NOVEL LOGIC SYNTHESIS TECHNIQUES FOR ASYMMETRIC LOGIC FUNCTIONS BASED ON SPINTRONIC AND MEMRISTIVE DEVICES

by

Vaibhav Vyas

APPROVED BY SUPERVISORY COMMITTEE:

Joseph S. Friedman, Chair

Mehrdad Nourani

Carl Sechen

William Swartz

This thesis document

is dedicated to my parents,

Ruchi and Rakesh Vyas.

NOVEL LOGIC SYNTHESIS TECHNIQUES FOR ASYMMETRIC LOGIC FUNCTIONS BASED ON SPINTRONIC AND MEMRISTIVE DEVICES

by

VAIBHAV VYAS, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

May 2018

ACKNOWLEDGMENTS

It has been a privilege to be a part of a highly creative, intellectual and productive research group at the NanoSpinCompute Lab. Not only has it contributed to polishing my technical skills and understanding, it has helped me to harness my creativity and problem solving capabilities to their fullest extent. Undoubtedly, I owe gratitude to the people who have made this possible.

I am highly grateful to my teacher, research advisor and mentor Dr. Joseph S. Friedman, for without him, none of this would have been possible. I couldn't have worked with a more intellectual, vibrant and understanding supervisor than him. He was quick to spot my interests and channelize my efforts in the most productive way possible. I am thankful to him for his encouragement and support in times of uncertainty, and for constantly making me realize my research's worth. Finally, I am grateful for his honest feedbacks that helped me stay on course.

Further, I would like to express my sincere gratitude to my supervisory panel - Dr. Carl Sechen, Dr. Mehrdad Nourani and Dr. William Swartz for taking out the time to incorporate valuable suggestions into my work. In the past two years, I have had the opportunity to work with, and learn from the brightest research scholars at UT Dallas. I thank Xuan Hu for being there as a mentor since the zeroth day, listening and solving all my problems patiently. In addition, it has been a complete pleasure working with one of the brightest minds of Bangladesh, Naimul Hassan. A sincere thank you to him for sharing his knowledge and always helping me out as a brother. It is hard to imagine our research group without these two guys. I would also like to thank Lucian Jiang-Wei for being the first point of consultation and support throughout my research. A special mention to Matthew Joslin and Ashish Ganesh Pai, who were not only good colleagues, but loyal friends. My heartfelt acknowledgments to all the current and former members of the lab.

A big thank you to my roommate, friend and colleague - Parth Mishra. He has always been a strong source of support and motivation since the day I started my graduate journey. I am fortunate to have had the most amazing friends - Rohit Khanna, Sarthak Gajjar, Divyansh Sharma, Archit Bagla, Swapnil Dolui, Abhishek Mishra, Utkarsh Gandhi, S.M. Rehan and Faisal Farooqui. Thank you so much for the unforgettable memories. I cannot thank Derek and Honey Sutt enough for their selfless help and loving friendship.

Most importantly, I am thankful to my Mummy and Papa for their unconditional love and selfless care. Thank you for placing your unreserved trust in me and always encouraging me to fulfill my ambitions. I am lucky to have a caring brother like Saurabh Bhaiyya and a loving sister-in-law like Shikha Bhabhi. I know you guys have always been, and will always be there for me. Thank you Sameeksha for being my best friend and a constant source of support. Finally, nothing would have been possible without the blessings of my grandparents and all my family members.

My sincere apologies to everyone I might have hurt, intentionally or otherwise.

April 2018

NOVEL LOGIC SYNTHESIS TECHNIQUES FOR ASYMMETRIC LOGIC FUNCTIONS BASED ON SPINTRONIC AND MEMRISTIVE DEVICES

Vaibhav Vyas, MSEE
The University of Texas at Dallas, 2018

Supervising Professor: Joseph S. Friedman, Chair

The development of beyond-CMOS technologies with alternative basis logic functions necessitates the introduction of novel design automation techniques. In particular, recently proposed computing systems based on memristors and bilayer avalanche spin-diodes both provide asymmetric logic functions as basis logic gates - the implication and inverted-input AND, respectively.

There has been a considerable amount of work done in the field of logic synthesis using alternative logic sets, especially stateful memristive implication logic. However, most of the previous works rely on the mapping of these alternative logic functions on to standard ones like NAND, NOR, AND, and OR gates respectively.

This work points out the possible overheads of such an approach, and the advantages of using asymmetric logic functions to directly implement circuits, which calls for suitable synthesis and optimization techniques, tailored specifically to asymmetric logic functions. Such techniques are rooted in the enablement of Boolean reduction methods without any translation to standard logic operators. This is made possible by the proposed set of Boolean identities and principles, and a modified Karnaugh mapping method that can be directly applied to systems with asymmetric logic functions as the basic logic sets.

A comparative study is presented, which highlights the statistical improvements over previously proposed approaches in terms of the total number of devices used to implement a standard function. Finally, a basic algorithm for the automated optimization of asymmetric functions is proposed, providing the groundwork for advanced design automation techniques for emerging device technologies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

For many years now, the continued scaling of CMOS devices has been the primary intent of both the academia and the industry alike. However, more recent works have imposed limitations on the extent of this scaling, accelerating the race towards finding alternative technologies. Recent years have witnessed several promising proposals such as the magnetic tunnel junction logic [1, 2], domain wall logic [1, 3–6], all-carbon spin logic [7], spin-FETs [8–11] that use magnets and electron spin as state variables instead of charge currents, and spin based logic [12–18].

As newer devices are being proposed, there is a need to develop better alternative methods to handle circuit design at the logic level. This entails the introduction of novel logic synthesis and minimization techniques. One approach of doing this is to optimize the existing methods for these novel techniques. The advantage here is lower resource allocation to research and development. The second approach is to rethink the entire process from the ground up. Surely, this requires a more comprehensive development process, but promises optimality at each step. The latter approach is the one adopted in this work, which in fact shows a marked improvement over the former.

The first step of implementing logic efficiently is to characterize a logic set using appropriate logic operations such that it can implement any given Boolean function. The logic operations are usually termed as basic gates for a given device technology. For example, NAND and NOR gates are considered to be the basic gates for the CMOS technology. In this work, two separate device technologies are used - bilayer avalanche spin-diode logic (BASDL) device and the memristor. Each provides a unique basic logic set - an inverted-input AND (IAND) and OR gate for the BASDL, and stateful implication (IMPLY) and NAND logic for the memristors. Either of these logic sets can be used to implement any given Boolean function. In this thesis document, I define an entire set of Boolean identities

and principles for both the two logic sets, which enables their complete algebraic treatment. This means that a mapping to any other CMOS-based 'universal gate' is not required.

After the characteristics of a required circuit are comprehensively defined, and is correctly mapped to the appropriate logic operations, the resultant Boolean function needs to be minimized. Logic minimization is a design automation technique through which the area, speed, and energy of logic circuits can be optimized by minimizing the quantity and complexity of the logic gates required to perform a desired logic function. Numerous techniques have been proposed for the optimization of logic circuits, many of which are based on Maurice Karnaugh's 1953 proposal for a map method for the synthesis of logic circuits [19]. While Karnaugh maps have found their greatest utility for circuits composed of complementary metal oxide semiconductor (CMOS) transistors, much of Karnaugh's scientific contributions related to logic with magnetic cores [20].

As logic circuits that act on magnetic and alternative phenomena have again become competitors to CMOS, modifications of Karnaugh's map method are necessary to take full advantage of emerging technologies. Karnaugh's map method enables logic synthesis based on the AND and OR functions, while various logic concepts based on memristors and spintronic devices provide basis logic gates for which there is no efficient technique for translation to AND and OR gates [12, 21–28]. In particular, the IMPLY and IAND functions are non-commutative and asymmetric, preventing direct use of conventional Karnaugh maps. Therefore, I propose modifications to the Karnaugh's map method that enable the synthesis of complex logic functions directly into minimized stateful memristor logic circuits and bilayer avalanche spin-diode logic circuits.

Logic synthesis for stateful memristor logic has previously been investigated, but has not provided a minimization technique that directly applies implication functions. In much previous work, logic minimization is performed with a basis set of conventional logic functions, with each basis logic gate mapped to an efficient memristor implementation. For example,

[29] minimizes a large function into NAND, OR, and parity gates, which are then realized with memristors; [30] minimizes a function into OR and inverter gates; [31] uses NOT, NAND, and OR gates; and [32] uses majority gates. Binary decision diagrams have also been used [33], as has an interpretation of memristors as threshold logic elements [34]. Most of the previous work has been performed at a purely algorithmic level, while the synthesis technique in [31] uses conventional Karnaugh maps that are then translated into memristor logic circuits. As these techniques all minimize circuits with conventional symmetric logic functions and then implement these circuits with memristors, the circuit that is realized is sub-optimal due to the fact that the minimization function is not applied directly to the asymmetric basis logic functions performed by the memristors. Though helpful in significantly reducing circuit complexity, none of this previous work provides a technique for realizing a fully minimized circuit with asymmetric logic functions.

This work is intended to propose an alternative strategy towards logic implementation using unconventional logic sets, and hopefully contribute towards developing better electronic design automation (EDA) techniques for these functions. Chapter 2 describes the background for the device technologies, conventional Karnaugh mapping method, and asymmetry in logic functions; while Chapter 3 presents Boolean algebraic laws for both IAND and IMPLY logic operations. Chapter 4 details the Karnaugh maps optimized for asymmetric logic functions and examples are provided to enable its practical use. Chapter 5 provides a study of different logic implementations styles proposed over the years, and compares them with the ones proposed in this work. Chapter 6 proposes a practical implementation of an algorithm that automates the optimization process for asymmetric logic functions. Finally, conclusions are presented in Chapter 7.

CHAPTER 2

BACKGROUND

The asymmetric basis logic functions provided by stateful memristor logic and bilayer avalanche spin-diode logic can be performed compactly, but their synthesis into optimal circuits requires the development of techniques tailored to these functions. In particular, the implication function can be performed by two non-volatile memristors, while a NAND function can be performed by three memristors. In bilayer avalanche spin-diode logic, a single device can perform the IAND and OR functions. The implication and IAND functions are both non-commutative and asymmetric, and their integration with NAND and OR functions, respectively, enables the development of an algebra and Karnaugh map method that enables logic minimization for both stateful memristor logic and bilayer avalanche spin-diode logic. This chapter sheds light on the device technologies, their corresponding basis logic sets, conventional Karnaugh mapping method and the previously proposed approaches for logic implementations using asymmetric gates.

2.1 Bilayer Avalanche Spin Diode Logic

In bilayer avalanche spin-diode logic, the current on two control wires modulates the current through a spin-diode [12]. These two-terminal spintronic devices have negative magnetoresistance, enabling an applied magnetic field to modulate the resistance. A constant voltage is applied across all spin-diodes at all times, thus enabling the two control wire input currents to create magnetic fields that modulate the spin-diode output current (see Fig. 2.1). The magnitude and direction of the magnetic fields relative to a threshold field determine the resistance state of the spin-diode. The spin-diode output currents can be used as the input control wire current to create directly cascaded logic without any amplification or control circuitry.

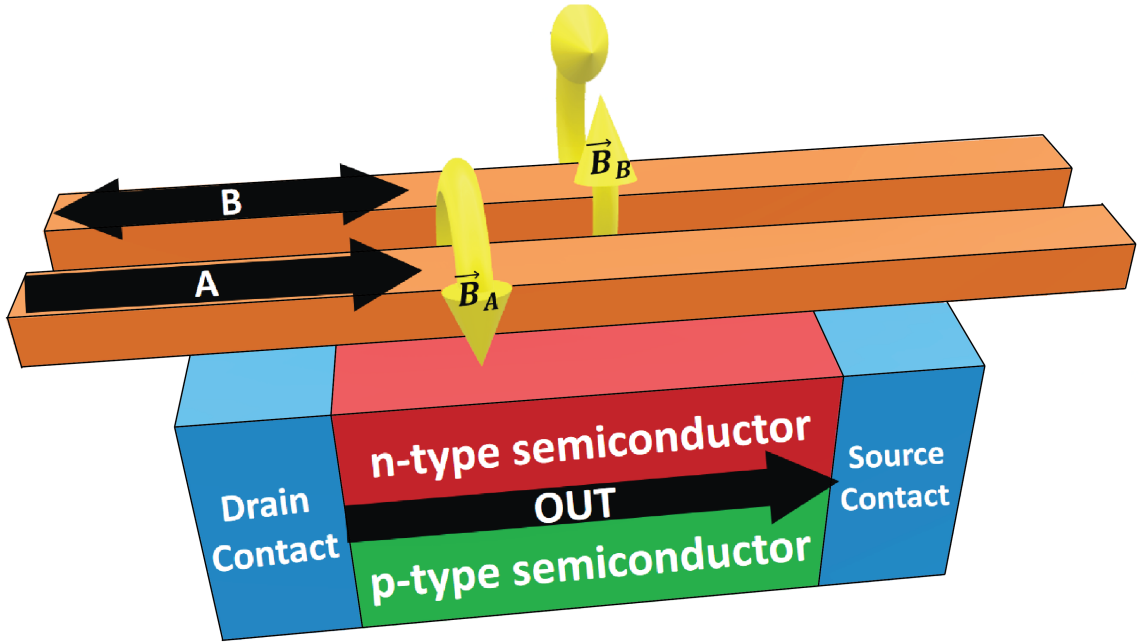

Figure 2.1. Bilayer avalanche spin-diode, where the magnetic fields due to input currents A and B modulate the output current.

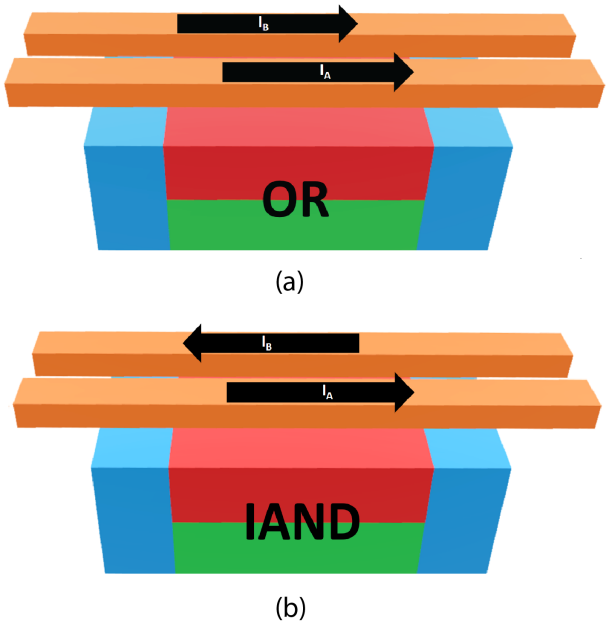

Figure 2.2. BASDL functioning as an (a) OR gate if the input currents I_A and I_B are in the same direction, or as (b) IAND gate if I_A and I_B have opposite current flows.

In this spintronic logic family, a $'1'$ is represented as a large current while a $'0'$ is represented by a small current. Depending upon the relative direction of current through the control wires, this spin-diode performs either the conventional OR function or the inverted-input AND (IAND) function (see Fig. 2.2 and Table 2.1). These distinct functions result from the additive or counteractive magnetic fields created by currents oriented in the same or opposite directions, respectively. Unlike memristors, spin-diodes are volatile; they return to their zero-magnetic field state immediately upon the removal of the applied input currents.

2.2 Stateful Memristor Logic

A memristor (or, more generally, a memristive device) is a two-terminal non-volatile device with a resistance that can be modified through application of a voltage across the two terminals [35, 36]. In general, the resistance is on a spectrum determined by the history of applied voltages; in the ideal binary case, applied voltages above a threshold magnitude

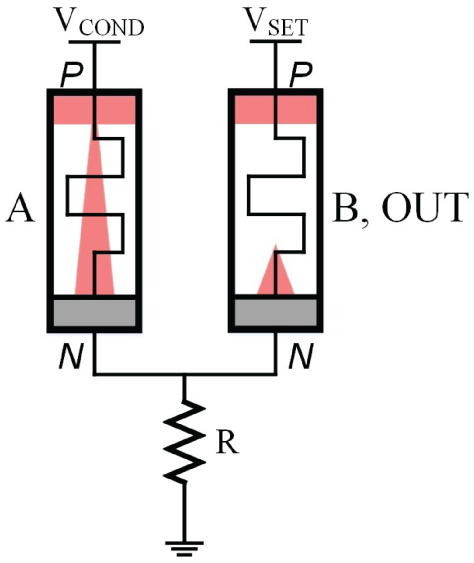

Figure 2.3. Schematic of memristive implication logic, where voltages applied to the memristors modulate the resistance state.

switch a memristor between purely resistive and conductive states. The memristors shown in Fig. 2.3 switch to a '1' conductive state when $V_P - V_N > V_{TH}$, and a resistive $'0'$ state when $V_N - V_P > V_{TH}$, where V_N is the voltage at node N, V_P is the voltage at node P, and V_{TH} is the threshold voltage. In some physical implementations, the resistance state is a result of the growth and retraction of a metallic filament [37].

As shown in Table 2.1, the implication function $OUT = A \rightarrow B$ is performed by applying V_{COND} to memristor A and V_{SET} to memristor B, where $V_{COND} < V_{TH} < V_{SET}$ and $V_{SET} - V_{COND} < V_{TH}$ [38]. When memristor A is in the resistive $'0'$ state, the V_{SET} voltage across memristor B is greater than V_{TH}, causing memristor B to switch to the conductive $'1'$ state or remain in that state. If memristor A is in the conductive $'1'$ state, the voltage across memristor B is $V_{SET} - V_{COND}$; as this is less than V_{TH}, no switching occurs. This implication function can thus be performed by two memristors in a single step, while a NAND function can be performed similarly with three memristors (Fig. 2.4) in two steps [25].

Note that while more complex operations have been proposed in a single step with ideal memristors, these operations have not been demonstrated experimentally or with physically

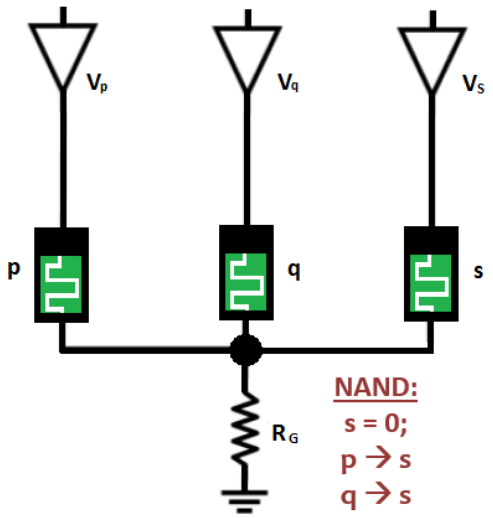

Figure 2.4. Schematic and computational steps for a typical NAND implementation using stateful memristive implication logic.

Table 2.1. Truth Table for IAND and Implication Logic

Input A	Input B	IAND	Implication
0	0	0	1
0	1	0	1
1	0	1	0
1	1	0	1

realistic device models. This work is therefore restricted to experimentally realizable operations with only two memristors in a single step.

2.3 Asymmetric Basis Logic Functions

An asymmetric logic operation can be defined as one whose logic value changes when the operands are interchanged. Equivalently, these operations can be regarded as 'non-commutative' in nature. The inverted-AND (IAND) gate and material implication implemented using the bilayer avalanche spin-diode logic and memristors respectively, are two such asymmetric functions discussed in this paper.

As the IAND gate performs the function of an AND gate with one inverted input (Table 2.1), a symbol ($\wedge\!\!\!\!-$) is defined for the IAND operation such that

$$IAND\,(A,B) = A \wedge \overline{B} = A \wedge\!\!\!\!- \; B \tag{2.1}$$

The symbol derives inspiration from the conventional logical operator for AND gates ($\wedge$), with the added underbar on the right arm indicating the inversion of the input to the right of the operator.

The implication function is the inverse of IAND and is defined as

$$IMP\,(A,B) = A \rightarrow B = \overline{A} \vee B = \overline{A \wedge\!\!\!\!- \; B}. \tag{2.2}$$

For clarity, the input that lies on the left side (right side) of the IAND (implication) operation is referred to as the *non-inverted* input, whereas the one of the right (left) is referred to as the *inverted* input. For example, A is the non-inverted input and B is the inverted input in (2.1); in (2.2), B and A are the non-inverted and inverted inputs, respectively.

Both technologies described in this work are limited to a two-input configuration. When more than two operands are present, only two literals should be operated upon at a time. Moreover, because the operations are non-commutative, it is important to pay attention to the order of operations. The order of operations can be summarized as:

- Order of operations for IAND: Two at a time, from left to right.

$$IAND\,(A,B,C) = (A \wedge\!\!\!\!- \; B) \wedge\!\!\!\!- \; C \tag{2.3}$$

- Order of operations for IMPLY: Two at a time, from right to left.

$$IMP\,(A,B,C) = (A \rightarrow (B \rightarrow C)) \tag{2.4}$$

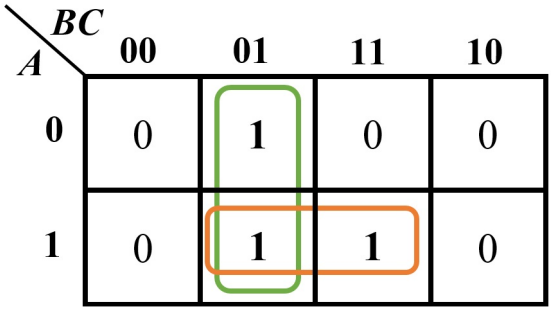

Figure 2.5. Karnaugh map for (2.5) and (4.1).

Further, since these operations are sensitive to the order of the operands, it is convenient to define a *pre-* and *post*-operation. For example, a pre-IAND operation of A with B is equivalent to $(A \wedge B)$, whereas a post-IAND of A with B means $(B \wedge A)$. A similar analogy follows for IMPLY operations as well.

2.4 Conventional Karnaugh Maps: A Functional Overview

Karnaugh Maps are tabular representations of Boolean logic functions consisting of 2^n cells, each for one among the possible combinations of n-bit binary data. The cells are arranged such that logically adjacent terms share physical adjacency. Graphical pairing of these adjacent terms reduces the function to its essential prime implicants [19, 39]. Whereas Karnaugh originally described the method only through examples [19], I intend to formally demonstrate the validity of the proposed method.

A conventional Karnaugh map is shown in Fig. 2.5 for the expression

$$f_{sop} = (\overline{A} \wedge \overline{B} \wedge C) \vee (A \wedge \overline{B} \wedge C) \vee (A \wedge B \wedge C) \tag{2.5}$$

This expression can be rewritten by duplicating the term $(A \wedge \overline{B} \wedge C)$,

$$f_{sop} = \{(\overline{A} \wedge \overline{B} \wedge C) \vee (A \wedge \overline{B} \wedge C)\} \vee \{(A \wedge \overline{B} \wedge C) \vee (A \wedge B \wedge C)\}. \tag{2.6}$$

This expression can then be simplified according to conventional Boolean algebra techniques, first with the OR distributive law:

$$f_{sop} = ((A \vee \overline{A}) \wedge \overline{B} \wedge C) \vee ((B \vee \overline{B}) \wedge A \wedge C). \tag{2.7}$$

$(A \vee \overline{A})$ and $(B \vee \overline{B})$ are $'1'$ by complement law, enabling the elimination of operands A and B from the respective product terms:

$$f_{sop} = (1 \wedge \overline{B} \wedge C) \vee (1 \wedge A \wedge C) \; = \; (\overline{B} \wedge C) \vee (A \wedge C). \tag{2.8}$$

Karnaugh's method reaches this result in a similar manner, but graphically. This can be noted in the Karnaugh map pair encircled with green and orange (Fig. 2.5): The logical adjacency enables the combination of literals with their complements and therefore minimize redundancies in a function. In chapter 4, this functional overview will help the reader understand the reason behind the correctness of the proposed modification in the Karnaugh mapping method.

2.5 Previously Proposed Approaches for Logic Implementation

With the growing and seemingly promising market of emerging device technologies, multiple schemes for improving their usability have been proposed in recent years. The primary objective of many of these papers have been to propose an efficient logic synthesis paradigm for these relatively newfound devices. Memristor is one such device that has received much attention in particular. From material implication to hybrid memristor-CMOS devices, there is a variety of proposed implementation techniques. However, it is not clear which one promises the most optimized computing capabilities.

Although, a detailed discussion over all of these techniques is beyond the scope of this work, [40] provides an appropriate classification:

- Material Implication: Logic implemented using stateful implication logic, as described in this work [25, 26, 29, 38].
- Memristor-only: Logic implementation using memristors only [41–43]
- Programmable Nanowire Interconnects [36, 44–47].
- Network-Based Computations: Parallel computations using a network of memristors [48–51]
- Hybrid CMOS-Memristor Logic: Families like MeMOS (Memristor-CMOS) [52] utilize the both memristive and CMOS circuits in the same implementation with either Boolean [53, 54] or threshold logic [55–57].

This thesis document takes into account only synthesis techniques using material implication and memristor-only implementations, since comparing the others with this work might turn out to be both unfair and complex at the same time.

CHAPTER 3

BOOLEAN ALGEBRA FOR ASYMMETRIC LOGIC FUNCTIONS

The inherent inverting nature of asymmetric logic functions like IAND and IMPLY make it an imperative to redefine the Boolean identities and principles. In this chapter, I have organized the properties in analogy to the standard Boolean algebra, hopefully making it more intuitive. A separate section is dedicated to each of the spintronic IAND logic and the stateful memristive implication logic. Each of the sections is subdivided into some core algebraic identities and the standard Boolean laws. This is followed by a detailed explanation about canonical forms for functions implemented using the basic logic set for each of the device technology. At the end, I present the relationship between IAND and IMPLY operations and describe a process of translation between them.

3.1 Spintronic Inverted-AND Logic

The bilayer avalanche spin-diode logic device provides for a unique basic logic gate - IAND, which is an AND gate with an inverted input. In this section, I define the Boolean laws and principles for functions implemented using a combination of IAND gates and other standard Boolean functions.

3.1.1 Core Algebraic Identities

Core algebraic identities define the basic properties of a logic operation, and how it fundamentally alters a function. The primary identities have been presented in this section.

Interaction with High (1) and Low Logic (0)

Here, I evaluate how the IAND operation alters a Boolean function when operated against a zero (0) or a one (1).

Identity 1 (IAND Annulment)**:** A pre-IAND of 0 or a post-IAND of 1 nullifies the given function.

$$A \not\wedge 1 \;=\; 0 \not\wedge \overline{A} \;=\; 0 \tag{3.1}$$

Proof. Representing the IAND operation in terms of AND,

$$A \not\wedge 1 \;=\; A \wedge \overline{1} \;=\; A \wedge 0 \;=\; 0 \tag{3.2}$$

$$\textit{and}, \quad 0 \not\wedge A \;=\; 0 \wedge \overline{A} \;=\; 0 \tag{3.3}$$

which proves the theorem. ■

Identity 2 (IAND Inversion)**:** A pre-IAND of 1 complements the given function.

$$1 \not\wedge A \;=\; \overline{A} \tag{3.4}$$

Proof. Representing the IAND operation in terms of AND,

$$1 \not\wedge A \;=\; 1 \wedge \overline{A} \;=\; \overline{A} \tag{3.5}$$

which proves the theorem. ■

Identity 3 (IAND Identity)**:** A post-IAND of 0 is the identity for the IAND operation.

$$A \not\wedge 0 \;=\; A \tag{3.6}$$

Proof. Representing the IAND operation in terms of AND,

$$A \not\wedge 0 \;=\; A \wedge \overline{0} \;=\; A \wedge 1 \;=\; A \tag{3.7}$$

which proves the theorem. ■

Idempotency for IANDs

Idempotency is the property that enables multiple iterations of an operation without changing the result. However, for asymmetric functions, this property is evaluated for the different cases of repeating operations even though they might not preserve the same value. I have classified these properties as per the results obtained.

Identity 4 (IAND Null Idempotency)**:**

$$A \barwedge A = 0 \tag{3.8}$$

Proof. Representing the IAND operation in terms of AND,

$$A \barwedge A = A \wedge \overline{A} = 0 \tag{3.9}$$

which proves the theorem. ■

Identity 5 (IAND Inverse Idempotency - I)**:**

$$A \barwedge \overline{A} = A \tag{3.10}$$

Proof. Representing the IAND operation in terms of AND,

$$A \barwedge \overline{A} = A \wedge A = A \tag{3.11}$$

which proves the theorem. ■

Identity 6 (IAND Inverse-Idempotency - II)**:**

$$\overline{A} \barwedge A = \overline{A} \tag{3.12}$$

Proof. Representing the IAND operation in terms of AND,

$$\overline{A} \barwedge A \;=\; \overline{A} \wedge \overline{A} \;=\; \overline{A} \tag{3.13}$$

which proves the theorem. ■

3.1.2 Boolean Algebraic Laws

This section develops the standard laws of Boolean algebra for IAND logic. As a generic note, some of the laws might deviate from their original definition, however a clear analogy can be noted for each of them.

Commutative Law

Theorem 1 (IAND Commutation)**:** The very idea that a logic function is 'asymmetric' implies that it does not follow the conventional commutative law, *i.e.*

$$A \barwedge B \;\neq\; B \barwedge A \tag{3.14}$$

However, considering two operands A and B, commutation can be achieved by complementing both the literals as shown in (3.15)

$$A \barwedge B \;=\; \overline{B} \barwedge \overline{A} \tag{3.15}$$

Proof. Replacing the IAND with AND,

$$A \barwedge B \;=\; A \wedge \overline{B} \;=\; \overline{B} \wedge A. \tag{3.16}$$

Changing the RHS of (3.16) back to IAND notation,

$$A \barwedge B \;=\; \overline{B} \barwedge \overline{A}, \tag{3.17}$$

which proves the theorem. ■

Associativity Laws

Here, I present associativity laws that define how three or more Boolean functions are grouped when operated upon by an asymmetric function.

Theorem 2 (Conventional 'Non-Associativity')**:**

$$(A \barwedge B) \barwedge C \;\neq\; A \barwedge (B \barwedge C) \tag{3.18}$$

It is important to keep in mind the fact that due to the asymmetric inversion of operands, IANDs do not follow associativity conventionally. This can be proved as follows. Note that the parentheses are solved two at a time from left to right as suggested earlier.

Proof. Converting the two sides of Theorem 2 to AND notation,

$$(A \barwedge B) \barwedge C = A \wedge \overline{B} \wedge \overline{C} \tag{3.19}$$

$$A \barwedge (B \barwedge C) = A \barwedge (B \wedge \overline{C}) = A \wedge (\overline{B} \vee C) \tag{3.20}$$

Clearly, (3.19) and (3.20) are not equivalent. ■

However, IAND logic does follow two special kinds of associative laws: Inverting and Non-Inverting. We explore these in detail in the following two theorems.

Theorem 3 (IAND Non-Inverting Associativity)**:** Non-inverting associativity means that on changing the order of certain operands, no inversion is required. This happens when any two inverted operands interchange places.

$$(A \barwedge B) \barwedge C \;=\; (A \barwedge C) \barwedge B \tag{3.21}$$

Proof. Converting the expression to AND notation:

$$(A \barwedge B) \barwedge C \;=\; A \wedge \overline{B} \wedge \overline{C} \;=\; A \wedge \overline{C} \wedge \overline{B}. \tag{3.22}$$

This can be rewritten using IAND notation as follows:

$$A \wedge \overline{C} \wedge \overline{B} \;=\; (A \barwedge C) \barwedge B. \tag{3.23}$$

The theorem is thus proven. ■

Theorem 4 (IAND Inverting Associativity)**:** Inverting associativity demands the inversion of both the operands that have changed places. This theorem should be applied when a non-inverted input interchanges its place with an inverted input.

$$(A \barwedge B) \barwedge C \;=\; (\overline{C} \barwedge \overline{A}) \barwedge B \tag{3.24}$$

Proof. Converting the LHS of (3.24) to AND notation:

$$(A \barwedge B) \barwedge C \;=\; A \wedge \overline{B} \wedge \overline{C}. \tag{3.25}$$

Rearranging the inputs on the RHS of the above expression,

$$(A \barwedge B) \barwedge C \;=\; \overline{C} \wedge A \wedge \overline{B}, \tag{3.26}$$

which can be rewritten using IAND notation as

$$(A \barwedge B) \barwedge C \;=\; (\overline{C} \barwedge \overline{A}) \barwedge B, \tag{3.27}$$

which is the same as (3.24). ■

Discussion: Theorem 3 and Theorem 4 can also be interpreted intuitively as the movement of inverted and non-inverted operands around the IAND operator:

- If the non-inverted operand trades places with an inverted operand within the IAND expression, both of these operands are complemented to maintain logical equivalence (inverting associativity).
- If an inverted operand trades places with another inverted operand within the IAND expression, the operands are not complemented (non-inverting associativity).

The above results also follow from Theorem 1 for two inputs.

Distributive Laws

This section detail the distributive laws for expressions involving a combination of IAND and OR/AND operations performed between three binary inputs. As the nomenclature would be quite challenging, these theorems are numbered rather than named. Further, please note that although some of these theorems might be more useful or relevant than others, all of the possible combinations for the three Boolean operations have been presented for completeness. However, these have been categorized into 'single operation' and 'cross-operation' distributive laws.

Single-Operation Distributive Laws: These laws relate two Boolean expressions, both of which are represented using a combination of IAND and AND/OR operations. Theorems 5-9 belong to this category.

Theorem 5 (IAND Distributive Law - I)**:**

$$A \barwedge (B \wedge C) = (A \barwedge B) \vee (A \barwedge C). \tag{3.28}$$

Proof. Changing LHS of (3.28) to AND notation and expanding using De Morgan's Law,

$$A \barwedge (B \wedge C) = A \wedge \overline{B \wedge C} = A \wedge (\overline{B} \vee \overline{C}). \tag{3.29}$$

Using the conventional OR distributive law,

$$A \barwedge (B \wedge C) = (A \wedge \overline{B}) \vee (A \wedge \overline{C}). \tag{3.30}$$

Finally, replacing the AND operations with IAND:

$$A \barwedge (B \wedge C) = (A \barwedge B) \vee (A \barwedge C), \tag{3.31}$$

which is the same as (3.28). ■

Theorem 6 (IAND Distributive Law - II)**:**

$$(A \barwedge B) \wedge C = (A \barwedge B) \barwedge \overline{C} \tag{3.32}$$

Proof. The proof for this theorem follows directly from the definition of the IAND operation. Changing the LHS of (3.32) to AND notation and subsequently back to IAND:

$$(A \barwedge B) \wedge C = A \wedge \overline{B} \wedge C \tag{3.33}$$

$$(A \barwedge B) \wedge C = (A \barwedge B) \barwedge \overline{C} \tag{3.34}$$

The above equation being the same as (3.32) ■

Theorem 7 (IAND Distributive Law - III)**:**

$$(A \vee B) \barwedge C \;=\; (A \barwedge C) \;\vee\; (B \barwedge C) \tag{3.35}$$

Proof. Changing LHS of (3.35) to AND notation and using the conventional OR distributive law,

$$(A \vee B) \barwedge C \;=\; (A \vee B) \wedge \overline{C} \;=\; (A \wedge \overline{C}) \;\vee\; (B \wedge \overline{C}). \tag{3.36}$$

Finally, replacing the AND operations with IAND:

$$A \barwedge (B \;\wedge\; C) \;=\; (A \barwedge C) \;\vee\; (B \barwedge C), \tag{3.37}$$

which is the same as (3.35). ■

Theorem 8 (IAND Distributive Law - IV)**:**

$$A \barwedge (B \vee C) \;=\; (A \barwedge B) \wedge (A \barwedge C) \tag{3.38}$$

Proof. Changing LHS of (3.38) to AND notation and expanding using De Morgan's Law,

$$A \barwedge (B \;\vee\; C) \;=\; A \wedge \overline{(B \;\vee\; C)} \;=\; A \wedge (\overline{B} \;\wedge\; \overline{C}). \tag{3.39}$$

This can be rewritten in IAND notation as

$$A \wedge (\overline{B} \;\wedge\; \overline{C}) \;=\; (A \wedge \overline{B}) \wedge (A \wedge \overline{C}) \;=\; (A \barwedge B) \wedge (A \barwedge C), \tag{3.40}$$

which is the same as (3.38). ■

Theorem 9 (IAND Distributive Law - V)**:**

$$A \wedge (B \barwedge C) \;=\; (A \wedge B) \barwedge C \;=\; (A \barwedge \overline{B}) \barwedge C \tag{3.41}$$

Proof. Similar to Theorem 6 the proof follows directly from the definition of the IAND operation. Changing the LHS of (3.41) to AND notation,

$$A \wedge (B \barwedge C) \;=\; A \wedge B \wedge \overline{C}, \tag{3.42}$$

$$\textit{and}, \quad (A \wedge B) \barwedge C \;=\; A \wedge B \wedge \overline{C}, \tag{3.43}$$

Changing (3.42) and (3.43) back to the IAND notation,

$$A \wedge (B \barwedge C) \;=\; (A \wedge B) \barwedge C \;=\; (A \barwedge \overline{B}) \barwedge C. \tag{3.44}$$

The above equation is the same as (3.41) ■

Cross-Operation Distributive Laws: These laws relate a Boolean expression represented in terms of IAND and AND/OR operations with an expression represented only in terms of AND/OR operations. Theorems 10-11 are cross-operation distributive laws.

Theorem 10 (IAND Distributive Law - VI)**:**

$$A \vee (B \barwedge C) \;=\; (A \vee B) \wedge (A \vee \overline{C}) \tag{3.45}$$

Proof. Changing the LHS of (3.45) to AND notation and using the conventional AND distribution law,

$$A \vee (B \barwedge C) \;=\; A \vee (B \wedge \overline{C}) \;=\; (A \vee B) \wedge (A \vee \overline{C}). \tag{3.46}$$

The above equation validates the theorem. ■

Theorem 11 (IAND Distributive Law - VII)**:**

$$(A \barwedge B) \vee C \;=\; (A \vee C) \wedge (\overline{B} \vee C) \tag{3.47}$$

Proof. Changing LHS of (3.47) to AND notation,

$$(A \barwedge B) \vee C \;=\; (A \wedge \overline{B}) \vee C \tag{3.48}$$

Applying conventional AND Distributive Law,

$$(A \barwedge B) \vee C \;=\; (A \vee C) \wedge (\overline{B} \vee C) \tag{3.49}$$

which is the equivalent to (3.47). ■

The interaction between three literals A, B and C using IAND and AND gates yield the same result irrespective of the placement of the parentheses.

De Morgan's Law for IAND operation

At this point in the thesis, it is clear that none of the conventional Boolean laws work for IAND gates without some sort of modification. This is true for the De Morgan's law as well. Luckily, when applied to the IAND operation, there is much similarity between the modified version for IAND logic (proposed here), and the accepted convention. The De Morgan's Law can be formally stated as,

Theorem 12: The negation of ordered IAND of two literals, is equal to the OR of the two literals with the non-inverted term complemented.

For two inputs:

$$\overline{A \barwedge B} = \overline{A} \vee B \tag{3.50}$$

$$Conversely, \qquad \overline{A \vee B} = \overline{A} \barwedge B \tag{3.51}$$

For three inputs:

$$\overline{(A \barwedge B) \barwedge C} = \overline{A} \vee B \vee C \tag{3.52}$$

$$Conversely, \qquad \overline{A \vee B \vee C} = (\overline{A} \barwedge B) \barwedge C \tag{3.53}$$

Proof. Converting the LHS of (3.50) to AND notation and applying conventional De Morgan's Law,

$$\overline{A \barwedge B} = \overline{A \wedge \overline{B}} = \overline{A} \vee B \tag{3.54}$$

Similarly for (3.52),

$$\overline{(A \barwedge B) \barwedge C} = \overline{A \wedge \overline{B} \wedge \overline{C}} = \overline{A} \vee B \vee C \tag{3.55}$$

Hence the De Morgan's law for IAND operations is verified. The inverse can also be proved in a similar fashion. ■

Note that it is important to solve the IAND operation in an ordered fashion as mentioned earlier. Although the law has been stated only for two and three input IAND operations, it can be extended to any number of inputs. In every such case, only the non-inverted input will appear as its own complement along with the change of IAND operation to OR.

Principle of Duality

Conventionally a 'dual' of a Boolean function is obtained by replacing the ANDs (ORs) and 1s(0s) with ORs (ANDs) and 0s(1s) respectively. For example, the dual of $(A \vee B)$ is $(A \wedge B)$, and $(D \vee 1)$ is the dual of $(D \wedge 0)$. Boolean duals for expressions involving IANDs can be found with a slight modification to this rule:

- Replace all ANDs with ORs, and vice versa.
- Replace all 1s with 0s, and vice versa.

- Change all IANDs to ORs and complement all inverted inputs.

For example, the dual of the Boolean expression $(A \barwedge B) \barwedge C$ is $(A \vee \overline{B} \vee \overline{C})$. It is worth noting that to correctly obtain the dual, the first two steps need to be performed *before* the third one.

The principle of duality states that if two Boolean expressions are equivalent, then their duals are also equivalent. This holds true for IAND operations as well. Consider the following Boolean equation (true as per Theorem 12):

$$\overline{(A \barwedge B) \barwedge C} = \overline{A} \vee B \vee C \tag{3.56}$$

Transforming the above equation by the rules we mentioned above,

$$Dual(LHS) = \overline{(A \vee \overline{B} \vee \overline{C})} \tag{3.57}$$

Or, by De Morgan's Law,

$$Dual(LHS) = (\overline{A} \wedge B \wedge C) \tag{3.58}$$

Similarly,

$$Dual(RHS) = (\overline{A} \wedge B \wedge C) \tag{3.59}$$

Clearly, (3.58) and (3.59) are equivalent.

Corollary: By careful examination of the above methodology, we discover another way of transforming any Boolean expression into its duals. Instead of the conventional change of ORs to ANDs, they can be changed to IANDs with the inverted inputs complemented.

3.1.3 IAND Operations in Equations

One needs to be careful while considering the equivalence of an IAND operation done across the sides of an equation. Consider, the expressions in equations (3.60)-(3.62).

$$A \wedge (A \vee B) = A \vee (A \wedge B) \tag{3.60}$$

$$Z \barwedge (A \wedge (A \vee B)) = Z \barwedge (A \vee (A \wedge B)) \tag{3.61}$$

$$(A \wedge (A \vee B)) \barwedge Z = Z \barwedge (A \vee (A \wedge B)) \tag{3.62}$$

It is known for a fact that, in most cases, performing the same operation with the same on both sides of an equation will keep its equivalence intact, and thus the above three equations appear to be the same. However, performing an IAND of the two sides of the equation with a generic function Z might disrupt its equivalence if the order of operation is not maintained. This means that one should pay attention to *pre*-IAND or *post*-IAND operations which is analogous to dealing with equations involving matrices. Thus,(3.60) and (3.61) are equivalent whereas (3.62) is not a correct equation.

3.1.4 Canonical Normal Form for IAND/OR Logic Set

There are primarily, two types of canonical normal forms used in Boolean Algebra, namely canonical disjunctive normal form (CDNF) and canonical conjunctive normal form (CCNF). These are described in detail in the following subsections, first for the conventional AND/OR logic set and then for IAND/OR.

Canonical Disjunctive Normal Form (CDNF)

CDNF, better known as minterm canonical form or canonical sum of products (SOP) is a Boolean expression obtained by the OR-ing of certain AND-ed combinations of Boolean

literals. For example,

$$f_{sop}(A,B,C) = (A \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge \overline{B} \wedge \overline{C}) \tag{3.63}$$

CDNF for the IAND/OR logic set, as proposed here, is called sum of IANDs (SOI), and is the OR-ing of certain Boolean literals IAND-ed in different combinations. (3.64) presents an example.

$$f_{soi}(A,B,C) = ((A \barwedge B) \barwedge C) \vee ((A \barwedge \overline{B}) \barwedge \overline{C}) \vee ((\overline{A} \barwedge \overline{B}) \barwedge \overline{C}) \tag{3.64}$$

Canonical conjunctive normal form (CCNF)

CCNF is simply the dual of CDNF and is also known as canonical product of sums (POS). For example,

$$f_{pos}(A,B,C) = (A \vee B \vee C) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee \overline{B} \vee \overline{C}) \tag{3.65}$$

CCNF for the IAND/OR logic set is called IAND of sums (IOS), and is the IAND of the sums of Boolean literals in certain combinations. (3.66) is an example.

$$f_{ios}(A,B,C) = ((A \vee B \vee C) \barwedge (\overline{A} \vee B \vee C)) \barwedge (\overline{A} \vee \overline{B} \vee C) \tag{3.66}$$

Translation to Canonical Normal Forms

It should be noted that in a CDNF (CCNF), all the product (sum) terms must contain every literal in the given set. As an example, the expression in (3.67) is not canonical SOP representation since the second product term does not contain the literal C.

$$f_{ios}(A,B,C) = (A \wedge B \wedge C) \vee (A \wedge \overline{B}) \tag{3.67}$$

Similar to canonical SOP and POS expressions, canonical SOI and IOS expressions can be developed from non-canonical SOI expressions by inserting the literals missing from each term, while maintaining logical equivalence. Applying Identity 3, appending an IAND operation of a null-valued (zero) expression (such as $A \wedge \overline{A}$) enables the expansion without modifying the logical value of the expression. Taking the non-canonical expression

$$f_{soi} \;=\; ((\overline{B} \barwedge \overline{C}) \barwedge \overline{\mathrm{D}}) \;\vee\; ((A \barwedge B) \barwedge \overline{C}) \tag{3.68}$$

$(A \wedge \overline{A})$ and $(D \wedge \overline{D})$ can be appended to the two terms, resulting in

$$f_{soi} \;=\; \{((\overline{B} \barwedge \overline{C}) \barwedge \overline{D}) \barwedge (A \wedge \overline{A})\} \;\vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge (D \wedge \overline{D})\} \tag{3.69}$$

Using Theorem 5,

$$\begin{aligned} f_{soi} = \{((\overline{B} \barwedge \overline{C}) \barwedge \overline{D}) \barwedge A\} \;\vee\; \{((\overline{B} \barwedge \overline{C}) \barwedge \overline{D}) \barwedge \overline{A}\} \\ \vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge D\} \;\vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \end{aligned} \tag{3.70}$$

Rearranging the inverted and non-inverted inputs according to Theorem 4:

$$\begin{aligned} f_{soi} \;=\; \{((\overline{A} \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \;\vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \\ \vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge D\} \;\vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \end{aligned} \tag{3.71}$$

Finally, by conventional OR idempotency, $(A \;\vee\; A) = A$,

$$f_{soi} \;=\; \{((\overline{A} \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \;\vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge D\} \vee\; \{((A \barwedge B) \barwedge \overline{C}) \barwedge \overline{D}\} \tag{3.72}$$

The above expression is a canonical SOI expression. In general, canonical IOS expressions can be achieved with the same method as canonical POS: by OR-ing a null expression of

the missing literals with each of the sum-terms, followed by ordinary Boolean algebraic simplification.

A tabular summary of results has been presented in the Appendix detailing all identities and laws mentioned in this chapter.

3.2 Stateful Memristive Implication Logic

Stateful implication logic is one of the fundamental logic functions implemented using memristors. An IMPLY function is an asymmetric OR of two variables, with one of the variable inverted. In this section, I define the Boolean laws and principles for functions implemented using a combination of IMPLY gates and other standard Boolean functions.

3.2.1 Core Algebraic Identities

Core algebraic identities define the basic properties of a logic operation, and how it fundamentally alters a function. Similar to section 3.1.1, the primary identities have been presented here.

Interaction with High (1) and Low Logic (0)

The section analyzes how the IMPLY operation alters a Boolean function when operated against a zero (0) or a one (1).

Identity 7 (IMPLY Annulment)**:** A post-IMPLY operation of a variable with high logic results in a high logic value (1).

$$A \to 1 \;=\; 1 \tag{3.73}$$

Proof. Representing the IMPLY operation in terms of OR,

$$A \to 1 \;=\; \overline{A} \vee 1 \;=\; 1 \tag{3.74}$$

which proves the theorem. ■

Identity 8 (IMPLY Identity)**:** A pre-IMPLY of high logic with a Boolean variable returns the same variable. In other words, pre-IMPLY operation is the identity for implication operation.

$$1 \to A \; = \; A \tag{3.75}$$

Proof. Representing the IMPLY operation in terms of OR,

$$(1 \to A) = (\overline{1} \; \vee \; A) = (0 \vee A) = A \tag{3.76}$$

which proves the theorem. ■

Identity 9 (IMPLY Inversion)**:** A post-IMPLY operation of a Boolean variable with complements the given variable.

$$A \to 0 \; = \; \overline{A} \tag{3.77}$$

Proof. Representing the IMPLY operation in terms of OR,

$$A \to 0 \; = \; \overline{A} \; \vee \; 0 \; = \; \overline{A} \tag{3.78}$$

which proves the theorem. ■

Idempotency for IMPLY operation

Idempotency is the property that enables multiple iterations of an operation without changing the result. Similar to IAND logic, this property is evaluated for the different cases of repeating operations even though they might not preserve the same value. The properties have been classified as per the results obtained.

Identity 10 (IMPLY Null Idempotency)**:**

$$A \to A \;=\; 1 \tag{3.79}$$

Proof. Converting IMPLY operation to OR notation,

$$A \to A \;=\; \overline{A} \vee A \;=\; 1 \tag{3.80}$$

Hence the theorem has been verified. ■

Identity 11 (IMPLY Inverse Idempotency - I)**:**

$$A \to \overline{A} \;=\; \overline{A} \tag{3.81}$$

Proof. Representing the IMPLY operation in terms of OR,

$$A \to \overline{A} \;=\; \overline{A} \vee \overline{A} \;=\; \overline{A} \tag{3.82}$$

which proves the theorem. ■

Identity 12 (IMPLY Inverse-Idempotency - II)**:**

$$\overline{A} \to A \;=\; A \tag{3.83}$$

Proof. Representing the IMPLY operation in terms of OR,

$$\overline{A} \to A \;=\; \overline{\overline{A}} \vee A \;=\; A \tag{3.84}$$

which proves the theorem. ■

3.2.2 Boolean Algebraic Laws

This section develops the standard laws of Boolean algebra for stateful implication logic. It can be noted that even though some of the laws might deviate from their conventional definitions, a clear analogy is notable for each of them.

Commutative Law

Theorem 13 (IMPLY Commutation)**:** Similar to IANDs, IMPLY being an asymmetric logic function, does not follow the conventional commutative law, *i.e.*

$$A \to B \neq B \to A \tag{3.85}$$

However, considering two operands A and B, commutation can be achieved by complementing both the literals as shown in (3.86)

$$A \to B = \overline{B} \to \overline{A} \tag{3.86}$$

Proof. Changing to IMPLY notation,

$$A \to B = \overline{A} \vee B = B \vee \overline{A}. \tag{3.87}$$

Changing the RHS of (3.87) back to IMPLY notation,

$$A \to B = \overline{B} \to \overline{A}, \tag{3.88}$$

thus proving the theorem. ■

Associativity Laws

Associativity laws define how three or more Boolean functions are grouped when operated upon by a Boolean function. In this section, associativity is defined for the IMPLY operation.

Theorem 14 (Conventional Non-Associativity)**:**

$$(A \to B) \to C \ \neq \ A \to (B \to C) \tag{3.89}$$

Implication basis logic does not follow associativity in its conventional sense. This can be proved as follows. Similar to IANDs, implication should be carefully performed in an ordered fashion.

Proof. Converting the RHS of (3.89) to OR notation and applying De Morgan's law,

$$(A \to B) \to C = \overline{(\overline{A} \ \vee \ B)} \ \vee \ C = (A \wedge \overline{B}) \vee C \tag{3.90}$$

Similarly, changing the LHS to OR notation,

$$A \to (B \to C) = \overline{A} \ \vee \ \overline{B} \ \vee \ C \tag{3.91}$$

Clearly, (3.90) and (3.91) are not equivalent. ■

The following two theorems describe the inverting and non-inverting associativity for implication logic.

Theorem 15 (IMPLY Non-Inverting Associativity)**:** Non-inverting associativity means that on changing the order of certain operands, no inversion is required. This happens when any two inverted operands interchange places.

$$A \to (B \to C) \ = \ B \to (A \to C) \tag{3.92}$$

Proof. Converting the expression to AND notation:

$$A \to (B \to C) \;=\; \overline{A} \vee \overline{B} \vee C \;=\; \overline{B} \vee \overline{A} \vee C. \tag{3.93}$$

This can be rewritten using OR notation as follows:

$$\overline{B} \vee \overline{A} \vee C \;=\; B \to (A \to C). \tag{3.94}$$

The theorem is thus proven. ■

Theorem 16 (IMPLY Inverting Associativity)**:** Inverting associativity demands the inversion of both the operands that have changed places. This theorem should be applied when a non-inverted input interchanges its place with an inverted input.

$$A \to (B \to C) \;=\; B \to (\overline{C} \to \overline{A}) \tag{3.95}$$

Proof. Converting the LHS of (3.95) to OR notation:

$$A \to (B \to C) \;=\; \overline{A} \vee \overline{B} \vee C. \tag{3.96}$$

Rearranging the inputs on the RHS of the above equation,

$$A \to (B \to C) \;=\; \overline{B} \vee C \vee \overline{A} \tag{3.97}$$

which can be rewritten using IMPLY notation as

$$A \to (B \to C) \;=\; B \to (\overline{C} \to \overline{A}), \tag{3.98}$$

which is the same as (3.95). ■

Discussion: As shown in the previous chapter for IANDs, Theorem 15 and Theorem 16 can also be interpreted intuitively as the movement of inverted and non-inverted operands around the IMPLY operator. See discussion in section 3.1.2.

Distributive Laws

This section details the distributive laws for expressions involving a combination of IMPLY and OR/AND operations performed between three binary inputs. As the nomenclature would be quite challenging, these theorems are numbered rather than named. Further, just as for IAND distributive laws, these are categorized into 'single-operation' and 'cross-operation' distributive laws.

Single-Operation Distributive Laws: These laws relate two Boolean expressions, both of which are represented using a combination of implication and AND/OR operations. Theorems 17-21 belong to this category.

Theorem 17 (IMPLY Distributive Law - I)**:**

$$A \rightarrow (B \ \wedge \ C) \ = \ (A \rightarrow B) \ \wedge \ (A \rightarrow C). \tag{3.99}$$

Proof. Changing LHS of (3.99) to OR notation,

$$A \rightarrow (B \ \wedge \ C) \ = \ \overline{A} \vee (B \wedge C). \tag{3.100}$$

Using the conventional AND distributive law,

$$A \rightarrow (B \ \wedge \ C) \ = \ (\overline{A} \vee B) \wedge (\overline{A} \vee C) \tag{3.101}$$

Finally, replacing with IMPLY operation:

$$A \to (B \ \wedge \ C) \ = \ (A \to B) \ \wedge \ (A \to C) \tag{3.102}$$

which is the same as (3.99). ■

Theorem 18 (IMPLY Distributive Law - II)**:**

$$(A \vee B) \to C \ = \ (A \to C) \ \wedge \ (B \to C) \tag{3.103}$$

Proof. Changing LHS of (3.103) to OR notation and using De Morgan's law,

$$(A \vee B) \to C = \overline{A \vee B} \ \vee \ C = (\overline{A} \wedge \overline{B}) \vee C \tag{3.104}$$

By conventional AND distributive law,

$$(A \vee B) \to C = (\overline{A} \vee C) \wedge (\overline{B} \vee C) \tag{3.105}$$

Finally, replacing in terms of IMPLY:

$$(A \vee B) \to C \ = \ (A \to C) \ \wedge \ (B \to C) \tag{3.106}$$

which is the same as (3.103). ■

Theorem 19 (IMPLY Distributive Law - III)**:**

$$A \vee (B \to C) \ = \ \overline{A} \to (B \to C) \tag{3.107}$$

Proof. Changing the LHS of (3.107) to OR notation and subsequently back to IMPLY,

$$A \vee (B \rightarrow C) \; = A \vee \overline{B} \vee C = \overline{A} \rightarrow (B \rightarrow C) \tag{3.108}$$

The above equation validates the theorem. ■

The interaction between three literals A, B and C using IMPLY and OR gates yield the same result irrespective of the placement of the parentheses as seen in Theorem 20.

Theorem 20 (IMPLY Distributive Law - IV)**:**

$$A \rightarrow (B \vee C) = (A \rightarrow B) \vee C = A \rightarrow (\overline{B} \rightarrow C) \tag{3.109}$$

Proof. Changing the left and central terms of (3.109) to OR notation,

$$A \rightarrow (B \; \vee \; C) = \overline{A} \vee B \vee C \tag{3.110}$$

$$\textit{and}, \quad (A \rightarrow B) \vee C = \overline{A} \vee B \vee C \tag{3.111}$$

The RHS of the (3.110) and (3.111) can be rewritten in IMPLY notation as $A \rightarrow (\overline{B} \rightarrow C)$. Thus,

$$A \rightarrow (B \vee C) = (A \rightarrow B) \vee C = A \rightarrow (\overline{B} \rightarrow C) \tag{3.112}$$

which is the same as (3.109). ■

Theorem 21 (IMPLY Distributive Law - V)**:**

$$(A \wedge B) \rightarrow C \; = \; A \rightarrow (B \rightarrow C) \tag{3.113}$$

Proof. Changing the LHS of (3.113) to OR notation and applying De Morgan's Law,

$$(A \wedge B) \rightarrow C \; = \; \overline{A \wedge B} \; \vee \; C = \overline{A} \; \vee \; \overline{B} \; \vee \; C, \tag{3.114}$$

Finally, changing it back to the IMPLY notation,

$$(A \wedge B) \rightarrow C \; = \; A \rightarrow (B \rightarrow C) \tag{3.115}$$

The above equation is the same as (3.113) ■

Cross-Operation Distributive Laws: These laws relate a Boolean expression represented in terms of implication and AND/OR operations with an expression represented only in terms of AND/OR operations. Theorems 22-23 are cross-operation distributive laws.

Theorem 22 (IMPLY Distributive Law - VI)**:**

$$(A \rightarrow B) \wedge C \; = \; (\overline{A} \wedge C) \vee (B \wedge C) \tag{3.116}$$

Proof. Changing the LHS of (3.116) to OR notation and subsequently back to IMPLY:

$$(A \rightarrow B) \wedge C \; = \; (\overline{A} \; \vee \; B) \wedge C \tag{3.117}$$

$$(A \rightarrow B) \wedge C \; = \; (\overline{A} \wedge C) \vee (B \wedge C) \tag{3.118}$$

The above equation being the same as (3.116) ■

Theorem 23 (IMPLY Distributive Law - VII)**:**

$$A \wedge (B \rightarrow C) = (A \wedge \overline{B}) \vee (A \wedge C) \tag{3.119}$$

Proof. Changing LHS of (3.119) to OR notation,

$$A \wedge (B \rightarrow C) = A \wedge (\overline{B} \vee C) \tag{3.120}$$

Applying conventional OR Distributive Law,

$$A \wedge (B \rightarrow C) = (A \wedge \overline{B}) \vee (A \wedge C) \tag{3.121}$$

which is the equivalent to (3.119). ■

De Morgan's Law for IMPLY operation

De Morgan's Law for implication logic works analogously to its IAND counterpart. This idea is evaluated and proved in the following theorem.

Theorem 24: The negation of ordered implication of two literals, is equal to the NAND of the two literals with the non-inverted term complemented. Note that the non-inverted term for IMPLY logic is not the same as that for IANDs.

For two inputs:

$$\overline{A \rightarrow B} = A \wedge \overline{B} \tag{3.122}$$

$$Conversely, \qquad \overline{A \wedge B} = A \rightarrow \overline{B} \tag{3.123}$$

For three inputs:

$$\overline{A \rightarrow (B \rightarrow C)} = A \wedge B \wedge \overline{C} \tag{3.124}$$

$$Conversely, \qquad \overline{A \wedge B \wedge C} = A \rightarrow (B \rightarrow \overline{C}) \tag{3.125}$$

For the purpose of actual realization of a circuit with the IMPLY/NAND logic set, the De Morgan's rule can be restated in terms of the fundamental gates only. The inverter can be easily realized using just a NAND gate with shorted inputs.

For two inputs:

$$\overline{A \to B} = \overline{\overline{A \wedge \overline{B}}} = \text{INV (NAND (A, B))} \tag{3.126}$$

$$Or, \qquad A \to B = \overline{A \wedge \overline{B}} = \text{NAND (A, B)} \tag{3.127}$$

For three inputs:

$$\overline{A \to (B \to C)} = \overline{\overline{A \wedge B \wedge \overline{C}}} = \text{INV (NAND (A, B, C))} \tag{3.128}$$

$$Or, \qquad A \to (B \to C) = \overline{A \wedge B \wedge \overline{C}} = \text{NAND (A, B, C)} \tag{3.129}$$

Proof. Converting the LHS of (3.122) to OR notation and applying conventional De Morgan's Law,

$$\overline{A \to B} = \overline{\overline{A} \vee B} = A \wedge \overline{B} \tag{3.130}$$

Similarly for (3.124),

$$\overline{A \to (B \to C)} = \overline{\overline{A} \vee \overline{B} \vee C} = A \wedge B \wedge \overline{C} \tag{3.131}$$

Hence the De Morgan's law for implication logic is verified. ■

Similar to an earlier discussion, IMPLY operations should be performed in an ordered manner when applying the De Morgan's Law. Further, it can be extended to any number of inputs and only the non-inverted input will appear as its own complement along with the change of IMPLY operation to AND.

Principle of Duality

Now that the algebra for both IAND and IMPLY logic is established, the process of finding the dual of any Boolean function, consisting of any one, all or a mixture of any of the Boolean operators can be generalized as follows:

- Replace all ANDs with ORs, and vice versa.
- Replace all 1s with 0s, and vice versa.
- Change all IANDs to ORs and complement all inverted inputs.
- Replace all IMPLY with ANDs and complement all the inverted inputs.

For example, the dual of the Boolean expression $A \to (B \to C)$ is $(\overline{A} \vee \overline{B} \vee C)$. It is worth noting that to correctly obtain the dual, the first two steps need to be performed *before* the third and the fourth ones.

To evaluate the principle of duality for implication operations, consider the following Boolean equation (true as per Theorem 24):

$$\overline{A \to (B \to C)} = A \wedge B \wedge \overline{C} \tag{3.132}$$

Transforming the above equation by the rules we mentioned above,

$$Dual(LHS) = \overline{\overline{A} \wedge \overline{B} \wedge C} \tag{3.133}$$

Or, by De Morgan's Law,

$$Dual(LHS) = A \vee B \vee \overline{C} \tag{3.134}$$

Similarly,

$$Dual(RHS) = A \vee B \vee \overline{C} \tag{3.135}$$

Clearly, (3.134) and (3.135) are equivalent.

Corollary: Observing the principle of duality for IAND and IMPLY logic, we discover an alternate method for finding the dual of a Boolean function: Instead of replacing the ORs with ANDs, they can be replaced with IANDs instead, whereas instead of replacing ANDs

with ORs, they can be replaced by IMPLYs, keeping in mind the appropriate inversion of inputs. As mentioned earlier for IANDs, implication function should be dealt with care when applied to equations.

3.2.3 Canonical Normal Form for IMPLY/NAND Logic Set

IMPLY/NAND is a basis logic set that can be used to implement any given Boolean function. Thus, it is vital to mention the canonical forms for this logic set.

Canonical Disjunctive Normal Form (CDNF)

CDNF for the IMP/NAND logic set, as proposed here, is called canonical NAND of implication (NOI), and is the NAND of certain Boolean literals IMPLY-ed in different combinations. (3.136) presents an example.

$$f_{noi}(A,B,C) = \overline{(\overline{A} \to (B \to C)) \wedge (A \to (B \to C)) \wedge (A \to (\overline{B} \to C))} \tag{3.136}$$

Canonical conjunctive normal form (CCNF)

CCNF for the IMP/NAND logic set is called canonical implication of NANDs (ION), and is the IMPLY of the NANDs of Boolean literals in certain combinations. (3.137) is an example.

$$f_{ion}(A,B,C) = (\overline{A \wedge B \wedge C}) \to ((\overline{\overline{A} \wedge B \wedge C}) \to (\overline{\overline{A} \wedge \overline{B} \wedge \overline{C}})) \tag{3.137}$$

Translation to Canonical Normal Forms

A given Boolean expression may not be in its canonical form, and thus similar to the methodology adopted for IANDs, canonical NOI expressions can be developed from non-canonical NOI expressions by inserting the literals missing from each term, while maintaining logical equivalence. Applying Identity 8, appending an IMPLY operation of a unity (1) expression

(such as $A \vee \overline{A}$) enables the expansion without modifying the logical value of the expression. Consider the non-canonical expression in (3.138),

$$f_{noi} \;=\; \overline{(\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D})) \;\wedge\; (A \rightarrow (B \rightarrow \overline{C}))} \tag{3.138}$$

$(A \vee \overline{A})$ and $(D \vee \overline{D})$ can be appended to the two terms, resulting in

$$f_{noi} \;=\; \overline{\{(A \vee \overline{A}) \rightarrow (\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D}))\} \;\wedge\; \{(D \vee \overline{D}) \rightarrow (A \rightarrow (B \rightarrow \overline{C}))\}} \tag{3.139}$$

Using Theorem 18,

$$\begin{aligned} f_{noi} = \overline{\{A \rightarrow (\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D}))\} \;\wedge\; \{\overline{A} \rightarrow (\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D}))\}} \\ \overline{\wedge\, \{D \rightarrow (A \rightarrow (B \rightarrow \overline{C}))\} \;\wedge\; \{\overline{D} \rightarrow (A \rightarrow (B \rightarrow \overline{C}))\}} \end{aligned} \tag{3.140}$$

Rearranging the inverted and non-inverted inputs according to Theorems 15 and 16:

$$\begin{aligned} f_{noi} \;=\; \overline{\{A \rightarrow (\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D}))\} \;\wedge\; \{\overline{A} \rightarrow (\overline{B} \rightarrow (\overline{C} \rightarrow \overline{D}))\}} \\ \overline{\wedge\, \{A \rightarrow (B \rightarrow (C \rightarrow \overline{D}))\} \;\wedge\; \{A \rightarrow (B \rightarrow (C \rightarrow D))\}} \end{aligned} \tag{3.141}$$

The above expression is a canonical NOI expression. In general, canonical ION expressions can be achieved with the same method as canonical POS: by AND-ing a unity expression of the missing literals with each of the sum-terms, followed by ordinary Boolean algebraic simplification.

3.3 Relationship between IAND and Implication Logic (De Morgan Duality)

Applying the De Morgan's to the implication operation, as stated in the previous section, changes the operation to AND/NAND. This leads to an interesting observation as noted in

(3.122) which is:

$$\overline{A \to B} = A \wedge \overline{B} \tag{3.142}$$

The RHS of the above equation is essentially equivalent to the IAND of A and B,

$$\overline{A \to B} = A \barwedge B \tag{3.143}$$

Similarly, for three inputs, (3.124) shows that,

$$\overline{A \to (B \to C)} = A \wedge B \wedge \overline{C} \tag{3.144}$$

Here, the RHS of the above equation is the IAND of A, $\overline{B}$ and C,

$$\overline{A \to (B \to C)} = (A \barwedge \overline{B}) \barwedge C \tag{3.145}$$

By observation, the converse of (3.142) and (3.145) are also true,

$$\overline{A \barwedge B} = A \to B \tag{3.146}$$

$$\overline{(A \barwedge B) \barwedge C} = A \to (\overline{B} \to C) \tag{3.147}$$

This idea can be extended to establish a general relationship between IAND and implication logic. Thus, keeping in mind the modified forms of the De Morgan's laws for asymmetric logic functions mentioned in sections 3.1.2 and 3.2.2, it can be stated that IAND and implication logic are "De Morgan duals". This is verified below.

By a generic definition, the De Morgan dual f_d of a any given function f is defined as,

$$f_d(a_1, a_2, a_3 ... a_n) = \overline{f(\overline{a_1}, \overline{a_2}, \overline{a_3} ... \overline{a_n})} \tag{3.148}$$

Note that in symmetric functions like AND and OR, the order of the operation does not matter. However, to account for the asymmetry of IANDs and implication logic, the operation must be performed keeping its order in mind. Interestingly, the definition for a De Morgan dual can be established by a mere mirroring of (3.148).

$$f_d(a_1, a_2, a_3 ... a_n) = \overline{f(\overline{a_n}, \overline{a_{n-1}}, \overline{a_{n-2}} ... \overline{a_2}, \overline{a_1})} \tag{3.149}$$

where f_d and f are the IAND and Implication functions respectively (or vice versa).

As a simple example, the dual of $A \wedge B$ is $\overline{\overline{B} \rightarrow \overline{A}}$. This is evident from the modified De Morgan's Law stated in the previous sections. It can be further verified by considering the Boolean equation (3.150), which is true as per Theorem 4.

$$(A \wedge B) \wedge C = (\overline{B} \wedge \overline{A}) \wedge C \tag{3.150}$$

Evaluating the duals of both sides of the above equation,

$$\overline{\overline{C} \rightarrow (\overline{B} \rightarrow \overline{A})} = \overline{\overline{C} \rightarrow (A \rightarrow B)} \tag{3.151}$$

Applying Theorem 13 to the LHS of (3.151),

$$\overline{\overline{C} \rightarrow (A \rightarrow B)} = \overline{\overline{C} \rightarrow (A \rightarrow B)} \tag{3.152}$$

Hence, the equality is maintained.

This translation can also be used to convert any SOI to its equivalent NOI (or vice versa) by using the following standard procedure:

a. Complement all literals in each of the terms.

b. Invert the order of the literals such that the last literal is the first and so on.

c. Rearrange the parentheses such that the operations are performed orderly.

c. Replace the IAND operators with implication operators, and the OR operators with NAND operators.

d. Complement the resultant expression, thus going from SOI to NOI.

Example: Conversion of SOI to NOI:

$$((A \not\leftarrow B) \not\leftarrow C) \vee ((D \not\leftarrow E) \not\leftarrow F) = \overline{(\overline{C} \rightarrow (\overline{B} \rightarrow \overline{A})) \wedge (\overline{F} \rightarrow (\overline{E} \rightarrow \overline{D}))} \tag{3.153}$$

The above relation can be proved similar to the Boolean laws in the previous sections, *i.e.* by solving it in the OR/AND notations and converting it back to implication/IAND.

All the identities proposed in this chapter can be used to simplify any function represented using IAND/OR or IMP/NAND logic sets, without converting them into conventional AND/OR notations. This not only simplifies the process for manual calculations, on an automated algorithm level, this helps to reduce performance overheads.

CHAPTER 4

MODIFIED KARNAUGH MAP METHOD FOR ASYMMETRIC LOGIC FUNCTIONS

The proposed Karnaugh map method enables a graphical technique for the minimization of memristor and spintronic logic with asymmetric basis functions. Section 2.4 provides an explanation of conventional Karnaugh maps. In this section, after the adapted Karnaugh map method is described, I compare both of the methodologies to build a conceptual understanding as well as evaluate the similarities and differences. The chapter ends by examples that provide instruction as to the use of the method.

4.1 Map Method for Asymmetric Functions

The proposed map method for asymmetric logic functions is performed in the following step-wise manner:

1. Transform the expression that is to be minimized into a canonical expression (*i.e.* SOI/IOS or NOI/ION).

2. Mark the canonical SOI/IOS/NOI/ION terms in the corresponding cells of a Karnaugh map.

3. Group the minterms/maxterms graphically according to standard Karnaugh map pairing techniques[19].

4. Create an expression with the resultant terms, analogous to reading a standard Karnaugh map.

5. If any input variable, other than the one that is not inverted in the canonical expression (leftmost operand in case of IANDs and rightmost operand in case of implication),

appears as a non-inverted operand of a product (sum) term in the simplified expression interpreted from a Karnaugh map, it should be complemented to obtain the correct reduced function. This step is exemplified below.

Example 4.1 (Reduction of canonical SOI)

In a four Bit Boolean function defined with ordered variables $\{A, B, C, D\}$, if any one of the inputs among $\{B, C, D\}$ appear as the non-inverted input, it must be complemented. While the first four steps are unsurprising, the evaluation of the example discussed in Section 2.4 (with AND operations replaced by IAND operations) demonstrates the necessity of the fifth step:

$$f_{soi} = ((\overline{A} \barwedge \overline{B}) \barwedge C) \vee ((A \barwedge \overline{B}) \barwedge C) \vee ((A \barwedge B) \barwedge C) \tag{4.1}$$

This expression can be simplified by applying the theorems outlined in Section 3.1. First using Theorem 6 twice in succession gives (4.2) and (4.3):

$$f_{soi} = (((\overline{A} \barwedge \overline{B}) \vee (A \barwedge \overline{B})) \barwedge C) \vee ((A \barwedge B) \barwedge C) \tag{4.2}$$

$$f_{soi} = (((\overline{A} \vee A) \barwedge \overline{B}) \barwedge C) \vee ((A \barwedge B) \barwedge C). \tag{4.3}$$

$(\overline{A} \vee A)$ is unity due to the OR complement law, and $(1 \barwedge \overline{B}) = B$ according to Theorem 2. Thus,

$$f_{soi} = ((1 \barwedge \overline{B}) \barwedge C) \vee ((A \barwedge B) \barwedge C) = (B \barwedge C) \vee ((A \barwedge B) \barwedge C). \tag{4.4}$$

When comparing (2.8) and (4.4), it can be seen that input B in the first term is inverted due to the difference in the of AND and IAND functions (i.e. $(1 \wedge \overline{B}) = \overline{B}$ as opposed to $(1 \barwedge \overline{B}) = B$). Thus, although the pairings in the Karnaugh map remain the same as in Fig. 2.5, the application of step 5 is necessary when reading the Karnaugh map to correctly

interpret the minimized function. Solving the expression further by applying Theorem 5 and Theorem 10 in succession:

$$f_{soi} = (B \vee (A \barwedge B)) \barwedge C \tag{4.5}$$

$$f_{soi} = ((B \vee A) \wedge (B \vee \overline{B})) \barwedge C. \tag{4.6}$$

With the conventional OR complement law, and $(B \vee \overline{B}) = 1$ using Theorem 6, this becomes:

$$f_{soi} = (B \barwedge C) \vee (A \barwedge C). \tag{4.7}$$

The same result is obtained by applying the proposed methodology to the Karnaugh map in Fig. 2.5. Note that the literal B is complemented as per step 5 of the proposed methodology since it is an inverted input and appears as an non-inverted input in the minimized expression. This method is equally applicable to other asymmetric functions, as shown for canonical NOI in the next example.

Example 4.2 (Reduction of canonical NOI):

Consider the Boolean expression in (4.8).

$$f_{noi} = \overline{(\overline{A} \to (B \to C)) \wedge (\overline{A} \to (B \to \overline{C})) \wedge (A \to (B \to C))} \tag{4.8}$$

As in the previous example, we solve this first by using the proposed Boolean laws for implication logic. Using Theorem 17 in succession,

$$f_{noi} = \overline{(\overline{A} \to ((B \to C) \wedge (B \to \overline{C}))) \wedge (A \to (B \to C))} \tag{4.9}$$

$$f_{noi} = \overline{(\overline{A} \to ((B \to (C \wedge \overline{C})))) \wedge (A \to (B \to C))} \tag{4.10}$$

Now, $(C \wedge \overline{C}) = 1$ as per OR complement law. Thus,

$$f_{noi} = \overline{(\overline{A} \rightarrow ((B \rightarrow 0))) \wedge (A \rightarrow (B \rightarrow C))} \tag{4.11}$$

By Theorem 9, $(B \rightarrow 0) = \overline{B}$. Hence,

$$f_{noi} = \overline{(\overline{A} \rightarrow \overline{B}) \wedge (A \rightarrow (B \rightarrow C))} \tag{4.12}$$

Rearranging the literals as per Theorems 15 and 16,

$$f_{noi} = \overline{(B \rightarrow A) \wedge (B \rightarrow (A \rightarrow C))} \tag{4.13}$$

Applying Theorem 17 once again,

$$f_{noi} = \overline{B \rightarrow (A \wedge (A \rightarrow C))} \tag{4.14}$$

By Theorem 23,

$$f_{noi} = \overline{B \rightarrow ((A \wedge \overline{A}) \vee (A \wedge C))} \tag{4.15}$$

Again, by OR complement law, $(A \wedge \overline{A}) = 0$.

$$f_{noi} = \overline{B \rightarrow (A \wedge C)} \tag{4.16}$$

Now finally, applying Theorem 17, and rearranging as per Theorem 16,

$$f_{noi} = \overline{(B \rightarrow A) \wedge (B \rightarrow C)} \tag{4.17}$$

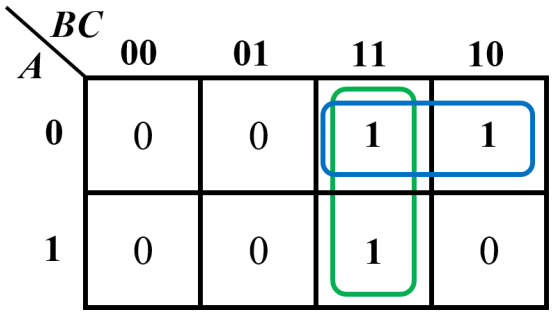

Figure 4.1. Karnaugh map for (4.8).

Rearranging the literals as per Theorem 16,

$$f_{noi} = \overline{(\overline{A} \rightarrow \overline{B}) \wedge (B \rightarrow C)} \tag{4.18}$$

Equation (4.18) is the minimized form of (4.8). Now, applying steps 1-4 of the modified Karnaugh method proposed in this section to the expression in (4.8) (Karnaugh map shown in Fig. 4.1), we obtain,

$$f^*_{noi} = \overline{(\overline{A} \rightarrow B) \wedge (B \rightarrow C)} \tag{4.19}$$

where f^*_{noi} is an intermediate result. Observe that literal B is a inverted input, and appears as a non-inverted input in the optimized solution above. Thus, as per step 5 of the proposed method, it must be complemented to get the correct answer.

$$f_{noi} = \overline{(\overline{A} \rightarrow \overline{B}) \wedge (B \rightarrow C)} \tag{4.20}$$

Hence, (4.18) and (4.20) show that the proposed method is valid.

4.2 Minimization Examples

To clearly explain the mapping methodology, I demonstrate the application of Karnaugh maps to the decomposition of a few sample Boolean expressions involving IAND and im-

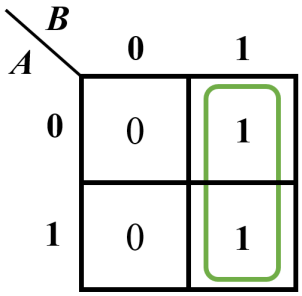

Figure 4.2. Karnaugh map for example 4.3.

plication logic. Although the method outlined in the previous section has been verified for Karnaugh maps containing up to five variables, for the sake of brevity we limit these examples to four variables.

Examples (4.3) and (4.4) explain the reduction procedure for a two and three variable SOIs respectively, whereas example (4.5) details the simplication of a four variable SOI with *don't care* terms. Example (4.6) deals with a function defined as canonical IAND of sums. Finally, example (4.7) solves for a 3-input NAND of implications (NOI). Possible gate-level circuit implementations using only the BASDL device (OR and IAND gates) has been laid out for examples (4.4) through (4.7).

Example 4.3 (Two variable SOI):

$$f_{soi}(A, B) \;=\; (\overline{A} \barwedge B) \;\vee\; (A \barwedge B) \tag{4.21}$$

We first simplify the function in terms of SOP and then convert it back to the reduced SOI. The corresponding Sum of Products is:

$$f_{sop}(A, B) \;=\; (\overline{A} \wedge \overline{B}) \;\vee\; (A \wedge \overline{B}) \tag{4.22}$$

Grouping terms,

$$f_{sop}(A, B) \;=\; (\overline{A} \vee A) \wedge \overline{B} \tag{4.23}$$

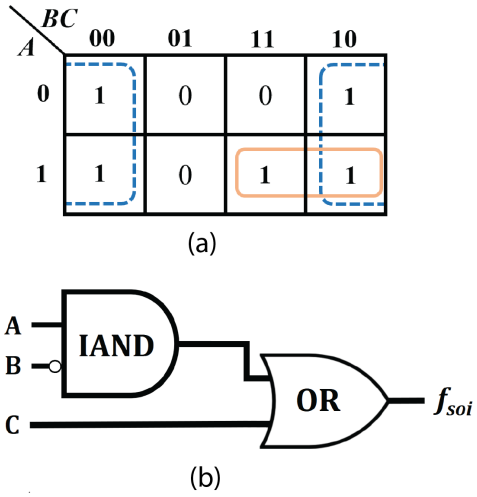

Figure 4.3. (a) Karnaugh map and (b) circuit implementation for example 4.4.

By Complement Law,

$$f_{sop}(A, B) \;=\; \overline{B} \tag{4.24}$$

which remains the same in SOI,

$$f_{soi}(A, B) \;=\; \overline{B} \tag{4.25}$$

Now from the graphical pairing of the K-map representation of (4.21) (as in Fig. 4.2) we obtain the following expression:

$$f^{*}_{soi}(A, B) \;=\; B$$

We see that input B is an inverted input and thus must be complemented to obtain the correct optimized solution:

$$f_{soi}(A, B) \;=\; \overline{B} \tag{4.26}$$

We see that (4.25) and (4.26) are the same which proves the validity of the result.

Example 4.4 (Three variable SOI):

$$f_{soi} = ((\overline{A} \barwedge \overline{B}) \barwedge \overline{C}) \vee ((\overline{A} \barwedge B) \barwedge \overline{C}) \vee ((A \barwedge \overline{B}) \barwedge \overline{C}) \vee ((A \barwedge B) \barwedge \overline{C}) \vee ((A \barwedge B) \barwedge C) \tag{4.27}$$

The equivalent sum of product is,

$$f_{sop} = (\overline{A} \wedge B \wedge C) \vee (\overline{A} \wedge \overline{B} \wedge C) \vee (A \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge C) \vee (A \wedge \overline{B} \wedge \overline{C}) \quad (4.28)$$

Grouping the terms and applying the OR Complement Law,

$$f_{sop} \ = \ (\overline{A} \wedge C \wedge (B \vee \overline{B})) \ \vee \ (A \wedge C \wedge (B \vee \overline{B})) \ \vee \ (A \wedge \overline{B} \wedge \overline{C}) \quad (4.29)$$

$$f_{sop} \ = ((A \vee \overline{A}) \wedge C) \ \vee \ (A \wedge \overline{B} \wedge \overline{C}) \quad (4.30)$$

$$f_{sop} \ = C \ \vee \ (A \wedge \overline{B} \wedge \overline{C}) \quad (4.31)$$

Using the AND distributive law and OR Complement Law in succession,

$$f_{sop} \ = (C \ \vee \ (A \wedge \overline{B})) \ \wedge \ (C \vee \overline{C}) \quad (4.32)$$

$$f_{sop} \ = C \ \vee \ (A \wedge \overline{B}) \quad (4.33)$$

which can be written back as sum of IANDs,

$$f_{soi} = C \ \vee \ (A \wedge\!\!\!\!\wedge \ B) \quad (4.34)$$

Now, we plot the minterms of (4.27) in a K-map as shown in Fig. 4.3(a). Graphical pairing gives the following reduced SOI:

$$f^*_{soi}(A, B, C) \ = \ \overline{C} \ \vee \ (A \wedge\!\!\!\!\wedge \ B) \quad (4.35)$$

Note that input C is an inverted input in the above expression and thus should be complemented as shown below:

$$f_{soi}(A, B, C) \ = \ C \ \vee \ (A \wedge\!\!\!\!\wedge \ B) \quad (4.36)$$

We see that (4.34) and (4.36) are the same, proving the validity of the result. Fig. 4.3(b) shows a possible circuit implementation of the optimized solution.

Example 4.5 (Incompletely specified four variable SOI):

In this example we deal with a four variable Boolean function with *don't care* terms. The function is specified below, where the terms with the superscript 'd_1' and 'd_2' are don't cares.

$$\begin{aligned} f_{soi} = (((\overline{A} \wedge \overline{B}) \wedge C) \wedge \overline{D})^{d_1} \vee (((\overline{A} \wedge \overline{B}) \wedge C) \wedge D) \vee (((\overline{A} \wedge B) \wedge \overline{C}) \wedge \overline{D}) \\ \vee (((\overline{A} \wedge B) \wedge \overline{C}) \wedge D) \vee (((\overline{A} \wedge B) \wedge C) \wedge \overline{D})^{d_2} \vee (((A \wedge \overline{B}) \wedge C) \wedge \overline{D}) \\ \vee (((A \wedge \overline{B}) \wedge C) \wedge D) \vee (((A \wedge B) \wedge \overline{C}) \wedge \overline{D}) \end{aligned} \quad (4.37)$$

The above SOI expression is plotted in the Karnaugh map in Fig. 4.4. Since there are two don't care terms ($d_1 = 2$ and $d_2 = 6$) in the function, there is a total of four possible representations depending on the binary combinations of $[d_1, d_2]$. Even though only one of these combinations lead to the maximally reduced expression, all of them are shown in order to test the proposed mapping methodology.

The possible K-Map representations are shown in Fig. 4.4($b - e$), which correspond to reduced forms of the function in (4.37). The resultant SOI expressions (described by (4.38)-(4.41)) can be deduced by applying the mapping method, and validated by simplifying as sum of products, similar to the previous examples.

$$f^b_{soi} = ((\overline{A} \wedge B) \wedge \overline{C}) \vee ((\overline{B} \wedge \overline{C}) \wedge \overline{D}) \vee (B \wedge C) \quad (4.38)$$

$$f^c_{soi} = ((\overline{A} \wedge B) \wedge \overline{C}) \vee ((\overline{B} \wedge \overline{C}) \wedge \overline{D}) \vee (B \wedge C) \vee ((\overline{A} \wedge B) \wedge \overline{D}) \quad (4.39)$$

$$f^d_{soi} = ((\overline{A} \wedge B) \wedge \overline{C}) \vee ((\overline{B} \wedge \overline{C}) \wedge \overline{D}) \vee ((B \wedge C) \wedge D) \vee ((A \wedge \overline{B}) \wedge C) \quad (4.40)$$

$$f^e_{soi} = \{f^d_{soi}\} \vee ((\overline{A} \wedge B) \wedge \overline{D}) \quad (4.41)$$

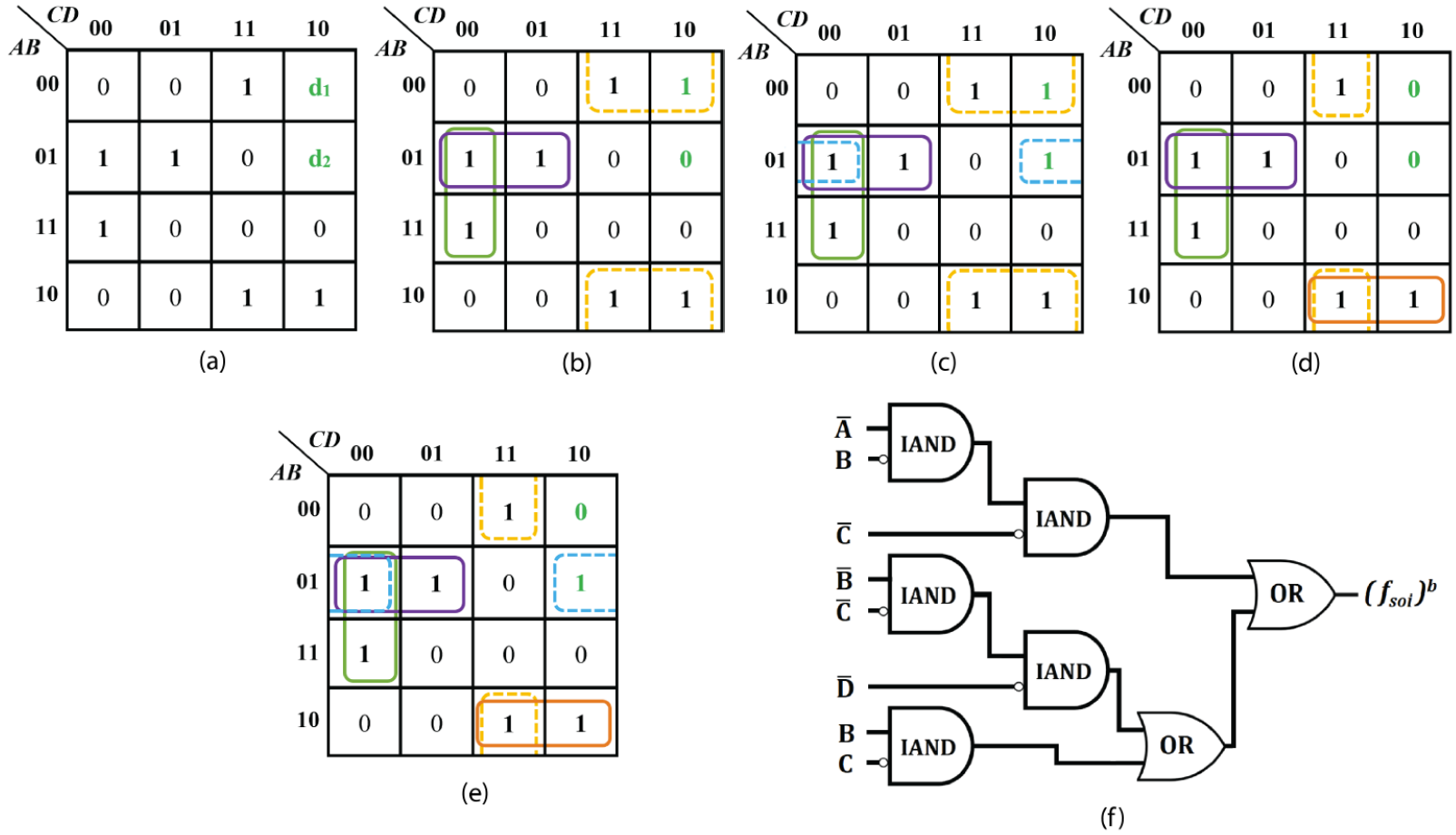

Figure 4.4. (a) Karnaugh map for example 4.5 with $[d_1, d_2]$ values: (b) [1,0] (c) [1,1] (d) [0,0,] (e) [0,1] (g) IAND-OR implementation for the optimal solution of example 4.5 (equation 4.38).

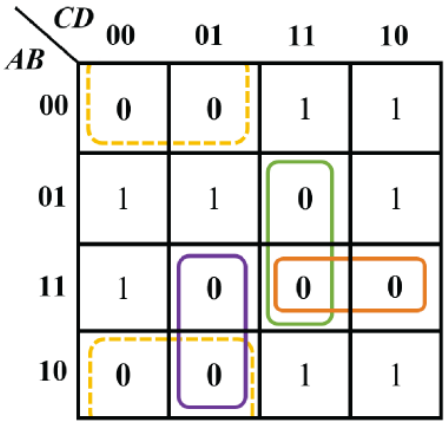

Figure 4.5. Karnaugh map for example 4.6.

where f_{soi}^{b-e} are the reduced SOI expressions for the corresponding parts of Fig. 4.4. It can be seen that f_{soi}^{b} gives the maximally minimized function in terms of gates.

Example 4.6 (Four variable IOS):

This example illustrates the mapping of the maxterms of a Boolean function and their reduction by our proposed method. For analogy, the maxterms of the function in (4.37) with $(d_1 = d_2 = 1)$ are considered here. Note that the below IOS must be evaluated in an orderly fashion (two-at-a-time from left to right) even though the parentheses are not shown here (to avoid cluttering).

$$f_{ios} = (\overline{A} \vee \overline{B} \vee \overline{C} \vee \overline{D}) \wedge (\overline{A} \vee \overline{B} \vee C \vee \overline{D}) \wedge (\overline{A} \vee B \vee C \vee D) \wedge (A \vee \overline{B} \vee \overline{C} \vee \overline{D}) \wedge (A \vee \overline{B} \vee \overline{C} \vee D) \wedge (A \vee B \vee \overline{C} \vee D) \wedge (A \vee B \vee C \vee \overline{D}) \wedge (A \vee B \vee C \vee D) \quad (4.42)$$

Fig. 4.5 shows the graphical pairing of maxterms in (4.42) which yields the reduced function f_{ios}^{f} given by:

$$f_{ios}^{f} = (((B \vee \overline{C}) \wedge (A \vee \overline{C} \vee D)) \wedge (A \vee B \vee C)) \wedge (\overline{B} \vee C \vee D) \quad (4.43)$$

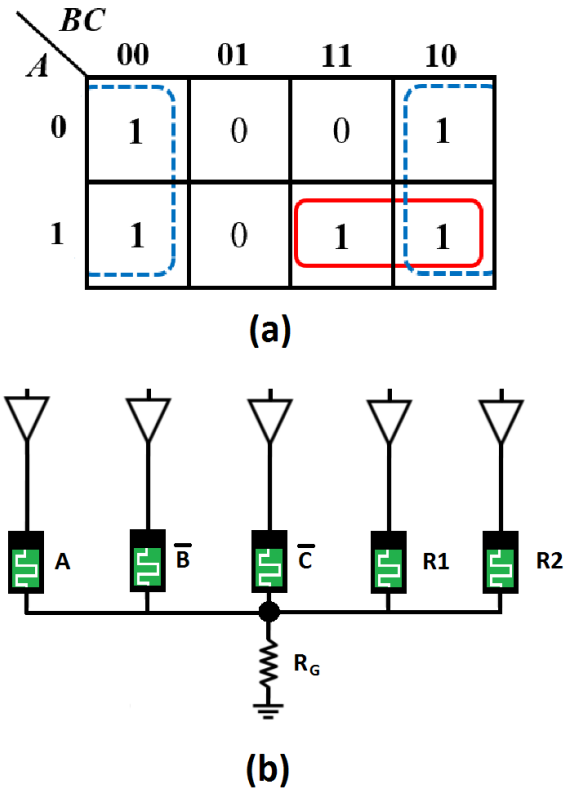

Figure 4.6. Example 4.7: (a) Karnaugh map for (4.44). (b) Memristor circuit for IMPLY-NAND implementation of (4.45). A, $\overline{B}$, and $\overline{C}$ contain the input values, while R_1 and R_2 are output memristors.

The above result can be verified by solving (4.42) in the SOP notation, and converting it back to IOS.

Example 4.7 (Three variable NO I):

This example illustrates the optimization of an NOI expression and its implementation using the IMPLY-NAND logic set (Fig. 4.6). Consider the function in (4.44), now represented as an NOI and plotted on a Karnaugh map as shown in Fig. 4.5(a).

$$f_{soi} = \{(A \to (\overline{B} \to C)) \vee (A \to (B \to C)) \vee (\overline{A} \to (\overline{B} \to C)) \\ \vee\ (\overline{A} \to (B \to C))\ \wedge\ (\overline{A} \to (B \to \overline{C}))\} \tag{4.44}$$

Using the proposed mapping method, non-inverted input B is complemented and the simplified function is written as

$$f_{noi} = \overline{\overline{C} \wedge (A \rightarrow \overline{B})} \tag{4.45}$$

Fig. 4.6(b) shows a possible memristor circuit for the IMPLY-NAND implementation. Here, complementary representation [29] is used and the function can be arrived at by executing the following computational sequence [29]:

$$\mathtt{A} \rightarrow \mathtt{R}_2;$$

$$\overline{\mathtt{B}} \rightarrow \mathtt{R}_2;$$

$$\mathtt{R}_2 \rightarrow \mathtt{R}_1;$$

$$\overline{\mathtt{C}} \rightarrow \mathtt{R}_2.$$

The desired value is stored in the output memristor R_2.

CHAPTER 5

COMPARATIVE ANALYSIS OF THE PROPOSED METHODOLOGY

As mentioned earlier, the primary purpose of proposing a new set of Boolean algebra for asymmetric gates like memristive implication, is to eliminate the mapping of these gates to conventional logic gates so as to use them directly. This chapter highlights the statistical advantages over the previous approach.

In this chapter, I compare my proposed method of using the IMPLY-NAND asymmetric logic set directly to implement Boolean functions, against a previous logic synthesis technique by Lehtonen *et. al* [29]. Note that the IAND-OR logic set has not been yet investigated for suitable logic synthesis techniques, and therefore this is the first work in that direction.

Further, since memristors do not cascade in a conventional manner within the stateful memristor logic paradigm, device count (*i.e.* the total number of devices required to implement a given Boolean function) is not an appropriate comparative parameter. Instead, the number of required *computational steps* is used. The calculation of this number depends on the specific operation and the technique used to implement logic. It should be noted that the analysis presented in this chapter is neutral towards other factors like the physical feasibility of the implementation technique, cost and performance overheads etc. The sample case considered for the comparisons is presented in the next section.

In general, each IMPLY operation requires a single computational step, while a NAND operation requires two [29]. In Fig. 5.1, the IMPLY operation takes place simply as $q' \leftarrow (p \,\text{IMP}\, q)$, where the output is stored in memristor q, while p remains unchanged. Whereas, the NAND operation is a three-step process:

Step 0: $s = 0$;

Step 1: $p \rightarrow s$;

Step 2: $q \rightarrow s$.

Here, step 0 is essentially a RESET, and the output is stored in the memristor s.

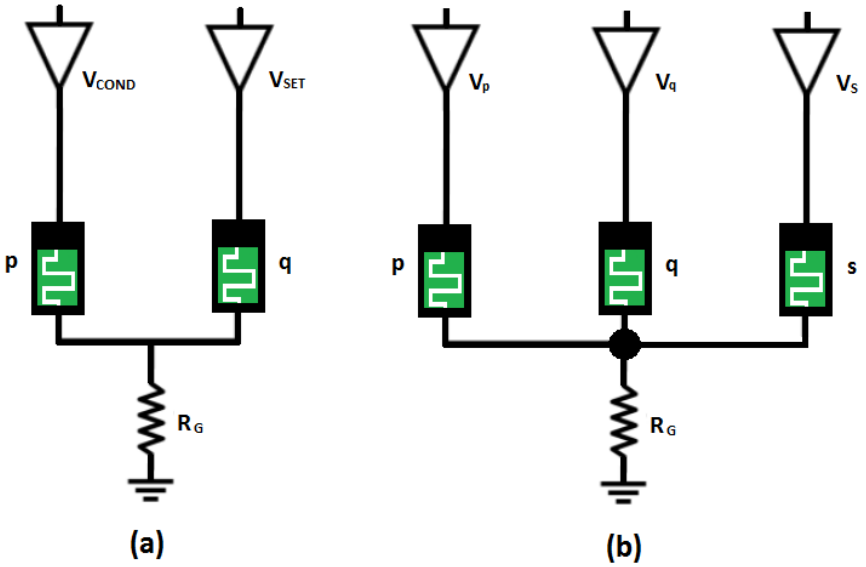

Figure 5.1. A schematic of (a) an IMPLY logic gate and (b) an IMPLY-based NAND gate

Sample Case: For a comparison between the proposed and previous methodology, a two bit full adder (FA) is considered here. In Fig. 5.2, the sum and carry-out functions are defined as follows:

$$S_0 = (A_0 \wedge \overline{B_0}) \vee (\overline{A_0} \wedge B_0) \tag{5.1}$$

$$C_1 = A_0 \wedge B_0 \tag{5.2}$$

$$S_1 = (A_1 \wedge B_1 \wedge C_1) \vee (\overline{A_1} \wedge \overline{B_1} \wedge C_1) \vee (\overline{A_1} \wedge B_1 \wedge \overline{C_1}) \vee (A_1 \wedge \overline{B_1} \wedge \overline{C_1}) \tag{5.3}$$

$$C_{out} = (A_1 \wedge B_1) \vee (B_1 \wedge C_1) \vee (A_1 \wedge C_1) \tag{5.4}$$

First, let's evaluate the previously proposed methodology, followed by the modification suggested by this work.

5.1 Previous Methodology

Lehtonen *et. al* [26, 29] were one of the first to describe a detailed logic synthesis technique for memristive implication. They proposed several schemes - All-NAND implementation (using

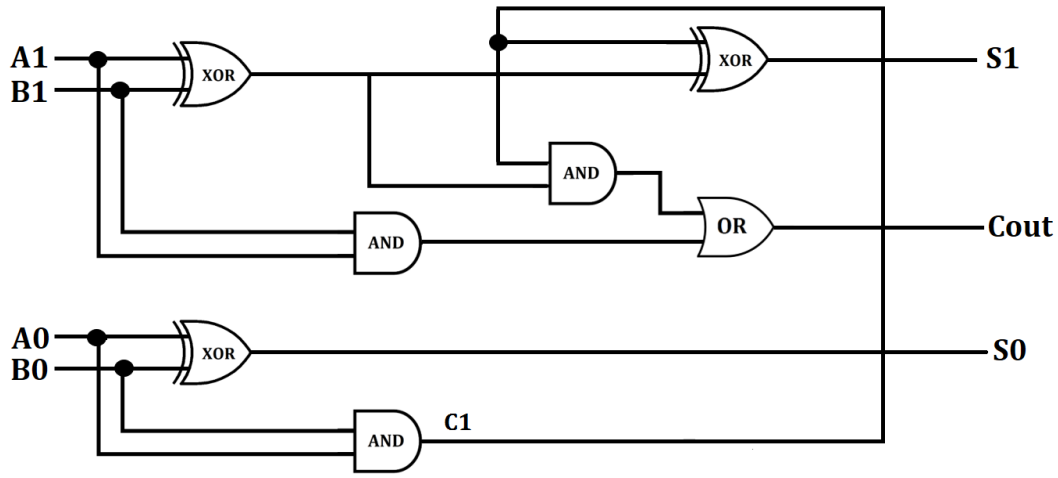

Figure 5.2. A schematic for a two bit full adder

IMPLY-based NANDs), complementary representation, NAND-OR, and multi-input representation. For the purpose of this document, we only consider the complementary all-NAND representation, since the others use logical operations beyond the IMPLY-NAND logic set. Further, comparing the all-NAND implementation without the complementary representation is expected to yield similar results for both the previous and proposed methods, and hence is not explicitly mentioned here.

The authors [29] propose an all-NAND realization of P_3 using complementary representation. Complementary representation means that each input and output variable has two dedicated memristors - one each for its inverted and non-inverted form (see Fig. 5.3 for an example). The universal nature of NAND gates entails that any given function can be implemented using a '2-depth' NAND form represented as,

$$f_{NAND} = NAND(x_1, x_2, ..., x_n) \tag{5.5}$$

where $x_1, x_2, ..., x_n$ might in-turn indicate a NAND of certain inputs. Thus, the all-NAND implementations of (5.1)-(5.4) look like:

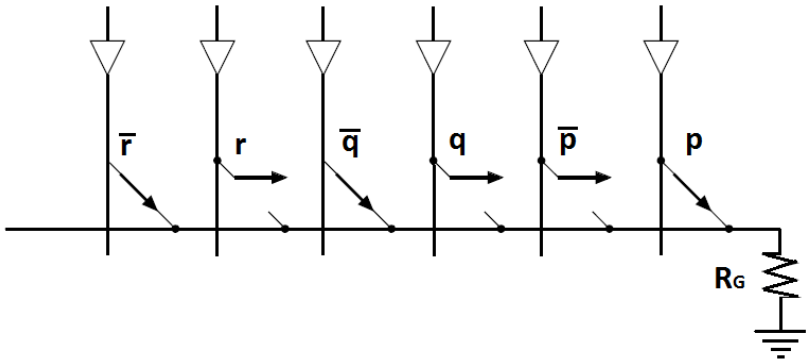

Figure 5.3. Schematic for a complementary representation scheme

$$S_{0(NAND)} = \overline{\overline{A_0 \wedge \overline{B_0}} \wedge \overline{\overline{A_0} \wedge B_0}} \tag{5.6}$$

$$S_{1(NAND)} = \overline{\overline{A_1 \wedge B_1 \wedge C_1} \wedge \overline{\overline{A_1} \wedge \overline{B_1} \wedge C_1} \wedge \overline{\overline{A_1} \wedge B_1 \wedge \overline{C_1}} \wedge \overline{A_1 \wedge \overline{B_1} \wedge \overline{C_1}}} \tag{5.7}$$

$$C_{out(NAND)} = \overline{\overline{(A_1 \wedge B_1)} \wedge \overline{(B_1 \wedge C_1)} \wedge \overline{(A_1 \wedge C_1)}} \tag{5.8}$$

To implement S_0, S_1 and C_{out}, 12 input memristors (two each for the six inputs), and two output memristors each for storing the three outputs (r_{1-6}) are required. However, since the inverted forms for all the inputs are readily available, there is no requirement for an auxiliary memristor [29]. Therefore, the computational sequences are listed below.

Computational Sequence for $S_{0(NAND)}$: The computation is divided into three sets of sequences - the first two compute the two NAND terms inside the overall NAND, while the third sequence performs the NAND of the two terms obtained from the first two sets of sequences.

$\overline{A_0 \wedge \overline{B_0}}$: $\mathtt{A_0} \rightarrow \mathtt{r_2}$;
$\overline{\mathtt{B_0}} \rightarrow \mathtt{r_2}$;
$\mathtt{r_2} \rightarrow \mathtt{r_1}$;
$\mathtt{r_2} = 0$;

$\overline{\overline{A_0} \wedge B_0}$: $\overline{\mathtt{A_0}} \rightarrow \mathtt{r_2}$;
$\mathtt{B_0} \rightarrow \mathtt{r_2}$;

$$\begin{aligned} & \mathtt{r}_2 \rightarrow \mathtt{r}_1; \\ & \mathtt{r}_2 = 0; \end{aligned}$$

$$S_{0(NAND)}: \quad \begin{aligned} & \mathtt{r}_1 \rightarrow \mathtt{r}_2; \\ & \mathtt{r}_1 \rightarrow \mathtt{r}_2; \end{aligned}$$

Finally, the desired result is stored in $\mathtt{r}_1$, while $\mathtt{r}_2$ contains the inverse of $\mathtt{r}_1$. Thus, the total number of computational steps required for $S_0 = 9$.

Computational Sequence for $\mathbf{S_{1(NAND)}}$: The computation is divided into five sets of sequences - the first four compute the four NAND terms inside the overall NAND, while the fifth sequence performs the NAND of all the four terms obtained from the first four sets of sequences.

$$\overline{A_1 \wedge B_1 \wedge C_1}: \quad \begin{aligned} & \mathtt{A}_1 \rightarrow \mathtt{r}_4; \\ & \mathtt{B}_1 \rightarrow \mathtt{r}_4; \\ & \mathtt{C}_1 \rightarrow \mathtt{r}_4; \\ & \mathtt{r}_4 \rightarrow \mathtt{r}_3; \\ & \mathtt{r}_4 = 0; \end{aligned}$$

$$\overline{\overline{A_1} \wedge \overline{B_1} \wedge C_1}: \quad \begin{aligned} & \mathtt{A}_1 \rightarrow \mathtt{r}_4; \\ & \overline{\mathtt{B}_1} \rightarrow \mathtt{r}_4; \\ & \overline{\mathtt{C}_1} \rightarrow \mathtt{r}_4; \\ & \mathtt{r}_4 \rightarrow \mathtt{r}_3; \\ & \mathtt{r}_4 = 0; \end{aligned}$$

$$\overline{\overline{A_1} \wedge B_1 \wedge \overline{C_1}}: \quad \begin{aligned} & \overline{\mathtt{A}_1} \rightarrow \mathtt{r}_4; \\ & \mathtt{B}_1 \rightarrow \mathtt{r}_4; \\ & \overline{\mathtt{C}_1} \rightarrow \mathtt{r}_4; \\ & \mathtt{r}_4 \rightarrow \mathtt{r}_3; \\ & \mathtt{r}_4 = 0; \end{aligned}$$

$$\overline{A_1 \wedge \overline{B_1} \wedge \overline{C_1}}: \quad \begin{aligned} & \overline{\mathtt{A}_1} \rightarrow \mathtt{r}_4; \\ & \overline{\mathtt{B}_1} \rightarrow \mathtt{r}_4; \\ & \mathtt{C}_1 \rightarrow \mathtt{r}_4; \\ & \mathtt{r}_4 \rightarrow \mathtt{r}_3; \\ & \mathtt{r}_4 = 0; \end{aligned}$$

$$S_{1(NAND)}: \quad \mathtt{r}_3 \rightarrow \mathtt{r}_4;$$

Therefore, the value of S_1 is stored in $\mathtt{r_3}$, while $\mathtt{r_4}$ contains the inverse of $\mathtt{r_3}$. Again, the total number of computational steps required for $S_1 = 21$.

Computational Sequence for $\mathbf{C_{out(NAND)}}$: The computation is divided into four sets of sequences - the first three compute the three NAND terms inside the overall NAND, while the fifth sequence performs the NAND of all the three terms obtained from the first three sets of sequences.

$$
\begin{aligned}
\overline{(A_1 \wedge B_1)}: \quad & \mathtt{A_1} \rightarrow \mathtt{r_6};\\
& \mathtt{B_1} \rightarrow \mathtt{r_6};\\
& \mathtt{r_6} \rightarrow \mathtt{r_5};\\
& \mathtt{r_6} = 0;\\
\overline{(B_1 \wedge C_1)}: \quad & \mathtt{B_1} \rightarrow \mathtt{r_6};\\
& \mathtt{C_1} \rightarrow \mathtt{r_6};\\
& \mathtt{r_6} \rightarrow \mathtt{r_5};\\
& \mathtt{r_6} = 0;\\
\overline{(A_1 \wedge C_1)}: \quad & \mathtt{A_1} \rightarrow \mathtt{r_6};\\
& \mathtt{C_1} \rightarrow \mathtt{r_6};\\
& \mathtt{r_6} \rightarrow \mathtt{r_5};\\
& \mathtt{r_6} = 0;\\
C_{out(NAND)}: \quad & \mathtt{r_5} \rightarrow \mathtt{r_6};
\end{aligned}
$$

Thus, the value of C_{out} is stored in $\mathtt{r_5}$, while $\mathtt{r_6}$ contains the inverse of $\mathtt{r_5}$. Again, the total number of computational steps required for $S_1 = 13$.

Thus we have the values of the three outputs stored in the output memristors r_1, r_3 and r_5, and the total number of computational steps required = 9 + 21 + 13 = 43.

5.2 Proposed Methodology

This section details the analysis of the IMPLY-NAND logic sets in terms of the number of required computational steps to be performed for the realization of a 2-bit full adder. I propose two improvements over the previous method, both of which result in a significant reduction in the computational length:

- *Ability to use IMPLY as a basis logic function:* As opposed to NANDs, implication uses a single step for computation, and therefore its increased use contribute towards step count reduction.

- *Ability to use some input memristors to store the result of computations:* The previous method does not allow for a change in the value of the input memristors. However, this leads to a drastic increase in the computational length, since additional steps are required to involve output memristors for the purpose of computation and for storing the result.

The Boolean algebra and optimization techniques proposed in this work are necessary to enable the above two improvements. This will be clearer as I analyze the full-adder more closely.

I use NOI representations of (5.1)-(5.4), which can be obtained by complementing these expressions twice and solving using De Morgan's law:

$$S_0 = \overline{(\overline{A_0} \vee B_0) \wedge (A_0 \vee \overline{B_0})} \tag{5.9}$$

Thus, (5.10) can be written as an NOI,

$$S_{0(NOI)} = \overline{(A_0 \rightarrow B_0) \wedge (\overline{A_0} \rightarrow \overline{B_0})} \tag{5.10}$$

Similarly, for (5.3) and (5.4),

$$S_{1(NOI)} = \{\overline{(A_1 \rightarrow (B_1 \rightarrow \overline{C_1})) \wedge (\overline{A_1} \rightarrow (\overline{B_1} \rightarrow \overline{C_1}))} \overline{\wedge (\overline{A_1} \rightarrow (B_1 \rightarrow C_1)) \wedge (A_1 \rightarrow (\overline{B_1} \rightarrow C_1))}\} \tag{5.11}$$

$$C_{out(NOI)} = \overline{(A_1 \rightarrow \overline{B_1}) \wedge (B_1 \rightarrow \overline{C_1}) \wedge (A_1 \rightarrow \overline{C_1})}. \tag{5.12}$$

It can be noted that the above implementation makes a better utilization of the basic IMPLY function. Further, using the IMPLY non-inverting associativity (Theorem 16), I propose specific modifications to S_1 and C_{out} that enable the use of input memristors to store computational results.

$$S_{1(NOI)} = \overline{(A_1 \rightarrow (B_1 \rightarrow \overline{C_1})) \wedge (C_1 \rightarrow (\overline{B_1} \rightarrow A_1))} \overline{\wedge (\overline{A_1} \rightarrow (B_1 \rightarrow C_1)) \wedge (\overline{C_1} \rightarrow (\overline{B_1} \rightarrow \overline{A_1}))} \quad (5.13)$$

$$C_{out(NOI)} = \overline{(A_1 \rightarrow \overline{B_1}) \wedge (C_1 \rightarrow \overline{B_1}) \wedge (A_1 \rightarrow \overline{C_1})} \quad (5.14)$$

The idea here is to maximize the number of input memristors that can reliably store computational results, such that these memristors are not used again elsewhere within the full adder.

Computational Sequence for $\mathbf{S_{0(NOI)}}$: The computation consists of four sequences - the first two perform the two IMPLY operations and store the results in B_0 and $\overline{B_0}$ memristors respectively, while the last couple of sequences compute the NAND.

$$\begin{aligned} A_0 \rightarrow B_0: &\quad \mathtt{A_0} \rightarrow \mathtt{B_0}; \\ \overline{A_0} \rightarrow \overline{B_0}: &\quad \overline{\mathtt{A_0}} \rightarrow \overline{\mathtt{B_0}}; \\ S_{0(NOI)}: &\quad \overline{\mathtt{B_0}} \rightarrow \mathtt{r_2}; \\ &\quad \mathtt{B_0} \rightarrow \mathtt{r_2}; \end{aligned}$$

Therefore, the desired result is stored in $\mathtt{r_2}$, and the total number of computational steps required for $S_0 = 4$.

Computational Sequence for $\mathbf{S_{1(NOI)}}$: The computation consists of five set of sequences - the first four perform the three variable IMPLY operations and store the results in r_3, A_1, C_1 and r_4 memristors respectively, while the last set of sequences compute the overall NAND.

$$
\begin{aligned}
A_1 \rightarrow (B_1 \rightarrow \overline{C_1}): \quad & \mathtt{C_1} \rightarrow \mathtt{r_4};\\
& \mathtt{B_1} \rightarrow \mathtt{r_4};\\
& \mathtt{A_1} \rightarrow \mathtt{r_4};\\
& \mathtt{r_4} \rightarrow \mathtt{r_3};\\
& \mathtt{r_4} = \mathtt{0};\\
C_1 \rightarrow (\overline{B_1} \rightarrow A_1): \quad & \overline{\mathtt{B_1}} \rightarrow \mathtt{A_1};\\
& \mathtt{C_1} \rightarrow \mathtt{A_1};\\
\overline{A_1} \rightarrow (B_1 \rightarrow C_1): \quad & \mathtt{B_1} \rightarrow \mathtt{C_1};\\
& \overline{\mathtt{A_1}} \rightarrow \mathtt{C_1};\\
\overline{C_1} \rightarrow (\overline{B_1} \rightarrow \overline{A_1}): \quad & \overline{\mathtt{B_1}} \rightarrow \overline{\mathtt{A_1}};\\
& \overline{\mathtt{C_1}} \rightarrow \overline{\mathtt{A_1}};\\
& \overline{\mathtt{A_1}} \rightarrow \mathtt{r_4};\\
S_{1(NOI)}: \quad & \mathtt{C_1} \rightarrow \mathtt{r_4};\\
& \mathtt{A_1} \rightarrow \mathtt{r_4};\\
& \mathtt{r_3} \rightarrow \mathtt{r_4};
\end{aligned}
$$

Therefore, the value of S_1 is stored in $\mathtt{r_4}$, and again, the total number of computational steps required for $S_1 = 15$.

Computational Sequence for $\mathbf{C_{out(NOI)}}$: The computation consists of four sets of sequences - the first three perform the IMPLY operations and store the results in r_5, $\overline{B_1}$ and $\overline{C_1}$ memristors respectively, while the last set of sequences compute the NAND.

$$
\begin{aligned}
A_1 \rightarrow \overline{B_1}: \quad & \mathtt{B_1} \rightarrow \mathtt{r_6};\\
& \mathtt{A_1} \rightarrow \mathtt{r_6};\\
& \mathtt{r_6} \rightarrow \mathtt{r_5};\\
& \mathtt{r_6} = \mathtt{0};\\
C_1 \rightarrow \overline{B_1}: \quad & \mathtt{C_1} \rightarrow \overline{\mathtt{B_1}};\\
A_1 \rightarrow \overline{C_1}: \quad & \mathtt{A_1} \rightarrow \overline{\mathtt{C_1}};\\
C_{out(NOI)}: \quad & \overline{\mathtt{C_1}} \rightarrow \mathtt{r_6};\\
& \overline{\mathtt{B_1}} \rightarrow \mathtt{r_6};\\
& \mathtt{r_5} \rightarrow \mathtt{r_6};
\end{aligned}
$$

Finally, the value of C_{out} is stored in $\mathtt{r_6}$, and the total number of computational steps required for $S_1 = 9$. Thus we have the values of the three outputs stored in the output memristors

r_2, r_4 and r_6, and the total number of computational steps required = 4 + 15 + 9 = 28.

Table 5.1 shows the summary of improvements marked by the proposed method.

Table 5.1. Comparative summary for the proposed and previous implementation of a 2-bit full adder in terms of computational length.

Function	**Previous Method**	**Proposed Method**	**Percentage Improvement**
S_0	9	4	55.56%
S_1	21	15	28.5%
C_{out}	13	9	30.76%
Total	**43**	**28**	**34.88%**

CHAPTER 6

AUTOMATED OPTIMIZATION ALGORITHM FOR ASYMMETRIC LOGIC FUNCTIONS

This chapter details a methodology to automate the process of logic synthesis and optimization for Boolean functions implemented using ALFs only. The objectives are clearly laid out, followed by possible implementation approaches and a real-life demonstration using the C++ programming language.

6.1 Objectives

Chapter 5 demonstrates that using asymmetric logic functions directly has significant advantages. Even though the much required set of algebraic identities and a modified Karnaugh map method is proposed in this work, a detailed algorithm is required for two primary purposes:

- *Objective #1:* Optimization of any Boolean function, simple or complex, represented using asymmetric logic sets.
- *Objective #2:* Automated translation between asymmetric and standard Boolean functions.

6.2 Implementation Approaches

There are two possible approaches to achieve the overall optimization of an asymmetric Boolean function:

- *Approach #1 (Accomplish objectives #1 and #2 in combination):* This approach first translates the input function (which is an asymmetric Boolean function like SOI/IOS or NOI/ION) into a standard Boolean function (like SOP/POS). This step is followed

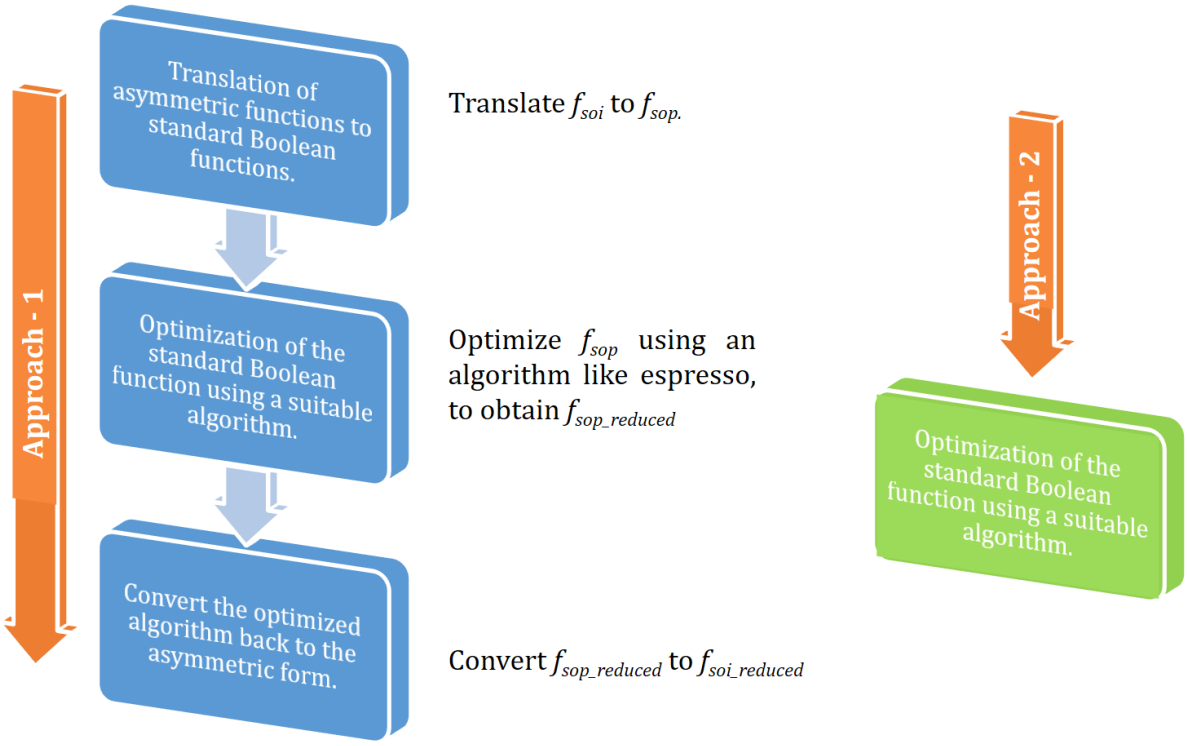

Figure 6.1. Graphical explanation of (a) approach #1 and (b) approach #2. Side notes detail an example.

by the optimization of the translated function using an existing algorithm, and then a translation back to the original asymmetric form.

- *Approach #2 (Accomplish objective #1 directly):* This involves developing a new optimization approach for asymmetric logic sets.

Fig. 6.1 explains the two approaches graphically. It is clear that approach #2 has considerable advantages in terms of implementation efficiency, but developing a new algorithm from the ground up requires strenuous research and development. Moreover, the translation overhead of approach #1 might not be significant, and thus the extra effort of rebuilding the algorithm might not be worth it. Possibly, a similar approach might work as well for asymmetric functions as any other standard Boolean function. However, considering the atypical

behavior of ALFs, there is a possibility that the results of adopting approach #2 might lead to a improved optimization scheme. This forms one of the primary future objectives of my work.

6.2.1 Realization of Approach #1

In this thesis document, I focus on implementing the first approach. Objective #1 can be achieved various methods proposed over past several years [59–66]. One of these methods is particularly popular, the Espresso algorithm [66]. The operation of this algorithm consists of three fundamental steps [66]:

- *Expand:* A cube is expanded until no further expansion is possible without including a vertex of the off-set, *i.e.*, the cube is expanded until it is prime. This operation involves complementing each input, in turn, to test if the new vertex is a member of the off-set or the on-set of the Boolean expression.

- *Irredundant Cover:* This procedure attempts to reduce the number of prime cubes to a minimum so that there are no redundant prime cubes covering the Boolean function.

- *Reduce:* The expand and irredundant cover procedures will give a locally optimal solution which may not be the global optimum solution, thus a reduce procedure is required to transform a prime cover into a new cover by replacing each cube by a smaller cube contained within it.

The above routine is iterated until there is no improvement in the optimality of the reduction. This approach has been adopted widely and several improvements have been made over the years.

Objective #2 is accomplished using a simple algorithm proposed in this work. The aim of this algorithm is to convert any standard Boolean representation (*e.g.* SOP, POS) to a

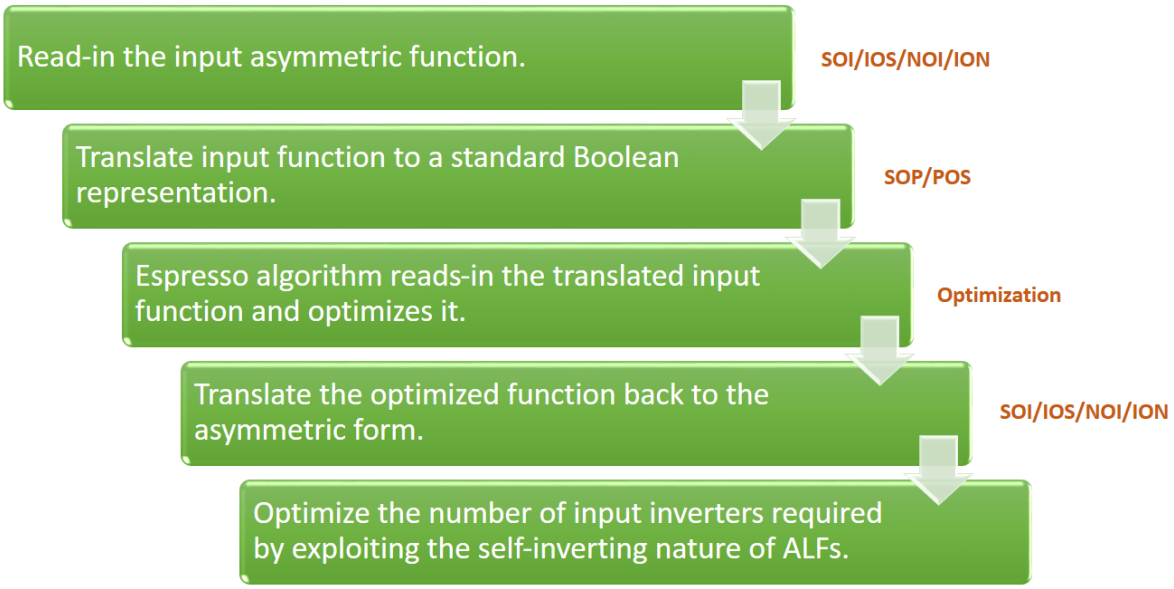

Figure 6.2. Overall flow of the proposed algorithm.

function implemented using ALFs only. The simplest approach of doing so is to detect the non-inverted input in each of the product (or sum) terms, and complement the value of all the remaining inputs (*i.e.* to produce the inverted inputs).

6.3 Practical Implementation

As mentioned in the previous section. The proposed algorithm uses the Espresso method for Boolean optimization. One the most popular open-source implementation of Espresso is the University of California - Berkeley's 'MVSIS' logic synthesis and verification tool [67]. Alongside Espresso, it includes several other packages for a variety of operations.

Following approach #1, the espresso algorithm fulfills objective #1, while a basic algorithm proposed here completes the back and forth translations between asymmetric and symmetric functions. The overall flow of the approach is detailed in Fig. 6.2. Algorithm 1 presents a simplistic pseudo-code for the translation of and SOP to an SOI. Suppose, the espresso algorithm returns the final result through the variable 'F'. This variable is then read in directly by the algorithm proposed below.

Algorithm 1: Receive optimized function from espresso and translate to SOI

while *F has data* **do**
| $testData \leftarrow get\_data(F)$;

end

while *testdata* **do**
| $testData \leftarrow convert\_soi(F)$;

end

while *testdata* **do**
| $finalOutput \leftarrow soi\_to\_literal(testData)$;

end

return*finalOutput*;

The optimized product/sum terms are received from the Espresso algorithm as strings of bits. Selected bits are complemented so as to appropriately depict the inverted inputs of an asymmetric function. In the specific case of Algorithm 1, all of the individual bits are complemented except for the most significant one, thus obtaining the inverted and non-inverted inputs of an SOI. The methodology can be applied to both ends of the Espresso optimization step, which completes the overall flow. In addition to this, the proposed algorithm minimizes the number of input inverters required by exploiting the self-inverting nature of ALFs. For example, $(\overline{A} \wedge \overline{B})$ is better implemented as $(B \wedge A)$.

Fig. 6.3 - 6.5 show the actual user interface and results of the implementation. The input for the espresso tool is a `.pla` file (Fig. 6.3) which describes the Boolean function that needs to be optimized. Fig. 6.4 and 6.5 show the input and output user-end displays respectively. Therefore, the algorithm has a demonstrated capability of optimizing any asymmetric logic function.

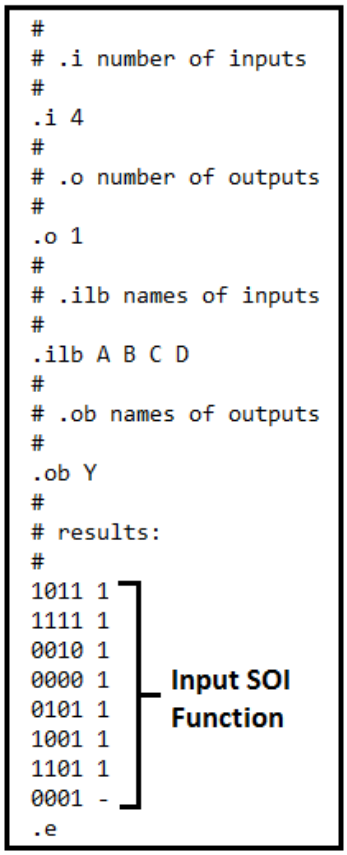

Figure 6.3. The `.pla` file used to describe the input function (which in this case is an SOI)

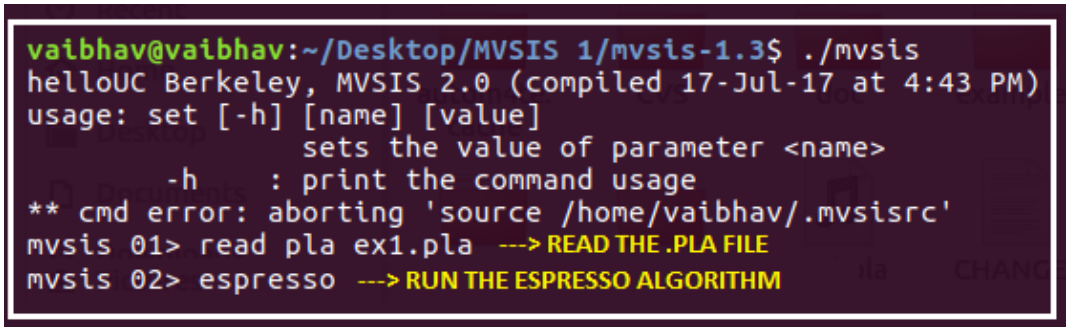

Figure 6.4. Command terminal showing the read-in of the input `.pla` file and execution of the algorithm using the 'espresso' command

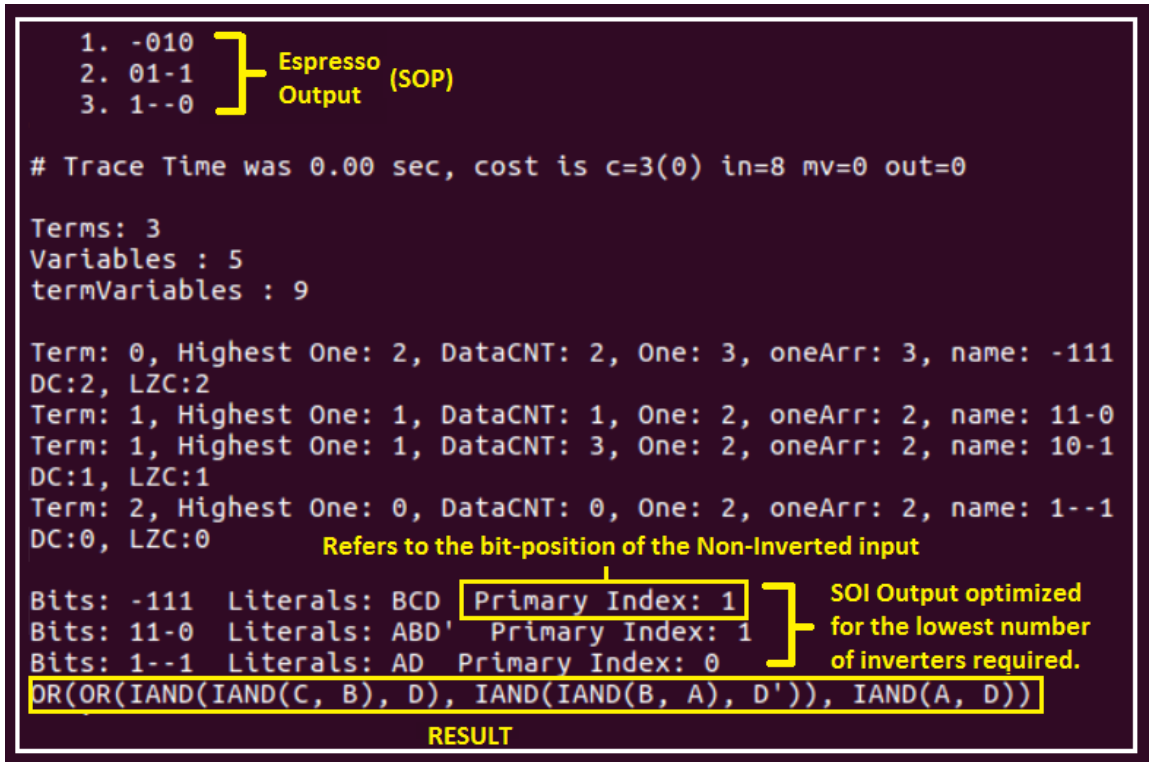

Figure 6.5. The final output window showing the result of Espresso optimization and the translated sum of IANDs. Espresso turns out the corresponding optimized equivalent for the input as an SOP, which is then translated back to SOI. The primary index refers to the non-inverted bit of each term.

CHAPTER 7

CONCLUSIONS

In this thesis, I have highlighted a novel approach to logic synthesis for logic sets formulated via memristive and spintronic devices. I propose the idea of asymmetric logic functions, with a particular emphasis on the inverted-AND and stateful memristive implication logic.

A complete Boolean algebra for both spintronic IAND logic and memristive IMPLY logic is presented. This includes all the primary identities and principles required to perform Boolean reduction. Further, I detail a simple logic minimization technique that enables the direct mapping of memristive and spintronic logic functions onto Karnaugh maps. This method is tailored to both the asymmetric IAND-OR and IMPLY-NAND logic sets. In fact, any asymmetric logic function of a similar form can be expected to fall into the broader category of ALFs to which a similar Boolean algebra and minimization techniques can be applied.

This logic minimization technique provides a foundation for logic synthesis tools based on memristors and spintronic logic, as well as a template for logic minimization with alternative beyond-CMOS computing structures. This Karnaugh map method adapted to non-commutative logic functions thus constitutes an important step toward the development of design automation techniques for the next generation of computing.

This work presents a comparative analysis of the proposed and previous logic synthesis approaches, demonstrating a statistical advantage of considering ALFs as basic logic functions. This is aimed to provide a path parallel to the traditional strategy of mapping all unconventional logic functions to universal or standard ones. Developing on the proposed method can potentially reduce implementation overheads to a significant degree.

Finally, I introduce a simplistic algorithm for the automation of logic minimization and translations between standard and asymmetric logic sets. Currently, the algorithm builds on the existing Espresso heuristic logic minimizer, with the added sequence of translations.

However, the computational overheads for these additional sequences are not expected to be too much. Nevertheless, a successful implementation of approach #2 should be promising.

7.1 Future Work

I am determined to take my work to the next level and achieve the following objectives in the near future:

- Extend the Karanugh mapping process to better and more recent optimization techniques like Quine-McCluskey, cube-reduction, and binary decision diagram based methods etc. In this process, I hope to come up with an approach that works best for asymmetric and other unconventional logic sets.
- Explore the advantages of ALFs in sequential circuits like flip-flops.
- Implement approach #2 for the optimization of ALFs. This will eliminate the need for the back and forth translation to and from standard Boolean functions.
- Benchmark the existing algorithm and evaluate its performance the algorithm's performance against conventional CMOS counterparts.

Overall, I hope my work will have a positive impact on academia, and help towards designing more efficient circuits for emerging devices aimed to replace CMOS.

APPENDIX

SUMMARY OF BOOLEAN LAWS AND IDENTITIES FOR ASYMMETRIC LOGIC FUNCTIONS

This appendix presents a tabular summary of all the properties mentioned in this chapter. Table A.1 and A.3 display the core algebraic and standard Boolean theorems respectively, for IAND logic. Whereas, Table A.2 and A.4 do the same for the IMPLY operation.

Table A.1. Core algebraic identities for IAND logic

Identity Name	**Expression**
Annulment	$A \wedge 1 = 0 \wedge \overline{A} = 0$
Inversion	$1 \wedge A = \overline{A}$
Identity	$A \wedge 0 = A$
Null Idempotency	$A \wedge A = 0$
Inverse Idempotency - I	$A \wedge \overline{A} = A$
Inverse-Idempotency - II	$\overline{A} \wedge A = \overline{A}$

Table A.2. Core algebraic identities for implication logic

Identity Name	**Expression**
Annulment	$A \rightarrow 1 = 1$
Inversion	$A \rightarrow 0 = \overline{A}$
Identity	$1 \rightarrow A = A$
Null Idempotency	$A \rightarrow A = 1$
Inverse-Idempotency - I	$A \rightarrow \overline{A} = \overline{A}$
Inverse-Idempotency - II	$\overline{A} \rightarrow A = A$

Table A.3. Boolean algebraic laws for IAND logic

Identity Name	**Expression**
Commutative Law	$A \barwedge B \neq B \barwedge A$
Conventional Non-Associativity	$(A \barwedge B) \barwedge C \neq A \barwedge (B \barwedge C)$
IAND Non-Inverting Associativity	$(A \barwedge B) \barwedge C = (A \barwedge C) \barwedge B$
IAND Inverting Associativity	$(A \barwedge B) \barwedge C = (\overline{C} \barwedge \overline{A}) \barwedge B$
IAND Distributive Law - I	$A \barwedge (B \wedge C) = A \wedge \overline{B \wedge C} = A \wedge (\overline{B} \vee \overline{C})$
IAND Distributive Law - II	$(A \barwedge B) \wedge C = (A \barwedge B) \barwedge \overline{C}$
IAND Distributive Law - III	$(A \vee B) \barwedge C = (A \barwedge C) \vee (B \barwedge C)$
IAND Distributive Law - IV	$A \barwedge (B \vee C) = (A \barwedge B) \wedge (A \barwedge C)$
IAND Distributive Law - V	$A \wedge (B \barwedge C) = (A \wedge B) \barwedge C = (A \barwedge \overline{B}) \barwedge C$
IAND Distributive Law - VI	$A \vee (B \barwedge C) = (A \vee B) \wedge (A \vee \overline{C})$
IAND Distributive Law - VII	$(A \barwedge B) \vee C = (A \vee C) \wedge (\overline{B} \vee C)$
De Morgan's Law	$\overline{(A \barwedge B) \barwedge C} = \overline{A} \vee B \vee C$
De Morgan's Law (converse)	$\overline{A \vee B \vee C} = (\overline{A} \barwedge B) \barwedge C$

Table A.4. Boolean algebraic laws for implication logic

Identity Name	**Expression**
Commutative Law	$A \to B \neq B \to A$
Conventional Non-Associativity	$(A \to B) \to C \neq A \to (B \to C)$
IMPLY Non-Inverting Associativity	$A \to (B \to C) = B \to (A \to C)$
IMPLY Inverting Associativity	$A \to (B \to C) = B \to (\overline{C} \to \overline{A})$
IMPLY Distributive Law - I	$A \to (B \wedge C) = (A \to B) \wedge (A \to C)$
IMPLY Distributive Law - II	$(A \vee B) \to C = (A \to C) \wedge (B \to C)$
IMPLY Distributive Law - III	$A \vee (B \to C) = \overline{A} \to (B \to C)$
IMPLY Distributive Law - IV	$A \to (B \vee C) = (A \to B) \vee C = A \to (\overline{B} \to C)$
IMPLY Distributive Law - V	$(A \wedge B) \to C = A \to (B \to C)$
IMPLY Distributive Law - VI	$(A \to B) \wedge C = (\overline{A} \wedge C) \vee (B \wedge C)$
IMPLY Distributive Law - VII	$A \wedge (B \to C) = (A \wedge \overline{B}) \vee (A \wedge C)$
De Morgan's Law	$\overline{A \to (B \to C)} = A \wedge B \wedge \overline{C}$
De Morgan's Law (converse)	$\overline{A \wedge B \wedge C} = A \to (B \to \overline{C})$

BIBLIOGRAPHY

[1] David M Bromberg, Daniel H Morris, Larry Pileggi, and Jian-Gang Zhu. Novel stt-mtj device enabling all-metallic logic circuits. *IEEE transactions on Magnetics*, 48(11):3215–3218, 2012.

[2] J. S. Friedman and A. V. Sahakian. Complementary magnetic tunnel junction logic. *IEEE Transactions on Electron Devices*, 61(4):1207–1210, 2014.

[3] Dan A Allwood, Gang Xiong, CC Faulkner, D Atkinson, D Petit, and RP Cowburn. Magnetic domain-wall logic. *Science*, 309(5741):1688–1692, 2005.

[4] KA Omari and TJ Hayward. Chirality-based vortex domain-wall logic gates. *Physical Review Applied*, 2(4):044001, 2014.

[5] Peng Xu, Ke Xia, Changzhi Gu, Ling Tang, Haifang Yang, and Junjie Li. An all-metallic logic gate based on current-driven domain wall motion. *Nature nanotechnology*, 3(2), 2008.

[6] A. Imre, G. Csaba, L. Ji, A. Orlov, G. H. Bernstein, and W. Porod. Majority logic gate for magnetic quantum-dot cellular automata. *Science*, 311(5758):205–208, 2006.

[7] Joseph S Friedman, Anuj Girdhar, Ryan M Gelfand, Gokhan Memik, Hooman Mohseni, Allen Taflove, Bruce W Wessels, Jean-Pierre Leburton, and Alan V Sahakian. Cascaded spintronic logic with low-dimensional carbon. *Nature communications*, 8:15635, 2017.

[8] Supriyo Datta and Biswajit Das. Electronic analog of the electro-optic modulator. *Applied Physics Letters*, 56(7):665–667, 1990.

[9] Hyun Cheol Koo, Jae Hyun Kwon, Jonghwa Eom, Joonyeon Chang, Suk Hee Han, and Mark Johnson. Control of spin precession in a spin-injected field effect transistor. *Science*, 325(5947):1515–1518, 2009.

[10] John Schliemann, J Carlos Egues, and Daniel Loss. Nonballistic spin-field-effect transistor. *Physical review letters*, 90(14):146801, 2003.

[11] Baigeng Wang, Jian Wang, and Hong Guo. Quantum spin field effect transistor. *Physical Review B*, 67(9):092408, 2003.

[12] J. S. Friedman, E. R. Fadel, B. W. Wessels, D. Querlioz, and A. V. Sahakian. Bilayer avalanche spin-diode logic. *AIP Advances*, 5(11):117102, 2015.

[13] J. S. Friedman, B. W. Wessels, G. Memik, and A. V. Sahakian. Emitter-coupled spin-transistor logic: Cascaded spintronic computing beyond 10 ghz. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(1):17–27, March 2015.

[14] Joseph S Friedman, John A Peters, Gokhan Memik, Bruce W Wessels, and Alan V Sahakian. Emitter-coupled spin-transistor logic. *Journal of Parallel and Distributed Computing*, 74(6):2461–2469, 2014.

[15] Joseph S Friedman, Nikhil Rangaraju, Yehea I Ismail, and Bruce W Wessels. A spin-diode logic family. *IEEE Transactions on Nanotechnology*, 11(5):1026–1032, 2012.

[16] Mayler Martins, Felipe Marranghello, Joseph Friedman, Alan Sahakian, Renato Ribas, and Andre Reis. Enhanced spin-diode synthesis using logic sharing. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 218–224. IEEE, 2015.

[17] Joseph S Friedman, Bruce W Wessels, Damien Querlioz, and Alan V Sahakian. High-performance computing based on spin-diode logic. In *Spintronics VII*, volume 9167, page 91671J. International Society for Optics and Photonics, 2014.

[18] Joseph S Friedman, Nikhil Rangaraju, Yehea I Ismail, and Bruce W Wessels. Inmnas magnetoresistive spin-diode logic. In *Proceedings of the great lakes symposium on VLSI*, pages 209–214. ACM, 2012.

[19] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, Nov 1953.

[20] M. Karnaugh. Pulse-switching circuits using magnetic cores. *Proceedings of the IRE*, 43(5):570–584, 1955.

[21] M. T. Niemier, G. H. Bernstein, G. Csaba, A. Dingler, X. S. Hu, S. Kurtz, S. Liu, J. Nahas, W. Porod, M. Siddiq, and E. Varga. Nanomagnet logic: progress toward system-level integration. *Journal of Physics: Condensed Matter*, 23(49):493202, 2011.

[22] D. Hampel and Robert O. Winder. Threshold logic. *IEEE Spectrum*, 8(5):32–39, 1971.

[23] A. Imre. Majority Logic Gate for Magnetic Quantum-Dot Cellular Automata. *Science*, 311(5758):205–208, 2006.

[24] J. S. Friedman, A. Girdhar, R. M. Gelfand, G. Memik, H. Mohseni, A. Taflove, B. W. Wessels, J.P. Leburton, and A. V. Sahakian. Cascaded spintronic logic with low-dimensional carbon. *Nature communications*, 8:15635, 2017.

[25] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, and U. C. Kolodny, A.and Weiser. Memristor-based material implication (IMPLY) logic: Design principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2014.

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

[27] D. E. Nikonov, G. I. Bourianoff, and T. Ghani. Proposal of a spin torque majority gate logic. *IEEE Electron Device Letters*, 32(8):1128–1130, 2011.

[28] M. Haykel Ben-Jamaa, Kartik Mohanram, and Giovanni De Micheli. An efficient gate library for ambipolar CNTFET logic. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 30, pages 242–255, 2011.

[29] E. Lehtonen, J. Poikonen, and M. Laiho. Implication logic synthesis methods for memristors. In *2012 IEEE International Symposium on Circuits and Systems*, pages 2441–2444, May 2012.

[30] A. Chattopadhyay and Z. Rakosi. Combinational logic synthesis for material implication. *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011*, pages 200–203, 2011.

[31] A. Raghuvanshi and M. Perkowski. Logic synthesis and a generalized notation for memristor-realized material implication gates. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 470–477, Piscataway, NJ, USA, 2014. IEEE Press.

[32] S. Shirinzadeh, M. Soeken, and R. Drechsler. Multi-objective BDD optimization for RRAM based circuit design. In *Formal Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2016*, 2016.

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

[34] D. Fan and K. Sharad, M.and Roy. Design and synthesis of ultralow energy spin-memristor threshold logic. *IEEE Transactions on Nanotechnology*, 13(3):574–583, 2014.

[35] L. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, September 1971.

[36] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 459(7250):1154–1154, 2009.

[37] Y. Yang, P. Gao, S. Gaba, T. Chang, X. Pan, and W. Lu. Observation of conducting filament growth in nanoscale resistive memories. *Nature Communications*, 3:732, 2012.

[38] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams. 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

[39] J. F. Miller. Principles in the Evolutionary Design of Digital Circuits - Part I. *Genetic Programming and Evolvable Machines*, 1:7–35, 2000.

[40] Ioannis Vourkas and Georgios Ch Sirakoulis. Emerging memristor-based logic circuit design approaches: A review. *IEEE Circuits and Systems Magazine*, 16(3):15–30, 2016.

[41] Ioannis Vourkas and Georgios Ch Sirakoulis. Memristor-based combinational circuits: A design methodology for encoders/decoders. *Microelectronics Journal*, 45(1):59–70, 2014.

[42] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[43] Georgios Papandroulidakis, Ioannis Vourkas, Nikolaos Vasileiadis, and Georgios Ch Sirakoulis. Boolean logic operations and computing circuits based on memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(12):972–976, 2014.

[44] Somnath Paul and Swarup Bhunia. A scalable memory-based reconfigurable computing framework for nanoscale crossbar. *IEEE transactions on Nanotechnology*, 11(3):451–462, 2012.

[45] Matthew M Ziegler and Mircea R Stan. Cmos/nano co-design for crossbar-based molecular electronic systems. *IEEE Transactions on Nanotechnology*, 2(4):217–230, 2003.

[46] Gregory S Snider and R Stanley Williams. Nano/cmos architectures using a field-programmable nanowire interconnect. *Nanotechnology*, 18(3):035204, 2007.

[47] Hao Yan, Hwan Sung Choe, SungWoo Nam, Yongjie Hu, Shamik Das, James F Klemic, James C Ellenbogen, and Charles M Lieber. Programmable nanowire circuits for nanoprocessors. *Nature*, 470(7333):240, 2011.

[48] Yuriy V Pershin and Massimiliano Di Ventra. Self-organization and solution of shortest-path optimization problems with memristive networks. *Physical Review E*, 88(1):013305, 2013.

[49] Feijun Jiang and Bertram E Shi. The memristive grid outperforms the resistive grid for edge preserving smoothing. In *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*, pages 181–184. IEEE, 2009.

[50] Yuriy V Pershin and Massimiliano Di Ventra. Solving mazes with memristors: A massively parallel approach. *Physical Review E*, 84(4):046703, 2011.

[51] Ioannis Vourkas and Georgios Ch Sirakoulis. On the generalization of composite memristive network structures for computational analog/digital circuits and systems. *Microelectronics Journal*, 45(11):1380–1391, 2014.

[52] Tejinder Singh. Hybrid memristor-cmos (memos) based logic gates and adder circuits. *arXiv preprint arXiv:1506.06735*, 2015.

[53] Julien Borghetti, Zhiyong Li, Joseph Straznicky, Xuema Li, Douglas AA Ohlberg, Wei Wu, Duncan R Stewart, and R Stanley Williams. A hybrid nanomemristor/transistor logic circuit capable of self-programming. *Proceedings of the National Academy of Sciences*, 106(6):1699–1703, 2009.

[54] Shahar Kvatinsky, Nimrod Wald, Guy Satat, Avinoam Kolodny, Uri C Weiser, and Eby G Friedman. Mrl—memristor ratioed logic. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*, pages 1–6. IEEE, 2012.

[55] Garrett S Rose, Jeyavijayan Rajendran, Harika Manem, Ramesh Karri, and Robinson E Pino. Leveraging memristive systems in the construction of digital logic circuits. *Proceedings of the IEEE*, 100(6):2033–2049, 2012.

[56] Ligang Gao, Fabien Alibart, and Dmitri B Strukov. Programmable cmos/memristor threshold logic. *IEEE Transactions on Nanotechnology*, 12(2):115–119, 2013.

[57] Jeyavijayan Rajendran, Harika Manem, Ramesh Karri, and Garrett S Rose. Memristor based programmable threshold logic array. In *Proceedings of the 2010 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 5–10. IEEE Press, 2010.

[58] Ioannis Vourkas and Georgios Ch Sirakoulis. A novel design and modeling paradigm for memristor-based crossbar circuits. *IEEE Transactions on Nanotechnology*, 11(6):1151–1159, 2012.

[59] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[60] Robert K Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, 1987.

[61] Ellen M Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R Stephan, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. 1992.

[62] Congguang Yang and Maciej Ciesielski. Bds: A bdd-based logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(7):866–876, 2002.

[63] Kwang-Ting Cheng and Luis A Entrena. Multi-level logic optimization by redundancy addition and removal. In *Design Automation, 1993, with the European Event in ASIC Design. Proceedings.[4th] European Conference on*, pages 373–377. IEEE, 1993.

[64] Yusuke Matsunaga and Masahiro Fujita. Multi-level logic optimization using binary decision diagrams. In *ICCAD*, volume 89, pages 556–559, 1989.

[65] Wolfgang Kunz and Prem R Menon. Multi-level logic optimization by implication analysis. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 6–13. IEEE Computer Society Press, 1994.

[66] Patrick C McGeer, Jagesh V Sanghavi, Robert K Brayton, and AL Sangiovanni-Vicentelli. Espresso-signature: A new exact minimizer for logic functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(4):432–440, 1993.

[67] BSIM Group at UC Berkeley. Berkeley short channel igfet model, 2011.

BIOGRAPHICAL SKETCH

Vaibhav Vyas was born in Bhopal, Madhya Pradesh, India in the September of 1993. He completed his high school in 2012 from Saint Xavier's Senior Secondary School, Bhopal. After that, he completed his Bachelor of Engineering (B.Tech) degree in Electronics and Communication Engineering with honors from Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal, Madhya Pradesh, India in May 2016. During the course of his undergraduate degree, he undertook several university and industrial level trainings in the field of VLSI design, Embedded systems and consumer mobility. He joined The University of Texas at Dallas for his Master of Science in Electrical Engineering (M.S.E.E) in Fall 2016. In January 2017, he joined the NanoSpinCompute Lab at UT Dallas under Dr. Joseph S. Friedman, where he began his research in logic design automation and device performance analysis for emerging device technologies. He aims to continue making contributions to the academia and develop robust mechanisms for automated logic synthesis and optimization of unconventional logic paradigms.

CURRICULUM VITAE

Vaibhav Vyas

Contact Information:

Department of Electrical and
Computer Engineering,
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Voice: (214) 280-9971
Email: vaibhav.vyas@utdallas.edu

Educational History:

Bachelor of Engineering (Electronics and Communication), Rajiv Gandhi Technical University, Bhopal (MP), India, 2016

Employment History:

Graduate Student Researcher, NanoSpinCompute Lab, The University of Texas at Dallas (January 2017 – May 2018).

Publications:

1. V. Vyas, L. Jiang-Wei, X. Hu, J.S. Friedman, "Karnaugh Map Method for Memristive and Spintronic Asymmetric Basis Logic Functions" (in preparation).

2. V. Vyas, L. Jiang-Wei, J.S. Friedman, "Boolean Algebra for Memristive and Spintronic Asymmetric Basis Logic Functions" (in preparation).

3. V. Vyas, J.S. Friedman, "Design of Sequential Circuits using Bilayer Avalanche Spin-Diode Logic" (in preparation).

4. N. Hassan, M. Joslin, V. Vyas, A.G. Pai, J.S. Friedman, "Performance Analysis of the All-Spin Logic Device" (in preparation).