of a crossbar gives rise to the phenomenon of sneak paths [47]. Sneak paths are trails of low resistance paths between two nanowires which are not directly connected with each other through an ON memristor. The probability of sneak-path-based disturbance increases exponentially with the length of the sneak path [15].

Flow-based in-memory computing leverages the abundance of sneak paths in nanoscale memristive crossbars for implementing Boolean functions. A memristive crossbar design creates a one-to-one correspondence between the value of the Boolean function and the existence of a sneak path between the bottom and the topmost nanowires of the crossbar.

Definition 3 (Crossbar Designs for Boolean Formula)**.** *Let $f : \{0,1\}^k \rightarrow \{0,1\}$ be a k-bit Boolean function over variables $v_1, v_2 \ldots v_k$ and $D : R \rightarrow \{v_1, v_2, \ldots v_k\}$ be the design of the crossbar*

mapping memristors $R = \{r_{11}, r_{12} \ldots r_{1n}, r_{21}, \ldots, r_{mn}\}$ to the values of the variables V. A crossbar design D is said to implement the Boolean formula f if and only if the following two conditions hold:

- *There exists a flow of current or a sneak path from the bottom nanowire to the topmost nanowire of the crossbar design D for a valuation of variables V if the Boolean formula f evaluates to true for the given valuation of the variables V.*
- *There is no sneak path connecting the bottom nanowire to the topmost nanowire of the crossbar design D for a valuation of variables V if the Boolean formula f evaluates to false for this valuation.*

The presence of a sneak path between the bottom and the topmost nanowire may be verified by applying a small voltage at the bottom nanowire and detecting the flow of current through the topmost nanowire. The flow of current in the topmost nanowire symbolizes that function is *true* while the absence of a significant flow of current implies that the function is *false*.

Binary Decision Diagrams

Binary decision diagrams (BDDs) are a natural choice for designing nanoscale memristive crossbars that implement flow-based computing using sneak paths. BDDs are compact structural representations of Boolean functions. Lee was the first to use them for representing switching circuits in 1959 [48]. Akers did a comprehensive study of binary decision diagrams in 1978 [49].

Let $f(x)$ be a k-bit function on the variable set $V = \{v_1, v_2, v_3 .... v_k\}$. The BDD representation for the function is a directed acyclic graph with one root node, two terminal nodes and possibly multiple intermediate nodes. All nodes except the terminal nodes have two outgoing edges. All

non-terminal nodes of BDDs are labeled by a variable $v_i \in V$, terminal nodes are labeled as 0 or 1. Each non-terminal node is connected to either of its children depending on the value of the variable v_i. Each node of a BDD represents a Boolean function, the root node represents the original function $f(x)$, the terminal node 1 represents *true*, the terminal node 0 represents *false*, while non-terminal nodes represent functions which are co-factors of the function represented by their predecessor. If the original function $f(x)$ is *true* for some $x \in \{0,1\}^k$, there exists a path from the root node to the terminal node labeled as 1; if $f(x)$ is *false*, the path reaches the terminal node marked as 0.

Recently, reduced ordered binary decision diagrams (ROBDDs) were employed for flow-based computing approach using sneak paths to implement Boolean functions on nanoscale memristive crossbars [6]. ROBDDs are a subclass of BDDs where variable ordering has to be maintained on each path from the root node to the terminal nodes. For example, if $\pi = \{v_1, v_2, \ldots v_k\}$ represents the variable ordering, v_1 should always appear before v_2 on each path from the root node to the terminal node. ROBDDs with a given variable ordering are canonical representations of Boolean functions [50]. Efficient inductive implementations of basic Boolean operations using BDDs have been implemented in popular software packages [51–54].

FBDD-based Synthesis of Crossbars

ROBDD-based synthesis for flow-based computing circuits can lead to large memristor crossbars for functions whose ROBDDs are exponential in the number of variables. There are several interesting Boolean functions with exponential-size ROBDDs but only polynomial-size Free Binary Decision Diagrams (FBDDs) [55]. FBDD-based synthesis seeks to exploit this fact to synthesize compact crossbar circuits for flow-based computing.

The requirement of a strict variable ordering along all paths of a ROBDD is relaxed in Free Binary Decision Diagrams (FBDDs); hence, different paths from the root to the terminal nodes of a FBDD may represent different orderings of the variables in the FBDD [55]. Like ROBDDs, FBDDs also do not allow repeated occurrences of variables along any path from the root node to the terminal nodes. In general, FBDDs are more compact than ROBDDs because FBDDs do not enforce the same strict variable ordering along all paths from the root node to the terminal node of the decision diagram.

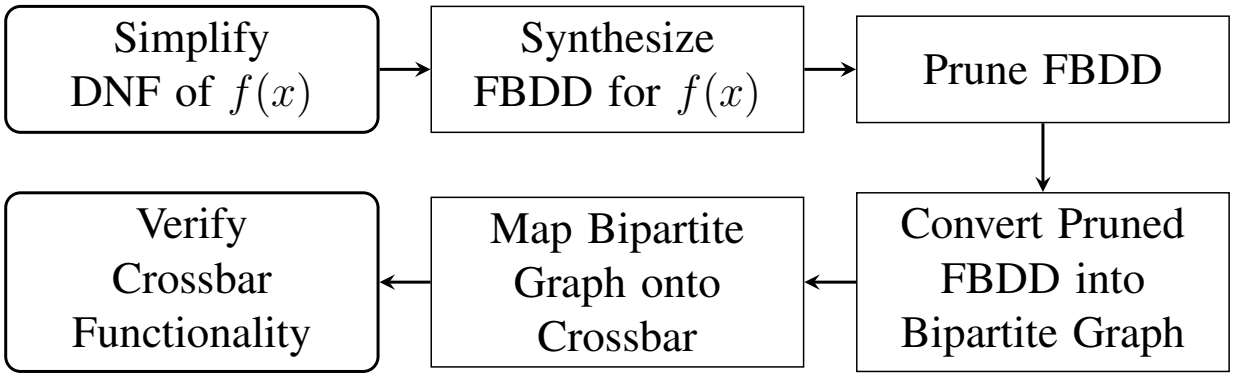
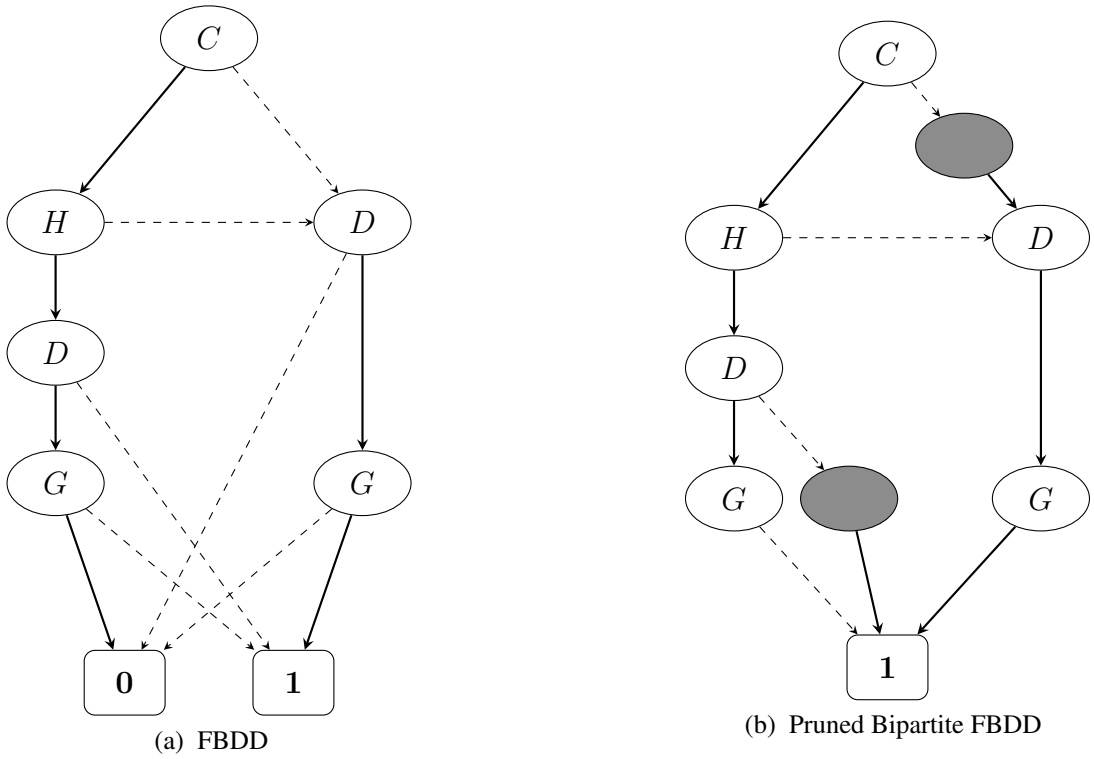


Figure 3.1: Flow diagram of our FBDD-based synthesis approach.

Figure 3.1 shows the flow diagram illustrating the steps of the synthesis process based on FBDDs. The first step transforms the given Boolean formula f into a simplified Disjunctive Normal Form (DNF). In the next step, a Free Binary Decision Diagram representation of the Boolean function f is synthesized. By definition of a FBDD, the functions represented by a node and its children are related by the Shannon expansion: $f(x) = af(x|_{a=1}) + \neg af(x|_{a=0})$. Here, f is the function implemented by the parent node, $f(x|_{a=1})$ and $f(x|_{a=0})$ are the functions implemented by the children nodes and a is the binary variable around which $f(x)$ is decomposed. In this approach to the synthesis of FBDDs, the variable a is obtained using a greedy heuristic. A Boolean variable a is chosen such that it appears most often in the DNF representation of the function f. The intuition behind choosing a using this greedy heuristic is that the DNF of the resulting co-factors $f(x|_{a=0})$

and $f(x|_{a=1})$ would be small for such a choice of a. Here, the size of a formula is computed as the total number of conjunctions and disjunctions in its DNF representation.



(a) FBDD

(b) Pruned Bipartite FBDD

Figure 3.2: (a) Free Binary Decision Diagram (FBDD) for second-output-bit of a 4-bit multiplier. A:D represent the first operand and E:H represent the second operand. (b) Bipartite graph of the pruned FBDD for the second-output-bit of a four bit multiplier synthesized using our approach. Dark nodes are dummy nodes used for converting the pruned FBDD into a bipartite graph.

Figure 3.2(a) shows the free BDD synthesized for the second-output-bit of a 4-bit multiplier using this heuristic. Incidentally, the resulting graph is same as a ROBDD for this particular function. As is clear from definition 3, we are interested in only those paths that end on the terminal node 1; therefore, the FBDD is pruned to get rid of the edges that are connected to the terminal node 0.

However, the pruned FBDD is not yet ready for mapping onto crossbars. All memristors in crossbars establish connections between horizontal nanowires and vertical nanowires. There are no di-

rect connections between two horizontal nanowires or two vertical nanowires in a crossbar. Hence, the underlying graph corresponding to a nanoscale memristor crossbar is bipartite. In the next step, the pruned FBDD is transformed into a bipartite graph by inserting dummy nodes to eliminate odd-length cycles. It is well known that a graph without odd-length cycles is bipartite. Figure 3.2(b) shows a bipartite graph obtained after pruning and the introduction of dummy nodes into the FBDD of Fig. 3.2(a).

In the final step of synthesis process, the pruned bipartite graph obtained from the FBDD is mapped onto a nanoscale memristor crossbar. First, the distance of each node from the root is measured. The root node is mapped onto the topmost nanowire, nodes with even numbered distance from the root node are mapped onto horizontal nanowires, and nodes with odd numbered distance from the root node are mapped onto the vertical nanowires. Since the graph is bipartite, no node can be at both even and odd distance from the root node. Figure 3.3 shows the synthesized crossbar for the second-output-bit of a 4-bit multiplier.

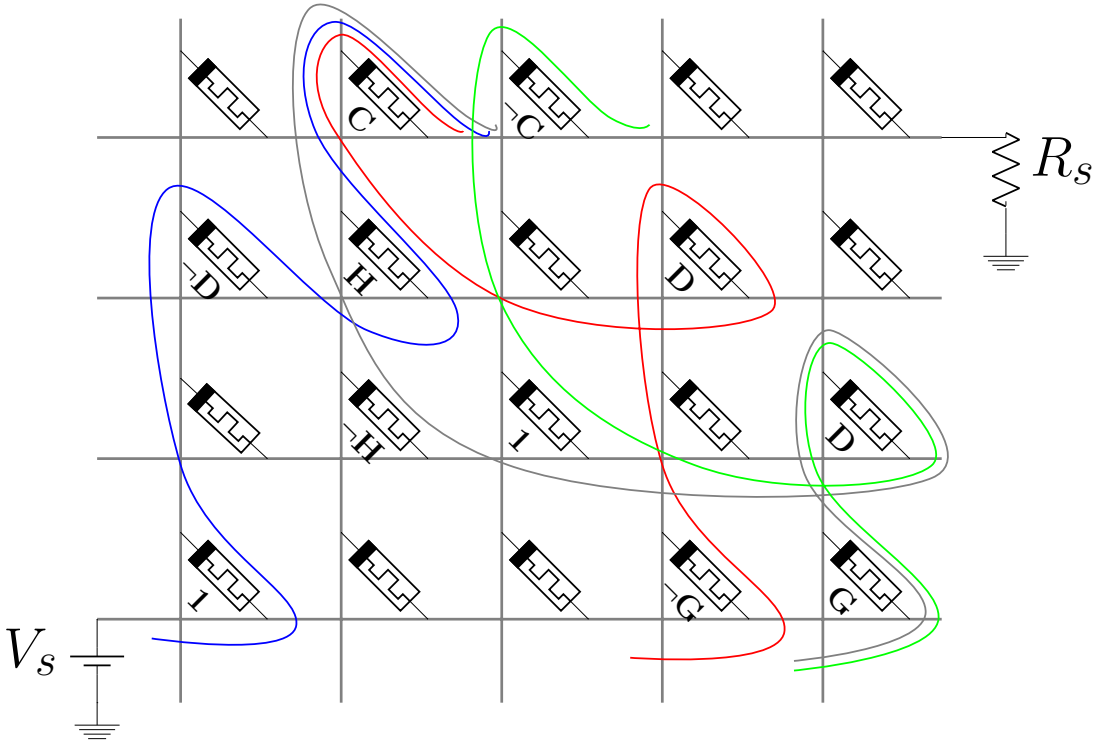


Figure 3.3: Crossbar for the second-output-bit of a 4-bit multiplier with A:D and E:H as operands. Memristors are labeled with the values stored in them and unlabeled memristors are always turned off. The highlighted lines show four sneak paths emanating from the bottom nanowire and reaching the top nanowire. These sneak paths are responsible for computation of second-output-bit of the multiplier.

Table 3.1: Comparison of ROBDD-based approach [6] with FBDD-based approach for output bits of a 4-bit multiplier

	ROBDD-based Synthesis			FBDD-based Synthesis			Best of Both Approaches		
Bit Index	Crossbar Size	Area	Configured Memristors	Crossbar Size	Area	Configured Memristors	Crossbar Size	Area	Configured Memristors
1 (LSB)	2 by 2	4	3	2 by 2	4	3	2 by 2	4	3
2	4 by 5	20	11	4 by 5	20	11	4 by 5	20	11
3	19 by 19	361	51	8 by 7	56	21	8 by 7	56	21
4	66 by 60	3960	186	35 by 34	1190	103	35 by 34	1190	103
5	42 by 40	1680	124	47 by 49	2303	136	42 by 40	1680	124
6	27 by 28	756	82	47 by 45	2115	129	27 by 28	756	82
7	17 by 20	340	55	28 by 28	784	80	17 by 20	340	55
8 (MSB)	7 by 9	63	22	10 by 11	110	30	7 by 9	63	22
Total		**7184**	534		**6582**	513		**4109**	421

Experimental Results

We have synthesized a 4-bit multiplier using our approach. It has two input operands: the first operand is comprised of bits A:D and the second operand is comprised of bits E:H. Since the output of a 4-bit multiplier is an eight bit number, we have synthesized eight crossbars. Table 3.1 presents the sizes of the synthesized crossbars and configured memristors for each output bit. The correctness of the synthesized crossbars is verified exhaustively by applying all input combinations on the synthesized crossbar designs. The sneak paths between the bottom and topmost nanowires existed only when the corresponding output was *true*; there was no path whenever the function output was *false*.

In order to verify the correctness of our designs, we perform quantitative SPICE resistive network simulations for all possible 256 input configurations of a 4-bit multiplier. We focus on the middle fourth-bit of the multiplier and used the values of $R_{ON} = 50\Omega$, $R_s = 100\Omega$, $V_s = 1V$ and $R_{OFF} = 500k\Omega$ for our simulations. Memristors with HRS (high resistance state) to LRS (low resistance state) ratio of 10^7 have been reported in literature [56]. Figure 3.4 summarizes the experimental observations. Flows corresponding to *true* formulae (shown in blue) are clearly distinguished from flows corresponding to the *false* formulae (shown in red). The minimum output voltage for a *true* formula was 0.177V while the maximum output voltage for a *false* formula was 0.053V; hence, the two truth values are clearly distinguishable in all cases.

We have investigated the impact of memristor variability on the correctness of our designs by changing the resistance values by 5%. We vary the resistance of each memristor in the crossbar corresponding to the middle fourth-bit of the 4-bit multiplier by increasing (decreasing) its value by 5%. The lowest voltage corresponding to *true* falls from 0.177V to 0.170V while the highest value corresponding to *false* rises from 0.053V to 0.056V; both the *true* and the *false* values are clearly distinct from each other in all cases.

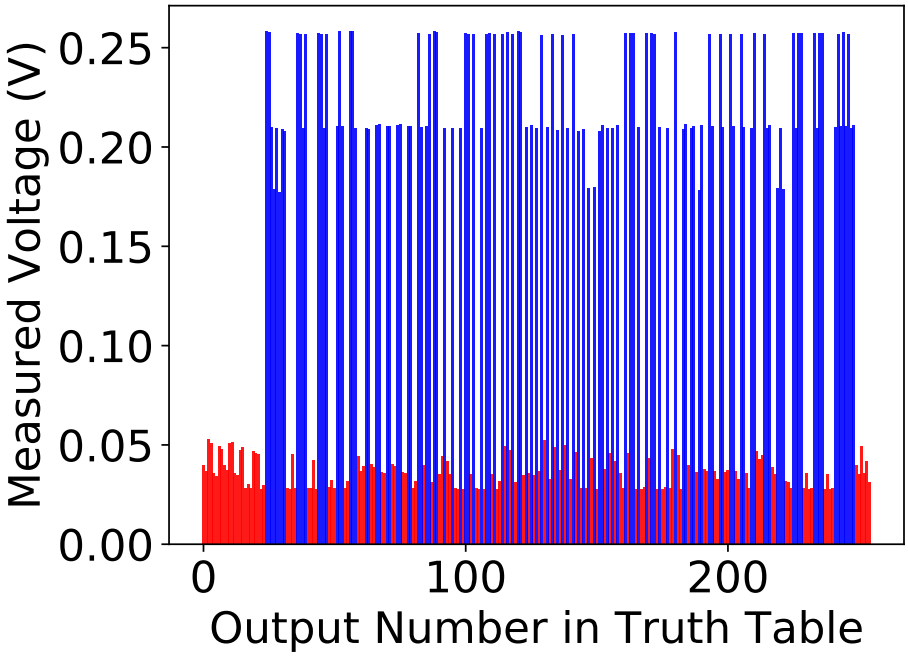


Figure 3.4: Output for the fourth-output-bit of the multiplier. The X-axis represents the index of truth table entries, and the Y-axis is the voltage across R_s. Blue lines correspond to input combinations with *true* outputs, red lines represent *false* outputs for the Boolean function. Output voltage is at least 0.177 V when the Boolean formula is *true*. Voltage is no more than 0.053 V when the Boolean formula is *false*.

Table 3.1 compares the performance of the FBDD based heuristic approach with the previous ROBDD-based approach [6]. For the first four output bits, the heuristic based variable ordering has produced either similar sized or smaller crossbars. But for the last four bits, ROBDD-based variable ordering has resulted in smaller crossbars. A 4-bit multiplier generated using our approach needs 8.4 percent less area than the ROBDD-based approach, while a multiplier designed using the best of both approaches needs 42.8 percent less area than an approach based only on ROBDDs.

Comparisons

This section presents a qualitative comparison of our approach with Memristor Ratioed Logic (MRL) [28], IMPLY-based syntheses [31, 57, 58], Memristor-Aided Logic (MAGIC) [3], and other flow-based syntheses such as mapping of Negation Normal Form of Boolean formulae on crossbars [16], formal methods-based crossbar synthesis [17], and model counting-based crossbar synthesis [59].

MRL uses both memristors and CMOS transistors (for inversion) for computing logic [28]. Besides inversion, the usage of transistors also facilitates signal restoration in MRL. But the larger size of CMOS nodes decreases the density of MRL-based circuits. Therefore, MRL-based circuits are not as compact as purely memristive circuits.

IMPLY gate can be implemented using two memristors. IMPLY and $false$ are sufficient to compute any Boolean function on memristive circuits [58]. A Boolean function is computed by executing a sequence of steps (called micro-operations) on an IMPLY-based memristive circuit. The number and order of micro-operations depends on the nature and complexity of the function being computed. Chakraborty et al. have replaced each node of a Binary Decision Diagram (BDD) with an IMPLY-based 2-to-1 multiplexer (MUX) [31]. Their MUX is realized with 5 memristors and needs 6 micro-operations. Thus, each node of their BDD needs 5 memristors and needs 6 micro-operations in their implementation of a BDD. In comparison, our unmapped designs configure at most two memristors for each node of a BDD. Additionally, our BDD-based designs are not only directly implemented on memristor crossbars, they are also free from micro-operations.

It is not yet clear how memristive circuits comprised of IMPLY gates can be implementedModern computers are based on the von Neumann architecture. The earliest description of the von Neumann architecture comes from The First Draft of a Report on EDVAC [8]. The EDVAC (Electronic

Discrete Variable Automatic Computer) was envisioned to have a Central Processing Unit (CPU) –comprised of an arithmetic logic unit (ALU) and a control unit (CU)– data and pro-gram memories, and input/ output peripherals. Program instructions were stored in a memory unit, just like other data in memory. Data and address buses were provided to move data between dif-ferent parts of the computer. Thanks to the exponential decrease in transistor size in the coming decades, computational capabilities of CPUs kept increasing exponentially. The size of memory also increased due to the same reason. However, the communication capacity of the shared bus, connecting CPU with memory units, didn't increase at the same pace. Consequently, an increase in the computational capabilities of CPU or the size of memory did not translate into a similar increase in the throughput of computing systems. The limited speed of the shared communication-bus adversely affects system performance due to the time wasted in transferring data between CPU and memory. This phenomenon is known as von Neumann-bottleneck, memory-processing bottle-neck , or communication bottleneck. For decades, exponential growth in computing capabilities due to Moore's law was sufcient to mask the inherent shortcomings of the von Neumann architecture. However, in recent years, not only has Moore's law slowed down due to transistor size reaching atomic scale, but fabrication technology is also facing problems on other fronts such as, increasing leakage currents due to quantum tunneling, process variations at such a small scale, and vulnerability of small-scale elec-tronics to radiation ips even at sea levels. The breakdown of Dennard's scaling in the mid 2000s due to increasing dynamic power meant 1 that the trend of increasing clock frequencies cannot continue indenitely. This ushered the era of multicore scaling, where multiple CPUs fabricated on the same die work in parallel to improve system performance. Although multicore scaling has improved the computing throughput of gen-eral purpose computers, this trend is obstructed by the inability to dissipate all the dynamic power generated by a fully powered CPU. This phenomenon is known as dark silicon, because all of the circuitry in a die cannot be powered-on at the same time without violating the thermal constraints. Recent advances in machine learning (ML), articial intelligence (AI), and data mining, and their widespread applications

have changed both the nature and size of computing. It has become in-creasingly clear that general purpose architecture cannot efciently fulll the demands of these specialized compute-intensive applications. For instance, Graphics Processing Units (GPU) have long eclipsed CPUs not only for multimedia, but also for AI and ML applications. Researchers are also developing AI and ML specic computing architectures such as Tensor Processing Unit (TPU), Deep Learning Accelerator (DLA), Vision Processing Unit (VPU), and Neural Network Processor (NNP). These Application Specic Integrated Circuits (ASIC) are also called AI accel-erators. It is important to mention that despite its deciencies, ASICs are not intended to replace general purpose computer. These new computing architectures such as GPUs, NPUs, and other ASICs are intended to augment the general purpose computer to increase overall computing ef-ciency. Besides advances on architectural fronts, recent advances in material science and nanotechnology haves propelled several new nanoelectronic devices to the forefront. Some recently discovered de-vices are memristors, magnetic tunnel junctions (MTJ), 3d transistors, spin-torque-transfer (STT) memories, phase change memories (PCM), etc. These newly discovered devices are collectively referred to as emerging devices, and the novel architectures employing these devices are referred to as emerging computing architectures. Small size, low to no standby power, non-volatility, and high read/ write speeds are distinct features of these emerging devices as compared to traditional 2 Complementary Metal Oxide Semiconductor (CMOS) transistors. Circuits built from these emerg-ing nanodevices devices are not only faster and more power efcient, they are also more compact than the traditional CMOS-based circuits. On the other hand, the problems associated with these devices are high variability, low reliability, smaller lifespan (number of read/ write cycles) and dif-fering read and write times. These devices are still in the research phase, and these parameters are likely to improve further in coming years. Nonetheless, the aforementioned properties of these devices make them excellent candidates to augment or improve existing computing systems for better power, area, delay, and computing efciency. Emerging ReRAM devices for Novel Architectures on a crossbar structure [3]. Other IMPLY-based circuits such as IMPLY-based NAND, IMPLY-based Majority

gate (MIG-IMP), and majority (MAJ) [5,57] are not designed for direct computation on crossbars either. On the other hand, our flow-based computing using decision diagrams is directly mappable on a crossbar without the need of any modifications in the crossbar structure.

MAGIC employs purely memristive circuits for in-memory computing [3,60], and it can be computed on either custom memristive circuits or memristor crossbars. When crossbars are used for computing MAGIC, the problem of sneak paths arises, which needs to be dealt with [15,47,61]. Flow-based computing is not affected by such sneak paths in crossbars. On the contrary, flow-based computing employs these sneak paths as flow paths between the bottom and the topmost nanowires. We have used FBDDs to map these sneak paths between the bottom and the topmost nanowires [62].

Velasquez et al. have proposed two approaches for designing crossbars for flow-based computing. Their first approach maps the NNF of a Boolean function on a crossbar [16]. NNF-based synthesis produces prohibitively large crossbar circuits. For example, their NNF-based design for a 1-bit adder requires a 16×16 crossbar. In their second approach, Velasquez et al. have used formal methods for synthesizing crossbars circuits [17]. By using formal methods, they have implemented the 1-bit adder on a 4×4 crossbar. However, this approach is also not scalable due to high computational complexity of the synthesis process. In comparison, the BDD-based synthesis for flow-based computing is more scalable due to lower computational requirements than the formal methods-based synthesis and smaller size of the synthesized crossbars than the NNF-based designs.

Conclusions

In this chapter, we have presented a new FBDD-based computer-aided design approach for synthesizing compact crossbars that implement Boolean formulae using flow-based computing [17,45,

59]. Free Binary Decision Diagrams are often more succinct than ROBDDs as they do not enforce the requirement of a strict variable ordering along all paths from the root to the terminal nodes of a decision diagram. We have taken advantage of this increased representational power of FBDDs for designing compact nanoscale memristor crossbars. In our experimental investigations, FBDDs designed using a simple greedy heuristic have resulted in identical or more compact crossbars for the first four output bits of a 4-bit multiplier. We have also used other heuristics for designing compact FBDD-based crossbars [63, 64]. The FBDD-based in-memory crossbar computing approach is not specific to memristor crossbars. The methodology can also be employed to design circuits using other Re-RAM devices [65, 66].

CHAPTER 4: AUTOMATED SYNTHESIS OF STOCHASTIC COMPUTATIONAL ELEMENTS [1]

Introduction

John von Neuman's work on "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components" was the first comprehensive study of stochastic computing [67]. During the 1950s and 1960s, further advances were made in stochastic computing techniques [18, 36, 37]. However, the progress in stochastic computing slowed down in the late 1960s because of the availability of reliable and cost-effective integrated circuits for digital computing. Over the last couple of decades, the interest in stochastic computing has picked up again due to slowdown in Moore's law. Transistor dimensions in commercially available integrated circuits are projected to reach 5 nm range in 2020. At such a small scale, their reliability decreases due to reduced noise margin, radiation-induced errors and manufacturing variations. Such uncertain behavior has driven the exploration of fault-tolerant computing paradigms. Stochastic circuits can inherently tolerate a large number of errors due to its probabilistic nature. Their intrinsic robustness along with their area and power efficiency have made them competitor to the deterministic implementations for energy-hungry fault-tolerant applications such as multimedia and pattern recognition.

Stochastic circuits are also called stochastic computational elements (SCE) [18]. In this chapter, we discuss the synthesis of finite state machines (FSMs) for approximating non-linear functions stochastically. In 2001, Brown and Card used finite state machines to approximate tanh and exponentiation functions using a stochastic bit stream [19]. Li et al. devised a scheme for the synthesis

[1] Related Publication: A. U. Hassen, B. Chandrasekar and S. K. Jha, "Automated Synthesis of Stochastic Computational Elements using Decision Procedures", 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, QC, 2016, pp. 1678-1681.

of SCEs and demonstrated its effectiveness by synthesizing polynomial, tanh, and exponentiation functions [7]. The synthesized circuits were more compact and robust to errors as compared to their deterministic implementations. Their implementation also had better area efficiency as compared to the circuits yielded by Bernstein polynomials [7]. Li et al. synthesize stochastic circuits by minimizing the square of error between the target function and its estimate [7] over the input range. This approach does not guarantee any upper bound on the maximum error between the target function and its estimate.

To optimize the worst-case error, we propose a new algorithm which guarantees an upper bound on the absolute error between the target function and its estimate. We use decision procedures to iteratively search the space of linear finite state machines till our error constraints are satisfied, thus limiting the maximum error between the target function and its estimate over the entire input range. To compare our results with previous state-of-the-art approach, we use the same number of states as in Li et al. [7] for synthesizing the finite state machines for polynomial, tanh, and piecewise exponential function. As shown in Table 4.1, the circuits synthesized using decision procedures have 1.17 to 1.65 times smaller worst-case error than the previous state-of-the-art approach [7]. All of this improvement is achieved without any additional hardware or delay.

In the remaining chapter, we discuss Moore's machine, its excitation with a stochastic bit stream, and its steady-state properties. Then, we present a decision procedures-based algorithm for synthesizing linear finite state machines (LFSM). The experimental section compares the worst-case errors of decision procedures-based algorithm with the approach in Li et al. [7] for polynomial, tanh, and piecewise exponential functions.

Linear Finite State Machines

Compact combinational circuits composed of a few logic gates or multiplexers are sufficient for computing elementary arithmetic operations, such as addition and multiplication, using stochastic bit streams. More complex computations such as non-linear function approximation can also be performed stochastically using simple sequential circuits modeled by Finite State Machines (FSM). Single-input single-output linear Moore machines with probabilistic outputs can be used for stochastic estimation of non-linear functions.

Definition 4. *A Moore machine with probabilistic outputs is defined as a 6-tuple* (S, s_i, X, Y, T, G) *where*

(i) S *is a finite set of states* $\{s_0, s_1, s_2, ...s_n\}$,

(ii) s_i *is the initial state,*

(iii) X *is a set of input alphabets, e.g.* $\{0, 1\}$,

(iv) Y *is a set of output alphabets, e.g.* $\{0, 1\}$,

(v) $T \subseteq S \times X \rightarrow S$ *is a transition function from current state* $s_i \in S$ *and the input* $x \in X$ *to the next state* $s_i' \in S$, *and*

(vi) $G \subseteq S \rightarrow P(Y)$ *is a probabilistic output function that assigns a probability* $P(Y|s_i)$ *of producing the output* $y \in Y$ *at state* $s_i \in S$.

Definition 5. *A Moore machine with probabilistic outputs* (S, s_i, X, Y, T, G) *is a linear Moore machine for binary input* x *provided,*

(i) if the input $x = 1$, *the state* s_i *makes a transition to* s_{i+1} *if* $i < n$; *otherwise, the state* s_i *does not change.*

(ii) if input $x = 0$, the state s_i makes a transition to s_{i-1} if $i > 0$; otherwise, the state s_i does not change.

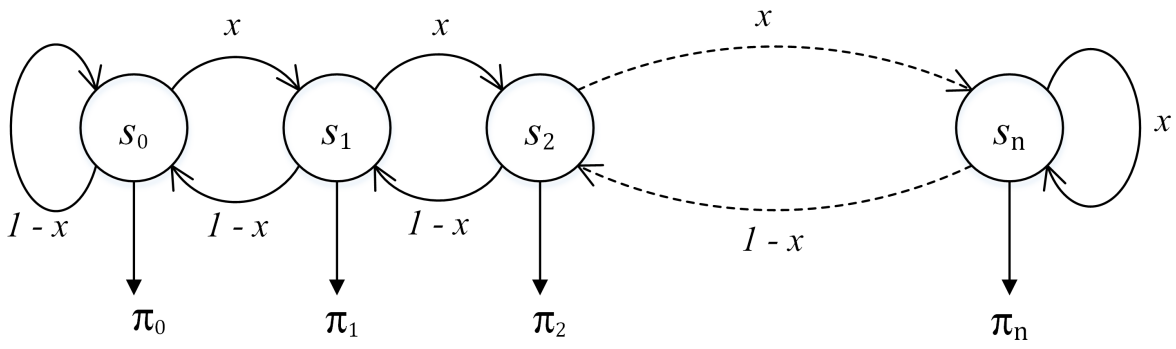


Figure 4.1: The structure of a linear finite state machine.

A linear Moore machine is shown in Fig. 4.1 . Input to this state machine is a Bernoulli sequence with probability of '1' given by $P(x = 1)$. To keep the notation compact, let us denote $P(x = 1)$ as $P(x)$. For a long enough Bernoulli sequence with fixed $P(x)$, the linear finite state machine will reach the equilibrium or steady state. The probability of being in a particular state (s_i) under equilibrium depends only on $P(x)$, let us denote it by $P(s_i|P(x))$.

When linear finite state machine is in equilibrium, the probability of leaving a state will be identical to the probability of entering that state. Formally,

$$P(x)P(s_i|P(x)) = (1 - P(x))P(s_{i+1}|P(x)).$$

In general

$$\Rightarrow P(s_i|P(x)) = \left(\frac{P(x)}{1 - P(x))}\right)^i P(s_0|P(x)). \tag{4.1}$$

Using total probability theorem

$$P(s_0|P(x)) + P(s_1|P(x)) + P(s_2|P(x)) + \cdots + P(s_n|P(x)) = 1.$$

By substituting $P(s_i|P(x))$ from equation 4.1 in the above equation, we get

$$P(s_0|P(x)) = \frac{1}{\sum_{i=0}^{n} \left(\frac{P(x)}{1-P(x)}\right)^i}.$$

When LFSM is in state k, it outputs a Bernoulli stream in which probability of '1' is given by $P(Y = 1|s_k)$. Let us denote this output probability by π_k. Using total probability theorem, the overall probability of '1' in the output stream is given by

$$P(Y = 1|P(x)) = \sum_{i=0}^{n} \pi_i P(s_i|P(x)). \tag{4.2}$$

Synthesis of Stochastic Computational Elements (SCEs) using Decision Procedures

Let $f(t)$ be our target function such that $f(t) \in [0, 1]$ where $t \in [0, 1]$. Since both $t \in [0, 1]$ and $P(x) \in [0, 1]$, therefore we can replace t by $P(x)$. Our objective is to find the output probabilities of linear finite state machine $\{\pi_0, \pi_1, \pi_2, ...\pi_n\}$ such that its steady state output $P(Y = 1|P(x))$ closely follows target function $f(P(x))$ for $0 < P(x) < 1$

$$P(Y = 1|P(x)) \approx f(P(x)).$$

Equation 4.2 is central to the synthesis of stochastic computational elements as described in algorithm 1. Let us define $\epsilon > 0$ as the maximum permitted error between $P(Y = 1|P(x))$ and the target function $f(P(x))$. Then, from equation 4.2,

$$\boldsymbol{f}(P(x)) - \epsilon \leq \sum_{i=0}^{n} \pi_i P(s_i|P(x)) \leq \boldsymbol{f}(P(x)) + \epsilon. \tag{4.3}$$

Algorithm 1 Decision Procedure-based Synthesis of Stochastic Computational Elements

(i) Choose a small value of $\epsilon > 0$.

(ii) Initialize a set of constraints, $C = \phi$.

(iii) Choose N samples $\{P(x_1)$,$P(x_2)$, $P(x_3)$,$\cdots$ $P(x_N)\}$ of $P(x)$ by sampling it uniformly between 0 and 1.

(iv) $For\ each\ P(x_k)$

- Compute $P(s_i|P(x_k))$ for all $s_i \in S$ using Eq. 4.1.
- Compute $\boldsymbol{f}(P(x_k))$.
- Substitute $P(s_i|P(x_k))$ and $\boldsymbol{f}(P(x_k))$ in Eq. 4.3 and create a constraint c_k in terms of $\{\pi_0, \pi_1, \pi_2, ...\pi_n\}$.
- Add c_k to the set of constraints C.

(v) Find the feasible region by solving the set of constraints C using decision procedures.

(vi) if (feasible region is found) $then\ return$ $\{\pi_0, \pi_1, \pi_2, ...\pi_n\}$, $else$ Set $\epsilon := \frac{\epsilon}{0.99}$ and repeat all the steps from Step (ii).

First step is the specification of error tolerance ϵ at sampled points. Next step is the creation of N samples of $P(x) \in [0, 1]$. Substitution of each value of $P(x_k)$ in equation 4.2 gives a constraint in terms of output probabilities $\{\pi_0, \pi_1, \pi_2, ...\pi_n\}$. After creating N such constraints, we can solve them using decision procedures. The solution $\{\pi_0, \pi_1, \pi_2, ...\pi_n\}$ satisfying all the constraints is the set of output probabilities of linear finite state machine approximating the target function $\boldsymbol{f}(t)$. If no such solution is found, we can relax our constraints by setting ϵ to a slightly larger value. We

again solve these new constraints using decision procedures. This process is repeated iteratively till the solution is found. In next section, we show that the output probabilities obtained from decision procedures result in bounded error at unsampled values of $P(x)$.

Theorem 1. *(Boundedness of Error) The error between a target function and its estimate between sampled points can be made arbitrarily close to ϵ by increasing the sampling rate N.*

Proof. Error between the target function and its estimate is always less than ϵ at sampled points. If target function and its estimate are Lipschitz continuous, then the error between sampled points decreases as we increase the sampling rate.

Let x be any of the N sampled points, let Δx be a small interval such that $\Delta x < \frac{1}{N}$, For compactness of notation, let us denote $P(Y = 1|x)$ by $g(x)$, then we can re-arrange inequality 4.3 as,

$$-\epsilon < f(x) - g(x) < \epsilon.$$

Since $\Delta x < \frac{1}{N}$, $x + \Delta x$ will be between sampled points. Let L_f be the absolute value of the Lipschitz constant of $f(x)$, then

$$f(x) - L_f \Delta x < f(x + \Delta x) < f(x) + L_f \Delta x, \tag{4.4}$$

Let L_p be the absolute value of Lipschitz constant of $P(Y = 1|x) = g(x)$, then

$$g(x) - L_p \Delta x < g(x + \Delta x) < g(x) + L_p \Delta x, \tag{4.5}$$

Subtracting inequality 4.5 from inequality 4.4,

$$\begin{aligned} f(x) - g(x) - (L_p + L_f)\Delta x \ &< \ (f(x + \Delta x) - g(x + \Delta x)) \\ &< (f(x) - g(x)) + (L_p + L_f)\Delta x, \end{aligned}$$

By using inequality 4.3,

$$-\epsilon - (L_p + L_f)\Delta x < f(x + \Delta x) - g(x + \Delta x) < \epsilon + (L_p + L_f)\Delta x,$$

Substituting $\Delta x < \frac{1}{N}$ in the above inequality, we get

$$-\epsilon - \frac{(L_p + L_f)}{N} < f(x + \Delta x) - g(x + \Delta x) < \epsilon + \frac{(L_p + L_f)}{N}. \tag{4.6}$$

It is clear from 4.6 that for a given ϵ, we can make the error arbitrarily small by increasing the sampling rate and making Δx arbitrarily small. □

Lipschitzness of Estimated Function

The synthesized function $P(Y = 1|P(x))$ given by equation 4.2 is Lipschitz continuous if steady state probabilities $P(s_i|P(x))$ of all states $s_i \in S$ are also Lipschitz continuous. Steady state probability of state s_i is given by,

$$P(s_i|P(x)) = \frac{\left(\frac{P(x)}{1-P(x))}\right)^i}{\sum_{k=0}^{n}\left(\frac{P(x)}{1-P(x))}\right)^k},$$

i.e,

$$P(s_i|P(x)) = \frac{r^i}{\sum_{k=0}^{n} r^k},$$

where $r = \frac{P(x)}{1-P(x))}$ and it changes from 0 to ∞ when x changes from 0 to 1. Using chain rule, the derivative of $P(s_i|P(x))$ with respect to $P(x)$ is given by,

$$P'(s_i|P(x)) = \frac{r^{i-1}(r+1)^2 \sum_{k=0}^{n}(i-k)r^k}{(\sum_{k=0}^{n} r^k)^2}. \tag{4.7}$$

$P'(s_i|P(x))$ is bounded for $0 < r < \infty$, which means $P(s_i|P(x))$ is Lipschitz continuous for all i. Therefore the synthesized function $P(Y = 1|P(x))$, which is the weighted sum of $P(s_i|P(x))$, is also Lipschitz continuous.

Experimental Results

We evaluate our approach by synthesizing linear finite state machines for approximating polynomial, tanh, and piecewise exponential functions given by equations 4.8, 4.9, and 4.10 respectively. Li et al. also used these functions as benchmarks [7]. For each of these functions, we synthesize a linear finite state machines by solved 10^3 constraints obtained from equation 4.2. These constraints are created by randomly sampling $P(x)$ in the range [0,1]. Next, we present the finite state machines and their output for each of these functions. We also compare our worst-case error with Li et al. [7] for each of these functions.

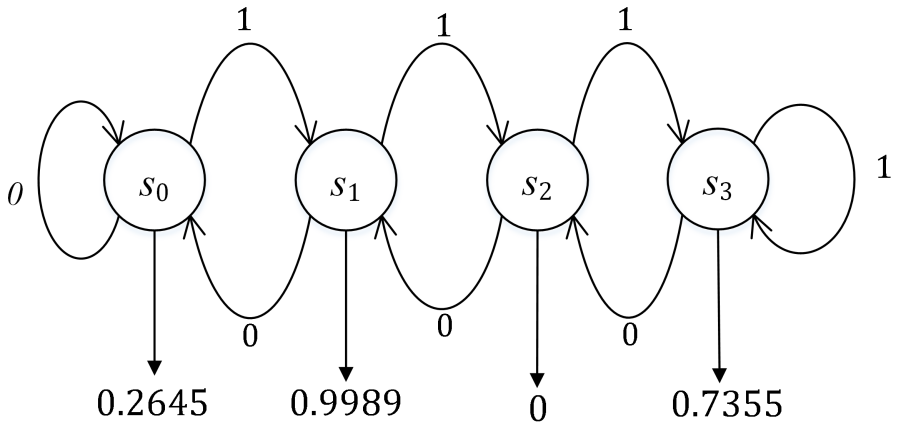


Figure 4.2: A linear finite state machine approximating the polynomial function using 4 states. Our design has 1.65 times smaller worst-case error than the state-of-the-art approach.

Synthesis of Polynomial Function

We have synthesized the following third degree polynomial given in equation 4.8.

$$P(Y = 1|P(x)) = \frac{1}{4} + \frac{9}{8}P(x) - \frac{15}{8}P^2(x) + \frac{5}{4}P^3(x). \tag{4.8}$$

Fig. 4.2 shows the synthesized 4-state finite state machine along with the output probabilities for the polynomial function. Fig. 4.3 compares the output of the synthesized machine with the exact output of equation 4.8 over [0,1] range. The output of our linear finite state machine deviates from the target polynomial function by 0.0145 in the worst case. While the worst-case error for the machine in Li et al. is 0.024, which is 1.65 times larger than our approach.

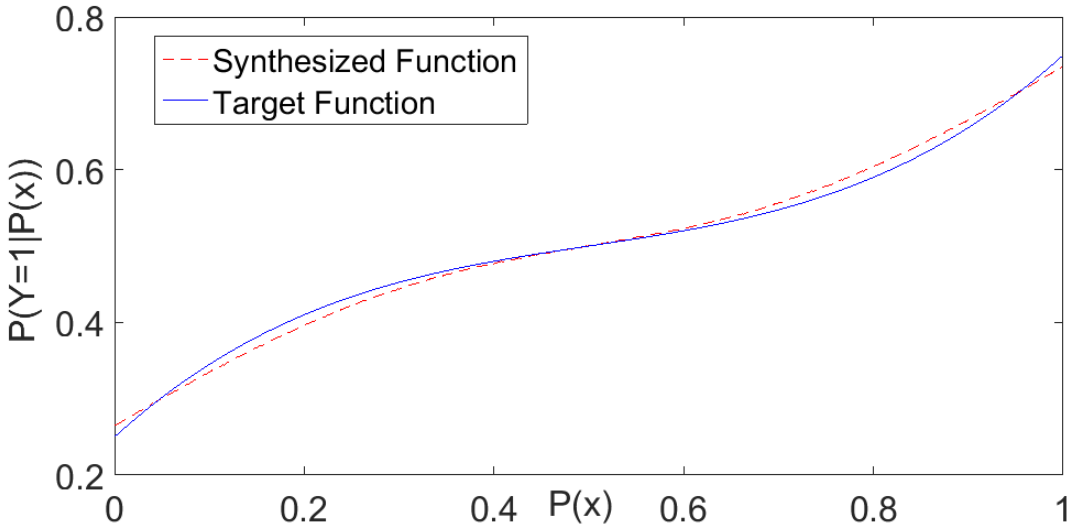


Figure 4.3: Comparison of the stochastic approximation of polynomial function and the exact function in equation 4.8. Our design has 1.65 times smaller worst-case error than the state-of-the-art approach.

Synthesis of Tan-hyperbolic Function

Stochastic tan-hyperbolic function can be approximated by the following equation,

$$f(x) \approx \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}},$$

where 'N' is the number of states of the linear finite state machine used for synthesizing this function. Here $f(x)$ and 'x' are represented using the bipolar coding format [18]. Before synthesizing this function, we have converted it into unipolar coding format resulting in following equation,

$$f(x) = \frac{e^{8(2P(x)-1)}}{e^{8(2P(x)-1)} + 1} \tag{4.9}$$

Fig. 4.4 shows an 8-state linear finite state machine synthesized using decision procedures for approximating this function. Fig. 4.5 compares the the stochastic output of this machine with the exact function given in equation 4.9. For this function, Peng Li's approach [7] resulted in the worst-case error of 0.065, while the worst-case error of our approach is 0.04, resulting in 1.625 times improvement.

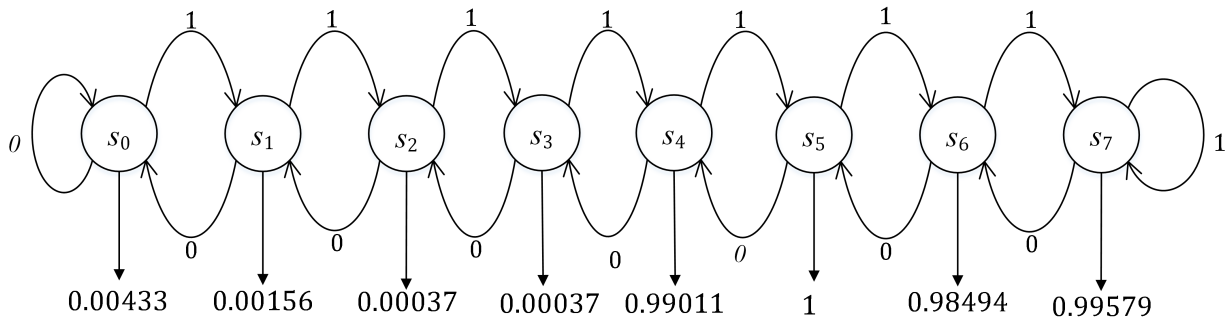


Figure 4.4: A linear finite state machine approximating the tanh function using 8 states. Our design provides 1.625 times smaller worst-case error than the state-of-the-art approach.

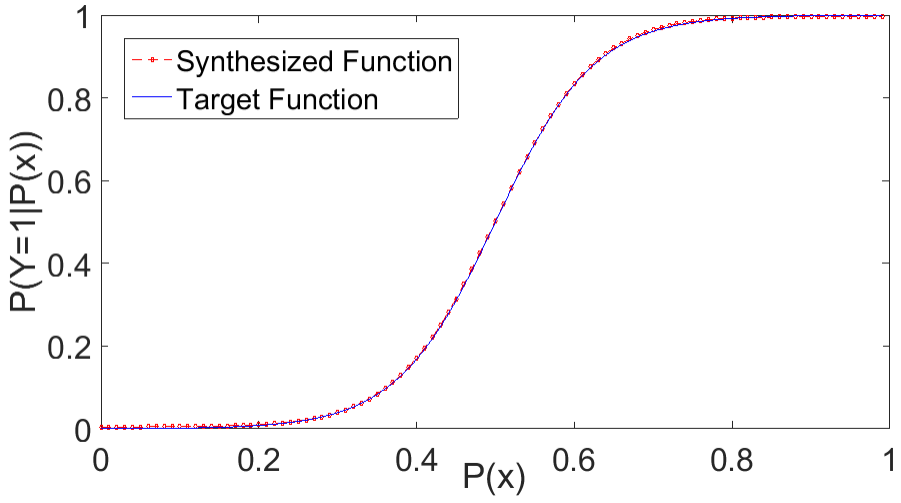


Figure 4.5: Comparison of the stochastic approximation of tanh and the target tanh function. Our design provides 1.625 times smaller worst-case error than the state-of-the-art approach.

Synthesis of Piecewise Exponential Function

We have synthesized the piecewise exponential function given by equation 4.10 using the 16-state linear finite state shown in Fig. 4.6. Fig. 4.7 compares the output of this machine is with the exact function. For this function, the maximum error between the target function and its approximation using our approach is 0.1119. While Peng Li's [7] finite state machine has the worst-case error of 0.125. Again, our approach produced smaller 1.17 times worst-case error as compared to Li et al. [7].

$$P(Y=1|P(x)) = \begin{cases} 1, & 0 \leq P(x) \leq 0.5 \\ e^{-4(2P(x)-1)}, & 0.5 \leq P(x) \leq 1 \end{cases} \tag{4.10}$$

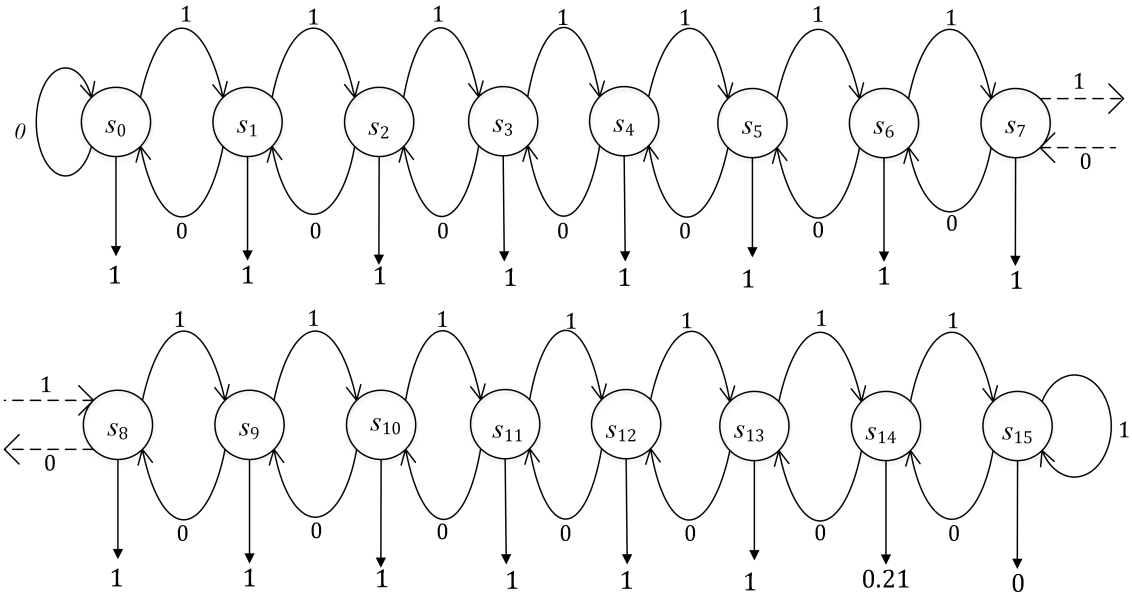


Figure 4.6: A linear finite state machine approximating the exponentiation of equation 4.10 using 16 states. Our design provides 1.17 times smaller worst-case error than the state-of-the-art approach.

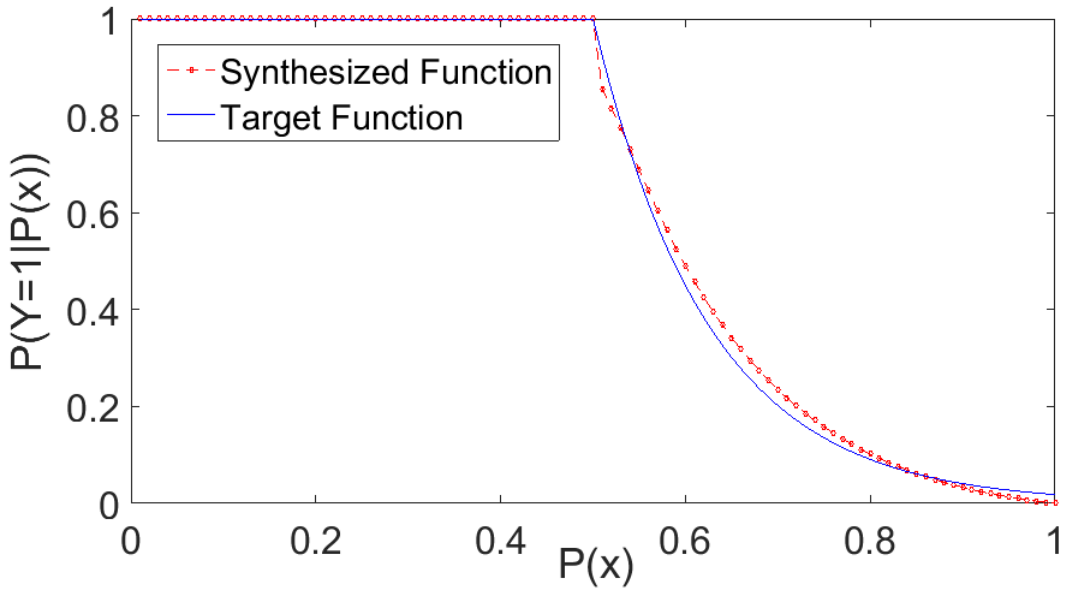


Figure 4.7: Comparison of the our stochastic approximation of exponential function and the target exponential given in equation 4.10. Our design provides 1.17 times smaller worst-case error than the state-of-the-art approach.

Summary of Results

Table 4.1 summarizes the results of our finite state machines synthesized using decision procedure with those synthesized using the approach described in Li et al. [7]. All three functions were synthesized on finite state machines having the same number of states as used by Li et al. [7]. Despite using the same amount of resources, our decision procedures-based synthesis has reduced the worst-case error by up to 65 percent.

Table 4.1: Comparison of worst-case errors of our synthesized linear finite state machines with those synthesized using Li et al. [7]

Function	Li et al. Error [7]	Our Error	Improvement
Polynomial	0.024	0.0145	1.65 times
tanh	0.065	0.04	1.625 times
exp	0.125	0.1119	1.17 times

Conclusions

We have proposed a decision procedures-based algorithm for the synthesis of stochastic computational elements. First, we uniformly sample the input range [0,1], then we convert each sample into an error constraint. Then we use decision procedures to search the space of linear finite state machines with probabilistic outputs such that absolute error between the actual function and its approximation does not exceed predefined threshold. Lipschitzness of the estimated function guarantees that the error between the target function and its estimate remains bounded. We have tested it on polynomial, piecewise exponential and hyperbolic functions. Using this approach [68], we have synthesized approximations for polynomial, exponentiation and tanh functions using linear finite state machines. Our approach resulted in 1.17 to 1.65 times smaller worst-case errors for these functions as compared to the previous state-of-the-art approach [7].

CHAPTER 5: CONCLUSION AND FUTURE WORKS

The decline of Moore's law, the increasing demand of computing capabilities, and the shortcomings of the von Neumann architecture are some of the factors behind increasing interest in unconventional ways of computing. We have explored two unconventional computing systems, stochastic computing and in-memory computing, that do not suffer from the traditional processor-memory bottleneck of the von Neumann architecture.

Besides being resilient, stochastic computing increases the computing efficiency for error-tolerant applications by creating simple but effective approximations. For example, multiplication, a computationally expensive operation on traditional circuits, can be stochastically realized by feeding stochastic streams to a digital AND gate. Error-tolerant multimedia applications, which require large number of such basic arithmetic operations, can be efficiently processed on stochastic circuits. Finite State Machines (FSM) can be used for synthesizing circuits for more complex tasks. We have presented a decision procedures-based iterative algorithm for synthesizing stochastic approximations of non-linear continuous functions, such as polynomial, tanh, and exponentiation, on linear finite state machines. Previously, Li et al. synthesizing stochastic implementation of these functions by optimizing a square-loss function [7]. Our approach has reduced the worst-case error by up to 1.65 times as compared to the state-of-the-art approach [7].

In-memory computing unifies data storage and data processing at the physically same location, thus eliminating the infamous memory-processor bottleneck. We synthesize crossbars circuits for flow/sneak path-based in-memory computing of Boolean functions. Sneak paths, which pervade memristor crossbars, play a central role in flow-based in-memory computing on crossbars. We have used a variant of Binary Decision Diagrams (also referred to as Free-BDDs or FBDDs) to layout sneak paths in crossbars in a controlled way. Computational resources associated with

the synthesis of FBDDs make them less popular than their reduced-ordered counterpart called ROBDDs. On the upside, the freedom of variable ordering lends far more representational power to FBDDs as compared to ROBDDs. Freedom of variable ordering means input variables can appear in different orders along different paths between the root node and the terminal nodes of an FBDD. We have used a simple greedy heuristic to choose different variable-orderings along different paths of an FBDD. Our FBDD-based multiplier was 8% more compact than ROBDD-based designs, while a multiplier using the best of both approaches (ROBDD and FBDD) was 42% more compact than the designs based on ROBDDs only and 37% more compact than the designs based on FBDDs only. Once a crossbar is synthesized, the synthesis process becomes immaterial for its subsequent applications. An important advantage of flow-based in-memory computing is that this technique is designed for generic crossbars, eliminating the need for custom fabrication of memristive crossbars. The use of generic crossbars can also make the process more cost-efficient due to economy-of-scale.

Although algorithmic-depth of BDDs has resulted in more compact crossbars than some previous approaches [16,17], a large number of memristors in the crossbar still remain idle. The problem of idle memristors in the synthesized crossbar is natural for BDD-based synthesis. When a bipartite BDD is mapped onto a crossbar, the BDD nodes are mapped onto crossbar nanowires while the edges between the nodes are mapped onto memristors between the perpendicular nanowires of the crossbar. Thus, the number of nanowires in the synthesized crossbar is proportional to the number of nodes in the BDD, while the number of available memristors in the crossbar is roughly proportional to the square of the number of BDD nodes. Since all non-terminal nodes of a BDD have two outgoing edges, the number of utilized memristor in such crossbars is also linear in terms of the number of nodes, while the number of unused memristors is roughly quadratic in terms of nodes. Consequently, as the size of a BDD increases, the ratio of utilized to unutilized memristors in the synthesized crossbar approaches zero at a quadratic rate.

The number of configured memristors in the crossbars designed using the best of the ROBDD and FBDD approaches varies according to the complexity of the Boolean function being synthesized. The configured memristors occupy 75%, 55%, 37.5%, 8.66%, 7.38%, 10.85%, 16.18% and 34.92% of the crossbar space for the first through eighth output bits. Thus, the decision-diagram based approach produces sparse crossbar designs. Our current FBDD-based approach relies on the availability of memristors with high HRS-LRS ratios. An approach that uses smaller and more dense crossbars is likely to reduce the need for memristors with high HRS-LRS ratios. In order to better utilize the densely packed memristors in a crossbar structure, a fundamentally different approach employing a different data structure may be explored. Approximate in-memory computing can synthesize compact crossbar with reduced-computational complexity by sacrificing some accuracy [69, 70]. Recently, Chakraborty et al. have used model counting and simulated annealing to synthesize highly compact adders for flow-based computing [46], where more than 90% of the crossbar has been configured during the design process. An interesting direction of future work would be to design compact multipliers that configure and employ a large fraction of the available memristors on a crossbar. As the size of a crossbar increases, so does the requirement of memristors with higher HRS-LRS ratios, thus a smaller crossbar will not only require smaller area and power, it will also reduce the requirement of memristors with higher HRS-LRS ratios. A deeper theoretical investigation into the computational capability of flow-based computing on crossbars and the size of Boolean formula that can be computed on a memristor crossbar is merited.

LIST OF REFERENCES

[1] Leon Chua. Memristor – the missing circuit element. *IEEE Transactions on circuit theory*, 18(5):507–519, September 1971.

[2] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 2008.

[3] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC–Memristor-Aided Logic. *IEEE TCASII*, 61(11):895–899, Nov 2014.

[4] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta. Bdd based synthesis of boolean functions using memristors. In *2014 9th International Design and Test Symposium (IDT)*, pages 136–141, Dec 2014.

[5] S. Shirinzadeh, M. Soeken, P. Gaillardon, and R. Drechsler. Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 948–953, March 2016.

[6] A. U. Hassen. Automated synthesis of compact multiplier circuits for in-memory computing using ROBDDs. In *2017 NANOARCH*, pages 141–146. IEEE, July 1989.

[7] Peng Li, Weikang Qian, Marc Riedel, Kia Bazargan, and David Lilja. The synthesis of linear Finite State Machine-based Stochastic Computational Elements. pages 757–762, 01 2012.

[8] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[9] HsuWei-Wei Zhuang Sheng Teng. Electrically programmable resistance cross point memory, 3 2003. US Patent 6,531,371.

[10] In-Gyu Baek Moon-Sook Lee. Methods of programming non-volatile memory devices including transition metal oxide layer as data storage material layer and devices so operated, May 3 2006. US Patent 7,292,469.

[11] Moore Terry L. Gilton Kristy A., Campbell John T. Single-polarity programmable resistance-variable memory element, 3 2004. US Patent 6,867,996.

[12] G. Snider. Computing with hysteretic resistor crossbars. *Applied Physics A*, 80(6):1165–1172, Mar 2005.

[13] S. P. Adhikari, M. P. Sah, H. Kim, and L. O. Chua. Three Fingerprints of Memristor. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(11):3008–3021, Nov 2013.

[14] Sascha Vongehr and Xiangkang Meng. The missing memristor has not been found. In *Scientific reports*, 2015.

[15] Yuval Cassuto, Shahar Kvatinsky, and Eitan Yaakobi. Information-Theoretic Sneak-Path Mitigation in Memristor Crossbar Arrays. *IEEE Transactions on Information Theory*, 62(9):4801–4813, Sept 2016.

[16] Zahiruddin Alamgir, Karsten Beckmann, Nathaniel Cady, Alvaro Velasquez, and Sumit Kumar Jha. Flow-based Computing on Nanoscale Crossbars: Design and Implementation of Full Adders. In *ISCAS, 2016*, pages 1870–1873. IEEE, 2016.

[17] Sumit Kumar Jha, Dilia E Rodriguez, Joseph E Van Nostrand, and Alvaro Velasquez. Computation of boolean formulas using sneak paths in crossbar computing, April 19 2016. US Patent 9,319,047.

[18] B. R. Gaines. Stochastic Computing. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, 1967. ACM.

[19] B. D. Brown and H. C. Card. Stochastic neural computation. I. Computational elements. *IEEE Transactions on Computers*, 50(9):891–905, Sep. 2001.

[20] Armin Alaghi and John P. Hayes. Survey of Stochastic Computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013.

[21] Alaghi, Armin and Li, Cheng and Hayes, John P. Stochastic circuits for real-time image-processing applications. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 136:1–136:6, New York, NY, USA, 2013. ACM.

[22] M. H. Najafi and M. E. Salehi. A Fast Fault-Tolerant Architecture for Sauvola Local Image Thresholding Algorithm Using Stochastic Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):808–812, Feb 2016.

[23] HT Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics, 1979.

[24] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R. Stanley Williams. Dot-product Engine for Neuromorphic Computing: Programming 1T1M Crossbar to Accelerate Matrix-vector Multiplication. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 19:1–19:6, New York, NY, USA, 2016. ACM.

[25] A. Velasquez and S. K. Jha. Parallel boolean matrix multiplication in linear time using rectifying memristors. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1874–1877, May 2016.

[26] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *Proceedings of the 43rd*

International Symposium on Computer Architecture, ISCA '16, pages 14–26, Piscataway, NJ, USA, 2016. IEEE Press.

[27] Baogang Zhang, Necati Uysal, Deliang Fan, and Rickard Ewetz. Handling Stuck-at-faults in Memristor Crossbar Arrays Using Matrix Transformations. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ASPDAC '19, pages 438–443, New York, NY, USA, 2019. ACM.

[28] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, and E. G. Friedman. MRL — Memristor Ratioed Logic. In *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, pages 1–6, Aug 2012.

[29] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

[30] E. Lehtonen and M. Laiho. Stateful implication logic with memristors. In *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 33–36, July 2009.

[31] A. Chakraborty, R. Das, C. Bandopadhyay, and H. Rahaman. Bdd based synthesis technique for design of high-speed memristor based circuits. In *2016 20th International Symposium on VLSI Design and Test (VDAT)*, pages 1–6, May 2016.

[32] Hadi Owlia, Parviz Keshavarzi, and Abdalhossein Rezai. A novel digital logic implementation approach on nanocrossbar arrays using memristor-based multiplexers. *Microelectronics Journal*, 45(6):597 – 603, 2014.

[33] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-inverter graph: A new paradigm for logic optimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 35(5):806–819, May 2016.

[34] Dwaipayan Chakraborty and Sumit Kumar Jha. Automated synthesis of compact crossbars for sneak-path based in-memory computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 770–775. IEEE, March 2017.

[35] John Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies, Annals of Mathematics Studies*, pages 43–98, 01 1956.

[36] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, Dec 1938.

[37] W. J. Poppelbaum, C. Afuso, and J. W. Esch. Stochastic Computing Elements and Systems. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 635–644, New York, NY, USA, 1967. ACM.

[38] H Sebastian Seung. Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6):1063–1073, 2003.

[39] V. C. Gaudet and A. C. Rapley. Iterative decoding using stochastic computation. *Electronics Letters*, 39(3):299–301, Feb 2003.

[40] Weikang Qian and Marc D. Riedel. The Synthesis of Robust Polynomial Arithmetic with Stochastic Logic. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 648–653, New York, NY, USA, 2008. ACM.

[41] A. Alaghi and J. P. Hayes. STRAUSS: Spectral Transform Use in Stochastic Circuit Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1770–1783, Nov 2015.

[42] M Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[43] Luis Ceze, Mark D Hill, and Thomas F Wenisch. Arch2030: A Vision of Computer Architecture Research over the Next 15 Years. *arXiv preprint arXiv:1612.03182*, 2016.

[44] H-S Philip Wong and Sayeef Salahuddin. Memory leads the way to better computing. *Nature Nanotechnology*, 10(3):191–194, 2015.

[45] Dwaipayan Chakraborty, Sunny Raj, Julio Cesar Gutierrez, Troyle Thomas, and Sumit Kumar Jha. In-Memory Execution of Compute Kernels using Flow-based Memristive Crossbar Computing. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–6. IEEE, Nov 2017.

[46] Dwaipayan Chakraborty and Sumit Kumar Jha. Design of compact memristive in-memory computing systems using model counting. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, May 2017.

[47] Mohammed Affan Zidan, Hossam Aly Hassan Fahmy, Muhammad Mustafa Hussain, and Khaled Nabil Salama. Memristor-based memory: The sneak paths problem and solutions. *Microelectronics Journal*, 44(2):176–183, 2013.

[48] Chang Yeong Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, July 1959.

[49] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[50] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, Aug 1986.

[51] Rune M. Jensen. *A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems*. School of Computer Science, Carnegie Mellon University, 2002.

[52] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE, Jun 1990.

[53] Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0. *University of Colorado at Boulder*, 2015.

[54] Kim Milvang-Jensen and Alan J Hu. BDDNOW: A parallel BDD package. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, FMCAD 98, pages 501–507. Springer, 1998.

[55] Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*, volume 4 of *Monographs on Discrete Mathematics and Applications*. SIAM, 2000.

[56] A. Bricalli, E. Ambrosi, M. Laudato, M. Maestro, R. Rodriguez, and D. Ielmini. Resistive Switching Device Technology Based on Silicon Oxide for Improved ON-OFF Ratio–Part II: Select Devices. *IEEE Transactions on Electron Devices*, 65(1):122–128, Jan 2018.

[57] Shahar Kvatinsky, Guy Satat, Nimrod Wald, E.G. Friedman, Avinoam Kolodny, and Uri C. Weiser. Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies. *IEEE Transactions on (VLSI) Systems*, 22(10):2054–2066, Oct 2014.

[58] Eero Lehtonen, JH Poikonen, and Mika Laiho. Two memristors suffice to compute all Boolean functions. *Electronics letters*, 46(3):239–240, 2010.

[59] Dwaipayan Chakraborty, Sunny Raj, and Sumit Kumar Jha. A compact 8-bit adder design using in-memory memristive computing: Towards solving the Feynman grand prize challenge. In *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 67–72. IEEE, July 2017.

[60] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology*, 15:1–1, 07 2016.

[61] C. Jung, J. Choi, and K. Min. Two-Step Write Scheme for Reducing Sneak-Path Leakage in Complementary Memristor Array. *IEEE Transactions on Nanotechnology*, 11(3):611–618, May 2012.

[62] A. Ul Hassen, D. Chakraborty, and S. K. Jha. Free Binary Decision Diagram-Based Synthesis of Compact Crossbars for In-Memory Computing. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):622–626, May 2018.

[63] A. U. Hassen, S. Anwar Khokhar, H. A. Butt, and S. Kumar Jha. Free BDD based CAD of Compact Memristor Crossbars for in-Memory Computing. In *2018 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 1–7, July 2018.

[64] A. U. Hassen, S. A. Khokhar, and B. Amin. Synthesis of Compact Crossbars for in-Memory Computing using Dynamic FBDDs. In *2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO)*, pages 1–4, July 2018.

[65] Advanced Low Power Spintronic Memories beyond STT-MRAM, author=Kang, Wang and Wang, Zhaohao and Zhang, He and Li, Sai and Zhang, Youguang and Zhao, Weisheng, booktitle=Proceedings of the on Great Lakes Symposium on VLSI 2017, series = GLSVLSI '17, pages=299–304, year=2017, organization=ACM.

[66] Kosuke Suzuki and Steven Swanson. A survey of trends in non-volatile memory technologies: 2000-2014. In *2015 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, May 2015.

[67] John Von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components.

[68] A. U. Hassen, B. Chandrasekar, and S. K. Jha. Automated synthesis of stochastic computational elements using decision procedures. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1678–1681, May 2016.

[69] S. Khokhar and A. Hassen and. Synthesis of Approximate Logic On Memristive Crossbars. In *2019 IEEE 17th IEEE International NEWCAS Conference (NEWCAS)*, 2019.

[70] A. Hassen and S. Khokhar. Approximate in-Memory Computing on ReRAM Crossbars. In *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2019.