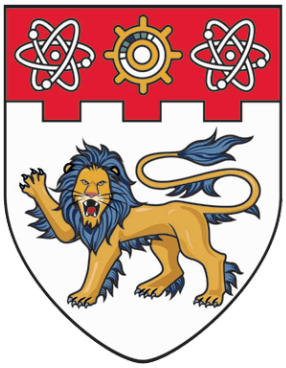


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# ARCHITECTURES AND AUTOMATION FOR BEYOND-CMOS TECHNOLOGIES


DEBJYOTI BHATTACHARJEE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING


2018

# ARCHITECTURES AND AUTOMATION FOR BEYOND-CMOS TECHNOLOGIES

**DEBJYOTI BHATTACHARJEE**

School Of Computer Science And Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy.

**2018**

If you would not be forgotten as soon as you are dead,
either write something worth reading or
do things worth writing.
— Benjamin Franklin

*To my parents.*

## Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

.....

**Debjyoti Bhattacharjee**
Hardware and Embedded Systems Laboratory (HESL)
School of Computer Science and Engineering,
Nanyang Technological University, Singapore.

Date : *12 December, 2018*

## Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

.....

**Dr.-Ing. Anupam Chattopadhyay**
Assistant Professor
School of Computer Science and Engineering,
Nanyang Technological University, Singapore.

Date : *12 December, 2018*

# Authorship Attribution Statement

This thesis contains material from 7 papers published in the following peer-reviewed conferences/journals and 2 preprints where I was the first and/or corresponding author.

The content of Chapter 3 has been published as —

- *ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting, Debjyoti Bhattacharjee, Rajeshwari Devadoss, and Anupam Chattopadhyay in Design, Automation & Test, in Europe Conference & Exhibition (DATE), 2017.*
  - I designed the algorithms with R. Devadoss, followed by implementation of the algorithms and benchmarking them.
  - A. Chattopadhyay provided the initial project direction.
  - The manuscript was prepared by me and edited by the co-authors.
- *SHA-3 Implementation Using ReRAM based In-Memory Computing Architecture, Debjyoti Bhattacharjee, Vikramkumar Pudi and Anupam Chattopadhyay, in 2017 18th International Symposium on Quality Electronic Design (ISQED).*
  - V. Pudi did the study of the SHA-3 algorithm and the existing hardware implementations.
  - I implemented the algorithm, along with carrying out experimental studies for the results. The manuscript was written by me.
  - A. Chattopadhyay guided the work and edited the manuscript.

The content of Chapter 4 has been published as —

- *Delay-Optimal Technology Mapping for In-Memory Computing using ReRAM Devices, Debjyoti Bhattacharjee and Anupam Chattopadhyay, in International Conference on Computer Aided Design (ICCAD), 2016.*
  - I undertook the study of the background of the problem, designed the algorithm in consultation with A.Chattopadhyay and implemented the solution.
  - The manuscript was written by me and edited by A. Chattopadhyay.
- *Area-constrained Technology Mapping for In-Memory Computing using ReRAM devices, Debjyoti Bhattacharjee, Arvind Easwaran and Anupam Chattopadhyay, in Asia and South Pacific Design Automation (ASP-DAC), 2017.*
  - I undertook the study of the background of the problem, designed the algorithm in consultation with A. Easwaran and implemented the solution.
  - The manuscript was written by me and edited by A. Chattopadhyay.

The content of Chapter 5 is available as preprint — *Crossbar-Constrained Technology Mapping for ReRAM based In-Memory Computing, Debjyoti Bhattacharjee, Yaswanth Tavva, Arvind Easwaran, and Anupam Chattopadhyay. arXiv preprint arXiv:1809.08195 (2018).*

– I and A. Easwaran studied the problem and derived theoretical bounds for the problem solution. In addition, we designed the algorithms for practical solution to the problem.

– Y. Tavva implemented a portion of the algorithm and assisted in writing and preparing figures for the manuscript.

– A. Chattopadhyay guided the problem solution with his insights from the solution of other related problems.

– The manuscript was prepared by me and edited by the co-authors.

The content of Chapter 6 has been published as — *Technology-Aware Logic Synthesis for ReRAM based In-Memory Computing, Debjyoti Bhattacharjee, Luca Amaru and Anupam Chattopadhyay, in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.*

– The problem was identified and formalized by me, along with insights to the optimization parameters.

– L. Amaru implemented the optimization techniques and optimized the benchmarks.

– The optimized benchmarks were used by me as an input to the back-end of the automation flow for generating the results.

– The manuscript was prepared by me, with inputs from L. Amaru.

– A. Chattopadhyay guided the work and edited the manuscript.

The content of Chapter 7 has been published as —

- *Multi-valued and Fuzzy Logic Realization using $TaO_x$ Memristive Devices, Debjyoti Bhattacharjee, Wonjoo Kim, Anupam Chattopadhyay, Rainer Waser, and Vikas Rana, in Scientific reports 8, no. 1 (2018): 8.*
  - I designed the experiments, interpreted the data and prepared the manuscript.
  - W. Kim prepared the devices and performed the measurements on the devices.
  - A. Chattopadhyay and V. Rana conceived the idea, initiated and supervised the research and co-wrote the manuscript.
  - R. Waser initiated and supervised the research.
- *Single output multi-valued logic function synthesis using Łukaszewicz logic operators, Anmol Prakash, Debjyoti Bhattacharjee, and Anupam Chattopadhyay. In IEEE International Symposium on Multiple-Valued Logic (ISMVL), 2018.*

- A. Chattopadhyay conceived the idea and supervised the research.
- I initiated the solution approach and helped A. Prakash in formalizing the solution and co-wrote the manuscript.
- A. Prakash implemented the solution and wrote the manuscript.

The content of Chapter 8 is available as preprint — *Depth-optimal quantum circuit placement for arbitrary topologies. Debjyoti Bhattacharjee and Anupam Chattopadhyay. arXiv preprint arXiv:1703.08540 (2017).*

- A. Chattopadhyay introduced the problem and the existing works in the domain.
- I designed and implemented the solutions for the investigated problem variants.
- The manuscript was written by both the authors.

.....

**Debjyoti Bhattacharjee**
Hardware and Embedded Systems Laboratory (HESL)
School of Computer Science and Engineering,
Nanyang Technological University, Singapore.

Date : *12 December, 2018*

# Acknowledgments

I would like to show my highest gratitude to my advisor, *Dr.-Ing. Anupam Chattopadhyay*, Nanyang Technological University, Singapore, for his guidance and continuous encouragement. He has taught me how to be a humble researcher and a vigorous learner, motivated me with great insights and always supported my pursuit of innovative ideas.

My deepest thanks to all the members of Thesis Adivisory Committee, for their valuable suggestions and discussions which added an important dimension to my research work. I would specially like to thank *Dr. Arvind Easwaran*, School of Computer Science and Engineering, Nanyang Technological University, Singapore, for his active support and guidance.

Furthermore, I am very grateful to *Prof. Giovanni De Micheli* and *Dr. Shahar Kvatinsky* for giving me the opportunity to be a visiting student at EPFL, Switzerland and Technion Institute of Technology, Israel respectively.

I would also like to thank *Dr. Rajeshwari Devadoss*, *Dr. Mathias Soeken*, *Dr. Vikram Pudi*, *Dr. Luca Amaru* and *Dr. Farhad Merchant* for the great research collaborations we had.

Tremendous thanks to all my fellow researchers and labmates for the stimulating discussions and great moments that I shared at Hardware and Embedded Systems Laboratory (HESL). I want to specially thank *Jeremiah Chua* for his extensive help and support during my PhD studies.

Finally, I am very much thankful to my parents for their everlasting support during my PhD studies as well as in my life.

Last but not the least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

**Debjyoti Bhattacharjee**
Hardware and Embedded Systems Laboratory (HESL)
School of Computer Science and Engineering,
Nanyang Technological Univeristy, Singapore.
*12 December, 2018*

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| 1S1R | A Select device in series with a memristive device |
| ACTMP | Area-constrained technology mapping problem |
| AIG | And Inverter Graph |
| BDD | Binary Decision Diagram |
| BRS | Bipolar Resistive Switches |
| CMOS | Complementary Metal Oxide Semiconductor |
| CRS | Complementary Resistive Switches |
| DAG | Directed Acyclic Graph |
| DCM | Data and Computation Memory |
| EDA | Electronic Design Automation |
| ESOP | Exclusive Sum-of-Product |
| HRS | High Resistive State |
| ILP | Integer Linear Programming |
| IoT | Internet of Things |
| LiM | Logic-in-Memory |
| LRS | Low Resistive State |
| LUT | Look Up Table |
| MDD | Multi-valued Decision Diagram |
| MIG | Majority Inverter Graph |
| NMOS | N-type Metal Oxide Semiconductor |
| NN | Nearest Neighbour |
| PC | Program Counter |
| PMOS | P-type Metal Oxide Semiconductor |
| ReRAM | Redox-based Random Access Memory |
| SDG | State Dependency Graph |
| SHA3 | Secure Hash Algorithm 3 |
| SRAM | Static Random Access Memory |
| STG | Single Target Gate |
| VLIW | Very Long Instruction Word |

# Abstract

The continued scaling of horizontal and vertical physical features of silicon based complementary metal oxide semiconductor (CMOS) transistors is termed as "More Moore". There has been tremendous effort to follow the Moore's law but it is currently approaching atomistic and quantum mechanical physics boundaries. This has led to active research in other emerging technologies such as memristive devices, carbon nanotube field-effect transistors, quantum computing, etc. Recent advances of memristive devices allow high endurance, non-volatile storage and low leakage power while at the same time allowing logic operations. Each device is a two-terminal resistor, with the internal resistive state of the device switched by the input voltage applied at the terminals. Also, these devices allow realization of new logic functionality.

The primary goal of the thesis is to analyze the problem of design automation for logic-in-memory computing. Specifically, we look at the task of technology mapping, that realizes a Boolean function by means of operations on the target technology platform. We have developed a polynomial-time delay-optimal mapping algorithm for technology mapping using memristors. To enable leveraging inherent parallelism of the memory while allowing logic operations using memristive crossbar array, we have developed a general purpose programmable architecture. For the proposed crossbar-based architecture, we have developed technology mapping algorithms that permit area-constraints as well as delay-constraints, with the input Boolean function represented as a multi-level logic network. We investigated multi-state memristors, that allow storage of multi-bit input on a single device. By means of crossbar fabrication and detailed experiments, we have demonstrated the realization of finite state multi-valued logic (MVL) using the multi-state memristors. Further, MVL operations have been used to implement fuzzy logic operations, that allow modelling of imprecise and incompletely specified systems.

Logic-in-memory operations are inherently sequential, and the placement of the operands affect the overall quality of mapping. This prevents direct application of logic optimization techniques to improve quality of technology mapping. By considering the logic structures that can be efficiently mapped using crossbar arrays, we developed technology-specific optimization techniques for the logic representation. We have also developed algorithms for synthesizing multi-valued primitive functions using multi-valued logic primitives, that can be realized using multi-state memristors.

Quantum computing systems offer the conceptual appeal of exponential growth of computational power with the number of quantum bits (qubits), that allows exponential reduction in time spent on executing combinatorial algorithms compared to classical computers. However, the number of qubits available is limited. This leads to the need for developing robust synthesis techniques that allow control of qubit count. We have created a synthesis algorithm to map a logic network to single target gates, using reversible pebble game to reduce number of qubits. Further, we have developed optimisation techniques to lower the number of single target gates in the synthesized circuit.

Physical realization of quantum programs involves coupling two distant qubits with some

communication overhead. Depending on the qubit topologies, the interacting qubits should be nearest neighbours (NN) to keep the circuit latency and error rate low when a sequence of operations is executed. This presents the need for an automated mapping of quantum circuits to qubit topologies such that the interacting qubits in the quantum circuit are nearest neighbours. Despite the wide variety of topologies available, the current literature focuses mostly on 1D chain qubit and 2D lattice structures. To address this gap, we have developed an ILP-based algorithm to realize depth-optimal nearest neighbour quantum circuits for arbitrary topologies. Our algorithm is also applicable, naturally, to simpler structures such as Linear Neighbours. We have benchmarked the algorithm for diverse topologies and quantum circuits.

**Keywords**: *Electronic Design Automation; Logic Synthesis; Technology mapping; Emerging Technologies; In memory computing; Redox Random Access Memory; Multi-valued logic; Fuzzy Logic; Quantum Circuits;*

# 1 Introduction

Electronic Design Automation (EDA) tools enable seamless transformation of *software* to *silicon*. In other words, EDA tools use a series of hierarchical steps to transform a high-level functionality specification into a completed integrated circuit layout ready for fabrication. Before EDA took center-stage, integrated circuits were designed manually. One of the early automation came in the form of geometric software to copy digital recordings to mechanically-drawn components in Graphic Data System (GDS) format, which is still an industry standard [1]. In 1980, Mead et al. advocated complex chip design by enabling access to digital simulation and verification tools [2]. Since then, EDA tools provide robust support for CMOS-based hardware design while meeting plethora of design constraints such as area of physical design, timing constraints, technology-specific constraints, etc. [3–7].

Along side advances in EDA tools, semiconductor fabrication has taken substantial strides from 10 $\mu m$ to the current day 7 $nm$ technology nodes. However, the scaling down of MOSFET has introduced severe limitations due to the decrease in gate oxide thickness and voltage level [8, 9]. The ratio between the operating voltage and thermal voltage decreases with the decrease in the operating voltage as the thermal voltage ($kT/q$) is constant at a given temperature [10]. As a result of this, source to drain leakage current increases because of the thermal diffusion of electrons. Furthermore, due to the decrease in the gate oxide thickness, there has been a sharp increase in the gate leakage current. To

replace CMOS with a new device or technology that would allow long term scaling is a hot research topic [9]. Although CMOS can continue to scale in size for some more time, our ability to achieve full benefits of scaling has become limited, as we are forced to trade-off between transistor density and speed to mitigate the power density increase. Fortunately, the emergence of these problems coincides with other technological advances. First, the lack of efficiency at the technology level is masked by the architectural techniques of building multi-core systems or approximate systems. Second, a plethora of new technologies, which may eventually complement and replace CMOS, is there. Some of these technologies are at prototype-level, some are provided as a complementary to the standard CMOS techniques, even with the same manufacturing steps. A few technologies are already in mainstream with storage devices.

One of the most promising beyond-CMOS technology capable of functioning as a non-volatile memory with computation capabilities is Redox based Resistive RAM, or more affectionately known as *ReRAM*. Passive crossbar configuration of ReRAM enables ultra-dense $4F^2$ integration [11, 12]. Recently, $TaO_x$ based ReRAMs have drawn significant attention due to excellent performance in term of high endurance ($> 10^{12}$) [13], long retention (10 years) [14], multi-level switching capability (3-bit) [15] and fast read/write speed of below $200ps$ [16]. Besides memory applications, ReRAM based passive crossbar arrays offer implementation of memory-intensive computing paradigms, i.e., the logic operations are directly processed in the memory. This merges the boundaries between memory and logic units and eases the von-Neumann bottleneck for computation [17]. Furthermore, memristive crossbar arrays can enable native implementation of a variety of algorithms, such as multi-parallel search algorithms for pattern recognition tasks, neuromorphic computing, ternary arithmetic [18–20].

Another beyond-CMOS technology that has the capability of providing a disruptive leap forward in computing performance is quantum computing. A quantum computer is a machine designed to harness quantum mechanics to perform computations. A variety of technologies are being pursued for realization of the quantum computing systems, such as trapped ion quantum computing, superconducting quantum computing, etc [21, 22]. The excitement about quantum computing is supported by the

fact that quantum computers are capable of achieving speedup compared to classical computers [23]. One such representative case has been demonstrated by Peter Shor in his landmark paper [24], that demonstrated integer factorization problem to be polynomial time solvable on a quantum computer but is intractable to solve on classical computers. Thereafter, a wide variety of quantum algorithms have been developed [25]. A number of automation frameworks for quantum computing are under development from both academia as well as industry [26–28].

The significant challenge in bringing such beyond-CMOS technologies to practical usage and adoption is the availability of robust design automation flows. In this thesis, we focus on the technologies that offer enhanced native functionality compared to standard Metal Oxide Semiconductor transistors. For example, a variety of emerging technologies such as ReRAM, quantum dot cellular automata, etc have a majority/minority gate as the computation primitive which is more expressive compared to NAND or NOR gate realized in CMOS. The design automation flows must be fully capable to harnessing the expressive power of the these emerging technologies, that would be otherwise not considered by traditional design automation flows. Also, contemporary design automation flows that support silicon CMOS technology, cannot rise to the occasion of supporting the design challenges using emerging technologies, since the technology constraints are varied and differ significantly from CMOS technology constraints. In this thesis, we have shown architectures and design automation flows for emerging technologies, that would enable adoption of emerging technologies.

## 1.1 Contributions

This thesis is focused around design of efficient hardware to harness the capabilities of emerging technology, supported by effective design automation tools.

We reviewed the concepts of stateful logic realization using ReRAM crossbar arrays and proposed an effective ReRAM-crossbar array based Logic-in-Memory (LiM) architecture with a lightweight controller — ReVAMP. This is a VLIW architecture for general purpose computing harness bit-

level parallelism inherent to the crossbar arrays. We developed an end-to-end automation flow to support this architecture, starting from a structural representation of input Boolean function that is mapped to the instructions supported by the architecture. As part of the automation flow, we formally defined the problem of technology mapping for LiM. We undertook a detailed study of technology mapping using ReRAM devices and proposed a novel delay-optimal algorithm for the same. Furthermore, we presented an optimal solution based on Integer Linear Programming (ILP) for solving the area-constrained technology mapping, along with a scalable heuristic. Thereafter, we investigated the problem of technology mapping for LiM using ReRAM crossbar arrays and proposed a multi-phase approach to solve delay-constrained and area-constrained variants of the problem. Finally, we presented a technology-aware logic synthesis flow to improve the overall quality of arbitrary logic realization using ReRAM crossbar arrays.

For realizing Multi-Valued Logic (MVL) natively using ReRAM devices, we fabricated multi-state ReRAM crossbar array and investigated the switching characteristics of the device. We then mapped the native MVL logic primitives to the inherent functionality of the multi-state ReRAM device by means of device operations. Thereafter, we demonstrated computation of fuzzy logic using MVL primitives on the fabricated multi-state ReRAM array. We undertook integration of algorithms in existing multi-valued synthesis tool to permit mapping from MVL function specification to MVL primitives, that can be natively realized by the multi-state ReRAM array.

In the context of design automation for quantum computing, we address two significant problems. Firstly, we address the challenge of constraint on the number of qubits (a qubit for a quantum computer has a similar meaning as that of a bit w.r.t a classical computer) available for mapping an algorithm. We proposed a heuristic algorithm based on reversible pebble games to constrain number of qubits in the synthesis phase of the design automation flow and integrated it in an open source automation framework. Secondly, we enable transformation of quantum circuits to nearest-neighbour complaint circuits for a given topology constraint. This is the first work that explores a generalized transformation technique, that is applicable to any arbitrary topology constraint.

## 1.2 Thesis Organization

The contributions of this work are on three distinct fronts: programmable architecture and automation for LiM using ReRAM (Part I), facilitating MVL using multi-state ReRAM array (Part II) and automation techniques for quantum computing (Part III). A summary of the contents of the following chapters in the thesis are presented below.

**Chapter** 2: We initiate a detailed review of existing logic-in-memory implementations using ReRAM devices. We describe a ReRAM device model that realizes three-input Boolean majority function with an input inverted. Then, we present an overview of logic operations on devices arranged in crossbar-array configuration.

**Chapter** 3: We introduce a novel general purpose logic-in-memory computing architecture, ReVAMP. We explain in detail the device-accurate simulation setup for the architecture and demonstrate the functioning of the architecture using implementation of Secure Hash Algorithm 3 as a representative case study.

**Chapter** 4: In this chapter, we introduce the notations of logic representation, logic network and design automation, that are used extensively in the following chapters. We formally define the problem of technology mapping for logic realization using ReRAM device and present an algorithm for delay-optimal technology mapping. We also address the problem of area-constrained technology mapping optimally using an ILP formulation and also introduce a scalable heuristic for the same.

**Chapter** 5: We investigate the problem of crossbar-constrained technology mapping, and present multi-phase automation flows to address the problem. The solution approach exploits the parallelism offered by the crossbar arrays, whenever feasible and at the same time, tries to minimize the number of devices required for computation.

**Chapter** 6: With the aim to aid the technology mapping flow, we analyse the possibility of optimizing the logic representation structure. We use the technology specific constraints to optimize the logic synthesis

flow, that takes a Majority Inverter Graph as an input. Our results show that such optimizations in the synthesis phase can lead to considerable reduction in delay of mapping and lowers the number of devices required for mapping.

**Chapter** 7: We begin by looking back at the history and fundamentals of Łukasiewicz logic (a $N$-valued or simply a multi-valued logic system), details of multi-state $TaO_x$ device fabrication and characterization. We proceed to compute the Łukasiewicz primitives natively using the fabricated devices and demonstrate a proof-of-concept implementation of a fuzzy logic controller. We analyze in depth the advantages of native MVL implementation over a binary-based implementation using CMOS devices. Thereafter, we review existing MVL synthesis approach, develop algorithms to address the gaps present in the existing synthesis tools and evaluate the proposed synthesis flow on a set of standard MVL benchmarks.

**Chapter** 8: In this chapter, we make the readers familiar with the idea of quantum computing starting from quantum gates to quantum circuits. We re-examine the state-of-the-art synthesis flow for quantum circuits and observe that qubit constraints are not currently supported. We address this drawback by developing an algorithm based on *reversible pebble games* and propose optimizations to reduce the overall delay in presence of qubit constraints. Our algorithms were tested on large arithmetic benchmarks.

**Chapter** 9: We address a practical challenge of nearest-neighbour transformation of quantum circuits for arbitrary topologies. We present an optimal solution based on ILP that guarantees minimum logical depth of the transformed circuit while meeting the nearest neighbour arrangement for interacting qubits and a given interaction topology. We tested our algorithm on a diverse variety of topologies with prominent quantum circuit benchmarks.

**Chapter** 10: The chapter concludes this thesis with a summary of the research accomplishments. Possible directions for future research are discussed.

# Part I:

# Programmable Architecture And Automation For LiM Using ReRAM

# 2 Background

In 1972, Leon Chua reasoned the existence of memristor $M$, as a two terminal fundamental circuit element after resistor $R$, inductor $L$ and capacitor $C$ [29]. Memristors are characterized by the following equation

$$\phi = f(q) \text{ or } q = g(\phi) \tag{2.1}$$

where $q$ and $\phi$ are the integrals of $i$ and $v$ respectively. The tetrahedron that lines the four electrical variables, namely $v$, $i$, $\phi$ and $q$, was identified by Leon and is shown in Figure 2.1. Active memristive devices such as ionic systems and discharge tubes were presented by Chua and Kang [30]. Much later in 2008, the topic of memristors and memristive devices garnered widespread attention, when Hewlett Packard demonstrated memristors using $TiO_2$ resistive switches [31]. This started the research in the direction of using memristors as a replacement of conventional DRAM, brain-inspired systems as well as nano-electronic systems and architectures.

Resistive switching memories allow the internal device resistance to be altered between two (or more) resisitve states — Low Resistance State (LRS) is obtained after a set programming while High Resistance State (HRS) is obtained after reset programming. The classification of resistive switching memories based on switching mechanisms is presented in Figure 2.2. Two terminal memristive devices

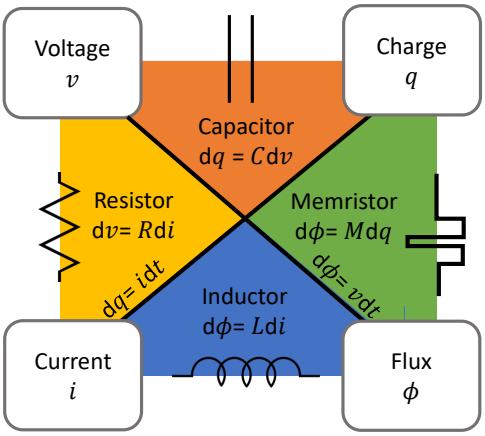



**Figure 2.1:** Constitutive relationship of Memristor $M$

exhibit a distinctive pinched hysteresis loop, regardless of the device material and physical operating mechanisms and can act as non-volatile memory devices [32].

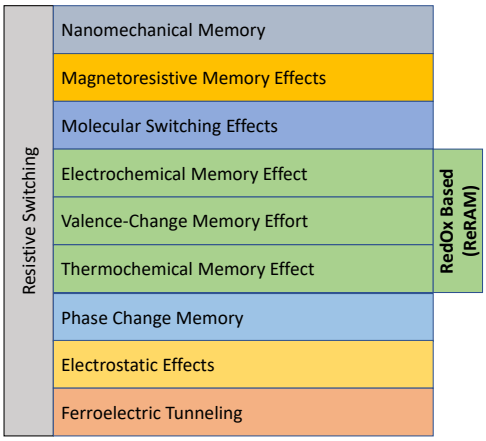



**Figure 2.2:** Classification of resistive switching memories based on the switching mechanisms [33].

One of the most promising emerging technology for logic and memory operations is Redox-based Random Access Memory (ReRAM) [34]. A ReRAM device is a two-terminal passive variable resistor that operates in a high resistance state (HRS) or OFF-state and a low resistance state (LRS) or ON-state. Initially, the device consists of pristine metal oxide is sandwidched between two metal electrodes. By application of a high voltage, the high resistive state of the as-prepared device changes to a low resistive state (for example, from several mega-ohms to the kilo-ohms range). This process is called

forming. With the increase in applied voltage, the electrical field in the metal oxide increases, leading to the growth of a conductive filament through the material from one electrode to the opposite electrode. The formed filament creates a switching capable non-volatile memory cell, as shown in Figure 2.3b. Now, the device can be switched between Low Resistive State (LRS) (shown in Figure 2.3c) and High Resisitive State (HRS) (shown in Figure 2.3d), by applying voltages in opposite directions for bipolar operations. A summary of the technological overview and the milestones achieved by the progress of ReRAM devices can be found in [35–38].

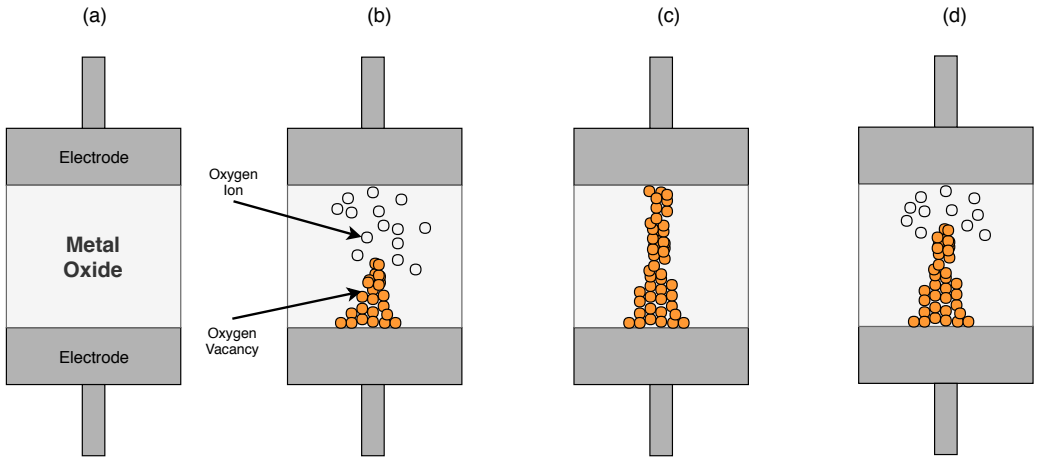



**Figure 2.3:** ReRAM device basic operation. (a) Device in pristine state. (b) Device after *forming*. (c) Device after SET operation in Low Resistive State. (d) Device after RESET operation in High Resistive State.

Passive crossbar configuration of memristive devices is ideal for memory applications, as it can offer a device area down to $4F^2$, where $F$ is the minimum feature size [11]. Figure 2.4a shows a 0T1R or simple a 1R configuration. This configuration has been realized by multiple physical prototypes [31, 39, 40]. However, the size of the crossbars are severely limited due to presence of parasitic current [41].

To enable large crossbar, devices that prevent parasitic current are required [42]. One conventional approach is to place each ReRAM device in series with a device-selection transistor, as shown in Figure 2.4b. The presence of the transistor aids in isolation of the selected device from other unselected devices and eliminates the problem of sneak path currents. Furthermore, read-disturbances are prevented from other half-selected cells, which improves reliability. However, such an organization

can lead to much larger cell size which reduces memory density and increases per bit fabrication cost. Various approaches have been suggested for low power and high speed operations using 1T1R crossbar arrays [43–46]. A variety of selector devices (both unipolar and bipolar) have been developed to allow a high density of integration, in a 1S1R configuration as shown in Figure 2.4c. Details of development of various selector devices, characterization of materials, processes and solutions have been shown in [47]. Another option could be to use complementary resistive switch (CRS), consisting of two anti-serially connected cells to prevent sneak path currents [42].

As presented above, ReRAM relies on the formation and the rupture of a conductive filament that leads to different resistive states, the modeling of ReRAM device and the memory array design (including peripheral circuitry design) is fundamentally different from that for STT-RAM and PCM. Modelling ReRAMs as memristive systems was suggested by Strukov et al. [31]. Device modelling of memristive devices is difficult due to complex physical mechanisms [48–50]. In the next section, we review the model of a 1S1R cell, that is used in the rest of the thesis.

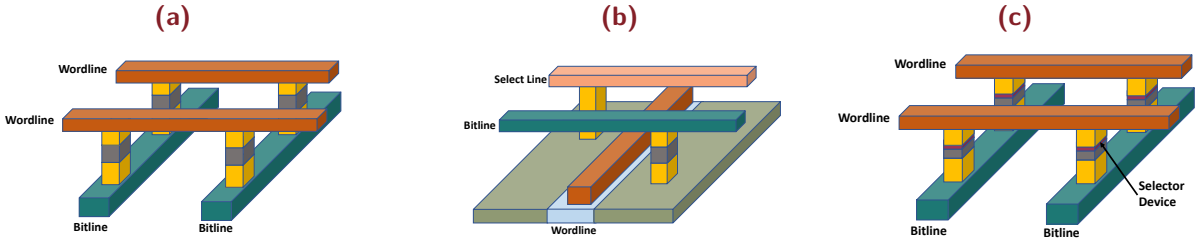



**Figure 2.4:** Various ReRAM cell configurations. (a) Crossbar array of 0T1R or simply 1R cells. (b) A 1T1R configuration. (c) A 1S1R configuration, with the selector device stacked above the device.

## 2.1 ReRAM Device Model

In this work, we consider the ReRAM device model proposed in [51] fitted to a $Pt/(11nm)TaO_x/Ta$ cell. The used selector device is the $Pt/TaO_x/TiO_2/TaO_x/Pt$ crested barrier device proposed in [52, 53]. The used ReRAM model considers a filamentary region in which the switching takes place by a redistribution of ionic defects, i.e., oxygen vacancies. The filament is modeled by three lumped circuit

elements: a Schottky-type diode representing the current flow through the $Pt/TaO_x$ interface, a disc resistance describing the region close to the Schottky-type interface and a resistance, which comprises the plug resistance describing the remaining part of the filament and the resistance of the electrodes. The state variable of the resistive switching model is the oxygen vacancy concentration $N$ close to the active electrode interface, which modulates the disc resistance and the electron transport through the Schottky-type diode. The change of the concentration with time is described by

$$\frac{dN}{dt} = \frac{-I_{ion}}{ez_{VO}Al_{disc}} \tag{2.2}$$

and depends on the ionic current $I_{ion}$, which is calculated using

$$I_{ion} = 2z_{VO}ec_{VO}afAexp(-\Delta\frac{W_A}{k_BT})\sinh(\frac{z_{VO}eaV_{disc}}{2k_BTl_{disc}}). \tag{2.3}$$

As a filament is considered, the dissipated power leads to a local temperature, which is calculated using

$$T = T_0 + R_{th}V_{disc}I_{disc} \tag{2.4}$$

In equations (2.2), (2.3) and (2.4), $e$ is the electron charge, $z_{VO}$ is the charge number of the oxygen vacancies, $A$ is the filament area, $l_{disc}$ is the length of the disc region, $c_{VO}$ is the mean oxygen vacancy concentration, $a$ is the hopping distance of the oxygen vacancies, $f$ is the attempt frequency, $\Delta W_A$ is the activation energy of ion migration, $k_B$ is the Boltzmann constant, $V_{disc}$ is the voltage drop over the disc resistance, $T_0$ is the ambient temperature and $R_{th}$ is the equivalent thermal resistance of the disc region. The disc concentration $N$ influences the disc resistance according to

$$R_{disc} = \frac{l_{disc}}{Ne\mu_nA} \tag{2.5}$$

where $\mu_n$ is the electron mobility. The current-voltage relation for the Schottky-type diode depends on the polarity of the voltage drop $V_{Schottky}$ over the diode. For all voltages, the current caused by

thermionic emission $I_{TE,Schottky}$ is considered according to

$$I_{TE,Schottky} = AA^*T^2 exp(\frac{-e\phi B}{k_B T})(exp(\frac{eV_{Schottky}}{k_B T}) - 1) \tag{2.6}$$

For negative voltages (reverse direction), an additional tunneling current is calculated using Fowler-Nordheim tunneling by

$$I_{FE,Schottky} = -Ae^2 \frac{V_{Schottky}^2}{8\pi h \phi_B w_{eff}^2} exp(-8\pi w_{eff} \frac{2m_e e(\phi_B)^3)^{1/2}}{3h|V_{Schottky}|}) \tag{2.7}$$

In (2.6) and (2.7), $A^*$ is the Richardson constant, $\phi_B$ is the effective Schottky barrier height, $m_e$ is the mass of an electron and $h$ is Planck's constant. The effective width $w_{eff}$ of the Schottky diode depletion layer is a function of $V_{Schottky}$, the maximum defect concentration $N_{max}$ and the permittivity $\varepsilon_s$. It is calculated using

$$w_{eff} = \phi_B(2\varepsilon_s \frac{\phi_B - V_{Schottky}}{eN_{max}})^{1/2}/(\phi_B - \phi_n - V_{Schottky}) \tag{2.8}$$

The effective barrier height depends on the electric field at the interface, which leads to a lowering of the barrier according to

$$\phi_B = \phi_{Bn0} - (\frac{e^3 N(\phi_{Bn0} - \phi_n - V_{Schottky}}{8\pi^2(\varepsilon_{s\phi B})^3})^{1/4} \tag{2.9}$$

Here, $\phi_{Bn0}$ denotes the nominal Schottky barrier height, $\phi_n$ is the distance between Fermi level and conduction band and $\phi_{s\phi B}$ is the relevant permittivity for the electron screening. Equation (x+7) applies as long as $\phi_{Bn0} - \phi_n - V_{Schottky} > 0$. The selector is modeled by a nonlinear resistor $RNL$ as a function of the voltage drop $VNL$ over the nonlinear resistor according to

$$R_{NL} = \frac{|V_{NL}|^{1-\alpha} V_0^\alpha}{J_0 A_{selector}} \tag{2.10}$$

**Table 2.1:** ReRAM device simulation parameters

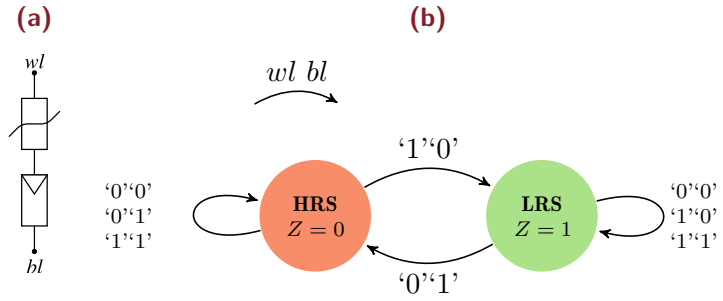
| | | | | | |
|---|---|---|---|---|---|
| $l_{disc}$ | $= 4nm$ | $\mu n$ | $= 1.6 \times 10^{-5} m^2 (Vs)^{-1}$ | $R_{ser}$ | $= 18.2k\Omega$ |
| $z_{VO}$ | $= 2$ | $A*$ | $= 11 \times 106 A(mK)^{-2}$ | $V_0$ | $= 1.6V$ |
| $A$ | $= 8 \times 10^{-17} m^2$ | $\phi_{Bn0}$ | $= 0.5V$ | $J_0$ | $= 1011 Am^{-2}$ |
| $f$ | $= 0.5 \times 1012Hz$ | $\phi_n$ | $= 0.1V$ | $\alpha$ | $= 15$ |
| $\Delta W_A$ | $= 0.93eV$ | $\varepsilon_{s\phi B}$ | $= 9.23 \times \varepsilon_0$ | $A_{selector}$ | $= 121nm^2$ |
| $\varepsilon_s$ | $= 26 \times \varepsilon_0$ | $R_0$ | $= 69k\Omega$ | $R_{\|\|}$ | $= 16.53G\Omega$ |
| $R_{th}$ | $= 50 \times 106 KW^{-1}$ | $c_{Vo}$ | $= 1026$ | $T_0$ | $= 300K$ |
| $a$ | $= 0.5nm$ | | | | |

Here, $V_0$ is a reference voltage, $J_0$ a current density prefactor, and $A_{selector}$ the area of nonlinear region. The nonlinear resistor is in parallel to a resistor $R_{||}$. A third resistor $R_{ser}$ is connected in series to the parallel resistances $R_{NL}$ and $R_{||}$. The parameters describing the selector have been determined by fitting the model to the experimental I-V characteristics of Figure 4 in [52]. The simulation parameters used in this study are listed in Table 2.1.

In this device model, the SET transition occurs at negative voltages (-4.8V) and the RESET transition at positive voltages (+4.8V). In the simulated circuit, the device is implemented in such a way that a positive (negative) voltage drop from *wl* to *bl* leads to a SET (RESET) process. Both devices were implemented in VerilogA. The read is performed by applying $V/2$ to the devices, which should be read out. If a high current is detected, the state is interpreted as 1. If a low current is detected, the state is interpreted as 0. The pulse width for operations is set to $50ns$.

## 2.2 Logic-in-Memory Approaches Using ReRAM

Novel memory intensive architectures can be enabled by means of logic-in-memory operations using memristors. This gives rise to various non von-Neumann architectures, that do not suffer from the *memory wall* [54]. By integrating CMOS circuits with memristors, logic operations have been realized [55]. Also, memristors have been used in FPGA-like architectures as reconfigurable switches, to reduce leakage power. This approach utilizes ReRAM devices as switchable interconnects where elementary




(c)

| wl | bl | $Z_n$ |
|---|---|---|
| '0' | '0' | $Z$ |
| '0' | '1' | 0 |
| '1' | '0' | 1 |
| '1' | '1' | $Z$ |

**Figure 2.5:** Logic operation of 1S1R ReRAM device. (a) Schematic of a 1S1R device with two input terminals — wordline (wl) and bitline (bl). (b) Logic operation represented as a Finite State Machine. The edge labels indicate the wordline (wl) and bitline (bl) inputs. (c) Truth Table representation of the logic operation, with the next state defined as follows: $Z_n = M_3(Z, wl, \neg bl)$.

CMOS cells are connected using discontinuous lines via ReRAM cells [56]. Crossbar arrays have been used for realization of Look-up-tables (LUT) in Field programmable gate arrays (FPGA) [57–59] . Hybrid CMOS and memristor-based standard cell libraries for circuit design have been proposed [60]. Two level logic representation has been realized directly on ReRAM crossbar arrays [61].

Borghetti et. al proposed to use ReRAM cells as conditionally switchable sequential logic devices for realizing material implication— enabling direct logic-in-memory operations [62]. Resistive states of the device is interpreted as binary logic operation — HRS indicates logic zero while LRS indicates logic one. In [17], a variety of Boolean functions have been realized by multi-cycle operations on single bipolar resistive switches (BRS) as well as complementary resistive switches (CRS). Parallel logic-in-memory realization of multiple NOR gates have been shown using crossbar array [63]. In this thesis, we focus ReRAM devices that realize Boolean Majority-three logic with a single input inverted, as described in the following sections.

### 2.2.1 Logic Operation Using 1S1R device

Figure 2.5a shows the schematics for a single 1S1R device. Each ReRAM is a two terminal device — top terminal (wordline $wl$) and bottom (bitline $bl$), where input voltages are applied. Each device acts a three input device $wl$, $bl$ and $Z$ where is the previous resistive device state. Since the device state is non-volatile, the updated state $Z_n$ is the output of the logic operation which can be used as input in

the upcoming operations.

We introduce the notations used for specifying input and state of ReRAM devices. Input '1' and '0' represent Boolean logic one and zero respectively. Input 0 denotes Ground (0 V). For the internal state of a device $Z$, 1 and 0 represent Boolean logic one and zero respectively.

The logic operation of 1S1R device (introduced in Section 2.1) can be represented as Finite State Machine (FSM) as shown in Figure 2.5b. HRS and LRS refer to High Resistance State and Low Resistance State respectively. The next state $S_n$ of the device can be expressed as follows.

$$\begin{aligned} S_n &= (wl.\neg bl).\neg S + (wl + \neg bl).S & &= wl.S + \neg bl.S + wl.\neg bl).\neg S \\ &= wl.S + \neg bl(S + wl.\neg S) & &= wl.S + \neg bl(S + wl) \\ &= wl.S + \neg bl.S + \neg bl.wl & &= M_3(S, wl, \neg bl) \end{aligned} \tag{2.11}$$

The state update function corresponds directly to the Boolean Majority function with one inverted input, $M_3(S, wl, \neg bl)$. This function forms a *functionally complete set* and hence can be used for realization of any Boolean function.

### 2.2.2 Logic-in-Memory Operations Using ReRAM Crossbar Array

A ReRAM crossbar memory consists of multiple ReRAM devices, arranged in the form of a crossbar array [64]. Multiple devices share wordlines and bitlines. Figure 2.6 shows a ReRAM crossbar array with 6 devices arranged in $2 \times 3$ configuration i.e. 2 wordlines and 3 bitlines. The internal state of device $D_{ij}$ at wordline $i$ and bitline $j$ is referred as $S_{ij}$. The devices $D_{00}, D_{01}$ and $D_{02}$ share wordline 0 whereas the devices $D_{10}, D_{11}$ and $D_{12}$ share wordline 1. Similarly, the devices $D_{00}$ and $D_{10}$ share bitline 0 and so on. Like conventional RAM arrays, ReRAM memories are accessed as words. It should be noted that all the devices in a word share a common wordline. For example, word 0 has devices $D_{00}, D_{01}$ and $D_{02}$.

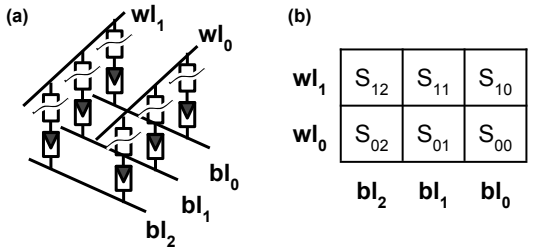



**Figure 2.6:** A $2 \times 3$ ReRAM crossbar array (a) Six 1S1R devices arranged as crossbar. (b) The crossbar represented as a schematic. $S_{ij}$ represents internal state of device at wordline $i$ and bitline $j$. $wl_i$ represents the $i^{th}$ wordline input while $bl_j$ represents the $j^{th}$ bitline input.

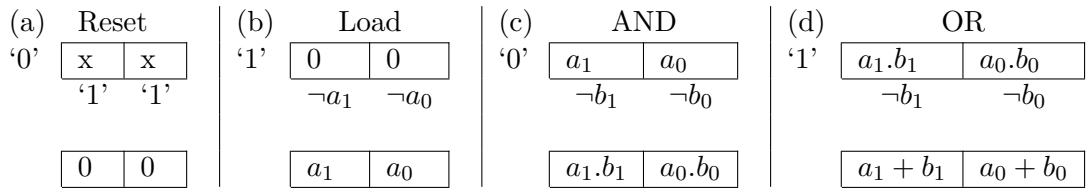



**Figure 2.7:** Realization of basic Boolean functions using ReRAM crossbar arrays.

Figure 2.7 presents realization of basic Boolean operation using $1 \times 2$ crossbar array, as a representative example. Figure 2.7a shows how the array can be reset to 0, irrespective of its contents. '0' is applied as wordline input with all the bitline inputs set to '1'. This resets the internal state of the devices to 0, since $M_3(x, 0, \neg 1) = 0$. To load a 2-bit input $[a_1\ a_0]$ into the array initialized to 0, the negated input $[\neg a_1\ \neg a_0]$ is applied to the bitlines with '1' as the wordline input, as shown in Figure 2.7b, since $M_3(0, 1, \neg(\neg a)) = a$. To compute AND of two inputs, the first input is loaded into the array and then the negated second input is applied to the bitlines with '0' as wordline input, because $M_3(0, a, \neg(\neg b)) = a.b$. Similarly, OR of two inputs can be computed, by changing the wordline input to '1', since $M_3(0, a, \neg(\neg b)) = a + b$.

We illustrate the realization of two-bit XOR function for operands $p_1p_0$ and $q_1q_0$, as a representative example of logic-in-memory realization of arbitrary Boolean function. To compute the XOR, we use the following equation :-

$$p_i \oplus q_i = p_i.\overline{q_i} + \overline{p_i}.q_i = p_i.\overline{q_i} + \overline{p_i + \overline{q_i}} \tag{2.12}$$

The computation is performed on a $3 \times 2$ crossbar. At the start of computation, all the devices are in 0 state, i.e., HRS. Figure 2.8 shows the sequence of operations performed to realize the 2-bit XOR function and the steps are described below.

- **Step 1:** Inputs $p_0$ and $p_1$ are loaded to wordline 0, since $M_3(0, 1, \overline{p_i}) = \overline{p_i}$.
- **Step 2:** Wordline 0 is read out. The read out values $\overline{p_0}$ and $\overline{p_1}$ are used in the following steps.
- **Step 3-4:** The read out value is loaded to wordline 1 and 2 by applying the values via the bitlines as $M_3(0, 1, \overline{\overline{p_i}}) = p_i$.
- **Step 5:** Input $q_0$ and $q_1$ are ANDed with the values in wordline 2 in inverted form by using 0 as wordline input since $M_3(p_i, 0, \overline{q_i}) = p_i.\overline{q_i}$.
- **Step 6:** Input $q_0$ and $q_1$ are ORed with the values in wordline 1 in inverted form by using 1 as wordline input since $M_3(p_i, 1, \overline{q_i}) = p_i + \overline{q_i}$.
- **Step 7:** The ORed values available in wordline 1 are read out.
- **Step 8:** The read out values are ORed with the contents of wordline 2 to complete the XOR operations, as $p_i.\overline{q_i} + \overline{p_i + \overline{q_i}}$.

This concludes the description of logic-in-memory operations using ReRAM crossbar array. We introduce the notations related to logic representation in the following sections.

## 2.3 Discussion

In this chapter, we presented the background of logic-in-memory approaches using memristors. We also reviewed in detail a model of a selector device in series with a ReRAM device (1S1R), that can be used for realizing a functionally complete Boolean operator set. We illustrated the computation of various Boolean functions using a crossbar array configuration of the 1S1R devices.

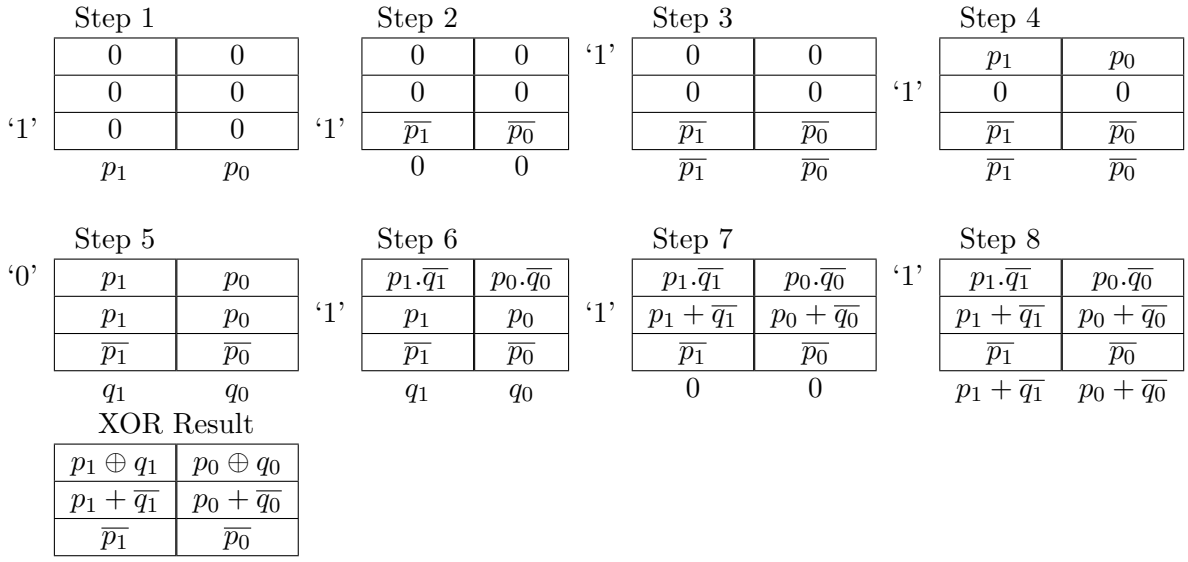



**Figure 2.8:** XOR computation of two-bit vectors $p_1p_0$ and $q_1q_0$. Input Boolean constants 1 and 0 are represented as ‘1’ (+2.4V) and ‘0’ (-2.4V) respectively. A word in the crossbar is read out by applying +2.4 to the wordline and grounding the bitlines, which is represented by ‘1’ and 0 as the wordline and bitline input respectively.

# 3 ReRAM-based VLIW Architecture For In-Memory Computing

With diverse types of emerging devices offering simultaneous capability of storage and logic operations, researchers have proposed novel platforms with promising gains in energy-efficiency. A detailed break-down of the applications using ReRAM devices can be found in [65]. Broadly, the platforms can be classified in two domains — application specific and general purpose. ReRAM devices have been used for efficient realization of various types of neural networks [66–68]. Also, efficient neuromorphic circuits have been designed using ReRAM devices [69–72], that use analog operations on crossbar arrays. Analog matrix operations have also been realized using ReRAM crossbar arrays [73, 74]. Furthermore, it has been demonstrated that traditional applications such as image processing can also benefit from logic-in-memory computations using ReRAM [75]. Novel application specific processor for automata processing have also been realized using crossbar arrays [74].

The idea of general purpose computing platforms stem from the idea that several in-memory computing logic devices support a universal set of Boolean logic operation and therefore, can be used for realizing arbitrary functionality. This can be used for design of low power systems for upcoming

applications, such as Internet-of-things [76, 77]. Xie et. al proposed a crossbar based architecture to realize Boolean functions, using a Truth table representation [78]. Starting from Sum-of-Products representation, Snider proposed computation of Boolean functions using devices in the crossbar for computation as well as routing [61]. Kvatinsky et al. proposed realization of logic using devices that realize material implication and suggested their use for logic-in-memory operations, but no architecture was proposed [79]. Recently, a crossbar-based general purpose processing architecture was proposed using devices, that can realize NOR2 operations [63]. Furthermore, a compilation flow for the same using theorem provers as well as heuristic methods have been proposed [80, 81]. Gaillardon et. al [82] proposed Programmable Logic-in-Memory (PLiM) computing architecture with a single instruction, which is identical to the native function computed by ReRAM devices realising majority, and a compiler for the same was presented later in [83]. Despite the fact that the large amount of memory cells available in standalone memories offer scope for exploiting massive parallelism by performing concurrent operations, the PLiM architecture has sequential computations, resulting in significant under-utilization of the platform.

In this chapter, we propose, ReVAMP — a three-stage general purpose architecture that uses logic-in-memory operations using ReRAM crossbar array. The architecture exploits the parallelism in computing multiple logic operations in the same word. There are two key factors that are not considered by the existing logic-in-memory architecture related literature. The overhead of the control circuitry needed for enabling the logic-in-memory operations on the crossbar arrays are ignored, which is a factor of the crossbar memory size. Also, the operations performed by the controller needs be stored in form of micro-instructions. The number of micro-instructions are dependent on the Boolean function being realized and the size of each micro-instruction is a function of the crossbar memory size. These factors have to be considered while evaluating the performance of the architecture. The proposed ReVAMP architecture explicitly considers these factors. We also present a device-accurate simulation framework for the architecture and demonstrate realization of Secure Hash Algorithm 3 on the proposed architecture as a representative case study.

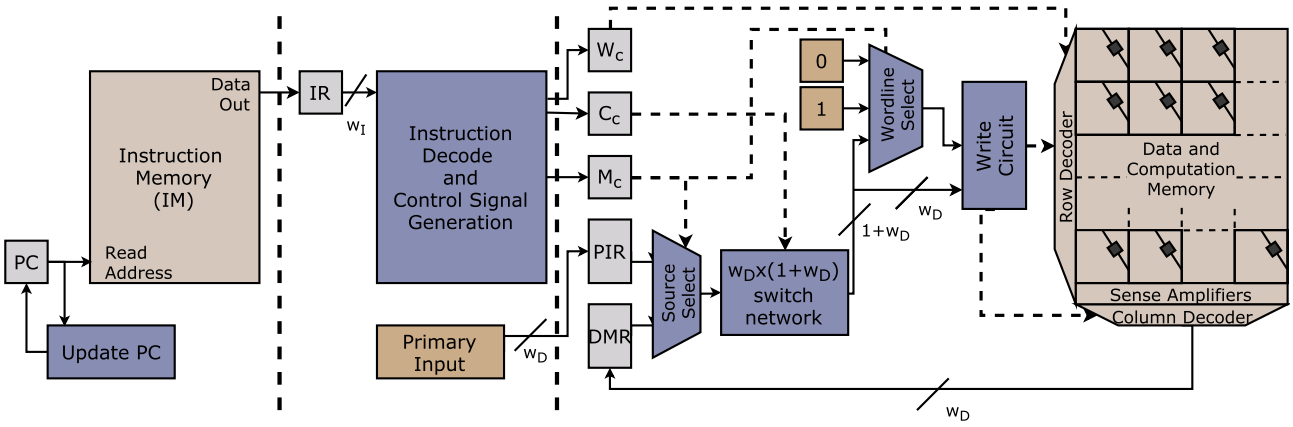



**Figure 3.1:** ReVAMP architecture. The architecture consists of 3 stages for instruction processing — Instruction Fetch, Instruction Decode and Execute. A ReRAM crossbar array is used as the Data and Computation Memory. PC: Program counter, IR: Instruction Register, $W_c$: Write control, $C_c$: Crossbar network control, $S_c$: Source select control, PIR: Primary Input Register, DMR: Data Memory Register, $w_I$, $w_D$ : word length of IM and DCM respectively.

## 3.1 ReVAMP Architecture

We present the details of *Re*RAM based *V*LIW *A*rchitecture for in-*M*emory com*P*uting (ReVAMP), depicted in Figure 3.1. The architecture uses a ReRAM crossbar memory as the Data Storage and Computation Memory (DCM), that hosts the data and performs in-memory computation.. The Instruction Memroy (IM) is a regular instruction memory accessed using the program counter (PC) — this could be either crossbar-based or conventional memory (SRAM). All the devices present in a single word of the DCM, can be operated in parallel, with each operation being the intrinsic state update function ($M_3(S, wl, \overline{bl})$). Since multiple compute operations operate in parallel, the proposed architecture is VLIW in nature. Splitting the instruction and data memory allows reduction in overall execution time, by pipelining instruction fetch and computation. The ReVAMP architecture is parameterized as shown in Table 3.1, and can be configured as necessary. The ReVAMP architecture has a three-stage pipeline with instruction fetch (IF), instruction decode (ID) and execute (EX) stages. In the IF stage, the instruction at the address held by the program counter (PC) is fetched from the IM and loaded into the instruction register (IR) before the PC is updated. In the ID stage, the instruction is read from IR to determine the control inputs for the source select multiplexer, the crossbar interconnect

**Table 3.1:** ReVAMP parameters.

| Parameter | Description |
|---|---|
| $S_D$ | Number of words in the DCM |
| $w_D$ | Number of bits in a word in DCM |
| $w_D$ | Number of primary input lines |
| $S_I$ | Number of words in the IM |
| $w_I$ | Number of bits in a word in IM |

| Read $w$ |
|---|

| Apply $w\ s\ ws\ wb\ (v\ val_1)\quad (v\ val_2)\ \ldots\ (v\ val_{w_D})$ |
|---|

**Figure 3.2:** ReVAMP instruction format.

and the write circuit.

The data memory register (DMR) stores the data read out from the DCM. The primary input register (PIR) buffers the primary input data. Both DMR and PIR are $w_D$ bits wide. Depending on the source control input $S_c$, the source select multiplexer selects either the DMR or the PIR as the data source. Thereafter, the crossbar-interconnect is used to generate the wordline and $w_D$ number of bitline inputs by appropriate permutation of the input data, as per the control signals stored in $C_c$. The crossbar-interconnect is basically a set of multiplexers, one per output, which selects one of the input $w_D$ bits. The write circuits reads the value of the target wordline from the output of the wordlines select multiplexer and the output of the crossbar-interconnect to determine the applied inputs to the row and column decoder of the DCM.

## 3.2 ReVAMP Instruction Set

The ReVAMP architecture supports two instructions—*Read* and *Apply*, in the format shown in Figure 3.2. The *Read* instruction reads the word at the address $wl$ from the DCM and stores it in the DMR. Now available in the DMR, this word can be used as input by the following instructions.

The *Apply* instruction is used for computation in the DCM. The address $w$ specifies the word in the DCM that will be computed upon. A bit flag $s$ chooses whether the inputs will be from primary input (PIR) or DMR. A two-bit flag $ws$ specifies the wordline input — 00 selects logic 0, 01 selects logic 1, 11 selects input specified by the $wb$ flag and 01 is not a valid input. The $wb$ bit-vector are used to specify the bit within the chosen data source for use as wordline input. Pairs ($v$ $val$) are used to specify bitline inputs. The bit flag $v$ indicates if the input is NOP or a valid input. Similar to $wb$, the bit-vector $val$ specifies the bit within the chosen data source for use as bitline input.

For each instruction, one bit is required to specify the opcode, and $log_2(S_D)$ bits are required to select the word. One bit is required for $s$ flag and two bits are required for the wordline source select flag $ws$. Each (v val) pair requires one bit for the $v$ flag and $log_2(w_D)$ bits for specifying the bit in the selected input source. The field $wb$ also requires $log_2(w_D)$ bits. Thus, the lengths of these instructions are:

$$IL_{Read} := 1 + log_2(S_D) \tag{3.1}$$

$$IL_{Apply} := 3 + log_2(S_D) + (1 + w_D)(1 + log_2(w_D)) \tag{3.2}$$

The word length $w_I$ of the IM should be greater than or equal to $\max(IL_{Read}, IL_{Apply})$. This concludes the description of the ReVAMP architecture.

### 3.2.1 Representative Example

We explain the functionality of the ReVAMP architecture, by means of a representative example. We consider the realization of a two input multiplexer, with $a$ and $b$ as inputs and $s$ as the select line. The multiplexer function $M$ can be expressed as :-

$$M = \overline{s}.a + s.b$$

(a)

| Cycle | Instruction | PIR | DMR |
|---|---|---|---|
| 1 | Apply 1 0 00 00 1 11 1 10 0 00 0 00 | 1 1 0 0 | 0 0 0 0 |
| 2 | Apply 0 0 00 00 1 11 1 10 0 00 0 00 | 1 1 0 0 | 0 0 0 0 |
| 3 | Read 1 | $s$ 0 0 0 | 0 0 0 0 |
| 4 | Apply 1 0 11 10 1 01 0 00 0 00 0 00 | $s$ 0 0 0 | 0 0 0 0 |
| 5 | Apply 1 0 00 00 1 10 0 00 0 00 0 00 | $s$ 0 0 0 | 0 0 0 0 |
| 6 | Apply 0 0 01 00 1 10 1 01 0 00 0 00 | $a$ $b$ 0 0 | 0 0 0 0 |
| 7 | Read 0 | $a$ $b$ 0 0 | 0 0 0 0 |
| 8 | Apply 1 1 00 00 1 01 1 10 0 00 0 00 | $a$ $b$ 0 0 | $\overline{a}$ $\overline{b}$ 0 0 |
| 9 | Read 1 | $a$ $b$ 0 0 | $\overline{a}$ $\overline{b}$ 0 0 |
| 10 | Apply 1 1 11 10 1 00 0 00 0 00 0 00 | $a$ $b$ 0 0 | $s.b$ $\overline{s}.a$ 0 0 |
| 11 | Read 1 | $a$ $b$ 0 0 | $s.b$ $\overline{s}.a$ 0 0 |

(b)

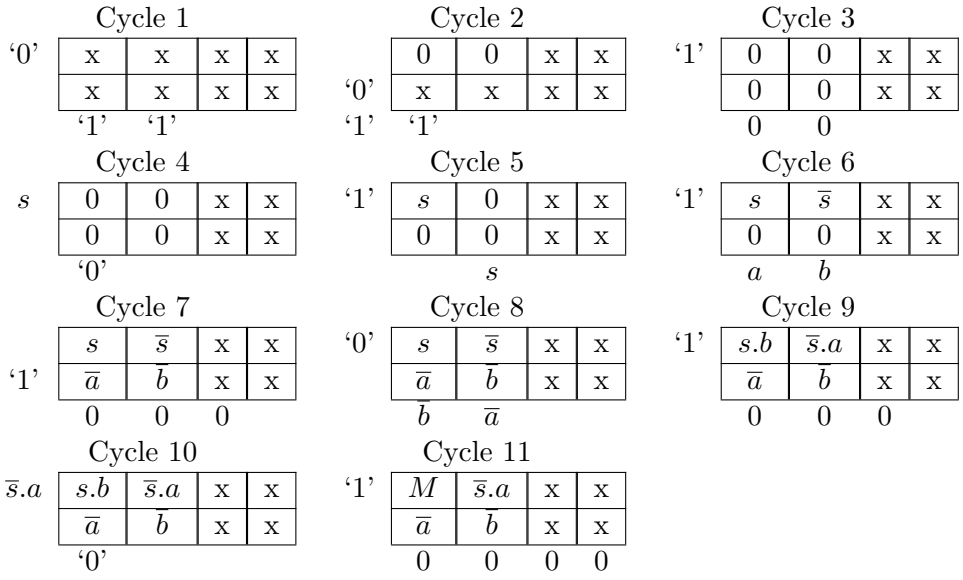



**Figure 3.3:** Multiplexer function realization using ReVAMP instructions on a $2\times3$ DCM. $M = \overline{s}.a+s.b$ (a) Instructions used to compute the function $M$, along with PIR contents. The DMR content is each cycle is also shown. (b) The state transition of the DCM in each cycle is shown.

The instructions, along with the PIR contents which realize the multiplexer function is shown in Figure 3.3a. In the first and second instructions, the device in bitline 3 and 2 of the wordline 1 and 0 are reset to HRS (0) by applying '0' to the wordline and '1' to the bitline respectively. The '1' is made available via the PIR. In the thrid cycle, we read out the content of wordline 1 to verify the RESET operation carried out in the first cycle.

In the fourth cycle, $s$ is made available via bit 2 of the PIR, and an Apply instruction is used to load it via the wordline. The bitlines 1 and 0 are not used by this instruction, by setting the valid bits to 0. Figure 3.4 explains the individual parts of this instruction. In cycle 5, $s$ is loaded via the bitline

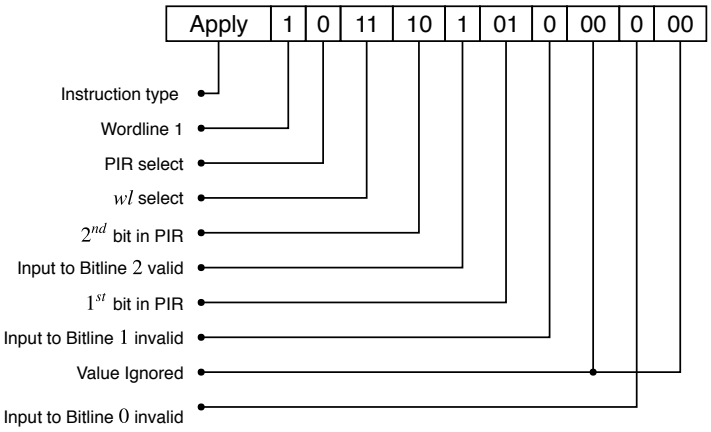



**Figure 3.4:** Functionality of individual bits in an Apply Instruction.

to be stored in inverted form, by setting the wordline input '1'. The wordline select bits are set to 01 to choose '1' as wordline input. Similarly, in cycle 6, $a$ and $b$ are loaded in inverted form to wordline 0.

In cycle 7, a Read instruction is used to read out the contents of wordline 0, which is stored in the DMR. It should be noted that the contents of the DMR is updated in the next cycle.

In cycle 8, the Apply instruction uses DMR as source, by using the source select bit $s$ to be set to 1. The Apply instruction uses 0 as wordline input, by using wordline select bits $ws$ to 00. $\overline{b}$ is available in bit 1 of the DMR, is used as input to bitline 2 by setting the bitline value bits $val_2$ to 01. Similarly, $\overline{a}$ is available in bit 2 of the DMR, is used as input to bitline 1 by setting the bitline value bits $val_1$ to

10. For bitlines 1 and 2, the bitline valid bits $v_1$ and $v_2$ are both set to 1 while $v_3$ is set to 0, as we are not using the bitline 3 in the computation.

In the cycles 9 and 10 , wordline 1 is read out and the multiplexer function $M$ is computed, as depicted in the state transition, in Figure 3.3b. In cycle 11, the wordline 1 is read out to verify the computed result.

## 3.3 Device-Accurate Simulation Setup

We setup a device-accurate simulation using Cadence AMS designer tool. AMS designer allows simulation of blocks with analog and digital components. This also permits providing stimuli to the respective blocks, i.e., either as voltage/current to the analog components or as logic 0's and 1's to the digital components. The simulation setup has the following components —

- Testbench: The testbench is specified in behavioral Verilog and is used to load instructions to IM and specify the contents of the PIR.
- Program Counter: The digital register keeps track of the current instruction address.
- Instruction Memory (Memory model : $TSN65LP8192X32$ Single Port SRAM): A traditional SRAM memory that stores the ReVAMP instructions.
- ReVAMP Controller (PDK: $TSMC\_CLN65LP$): This digital component decodes the instructions and generates the control signals for driving the peripherals connected to the crossbar array.
- ReRAM crossbar array (VerilogA model): Multiple ReRAM devices are arranged in the form of a crossbar array. The ReRAM device model has been implemented in VerilogA, using the parameters presented in Section 2.1.

- Crossbar peripherals (PDK: $TSMC65n$): The crossbar peripheral components acts a bridge to convert between digital control signals generated by the controller to voltage levels used to perform operations on the crossbar array and convert the readout current from the crossbar into digital form.

The details of the simulation setup are described in Appendix A.

## 3.4 Case Study : Secure Hash Algorithm 3

The secure transmission of IoT sensory data is of paramount importance to guard confidentiality and authenticity. However, encryption and authentication requires additional computing resources leading to significant performance overhead. An alternative approach is to use the in-memory computing capability of ReRAM. In this section, we study logic-in-memory implementation for round function of cryptographic hash algorithm known as Secure Hash Algorithm 3 (SHA-3) or Keccak. Our carefully done mapping reveals a bit/word-serial architecture for SHA-3. In that respect, the estimated throughput for ReRAM-based implementation is comparable to a highly optimized, bit-serial, lightweight CMOS realization.

### 3.4.1 SHA-3 Algorithm

SHA-3 also known as $Keccak$ is based on sponge construction function [84]. Sponge construction function takes arbitrary length of input and produces a fixed length output. The $Keccak$ algorithm uses the permutation block known as $Keccak\text{-}f$ as its core operation. The $Keccak\text{-}f$ block operates on the fixed number of $b$-bits which is the width of the permutation or bit state. We can use $b = 25$, 50, 100, 200, 400, 800 or 1600 for implementing $Keccak\text{-}f$ function. In this case study, we study the implementation of $Keccak\text{-}f$ with default size $b = 1600$.

The $Keccak$-$f$ round function consists of 5-steps with logic operations and bit-wise permutations. Each message block is operated upon by 24 rounds. The entries of input and output of $Keccak$-$f$ round function are $5 \times 5$ matrices with each entry size equal to 64-bits. Figure 3.5 shows the computations involved in $Keccak$-$f$ round function. $A$ and $RC$ (round constant) are inputs. The size of $A$ is $5\times5$ matrix. The each round function has 5 steps known as THETA, RHO, PI, CHI and IOTA. Operators $\oplus$, $\lll$, $\wedge$ and $\overline{a}$ represents the Boolean bit-wise XOR, Rotate, AND and NOT operation respectively. Variables $i$ and $j$ represents the matrix index and operations on $i$ and $j$ are in modulo 5. In RHO step, $r[i,j]$ represents the rotation matrix and $A$ is rotated according to the $r$ matrix values. The $RC$ is XORed with $A$ in IOTA step and $RC$ is different for each round. The round and rotation constants are given in [85, 86].

**Round[b](A,RC)**

| | |
|---|---|
| **THETA Step** | |
| $C[i] = A[i,0] \oplus A[i,1] \oplus A[i,2] \oplus A[i,3] \oplus A[i,4]$ | $\forall i$ in $0 \ldots 4$ |
| $D[i] = C[i\text{-}1] \oplus (C[i+1] \lll 1)$ | $\forall i$ in $0 \ldots 4$ |
| $A[i,j] = A[i,j] \oplus D[i]$ | $\forall (i,j)$ in $(0 \ldots 4, 0 \ldots 4)$ |
| **RHO and PI Step** | |
| $B[j,2i+3j] = A[i,j] \lll r[i,j]$ | $\forall (i,j)$ in $(0 \ldots 4, 0 \ldots 4)$ |
| **CHI Step** | |
| $A[i,j] = B[i,j] \oplus ((\overline{B[i+1,j]}) \wedge B[i+2,j])$ | $\forall (i,j)$ in $(0 \ldots 4, 0 \ldots 4)$ |
| **IOTA Step** | |
| $A[0,0] = A[0,0] \oplus RC$ | |
| **return A** | |

**Figure 3.5:** Round Function of $Keccak$-$f$

$Keccak$ algorithm has absorb and squeeze phases. Initially, the message is divided into blocks of size $r = 1088$ bits (for state size $b = 1600$ bits). During the absorb phase, message blocks are XORed with the first $r$-bits of the state (which are initialized to 0) and followed by a single permutation $Keccak$-$f$. After complete absorption of whole message, algorithm switches to squeeze phase. In squeeze phase, the first $r$-bits are output iteratively (again followed by a single permutation $Keccak$-$f$). The output is truncated to 256-bit hash value.

In the next section, we present the ReVAMP mapping of THETA, RHO, PI, CHI and IOTA stages of the $Keccak\text{-}f$ function.

### 3.4.2 DCM Layout For SHA-3 Implementation

For the implementation, we consider the word length $w_D$ of the DCM to be 64. The DCM can be logically partitioned into three logical partitions, as shown in Figure 3.6. The SHA-3 state partition, is 25 words long and holds the state of SHA-3. The computation memory partition is 7 words long and used for intermediate computations such as 5-input XOR, shift operations, etc. The Unshifted XOR (UX) partition holds two inverted copies of results of the 5-input XOR operations performed in Theta and is 10 word long. This partition is also used during CHI. We represent $i^{th}$ word as $w^i$ and the $k^{th}$ bit in the word as $w_k$.

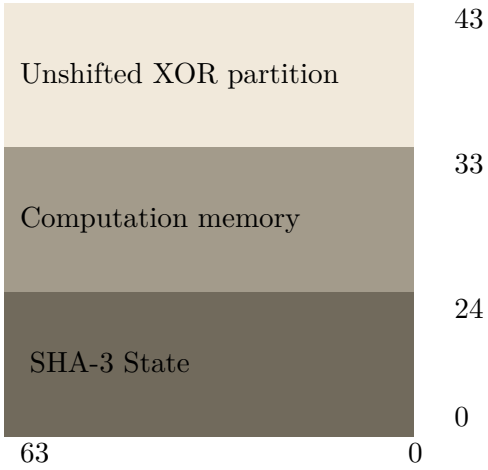



**Figure 3.6:** DCM layout for SHA-3 computation. The DCM is partitioned into 3-logical partitions for the computation, with each word 64-bits wide.

### 3.4.3 Loading State Into DCM

SHA-3 has a 1600-bit state, composed of 25 words, each of 64 bit length. The state is loaded into the SHA-3 state partition of the DCM, one word at a time. The word $w^i$, available in the PIR, is

| Cycle | Instructions | PIR content |
|---|---|---|
| $i$ | Apply 25 0 01 1 (1 63) (1 62) ... (1 0) | $w^i_{63}$ $w^i_{62}$ ... $w^i_0$ |
| $i+1$ | Read 25 | 0 0 ... 0 |
| $i+2$ | Apply i 1 01 1 (1 63) (1 62) ... (1 0) | 0 0 ... 0 |
| $i+3$ | Apply 25 0 01 1 (1 63) (1 62) ... (1 0) | 0 0 ... 0 |

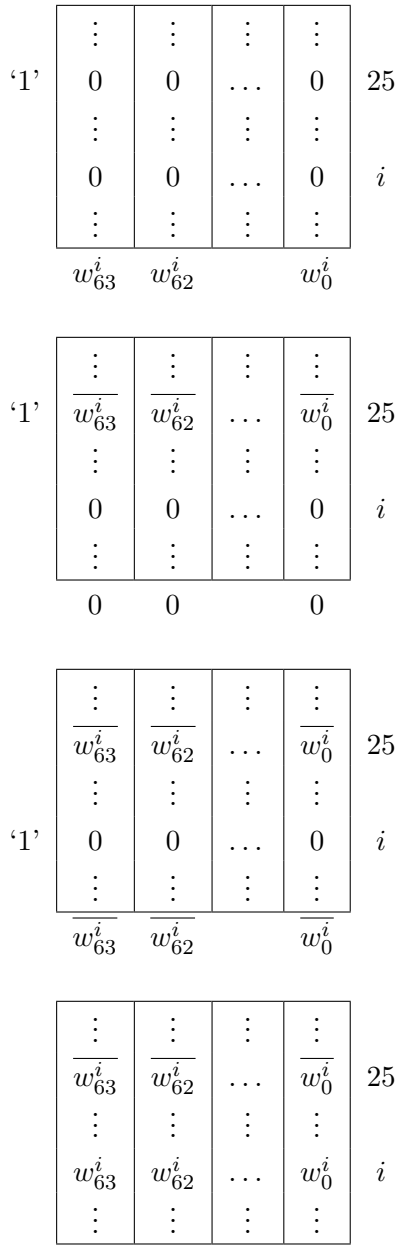
**Figure 3.7:** Instructions and PIR content to load the $i^{th}$ word of SHA-3 state, $0 \leq i \leq 63$.

loaded into the DCM by using an Apply instruction. This loads the words in inverted form, as shown in Figure 3.8 Step 1. In the next cycle, the inverted word is read out using Read instruction, which stores it in the DMR. Another Apply instruction is used to load the word in non-inverted form, by selecting writing the contents of the DMR. The temporary memory location 25 is reset using an Apply instruction and the process is repeated to load the next word. A reset operation is equivalent to Apply instruction with wordline input '0' and all the bitline inputs set to '1'. The sequence of instructions for loading the $i^{th}$ word is presented below and the corresponding changes in DCM is shown in Figure 3.8.

Each load operation for a single word requires 4 instructions, as shown in Figure 3.7. Therefore, 100 cycles are required to load the entire SHA-3 state.

### 3.4.4 Computation Of THETA

The computation of the 5-input XOR is performed in the Computation memory partition of the DCM. As evident from the steps to compute XOR of 2-inputs (presented in subsection 2.2.2), two copies of one input are required for computation. In-order to compute XOR of 4-inputs say $a$, $b$, $c$ and $d$, two copies of $a \oplus b$ can be computed, followed by separately computing $c \oplus d$. Using the procedure for computation of 2-input XORs, these intermediate XOR terms can be XORed to complete computation of the 4-input XOR. Thereafter, words except the one holding the 4-input XOR result are reset to 0. A copy of the 4-input XOR result is created and XORed with the fifth input $e$ using the 2-input XOR steps. Figure 3.9 shows the intermediate states of computation.




**Step 1:** The $i^{th}$ word in SHA-3 state is made available via the PIR and loaded to the $25^{th}$ wordline in inverted-form.

**Step 2:** The word in inverted form is read out and is stored in the DMR using *Read* instruction.

**Step 3:** The read out word, available in DMR is written to the $i^{th}$ wordline in the SHA-3 State partition using an *Apply* instruction.

The $25^{th}$ wordline can be reset to 0 and the process can be repeated for loading the next word.

**Figure 3.8:** Loading the initial SHA-3 state word into the DCM.

We deduce the number of cycles required to complete the 5-input XOR computation. Two cycles are required to make an inverted copy of the first input $a$. Six cycles are required to make three non-inverted copies of the $a$. Six cycles are needed to compute the intermediate terms and four cycles to complete computation of two copies of $a \oplus b$. For computation of $c \oplus d$, twelve cycles are required. Six additional cycles are needed to compute $a \oplus b \oplus c \oplus d$. To reset the words except the word with the result, six cycles are needed. To compute the XOR of $e$ with intermediate result, ten cycles are required. Three cycles are required to reset the words in the working memory to allow computation of the next 5-input XOR to start. Therefore, 55 cycles are required to compute 5-input XOR. We account for the delay to copy the result of the XOR to other memory location while performing circular shift with XOR.

**Fused circular shift with XOR** : From the previous stage, the 5-input XOR result is read out and two copies of it is written to the UX partition of the DCM, in inverted form. One more copy of the result is also written in inverted form. Once the previous stage terminates, one of the inverted copies is read out and written in shifted form in the Computation memory partition. The Apply instruction shown below is used to realize the circular shift operation by 1-position.

```
Apply i 1 01 1 (1 62) (1 61) ... (1 0) (1 63)
```

XOR operation of the shifted ($s^i$) and non-shifted ($r^i$) results can be performed.

$$m^i = s^i \oplus r^{(i+1)\%5},\ 0 \leq i \leq 4 \tag{3.3}$$

The computation of each $m^i$ requires six clock cycles To perform fused circular shift with XOR of the five results from the previous stage, 30 cycles would be required to complete loading of the inverted results and shifted non-inverted results, followed by 30 cycles to compute the five $m^i$ terms. Additionally, 5 cycles are needed to reset the Computation memory partition holding the $s^i$ terms. Thus, the fused circular shift with XOR requires 65 cycles in total.

| | |
|---|---|
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |

Cycle 8

| | |
|---|---|
| $a_1$ | $a_0$ |
| $a_1$ | $a_0$ |
| $a_1$ | $a_0$ |
| 0 | 0 |
| 0 | 0 |
| $\overline{a_1}$ | $\overline{a_0}$ |
| 0 | 0 |

Cycle 14

| | |
|---|---|
| $a_1.\overline{b_1}$ | $a_0.\overline{b_0}$ |
| $a_1.\overline{b_1}$ | $a_0.\overline{b_0}$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| 0 | 0 |
| 0 | 0 |
| $\overline{a_1}$ | $\overline{a_0}$ |
| 0 | 0 |

Cycle 24

| | |
|---|---|
| $a_1 \oplus b_1$ | $a_0 \oplus b_0$ |
| $a_1 \oplus b_1$ | $a_0 \oplus b_0$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| c | c |
| c | c |
| $\overline{a_1}$ | $\overline{a_0}$ |
| $\overline{c_1}$ | $\overline{c_0}$ |

Cycle 30

| | |
|---|---|
| $a_1 \oplus b_1$ | $a_0 \oplus b_0$ |
| $a_1 \oplus b_1$ | $a_0 \oplus b_0$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| $c_1 \oplus d_1$ | $c_0 \oplus d_0$ |
| $c_1 + \overline{d_1}$ | $c_0 + \overline{d_0}$ |
| $\overline{a_1}$ | $\overline{a_0}$ |
| $\overline{c_1}$ | $\overline{c_0}$ |

Cycle 32

| | |
|---|---|
| $a_1 \oplus b_1.(\overline{c_1 \oplus d_1})$ | $a_0 \oplus b_0.(\overline{c_1 \oplus d_1})$ |
| $a_1 \oplus b_1$ | $a_0 \oplus b_0$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| $c_1 \oplus d_1$ | $c_0 \oplus d_0$ |
| $c_1 + \overline{d_1}$ | $c_0 + \overline{d_0}$ |
| $\overline{a_1}$ | $\overline{a_0}$ |
| $\overline{c_1}$ | $\overline{c_0}$ |

Cycle 34

| | |
|---|---|
| $a_1 \oplus b_1.(\overline{c_1 \oplus d_1})$ | $a_0 \oplus b_0.(\overline{c_1 \oplus d_1})$ |
| $a_1 \oplus b_1 + (\overline{c_1 \oplus d_1})$ | $a_0 \oplus b_0 + (\overline{c_1 \oplus d_1})$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| $c_1 \oplus d_1$ | $c_0 \oplus d_0$ |
| $c_1 + \overline{d_1}$ | $c_0 + \overline{d_0}$ |
| $\overline{a_1}$ | $\overline{a_0}$ |
| $\overline{c_1}$ | $\overline{c_0}$ |

Cycle 36

| | |
|---|---|
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1$ | $a_0 \oplus b_0 \oplus c_0 \oplus d_0$ |
| $a_1 \oplus b_1 + (\overline{c_1 \oplus d_1})$ | $a_0 \oplus b_0 + (\overline{c_1 \oplus d_1})$ |
| $a_1 + \overline{b_1}$ | $a_0 + \overline{b_0}$ |
| $c_1 \oplus d_1$ | $c_0 \oplus d_0$ |
| $c_1 + \overline{d_1}$ | $c_0 + \overline{d_0}$ |
| $\overline{a_1}$ | $\overline{a_0}$ |
| $\overline{c_1}$ | $\overline{c_0}$ |

Cycle 42

| | |
|---|---|
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1$ | $a_0 \oplus b_0 \oplus c_0 \oplus d_0$ |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |

Cycle 46

| | |
|---|---|
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1$ | $a_0 \oplus b_0 \oplus c_0 \oplus d_0$ |
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1$ | $a_0 \oplus b_0 \oplus c_0 \oplus d_0$ |
| 0 | 0 |
| 0 | 0 |
| $\overline{a_1 \oplus b_1 \oplus c_1 \oplus d_1}$ | $\overline{a_0 \oplus b_0 \oplus c_0 \oplus d_0}$ |
| 0 | 0 |
| 0 | 0 |

Cycle 50

| | |
|---|---|
| $(a_1 \oplus b_1 \oplus c_1 \oplus d_1).\overline{e_1}$ | $(a_0 \oplus b_0 \oplus c_0 \oplus d_0).\overline{e_0}$ |
| $(a_1 \oplus b_1 \oplus c_1 \oplus d_1) + .\overline{e_1}$ | $(a_0 \oplus b_0 \oplus c_0 \oplus d_0) + .\overline{e_0}$ |
| 0 | 0 |
| 0 | 0 |
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1$ | $\overline{a_0 \oplus b_0 \oplus c_0 \oplus d_0}$ |
| 0 | 0 |
| 0 | 0 |

Cycle 52

| | |
|---|---|
| $a_1 \oplus b_1 \oplus c_1 \oplus d_1 \oplus e_1$ | $a_0 \oplus b_0 \oplus c_0 \oplus d_0 \oplus e_0$ |
| $(a_1 \oplus b_1 \oplus c_1 \oplus d_1) + .\overline{e_1}$ | $(a_0 \oplus b_0 \oplus c_0 \oplus d_0) + .\overline{e_0}$ |
| 0 | 0 |
| 0 | 0 |
| $\overline{a_1 \oplus b_1 \oplus c_1 \oplus d_1}$ | $\overline{a_0 \oplus b_0 \oplus c_0 \oplus d_0}$ |
| 0 | 0 |
| 0 | 0 |

**Figure 3.9:** Intermediate states of computation of 5-input XOR using 7 wordlines.

**2-input XOR to update word** : This stage involves XORing each word $w^i$ in the SHA-3 state with the proper $m^j$ terms. We compute XNOR of $w^i$ with proper $m^j$ term in the Compute memory partition, using the operations shown in Figure 3.10. Once the XNOR has been computed, the original location in SHA-3 state partition where word $w^i$ is present, is reset. Then, the newly computed result is written to that location using Apply instructions, thereby storing it in non-inverted form. Additionally, two cycles are needed to reset the Compute memory locations before updating the next state word begins. Each word update requires 15 cycles and thereby the entire state update requires 375 cycles.

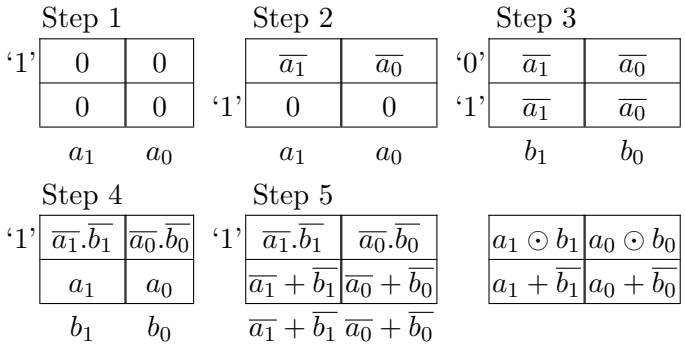



**Figure 3.10:** Computing XNOR of two variables $a_i$ and $b_i$.

### 3.4.5 Computation Of The RHO stage integrated With PI stage

We merge the RHO stage with PI stage. Let $k^i$ denote the inverted output of RHO stage for SHA-3 state word $w^i$. The word $w^i$ is read out and the RHO output is written to a temporary location in the Computation memory partition. The new location $j$ in SHA-3 state partition where $i^{th}$ word will be written to is then read out and the shifted value is written to another temporary location. The $j^{th}$ word location in Computation memory is reset and the shifted word $k^i$ is written to that location. The temporary location holding $k^i$ is reset. This process is repeated till all the words have been shifted and written to their new locations. The computation of RHO with PI is visually shown in Figure 3.11.

Using the proposed steps, each word effectively requires 4 steps, two steps require Read and Apply instructions while other two steps are reset operations, hence requires 6 cycles each. Therefore, this

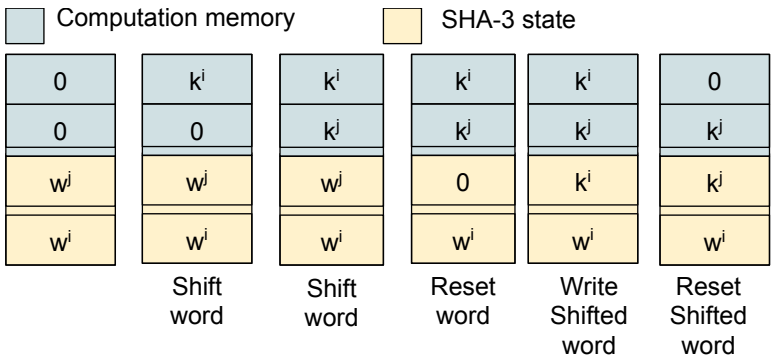



**Figure 3.11:** Schematic for combined execution of RHO with PI.

stage will require 150 cycles in total.

### 3.4.6 Computation Of The CHI stage

The CHI stage of the algorithm effectively partitions the SHA-3 state into 5 sets, each with 5 words. The updated word $u^i$ can expressed as $u^i = \overline{w^i \odot \overline{w^j}.w^k}$, where $\odot$ represents the XNOR operation. Each word is read out and stored in inverted form in three locations — two in UX partition and one in Computation memory partition. One of the inverted copies of $w^k$ is read out from UX partition and ANDed with $w^j$ in the Computation memory partition. Similarly, the remaining AND terms are computed for each word in the set. Then XNOR term is computed in the UX partition for each word in the set. The word locations in the SHA-3 state partition are reset and the XNOR results are written to store the updated word $u^i$. The UX partition and Computation memory partition are reset to update the next set of 5 words. Intermediate steps of the CHI stage are shown in Figure 3.12. To update a set of 5 words, 85 cycles are needed with 15 additional cycles to reset the temporary result locations. 500 cycles are needed to complete this stage.

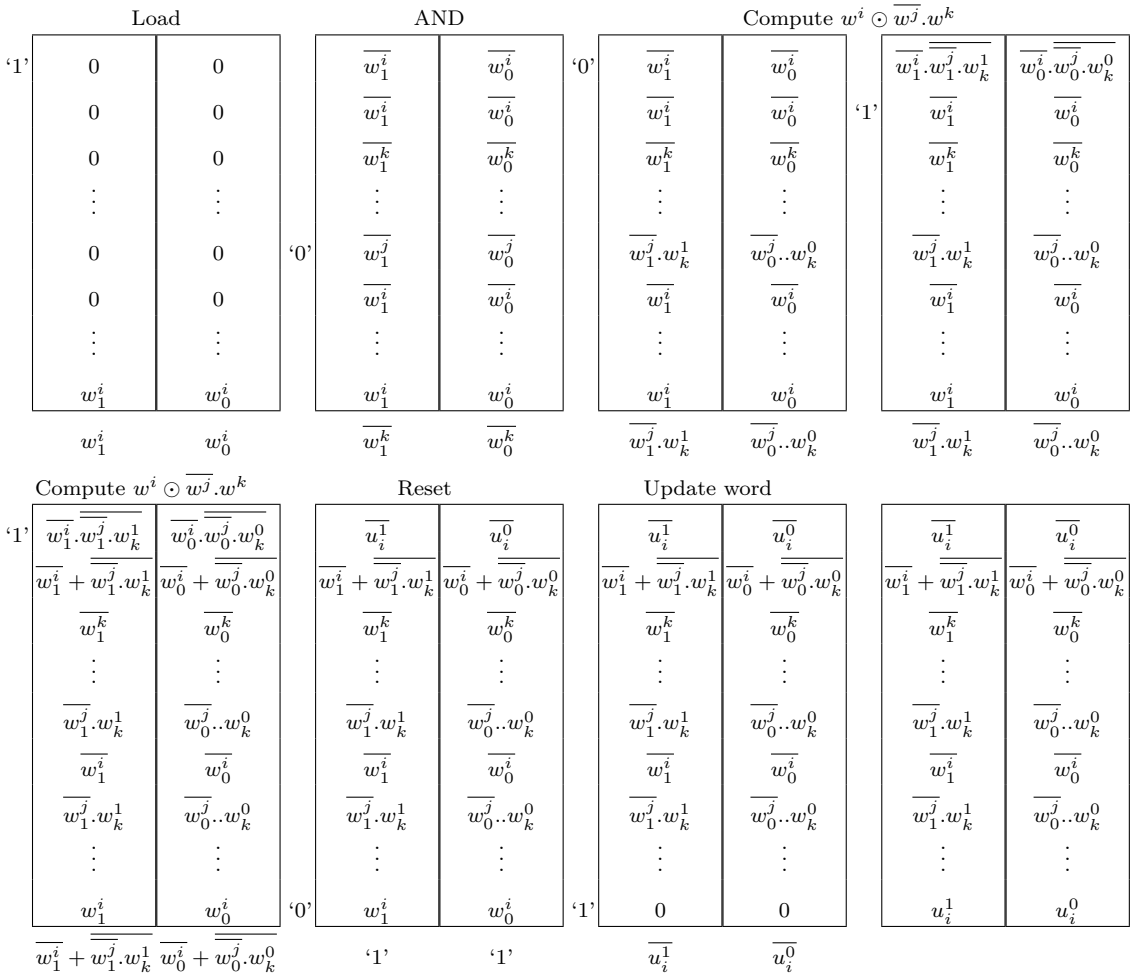



**Figure 3.12:** Intermediate stages of the DCM during computation of CHI.

### 3.4.7 Computation Of The IOTA Stage

The IOTA stage involves computation of a single XOR with two inputs which requires 12 cycles with 2 additional cycles to reset locations used for computation.

This completes the implementation details of the SHA-3 round implementation. In the next section, we present the estimates of the throughput and present comparison against existing implementations.

### 3.4.8 Experimental Results

The number of instructions required to complete each stage of SHA-3 round is shown in Table 3.2. Each instruction effectively takes one cycle to get executed, except the first instruction which would require 3 cycles. Therefore, each round requires 1261 clocks cycles to complete execution for state size 1600 bits. Equation (3.4) used for calculation of throughput.

$$Throughput = \frac{Blocksize}{Latency} \times Frequency \tag{3.4}$$

Where *Blocksize* is the length of message block (Blocksize is 1088 for SHA-3 state size $b = 1600$ bits) and *Latency* is number of clock cycles required for processing single message block. Table 3.2 shows the number of instructions required in each stage and the overall implementation requires 1309 instructions. For processing single message block, 24 rounds are required. The latency for the proposed ReRAM based SHA-3 implementation is 27918 ($= 1261 + 23 \times 1159$). This is because only for the first round, initial state has to be loaded. For the rest of the rounds, the initial state is the final state from the previous round and hence does not require the Loading State stage. To estimate the performance, we assume mature ReRAM technology with $1ns$ access time, based on [87]. The effective throughput of the implementation is 38.97 Mbps. From the perspective of area in terms of ReRAM devices required, the proposed implementation requires 43 words only, each of 64-bit length for computation. Assuming the DCM to be addressed by 6 bits ($2^6 > 43$), each Apply instruction requires at 465 bits and we assume

that the Read instruction is padded with 0s to make it of the same length as the Apply instruction. The proposed implementation requires ≈ 80 KB of memory for storing the instructions, considering 32-bit aligned memory access.

**Table 3.2:** Results of Logic-in-memory implementation of SHA-3 round function using ReVAMP instructions.

| **Operation** | **#Instructions** |
|---|---|
| Loading State | 100 |
| 5-input XOR | 55 |
| Fused Circular Shift with XOR | 65 |
| 2-input XOR to update state | 375 |
| RHO with PI | 150 |
| CHI | 500 |
| IOTA | 14 |
| Total | 1259 |

**Table 3.3:** Summary of SHA-3 implementation using various technologies.

| Impl. | Tech. | $f$ (MHz) | Area | #Rounds | Latency (cycles) | Tput. (Mbps) |
|---|---|---|---|---|---|---|
| [88] | Virtex 4 | 143 | 2.02k slices | 24 | 25 | 6070 |
| [89] | Virtex 5 | 122 | 1.4k slices | 24 | 25 | 4800 |
| [88] | 90nm | 455 | 10.5 KGE | 24 | 25 | 19320 |
| [90] | 180nm | 488 | 56.31 KGE | 24 | 25 | 21230 |
| V1[91] | 130nm | 1 | 5.52 KGE | 24 | 10.7k | 0.044 |
| V2[91] | 130nm | 1 | 5.9 KGE | 24 | 7.4k | 0.064 |
| Proposed ReVAMP | | $10^3$ | | 24 | 27.92k | 38.97 |

Even though comparison of implementation across various technology nodes is difficult, we present a brief overview of the existing implementations. Table 3.3 presents a summary of the various existing implementations along with the proposed implementation. In [88] and [89], fast single cycle implementation based on FPGA is presented. For ASIC, [88] and [90] presented single cycle implementation with maximum throughput of about 21 Gbps. P. Pessl et al. presented bit-serial implmentation ASIC

implementation, targeted towards RFID applications [91]. The proposed SHA-3 implementation harnesses bit-level parallelism and has latency similar to [91], but allows a much higher clock frequency. The current implementation is natively word-serial in nature. By increasing word length $w_D$, overall latency can be reduced. However, that would come at an expense of increased peripheral circuitry and increased width of the instruction memory. Therefore, the proposed implementation cannot be optimized to achieve latency close to fast ASIC implementations such as [88] and [90]. However, the proposed implementation has a very low memory footprint for computation (2.6875 Kb) along with an instruction memory of about 80 KB enabled by a light-weight controller.

## 3.5 Discussion

In this chapter, we have proposed a novel general purpose computing architecture using ReRAMs. This is the first ReRAM-crossbar based general purpose computing architecture that harnessess the inherent parallelism of the crossbar arrays. This is based on the observation that computations in same word can be executed in parallel, under certain crossbar constraints. This is enabled by the instruction set of the proposed architecture which packs multiple computations into a single instruction. We presented a detailed case-study of logic-in-memory realization of SHA-3 using the ReVAMP architecture. To enable adoption of the architecture, automated design flow support to map arbitrary functionality on the architecture is needed. This is the exact problem addressed in the following chapters. In chapter 4, a generic computation model for in-memory computing is presented, along with technology mapping solutions for achieving optimal delay and mapping under area-constraints. Chapter 5 formally defines the problem of crossbar-constrained technology mapping and presents a multi-step approach addressing the problem under various constraints.

# 4 Technology Mapping For 1S1R devices

Traditionally, the design automation process for realization of a desired function using a specific technology is a multi-step process [92]. Any arbitrary Boolean function with common sub-expression can be represented with a DAG, where the vertices in the graph represent some Boolean operator (AND, OR, Majority, etc.) while the edges can be labelled with {0,1} — 0 signifying inverted value of the signal associated while 1 represents non-inverted value of the associated vertex [93]. Coarsely, the DAG can be thought of as a circuit with the directed edges signifying the computation direction from the inputs to the outputs. Technology-independent logic synthesis is the first step, where the input Boolean function is restructured without any specific technology constraints. The hardness of logic synthesis has been a widely studied topic for a variety of representations [94, 95]. This is generally followed by a technology-dependent optimization phase, where technology specific hints are used for optimization of the data structure obtained from the first step. The final step is technology mapping, which takes the optimized function representation to implement it using technology-specific constraints.

In this chapter, we introduce a generic computation model for in-memory computing using logic-

in-memory devices, that we use to define the technology mapping problem. Thereafter, we explore variants of the technology mapping problem for in-memory computing specific to ReRAM devices that realize Majority with a single input inverted. We use MIG for representation of Boolean function, for use as input to the technology mapping flow. Given an MIG, we develop a delay optimal mapping to ReRAM devices. For any $k$-level MIG, our optimal mapping algorithm generates an optimal mapping with $k$ delay. The delay reduction, achieved by our mapping is at least 3× better than that attained by the naive technology mapping proposed in [96]. We propose a heuristic to reduce device count. The proposed algorithm, allows reduction in number of devices required for mapping on average by *56%* (max *92.63%*).

We also explore the problem of area-constrained technology mapping (ACTMP) for ReRAM devices. We propose an ILP formulation for ACTMP to obtain optimal solution to the problem. We start from the mapping idea of the delay optimal technology mapping and construct a so called State Dependency Graph (SDG). SDG reflects the dependencies between subsequent operations on a device and read dependencies between operations of different devices. The ReRAM devices that will be used for area-constrained technology mapping are termed as *elements*. Under area constraints, the problem is to obtain a valid schedule that does not violate the dependencies present in the SDG and at the same time maps each device in the SDG to an element such that no two devices are mapped to the same element simultaneously. However the ILP does not scale for larger problem instances, therefore we propose a scalable heuristic algorithm for the same.

## 4.1 Preliminaries

A representative overall design automation flow is shown in Figure 4.1.

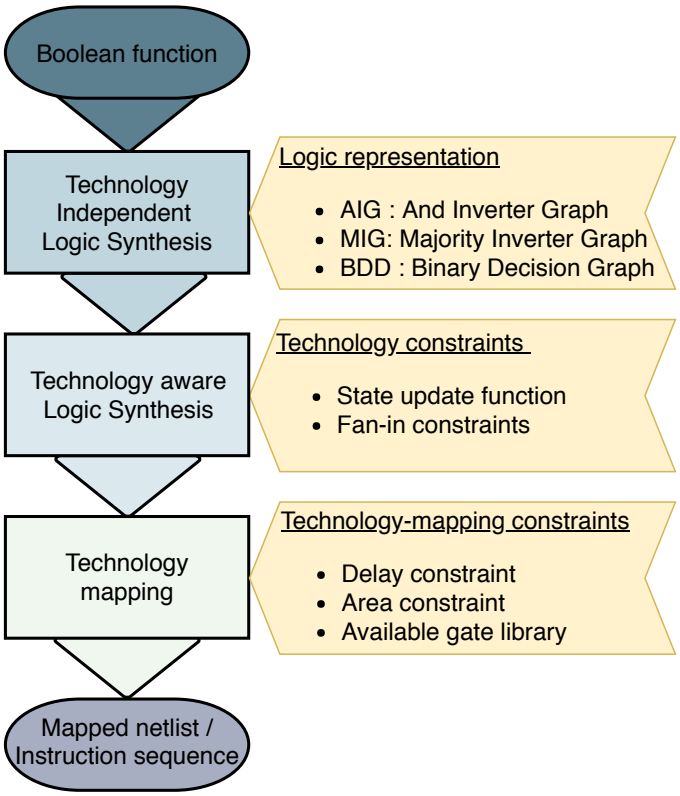



**Figure 4.1:** Design automation flow from Boolean function specification to a technology mapped netlist.

### 4.1.1 Technology-independent logic synthesis

The functionality is specified as an input Boolean function. This can either be in the form of a truth table, or as a description written in a Hardware Description Language, such as Verilog, blif, etc. The process of logic synthesis converts the input representation into a generic representation format — either a two level representation or a logic network for optimization.

One such representation of Boolean function is using Sum-of-Product form, which is the disjunction (OR) of *minterms*, with each minterm consisting of the conjunction (AND) of variables, either in regular or complemented form [97]. Another similar two-level representation is Exclusive Sum-of-Product (ESOP) that uses Exclusive-OR of minterms for function representation [98].

Binary Decision Diagram (BDD) is another data-structure that can be used for canonical rep-

resentation of Boolean function, that uses the idea of *Shannon expansion* [99]. Further, multi-level logic networks have been used extensively for logic minimization and circuit synthesis [100–102]. We introduce some notations related to graphs and formally define logic networks that are used extensively in this thesis for addressing the design automation problems.

**Definition 4.1** (DAG)**.** *A DAG $G = \langle V, E \rangle$, is a simple digraph without cycles, where $V$ is the set of nodes and $E$ is the set of edges. A directed edge in $G$ is defined as $v_i \longrightarrow v_j$, where $v_i, v_j \in V$. $v_i$ is the predecessor of node $v_j$ and $v_j$ is the successor of node $v_i$.*

We define the sets $pred(v)$ and $succ(v)$ as follows :

$$pred(v) = \{x\}, \forall x \in V \mid x \longrightarrow v$$

$$succ(v) = \{x\}, \forall x \in V \mid v \longrightarrow x$$

The indegree $\delta^-(v)$ and outdegree $\delta^+(v)$ of a node $v$ is defined as :-

$$\delta^-(v) = |pred(v)|$$

$$\delta^+(v) = |succ(v)|$$

A node $v$ is termed as *leaf* node, if $\delta^-(v) = 0$. Rest of the nodes in the DAG are termed as *internal* nodes.

**Definition 4.2** ($level(n)$)**.** *The level of a node $n$ is defined as :-*

$$level(n) = \begin{cases} 1, \textit{if } n \textit{ is a leaf node} \\ \max(level(p)) + 1, s \in pred(v) \textit{and } n \textit{ is an internal node} \end{cases}$$

### Logic Network

A logic network is defined as a DAG with nodes that represent logic functions and incoming edges signify input to the function. A directed edge $i \rightarrow j$ exists if the output of the node $i$ is an input to node $j$. A Primary Input (PI) node is either a logic constant 0/1 or Boolean variable that are input to the Boolean function. Two logic networks are termed *equivalent* when they represent the same Boolean function. A logic network is termed as *homogeneous* if each node has the same indegree and represents the same logic function. In homogeneous logic networks, an edge can be marked as regular or complemented.

**Definition 4.3** (AIG)**.** *An And Inverter Graph (AIG) is a homogeneous logic network with indegree equal to 2 and with each node representing the Boolean AND function. In an AIG, the edges are marked by a regular or complemented attribute [6].*

*Majority function:* The $n$-input ($n$ is odd) majority function $M_n$ returns logic value assumed by more than half of the inputs. For Boolean logic and $n$=3, the majority operator is expressed in (4.1).

$$M_3(a, b, c) = ab + bc + ca \tag{4.1}$$

**Definition 4.4** (MIG)**.** *A Majority Inverter Graph (MIG) is a homogeneous logic network with indegree equal to 3 and with each node representing the majority function. In an MIG, the edges are marked by a regular or complemented attribute [103].*

**Example 4.1.** *We explain the terms introduced w.r.t the MIG shown in Figure 4.2. Nodes A, B, C, D and E are the PIs, represented by square nodes. $S_1$, $S_2$, $S_3$ and $S_4$ are internal nodes, represented by circular nodes. For node $S_4$, node $S_1$, $S_2$ and $S_3$ are predecessors while $S_4$ is the neighbour for the same set of nodes. Edge $S_1 \rightarrow S_4$ is a regular edge while $S_3 \rightarrow S_4$ is an inverted edge. Nodes A, B, C, D and E are at level zero. Nodes $S_1$, $S_2$ and $S_3$ are at level two. Node $S_4$ is at level three. The output node $O_1$ is driven by internal node $S_4$. The depth of the MIG is equal to three, since two is the largest*

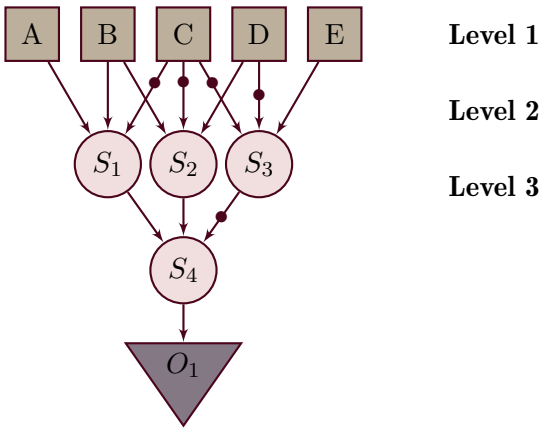



**Figure 4.2:** A Majority Inverter Graph

*node level in the graph.*

Logic synthesis transforms the input logic network to optimize some criteria, such as reduction in the number of vertices or number of logic levels, etc. An excellent background on the topic can be obtained in [104]. A variety of tools have been developed both by industrial efforts as well as in academic efforts, for logic synthesis and verification [5, 101, 105–107]. ABC is an open-source tool that allows scalable logic transformations based on AIGs, with a variety of innovative algorithms [105]. Generic logic synthesis techniques based on MIGs have been developed extensively in the recent years [108–110]. The output of the existing logic synthesis tools can be used directly as input to the technology mapping phase. We explain the technology-aware logic synthesis optimizations in Chapter 6.

### 4.1.2 Technology mapping for ReRAM based in-memory computing

The technology mapping problem for ReRAM based in-memory computing is to determine a sequence of inputs, to be applied, to the wordlines and the bitlines of ReRAM devices to compute a given function. The *delay* of the obtained mapping is equal to the number of steps that the mapping contains. We say that the mapping is *delay optimal* if the number of steps is minimum. The number of devices used in a mapping solution determines the *area* of the solution. We use the terms operation and

instruction interchangeably.

Multiple works address the issue of design automation for computation using memristive devices. Lehtonen et al. presented a methodology for computing arbitrary Boolean functions using devices that realize material implication [111]. The authors consider two disjoint set of memristors — one for storage of inputs while the other set of devices are used for computation and are termed as *working memristors*. To represent the input Boolean function, *conjunctive normal form* is used. For any Boolean function with $n$ inputs and $m$-outputs, $m + 2$ working memristors are required for computing the function. For $n$-input Boolean function with a single output, three working memristor are sufficient for computation. These bound was further to reduced to two working memristor by Poikonen et al. [112]. However, no bound on the delay of mapping was presented. Raghuvanshi et al. presented the realization of various Boolean operators (NOT, NAND, XOR, etc) using memristors realizing material implication [113]. A novel data structure, called *Imply Sequence Diagram* was introduced to represent the computation using memristors and presented mapping techniques using Karnaugh Map as well as Sum-of-Products representation. Techniques were introduced for reduction of delay as well as number of computations required to compute a given Boolean function. Chattopadhyay et al. presented a multi-stage mapping technique for Boolean functions to devices realizing material implication, starting from an Or-Inverter Graph representation and used heuristics for reduction of number of devices as well as delay [114].

Shirinzadeh et al. proposed heuristics for logic synthesis of MIG for two variants of ReRAM —one realising material implication and the other realising majority function. The authors used a naïve technology mapping with delay of $3k + c$ cycles, for an MIG with $k$ levels and $c$ number of levels with ingoing complemented edges. Soeken et al. used MIG for representing the Boolean function, and presented some techniques to optimize the MIG to reduce number of device required to map the function using devices realizing Boolean majority with a single input inverted [83].

The hardness of the technology mapping problem has been studied in the context of traditional CMOS circuits and FPGA mapping. For CMOS circuits, the delay optimal technology mapping problem can be solved in polynomial time [115] while the *minimum-area* variant is NP-hard [116]. Similar results

have been established for technology mapping for FPGAs. FPGAs can realize any $k$-input Boolean function by means of a $k$-input Look-Up Table (LUT). Cong et al. established that the minimal-delay technology mapping for LUT based FPGAs is polynomial time solvable [117]. The *minimum area* variant of the problem was proved to be NP-hard [118, 119]. In the context of in-memory computing, we provide an algorithm and a formal proof to show the delay optimal technology mapping problem can be solved in polynomial time.

## 4.2 Computation Model and Operation Representation

We refer to the in-memory computing devices used in a set of operations as *devices*. In order to uniquely refer to the devices on a target platform available for mapping, we refer to these as *elements*. Thus, multiple devices used in a set of operations can be mapped to the same element, as long as this mapping does not overlap in time. We consider a controller unit that enables free communication between all lines, i.e., in a clock cycle or in a step, a device can be read out and the read out value can be applied as input to the wordline or bitline of any other device. In addition, we also assume that the inputs and the inverted values of the inputs, of the Boolean function to be mapped, constants '1' and '0', can be applied directly to the wordlines and bitlines of the devices.

Each in-memory computing device has an internal state ($S^t$), where $t$ denotes an arbitrary time step. The next state of the device ($S^{t+1}$) is the result of a state update function $f$ defined over the current state ($S^t$) and inputs $I_j$, where $S^t, I_j \in \{0, 1\}$. The inputs can be either constants or previously computed states of other devices. The exact number of inputs is determined by the technology. $f$ is assumed to complete computation in a single logical time step.

$$S^{t+1} = f(S^t, I_1, I_2, \ldots, I_n) \tag{4.2}$$

For 1S1R devices, $f$ is the Majority three function with an input inverted. In general, an in-memory

computing device permits three basic operations — COMPUTE, READ and RESET. Additionally, the devices natively support COPY operations, since they are fundamentally memory devices. The representation of an operation is-

$$\boxed{DevOpId \quad t \quad Operation}$$

where *DevOpId* denotes the operation identifier in step $t$, and *Operation* specifies the details of the in-memory operation. Individual operation types are explained below.

- COMPUTE - A COMPUTE operation updates the internal state of a device. It can be written as-

$$\boxed{d_x^i \quad t \quad \text{COMPUTE } I_1, I_2, \ldots, I_n \rightarrow S_x^{t+1}}$$

  where $I_1, I_2, \ldots, I_n \in \{C_j, S_{x'}^t\} |\ x' \neq x$. $C_j$ represents the primary inputs (PI), which are logical constants ('0' or '1'). $S_{x'}^t$ represents the internal state of device $x'$ in step $t$. $S_x^{t+1} = f(S_x^t, I_1, I_2, \ldots, I_n)$, is the state of device $x$ at the $(t+1)^{th}$ instant. Therefore, the $i^{th}$ operation on device $x$ in the $t^{th}$ step updates the state of device $x$ from $S_x^t$ to $S_x^{t+1}$ on application of inputs $I_1, I_2, \ldots, I_n$.

- READ - The READ operation reads the internal state of a device. The general representation of a READ operation is given by-

$$\boxed{d_x^i \quad t \quad \text{READ} \leftarrow S_x^k}$$

  where $k < t$ and $S_x^k$ is the last computed state of device $x$. A READ operation in step $t$ allows the state of the device $S_x^k$ to be used as input by any COMPUTE operation performed in the same step $t$. We consider that the reads are non-destructive, i.e., the READ operation does not change the internal state of the device.

- RESET - A device can be reset to a known state. The general representation of a RESET operation is given by-

$$\boxed{d_x^i \quad t \quad \text{RESET} \rightarrow S^R}$$

The $i^{th}$ operation on device $x$ in the $t^{th}$ step resets the internal state of device $x$ to a known state $S^R$.

As we are considering in-memory computing, the devices are memory elements where COPY is a native operation.

- COPY - Using a COPY operation, the state of a device is stored in a new device. The general representation of a COPY operation is given by-

$$\boxed{d_x^i \quad t \quad \text{COPY } S_y^k \rightarrow S_x^{t+1}}$$

The $i^{th}$ operation on device $x$ at $t^{th}$ step is a COPY operation. A COPY operation stores the state of device $y$, computed at some time step $k$, to the device $x$, where $k < t$.

In a given step, these four basic operations cannot occur simultaneously on a device. A sequence of these operations realize any given input function $F$, depending on the intrinsic state update function $f$.

## 4.3 Delay Optimal Technology Mapping

The native function computed by ReRAM devices (1S1R) is Boolean Majority three function with a single input inverted. MIG allows representation using Majority and inversion which makes it inherently suitable for logic representation for usage in the design automation flow for logic realization using ReRAM devices. We begin by presenting some MIG transformation techniques, that are used to develop the delay optimal technology mapping algorithm. This is followed by the algorithm for delay optimal

mapping for arbitrary MIGs.

### 4.3.1 MIG Transformation Techniques

The majority function with the bitline input inverted is the intrinsic state update function $f$ for ReRAM devices (specifically 1S1R devices).

$$f = M_3(S, wl, \neg bl) = S.wl + wl.\neg bl + \neg bl.S \tag{4.3}$$

We use MIG for representation of the input target function $F$ to be mapped using ReRAM devices. The goal to create an effective mapping for MIG nodes such that, each node in the MIG can be effectively computed in a single cycle. To do so, we begin by formally defining *host* and *device* for a node.

**Definition 4.5** (host(n))**.** *If a node n in a MIG is evaluated in a device that holds one of its predecessor p, which is not a constant or a primary input, then p is the host of node n.*

**Definition 4.6** (device(n))**.** *The device in which a node n is evaluated, is the device of the node.*

Let us consider that $S_i$ is the state stored in the device $D_i$. Intuitively, for mapping a MIG node say, $S_4 = M_3(S_1, S_2, \neg S_3)$, we can read out $S_2$ and $S_3$ and apply to the wordline and bitline of device $D_1$ respectively. In this case, $S_1$ is used as *host* for node $S_4$ and the corresponding *device* for $D_1$. Thus, it can be seen that for this case, we can compute an MIG node in a single cycle/step, without using any additional devices as shown in Figure 4.3.

Let us consider the MIG in Figure 4.4 (on the left). Node $S_4$ computes the majority function $M_3(S_1, S_2, S_3)$. We can observe that it does not have a negated input term, which is required for mapping directly to a 1S1R device. We could use a step to compute the inversion of $S_3$ and in the next cycle, apply $S_2$ and $\neg S_3$ to the wordline and bitline of the device holding $S_1$, with node $S_1$ acting as host for node $S_4$, but such a mapping will require two cycles. To perform the computation in a single

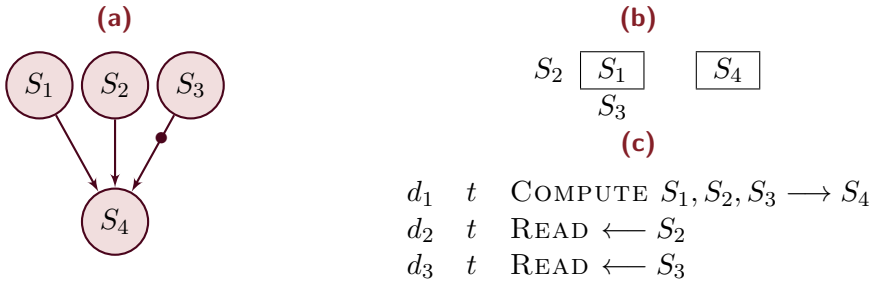



**Figure 4.3:** Single cycle computation of a MIG node $S_4$. (a) A MIG node with a single inverted input. (b) Computing $S_4$ in the device holding $S_1$, by applying $S_2$ and $S_3$ as wordline and bitline input. (c) $d_1$, $d_2$ and $d_3$ are three devices that are assumed to have the values $S_1$, $S_2$ and $S_3$ respectively. The computation of $S_4$ represented in terms of operations in some cycle $t$.

step, we can rewrite the computation for node $S_4$ as :

$$M_3(S_1, S_2, S_3) = M_3(S_1, S_2, \neg(\neg S_3)) \tag{4.4}$$

This can be represented as the transformed MIG shown in Figure 4.4 (on the right). Now, for mapping $S_4$, we can choose $S_1$ to be *host* and apply $S_2$ and $S_3$ to the wordline and bitline of the device holding $S_1$ in a single cycle. As seen in the transformed MIG, we compute $\neg S_3$ directly by inverting its inputs. We term such a transformation as *inversion* transformation.

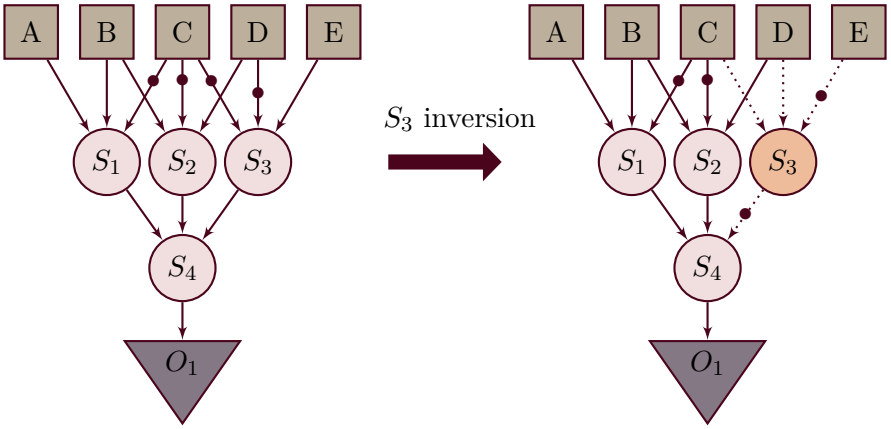



**Figure 4.4:** Inversion transformation

In the previous examples, we chose a predecessor node of the current node to act as *host*. For the MIG shown in Figure 4.5, each predecessor $p$ of node $S_4$, with non-inverted edges have more than one

out-degree. The predecessor $p$ has to be read out and and applied to multiple devices in a single step, hence we cannot use $p$ as *host*. Formally, a predecessor can be defined *busy* as follows.

**Definition 4.7** (busy predecessor)**.** *A predecessor node $p$ is considered busy for a node $n$ at level $l_n$, iff the number of successors of node $p$ at level $l \geq l_n$, is greater than one.*

For the current example, if we replicate the *busy* node $S_1$, each replica node will have one outdegree and hence each of the replica nodes can be used as *host*. For node $S_4$ and $S_5$, node $S_1^{r1}$ and $S_1^{r2}$ is used as *host* respectively. We term such a transformation as *busy replication* transformation.

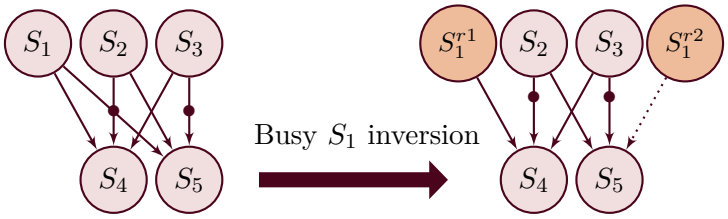



**Figure 4.5:** Busy replication transformation

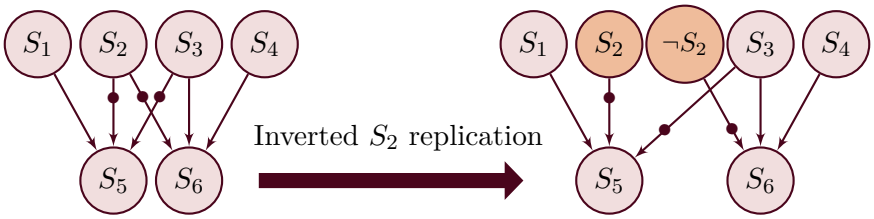



**Figure 4.6:** Inversion replication transformation

A predecessor node $p$ can have two neighbors such that one neighbor requires $p$ and other neighbor requires $\neg p$ as input. Let us consider the MIG in Figure 4.6. For node $S_6$, node $S_4$ can be used as a *host* and $S_3$ and $S_2$ is applied to the wordline and bitline of the device holding $S_4$. For node $S_5$, node $S_1$ can be chosen as *host*. Since two incoming edges to node $S_5$ are inverted, we can apply one predecessor to the bitline for inversion. For the other, we need the inverted input directly. To do so, we introduce a new node $\neg S_2$ which is inverted $S_2$. With this new node, we can apply $\neg S_2$ and $S_3$ to the wordline and bitline of the device holding $S_1$ for computation of node $S_4$. This transformation of MIG is termed as *inversion replication* and the transformed MIG is shown in Figure 4.6 (b).

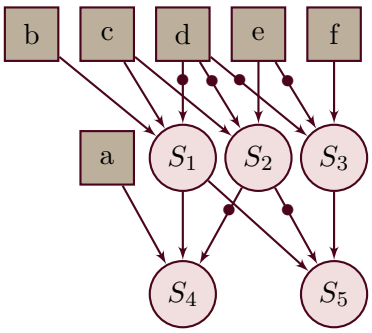



**Figure 4.7:** Preloading for node $S_4$ prevents *busy* replication

Replication of internal nodes at a higher level can lead to replication of nodes at lower levels, leading to higher number of devices being used for mapping and therefore should be avoided wherever possible. In Figure 4.7, for node $S_4$, predecessor node $S_1$ is *busy* and hence cannot be used as *host* without replication. Instead, we can load the primary input $a$ in the previous cycle in a new device $d$ and in the current cycle, apply $S_1$ and $S_2$ to the wordline and bitline of the device $d$ respectively. This concept is termed as *preloading*, where a primary input or a constant is loaded in a new device $d$ in the immediately previous cycle, followed by applying the appropriate inputs to the wordline and bitline of device $d$ in the current cycle for computation of the node.

### 4.3.2 Delay Optimal Mapping Algorithm

The delay optimal technology mapping of MIG to 1S1R devices is presented in Algorithm 1. In this subsection, we explain the algorithm briefly. We begin to process the nodes bottom up, i.e., all nodes at a higher level are always processed before a node in a lower level is processed. This is achieved by storing the nodes in *nodeHeap*, which is a max-heap [120] with the priority defined by the *level* of the node.

The key component of the mapping algorithm is the ProcessNode procedure. For processing a node, we need to determine which predecessor will be used as *host* or if we need to *preload* a new device for evaluating the node. If a node is evaluated in a predecessor of the node acting as *host*, then

**Algorithm 1:** Delay Optimal Technology Mapping algorithm for In-memory Computing

```
Data: G, pi, po
Result: n2DMap,opList
/* prio(n_i)>prio(n_j), if level(n_i)>level(n_j)                    */
nodeHeap = maxHeap ();
n2DMap = HashMap ();
for node ∈ po do
    nodeHeap.add (node);
while nodeHeap ≠ ∅ do
    node = nodeHeap.pop();
    ProcessNode(node);
PropagateDevice();
```

no new device is introduced in the mapping solution, else a new device is required for evaluation of the node, increasing the device count by one. If a predecessor of node is an internal node and the edge connecting them is non-inverted, we denote the predecessor by $S_i$, else if the edge is inverted, denote it by $\neg S_i$. If a predecessor is a PI or a constant, we represent it as $c_i$. With this notation, we can define the predecessor set for a node. Based on elements in the predecessor set, we can determine a mapping for the node. We should note that since majority function is commutative, the order of predecessors do not matter for determination of the mapping.

The mapping for all the predecessor sets, is shown in Table 4.1. The read operations are not shown for the sake of brevity in the table. It should be noted that irrespective of the predecessor set of a node $n$ at level $l$, only a single instruction is executed in clock cycle $l$ for mapping the node $n$. This is critical for achieving the delay optimal mapping.

**Example 4.2.** *For node $S_4$ in Figure 4.7, the predecessor set is $(S_1, \neg S_2, a)$, which is enumeration T4. We explain the mapping for this enumeration. Since the non-inverted predecessor $S_1$ is busy, constant $a$ is preloaded in a new device $d_{new}$ in cycle 1, as node $S_4$ is in level 2. In cycle 2, node $S_1$ and $S_2$ are read out and applied to the wordline and bitline of device $d_{new}$ to compute $S_4$.*

**Algorithm 2:** Procedures used for Delay Optimal Mapping

```
1  Procedure ProcessNode(node)
2      global G, nodeHeap, n2DMap;
3      p = GetPredecessorSet(node);
4      Schedule operation based on NodeMapRules(Table 1) ;
5      node.processed = True;
6      for pnode ∈ node.predecessors do
7          if node.processed == False then
8              nodeHeap.add (node);
9  Procedure PropagateDevice()
       /* Allocate specific device to node                                  */
10     for node ∈ n2DMap do
11         GetDevice(node);
12 Procedure GetDevice(node)
13     if n2DMap[node].allocated == True then
14         return n2DMap[node].device;
15     n2DMap[node].device = GetDevice(n2DMap[node].host);
16     n2DMap[node].allocated = True;
17     return n2DMap[node].device;
```

$$
\begin{array}{lll}
d^1_{new} & 1 & \textsc{Compute}\ 0, a, 0 \longrightarrow S \\
d^2_{new} & 2 & \textsc{Compute}\ S, S_1, S_2 \longrightarrow S_4 \\
d^k_1 & 2 & \textsc{Read} \longleftarrow S_1 \\
d^{k'}_2 & 2 & \textsc{Read} \longleftarrow S_2
\end{array}
$$

*The internal state $S$ of the device $d_{new}$ in cycle 1 is $M_3(0, a, \neg 0) = a$. In the next cycle 2, $S_4 = M_3(a, S_1, \neg S_2)$ is computed. $d^1_{new}$ and $d^2_{new}$ represents the first and second operation on the device $d_{new}$ respectively. $d^k_1$ and $d^{k'}_2$ represent $k$ and $k'$ operation on the device $d_1$ and $d_2$ respectively.*

The mapping phase is complete when the *nodeHeap* is empty. At the end of mapping phase, each internal node in the MIG is associated with a *host* node and there are multiple constant/variables nodes which are associated with a specific *device*, due to pre-loading. The hashmap $n2DMap$ stores the *host* corresponding to a node, the *device* onto which the node has been mapped and a Boolean flag *isAllocated* to indicate whether the node has already been mapped to a device. To determine the actual mapping of internal node to MIG, the procedure PropagateDevice is invoked, which uses

**Table 4.1:** NODEMAPRULES: Mapping rules for an internal node $S_f$, at level $l$, based on the predecessor set. $S_j^r$ refers to replica of node $S_j$. $\mathcal{B}$ represents busy 'non-inverted' predecessor. *Insert* indicates which predecessors are added to the $nodeHeap$ for processing. If the constants are inverted, then the instructions should use appropriate inversions for those inverted constants. For templates T2, T4, T5 and $\exists S_i \notin \mathcal{B}$, the table shows the mapping for instance when $S_0 \notin \mathcal{B}$. The mapping has to be appropriately changed if some other non-inverted input node is not busy. $d_{new}$ refers to a new device while $d_{S_i}$ represents the device with the internal state $S_i$.

| | $\forall S_i \in \mathcal{B}$ | $\exists S_i \notin \mathcal{B}$ |
|---|---|---|
| T1: $M_3(S_i, c_0, c_1)$ | $d_{new}$ $l-1$ COMPUTE $0, c_0, 0 \longrightarrow S$<br>$d_{new}$ $l$ COMPUTE $S, S_i, \neg c_1 \longrightarrow S_f$ | $d_{S_i}$ $l$ COMPUTE $S_i, c_0, \neg c_1 \longrightarrow S_f$ |
| | Insert $S_i$ | Insert $S_i$ |
| T2: $M_3(S_0, S_1, c_0)$ | $d_{new}$ $l-1$ COMPUTE $0, c_0, 0 \longrightarrow S$<br>$d_{new}$ $l$ COMPUTE $S, S_0, \neg S_1 \longrightarrow S_f$ | $d_{S_0}$ $l$ COMPUTE $S_0, S_1, \neg c_0 \longrightarrow S_f$ |
| | Insert $S_0$, $\neg S_1$ | Insert $S_0$, $S_1$ |
| T3: $M_3(S_0, S_1, S_2)$ | $d_{S_0^r}$ $l$ COMPUTE $S_0, S_1, \neg S_2 \longrightarrow S_f$ | $d_{S_0}$ $l$ COMPUTE $S_0, S_1, \neg S_2 \longrightarrow S_f$ |
| | Insert $S_1$, $\neg S_2$, $S_0^r$ | Insert $S_0$, $S_1$, $\neg S_2$ |
| T4: $M_3(S_0, \neg S_1, c_0)$ | $d_{new}$ $l-1$ COMPUTE $0, c_0, 0 \longrightarrow S$<br>$d_{new}$ $l$ COMPUTE $S, S_0, S_1 \longrightarrow S_f$ | $d_{S_0}$ $l$ COMPUTE $S_0, c_0, S_1 \longrightarrow S_f$ |
| | Insert $S_0$, $S_1$ | Insert $S_0$, $S_1$ |
| T5: $M_3(S_0, S_1, \neg S_2)$ | $d_{S_0^r}$ $l$ COMPUTE $S_0, S_1, S_2 \longrightarrow S_f$ | $d_{S_0}$ $l$ COMPUTE $S_0, S_1, S_2 \longrightarrow S_f$ |
| | Insert $S_0^r$, $S_1$, $S_2$ | Insert $S_0$, $S_1$, $S_2$ |
| T6: $M_3(S_0, \neg S_1, \neg S_2)$ | $d_{S_0^r}$ $l$ COMPUTE $S_0, \neg S_1, S_2 \longrightarrow S_f$<br>Insert $S_0^r$, $\neg S_1$, $S_2$ | $d_{S_0}$ $l$ COMPUTE $S_0, \neg S_1, S_2 \longrightarrow S_f$<br>Insert $S_0$, $\neg S_1$, $S_2$ |
| T7: $M_3(c_0, c_1, c_2)$ | $d_{new}$ $l$ COMPUTE $S, c_1, \neg c_2 \longrightarrow S_f$ | |
| | NA | |
| T8: $M_3(\neg S_0, c_0, c_1)$ | $d_{new}$ $l-1$ COMPUTE $0, c_0, 0 \longrightarrow S$<br>$d_{new}$ $l$ COMPUTE $S, c_1, S_0 \longrightarrow S_f$ | |
| | Insert $S_0$ | |
| T9: $M_3(\neg S_0, \neg S_1, c_0)$ | $d_{new}$ $l-1$ COMPUTE $0, c_0, 0 \longrightarrow S$<br>$d_{new}$ $l$ COMPUTE $S, \neg S_1, S_0 \longrightarrow S_f$ | |
| | Insert $S_0$, $\neg S_1$ | |
| T10: $M_3(\neg S_0, \neg S_1, \neg S_2)$ | $d_{\neg S_0^r}$ $l$ COMPUTE $\neg S_0^r, \neg S_1, S_2 \longrightarrow S$ | |
| | Insert $\neg S_0^r$, $\neg S_1$, $S_2$ | |

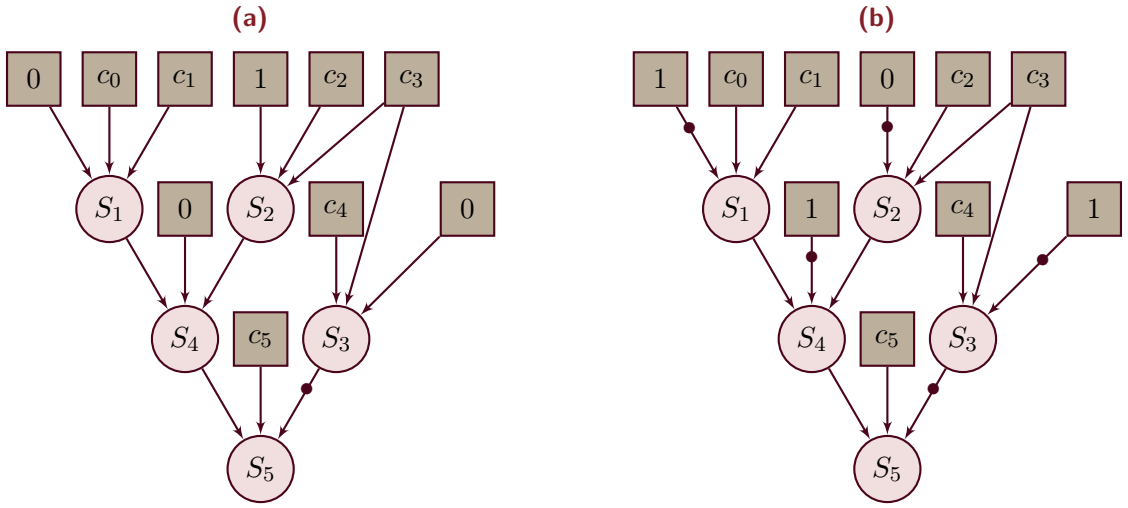
the recursive function GETDEVICE, for propagating the *host* to *device* mapping information.

**Example 4.3.** *Figure 4.8 shows a complete example of the delay optimal mapping. Figure 4.8a shows the input MIG, which is transformed into the Figure 4.8b MIG. This transformed MIG is mapped to in-memory compute operations with an optimal delay of 4 cycles, as shown in Figure 4.8c. The target function $F$ realized by the operations is*

$$F = M_3(M_3(M_3(c_0, c_1, \neg 1), M_3(c_3, c_4, \neg 1), \neg 1), c_6, \neg M_3(c_5, c_4, \neg 1)) \tag{4.5}$$

*This is logically equivalent to the function realized by the input MIG.*

**Theorem 4.1.** *A MIG with $l_{max}$ depth needs at least $l_{max}$ cycles to be evaluated using 1S1R devices.*




(c)

| DevOpId | t | Operation | State update |
|---|---|---|---|
| $d_2^1$ | 1 | COMPUTE $0, c_0, 0 \longrightarrow S_1^1$ | $S_1^1 = M_3(0, c_0, \neg 0) = c_0$ |
| $d_1^1$ | 1 | COMPUTE $0, c_2, 0 \longrightarrow S_2^1$ | $S_2^1 = M_3(0, c_2, \neg 0) = c_2$ |
| $d_3^1$ | 1 | COMPUTE $0, c_3, 0 \longrightarrow S_3^1$ | $S_3^1 = M_3(0, c_3, \neg 0) = c_3$ |
| $d_2^2$ | 2 | COMPUTE $S_1^1, c_2, 1 \longrightarrow S_1$ | $S_1 = M_3(S_1^1, c_2, \neg 1)$ |
| $d_1^2$ | 2 | COMPUTE $S_2^1, c_3, 0 \longrightarrow S_2$ | $S_2 = M_3(S_1^1, c_3, \neg 0)$ |
| $d_3^2$ | 2 | COMPUTE $S_3^1, c_5, 1 \longrightarrow S_3$ | $S_3 = M_3(S_3^1, c_5, \neg 1)$ |
| $d_2^3$ | 3 | COMPUTE $S_1, S_2, 1 \longrightarrow S_4$ | $S_4 = M_3(S_1, S_2, \neg 1)$ |
| $d_1^3$ | 3 | READ $\longleftarrow S_2$ | |
| $d_2^4$ | 4 | COMPUTE $S_4, f, S_3 \longrightarrow S_5$ | $S_5 = M_3(S_4, c_6, \neg S_3)$ |
| $d_3^3$ | 4 | READ $\longleftarrow S_3$ | |

**Figure 4.8:** An example for delay optimal mapping of an MIG. (a) Input MIG with 4 levels. (b) Transformed MIG. (c) The in-memory compute operations to map the transformed MIG in 4 cycles.

*Proof.* Let $p_{long}$ be the longest path from PI to PO in an MIG. The nodes in the same level do not have any data dependency amongst each other, and hence can be evaluated simultaneously. It is evident that the time required for evaluating the nodes in $p_{long}$ will determine the delay of evaluating the MIG, since the nodes in the adjacent levels have data dependency between each other, and hence have to evaluated sequentially. The number of nodes in $p_{long}$ is equal to the depth of the MIG, $l_{max}$. At least one cycle is needed to evaluate each node in MIG, since a 1S1R device requires one cycle to perform a majority operation. Therefore, perform the computation of each node in $p_{long}$, at least $l_{max}$ number of cycles are needed. This includes the first node in the path, which is loaded as the initial state in

the device, which acts as the third input to the majority, the other two inputs being provided by the wordline and bitline of the device. Thereby to evaluate an MIG with maximum level $l_{max}$, at least $l_{max}$ cycles are needed. ■

**Lemma 4.1.** *For a MIG with $l_{max}$ depth, Algorithm 1 generates a mapping with $l_{max}$ cycles delay.*

*Proof.* Each MIG is a DAG. In the proposed mapping algorithm, for each node at level $l$, only one instruction is needed to be executed at clock cycle $l$, as evident from Table 4.1. Therefore, the last set of nodes will be executed in clock cycle $l_{max}$. The internal nodes at level 2 always have the predecessor set $M_3(c_0, c_1, c_2)$, will be evaluated in the first two cycles 0 and 1. Thus, to process a MIG with $l_{max}$ levels, by evaluating the nodes *level*-wise, with the lower level nodes being processed first, we will require $l_{max}$ cycles, which is the minimum delay. ■

This result significantly improves upon to the delay obtained by the naïve technology mapping proposed in [96]. The work assumes that all nodes in a level can be evaluated in *three* cycles and *one* more cycle is needed to evaluate a level, if any ingoing edge to the level is complemented. From our algorithm, it is evident that such a mapping is inefficient, since with our approach, each node effectively requires a single cycle for evaluation.

### 4.3.3 Heuristic For Area Minimization

A ReRAM device can be reset to logic 0 in one cycle by applying '0' and '1' to the wordline and bitline of the device respectively. After the mapping algorithm has completed execution, we can determine the *sTime* and *eTime* of a device.

**Definition 4.8** (sTime)**.** *sTime or the start time of a device is the first cycle in which the device is being used.*

**Definition 4.9** (etime)**.** *eTime or the end time of a device is the last cycle in which the device is used.*

Formally, if there are two devices $d$ and $d'$ such that $d.eTime + 1 > d'.sTime$, then we can re-use the device $d$ for performing all the computations that are scheduled on device $d'$. The additional one cycle is needed to reset the device $d$, before reusing.

The Algorithm 3 enforces device reuse after the mapping of nodes to devices has been completed. $devUseTable$ is a table that has the following fields: device $d$, $sTime$, $eTime$ and a Boolean field $isAssigned$ to mark whether device $d$ has been considered for reuse. The table is sorted in non-decreasing order by the $sTime$ field. The hashmap $reassignMap$ stores the mapping about which new device $d'$ has been reallocated to device $D$. For a device $d$, the algorithm scans the $devUseTable$ to find the device $d'$ which satisfies the reuse criteria $d.eTime + 1 > d'.sTime$, and adds an entry into $reassignMap$ to store the reuse information.

**Algorithm 3:** Device Reuse Algorithm

```
1  Procedure DeviceReassign(devUseTable)
2      reassignMap = HashMap();
3      for d ∈ devUseTable do
4          freeTime = devUseTable[d].eTime + 1;
5          devUseTable[d].isAssigned = True;
6          for d′ ∈ devUseTable do
7              if devUseTable[d′].isAssigned == False && devUseTable[d′].sTime > freeTime then
8                  reassignMap[d′] = d;
9                  devUseTable[d′].isAssigned = True;
10                 freeTime = devUseTable[d′].eTime + 1;
11 return reassignMap;
```

## 4.4 Area-Constrained Technology Mapping

Given a set of in-memory computing operations, each operation needs to be executed on an available in-memory computing device. However, the number of devices available on a target platform to map the operations can be lesser than the number of devices required for mapping. This gives rise to the area-constrained technology mapping problem. In this section, we present the construction of state dependency graph (SDG) from a given schedule of operations, which will be used for performing

area-constrained technology mapping. Thereafter, we present an Integer Linear Programming (ILP) formulation for the area-constrained technology mapping problem. We also propose a heuristic for solving large instances of the problem.

**Definition 4.10** (elements)**.** *Individual devices available for mapping on a target platform are defined as* ***elements****.*

Multiple devices used in a set of operations can be mapped to the same element, as long as this mapping does not overlap in time.

**Definition 4.11** (Feasible schedule)**.** *Given a sequence of operations over n devices, a* ***feasible schedule*** *maps the devices to k ($k \leq n$) available elements so that the dependencies present among the operations defined over devices are preserved and no two devices are mapped to the same element simultaneously. The dependencies are as follows-*

1. *The four operations,* Compute, Read, Reset *and* Copy *are mutually exclusive. The four operations cannot be performed simultaneously on the device in any cycle.*

2. *A device cannot be reset until all the* Compute *and* Read *operations on that device are completed. No operation is permitted on a device following the* Reset *operation.*

3. *Given a* Compute *operation:*

$$d_x^i \quad t \quad Compute \quad C_1, C_2, \ldots, S_{x1}^{i1}, S_{x2}^{i2}, \ldots, S_{xk}^{ik} \rightarrow S_j^i$$

*The following* Read *operations must be present in step t, for $\forall i \mid 1 \leq i \leq k$.*

$$d_{xi}^i \quad t \quad \textsc{Read} \leftarrow S_{xi}^i$$

.

*4. Given two* COMPUTE *operations, such that* $i < i'$*:*

$$d_x^i \quad t \quad Compute \quad C_1, C_2, \ldots, S_{x1}^{i1}, S_{x2}^{i2}, \ldots, S_{xk}^{ik} \rightarrow S_x^i$$

$$d_x^{i'} \quad t' \quad Compute \quad C_1', C_2', \ldots, S_{x1'}^{i1'}, S_{x2'}^{i2'}, \ldots, S_{xk'}^{ik'} \rightarrow S_x^{i''}$$

*Then,* $i^{th}$ COMPUTE *operation on device* $x$ *must precede* $i'^{th}$ COMPUTE *operation on device* $x$*, i.e.,* $t < t'$*.*

**Definition 4.12** (Equivalent schedules)**.** *Given a schedule* $S$ *over* $n$ *devices for a target function* $F$ *and a schedule* $S'$ *over* $k$ *elements for a target function* $F'$*, then* $S$ *and* $S'$ *are termed as* ***equivalent schedules*** *if* $F$ *is logically equivalent to* $F'$*.*

A State Dependency Graph (SDG) is used to capture the dependencies among the states of devices, constrained by the technology. Each device has a restriction on the number of inputs, specified by the technology. For example, 1S1R devices have three inputs — the internal state, wordline and bitline.

**Definition 4.13** (SDG)**.** *A* ***SDG*** *is a DAG with bounded indegree* $G_B = \langle V, E \rangle$*, where* $B$ *is the maximum number of inputs specified by a technology. Node* $S_i^j \in V$ *represents the state of the* $i^{th}$ *device after completion of the* $j^{th}$ *operation. Edges can be one of three types:* COMPUTE*,* READ *or* COPY*. A directed edge* $S_x^y \rightarrow S_i^j$ *implies that the* $j^{th}$ *operation on device* $i$ *can only be executed after completion of the* $y^{th}$ *operation on device* $x$*.*

Each device has an initial state or the start state and a final known state or the reset state. Edges are of three types corresponding to the possible operations on the device.

- COMPUTE EDGE - An edge $(S_i^j, S_i^{j+1})$ is a COMPUTE EDGE if the $(j+1)^{th}$ operation on device $i$ is a COMPUTE operation. A COMPUTE operation on device $i$ updates the state of device $i$ from $S_i^j$ to $S_i^{j+1}$.

- READ EDGE - An edge $(S_x^y, S_i^j)$ where $x \neq i$, is termed as a READ EDGE if the $j^{th}$ operation on device $i$ is a COMPUTE operation with $S_x^y$ as one of the inputs.
- COPY EDGE - An edge $(S_y^k, S_x^i)$ where $y \neq x$, is termed as a COPY EDGE if the $i^{th}$ operation on device $x$ copies the state of device $y$ after completion of $k^{th}$ operation.

Each of these operations are performed obeying the dependencies among different devices in a SDG. A node in a SDG cannot be executed until all its predecessor nodes have completed their execution. Since all the devices and elements are identical in their functions, we can allocate a device to any available element. Hence, it is a many-to-one mapping with the constraint that we do not map two devices to the same element in a step. Each state update is associated with a specific device operation. Therefore, the nodes in the SDG can be alternatively marked using the device operation identifiers, instead of device states.

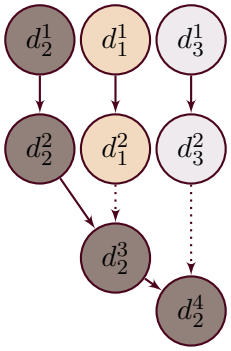
**Example 4.4.** *For the operations in Figure 4.8c, the corresponding SDG is shown in Figure 4.9a. We consider the operation $d_2^3$,i.e., the operation 3 on device 2. The operation updates it's own state, thus there is a* COMPUTE *edge from $d_2^2$ to $d_2^3$. Also, the operation uses $S_2$ as one of it wordline input. Thus, there is a read dependency from operation $d_1^2$ (that created the state $S_2$) to $d_2^3$. Similarly, the rest of the edges are added for constructing the SDG. We should not that the read operations $(d_1^3, d_3^3)$ do not appear in the SDG as nodes, since read operations do not change state of the device on which the read was performed.*

**Definition 4.14** (Feasible SDG schedule)**.** *A feasible schedule for a SDG with $n$ devices and $k$ elements over a sequence of $\tau$ steps is defined over $a_{ij}$, where $1 \leq i \leq n$, $1 \leq j \leq k$, governed by the following rules.*

1. *The assignment of device $i$ to element $j$ is represented by the variable $a_{ij}$.*

$$a_{ij} = \begin{cases} t, \textit{if device } d_i \textit{ is assigned to element } e_j \textit{ at step } t,\ t > 0 \\ 0, \textit{otherwise} \end{cases}$$

(a)




(b)

| DevOpId | Mapping | t | Operation | State update |
|---|---|---|---|---|
| $d_2^1$ | $d_2 \rightarrow e_1$ | 1 | COMPUTE $0, c_0, 0 \longrightarrow S_1^1$ | $E_1^1 = M_3(0, c_0, \neg 0) = c_0$ |
| $d_1^1$ | $d_1 \rightarrow e_2$ | 1 | COMPUTE $0, c_2, 0 \longrightarrow S_2^1$ | $E_2^1 = M_3(0, c_2, \neg 0) = c_2$ |
| $d_2^2$ | $d_2 \rightarrow e_1$ | 2 | COMPUTE $S_1^1, c_2, 1 \longrightarrow S_1$ | $E_1^2 = M_3(E_1^1, c_2, \neg 1)$ |
| $d_1^2$ | $d_1 \rightarrow e_2$ | 2 | COMPUTE $S_2^1, c_3, 0 \longrightarrow S_2$ | $E_2^2 = M_3(E_2^1, c_3, \neg 0)$ |
| $d_2^3$ | $d_2 \rightarrow e_1$ | 3 | COMPUTE $S_1, S_2, 1 \longrightarrow S_4$ | $E_1^3 = M_3(S_1, S_2, \neg 1)$ |
| $d_1^3$ | $d_1 \rightarrow e_2$ | 3 | READ $\longleftarrow E_2^2$ | |
| | | 4 | RESET $e_1$ | $E_2^3 = M_3(E_2^2, 0, \neg 1) = 0$ |
| $d_3^1$ | $d_3 \rightarrow e_2$ | 5 | COMPUTE $0, c_3, 0 \longrightarrow S_3^1$ | $E_2^4 = M_3(0, c_3, \neg 0) = c_3$ |
| $d_3^2$ | $d_3 \rightarrow e_2$ | 6 | COMPUTE $E_2^4, c_5, 1 \longrightarrow S_3$ | $E_2^5 = M_3(S_3^1, c_5, \neg 1)$ |
| $d_2^4$ | $d_2 \rightarrow e_1$ | 7 | COMPUTE $S_4, f, S_3 \longrightarrow E_1^4$ | $S_5 = M_3(E_1^3, c_6, \neg E_2^5)$ |
| $d_3^3$ | $d_3 \rightarrow e_2$ | 7 | READ $\longleftarrow E_2^5$ | |

**Figure 4.9:** An example for area-constrained mapping for ReVAMP. (a) SDG corresponding to operations shown in Figure 4.8c (b) A feasible schedule for this SDG using two available elements, $e_1$ and $e_2$.

2. *At any time step $t$, the number of available elements for allocation cannot be greater than $k$.*

$$\sum_{i=1}^{n} \sum_{j=1}^{k} a_{ij} \leq k * t$$

3. *At any step $t$, an element can be assigned to only one device. For every $j$, $\sum_{i=1}^{n} a_{ij} \leq t$.*

4. *Other than the above rules, a feasible SDG schedule must obey the dependencies specified in Definition 4.11.*

Each operation takes a unit step to compute, which implies every transition in the SDG takes a unit step. Multiple transitions in a SDG can be executed in parallel if there are sufficient number of available elements to allocate the corresponding devices and the dependencies among different devices are not violated.

**Definition 4.15** (Makespan)**.** *Makespan of a SDG schedule is equal to the maximum completion time of any operation in the SDG.*

It should be noted that if the last operation scheduled on an element is a reset operation, it does not meaningful w.r.t to computation of the logic function being mapped. Therefore, trailing reset operations are omitted from the final schedule.

**Problem statement :** *Given a schedule $\mathcal{S}$ over $n$ devices for computing a given target function $F$, whether there is a feasible schedule $\mathcal{S}'$ over $k$ elements for computing a logically equivalent function $F'$ so that the dependencies (stated in Definition 4.11.) present among the devices are not violated and no two devices are mapped to the same element in the same step.*

In the following subsections, we present an ILP formulation and heuristic for finding a feasible schedule for a SDG with $n$-devices, using $k$-elements. If $k \geq n$, the schedule is trivial, as each device can be mapped to a unique element. It is also possible that scheduling a SDG with $k$-elements is not feasible — in that case, scheduling has to be tried with a greater number of elements.

### 4.4.1 ILP Formulation For Area-Constrained Technology Mapping

Let $n$ be the number of devices present in SDG and $k$ be the number of ReRAM elements available for mapping. The set of variables that will be used for the ILP formulation is listed in Table 4.2.

**Table 4.2:** Variables used in ILP formulation.

| Variable | Type | Description |
|---|---|---|
| $y$ | int | Makespan of schedule |
| $x_{ie}$ | bin | 1, if device $i$ is mapped to element $e$ |
| $t^c_{ij}$ | int | Completion time of operation $j$ of device $i$ |
| $t^C_i$ | int | Completion time of device $i$ |
| $t^S_i$ | int | Start time of device $i$ |
| $u_{ie}$ | int | $x_{ie}.t^C_i$ |
| $v_{ie}$ | int | $x_{ie}.t^S_i$ |
| $w_{ije1}$ | bin | 0, if device $i$ is scheduled after device $j$ on element $k$. |
| $w_{ije2}$ | bin | 0, if device $j$ is scheduled after device $i$ on element $k$. |
| $w_{ije3}$ | bin | 0, if device $i$ and $j$ are scheduled on two different elements. |
| $g_{ie}$ | int | Used to linearize $x_{ie}.t^C_i$ |
| $h_{ie}$ | int | Used to linearize $x_{ie}.t^S_i$ |

**Objective function:** Minimize $y$.

**Makespan constraints:**

$$t_i^C - y \leq 0,\ 1 \leq i \leq n \tag{4.6}$$

The makespan of the schedule is equal to the maximum completion time of any element used.

**Completion and start time constraints:**

$$t_{i,1}^C \geq 1 \tag{4.7}$$

$$t_i^S = t_{i,1}^C - 1 \tag{4.8}$$

$$t_i^C = t_{i,j_{max}}^C \tag{4.9}$$

Each operation needs one cycle to complete. Therefore, the completion time of the first operation $t_{i,1}^C$ on any device $i$ will be at least 1. Also, the start time $t_i^S$ of a device $i$ is equal to one less than completion time $t_{i,1}^C$ of the first operation on that device. The completion time of a device is equal to the completion time of the last operation scheduled on that device.

For COMPUTE edge $S_i^j \rightarrow S_i^{j+1}$,

$$t_{i,j+1}^C \geq t_{i,j}^C + 1 \tag{4.10}$$

For READ edges $S_i^j \rightarrow S_{i'}^{j'}$ and $i \neq i'$,

$$t_{i',j'}^C \geq t_{i,j}^C + 1 \tag{4.11}$$

$$t_{i,j+1}^C \geq t_{i',j'}^C + 1 \tag{4.12}$$

The constraint (4.12) ensures that the result of operation $j$ on device $i$ is held till the read by operation $j'$ of device $i'$. It also ensures that if there are no more operations after $j$ on device $i$, the extra 1 cycle can be used for resetting the element to which device $i$ was mapped and thereafter it can be reused.

**Scheduling dependency constraints:** For element $e$, $e \in \{1, .., k\}$ and a pair of devices $(i, j)$, $i \in \{1, .., n\}$,

$j \in \{i+1, ..., n\}$,

$$M.w_{ije1} + v_{ie} - u_{je} > 0 \tag{4.13}$$

$$M.w_{ije2} + v_{je} - u_{ie} > 0 \tag{4.14}$$

$$M.w_{ije3} + 2 - x_{ie} - x_{je} > 0 \tag{4.15}$$

$$w_{ije1} + w_{ije2} + w_{ije3} = 2 \tag{4.16}$$

$$u_{ie} = x_{ie}.t_i^C \tag{4.17}$$

$$v_{ie} = x_{ie}.t_i^S \tag{4.18}$$

$M$ is any integer greater than twice the number of operations in the SDG. These constraints ensure that if device $i$ and $j$ are mapped to the same element $e$, then either the start time of device $i$ is greater than the completion time of device $j$ or the start time of device $j$ is greater than the completion time of operations on device $i$. Otherwise, the devices $i$ and $j$ are mapped to two different elements.

**Linearize constraints:** Two variables namely $u_{ie}$ and $v_{ie}$ are the product of a binary indicator variable and an integer variable. These can be expressed as linear constraints by using linear and SOS1 constraints. SOS1 is used to refer to Special Ordered Sets of type 1, i.e., at most one member of the set can take a strictly positive value, all others being 0.

$$u_{ie} + g_{ie} + t_i^C = 0 \tag{4.19}$$

$$g_{ie} + x_{ie} >= 1 \tag{4.20}$$

$$\text{SOS1 } \{g_{ie}, x_{ie}\} \tag{4.21}$$

$$v_{ie} + h_{ie} + t_i^S = 0 \tag{4.22}$$

$$h_{ie} + x_{ie} >= 1 \tag{4.23}$$

$$\text{SOS1 } \{h_{ie}, x_{ie}\} \tag{4.24}$$

This concludes the formulation of the ILP for area-constrained technology mapping of a SDG with $n$-devices to $k$-elements.

**Example 4.5.** *Figure 4.9b presents a feasible schedule for the SDG on three devices $d_1$, $d_2$ and $d_3$ using two available elements, $e_1$ and $e_2$ using the ILP. We refer to the states of the elements using the $E_i^j$ notation, that denotes $j^{th}$ state of device $i$. In the first cycle, device $d_1$ and $d_2$ are mapped to the elements $e_1$ and $e_2$ respectively. Operations on device $d_3$ cannot begin due to the lack of available elements. Once all the operations using $d_2$ are complete, the element $e_2$ is reset. Thereafter, $d_3$ is mapped to element $e_2$ for the rest of the operations to be performed. The target function $F'$ realized by area-constrained mapping is*

$$F' = M_3(M_3(M_3(c_0, c_1, \neg 1), M_3(c_3, c_4, \neg 1), \neg 1), c_6, \neg M_3(c_5, c_4, \neg 1)) \tag{4.25}$$

*As expected, this target function $F'$ is logically equivalent to the target function $F$ [in equation (4.5)] realized by the delay optimal solution. Thus, the schedule obtained by area-constrained mapping and the delay optimal schedule are equivalent schedules.*

The ILP does not scale for large problem instances. For small problem instances, the technology mapping for SDGs with less than 40 nodes takes longer than an hour to solve using an ILP solver. Therefore, we propose a scalable and fast heuristic algorithm for solving the same problem, presented in the next subsection.

### 4.4.2 Heuristics For Area-Constrained Technology Mapping

We propose a heuristic for area-constrained technology mapping. We begin by defining a few terms w.r.t a node in a SDG — Earliest Start Time (EST), Latest Start Time (LST) and *readout*.

**Definition 4.16** (EST)**.** *Earliest Start Time (EST) of a node in SDG without any incoming edge is 1. For other nodes, EST is equal to the maximum EST among it's predecessors plus one.*

**Definition 4.17** (LST)**.** *Latest Start Time (LST) for nodes without successors is equal to the maximum EST. For other nodes, LST is the minimum EST among its successors minus one.*

**Definition 4.18** (Readout)**.** *Readout of a device in a SDG is equal to the number of outgoing read edges that the device has.*

The EST and LST of a node in SDG are determined by the position of the node in the SDG. Nodes with a smaller EST and LST can be started earlier than one with a higher EST and LST. A device with a high *readout* implies that the result of operation on that device has to be held until the last operation which has the read dependency has been allocated an element and finished execution.

**Example 4.6.** *For the SDG in Figure 4.9a, the EST/LST of the nodes in SDG are shown below. Devices $d_2$ and $d_3$ have readout equal to 1 while device $d_1$ has readout 0.*

| Node | EST | LST |
|---|---|---|
| $d_1^1$ | 1 | 1 |
| $d_1^2$ | 2 | 2 |
| $d_2^1$ | 1 | 1 |
| $d_2^2$ | 2 | 2 |
| $d_2^3$ | 3 | 3 |
| $d_2^4$ | 4 | 4 |
| $d_3^1$ | 1 | 2 |
| $d_3^2$ | 2 | 3 |

The area-constrained heuristic mapping algorithm is presented in Algorithm 2. At the beginning, all the operations are in *unprocessedQ* priority queue with priority determined by *EST, LST, readout*, with *EST* considered first, followed by *LST* and *readout* — smaller value implies a higher priority. This is due to the fact that nodes with lower EST will have lesser dependencies and hence can be processed earlier. Once a node has been processed, the successors of the node are processed first. Otherwise, devices that are allocated elements, might be wide apart in the SDG, which would lead to failure in allocation, due to the need for holding result of operations on multiple elements simultaneously, in order to satisfy read dependencies of the successors.

A node can be processed only after all its predecessors have been processed. This is because unless the read dependencies of an operation have been completed, the next operation cannot proceed. The

device to which the operation belongs, is mapped to a valid element. getValidElement function returns the element which is not being assigned to any other device, and which has the largest time elapsed since last use. Once a valid element has been found, the operation is scheduled on that element, such that all constraints of the operation are met. Thereafter, the successors of this node are added to the successor queue. The priority of the successor queue is determined by *readout, EST, LST*, with *readout* considered first, followed by *EST* and *LST* — smaller value implies a higher priority.

**Algorithm 4:** Heuristic based area-constrained technology mapping

```
1  Procedure ProcessNode(op)
2      global SDG, lastUse, schedule;
3      for p in op.predecessors do
4          if p not processed then
5              ProcessNode(p);
6          for s in p.successors and s != op do
7              if s not processed then
8                  ProcessNode(s);
9      e = getValidElement();
10     if e == None then
11         Allocation failed; Exit ;
12     lastUse[e] = lastUse[e] + 1 ;
13     schedule[op] = e, lastUse[e] ;
14     for p in op.predecessors do
15         if schedule[p].element != e then
16             if lastUse[schedule[p].element] < lastUse[e] then
17                 lastUse[schedule[p].element] = lastUse[e];
18     Add op.successors to succQ;
19     Remove op.successors from unprocessedQ;
20 Procedure MapSDG(SDG)
21     global SDG, lastUse, schedule;
       /* priority = (EST, LST, readout)                                  */
22     unprocessedQ = prioQ();
       /* priority = (readout, EST, LST)                                  */
23     succQ = prioQ();
24     while True do
25         if succQ != empty then
26             processNode(succQ.pop()) ;
27         else if unprocessedQ != empty then
28             processNode(unprocessedQ.pop());
29         else
30             break;
```

## 4.5 Experimental Results

The proposed optimal mapping algorithm was implemented in Python3, executed on a system running Ubuntu 14.04 with 16 cores and 64 Gb RAM. The technology mapping solutions have been evaluated using the EPFL benchmarks*.

### 4.5.1 Results Of Delay Optimal Technology Mapping

The results of benchmarking the delay optimal technology mapping algorithm is shown in Table 4.3. The execution time for mapping for most of the MIGs took under 10 minutes. For larger MIGs, the execution time ran into a few hours, due to access to large MIG data structures. As expected, the delay (#C) of the mapping is equal to number of levels (#L) in the MIG plus. Due to the node replication during MIG transformation to achieve optimal mapping, the transformed MIG has greater number of nodes than the original MIG. The number of devices used in the delay optimal mapping solution is significantly reduced, when the device reduction heuristic is applied. For this set of benchmarks, we achieve savings of 56% in number of devices used, with maximum savings of 92.63% in case of benchmark $B_{13}$.

**Impact of MIG heuristics:** Given the algorithms implemented in [96] is complementary to our technology mapping flow, we experimented with the heuristics introduced there. For the MIGs $U_1$ and $U_2$, we employed the depth reduction heuristics in [96] to obtain the MIGs in $D_1$ and $D_2$. The resulting MIGs have considerably smaller depth, which results in improvement in delay of the mapping solution. However, the depth optimized MIGs have a larger node count and more transformations, which results in mapping with higher number of devices being used, indicating the necessity of further research in this direction.

*http://lsi.epfl.ch/benchmark

**Table 4.3:** Benchmarking results of delay optimal technology mapping. #L: number of levels in MIG, #PI/#PO: number of primary inputs/ outputs, #N: Number of nodes in MIG, #Nr: Increase in the number of MIG nodes after transformation, Inc.%: Increase in node count (in %), #D: Number of devices in mapping, #D_r: Number of devices after implementing device reuse, Sav.%: Device count reduction (in %), #Ins.: Number of instructions, #C: Delay of mapping

| | Benchmark | #L | #PI#PO | #N | #Nr | Inc.% | #D | #D_r | sav.% | #Ins. | #C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $B_1$ | MAC32 | 42 | 96/65 | 9391 | 14347 | 52.77 | 10631 | 3488 | 67.19 | 29214 | 42 |
| $B_2$ | MUL32 | 37 | 64/64 | 9160 | 12981 | 41.71 | 8929 | 2968 | 66.76 | 25365 | 37 |
| $B_3$ | ac97_ctrl | 9 | 2255/ 2250 | 12995 | 15932 | 22.60 | 11516 | 9080 | 21.15 | 27127 | 9 |
| $B_4$ | comp | 78 | 279/193 | 18686 | 25710 | 37.00 | 15233 | 5042 | 66.90 | 47938 | 78 |
| $B_5$ | des_area | 23 | 368/72 | 4258 | 5930 | 39.27 | 3825 | 1913 | 49.99 | 10686 | 23 |
| $B_6$ | div16 | 103 | 32/32 | 4406 | 6798 | 54.29 | 3776 | 440 | 88.34 | 12909 | 103 |
| $B_7$ | hamming | 62 | 200/7 | 2078 | 3089 | 48.65 | 2117 | 569 | 73.12 | 6178 | 62 |
| $B_8$ | i2c | 9 | 147/142 | 1113 | 1323 | 18.87 | 850 | 661 | 22.24 | 2200 | 9 |
| $B_9$ | max | 30 | 512/130 | 4340 | 6604 | 52.17 | 3851 | 1422 | 63.07 | 11670 | 30 |
| $B_{10}$ | mem_ctrl | 20 | 1198/1225 | 8368 | 10588 | 26.53 | 7148 | 2959 | 58.60 | 20141 | 20 |
| $B_{11}$ | pci_bridge32 | 17 | 3519/3528 | 22131 | 27368 | 23.66 | 17242 | 76.48 | 55.64 | 49307 | 17 |
| $B_{12}$ | pci_spoci_ctrl | 12 | 85/76 | 1008 | 1280 | 26.98 | 757 | 477 | 36.99 | 2127 | 12 |
| $B_{13}$ | revx | 144 | 20/25 | 7541 | 11434 | 51.62 | 7697 | 567 | 92.63 | 24388 | 144 |
| $B_{14}$ | sasc | 7 | 133/132 | 753 | 941 | 24.97 | 669 | 501 | 25.11 | 1605 | 7 |
| $B_{15}$ | simple_spi | 9 | 148/147 | 984 | 1220 | 23.98 | 813 | 531 | 34.69 | 2109 | 9 |
| $B_{16}$ | spi | 20 | 274/276 | 3613 | 4865 | 34.65 | 3242 | 1261 | 61.10 | 9188 | 20 |
| $B_{17}$ | sqrt32 | 165 | 32/16 | 2172 | 3627 | 66.99 | 2163 | 231 | 89.32 | 7147 | 165 |
| $B_{18}$ | square | 41 | 64/127 | 18014 | 27311 | 51.61 | 19514 | 7332 | 62.43 | 52767 | 41 |
| $B_{19}$ | ss_pcm | 7 | 106/98 | 495 | 553 | 11.71 | 368 | 305 | 17.11 | 926 | 7 |
| $B_{20}$ | systemcaes | 26 | 930/819 | 10366 | 13244 | 27.76 | 9100 | 2983 | 67.22 | 25970 | 26 |
| $B_{21}$ | systemcdes | 20 | 314/258 | 2711 | 3652 | 34.71 | 2557 | 917 | 64.13 | 7321 | 20 |
| $B_{22}$ | tv80aig | 31 | 373/404 | 7801 | 10287 | 31.86 | 6702 | 2181 | 67.46 | 19853 | 31 |
| $B_{23}$ | usb_funct | 20 | 1860/1846 | 14841 | 17689 | 19.19 | 11763 | 6207 | 47.23 | 32819 | 20 |
| $B_{24}$ | usb_phy | 8 | 113/111 | 483 | 537 | 11.18 | 337 | 289 | 14.24 | 883 | 8 |
| $U_1$ | adder | 256 | 256/129 | 1405 | 1406 | 0.07 | 1020 | 389 | 61.86 | 2801 | 256 |
| $D_1$ | adder.dep | 22 | 256/129 | 2647 | 4440 | 67.73 | 2387 | 1639 | 31.34 | 7319 | 22 |
| $U_2$ | sin | 225 | 24/25 | 5459 | 6124 | 12.18 | 4259 | 712 | 83.28 | 13906 | 225 |
| $D_2$ | sin.dep | 122 | 24/25 | 7558 | 10077 | 33.32 | 6895 | 1480 | 78.53 | 22363 | 122 |

### 4.5.2 Results Of Area-constrained Technology Mapping

For the ILP solver, we used Gurobi [121]. We set the TIME_LIMIT parameter of Gurobi to 3600 seconds, to find the optimal solution. As base case, we assume the number of elements is equal to the number of devices #D obtained by delay optimal mapping of the benchmark MIGs. We compare the improvement of area-constrained mapping over the base case in two ways. First, we determine the minimum of elements $E^f$ required to obtain a feasible schedule. Secondly, we determine the minimum number of elements $E^o$ required to achieve optimal delay $delay^o$. To obtain $E^f$, the SDG is scheduled

**Table 4.4:** Results of area-constrained technology mapping using ILP on small benchmarks.

| ID | Benchmark | #N | #D | $E_i^f$ | $delay_i^f$ | $E_h^f$ | $delay_h^f$ | $E_i^o$ | $E_h^o$ | $delay^o$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $B_1$ | Comparator | 19 | 7 | 4 | 12 | 4 | 13 | 7 | 7 | 5 |
| $B_2$ | Identity.dep | 23 | 8 | 4 | 13 | 4 | 13 | 8 | 8 | 5 |
| $B_3$ | Identity.skew | 23 | 8 | 3 | 18 | 3 | 18 | 8 | 8 | 6 |
| $B_4$ | Mux | 15 | 4 | 3 | 10 | 3 | 10 | 4 | 4 | 5 |
| $B_5$ | PC adder | 20 | 8 | 3 | 12 | 3 | 15 | 7 | 8 | 6 |
| $B_6$ | Prio Encoder | 22 | 7 | 2 | 18 | 2 | 18 | 6 | 7 | 6 |
| $B_7$ | Random.1 | 35 | 16 | – | – | 6 | 24 | – | 13 | 8 |
| $B_8$ | Random.2 | 31 | 18 | – | – | 6 | 17 | – | 7 | 7 |
| $B_9$ | Random.3 | 31 | 14 | 3 | 23 | 4 | 22 | 7 | 9 | 8 |
| $B_{10}$ | Random.4 | 37 | 14 | 3 | 30 | 4 | 21 | 7 | 9 | 12 |

**Table 4.5:** Results of area-constrained mapping using heuristics on large benchmarks.

| ID | Benchmark | #N | #D | $E_h^f$ | $delay_h^f$ | $R^f\%$ | $E_h^o$ | $delay^o$ | $R^o\%$ |
|---|---|---|---|---|---|---|---|---|---|
| $L_1$ | des_area | 8693 | 3789 | 868 | 86 | 77.09 | 1768 | 25 | 53.33 |
| $L_2$ | div16 | 9538 | 3732 | 325 | 191 | 91.29 | 477 | 122 | 87.21 |
| $L_3$ | hamming | 4606 | 2102 | 179 | 161 | 91.48 | 523 | 63 | 75.11 |
| $L_4$ | i2c | 2011 | 850 | 81 | 165 | 90.47 | 748 | 10 | 12.00 |
| $L_5$ | max | 9033 | 3737 | 872 | 111 | 76.66 | 1755 | 31 | 53.03 |
| $L_6$ | pci_spoci_ctrl | 1853 | 746 | 301 | 38 | 59.65 | 655 | 14 | 12.19 |
| $L_7$ | sasc | 1853 | 746 | 68 | 117 | 90.88 | 645 | 8 | 12.19 |
| $L_8$ | simple_spi | 1827 | 813 | 134 | 49 | 83.51 | 636 | 10 | 21.77 |
| $L_9$ | spi | 7202 | 3234 | 712 | 96 | 77.98 | 1386 | 24 | 57.14 |
| $L_{10}$ | ss_pcm | 865 | 367 | 88 | 113 | 76.02 | 321 | 8 | 12.53 |
| $L_{11}$ | systemcdes | 5673 | 2553 | 494 | 102 | 80.64 | 1052 | 23 | 58.77 |
| $L_{12}$ | usb_phy | 836 | 338 | 31 | 96 | 90.82 | 267 | 9 | 21.00 |
| $L_{13}$ | MUL32 | 19341 | 8898 | 707 | 212 | 70.01 | 5041 | 41 | 43.34 |
| $L_{14}$ | MEM_CTRL | 15955 | 7148 | 1179 | 62 | 83.50 | 4167 | 22 | 41.70 |

#N, #D: Number of nodes and devices in SDG respectively.
$E_i^f$, $E_h^f$: Minimum number of elements for feasible solution using ILP and heuristics respectively.
$E_i^o$, $E_h^o$ : Minimum number of elements to achieve optimal delay $delay^o$ using ILP and heuristics respectively.
$delay^o = max(EST)$ : Number of cycles of optimal solution.
$delay_i^f$, $delay_h^f$: Number of cycles in feasible schedule with minimum number of elements using ILP and heuristics respectively. $R^o\% = \frac{(\#D - E_h^o)}{\#D} * 100$. $R^f\% = \frac{(\#D - E_h^f)}{\#D} * 100$.

with #D elements and the number of elements is decremented till no feasible solution was found, using binary search. Similarly, $E^o$ is determined by decrementing number of elements from #D, till the minimum number of elements that can achieve $delay^o$ is obtained.

**Evaluation for heuristic solution vs ILP based optimal solution** ($B_1$-$B_{10}$): We compared the performance of the heuristics based solution against the optimal solution obtained by ILP on small benchmarks. For the benchmarks $B_1$-$B_6$, the heuristic based solution performs as good as the optimal solution, except $B_1$ and $B_5$ where $delay_h^f > delay_m^h$ for the same number of elements $E^f$. Due to small size of MIGs, reuse of elements is not possible in these cases and hence #D and $E_m^o$ are same.

We then tested the approaches on slightly larger MIGs - $B_7$-$B_{10}$, each with about 30 nodes in SDG. The ILP solver could not find the optimal schedule within the time limit for benchmarks $B_7$ and $B_8$. The heuristic on the other hand took a few seconds to find the schedule for all the four benchmarks. Since the ILP does not scale for mapping large benchmarks, we only evaluate the heuristic solution for large benchmarks $L_1$-$L_{14}$ and observe the improvement over base case, where each device is scheduled to an individual element.

**Delay and device usage for large cases** ($L_1$-$L_{14}$): The results are presented in Table 4.5. The minimum number of elements required for feasible schedule $E_h^f$ is on average reduced by 81.42% (max 91.48%) of the number of elements in base case. Even for achieving optimal makespan of $delay^o$, the heuristic obtained a reduction of 40.08% (max 87.21%) in the number of elements compared to the base case. This highlights the fact that a lot of reuse of elements is feasible.

By using the delay optimal mapping algorithm, only a single technology mapping solution could be obtained with #D elements. By using the area-constrained delay mapping, the entire range of elements between $E_h^f$ and $E_h^o$ can be used for mapping a design successfully. For example, the range of elements from 3234 till 7202 can be used for mapping the benchmark $L_9$. Using elements more than $E_h^o$ would not help in reducing delay further, since optimal delay of $delay^o$ cycles is achieved by using only $E_h^o$ elements.

## 4.6 Discussion

In this chapter, we introduced a computation model for in-memory computing. We defined the technology mapping problem for in-memory computing and also the corresponding area-constrained variant. We presented a delay optimal technology mapping algorithm for mapping MIG to ReRAM devices and tested our proposal on a set of benchmarks. The proposed algorithm maps any MIG with $k$ levels into ReRAM devices using $k$ steps. Further, we proposed an effective device reuse algorithm, that helped in reducing number of devices used by up to 92.63%, compared to the basic solution.

In addition, we formulated the problem of area-constrained technology mapping of MIGs, in terms of scheduling a State Dependency Graph (SDG) using a fixed number of elements. The SDG can be easily adapted for area constrained mapping for other logic representation structures and other in-memory computing architectures. We presented an ILP formulation for area-constrained technology mapping that achieves minimum makespan. But since the ILP formulation does not scale for scheduling large SDGs, we propose a delay minimizing mapping heuristics based on Earliest Start Time (EST), Latest Start Time (LST) and *readout* of operations in the SDG. The experimental results show that considerable reduction in number of devices can be achieved for large SDGs, without compromising on the optimal achievable delay $F^o$. We also observe that there is a difference in the minimum number of devices required for a feasible schedule and the number of devices required to achieve optimal delay. This implies that any number of devices in this range can be used effectively for scheduling a SDG.

The current chapter considered the technology mapping problem for ReRAM devices. To enable high density of fabrication, ReRAM devices are organized in crossbar configuration. Therefore, the ReVAMP architecture that was introduced in the previous chapter, uses crossbar arrays for computation. Crossbar configuration imposes additional constraints on the technology mapping flow. For example, only the devices that share a common wordline can be readout in a given cycle. In the next chapter, we address the problem of crossbar-constrained technology mapping by using ReVAMP as a target architecture.

# 5 Technology Mapping For The ReVAMP Architecture

A ReRAM crossbar array consists of multiple ReRAM devices that share wordlines and bitlines. A word consists of the devices that share a common wordline. All the bits present in a word can be operated upon or read out in a single cycle. The key challenge is to leverage the available bit-level parallelism to speed up logic-in-memory operations. The ReVAMP architecture permits harnessing this bit-level parallelism by means of *Apply* instruction. This chapter introduces the variants of crossbar-constrained technology mapping by using ReVAMP as the target logic-in-memory architecture. We present a multi-phase solution approach for each problem variant, starting from a structural representation of the input Boolean function.

Some works exist in literature that consider design automation flow for logic-in-memory using crossbar arrays. For a ReRAM crossbar array with devices realizing Majority with a single negated input, a serial approach to computing was proposed in [96], based on an architecture that takes 9 cycles to execute one logic-in-memory operation [82]. This approach seriously undermines the capabilities of ReRAM crossbar arrays that offer inherent bit-level parallelism. For crossbar capable of realizing

multiple NOR operations in parallel, a multi-stage design automation flow was proposed, using the `ABC` tool and an optimization solver backend for technology mapping [80]. For the same platform, a scalable heuristic was presented that uses simulated annealing for alignment of operations in the crossbar memory to reduce the delay of mapping [81]. Xie et al. proposed a mapping flow that partitions the input function into smaller functions, which are implemented using *computing elements* that are mapped onto a crossbar array with a dedicated controller synthesized for the specific input function mapped [78].

## 5.1 Formal Definition Of The Crossbar-constrained Technology Mapping Problem

In this section, we present two variants of the technology mapping problem for the ReVAMP architecture, along with overview of the proposed solutions.

### 5.1.1 Problem Definition

*Area constrained technology mapping :* Given a Boolean function represented as a Boolean logic network $G$ and crossbar dimension $S_D \times w_D$, determine a sequence of instructions $I_1, I_2, ..., I_T$, $I_t \in \{Read, Apply\}$ and $1 \leq t \leq T$ and PIR inputs for the ReVAMP architecture that computes the output nodes of the network $G$ with the goal to minimize number of instructions..
*Delay constrained technology mapping :* Given a Boolean function represented as a Boolean logic network $G$ and crossbar width $w_D$, determine a sequence of instructions $I_1, I_2, ..., I_T, I_t \in \{Read, Apply\}$ and $1 \leq t \leq T$, PIR inputs and number of words $S_D$ for the ReVAMP architecture that computes the output nodes of the network $G$ with the goal to minimize number of instructions.

The quality of the solution is measured in terms of the *delay* and the *area* required for the mapping. The *delay* of a solution is equal to the number of instructions ($T$). the *area* of the solution is measured

in terms of the total number of devices used for mapping and is equal to $S_D \times w_D$.

### 5.1.2 Solution Approach

Figure 5.1 shows the overall flowchart of the technology mapping problem for the ReVAMP architecture. In the first approach, we consider the area constrained version of the problem, where we represent the Boolean function as an AIG. We begin by partitioning the AIG into $k$-input Look-up Tables (LUTs). A $k$-input LUT is a function with at most $k$-inputs and a single output. Once the graph has been partitioned, the LUTs for computation are scheduled in topological ordering, i.e., the LUTs close to the primary input are scheduled first and so on, till the output LUTs are computed. In order to compute a LUT, we express the functionality of the LUT using Exclusive Sum-Of-Products (ESOP) [98]. Any arbitrary ESOP can be computed on the DCM with at least 3-wordlines and 2-bitlines (explained in detail in Theorem 5.2) — the variables which have to be used in inverted form are negated first (to be applied via bitlines), followed by computing the product terms and finally XORing them. To reduce the delay, the AND computation for realizing the ESOP needs to minimize number of reads performed and maximize number of AND operations that can be done in parallel. Thereafter, we perform the XOR of computed AND terms by means of a XOR reduction tree of logarithmic depth in the number of AND terms.

In the second approach, we focus on minimizing the *delay* of the mapping, without any constraints on the number of words. We use MIGs for logic representation in this approach. We propose an algorithm with four phases — assignment of nodes as host or input for computation, grouping nodes to blocks, packing blocks to words followed by generation and scheduling of instructions. We explain both the technology mapping solutions in detail in the following sections.

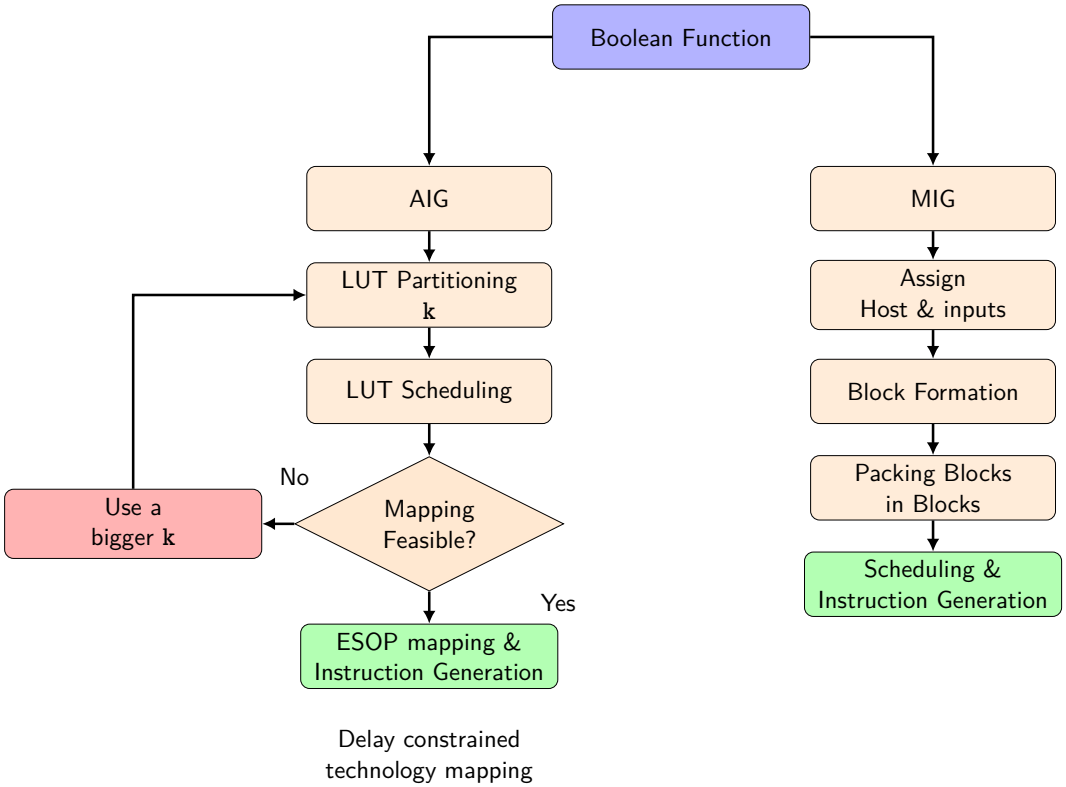



**Figure 5.1:** Technology mapping flow for the ReVAMP architecture.

## 5.2 Area-constrained Technology Mapping for ReVAMP

In this section, we establish a lower bound on the number of devices in crossbar organization required to map any arbitrary AIG or MIG. Thereafter, we present a scalable technique for area-constrained technology mapping for logic-in-memory computation using ReVAMP.

**Theorem 5.1.** *Any AIG or MIG with $k$-levels can be mapped using $2(k+1)$ devices, arranged as a crossbar with atleast two bitlines.*

**Proof:** Since any AIG can be expressed as MIG, we prove the theorem for MIG by means of an inductive proof. Before explaining the proof, we describe a transformation to the input MIG and prove the theorem on the transformed MIG. We transform the MIG such that

- Each internal node has a single child. Nodes with multiple fanout can be replicated bottom-up, i.e., from the output to the primary inputs.

- Each node has two non-inverted inputs and a single inverted input. This can be realized by propagating the inverts across nodes or by creating an inverted copy of the node as required, using the following axiom for Boolean majority.

$$\neg M_3(a, b, c) = M_3(\neg a, \neg b, \neg c) \tag{5.1}$$

Now, we present the inductive proof for the transformed MIG. The device at wordline 0 bitline 0 is used for inverting any input $v$ as needed by applying the input via the bitline with '1' as wordline input and 0 as internal state.

$$M_3(0, 1, \neg v) = \neg v$$

The inverted value $\overline{v}$ can be read out in the next cycle and used in the following cycles using Apply instructions. Any device can be reset to logic 0, by applying '0' and '1' as wordline and bitline input respectively.

$$M_3(x, 0, \neg 1) = 0$$

**Base Case:** A MIG with a single level basically implies inputs act as outputs and hence does not require any devices for computation. Therefore, we consider the MIG in Figure 5.2a with 2-levels as the base case. One of the non-inverted input $W$ and the inverted input $B$ can be loaded to wordline 1. The second non-inverted inverted input $H$ is loaded to wordline 0. The wordline 1 can be readout and in the next cycle, $W$ and $B$ are applied as wordline and bitline inputs of the device holding $H$ to compute $T$.

**Inductive Case:** Let us assume that for an MIG with $k$-levels, the theorem holds true. Now, consider an MIG with $(k+1)$-levels, as shown in Figure 5.2b. The subtrees $MIG_{wk}$, $MIG_{bk}$ and $MIG_{hk}$ have $k$-levels. Therefore, these MIGs can be computed using $2(k+1)$ devices. Let the subtree $MIG_{wk}$ be computed on wordlines 1 to $(k+1)$ and the result $W_k$ be stored at wordline 1 bitline 1. All the devices,

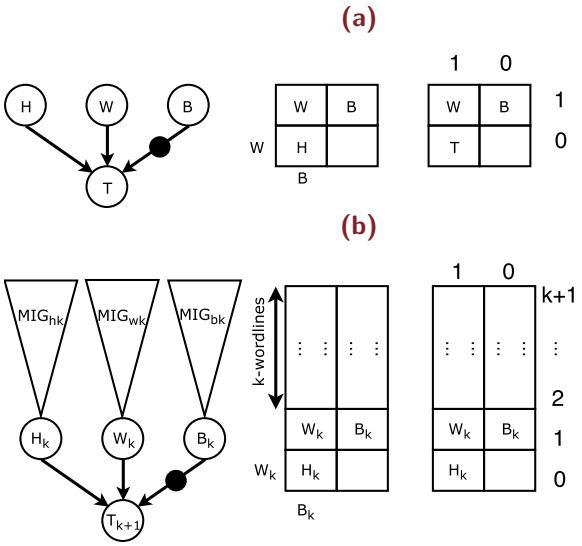



**Figure 5.2:** (a) Mapping an MIG with 2-levels (b) Mapping an MIG with $(k+1)$-levels.

except the device holding $W_k$ is reset. Similarly, subtree $MIG_{bk}$ be computed on wordlines 1 to $k+1$ and the result $B_k$ is stored at wordline 1 bitline 0, followed by reset of all the devices, except wordline 1. The last subtree $MIG_{wk}$ is computed using wordlines 0, 2 to $k+1$ with the result $H_k$ stored at wordline 0 and bitline 1. Therefore, to compute the final output $T_{k+1}$, wordline 1 is read out and then $W_k$ and $B_k$ are applied to the wordline and bitline of device holding $H_k$ to compute $T_{k+1}$. This completes the proof. ■

### 5.2.1 Logic Network Partitioning and Scheduling

We represent the Boolean function as an AIG. We partition the graph into $k$-input LUTs using ABC [6]. From here on, we refer to the partitioned graph as LUT graph and each node in the partitioned graph represents a LUT.

*Bound on number of devices required :* To determine the bound on number of devices required for the storage of intermediate results, we define *transient node.*

**Definition 5.1** (Transient node)**.** *In a LUT graph, a node n is termed as transient node in level l if*

*node* $level(n) < l$ *and there exists an edge,* $n \rightarrow n'$ *such that* $level(n') > l$.

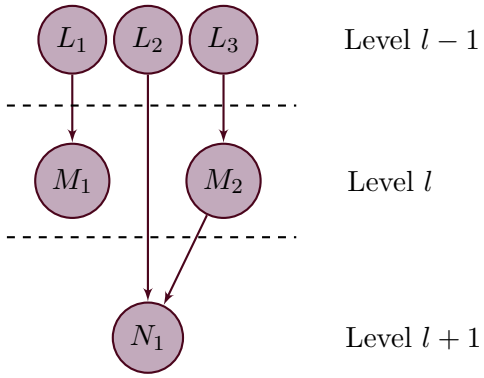



**Figure 5.3:** A portion of the LUT graph. Each node in this graph represents a LUT.

**Example 5.1.** *In Figure 5.3, LUT L2 in level* $l-1$ *has an edge to LUT N1 in level* $l+1$*, therefore it is a transient node for level* $l$.

Let the number of nodes, including transient nodes in a level $l$ be $N_l$. We can schedule the nodes of the LUT Graph in topological ordering, i.e, all nodes at level $l-1$ are scheduled before any node in level $l$ is scheduled. A node in level $l$ is dependent only on the nodes (including transient nodes) that are present in level $l-1$. Therefore, once all the nodes in level $l$ have been scheduled, the devices that store the nodes in level $l-1$ can be reset. Doing this iteratively, the number of devices required for scheduling a LUT graph is

$$Min_{Dev} = \max_{0 \leq l \leq L_{max}-1} (N_l + N_{l+1}) \tag{5.2}$$

The memory layout of the DCM, with $S_D$=$(T+3)$ wordlines and $w_D$ bitlines is shown in Figure 5.4. The top $T$ wordlines are used for storing the output of each LUT. The bottom three wordlines $e_0, e_1$ and $e_2$ are reserved for computation of each LUT.

For the scheduling to be feasible, $Min_{Dev}$ should be less than or equal to $(T \times w_D)$. If the scheduling condition is not feasible, a larger value of $k$ is used to partition the graph and feasibility is checked. If the feasibility condition is not satisfied, a larger value of $k$ is used to lower number of LUTs in the

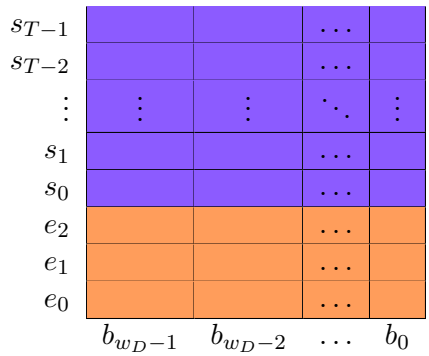



**Figure 5.4:** Memory layout of DCM with $S_D$ wordlines and $w_D$ bitlines, for area-constrained technology mapping. The three wordlines $e_0$, $e_1$ and $e_2$ are used for computation. The rest of the wordlines $s_0, \ldots, s_T$ are used for storage of the intermediate results. $S_D = T + 3$.

**Algorithm 5:** LUT Graph Scheduling

```
Data: Lut Graph G,S_D,w_D
for l = 1; l ≤ l_max; l++ do
    Allocate unscheduled nodes in level l to free devices in a wordline considering Best-fit;
    if s.scheduled = True : ∀s ∈ succ(n) then
        // Device allocated to node n is marked dirty
        dev(n).dirty = True;
    while level(n) = l and n.scheduled = False : ∃n ∈ G(V) do
        // No free device is available
        if ∄free(D) then
            w = wordline with maximum number of dirty devices;
            Reset the dirty devices in wordline w;
        Allocated unscheduled nodes of level l to the free devices in wordline w;
```

LUT graph, which lowers the $Min_{Dev}$. Once the scheduling condition is satisfied for a given crossbar size, nodes are scheduled in topological order. The device where the output of an LUT (node in the LUT graph) would be stored, is determined according to the best fit method. The wordline in the crossbar with minimum number of free devices is chosen if the number of nodes to schedule is less than or equal to the number of free devices in that wordline. If no such wordline exists, a wordline with maximum number of free devices is chosen iteratively, till all the nodes have been allocated a device. A device storing a node $n$ is marked *dirty* if all the successors of $n$ have already been allocated. If none of the devices are *free*, then the wordline with maximum dirty bits is reset and allocation starts. This process is repeated till all the nodes have been scheduled, along with target device allocation. The overall technique has been shown in Algorithm 5.

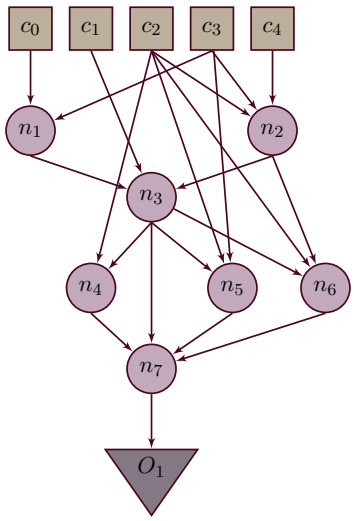



**Figure 5.5:** An LUT graph with five primary inputs ($c_0, \ldots, c_5$) and LUTs ($n_1, \ldots, n_7$). The output of LUT $n_7$ drives the the output $O_1$ of the LUT graph.

**Example 5.2.** *We explain the device allocation and scheduling technique, presented in Algorithm 5 using a representative LUT graph, shown in Figure 5.5. The nodes are scheduled in topological ordering. Nodes in level 1, $n_1$ and $n_2$, are allocated to wordline 5, as shown in Figure 5.6. Node $n_3$, in level 2, is assigned another device in wordline 5, using the Best-fit allocation strategy. Since the only successor of $n_1$ has been allocated, device allocated to $n_1$ is now marked dirty. In level 3, there are 3 nodes ($n_4, n_5$ and $n_6$). Since there is only a single device free in wordline 5, it is not possible to allocate these nodes together. Therefore, these nodes are allocated to wordline 4. All the successors of node $n_2$ have been allocated, hence the corresponding device is marked as dirty. Finally, the node $n_7$ in level 4 is allocated to the free device in wordline 5. This completes the allocation and scheduling of the LUT nodes.*

### 5.2.2 ESOP Computation Using ReRAM Crossbar Array

Each function realized by the LUT can be expressed as an Exclusive Sum-Of-Product (ESOP). For many Boolean functions, minimal ESOPs have lesser number of cubes compared to Sum-Of-Products [122]. In addition, there are multiple ESOP minimizers available which can be used to reduce the ESOP size [123–126]. Before presenting the ESOP computation algorithm on ReVAMP, we present a brief

| DCM | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

| Level 1 | | | |
|---|---|---|---|
| 0 | 0 | $\overline{n_2}$ | $\overline{n_1}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

| Level 2 | | | |
|---|---|---|---|
| 0 | $\overline{n_3}$ | $\overline{n_2}$ | $\overline{n_1}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

| Level 3 | | | |
|---|---|---|---|
| 0 | $\overline{n_3}$ | $\overline{n_2}$ | $\overline{n_1}$ |
| 0 | $\overline{n_6}$ | $\overline{n_5}$ | $\overline{n_4}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

| Level 4 | | | |
|---|---|---|---|
| $\overline{n_7}$ | $\overline{n_3}$ | $\overline{n_2}$ | $\overline{n_1}$ |
| 0 | $\overline{n_6}$ | $\overline{n_5}$ | $\overline{n_4}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

**Figure 5.6:** Scheduling using Algorithm 5 for the LUT graph in Figure 5.5. The wordlines 0-3 (colored in orange) are reserved for XOR computation and the remaining wordlines 4-5 (colored in violet) are used for storage of the LUT outputs.

description of the related terms.

**Definition 5.2** (Literal)**.** *A* literal *is a Boolean variable either in inverted or non-inverted form.*

**Definition 5.3** (Cube)**.** *A* cube *is a product term composed of literals using Boolean AND.*

**Example 5.3.** *The ESOP $\overline{a}bc \oplus a\overline{b}c$ has two cubes, $\overline{a}bc$ and $a\overline{b}c$. The cube $\overline{a}bc$ has literal a in inverted form and b,c in non-inverted form.*

**Theorem 5.2.** *Any Boolean function, expressed as an ESOP, can be computed using three wordlines and atleast two bitlines.*

**Proof :** We present a constructive proof for the theorem. Let us consider three wordlines, $e_0$, $e_1$ and $e_2$ with bitlines $b_0$ and $b_1$. We consider two cases.

*Case 1:* The ESOP has a single cube $\alpha$, say $\alpha = l_1 \wedge l_2 \wedge ... \wedge l_n$. If a literal $l_i$ is inverted, it is applied via wordline $e_0$ and '0' is applied as input to bitline $b_0$. Else, the literal is applied via bitline $b_0$ and '1' as input to wordline $e_0$ to store in non-inverted form. Then, wordline $e_0$ is read out and stored value is applied via the bitline with '0' as wordline input to wordline $e_2$. The wordline $e_0$ is reset. The process is repeated till all the literals have been ANDed and the computed cube is available at wordline $e_2$ and bitline $b_0$.

*Case 2:* The ESOP has more than one cube, say $\alpha_1, \alpha_2, \ldots, \alpha_m$. The cube $\alpha_1$ can be computed, as stated in *Case 1*. Similarly, $\alpha_2$ can be computed at wordline $e_2$ and bitline $b_1$ by applying the bitline

inputs via bitline $b_1$. The cubes $\alpha_1$ and $\alpha_2$ can be XORed as shown in Figure 5.7 with the result stored at wordline $e_2$ and bitline $b_1$. Rest of the devices are reset to 0 by using '0' as the wordline input and '1' as the bitline input. Now, the third cube $\alpha_3$ can be computed, using steps identical to *Case 1* and the XOR can be performed with the result $\alpha_1 \oplus \alpha_2$. This process can be repeated till the entire ESOP has been computed. ■

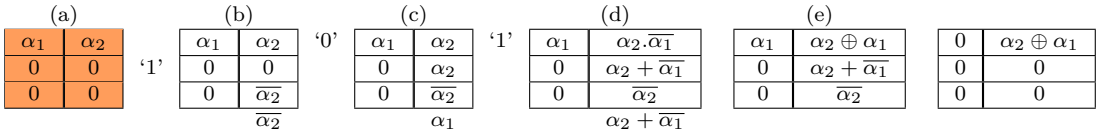



**Figure 5.7:** ESOP computation on a $3 \times 2$ crossbar. (a) Cubes are computed on devices at wordline $e_2$. (b-e) Some of the intermediate steps of computing XOR of the two cubes are shown. (f) All devices, except the device holding the XOR of the cubes is reset to 0. If the ESOP has more cubes, the next cube $\alpha_3$ would be computed at wordline $e_2$ and bitline $b_0$ and XOR would be computed for $\alpha_3$ and $\alpha_2 \oplus \alpha_1$, followed by reset. This process is repeated till the entire ESOP has been computed.

The theorem 5.2 guarantees that any ESOP can be computed in a crossbar with three wordlines and two bitlines. If the number of bitlines is greater, it is possible to reduce the delay by parallelizing operations. Boolean AND of two literals $a$ and $b$ can be expressed as $M_3(a, 0, b)$. '0' can be used a common wordline input during computation of cubes in parallel feasible. Figure 5.8 shows the computation of the cubes of an ESOP. Due to the crossbar constraints, all the bitline-applied literals must be either available via the PIR or DMR simultaneously. This implies that all the applied literals either have to be primary inputs or must reside on the same wordline for parallel computation of the cubes.

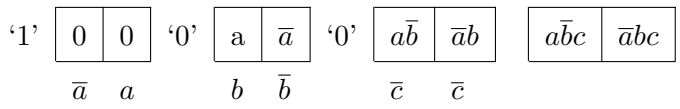



**Figure 5.8:** Computation of two cubes of an ESOP $\alpha_1 \oplus \alpha_2$. $\alpha_1 = \overline{a}bc$ and $\alpha_2 = a\overline{b}c$.

At the end of completion of computation of the cubes, the cubes have to be XORed. Each XOR can be performed using that steps similar to the example shown in Figure 2.8. Multiple XORs can be performed by means of a XOR reduction tree with logarithmic depth, in the number of terms to be XORed. In Figure 5.9, there are four terms $\alpha_i$ to be XORed. The XOR of $\alpha_1$ and $\alpha_2$ can proceed in

parallel with the XOR of $\alpha_3$ and $\alpha_4$. Thereafter, the results $\alpha_{12}$ and $\alpha_{34}$ are XORed. It might happen that the number of cubes in an ESOP is greater the number of available bitlines in the crossbar. In that case, the computation of the cubes, followed by XOR reduction has to be iterated. The technique for ESOP computation is presented in Algorithm 6. Once the ESOP has been evaluated, the result is written back to the position in the intermediate storage area, as determined by the scheduling algorithm.

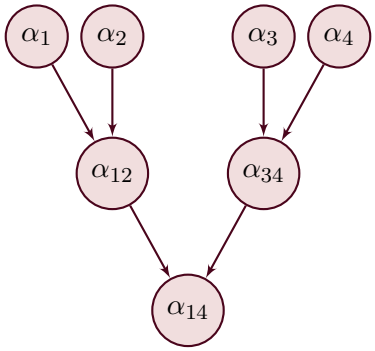



**Figure 5.9:** A XOR-reduction tree consisting of 4 terms — $\alpha_1$, $\alpha_2$, $\alpha_3$ and $\alpha_4$. $\alpha_{12} = \alpha_1 \oplus \alpha_2$, $\alpha_{34} = \alpha_3 \oplus \alpha_4$, $\alpha_{14} = \alpha_{12} \oplus \alpha_{34}$

The proposed approach provides a novel solution to the area-constrained technology mapping problem. The target Boolean function is represented as an AIG, followed by partitioning into $k$-input LUTs and finally scheduling and computing these LUTs on the crossbar. The approach allows a feasible mapping for a variety of crossbar sizes, with some portion of the crossbar reserved for computation of ESOPs. Instead of using AIGs for representing the functions, it is also feasible to represent the function other structural representation, such as MIG. This structural representation can be partitioned into LUT graph and the area-constrained technology mapping can be performed similarly. In the following section, we look at delay-constrained technology mapping problem. We restrict the solution of problem by only the word length of the DCM, without any restrictions on the number of words available for mapping.

**Algorithm 6:** ESOP computation

```
   Data: E, CrossbarState, Loc
1  result = null;
2  do
3      activeCubes = set();
4      currentLiterals = list();
5      v = max_{l∈C} occ(l);
6      currentLiteral.append(v);
7      activeCubes.add(cube(v));
8      while occ(currentLiteral) ≤ w_D do
9          allowedLiterals = findAllowed(E,currentLiterals);
10         Choose v' ∈ allowedLiterals with max occurence in cube(v') − activeCubes and occ(currentLiteral + v') ≤ w_D;
             currentLiteral.append(v');
11         activeCubes.add(cube(v'));
12     And(currentLiterals);
13     do
14         activeCubes' = set();
15         currentLiterals' = list();
16         v = max_{l∈C} occ(l,activeCubes);
17         while occ(currentLiteral) ≤ w_D do
18             allowedLiterals = findAllowed(E,currentLiterals);
19             Choose v' ∈ allowedLiterals with max occurence in cube(v') − activeCubes and
                 occ(currentLiteral + v') ≤ w_D; currentLiteral.append(v');
20             activeCubes'.add(cube(v'));
21         And(currentLiterals);
22     while All the activeCubes have not been computed;
23     result = XorReduction(activeCubes, result);
24 while Some cube has not been computed;
```

## 5.3 Delay-Constrained Technology Mapping for ReVAMP

In this section, we present a technology mapping technique for the ReVAMP architecture. The method is focused at reducing the delay of mapping for a given word length $w_D$, without any constraint on the number of words available for mapping. This approach is practical, when the size of the memory available for computing is large. Since ReRAM devices inherently compute 3-input Boolean majority with one input inverted, MIG is apt for logic representation.

### 5.3.1 Assign Host And Inputs To Nodes

A ReRAM device has an internal resistive state, and two inputlines — wordline and bitline. A computation on the device updates the internal state of the device, in effect making the device the *host* for the computation. For each internal node in an MIG, one of the predecessors hosts the computation and the remaining predecessors act as wordline and bitline inputs. The computation of multiple independent nodes can be grouped into an Apply instruction if they have a common wordline input. Based on this, we present a few rules to assign the host and the inputs of the nodes of an MIG.

- If a node has multiple successors in the same level, then it can be used as common wordline input for computing those nodes. For instance, in Figure 5.10, input $c_2$ can be used as common wordline input to compute $S_1$ and $S_2$.
- If an incoming edge to a node is marked inverted, then the corresponding predecessor can be used as the bitline input. In Figure 5.10, $c_3$ and $S_2$ are used as bitline inputs to compute $S_1$ and $S_3$ respectively.
- If there are no inverted incoming edges to a node, then a negated predecessor is used as input to that node. For node $S_2$ in Figure 5.10, input $\overline{c_3}$ is used as bitline input.
- The remaining predecessor is used as *host* for the node. The nodes $c_1$ and $S_1$ act as *host* to

compute $S_1$ and $S_3$ respectively in Figure 5.10.

These rules ensure that the nodes with common inputs can share wordline inputs which is used for scheduling computation. We mark these assignments on the edges of the MIG, as shown in Figure 5.10.

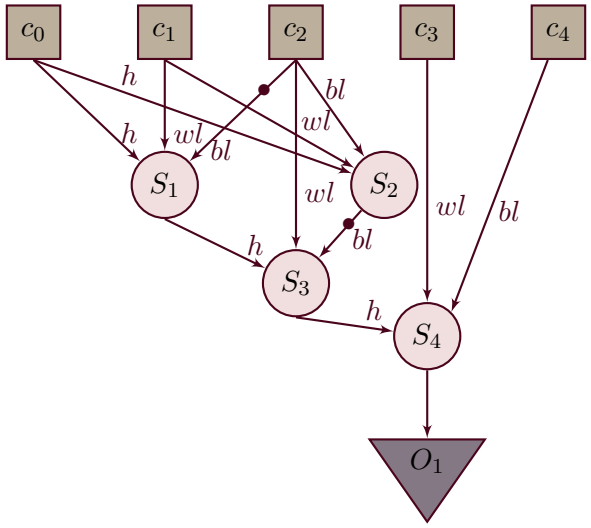



**Figure 5.10:** Edge marking on the MIG to signify input assignments — *host* indicates the predecessor of the node is used as host for computation while *wl* and *bl* indicates that the predecessor of the node is used as wordline and bitline input respectively.

### 5.3.2 Group Nodes To Blocks

To compute an internal node in a MIG, we need to read out the wordline and bitlines inputs of the node and then apply these inputs to the host. Given that only a single word can be read out in a clock cycle, the wordline and bitline inputs of the node must reside on the same wordline to allow computation of the node. This creates a constraint that for each node in an MIG — the wordline and the bitline inputs must be placed in the same word. We call this grouping a *block*.

Further, as read-outs are non-destructive, blocks can be merged if they have common inputlines. This reduces the number of devices required, with the merged block having only one copy of the common inputline. Note that blocks can be merged only if the number of inputs in the resultant block does not exceed the word length.

Also, a pair of blocks in the same level that have hosts which share a wordline input should be merged. This host-based merge along with merge of the corresponding blocks with the inputlines of these hosts permits computation of the nodes in the same level with shared wordline in a single cycle, thereby reducing delay.

**Algorithm 7:** Block Formation Algorithm

```
   Data: G, pi, po
   Result: blockList
1  blockCount = 0;
2  for node_out ∈ po do
3      addBlock([(node_out.host)],blockCount);
4      addBlock([(node_out.wl, node_out.bl)],blockCount);
5      addInversionBlock(el);
6  mergeBlock();
7  for l = level_max; l > 0; l = l − 1 do
8      for block ∈ blockList do
9          for el ∈ block do
10             if el ∉ pi and el.level == l then
11                 replace(el, el.host);
12                 addBlock([el.wl, el.bl], blockCount);
13                 addInversionBlock(el);
14     mergeBlock();
```

The algorithm of the block formation is shown in Algorithm 7. The lines $2-5$ creates the blocks considering the placement constraint on the input lines of the output nodes. The addInversionBlock method adds the positive nodes as blocks to the blockList, if the added blocks have inverted values. Only a single positive node is added to blockList, corresponding to multiple copies of a negated node. The mergeBlock method merges blocks based on the input line and host based merge constraints. The replace method replaces a node in a block with its host node.

**Table 5.1:** BlockList update with BlockMerge algorithm for MIG of Figure 5.10

| Level | BlockList |
|---|---|
| Output | [[1,$S_4$)]] |
| 3 | [[(1,$S_3$,h)],[(2,$c_4$,i),(2,$\overline{c_5}$,i)]] |
| 2 | [[(1,$S_1$,h)],[(2,$c_4$,i),(2,$\overline{c_5}$,i)],[(3,$c_3$,i),(3,$S_2$,i)]] |
| 1 | [[1:($c_1$,h)],[2:($c_4$,i),($\overline{c_5}$,i)],[3:($c_3$,i),($c_1$,h)],[4:($c_2$,i),($c_3$,i)],[5:($c_2$,i),($\overline{c_3}$,i)]] |
| 1 | [[1:($c_1$,h)],[2:($c_4$,i),($\overline{c_5}$,i)],[3:($c_3$,i),($c_1$,h)],[4:($c_2$,i),($c_3$,i),($\overline{c_3}$,i)]] |
| 1 | [[1:($c_1$,h),($c_3$,i),($c_1$,h)],[2:($c_2$,i),($\overline{c_5}$,i)],[4:($c_2$,i),($c_3$,i),($\overline{c_3}$,i)] |

**Example 5.4.** *For a word length ($w_D$) of 3, Table 5.1 shows the working of the block formation algorithm*

*on the MIG of Figure 5.10. Starting at the output node, blockList has a single block. At level 3, node* $S_4$ *is replaced with its host and input lines. Since these two blocks do not have any common inputlines or hosts, they cannot be merged. At level 2, node* $S_3$ *gets replaced and the inputlines are added to a new block. At level 1, nodes* $S_1$ *and* $S_2$ *are replaced by their hosts a, and the inputlines are inserted in two new blocks. Blocks 4 and 5 have a common inputline* $c_2$ *and are hence merged. Blocks 2 and 4 have common inputs, but cannot be merged as the length* 4 *of the resultant block will exceed the given word length* 3*. Thereafter, since the two* $c_1$ *host nodes have the same wordline, blocks 1 and 3 get merged, but both copies of the host are retained, using the host-merge constraint.*

### 5.3.3 Pack Blocks In Words

At the end of scheduling computation, we have blocks of elements, which have to be placed in the same wordline. The number of elements in each block is less than or equal to $w_D$, the number of bits in a word. Now, these blocks have to packed in the DCM using the minimum number of words. The problem can be formulated as a bin packing problem as defined below.

**Algorithm 8:** First-fit Algorithm

```
   Data: blockList,wordlength
   Result: blockToWord
1  wordToBlock = HashMap();
2  wc = 0;
3  for block ∈ blockList do
4      assigned = False;
5      for w ∈ wordToBlock do
6          if wordToBlock[w].occupied + block.length < wordlength then
7              wordToBlock[w].occupied =wordToBlock[w].occupied + block.length;
8              wordToBlock[w].append(block);
9              assigned = True;
10         if assigned == False then
11             wc = wc+1;
12             wordToBlock[wc].append(block) wordToBlock[wc].occupied = block.length;
```

Consider each word in the DMR as a bin, with capacity $w_D$. Each block $b_i$ has a value $v_i$, $v_i > 0$. Each block must be assigned to a bin such the total value of the objects assigned to the bin is less than or equal to $w_D$. The objective is to minimize the number of bins required to assign all the block,

without violating the capacity constraint.

This *first-fit* algorithm provides a 2-factor approximation, i.e., the number of words required by the algorithm is at most twice the number of words required by the optimal solution.

**Example 5.5.** *For the example, the blocks determined by the Block Formation algorithm are placed in a separate wordline, as shown in Figure 5.11 (a).*

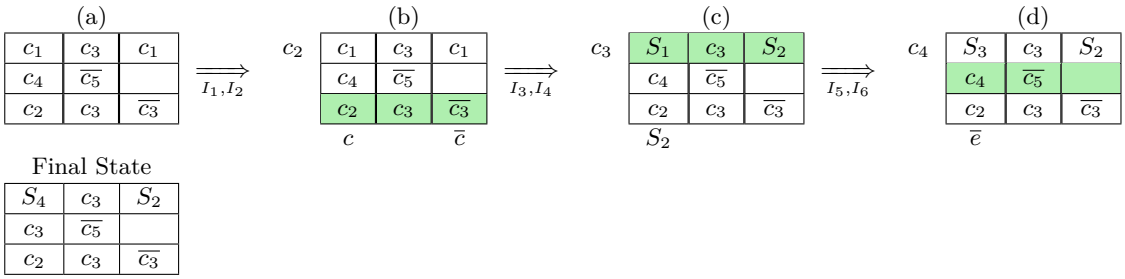



**Figure 5.11:** DCM state transition during computation using delay-constrained technology mapping. (a) DCM state after loading the primary inputs. (b-d) The intermediate DCM states during computation. The final DCM state is shown. The green coloured row represents the read out wordline.

### 5.3.4 Generation And Scheduling Instructions

The primary inputs have to be loaded into the DCM before computation of the internal nodes of the MIG can begin. In each clock cycle, $w_D$ primary inputs can be read. The primary inputs are loaded via the bitline and hence the inverted values are stored in a single clock cycle. To store non-inverted primary inputs, the primary inputs are written to a wordline, thereby storing it in inverted form. Then, the inverted value is read out and applied via the bitline to store the non-inverted value to the required wordline. A single extra wordline is used for storage of the intermediate inverted primary input, and this wordline is reset, after each use.

All the nodes in level $i$ are scheduled for computation before any node at level $i+1$ is scheduled. The nodes in the same level can be scheduled in any order as they do not have any data dependencies. The nodes in a level with hosts of the which are in the same block, and the corresponding inputlines

are also placed together in the same block, are scheduled for computation together. Once all the nodes in a level have been computed, we determine whether any inverted copies of the nodes are required for computation of nodes are greater level. If inverted copies are needed, the node is read out and stored in inverted form in the required block by writing through the bitline. Each computation is expressed as an *Apply* instruction and read operations are expressed as *Read* instructions.

**Table 5.2:** Instruction sequence to compute MIG in Figure 5.10.

| | **Instruction** |
|---|---|
| $I_1$ | Read 0 |
| $I_2$ | Apply 2 11 0 1 1 0 0 1 2 |
| $I_3$ | Read 2 |
| $I_4$ | Apply 2 11 1 1 2 0 0 0 0 |
| $I_5$ | Read 1 |
| $I_6$ | Apply 2 11 0 1 1 0 0 0 0 |

**Example 5.6.** *Table 5.2 shows the sequence of instructions used to compute the example MIG, and Figure 5.11 shows the changes in DCM state on application of the Apply instructions. Note that the additional instructions needed to initialize the DCM are not shown. The inputs to compute nodes* $S_1$ *and* $S_2$ *are in word 0 and are read out. The hosts of nodes* $S_1$ *and* $S_2$ *are in word 2, and therefore* $I_2$ *computes these nodes in word 2. The inputs to compute* $S_3$ *are in word 2, and are read out by* $I_3$*.* $I_4$ *computes* $S_3$ *in host* $S_1$*. Finally to compute* $S_4$*,* $I_5$ *reads out word 1 and* $I_5$ *applies the required inputs to* $S_3$*.*

## 5.4 Experimental Results

We have implemented the proposed compilation flow for the ReVAMP architecture using Python. The algorithm was evaluated using the EPFL benchmarks*. For area-constrained mapping, we used ABC for generating the initial AIG and also for ESOP expansion [6]. Each run is limited to 2 hours, exceeding which the program is terminated. The major amount of time in mapping is spent in ESOP expansion.

*http://lsi.epfl.ch/benchmark

**Table 5.3:** Performance of area-constrained mapping on a fixed crossbar dimension $S_D \times w_D = 64 \times 64$. $\times\times$ denotes the benchmark that cannot be mapped with the current crossbar dimension due to $Min_{Dev} > 3904$. $--$ indicates the benchmark that did not complete mapping within the time limit (2 hours).

| *Benchmark* | $k = 4$ | | | | $k = 8$ | | | | $k = 16$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\#N_{LUT}$ | $\#L$ | $Min_{Dev}$ | $\#C$ | $\#N_{LUT}$ | $\#L$ | $Min_{Dev}$ | $\#C$ | $\#N_{LUT}$ | $\#L$ | $Min_{Dev}$ | $\#C$ |
| ac97_ctrl.v | 3900 | 4 | 2743 | 88893 | 2630 | 3 | 2102 | 83464 | 2409 | 2 | 2409 | 87908 |
| aes_core.v | 8978 | 8 | 7006 | $\times\times$ | 902 | 3 | 890 | 33804 | 816 | 2 | 816 | 40091 |
| comp.v | 8918 | 50 | 2624 | 188156 | 5695 | 29 | 1987 | 179905 | 4169 | 19 | 2143 | 301379 |
| des_area.v | 2020 | 10 | 1085 | 44303 | 699 | 5 | 505 | 29885 | 884 | 3 | 859 | 57017 |
| des_perf.v | 34260 | 6 | 26309 | $\times\times$ | 11500 | 3 | 11476 | $\times\times$ | 9713 | 2 | 9713 | $\times\times$ |
| diffeq1.v | 6652 | 72 | 1736 | 164355 | 3949 | 38 | 1247 | 169057 | 2710 | 22 | 1079 | 414193 |
| div16.v | 1033 | 85 | 151 | 23557 | 710 | 39 | 131 | 27276 | 507 | 20 | 155 | 51407 |
| DSP.v | 14679 | 22 | 4513 | $\times\times$ | 9025 | 11 | 3823 | 323789 | 7195 | 7 | 3733 | 416307 |
| ethernet.v | 20178 | 10 | 11855 | $\times\times$ | 14385 | 6 | 10462 | $\times\times$ | 12113 | 3 | 9869 | $\times\times$ |
| hamming.v | 696 | 24 | 243 | 16676 | 498 | 13 | 221 | 19841 | 413 | 8 | 220 | $--$ |
| i2c.v | 347 | 7 | 255 | 7264 | 219 | 3 | 157 | 5929 | 156 | 2 | 156 | 5921 |
| log2.v | 10331 | 107 | 2996 | 257920 | 4415 | 48 | 1208 | 240998 | 2859 | 25 | 723 | $--$ |
| MAC32.v | 2828 | 29 | 1347 | 75746 | 1624 | 13 | 979 | 72908 | 967 | 7 | 775 | 110020 |
| max.v | 1023 | 54 | 735 | 25815 | 687 | 24 | 658 | 30267 | 615 | 12 | 615 | 45412 |
| mem_ctrl.v | 3523 | 15 | 1252 | 71790 | 2372 | 7 | 1156 | 64689 | 2038 | 4 | 1371 | 84581 |
| MUL32.v | 2458 | 18 | 1103 | 66538 | 1580 | 9 | 942 | 72954 | 879 | 6 | 617 | 189105 |
| mult64.v | 7438 | 87 | 3105 | 229381 | 3937 | 40 | 1418 | 208960 | 2860 | 20 | 1109 | 468718 |
| pci_bridge32.v | 6384 | 10 | 2435 | 143697 | 4585 | 6 | 2294 | 142557 | 4189 | 3 | 2927 | $--$ |
| pci_spoci_ctrl.v | 373 | 7 | 296 | 7294 | 242 | 4 | 197 | 7144 | 119 | 3 | 96 | 4859 |
| revx.v | 2659 | 61 | 252 | 62304 | 1644 | 34 | 267 | 71856 | 559 | 15 | 234 | 153372 |
| sasc.v | 207 | 3 | 161 | 4876 | 147 | 2 | 147 | 4411 | 132 | 1 | 132 | 5038 |
| simple_spi.v | 279 | 6 | 182 | 6002 | 187 | 3 | 137 | 5641 | 150 | 2 | 150 | 5762 |
| spi.v | 1211 | 11 | 656 | 27253 | 906 | 5 | 692 | 30900 | 491 | 3 | 359 | 36109 |
| sqrt32.v | 264 | 112 | 83 | 8103 | 206 | 37 | 79 | 10899 | 158 | 15 | 70 | 33350 |
| square.v | 5721 | 83 | 3128 | 154151 | 3466 | 36 | 1894 | 131707 | 2557 | 17 | 1672 | 193112 |
| ss_pcm.v | 127 | 3 | 109 | 3040 | 100 | 2 | 100 | 3281 | 98 | 1 | 98 | 3243 |
| systemcaes.v | 3255 | 11 | 1830 | 79168 | 1565 | 6 | 693 | 59813 | 1033 | 4 | 692 | 49199 |
| systemcdes.v | 1109 | 9 | 576 | 26040 | 516 | 3 | 497 | 15857 | 290 | 2 | 290 | 19303 |
| tv80.v | 2822 | 19 | 1150 | 64901 | 1523 | 10 | 802 | 54089 | 769 | 5 | 562 | 49165 |
| usb_funct.v | 4678 | 12 | 2330 | 110240 | 2765 | 6 | 1383 | 91873 | 2177 | 3 | 1455 | 126515 |
| usb_phy.v | 181 | 4 | 151 | 3382 | 120 | 2 | 120 | 3061 | 111 | 1 | 111 | 3046 |

For all the EPFL benchmarks, Table 5.3 presents the results of the area-constrained mapping for varying number of LUT inputs $k$ for fixed crossbar dimension of $64 \times 64$. With increase in $k$, the number of LUTs ($\#N_{LUT}$) in the LUT graph reduces, along with reduction in the number of levels ($\#L$). For the given crossbar dimensions, 61 words are available for storing the intermediate results and 3 are reserved for computing the ESOPs. Some of the benchmarks could not be mapped (marked by $\times\times$) due to violation of the feasibility criteria ($Min_{Dev} > 3904$), presented in Equation (5.2).

To analyze the impact of increasing number of LUT inputs ($k$) on delay and $Min_{Dev}$ in detail, we consider four large benchmarks from the EPFL benchmark suite for crossbar dimension $64 \times 64$. The results are shown in Figure 5.12. The effect of $k$ on $Min_{dev}$ is dependent on the benchmark itself. For example, with increase in value of $k$, $Min_{dev}$ for the benchmark *mul32* decreases but for *mem_ctrl*, $Min_{dev}$ increases for larger values of $k$, as evident from the Figure 5.12. The delay of the mapping (in terms of number of cycles $\#C$) closely follows the trend of $Min_{dev}$ i.e. with increase in $Min_{dev}$, the overall number of cycles required for mapping increases. This is because with increase in $Min_{dev}$, less number of crossbar devices can be reset at any given time, which leads to reduction in the parallelization of operations during the ESOP computation. For the benchmark *mul*32, notice the sharp rise in delay of mapping on changing $k$ from 13 to 16. The number of cubes in the ESOP expression for increased consistently, resulting in the increased time of computation of the ESOP expression, that increases the overall delay of mapping. Also, for large values of $k$ ($k \geq 28$), the time required for each ESOP expansion increases considerably ($> 2$ hours), which leads to long execution time for mapping an entire benchmark.

We analyze the impact of crossbar dimension on the delay of mapping. Keeping the overall number of devices fixed to 4096, we vary the number of bitlines from 4 to 1024. The results are shown in Figure 5.13. With increase in the number of bitlines, the crossbar permits greater number of parallel operations that can be carried out in a word. This parallelism is harnessed by the ESOP computation technique, which leads to reduction in delay of mapping for the entire benchmark.

The results of delay constrained mapping are presented in Table 5.4 for word length ($w_D$) of 16-bits.

**Table 5.4:** Performance of delay-constrained mapping on the EPFL benchmarks for $w_D$=16.

| ID | Benchmark | $N_{Maj}$ | $I_A$ | $I_R$ | $I_{total}$ | #B | $S_D$ | $W_{Util}$ | $\#C$ | $D_{P*}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ac97_ctrl | 15253 | 8803 | 7520 | 16323 | 2330 | 933 | 99.88 | 16325 | 137277 |
| 2 | comp | 18967 | 32297 | 31293 | 63590 | 3114 | 1182 | 99.96 | 63592 | 170703 |
| 3 | des_area | 4629 | 5971 | 5639 | 11610 | 1073 | 305 | 99.92 | 11612 | 41661 |
| 4 | div16 | 4440 | 8375 | 7825 | 16200 | 702 | 342 | 99.96 | 16202 | 39960 |
| 5 | hamming | 2280 | 3603 | 3450 | 7053 | 623 | 180 | 99.58 | 7055 | 20520 |
| 6 | i2c | 1263 | 1450 | 1247 | 2697 | 297 | 83 | 99.32 | 2699 | 11367 |
| 7 | MAC32 | 9489 | 15980 | 15363 | 31343 | 3045 | 784 | 99.85 | 31345 | 85401 |
| 8 | max | 4854 | 4431 | 4184 | 8615 | 990 | 314 | 99.2 | 8617 | 43686 |
| 9 | mem_ctrl | 9569 | 9497 | 8405 | 17902 | 2032 | 625 | 99.65 | 17904 | 86121 |
| 10 | MUL32 | 9226 | 14047 | 13389 | 27436 | 2406 | 718 | 99.28 | 27438 | 83034 |
| 11 | pci_bridge32 | 25653 | 23826 | 21914 | 45740 | 6535 | 1546 | 99.82 | 45742 | 230877 |
| 12 | pci_spoci_ctrl | 1096 | 1523 | 1328 | 2851 | 196 | 85 | 99.04 | 2853 | 9864 |
| 13 | revx | 7563 | 14575 | 14004 | 28579 | 1703 | 611 | 100 | 28581 | 68067 |
| 14 | sasc | 889 | 602 | 514 | 1116 | 170 | 57 | 99.01 | 1118 | 8001 |
| 15 | simple_spi | 1135 | 930 | 794 | 1724 | 209 | 75 | 99.17 | 1726 | 10215 |
| 16 | spi | 3890 | 4615 | 4301 | 8916 | 885 | 267 | 99.98 | 8918 | 35010 |
| 17 | sqrt32 | 2206 | 4216 | 3948 | 8164 | 442 | 170 | 99.85 | 8166 | 19854 |
| 18 | square | 18080 | 30988 | 29880 | 60868 | 5640 | 1454 | 99.78 | 60870 | 162720 |
| 19 | ss_pcm | 604 | 313 | 257 | 570 | 97 | 31 | 98.59 | 572 | 5436 |
| 20 | systemcaes | 11299 | 11100 | 10229 | 21329 | 2602 | 721 | 99.84 | 21331 | 101691 |
| 21 | systemcdes | 3028 | 3312 | 3090 | 6402 | 627 | 195 | 99.49 | 6404 | 27252 |
| 22 | tv80 | 8177 | 11219 | 10368 | 21587 | 1672 | 578 | 99.73 | 21589 | 73593 |
| 23 | usb_funct | 16704 | 16054 | 14269 | 30323 | 2943 | 1057 | 99.77 | 30325 | 150336 |
| 24 | usb_phy | 599 | 538 | 460 | 998 | 140 | 37 | 97.47 | 1000 | 5391 |

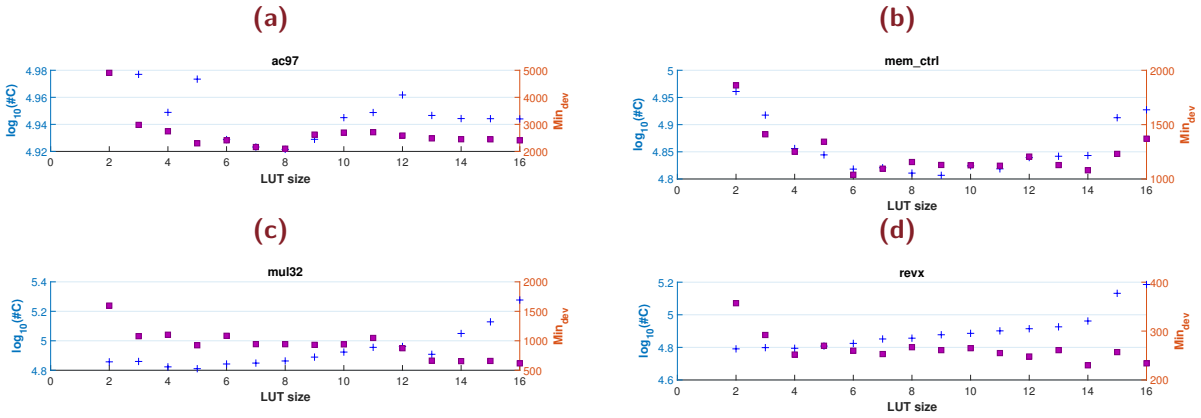



**Figure 5.12:** Impact of LUT size (k) on delay (#C) for a fixed crossbar dimension $64 \times 64$. The blue $+$ symbol indicates the delay of mapping $\#C$ in the log scale. The violet square denotes the minimum number of devices $Min_{dev}$ required for a feasible mapping.

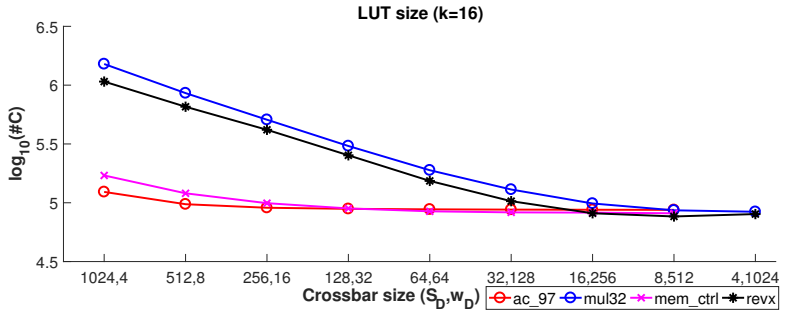



**Figure 5.13:** Impact of crossbar dimensions on delay ($\#C$) of area-constrained mapping using constant number of devices. 4096 devices are used for all the mappings and the number of bitlines ($w_D$) is increased from 4 to 1024, doubling at each step.

For most of the benchmarks, the compilation time to generate the instructions, was a few seconds while for the larger benchmarks, the compilation process finished under 20 minutes. The number of Read and Apply instructions are shown in column $I_A$ and $I_R$ respectively while the total number of instructions is $I_{Total}$. The number of blocks created by mapping is $\#B$. The total delay ($\#C$) of the mapping solution is the number of cycles to complete computation of the benchmark by the ReVAMP architecture. The number of words ($S_D$) used determines the area of the mapping solution. To determine the effectiveness of the packing algorithm to pack blocks into words, we utilize the word utilization ($W_{Util}$) metric. $W_{Util}$ is the percentage of total number of bits in $S_D$ words, that are used by the mapping solution. For the example MIG, out of 9 bits (3 words, each with 3 bits), 8 bits are used and therefore $W_{Util}$ is 88.8%.

The proposed packing algorithm achieves more than 97% utilization for all the benchmarks, when $w_D$ = 16, including 100% utilization for the *revx* benchmark. Figure 5.14a shows that with increase in word length from 4 to 8, leads to considerable improvement in $W_{Util}$. However, the $W_{Util}$ is comparable for the word lengths, 8, 16 and 32, and approaches 100%. This shows the effectiveness of delay-constrained mapping to pack the blocks into words.

Gaillardon et al. [82] proposed the PLiM computer, which has a single instruction — $RM_3\ A, B, Z$. Assuming 16-bit words, each instruction results in the following micro operations on the memory array: Read @A (32 bits), Read @B (32 bits), Read @Z (32 bits), Read A (1 bit), Read B (1 bit), Write @Z (1 bit). This corresponds to 9 R/W cycles on the considered machine. Therefore, minimum number of cycles $D_{P*}$ required by PLiM [82] to compute any MIG is 9#N, where #N is the number of nodes in MIG. This delay does not include the additional delay required for computing the negated valued of the nodes. For each benchmark, $\#C$ is significantly lower than the $D_{P*}$ achieved by the PLiM computer. This is a fair comparison since PLiM computer also used a word-length of 16-bits. The ReVAMP architecture outperforms PLiM computer for the same 16-bit word by a factor of 4.38× on average and 9.5× at the maximum. For the ReVAMP architecture, on average, almost 30% of the computation time is spent in computing negated value of the nodes. Thus, the ReVAMP architecture would further outperform the PLiM computer, when the actual number of cycles required by PLiM computer for computation with negations will be considered.

In Figure 5.14b, the speed up achieved by the ReVAMP architecture against the PLiM computer is presented for various word lengths. Even for a small word length of 4, the ReVAMP architecture gains in performance over the PLiM architecture by a factor of 2.9× on average. This shows that harnessing the inherent parallelism of ReRAM crossbar arrays for computation provides considerable performance gains. This justifies the VLIW nature of the ReVAMP architecture and demonstrates the effectiveness of the delay constrained mapping.

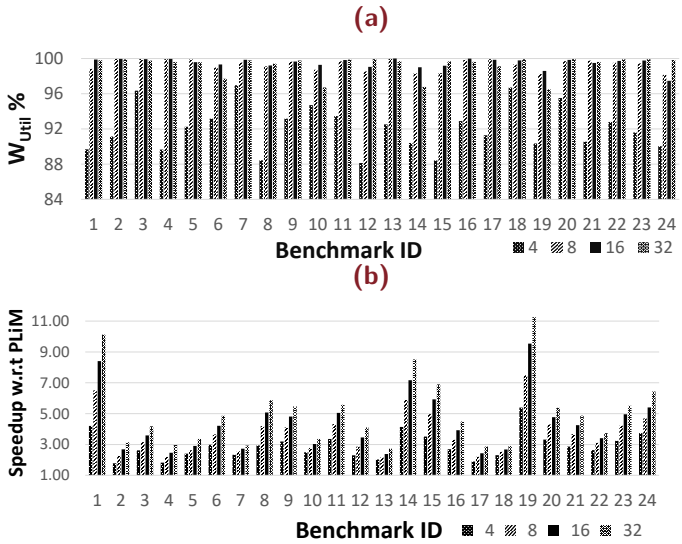



**Figure 5.14:** Impact of word length $w_D = \{4, 8, 16, 32\}$ on word utilization and delay for delay-constrained mapping. (a) Word utilization achieved by the delay constrained mapping. (b) Speedup achieved by the delay constrained mapping against PLiM.

## 5.5 Discussion

In this chapter, we presented two approaches to the technology mapping problem for logic-in-memory computation using ReVAMP. The area-constrained method allows high flexibility of mapping to a variety of crossbar dimensions while harnessing the available parallelism of the ReRAM crossbar array. The delay-constrained method reduces the overall delay of mapping by using a multi-step approach that takes into account crossbar constraints while placing the operands. The proposed approach outperforms the state-of-the-art serial logic-in-memory approach using ReRAMs. Specifically, the key contributions of this chapter are as follows.

- Any arbitrary AIG/MIG with $k$-levels can be mapped with $2(k + 1)$ devices, arranged as a crossbar with atleast two bitlines.
- Any Boolean expression, expressed as a Exclusive-Sum-Of-Product (ESOP), can be computed on a crossbar with three wordlines and atleast two bitlines.
- We present two technology mapping approaches for the ReVAMP platform.

- The area-constrained technology mapping approach uses And-Inverter Graph for logic representation and then uses a hierarchical method for generating ReVAMP instructions, aware of the crossbar dimensions. The method supports mapping to a wide variety of crossbar dimensions.
- The delay-constrained mapping approach relies on harnessing bit-level parallelism of the ReRAM crossbar array by maximizing parallel operations across multiple devices that share the same wordline. This method achieves significant lower delay compared to existing ReRAM-based serial logic-in-memory architecture.

The synthesis approaches used in the technology mapping flow, such as partitioning algorithm used for LUT mapping, are not aware of the crossbar constraints. The LUT partitioning algorithm should ideally try to partition the graph, so that each of the ESOP expression corresponding to each LUT has roughly the same number of cubes, instead of solely minimizing the number of LUTs covering the graph. Also, the initial representation of the Boolean functions into AIG/MIG are not explicitly optimized w.r.t to the quality of the resulting mapping. We believe optimizing the synthesis algorithms w.r.t to the crossbar constraints, would allow further reduction in delay of mapping, when combined with the proposed technology mapping approaches. This is the exact problem that is addressed in Chapter 6 that present technology-aware optimization techniques for logic synthesis to aid in improving the overall quality of mapping.

# 6 Technology-aware Logic Synthesis

In the previous chapters, we introduced ReVAMP — a logic-in-memory architecture based on a ReRAM crossbar arrays. We analyzed multiple variants of the technology mapping problem not only for 1S1R devices but also from the ReVAMP architecture. Recall, the design automation flow presented in Figure 4.1. So far, we have focused on technology mapping, without considering any technology specific optimization to the output of logic synthesis. This deficit in the logic synthesis process naturally leads to a sub-optimal computing solution, as far as the ReRAM crossbar arrays are concerned. While technology-aware logic synthesis has been done for Quantum-dot cellular automata in [127, 128], and for ReRAM device-level endurance management in the context of in-memory computing [129], to the best of our knowledge, none of these works have considered the opportunity to exploit bit-level parallelism available in crossbar arrays. In this chapter, we extend in this direction by proposing a novel logic synthesis flow that optimizes the logic network to harness the inherent parallelism of the ReRAM crossbar arrays. The key contributions of this chapter w.r.t logic-in-memory computing using crossbar arrays composed of 1S1R devices are the following.

- Novel logic synthesis techniques considering crossbar constraints, which can be applied, in principle, to any in-memory computing technology.
- Combined synthesis and technology mapping flow for harnessing the bit-level parallelism.
- Experimental study using representative ReVAMP as a representative crossbar architecture and complex benchmark functions.

## 6.1 Crossbar-aware MIG Optimization Methodology

The overall optimization flow is presented in Figure 6.1. The optimization techniques on MIGs that are tuned to crossbar constraints are described in detail in this section. The proposed optimization techniques directly improve the overall quality of results obtained via the technology mapping flow introduced in Chapter 5.

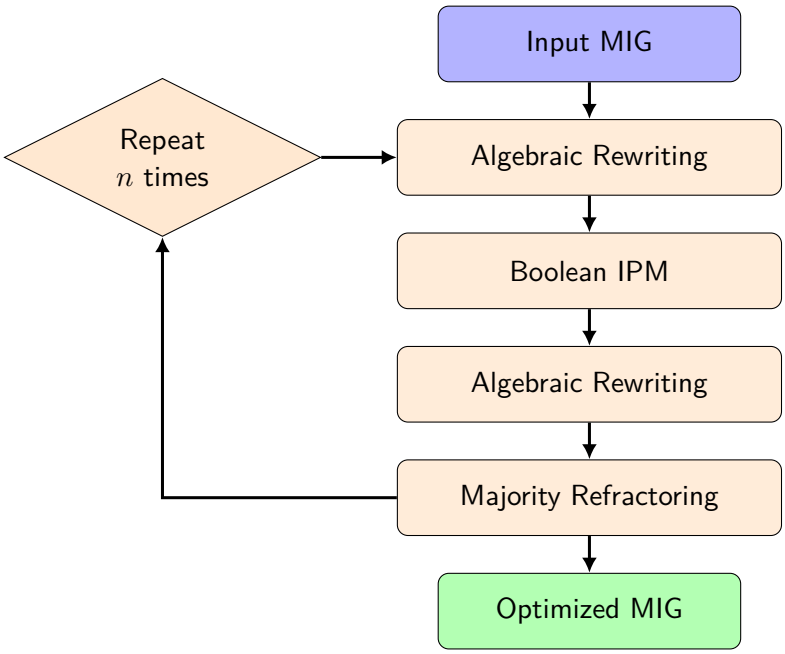



**Figure 6.1:** MIG optimization flow for crossbar-aware logic synthesis. Empirically, the number of iterations $n$ is limited to 10 or less till improvements are obtained.

### 6.1.1 MIG Algebraic Optimization

We are interested in compact MIG representations because they translate in smaller and faster physical implementations, especially in the ReRAM nanotechnology. In order to manipulate MIGs and reach advantageous MIG representations, a dedicated Boolean algebra was introduced in the original paper [103]. The axiomatic system for the MIG Boolean algebra, referred to as $\Omega$, is defined by the five following primitive transformation rules.

**Commutativity:** — $\Omega.C$

$$M(x, y, z) = M(y, x, z) = M(z, y, x)$$

**Majority:** — $\Omega.M$

$$if(x = y) : M(x, y, z) = x = y$$
$$if(x = \overline{y}) : M(x, y, z) = z$$

**Associativity:** — $\Omega.A$

$$M(x, u, M(y, u, z)) = M(z, u, M(y, u, x))$$

**Distributivity:** — $\Omega.D$

$$M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$$

**Inverter Propagation:** — $\Omega.I$

$$\overline{M}(x, y, z) = M(\overline{x}, \overline{y}, \overline{z})$$

Some of these axioms are inspired from median algebra [130, 131] and others from the properties

of the median operator in a distributive lattice [132]. A strong property of MIGs and their algebraic framework is about reachability. It has been proved that, by using a sequence of transformations drawn from Ω, it is possible to traverse the entire MIG representation space. Unfortunately, deriving a sequence of Ω transformations is an intractable problem. As for traditional logic optimization, heuristic techniques provide here fast solutions with reasonable quality. Details of the optimization algorithms targeting depth, size and power reductions in an MIG can be found in [103] .

### 6.1.2 MIG Boolean Optimization

MIG optimization techniques based on Ω rules are called *algebraic*. More powerful transformations are possible by using Boolean methods. For the purposes of this work, we focus on MIG Boolean optimization based on Input Partitioning Methods (IPM) and safe errors.

*Safe Errors:* MIGs are hierarchical majority voting systems. One notable property of majority voting is the ability to correct different types of bit-errors. This feature is inherited by MIGs, where the error masking property can be exploited for logic optimization. The idea is to purposely introduce logic errors that are succesively masked by the voting resilience in MIG nodes. For example, rewrite an MIG node $w$ as $w = M(wA, wB, wC)$, where $A$, $B$, $C$ are errors on node $w$. If such errors are advantageous, in terms of logic simplifications, better MIG representations can be generated. To be considered safe in an MIG, logic errors must be pairwise orthogonal.

*Input Partitioning Methods:* The rationale behind IPM for MIG is to select inputs leading to advantageous simplifications when erroneous. For this purpose, we use a decision metric, called dictatorship, to select the most profitable inputs for logic error insertion. The dictatorship is the ratio of input patterns over the total ($2^n$) for which the output assumes the same value than the selected input. For example, in the function $f = (a + b).c$, the inputs $a$ and $b$ have equal dictatorship of 5/8 while input $c$ has an higher dictatorship of 7/8. The inputs with the highest dictatorship are the ones where we want to insert logic errors. Indeed, they have the largest influence on the circuit functionality and its structure.

Exact computation of the dictatorship requires exhaustive simulation of an MIG structure, which is not feasible for practical reasons. Heuristic approaches to estimate dictatorship involve partial random simulation and graph techniques.

After exact or heuristic computation of the dictatorship, we select a subset of the primary inputs with highest dictatorship. Next, for each selected input, we determine a condition that causes an error. We require these errors to be orthogonal. Since we operate directly on the primary inputs, we already divide the Boolean space into disjoint subsets that are orthogonal. Because we need at least three errors, we need to consider at least three inputs to be made erroneous, say $x$, $y$ and $z$. A possible partition is the following: $\{x! = y, x = y = z, x = y = z'\}$. The corresponding errors are $A : x = y$ for $\{x! = y\}$, $B : z = y'$ when $x = y$ for $\{x = y = z\}$ and $C : z = y$ when $x = y$ for $\{x = y = z'\}$. We refer the reader to [103] for a formal proof on $A$, $B$ and $C$ orthogonality.

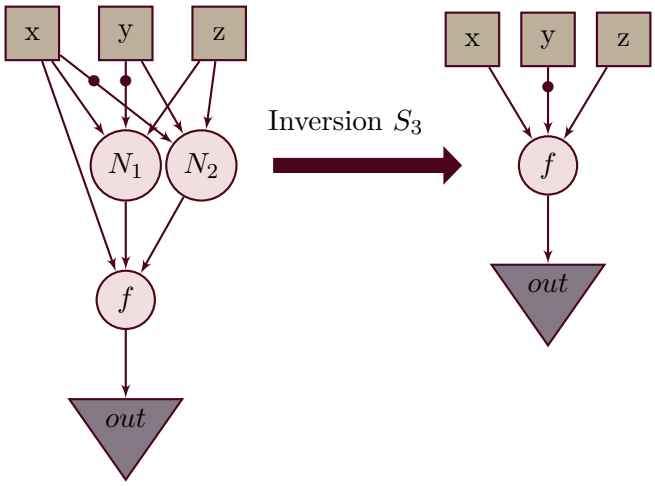



**Figure 6.2:** Input Partitioning Method for simplification of an example MIG.

We illustrate input partitioning method on a function $f = M(x, M(x, y', z), M(x', y, z))$. Corresponding to $f$, the input partition is $\{x! = y, x = y = z, x = y = z'\}$ which is affected by errors A, B and C, respectively. The first error A imposes $x = y$ which leads to $fA = M(x, M(y, y', z), M(x', x, z))$. This can be simplied by using $\Omega.M$ to $fA = M(x, z, z) = z$. The second error B imposes $z = y'$ when $x = y$. This is the case for the bottom level majority operators $M(x, y', z)$ and $M(x', y, z)$ which are

transparent when $x = y$. Therefore, error B leads to $fB = M(x, M(x, y\hat{a}\check{A}\acute{Z}, y'), M(x', y, y'))$ which can be further simplified to $fB = M(x, y', x') = y'$ by applying $\Omega.M$. The third error C imposes $z = y$ when $x = y$ holds. Analogously to error B, error C leads to $fC = M(x, M(x, y', y), M(x', y, y))$ which can be further simplified into $fC = M(x, x, y) = x$ by $\Omega.M$. A top majority node finally merges the three functions into $f = M(fA, fB, fC) = M(z, y', x)$ which correctly represents the objective function but has 2 fewer nodes and 1 level less than the original representation. This is depicted in Figure 6.2.

Safe error insertion in MIGs can be used for size reduction. In this case, the branch triplication overhead in $w = M(wA, wB, wC)$ imposes tight simplification requirements. We use the input partitioning method to focus on the most influent inputs of a MIG, and introduce selective simplification on them.

*Majority Refactoring:* An effective Boolean technique for MIG minimization is majority refactoring. The idea is to first compute a large cone of logic rooted at a MIG node. This cone of logic is transformed into a canonical logic representation, e.g., a BDD or a truth table. Starting from the canonical logic representation, a new local MIG is derived by means of decomposition. Novel techniques for majority decomposition have been recently proposed [108, 133], which can be used in conjunction with the state-of-the-art solutions [134]. Once the new local MIG is formed, its cost is calculated and compared to the original MIG, where the cone of logic was extracted. If an advantage in the cost metric used is seen, the refactored MIG is imported back to the global network. This procedure can be iterated for every node of the MIG in topological order, and then until an improvement is seen (or a maximum runtime limit is hit).

Now, we discuss the cost metrics used for MIG optimization. The number of nodes in an MIG is a cost metric of the number of computations needed. So, reduction of number of nodes in MIG is one of our synthesis objectives. In synthesis of traditional CMOS circuits, depth of the logic representation structure is an indication of the delay of the synthesized circuit. However, in case of mapping to ReRAMs, this might not hold true, since all the operations in a level of the logic networks might not

be computed in parallel under the crossbar constraints. In Figure 6.3, we show an MIG with $w$ as a common input for nodes $R_1$, $R_2$, $R_3$ and $R_4$. There is a predecessor $b_i$ for node $R_i$ which is connected with an inverted edge. Such an MIG can be computed in a single step if all the nodes $S_i$ are stored in a word and $w$ is applied as wordline input and corresponding $b_i$ as bitline inputs. Based on this observation, the rationale for crossbar-aware optimization is to: *first*, enforce logic sharing in the MIG already starting from the lower levels of the network, and *second*, share non-inverted edges, if multiple inverted-edges are shared, propagate the inverts above. With this background, we have tailored the previous MIG techniques for crossbar-aware logic optimization.

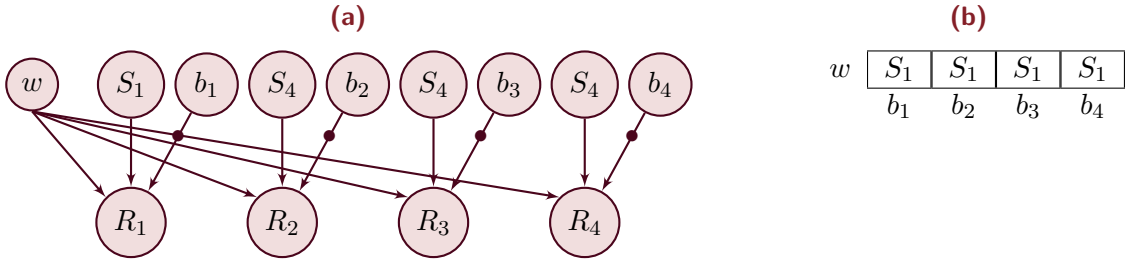



**Figure 6.3:** Leveraging common inputs in MIG to enable multiple computations in a cycle. (a) An MIG with a shared non-inverted input $w$. (b) $w$ can be used as a wordline input while $S_i$'s act as host for computation and the inverted inputs $b_i$ are used as bitline inputs.

We created a new MIG optimization flow, targeting crossbar-aware logic realizations, comprising of MIG algebraic rewriting, MIG Boolean IPM and MIG majority refactoring. All three techniques have been made aware of the optimization goals convenient to crossbar ReRAMs.

A high level execution flow is shown in Figure 6.1. The process is typically iterated less than 10 times ($n < 10$). In practice, we found that good size reductions, with controlled net count increase and no depth overhead, produce the most advantageous MIG representation for later mapping into the ReRAM array. In the next section, we present the results of benchmarking the proposed crossbar-aware end-to-end flow from MIG to generation of instructions.

## 6.2 Experimental Results For Crossbar-aware Logic Synthesis

The proposed end-to-end flow is implemented for the ReVAMP architecture. To evaluate our optimization techniques, we choose the depth-optimized hard EPFL benchmarks, available as MIGs [*]. The results are shown in Table 6.1 for DCM word length $w_D = 16$. We explain the results by means of a benchmark, say *adder*. The initial MIG has 3369 nodes along with 9333 edges while the optimized MIG has 2811 nodes along with 7662 edges. The reduction in number of nodes might not directly lead to reduction in delay of mapping. The quality of the MIG for mapping can be determined by the number of blocks required to group the nodes, and the number of words which are required to pack the blocks into words. For the *adder* benchmark, the optimized MIG requires 27.4% less blocks with 31% reduction in number of words. Finally, we can see that these optimizations lead to 39.6% reduction in overall delay of mapping.

**Table 6.1:** Synthesis results for hard EPFL benchmarks ($w_D = 16$)
$\delta = \frac{unoptimized - optimized}{unoptimized} \times 100$

| Benchmark | #Nodes | | #Edges | | #Blocks | | #Words | | Delay | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Unopt. | Opt. ($\delta$) | Unopt. | Opt. ($\delta$) | Unopt. | Opt. ($\delta$) | Unopt. | Opt. ($\delta$) | Unopt. | Opt.($\delta$) |
| adder | 3369 | 2811(16.6) | 9333 | 7662(17.9) | 669 | 486(27.4) | 255 | 176(31.0) | 12597 | 7603(39.6) |
| div | 76005 | 69478(8.6) | 227625 | 207834(8.7) | 12935 | 12286(5.0) | 5475 | 4945(9.7) | 303585 | 265795(12.4) |
| log2 | 37650 | 36359(3.4) | 112848 | 108963(3.4) | 7546 | 7330(2.9) | 2714 | 2617(3.6) | 129851 | 124916(3.8) |
| max | 7846 | 6108(22.2) | 21996 | 16500(25.0) | 1153 | 946(18.0) | 428 | 323(24.5) | 16896 | 11575(31.5) |
| mult | 42194 | 34615(18.0) | 126192 | 103458(18.0) | 9262 | 8824(4.7) | 3418 | 2667(22.0) | 139236 | 107886(22.5) |
| sin | 7946 | 7218(9.2) | 23760 | 21534(9.4) | 1509 | 1419(6.0) | 629 | 566(10.0) | 31075 | 27147(12.6) |
| sqrt | 52538 | 49553(5.7) | 157224 | 148182(5.8) | 9525 | 9211(3.3) | 3744 | 3525(5.8) | 208948 | 192524(7.9) |
| square | 19395 | 18838(2.9) | 57987 | 56178(3.1) | 5480 | 5560(-1.5) | 1593 | 1537(3.5) | 70678 | 68583(3.0) |

**Table 6.2:** Synthesis results for small EPFL benchmarks for various crossbar word length $w_D$.

| Benchmark | Nodes | | $w_D$=32 | | | | $w_D$=64 | | | | $w_D$=128 | | | | $w_D$=256 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Delay | | Words | | Delay | | Words | | Delay | | Words | | Delay | | Words | |
| | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. | Unopt. | Opt. |
| div16 | 4440 | 3875 | 14088 | 10764 | 164 | 126 | 11186 | 7815 | 81 | 62 | 8193 | 5196 | 40 | 31 | 5941 | 3811 | 20 | 16 |
| hamming | 2280 | 2080 | 6007 | 4598 | 87 | 72 | 4824 | 3495 | 44 | 36 | 3767 | 2708 | 22 | 19 | 3045 | 2175 | 12 | 10 |
| spi | 3889 | 3865 | 7696 | 4953 | 130 | 86 | 6501 | 4167 | 65 | 43 | 5729 | 3281 | 33 | 22 | 5004 | 2803 | 17 | 12 |
| sqrt32 | 2206 | 2082 | 7182 | 6293 | 82 | 72 | 5908 | 4971 | 41 | 36 | 4458 | 3685 | 21 | 18 | 3810 | 2699 | 11 | 10 |
| tv80 | 8176 | 8025 | 21305 | 17089 | 283 | 224 | 18249 | 14087 | 143 | 113 | 16007 | 10916 | 72 | 57 | 13237 | 8553 | 36 | 28 |

In general, we have managed to reduce the number of nodes as well as edges for all the benchmarks. This in turn has lead to reduction in the number of blocks, thereby leading to the reduction of the

[*]http://lsi.epfl.ch/benchmark

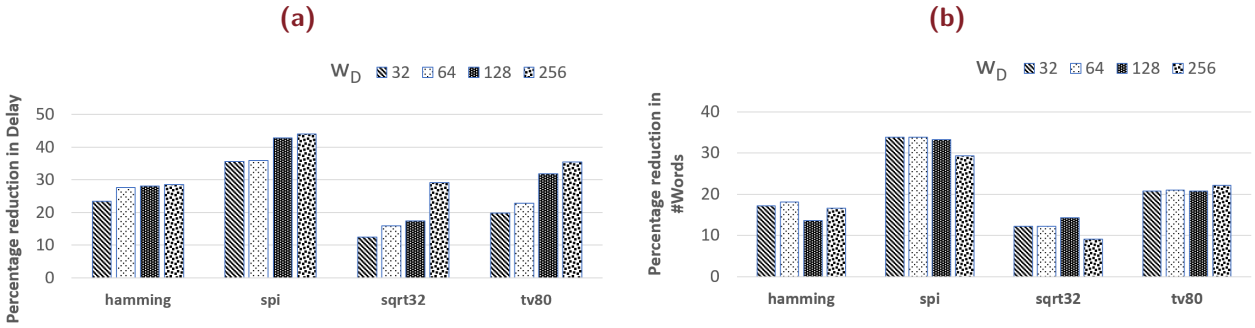



**Figure 6.4:** Impact of crossbar dimensions on the (a) delay of mapping (b) mumber of words required to map.

number of words required for the mapping. On average, we have been able to reduce the number of nodes by 10.8% (maximum reduction of 16.56% for adder) along with the average overall delay reduced by 16.67% (maximum reduction of 39.64% for adder). In addition, our experiments show that the packing acheives more than 99% device utlization. This clearly demonstrates the overall efficiency of our crossbar aware MIG optimization in reducing both the number of devices required as well as mapping delay.

The word length of the DCM determines the number of parallel operations that can be performed. Therefore, we analyse the impact of word length on our flow using another set of EPFL benchmarks. The results of mapping on varying word length $w_D$ from 32 to 256 is shown in Table 6.2. With increase in word length, the overall mapping delay reduces for all the benchmarks, showing the effectiveness of the compilation flow to harness the parallelism offered by the DCM. Also for these benchmarks, our MIG optimization method reduces the number of nodes in the MIG. Figure 6.4a shows the percentage reduction in delay for the optimized MIGs compared to the base MIGs. With increase in word length, our optimized MIGs are better suited for mapping to the crossbar compared to the base MIGs. Figure 6.4b shows the percentage reduction in number of words compared to base MIGs. For all the word lengths, our optimized MIGs always require fewer number of words compared to the base MIGs.

## 6.3 Discussion

In-memory computing fabrics are getting major research attention nowadays, thanks to inherent advantage of tremendous boost in data locality. Traditional design automation flows are not designed for such computing architectures and therefore, poses a challenge towards adoption. Recent works in this direction have focused on logic synthesis and technology mapping with the capabilities of individual devices in mind. In this chapter, we propose novel technology-aware logic synthesis optimizations, which, for the first time, enables one to exploit the bit-level parallelism available in ReRAM crossbar arrays. Our results show that it improves the area and delay metrics compared to a crossbar-agnostic design automation flow.

# Part II:

# Facilitating MVL Using Multi-State ReRAM Array

# 7 MVL Realization Using $TaO_x$ Devices & Addressing the MVL Synthesis Challenges

Claude Shannon, in his landmark work [135], demonstrated that the two-valued logic system developed by George Boole [136], can be mimicked through operations of an electrical circuit. This resulted in widespread adoption of two-valued switching algebra or Boolean algebra. For every digital device in the modern world, Boolean algebra is used to perform the underlying computation. Boolean logic uses two truth values, *true* and *false*, even though in real life, we often require more than two truth values for describing an event. For example, we describe a day as 'sunny', 'partly clouded' or 'clouded' , which means that the element weather cannot be discretely classified into the set 'sunny' or 'clouded' To capture such phenomena and their underlying logical process, *multi-valued* logic system was introduced.

In 300 BC, Aristotle proposed the *principle of non-contradiction* which ruled out simultaneous existence of two contradictory propositions [137]. This principle, also known as the *law of excluded middle*, is one of the classical laws of thought. In modern times, Jan Łukasiewicz introduced a third truth value, thereby first formally studying the field of multi-valued logic (MVL) in 1920 [138]. In the following year, Emil Post published a system of functionally complete MVL algebra with additional

truth degrees ($n \geq 2$), 'chained Post algebra' [139]. Later on, finite-valued Łukasiewicz logic was rigorously axiomatized and extended to arbitrarily many truth values [140, 141]. Applications of MVL can be found in linguistics [142], circuit simulation and manufacturing testing of digital circuits [143].

Despite these wide ranging applications of MVL, the present computing technology is heavily based on Boolean logic. There have been prior studies on porting MVL to digital and analog circuits with promising results. MVL has been demonstrated to improve energy-efficiency by reducing the switching activity of VLSI interconnects [144, 145]. MVL based arithmetic circuits are simpler and more efficient over corresponding Boolean logic based implementations [146–148]. A limiting factor of MVL realization has been the inherent representation of information in binary format in semiconductor devices, thereby forcing a designer to switch between logic formats, which was clearly an inefficient solution. In past, Łukasiewicz logic arrays has been demonstrated for realization of fuzzy inference engines and expert systems [149–151] with CMOS-based circuitry. However, such realizations did not have any multi-valued storage devices for storing the intermediate results thereby requiring costly conversions to-and-from binary representation.

In this chapter, we enable the usage of multi-state memristive devices natively in the multi-valued domain. The multi-state memristive devices used in this experiment are Redox-based resisitve switches (ReRAMs), which are considered as one of the most promising emerging non-volatile memory technologies [34, 87, 152]. Subsequently, we look at the problem of synthesis of multi-valued combinational functions. The synthesis of multi-valued functions consists of bi-decomposition or functional decomposition of the given target function to obtain a multilevel network comprising of min and max gates. Synthesis tools, such as YADE and those based on Multiple-Valued Decision Diagrams make the implicit assumption regarding the availability of literals or CASE operator, while focusing on the optimization of the logic network solely based on the MIN and MAX gates. However, a literal cannot be assumed to exist as a primitive in a multi-valued logic system and therefore, renders it difficult for one to directly apply the existing synthesis flows in practical settings, such as MVL realization using multi-state ReRAMs.

An important offshoot of MVL that permits *inference under vagueness* and allows real-valued member elements is fuzzy logic. The term *fuzzy logic* was introduced by Lotfi A. Zadeh in the context of fuzzy set theory [153]. In contrast to the classical logic systems that adheres to a set of elements with *crisp* truth values, fuzzy logic operates on fuzzy sets. In a fuzzy set, elements of the set can have a degree of membership. Operators from a MVL, e.g., Łukasiewicz logic, can be applied to the fuzzy set, akin to how Boolean logic operators are applied to the crisp set. In fuzzy logic, a linguistic model is built from a set of IF−THEN rules which describe the control model. Mamdani Warren demonstrated that fuzzy logic could be used for developing operational automated control systems [154] and clinical practice decision support systems [155]. The most well known application of fuzzy logic based control system was deployed in Sendai Subway 1000 series subway trains in Japan for speed control [156]. The fuzzy controller based train had a higher relative smoothness of the starts and stops when compared to other trains, and was 10% more energy efficient than human-controlled accelerated trains. Further applications of fuzzy logic include expert systems [157], robotics [158] and diverse sub-domains of machine intelligence [159]. In this chapter, we demonstrate how Łukasiewicz logic operators can be used for realizing a fuzzy system. We experimentally show for the first time how fuzzy inference can be performed using multi-state memristive devices. The proposed approach does not use any intermediate binary/Boolean representation. Recently, an implementation of Boolean minimum and maximum gate has been demonstrated using memristive devices [160] with their application restricted to the implementation of sorting networks.

The contributions of this chapter can be summarized as follows.

- We present a novel native implementation of Łukasiewicz logic using the multi-state ReRAM devices. We do not impose any theoretical limit on the number of states for the memristive devices [161] and hence this work can be used for realizing any application that uses finite-valued Łukasiewicz logic family $L_n$.
- We present a novel MVL synthesis flow with Łukasiewicz logic primitives as the target operators. We derive literals and CASE operators using these primitives, and propose a heuristic algorithm

for automated synthesis.

- Our experimental studies on a wide range of benchmarks reveal that an average overhead of 216% in terms of number og implication gates, along with 55% increase in the number of levels is encountered, in contrast to a synthesis flow that assumes existence of literals (single input mult).

- We also present a detailed case study to realize fuzzy logic control using Łukasiewicz logic and the implementation of a fuzzy logic controller using multi-state ReRAM crossbar arrays.

## 7.1 Preliminaries Of Łukasiewicz Logic

Łukasiewicz logic is multi-valued logic system defined by Jan Łukasiewicz [162]. He first proposed the ternary logic $Ł_3$ on the set of truth values $\{0, \frac{1}{2}, 1\}$, then extended it to $N$-valued logic $Ł_N$ and infinite-valued logic $Ł_\infty$. The truth values $W_m^N$ of an $N$-valued Łukasiewicz logic are defined as:

$$W_m^N = \frac{m}{N-1} \quad | \quad 0 \leq m \leq N-1 \tag{7.1}$$

The set of all truth values $W_m^N$ is denoted as $W^N$. The original system of Łukasiewicz logic used implication and negation as the primary connectives. For multi-valued operands $x$ and $y$, $x, y \in L_N$, the operators implication IMP ($\rightarrow$) and negation NEG ($\neg$) are defined as

$$u \rightarrow v = \min(1, 1 - u + v) \tag{7.2}$$

$$\neg u = 1 - u \tag{7.3}$$

MAX($\vee$) and MIN($\wedge$) are standard connectives in Łukasiewicz logic. The output of MAX (MIN) operator is equal to the maximal (minimal) value of the inputs. The MAX and MIN operators are

defined using implications and negations as

$$u \vee v = (u \to v) \to v \tag{7.4}$$

$$u \wedge v = \neg(\neg u \vee \neg v) \tag{7.5}$$

Furthermore, axioms of MIN and MAX operations can be directly defined as:

$$(u \wedge v) \vee (u \wedge z) = u \wedge (v \vee w) \tag{7.6}$$

$$(u \vee v \vee w) = (u \vee (v \vee w)) \tag{7.7}$$

$$(u \vee v) \wedge (u \vee w) = u \vee (v \wedge w) \tag{7.8}$$

$$(u \wedge v \wedge w) = (u \wedge (v \wedge w)) \tag{7.9}$$

$$0 \wedge u = 0 \tag{7.10}$$

$$1 \vee u = 1 \tag{7.11}$$

$$0 \vee u = u \tag{7.12}$$

$$1 \wedge u = x \tag{7.13}$$

Now we define some of the basic concepts of multi-valued logic.

**Definition 7.1** (cardinality of a variable)**.** *Let $x$ be a $N$-valued variable ($x \in W^N$), where the integer $N \geq 2$ is called the cardinality of the variable $x$. A set $X = \{x_1, x_2, ..., x_p\}$ of MVL variables, with cardinalities $N_1, N_2, ...N_p$ respectively, can assume any element of*

$$\mathbb{N}_X = W^{N_1} \times W^{N_2} \times ... \times W^{N_p}$$

*where $\times$ denotes the Cartesian product of sets. The set $\mathbb{N}_X$ is the MVL space for the set of variables $X$. A MVL minterm is an element of $\mathbb{N}_X$.*

**Definition 7.2** (MVL function)**.** *A MVL function $f(X)$ is a many-to-one mapping (n-ary function)*

$$f(X) : \mathbb{N}_X \to W^{N_o}$$

*from a set of minterms $\mathbb{N}_X$ onto the set of truth values $W^{N_o}$, where $N_o$ is the output cardinality of the function.*

**Definition 7.3** (multi-valued literal)**.** *A multi valued literal ($L_x$) is a unary multi valued function $f(x)$, where $x$ is a $N$-valued variable, which is a one-to-one mapping*

$$L_x = f(x) : W^N \rightarrow W^{N_o}$$

*from a set of truth values $W^N$ onto a set of truth values $W^{N_o}$, where $N$ and $N_o$ are the input and output cardinalites of the literal.*

**Example 7.1.** *Consider a literal $L_x = \{0, 0.2, 0.2, 1\}$ with input $x$, having input and output cardinalities 4 and 6 respectively. $L_x$ is a one to one mapping from the truth values $\{0, 0.33, 0.66, 1\}$ of $x$ to $\{0, 0.2, 0.2, 1\}$ respectively.*

**Definition 7.4** (delta literal)**.** *A delta literal($\Delta_i^{N,x}$) is a multi valued literal which is defined as,*

$$\Delta_i^{N,x} = \begin{cases} 1, & if x = \frac{i}{N-1} \\ 0, & otherwise \end{cases}$$

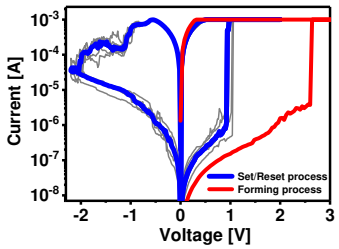
*where $x$ is a $N$-valued variable and $0 \leq i < N$. There are $N$ delta literals for an $N$-valued variable. The set of all $N$ delta literals is denoted as $\Delta^{N,x}$.*

**Example 7.2.** *Consider a 4-valued variable $x$. The delta literals for the variable $x$ are $\Delta_0^{N,x} = \{1, 0, 0, 0\}, \Delta_1^{N,x} = \{0, 1, 0, 0\}, \Delta_2^{N,x} = \{0, 0, 1, 0\}$ and $\Delta_3^{N,x} = \{0, 0, 0, 1\}$ for the inputs $x = \{0, 0.33, 0.66, 1\}$.*

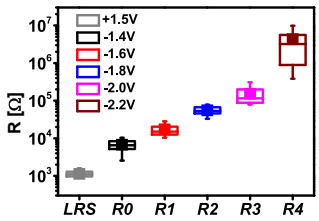
## 7.2 Łukasiewicz Logic Realization

Without loss of generality, we consider the Łukasiewicz logic family $L_3$ (or three-valued Łukasiewicz logic family) for realization. The details of the device fabrication and measurement setup is presented

(a)




(b)




(c)

| States | Δ [V] | R [Ω] |
|---|---|---|
| LRS | DC | 1.1k |
| R0 | -1.4 | 6.5k |
| R1 | -1.6 | 15.0k |
| R2 | -1.8 | 53.1k |
| R3 | -2.0 | 115.5k |
| R4 | -2.2 | 3.2M |

**Figure 7.1:** Device characteristics. (a) Current−Voltage ($I-V$) characteristics of $TaO_x$ based ReRAM device. (b) Resistance distribution based on median, obtained by pulse duration of $200ns$ and amplitude in the range of $-1.4V$ to $-2.2V$ ($0.2V$ steps) enable highly accurate resistive state control. (c) Mean value of final resistance is estimated based on measurements of $4$ devices for $10$ cycles per state.

in Appendix B. Figure 7.1 presents the characteristics of the multi-state ReRAM devices. The three resistance states (R0, R1, R2) of the multi-level device are used to represent to logic values (0, 0.5, 1) respectively in the Łukasiewicz logic family $L_3$. Additionally, the intermediate results in $L_3$ can be (1.5 or 2) which require two additional logic values. The resistance states (R3, R4) represent the intermediate logic values (1.5, 2) respectively. Corresponding to the logic values $u = \{0,0.5,1,1.5,2\}$, the operand voltage $V_u$ is $\{0, 0.2, 0.4, 0.6, 0.8\}V$ respectively. The multi-valued operands say $u$ and $v$ can be applied as operand voltages to the top electrode (TE) and the bottom electrode (BE). Note that, $u$ and $v$ are always from the multi-valed logic set $L_3$ {0, 0.5, 1}. A predefined OFFSET voltage $V_{OFFSET}$ is used for each pulse to allow equidistant voltage stepping. The voltage applied to the TE is $V_{TE} = -(V_{OFFSET} + V_u)$. Depending on the operation being realized, the actual voltage applied to the BE is $V_{BE} = V_{OFFSET} \pm V_v$. The effective potential difference across the device is $V_{eff} = V_{TE} - V_{BE}$. If $V_{BE} = V_{OFFSET} + V_v$, the resulting resistance state of the device is $R_{u+v}$. Otherwise if $V_{BE} = V_{OFFSET} - V_v$, the resulting resistance state of the device is $R_{u-v}$. Figure 7.2 demonstrates the multi-level operation of the device.

**Proposed Łukasiewicz Logic Implementation:** The developed implementation strategy for realization of the Łukasiewicz logic operates and stores the multi-valued results directly in the ReRAM devices. The computed results are available as the resistive states of the device and can be read out. Before any operation, the devices are initialized to LRS. The realization of the NEG $\neg$ and IMP $\rightarrow$ operator

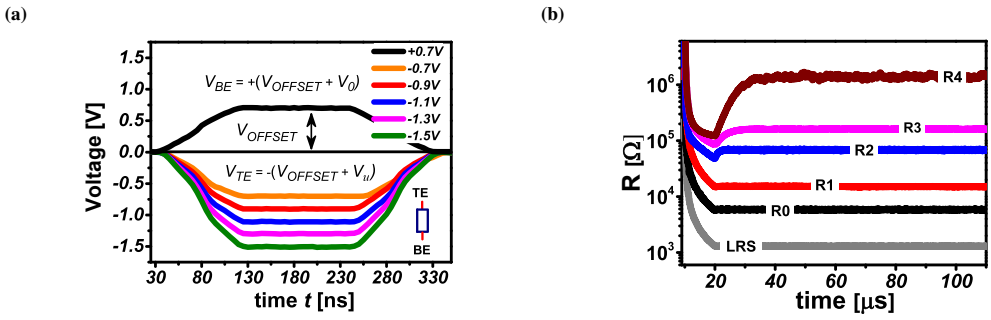



**Figure 7.2:** Primary MVL operation. The logic operands $u$ and $v$ are applied to top (TE) and bottom (BE) electrode, respectively. Operand voltages $V_u$ and $V_v$ range from $0V$ to $0.8V$ in steps of $0.2V$. Equal stepping of operand voltages is enabled using an OFFSET voltage $V_{OFFSET} = 0.7V$. (a) Keeping the $V_{BE} = 0.7V$ constant, $V_{TE}$ is varied from $-0.7V$ to $-1.5V$ in steps of $-0.2V$ i.e. $V_v$=$0V$ and $V_u = 0, 0.2, \ldots, 0.8V$. (b) The corresponding resistance levels $R0, \ldots, R4$ states are programmed to the device. The actual resistive states are read by means of a $120\mu s$ long voltage pulse with $0.1V$ ($V_{READ}$ ) amplitude.

for $L_3$ is explained below.

**NEG $\neg$ operator:** The negation operator works on a single operand. For computing $\neg u$, a constant voltage $-(V_{OFFSET} + V_1)$ is applied to the TE while the $V_{OFFSET} - V_u$ is applied to the BE. The negated operand is stored in the ReRAM device as a corresponding resistive state.

**IMP $\rightarrow$ operator:** The implication operator works on two operands. The flowchart for performing the implication operation $u \rightarrow v$ is shown in Figure 7.3a and the steps are described below.

**Step 1:** In the beginning, all the devices are initialized to the LRS.

**Step 2:** To compute $\neg u$ in device $D_0$, $-(V_{OFFSET} + V_1)$ is applied to the TE and $V_{OFFSET} - V_u$ to the BE.

**Step 3:** The resistive state of the device $D_0$ is read out by means of $V_{READ} = 0.1V$.

**Step 4:** In the second device $D_1$, the negated sum of offset voltage $V_{OFFSET}$ and the voltage corresponding to read out value $V_{\neg u}$ is applied to the TE i.e $V_{TE} = -(V_{OFFSET} + V_{\neg u})$ whereas at the BE, $V_{BE} = V_{OFFSET} + V_v$ is applied.

**Step 5:** In the following step, resistive state $R$ of the $D_1$ is read out. If the resistive state $R < R2$,

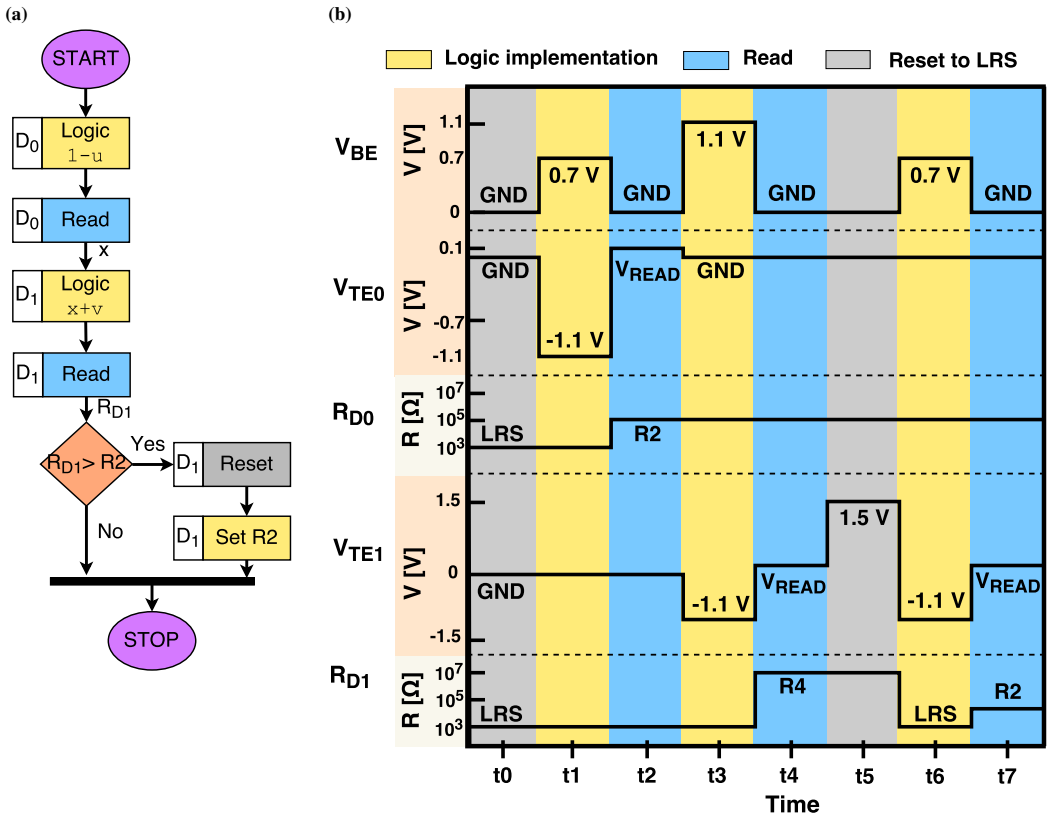



**Figure 7.3:** Implication implementation. (a) Flowchart for the implication ($u \rightarrow v = min(1, 1-u+v)$) computation. First, the negation of operand $u$, i.e. $1-u$, is computed. Next, the logic operation is conducted to compute $1-u+v$ in device $D_1$. The device state $R$ is read out from the $D_1$. To compute $min(1, 1+u-v)$, we check whether $R > R2$ or not. The comparison can be implemented by means of a comparator that would check the if the read out current of the device $I_{READ}$ is greater than a constant $I_{R2}$. If $R > R2$, then the device $D_1$ is reset to the LRS and set to resistive state $R2$. (b) Implication computation $u \rightarrow v$ for $u = 0$ and $v = 1$. $V_{BE}$ is the voltage sequence applied to the BE of the devices whereas $V_{TE0}$ and $V_{TE1}$ is the voltage applied to the TE of device $D_0$ and $D_1$ respectively. The transition of resistive state for device $D_0$ and $D_1$ is shown as $R_{D0}$ and $R_{D1}$ respectively. After the computation, the final resistive state of device $D_1$ is the result of $u \rightarrow v$, which in this case is equal to $R2$, corresponding to logic 1.

then the operation is complete. Otherwise, device $D_1$ is toggled to the LRS and the set operation is used to change the device state to $R2$, i.e., $V_{TE} = -1.1V$ and $V_{BE} = 0.7V$ is applied.

Figure 7.3b demonstrates the computation of $0 \rightarrow 1$ using the proposed method in terms of applied operation voltages and corresponding states. The overall operation requires seven steps. Initially, the device $D_0$ and $D_1$ are in LRS state, shown as $R_{D0}$ and $R_{D1}$ respectively. In cycle $t1$, $1-u$ is computed by applying $0.7V$ to the BE and $1.1-V$ to the TE of device $D_0$, shown as $V_{BE}$ and $V_{TE0}$ respectively. In cycle $t2$, the resistive state of $D_0$ is read out, which is $R2$. In the next cycle, $1-u+v$ is computed by applying $1.1V$ and $-1.1$ to the BE and TE of device $D_1$. In cycle $t4$, the current resistive state($R_{D1}$) of device $D_1$ is read out. Since $R_{D1} = R4 > R2$, device $D_1$ is reset to the LRS and then set to $R2$ in cycles $t5$ and $t6$. The computation of $0 \rightarrow 1$ is complete. To verify the correctness of computation, we read out the state of the device $D_2$ in the last cycle $t7$. The state $R_{D2}$ is $R2$ (corresponding to logic 1) which is the correct result for $0 \rightarrow 1$.

## 7.3 Analysis Of Proposed Native MVL Computation Technique

We have successfully demonstrated Łukasiewicz logic operation on $2\mu m \times 2\mu m$ ReRAM devices. However, these devices are fully compatible to $4F^2$ configuration in crossbar array in conjunction with a selector device and can be scaled down to $5nm$ [39, 163].The integration of the selector device would prevent the problem of sneak paths in the crossbar array. Multi-level ReRAM devices reduces the complexity of state representation and thus, brings fundamental benefits across arithmetic and logical primitives.

Regarding the representation of truth values, it is well understood that for higher radix, the number of literals reduce in logarithmic order in comparison to lower-radix. For example, the efficiency of a $n$-valued representation of a truth-value $N$ compared to its corresponding Boolean representation is

equal to $\frac{log_n(N)+1}{log_2(N)+1}$. Our demonstrated method can be scaled up for arbitrary $n$-valued Łukasiewicz logic $L_3$, $n \geq 3$, depending on the number of resistive states available. For the realization of $L_n$, the memristive device should support at least $2n$ states. From the perspective of area, the implementation of a higher-valued logic system does not increase the area per device since it is dependent on the number of resistive states. However with increase in number of resistive states, the peripheral circuitry has to be more robust.

Each multi-valued operation requires a constant number of steps, 1 step for negation and 7 steps for implication, to be realized, irrespective of the value of $n$ in an $n$-valued logic system. For Boolean realization of the implication and negation operators, the number of steps would increase with the value of $n$ [17, 62, 79]. Furthermore, parallel operations across multiple devices that share the same wordline, can be enabled by carefully packing operations that have the same input, similar to the strategy proposed by Bhattacharjee et al. [164] In contrast, to leverage such parallelism, the Boolean circuits corresponding to the implication and negation operations need to be replicated.

## 7.4 MVL Synthesis

Multi-valued logic synthesis is the process of converting a high-level description of a multi-valued function into a network of multi-valued gates. One of the well-studied approaches for MVL synthesis is bi-decomposition [165, 166]. Bi-decomposition decomposes a given function $f(I_1, I_2, I_3)$ into two decomposition functions $g(I_1, I_3)$ and $h(I_2, I_3)$ that are combined by an operator function $\phi(x, y)$ such that

$$f(I_1, I_2, I_3) = \phi(g(I_1, I_3), h(I_2, I_3)) \tag{7.14}$$

where both decomposition functions are dependent on fewer variables than the original function. Another approach to synthesize a MVL function is by using a MV Decision Diagram (MDD) and a MV-CASE operator [167, 168]. The existing tools assume the presence of cost-free operators (such as multi-valued select/CASE operator, etc), that are not native to the logic family, during MVL synthesis.

We study this critical assumption by presenting an elaborate algorithm for synthesizing these operators in terms of native MVL primitives and show that the operators contribute a significant overhead to the overall synthesis results.

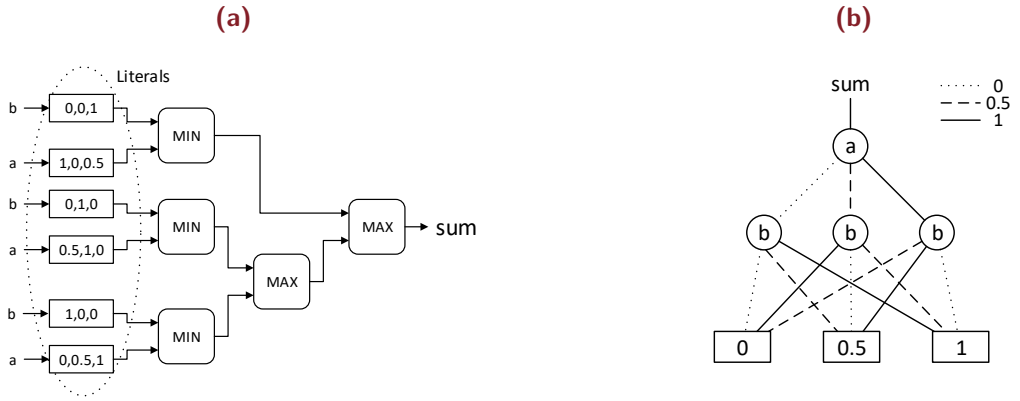



**Figure 7.4:** (a) Decomposition of a 3-valued sum function in terms of literals, MIN and MAX functions. (b) The 3-valued sum function represented as a MDD.

**Example 7.3.** *Figure 7.4a shows the result of decomposition of sum function for a 3-valued addition operation for operands a and b in terms of literals, MIN and MAX gates. The sum function can also be expressed as a MDD, shown in Figure 7.4b.*

## 7.5 MVL Multiplexer Synthesis

The MV-CASE operator can be interpreted as a multi-valued multiplexer. In this section, we outline an algorithm to synthesize a multi-valued $N : 1$ multiplexer using Łukasiewicz logic, as shown in Figure 7.5a. The synthesis is divided into two parts. First, the multiplexer is synthesized using delta literals, MIN and MAX gates described in section 7.5.1. Second, the delta literals are synthesized using implication and negation operators in section 7.5.2. The MIN and MAX gates obtained are represented using implication and negation. Finally, the depth and gate count of the synthesis is analyzed in section 7.5.3. The synthesis of a $4 : 1$ multiplexer, shown in Figure 7.5b is used as a running example.

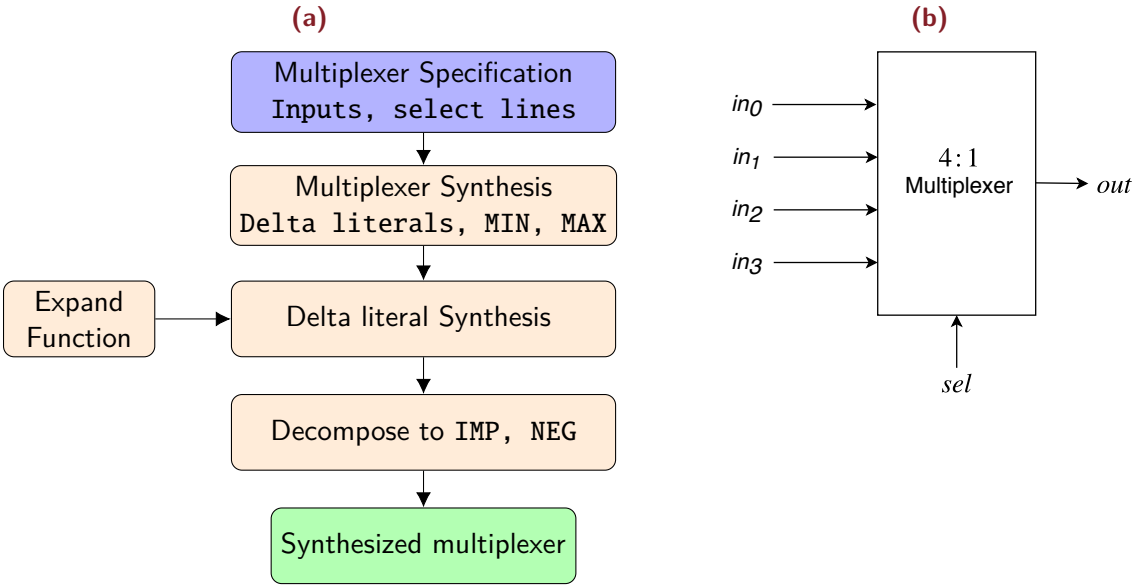



**Figure 7.5:** (a) A generalized multi-valued multiplexer synthesis flow. (b) Schematic of a $4:1$ multiplexer.

### 7.5.1 Synthesis Of Multiplexer

**Definition 7.5** (Multi-valued multiplexer)**.** *A $N:1$ multiplexer is a device with N-valued select line (sel), which forwards one of the $N$ inputs$\{in_0, in_1, ..., in_{N-1}\}$ to the single output (out), depending on the value of the select line. $N$ is also the cardinality of the multiplexer. The output (out) of the multiplexer is defined as:*

$$out = in_i \qquad where\ i = sel * (N-1) \tag{7.15}$$

**Theorem 7.1.** *A N-valued multiplexer can be composed as a $(\lceil \log_2 N \rceil + 1)$ level network of $N$ delta literals, $N$ MIN gates and $N-1$ MAX gates.*

*Proof.* Let $0 \leq i < N$. For each input $in_i$ of the multiplexer with select line $sel$, a minimum of $in_i$ and delta literal $\Delta_i$ is determined. By using the Axioms 7.10 and 7.13 of minimum and maximum

operations, the output $Min_i$ of the minimum is

$$Min_i = (in_i \wedge \Delta_i) = \begin{cases} in_i, & \text{if } sel = \frac{i}{N-1} \\ 0, & \text{otherwise} \end{cases} \tag{7.16}$$

The output of the multiplexer ($out$) is the maximum of all minimums $Min_i$.

$$\begin{aligned} out &= (Min_0 \vee Min_1 \vee ... \vee Min_{N-1}) \\ &= in_i \qquad where, i = sel * (N-1) \end{aligned} \tag{7.17}$$

Hence, $N$ literals and $N$ MIN gates are used. A parallel reduction tree algorithm [169] is used to find the maximum of all $Min_i (0 \leq i < N)$. This requires $(N-1)$ max gates and $\left\lceil log_2^N \right\rceil$ levels of MAX gates. Algorithm 9 describes the synthesis algorithm of a multiplexer using delta literals, MIN and MAX gates. ■

**Algorithm 9:** Synthesis of $N$-input multiplexer using delta literals, MIN and MAX gates.

```
Procedure DeviceReassign(N)
    for i = 0; i < N; i++ do
        Min_i = (in_i ∧ Δ_i) ;
    out = (Min_0 ∨ Min_1 ∨ ... ∨ Min_{N-1}) by parallel reduction tree algorithm ;
```

**Example 7.4.** *Figure 7.6a shows the synthesis of a* $4:1$ *multiplexer using the delta literals, MIN and MAX gates. The synthesis requires* 4 *delta literals,* 4 *MIN gates and* 3 *MAX gates.*

### 7.5.2 Synthesis Of Delta Literals

Algorithm 10 describes a method to synthesize all the delta literals $\Delta_i^{sel}$ $(0 \leq i < N)$ also referred to as $\Delta_i$. The synthesis is carried out in two steps. First, three levels of implications are carried out with

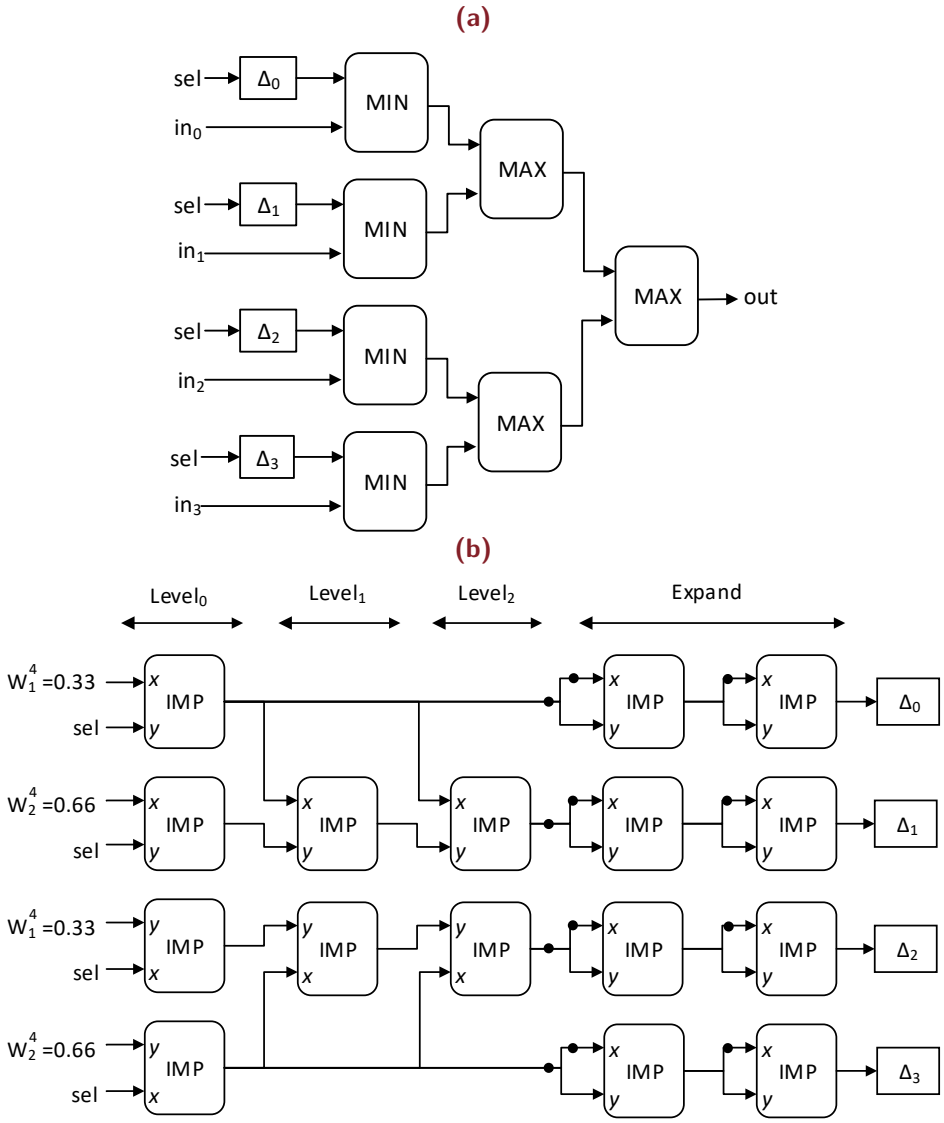



**Figure 7.6:** (a) Synthesis of multiplexer using delta literals, MIN and MAX gates. (b) Synthesis of Delta literals $\Delta^{N,sel}$. For simplicity of representation, delta literals $\Delta_i^{N,sel}$ are denoted as $\Delta_i$. The inverters are not optimized to maintain congruence with Algorithms 10 and 11.

outputs $Level_i^j$ (output $i$ of level $j$). The inverted output $\neg Level_i^2$ is defined as:

$$\neg Level_i^2 = \begin{cases} W_1^N, & \text{if } sel = W_i^N \\ 0, & \text{otherwise} \end{cases} \tag{7.18}$$

The delta literals can be expressed as a function of $\neg Level_i^2$ as $\Delta_i = E(\neg Level_i^2)$. The function $E(x)$ is the Expand function defined as:

$$E(x) = \begin{cases} W_{N-1}^N = 1, & \text{if } x > 0 \\ 0, & otherwise \end{cases} \tag{7.19}$$

Algorithm 11 shows the implementation of the Expand Function $E(x)$ using a series of $\lceil log_2(N-1) \rceil$ implication gates. Each implication in the algorithm doubles the value at the input.

**Algorithm 10:** *Delta Literal Synthesis*

**Procedure** `DeviceReassign(`$N$`)`
  **for** $i = 0$ ; $i < N$ ; $i++$ **do**
    **if** $i < \left\lceil \frac{N}{2} \right\rceil$ **then**
      $Level_i^0 = (W_{i+1}^N \rightarrow sel)$ ;
    **else**
      $ni = \left\lceil \frac{N}{2} \right\rceil - (N - 1 - i)$ ;
      $Level_i^0 = (sel \rightarrow W_{ni}^N)$ ;
  **for** $j = 1; j \leq 2; j++$ **do**
    $Level_0^j = Level_0^{j-1}$ ;
    $Level_{N-1}^j = Level_{N-1}^{j-1}$ ;
    **for** $i = 1; i < (N-1); i++$ **do**
      **if** $i < \left\lceil \frac{N}{2} \right\rceil$ **then**
        $Level_i^j = (Level_{i-1}^{j-1} \rightarrow Level_i^{j-1})$ ;
      **else**
        $Level_i^j = (Level_{i+1}^{j-1} \rightarrow Level_i^{j-1})$;
  **for** $i = 0; i < N; i++$ **do**
    $\Delta_i =$ `Expand(`$\neg Level_i^2$`)`;

**Algorithm 11:** *Expand Function*

```
1 Procedure Expand(x)
2     Level^0 = x ;
3     num = ⌈log_2(N − 1)⌉ ;
4     for i = 1; i ≤ num; i++ do
5         Level^i = (¬Level^{i−1} → Level^{i−1});
6     return Level^{num};
```

**Example 7.5.** *Figure 7.6b shows the synthesis of delta literals* $\Delta^{N,sel}$. *The output of* $Level_0^2$, $Level_1^2$, $Level_2^2$ *and* $Level_3^2$ *are* $\{0.33, 0, 0, 0\}$, $\{0, 0.33, 0, 0\}$, $\{0, 0, 0.33, 0\}$ *and* $\{0, 0, 0, 0.33\}$ *respectively, for* $sel = \{0, 0.33, 0.66, 1\}$. *The expand function increases the value of* 0.33 *in the above outputs to* 1 *by a series of* 2 *implications.*

### 7.5.3 Depth And Gate Count Analysis

The MIN and MAX gates obtained during synthesis in Algorithm 9 are represented using implication and negation operators, in order to restrict the synthesis to primitive operators and maintain uniformity. Each MIN and MAX gates is a two-level network of 2 implication gates. Hence, the synthesis using Algorithm 9 uses $(4N-2)$ implication gates with $2+2\lceil \log_2 N \rceil$ levels.*. The synthesis of all delta literals using Algorithm 10 requires $3N-4+(N\lceil \log_2(N-1) \rceil)$ implication gates spread across $3+\log_2(N-1)$ levels.

Therefore, the synthesis of a $N:1$ multiplexer requires $7N-6+N(\lceil \log_2(N-1) \rceil$ implication gates with $5+2(\lceil \log_2 N \rceil)+\lceil \log_2(N-1) \rceil$ levels.

**Example 7.6.** *From the synthesis of the* $4:1$ *multiplexer discussed previously (Example 7.4),* 4 *delta literals,* 4 *MIN and* 3 *MAX gates are required. Implementing MIN and MAX in terms of implication and negation gates, requires* 14 *implication gates with* 6 *levels. Synthesis of delta literals (Example 7.5 requires* 5 *levels consisting of* 16 *implication gates. Hence, the synthesis of a* $4:1$ *multiplexer needs* 30

*All levels are measured in terms of implication gates unless specified otherwise

*implication gates with* 11 *levels.*

## 7.6 Optimized Multiplexer Synthesis For Constant Inputs

Let $L_x$ be a literal with input $x \in W^N$ with output values $\{a_0, a_1, ..., a_{N-1}\} \in W_o^N$, where $N_o$ is the output cardinality of the literal. The literal can be synthesized as a multiplexer with inputs $in_i = a_i (0 \leq i < N)$. Due to the inputs being constant, the synthesis can be optimized using axioms presented in Section 7.1. Algorithm 12 describes the optimized multiplexer synthesis for constant inputs.

**Algorithm 12:** Synthesis of optimized multiplexer for constant inputs.

**Procedure** `OptiMux`($S$,$\Delta^{N,sel}$)
  **if** $(|W_{N-1}^N| > |W_0^N|) \in S$ **then**
    **for** $i \in \{0, 1, ..., N-1\}$ **do**
      $S[i] = \neg S[i]$;
  $D$ = array of delta literals;
  $Map$ = map of set $S$ values and corresponding delta literals;
  Sort $Map$ in decreasing order of number of delta literals;
  **for** $i = 0; i < Map.size; i++$ **do**
    $D = Map.delta\_literals$;
    $val = Map.value$;
    **if** $D.size < \left\lceil \frac{N}{2} \right\rceil$ **then**
      **forall** $d \in D$ **do**
        $Max_i = (Max_i \vee d)$;
      $Min_i = (val \wedge Max_i)$;
    **else**
      **forall** $(d \in \Delta^{N,sel}$ *and* $d \notin D)$ **do**
        $Max_i = (Max_i \vee d)$;
      $Min_i = (val \wedge \neg Max_i)$;
  $out = (Min_0 \vee Min_1 \vee ... \vee Min_{Map.size-1})$ by parallel reduction tree algorithm;
  **if** $(|W_{N-1}^N| > |W_0^N|) \in S$ **then**
    $out = \neg out$;

A MIN and MAX gate can be saved if the input to the multiplexer is $W_0^N$ (Axioms 7.10 and 7.12), whereas a MIN gate is saved if the input is $W_{N-1}^N$ (Axiom 7.13). Hence, we try to maximize the number of $W_0^N$ as inputs to the multiplexer, by inverting the literal output values if the number of $W_0^N$ is less than $W_{N-1}^N$ (Lines 4-7). In Algorithm 9, the minimum of each input to the multiplexer and the corresponding delta literal is determined. In Algorithm 12, a map of multiplexer input values and the corresponding delta literals is generated. The map is sorted in the descending order of the number of delta literals corresponding to each value (Lines 10-11). The maximum of delta literals $Max_i$ corresponding to each value is computed (Lines 15-18). If the number of delta literals corresponding to a value is greater than or equal to $\left\lceil \frac{N}{2} \right\rceil$, the maximum of the delta literals *not associated* with the value is determined instead. In the above case, the maximum $Max_i$ is inverted (Lines 21-25). Then, the minimum of each value and the corresponding maximum of delta literals is computed. Finally, the output of the multiplexer is the maximum of all minimums $Min_i$ $(0 \leq i < N)$, determined by parallel tree reduction algorithm using 2 input MAX gates. If the literal values were inverted initially, the output *out* is also inverted (Lines 28-30). The multiplexer is further optimized using Axioms 7.10, 7.12 and 7.13. It is possible to reduce gate count, by sharing delta literals and identical gates, during synthesis of multiple literals having the same input.

**Example 7.7.** *Figure 7.7 shows the synthesis of a literal* $L_{sel} = \{0, 0.2, 0.2, 1\}$. *The input and output cardinality of the literal are* 4 *and* 6 *respectively. Table 7.7a shows the values at each step of the synthesis procedure. First, the values and corresponding delta literals are mapped in Column 2 and 3. The map is sorted in decreasing order of number of delta literals. Next, the maximum of the delta literals is determined in Column 4. It should be noted in the Column 4 that for the value* 0.2 *the delta literals* $\Delta_0$ *and* $\Delta_3$ *are used instead of* $\Delta_1$ *and* $\Delta_2$, *to reuse the literals* (Lines 21-23). Finally, the minimum of value and maximum of delta literals is described in Column 5. Figure 7.7b shows the above synthesized circuit. The circuit is optimized using Axioms 7.10, 7.12 and 7.13. The optimized circuit representing literal is shown in Figure 7.7c.*

**(a)**

| $i$ | Value | Delta Literals | $(Max_i)$ | $(Min_i)$ |
|---|---|---|---|---|
| 0 | 0.2 | $\Delta_1, \Delta_2$ | $\neg(\Delta_0 \vee \Delta_3)^*$ | $(0.2 \wedge (\neg(\Delta_0 \vee \Delta_3))$ |
| 1 | 0 | $\Delta_0$ | $\Delta_0$ | $(0 \wedge \Delta_0) = 0$ |
| 2 | 1 | $\Delta_3$ | $\Delta_3$ | $(1 \wedge \Delta_3) = \Delta_3$ |

**(b)** **(c)**

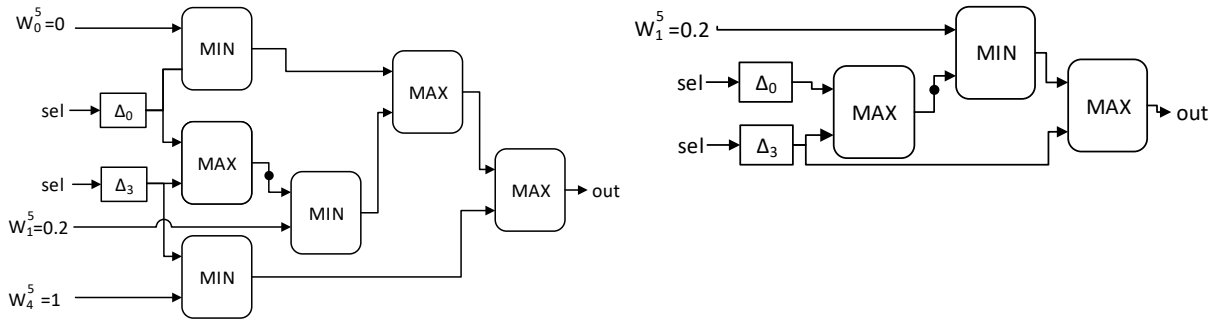



**Figure 7.7:** Synthesis of literal $L_{sel} = \{0, 0.2, 0.2, 1\}$. (a) Table consisting of intermediate values of the synthesis procedure. (b) Synthesized literal before optimization. (c) Synthesized literal after optimization.

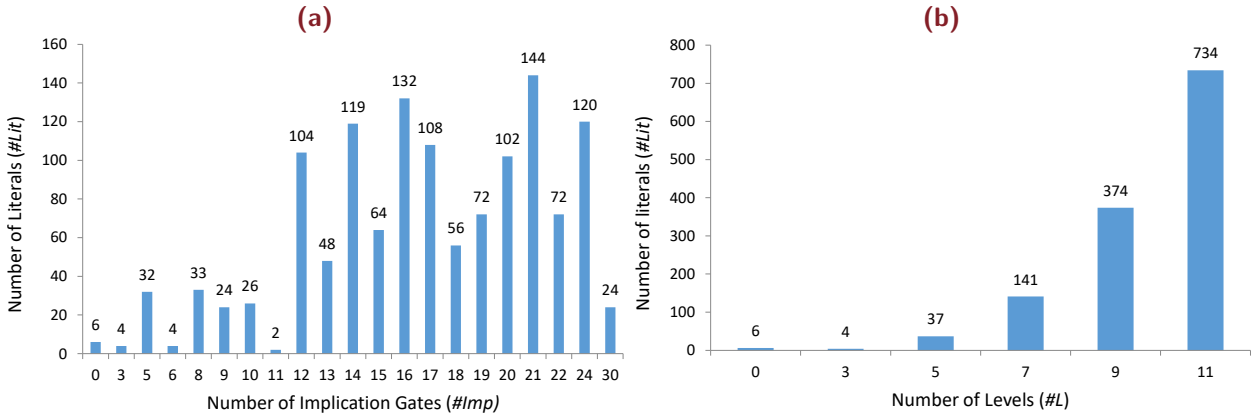



**Figure 7.8:** Graph showing number of implication gates (a) and levels (b) vs Number of literals, for all possible literals with input/output cardinality 4/6.

## 7.7 Experimental Results For MVL Synthesis

This section describes the experimental results of the synthesis. The synthesis was carried out on Intel i5-5300 processor with 8GB of RAM. The average run time for the synthesis was less than a minute. We evaluate the optimized Algorithm 12 for synthesis of literals by a considering exhaustive combinations of all literals having input and output cardinalities of 4 and 6 respectively. Using Algorithm 9, irrespective of the exact input/output combinations, the synthesis would require 30 implication gates with 11 levels in the circuit. Figure 7.8a shows the graph of the number of implication gates ($\#Imp$) vs the number of literals ($\#Lit$). Out of the total number of 1296 literals with input/output cardinalities 4/6, only 24 literals require 30 gates. All the other literals are optimized to be realized with fewer gates (for 4 literals, just 3 implication gates suffice), demonstrating the effectiveness of the Algorithm 12. Similarly, Figure 7.8b shows the graph of number of levels ($\#L$) vs number of literals ($\#Lit$).

We evaluate the proposed method on MVL functions obtained from POLO benchmarks. The benchmarks are first decomposed into a network of literals, MIN and MAX gates using the SEPALL3 mode of YADE [165][†]. The literals obtained are synthesized, using Algorithm 9. We also synthesize literals using optimized synthesis Algorithm 12. Finally, all the gates (MIN and MAX) are implemented as implication and negation by using Axioms 7.4 and 7.5.

The results are tabulated in Table 7.1. We explain the results using the *alet* benchmark circuit, as a representative case. The decomposition of *alet* benchmark using YADE yields a 31 level circuit with 135 literals and 346 implication gates. Thereafter, synthesizing the literals using Algorithm 9 requires additional 5724 gates and 16 levels. Using the optimized Algorithm 12, implementation of literals require 888 gates and the circuit requires 43 levels. This is a 8.5% reduction in number of levels and 79.7% reduction in number of gates compared to Algorithm 12. The implementation of literals using Algorithm 12 requires an overhead of 38.7% levels and 256.6% gates.

For the considered benchmark suite, the optimized Algorithm 12 leads to an average reduction of

[†]The authors would like to thank Dr. Christian Lang, for providing us the decomposer YADE.

**Table 7.1:** Experimental results of MVL synthesis on POLO benchmarks.

| | | YADE RESULTS | | | Algorithm 9 | | Algorithm 12 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | I/O | $\#L$ | $\#Imp$ | $\#Lit$ | $\#L$ | $\#Imp$ | $\#L$ | $\#Imp$ | $LI\%$ | $GI\%$ | $LO\%$ | $GO\%$ |
| alet | 18/1 | 31 | 346 | 135 | 47 | 6070 | 43 | 1234 | 8.5 | 79.7 | 38.7 | 256.6 |
| audiology | 69/1 | 25 | 226 | 83 | 37 | 1500 | 33 | 652 | 10.8 | 56.5 | 32.0 | 188.5 |
| balance | 4/1 | 27 | 294 | 59 | 39 | 1440 | 37 | 574 | 5.1 | 60.1 | 37.0 | 95.2 |
| breastc | 9/1 | 17 | 100 | 48 | 33 | 2320 | 29 | 538 | 12.1 | 76.8 | 70.6 | 438.0 |
| bridges1 | 9/1 | 17 | 98 | 40 | 27 | 662 | 27 | 301 | 0.0 | 54.5 | 58.8 | 207.1 |
| bridges2 | 10/1 | 25 | 168 | 52 | 35 | 894 | 35 | 435 | 0.0 | 51.3 | 40.0 | 158.9 |
| car | 6/1 | 21 | 98 | 23 | 31 | 452 | 27 | 211 | 12.9 | 53.3 | 28.6 | 115.3 |
| cloud | 6/1 | 17 | 74 | 32 | 32 | 1188 | 26 | 329 | 18.8 | 72.3 | 52.9 | 344.6 |
| flag | 28/1 | 23 | 272 | 89 | 36 | 2623 | 36 | 851 | 0.0 | 67.6 | 56.5 | 212.9 |
| flare1 | 10/3 | 17 | 144 | 37 | 30 | 740 | 28 | 346 | 6.7 | 53.2 | 64.7 | 140.3 |
| flare2 | 10/3 | 23 | 310 | 60 | 36 | 1294 | 32 | 530 | 11.1 | 59.0 | 39.1 | 71.0 |
| hayes | 4/1 | 11 | 18 | 8 | 23 | 225 | 21 | 83 | 8.7 | 63.1 | 90.9 | 361.1 |
| iris | 4/1 | 9 | 16 | 9 | 25 | 464 | 23 | 181 | 8.0 | 61.0 | 155.6 | 1031.3 |
| lensesmv | 4/1 | 9 | 18 | 8 | 18 | 88 | 16 | 36 | 11.1 | 59.1 | 77.8 | 100.0 |
| let | 18/1 | 25 | 290 | 107 | 41 | 4684 | 35 | 920 | 14.6 | 80.4 | 40.0 | 217.2 |
| mm3 | 5/1 | 15 | 50 | 17 | 24 | 260 | 22 | 118 | 8.3 | 54.6 | 46.7 | 136.0 |
| mm4 | 5/1 | 17 | 94 | 29 | 27 | 580 | 27 | 276 | 0.0 | 52.4 | 58.8 | 193.6 |
| mm5 | 5/1 | 23 | 126 | 39 | 35 | 933 | 33 | 420 | 5.7 | 55.0 | 43.5 | 233.3 |
| monks1te | 6/1 | 9 | 12 | 7 | 18 | 118 | 12 | 31 | 33.3 | 73.7 | 33.3 | 158.3 |
| monks2te | 6/1 | 21 | 110 | 12 | 31 | 266 | 23 | 123 | 25.8 | 53.8 | 9.5 | 11.8 |
| monks2tr | 6/1 | 21 | 214 | 29 | 31 | 584 | 27 | 270 | 12.9 | 53.8 | 28.6 | 26.2 |
| monks3te | 6/1 | 5 | 6 | 4 | 15 | 86 | 9 | 18 | 40.0 | 79.1 | 80.0 | 200.0 |
| monks3tr | 6/1 | 13 | 46 | 16 | 23 | 266 | 19 | 88 | 17.4 | 66.9 | 46.2 | 91.3 |
| sensory | 11/1 | 33 | 1642 | 340 | 43 | 6394 | 43 | 3096 | 0.0 | 51.6 | 30.3 | 88.6 |
| ships | 4/1 | 13 | 28 | 13 | 25 | 245 | 23 | 100 | 8.0 | 59.2 | 76.9 | 257.1 |
| shuttlem | 6/1 | 11 | 24 | 8 | 21 | 114 | 15 | 49 | 28.6 | 57.0 | 36.4 | 104.2 |
| sleep | 9/1 | 11 | 30 | 14 | 30 | 714 | 26 | 266 | 13.3 | 62.7 | 136.4 | 786.7 |
| sponge | 44/1 | 7 | 10 | 6 | 20 | 230 | 14 | 34 | 30.0 | 85.2 | 100.0 | 240.0 |
| tic-tac-toe | 9/1 | 31 | 902 | 47 | 40 | 1436 | 34 | 966 | 15.0 | 32.7 | 9.7 | 7.1 |
| u1 | 60/1 | 27 | 388 | 117 | 40 | 2944 | 38 | 932 | 5.0 | 68.3 | 40.7 | 140.2 |
| u2 | 60/1 | 25 | 382 | 119 | 37 | 2389 | 35 | 860 | 5.4 | 64.0 | 40.0 | 125.1 |
| u3 | 60/1 | 23 | 414 | 121 | 35 | 2577 | 35 | 971 | 0.0 | 62.3 | 52.2 | 134.5 |
| u4 | 60/1 | 25 | 360 | 108 | 37 | 2220 | 33 | 864 | 10.8 | 61.1 | 32.0 | 140.0 |
| u5 | 60/1 | 25 | 402 | 120 | 37 | 2559 | 35 | 958 | 5.4 | 62.6 | 40.0 | 138.3 |
| zoo | 16/1 | 11 | 20 | 10 | 26 | 224 | 20 | 89 | 23.1 | 60.3 | 81.8 | 345.0 |

I/O : Number of Inputs/Outputs
$LI\%$ : Percentage reduction in levels of Algorithm 12 compared to Algorithm 9.
$GI\%$ : Percentage reduction in gate count of Algorithm 12 compared to Algorithm 9.
$LO\%$ : Percentage overhead in levels after implementing literals obtained from YADE, using Algorithm 12.
$GO\%$ : Percentage overhead in gate count after implementing literals obtained from YADE, using Algorithm 12.

12.1% in the number of levels and 63.5% in the number of gates, compared to the synthesis Algorithm 9. The overhead of implementation of literals have been largely ignored for synthesis of multi-valued functions [165–167], with the assumption that multi-valued multiplexers can be natively realized. However, a physical implementation of multi-valued system would only have the basic logic primitives (for Łukasiewicz logic, the primitives are implication and negation) available for realizing a multi-valued function, as demonstrated earlier in this chapter. Our current work shows that implementation of the literals using the Łukasiewicz logic primitives lead to an average overhead of 55% levels and 216% gates. This highlights the need for optimized algorithms for synthesis of the literals. In the next section, we look at the realization of fuzzy logic using Łukasiewicz logic operators and present a demonstrative experimental case study.

## 7.8 Proof of Concept: Fuzzy Logic Controller Implementation

The implementation of a given fuzzy system in Boolean logic requires the treatment of every member with varied degree in a separate set and performing Boolean logic operations on those sets. Therefore, the computation steps do also increase in logarithmic proportion when using the Boolean logic in comparison to the fuzzy logic. Traditionally, Mamdani-type fuzzy systems use *min* and *max* functions for evaluation of fuzzy rules and combining the output of the rules [154]. *min* and *max* functions can be expressed as in terms of Łukasiewicz logic operators:

$$min(u, v) = (u \to v) \to v \tag{7.20}$$

$$max(u, v) = \neg(min(\neg u, \neg v)) \tag{7.21}$$

Therefore, it is possible to use the multi-valued Łukasiewicz logic for realization of Mamdani type fuzzy systems as well. In general, Łukasiewicz logic is capable of dealing with a wide range of approximate reasoning paradigms, since it can express evaluation function of multi-valued logic classes described in terms of *+, -, min* and *max* [170–172].

We demonstrate the realization of a fuzzy logic controller as a representative application of Łukasiewicz logic. A conventional fuzzy controller has three major steps, as shown in Figure 7.9a.

1. Fuzzify input variables using fuzzy membership functions. If the inputs to the system are analog, analog-to-digital converters would be used to convert the inputs to the corresponding values in Łukasiewicz logic. Then, these multi-valued inputs are *fuzzified* using the membership functions.

2. Execute all the fuzzy inference rules from the rule database to determine the fuzzy output functions.

3. Defuzzify the fuzzy output functions to get *crisp* output value, i.e, a single multi-valued output value.

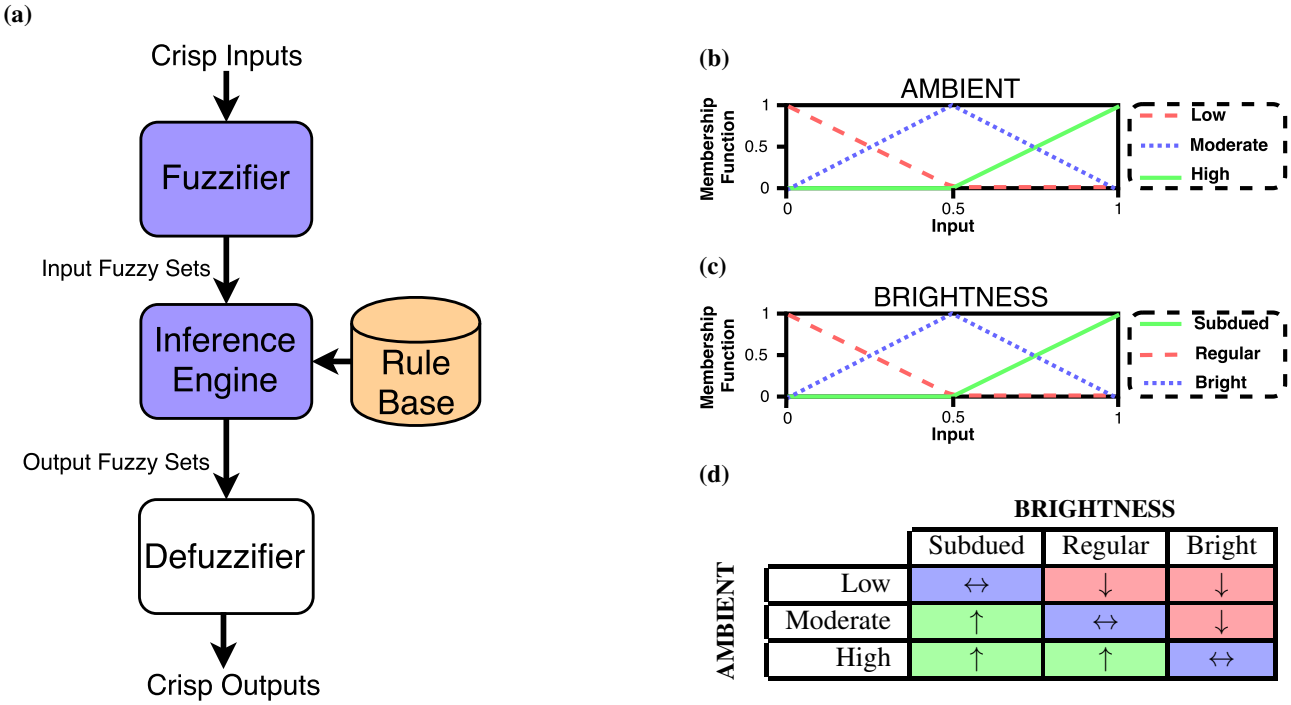



**Figure 7.9:** Overview of fuzzy logic control. (a) Processing blocks used for fuzzy logic control are shown. The violet colored blocks are implemented using the multi-state ReRAM devices. The rule base is stored as control/instruction steps. (b) Membership functions for variable *AMBIENT* (c) Membership functions for variable *BRIGHTNESS* (d) Rule table for fuzzy brightness controller.

To illustrate the working of a fuzzy logic control, we consider a fuzzy logic controller for regulating

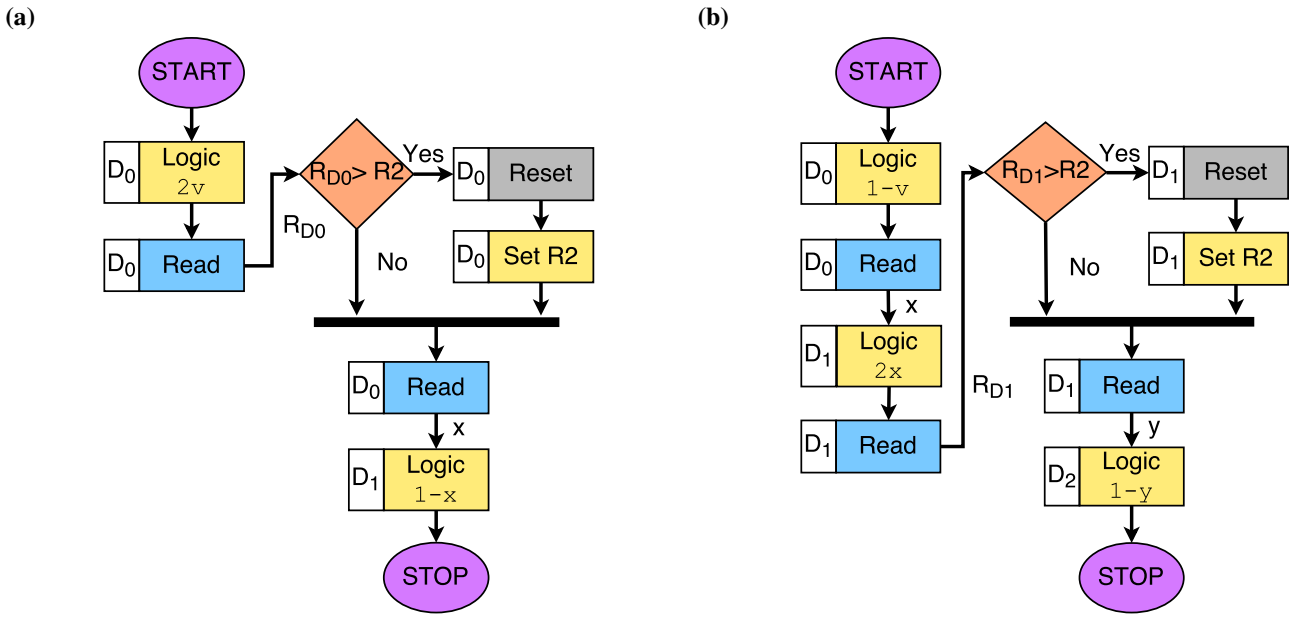



**Figure 7.10:** Realization of two fuzzy membership functions. (a) Steps to compute inverted notch$[(\neg v \to v) \to 0]$. (b) Steps to compute flipped notch$[(v \to \neg v) \to 0]$.

screen brightness. The screen brightness has to be regulated based on the ambient light *AMBIENT* and screen brightness *BRIGHTNESS*. Each of the variables can be represented using three gradations.

- $AMBIENT \in$ {Low (L), Moderate (M), High (H)}
- $BRIGHTNESS \in$ {Subdued (S), Regular (R), Bright (B)}

We use the fuzzy membership functions shown in Figure 7.9b and Figure 7.9c to fuzzify the input variables *AMBIENT* and *BRIGHTNESS* respectively. The fuzzy membership functions can be expressed in terms of Łukasiewicz operators [173, 174]. Therefore, we can realize the "Fuzzifier" block of a fuzzy control system (shown in Figure 7.9a) using Łukasiewicz logic operations only. Let us consider the inverted notch membership function $f_{(LOW,AMBIENT)}$ for the grade *LOW* of variable *AMBIENT*. $f_{(LOW,AMBIENT)}$. It can be expressed as $(\neg v \to v) \to 0$, where $v$ is the input. Similarly, the flipped notch membership function $f_{(SUBDUED,BRIGHTNESS)}$ for the grade $SUBDUED$ of variable *BRIGHTNESS* can be written as $(v \to \neg v) \to 0$. These membership functions can be simplified and expressed in terms

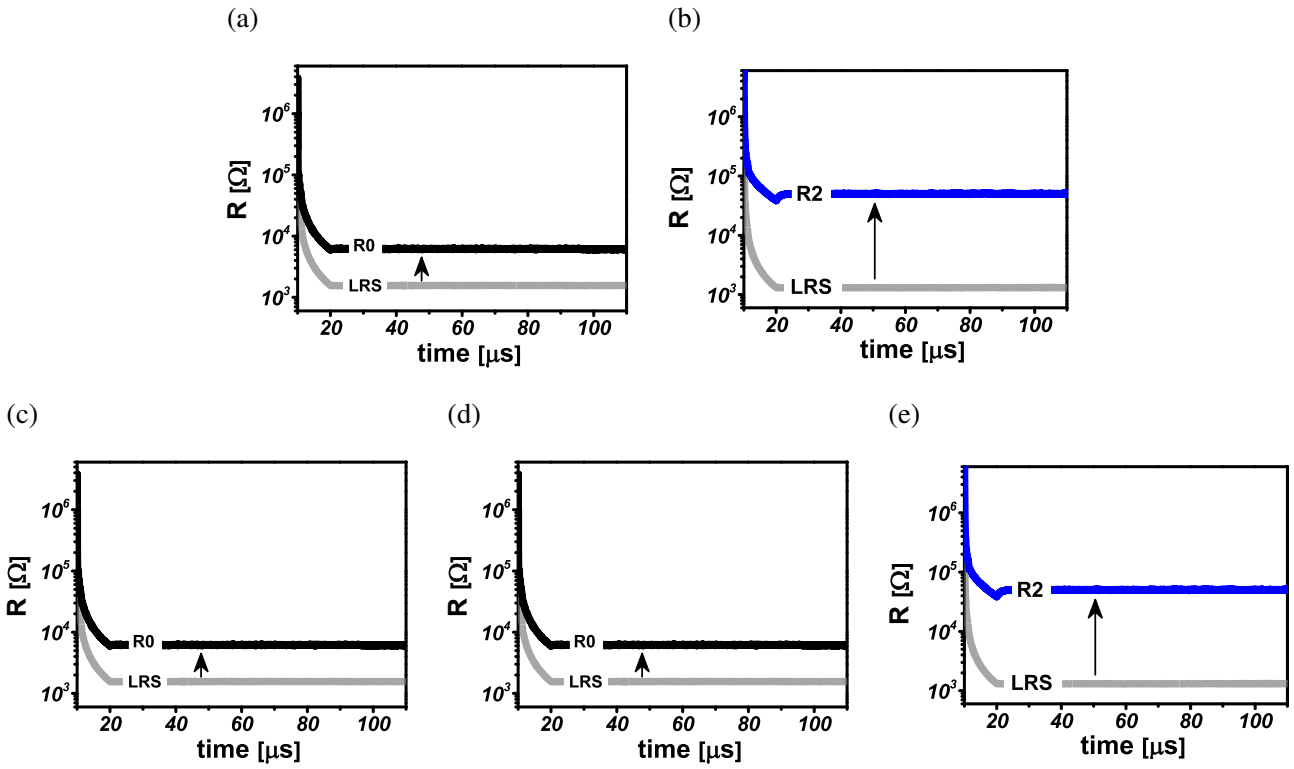



**Figure 7.11:** Membership function realization using multi-state ReRAM. (a-b) State transition of device $D_0$ and $D_1$ for realization of inverted notch membership function for $v = 0$. (c-e) State transition of devices $D_0$, $D_1$ and $D_2$ for realization of flipped notch membership function for $v = 1$.

of $min(u, v)$ and $\neg u$ $(1 - u)$ operation as shown below.

$$f_{(LOW,AMBIENT)} = (\neg v \rightarrow v) \rightarrow 0 = 1 - min(1, 2v) \tag{7.22}$$

$$f_{(SUBDUED,BRIGHTNESS)} = (v \rightarrow \neg v) \rightarrow 0 = 1 - min(1, 1 - v + 1 - v) \tag{7.23}$$

The series of steps required to realize the inverted notch and flipped notch membership function using multi-state ReRAM is presented in Figure 7.10a and Figure 7.10b respectively. We experimentally verified the correctness of computation. Figure 7.11 shows the experimental results when the input variable *AMBIENT* is 0 and also for value 1 for input *BRIGHTNESS*.

The fuzzy inference engine determines the *ACTION* to be taken based on fuzzified inputs and be stated as a set of rules. The *ACTION* can be increase brightness (↑), decrease brightness (↓) or

no action (↔). A rule for example can the following — if *AMBIENT* is Low and *BRIGHTNESS* is Subdued, then *ACTION* is no action (↔). Figure 7.9d presents multiple such control rules to determine the *ACTION* represented compactly as a rule table. To evaluate the output of a fuzzy rule, a suitable T-norm function is used. We use Łukasiewicz T-norm — $max(0, u + v - 1)$ for evaluation of the fuzzy controller rules. The Łukasiewicz T-norm can be expressed in terms of *neg* and *min* functions as $\neg min(1, (1 - u) + (1 - v))$. Figure 7.12a show the sequence of steps to realize Łukasiewicz T-norm using the ReRAM devices. As a representative example, Figure 7.13 shows the state transitions of the ReRAM devices during computation of T-norm for inputs $u = 1$ and $v = 1$. Once all the rules have

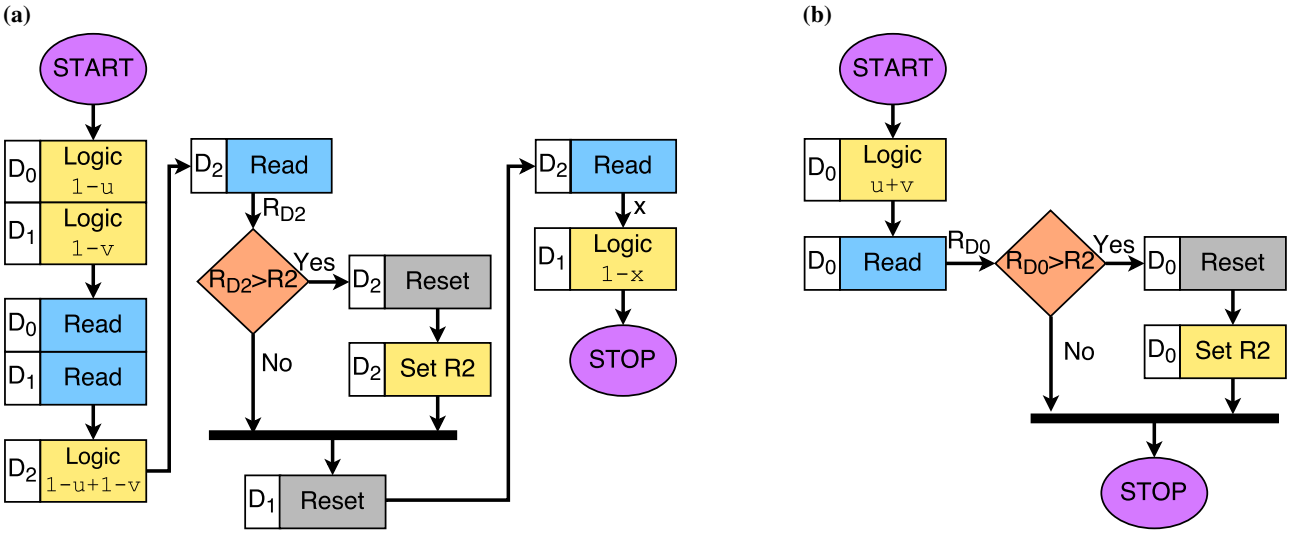



**Figure 7.12:** Fuzzy rule evaluation. (a) Computation of Łukasiewicz T-norm. (b) Computation of Łukasiewicz T-conorm.

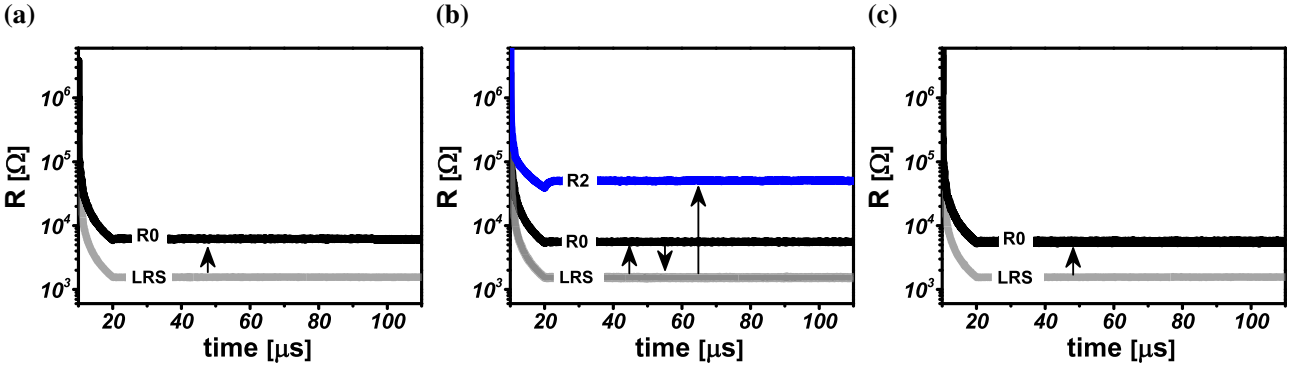



**Figure 7.13:** Łukasiewicz T-norm computation for $u = 1$ and $v = 1$. State transitions for (a) $D_0$ (b) $D_1$ and (c) $D_2$ ReRAM device respectively.

been evaluated, the outputs of these is combined using a suitable T-conorm. The T-conorm should be the dual of the T-norm used for rule evaluation. Therefore, we use Łukasiewicz T-conorm which is $min(1, a + b)$, for combining the output of the rules. Figure 7.13b shows the steps for computation of the Łukasiewicz T-conorm using the multi-state memristive devices. To obtain *crisp* output, the combined fuzzy output in the end has to be defuzzied. The defuzzifier block shown in Figure 7.9a computes *crisp* output using an appropriate defuzzification method [175]. Note that the defuzzifier block has not been implemented in the presented prototype, which can be realized using conventional methods.

## 7.9 Discussion

Among emerging non-volatile storage technologies, redox-based resistive switching Random Access Memory (ReRAM) is a prominent one. The realization of Boolean logic functionalities using ReRAM adds an extra edge to this technology. In this manuscript, we report realization of multi-valued logic primitives natively using multi-state ReRAM devices. We also demonstrated an end-to-end MVL synthesis flow using Łukasiewicz logic primitives. Then, we proposed an algorithm to synthesize and optimize literals which are constant-input multiplexers. The synthesis of literals is applied to circuits obtained by decomposition of POLO [176] benchmark functions using YADE [165]. An average overhead of 55.6% in the number of levels and 215% in the number of gates are required to implement the literals.

Knowledge-based system is capable to reason with judgmental, imprecise, and qualitative knowledge as well as with formal knowledge of established theories. The design of such systems is an important challenge in the realm of Artificial Intelligence (AI). The incompleteness and uncertainty associated with the knowledge-base is handled through fuzzy logic. Fuzzy logic allows linguistic variable [170] to be assigned inexact or partial truth values for modeling logical reasoning. In this chapter, we demonstrated a practical fuzzy inference system, by realizing the fuzzy logic operations using the multi-state $TaO_x$ devices. Further, each fuzzy operation is mapped to a series of multi-valued logic primitives, namely,

Łukasiewicz logic. We showed a practical fuzzy inference system through a limited number of logical steps and $1 \times 3$ memristive crossbar array. The multi-state $TaO_x$ devices enable computation entirely using multi-valued truth values for the operations, without need for any intermediate representations. Therefore, these devices provide a natural platform to undertake multi-valued logic and thus, fuzzy inference operations.

# Part III:

# Automation Techniques For Quantum Computing

# 8 Qubit-constrained Quantum Circuit Synthesis

The race for establishing quantum supremacy [177] is at its height right now. Researchers across the world are identifying applications, which can achieve at least super-polynomial speed-up for a quantum computer over the best possible classical one. In parallel, by extending the limits of classical computing systems, up to 56 qubit quantum circuit is recently simulated on a supercomputer [178]. Though this limit is surpassed by the recent quantum processor from Google with 72 qubits [179], yet it is not sufficient to establish quantum supremacy due to the error that builds up during the quantum computation [180]. To tackle this challenge, it is imperative to physically implement multi-qubit gates with low error rates while quantum algorithms have been simultaneously mapped to obtain a compact circuit as low resources as possible.

## 8.1 Preliminaries

We introduce the concepts of computing using a quantum computer and the proceed to describe the challenges of synthesizing an algorithm on to a set of quantum gates.

**Definition 8.1** (Qubit)**.** *A* quantum bit*, or* qubit *for short, is a 2-dimensional Hilbert space $H_2$. The orthonormal basis of $H_2$ is labeled by $\{|0\rangle, |1\rangle\}$. The* state *of the qubit is an associated unit length vector in $H_2$.*

A qubit can be in any state $a|0\rangle + b|1\rangle$, where $a, b \in \mathcal{C}$ and $|a|^2 + |b|^2 = 1$. There are exactly two possible *pure* states for a qubit, $|0\rangle$ and $|1\rangle$. A classical bit can be thought of as *qubit* in pure state. It is assumed that the state of a qubit can be initialized, to a pure state, usually $|0\rangle$.

**Definition 8.2** (Tensor Product of Vectors)**.**

$$(\sum_{i=1}^{n} \alpha_i |x_i\rangle) \otimes (\sum_{j=1}^{m} \beta_j |y_i\rangle) = \sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_i \beta_j |x_i y_j\rangle$$

**Definition 8.3** (Quantum Register)**.** *A quantum register of length $m$ can be defined as $H_{2^m} = H_2 \otimes H_2 \ldots H_2$ ($m$ times). The basis of a quantum register of length $m$ is defined as $\{|x_0\rangle|x_1\rangle \ldots |x_m\rangle : x_i \in \{0, 1\}\}$.*

### 8.1.1 Quantum Gates

**Definition 8.4** (Unary Quantum Gate)**.** *A* unary quantum gate *is a unitary linear map on a single qubit. Some important unitary quantum gates are — NOT gate, phase flip gate $F_\theta$, the Hadamard gate $H$ and the $\frac{\pi}{8}$ gate, simply called $T$-gate.*

$$NOT := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad F_\theta := \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \quad H := \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad T := \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$$

**Definition 8.5** (Binary Quantum Gate)**.** *A binary quantum gate is a unitary operation on two qubits, i.e., a unitary map* $H_2 \otimes H_2 \to H_2 \otimes H_2$.

By using the concept of tensor product of matrices, controlled gates can be defined.

**Definition 8.6** (Tensor Product of Matrices)**.** *Given two matrices* $A$ *and* $B$ *with dimensions* $r_A \times c_A$ *and* $r_B \times c_B$ *respectively, the tensor product is defined as :-*

$$A \otimes B := \begin{bmatrix} a_{11}B & a_{12}B & \dots & a1c_AB \\ a_{21}B & a_{22}B & \dots & a2c_AB \\ \vdots & \vdots & \ddots & \vdots \\ a_{r_A1}B & a_{r_A2}B & \dots & ar_Ac_AB \end{bmatrix}$$

**Definition 8.7** (Controlled $M$-gate)**.** *Given two qubits* $A$ *and* $B$ *and a unary quantum gate* $M$ *acting on* $B$*, the controlled-*$M$ *gate is defined a binary gate on* $A \otimes B$ *:-*

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes I_2 + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \otimes M$$

**Example 8.1.** *The controlled-not gate or simply CNOT gate, also known as Feynman gate is defined as follows.*

$$M_{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

**Definition 8.8** (Toffoli gate)**.** *A Toffoli gate is a gate defined over three qubits that maps* $|c_1\rangle|c_2\rangle|t\rangle \to |c_1\rangle|c_2\rangle|(c_1 \wedge c_2) \oplus t\rangle$*, where* $\wedge$ *and* $\oplus$ *represent Boolean AND and exclusive OR (XOR) operations respectively.*

**Definition 8.9** (Single Target gate)**.** *A STG* $T_c$ $(\{x_1, x_2, \dots, x_k\}, x_{k+1})$ *has* $k$ *control lines* $x_1, \dots, x_k$*,*

*a single target line* $x_{k+1}$ *and a control function* $c : \mathbb{B}^{k+1} \to \mathbb{B}$. *The gate realizes a reversible function* $f : \mathbb{B}^{k+1} \to \mathbb{B}^{k+1}$ *with* $x_i \longmapsto x_i$, $\forall i \leq k$ *and* $x_{k+1} \longmapsto x_{k+1} \oplus c(x_1, \ldots, x_k)$.

In general, a quantum gate is a unitary mapping that acts on a quantum register of length $m$ that acts on a fixed number of qubits, that is independent of $m$. Since each quantum gate $M$ has an inverse, we denote the inverse of the gate as $M^{\psi}$.

### 8.1.2 Quantum Circuits

A quantum circuit can be coarsely defined as a quantum circuit as a concatenation of quantum gates acting on a quantum register of some length $m$. All operations on qubits are reversible in nature, except measurement.

Let $\mathbb{B} = \{0, 1\}$ denote *Boolean values*. We refer to $\mathcal{B}_{r,s} = \{f \mid f : \mathcal{B}^r \to \mathcal{B}^s\}$ as the set of all *Boolean multiple-output functions* with $r$ inputs and $s$ outputs. A function $f \in \mathcal{B}_{r,r}$ is called *reversible* if $f$ is *bijective*, that is, for each input pattern, the output is uniquely determined.

Formally, a quantum circuit for a reversible function $B_{r,r}$ has at least $r$ logical qubits on which quantum gates operate, aligned into a cascade. One of the popular reversible gate library is the NCT library, consisting of NOT, CNOT and Toffoli gates. It is well known that any reversible circuit can be mapped using single target gates (STG) [181].

#### Quantum Circuit Visualization

Quantum circuits can be visualized using a quantum circuit diagram. The circuit is read from left to right. Each line represents a qubit. A rectangle with a letter, spanning across multiple lines, indicates a gate operating on those qubits. For example, Figure 8.1a shows a Hadamard gate operating on a single qubit. The input is on the left of the gate while the output of the gate is on the right. For a controlled gate, filled dot indicate the control lines, while $\oplus$ on a line marks it as the target gate. Figure 8.1b

shows a CNOT gate with a single control line while Figure 8.2 shows a Toffoli gate with two control lines.

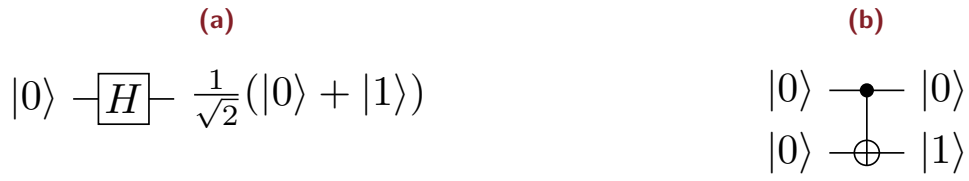



**Figure 8.1:** (a) Hadamard gate. (b) CNOT gate.

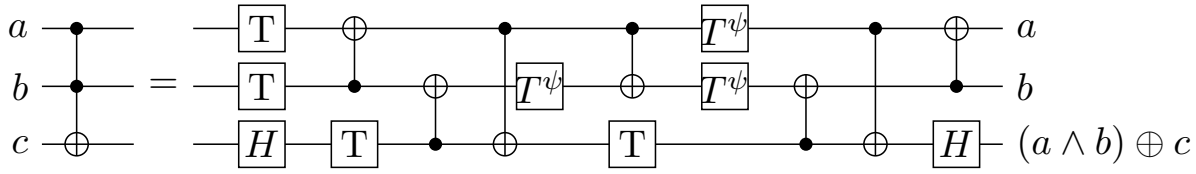



**Figure 8.2:** Toffoli gate and decomposition of the Toffoli gate into the Clifford+T library. The decomposed circuit has 3 qubits, T-count of 7 and T-depth of 3. The inverse of any quantum gate $M$ is represented by $M^\psi$.

### Quantum Circuit Cost

The most promising universal quantum gate library that efficiently implements fault-tolerant Quantum error correction codes, is the Clifford+T library. This library is universal for reversible and contain quantum gates that realize the unitary matrices $\{NOT, CNOT, H, Z, S, S^\dagger, T, T^\dagger\}$. In general, a STG $(T_c(\{x_1, x_2, \ldots, x_k\}, x_{k+1}))$ can be decomposed into a cascade of multi-control Toffoli gates (a special STG with control function as either 1, i.e., tautology or a single product term).

$$T_{c1}(X_1, x_{k+1}) \circ T_{c1}(X_1, x_{k+1}) \circ \ldots \circ T_{c1}(X_1, x_{k+1})$$

Individual Toffoli gates can be mapped to the fault tolerant Clifford+T gate library for native implementation of a quantum computer. Multiple techniques exist in literature for mapping multi-control Toffoli gates into Clifford+T circuits [182, 183]. For example, Figure 8.2 shows the mapping of a Toffoli gate using the Clifford+T library. The mapped circuit has 3 qubits, T-count equal to 7 and T-depth equal to 3.

In the Clifford+T library, the cost of implementing a T-gate is sufficiently high to customarily neglect the cost of Clifford group gates, while determining the cost of the quantum circuit [183]. Therefore, the number of T-gates or simply *T-count*, is another metric to judge the cost of a quantum circuit. The delay of a circuit realized using Clifford+T library is measured in terms of T-depth. The *T-depth* is the minimum number of T-stages in a quantum circuit where each stage consists of one or more T or $T^{\dagger}$ gates performed concurrently on separate qubits. The count of qubits used in a quantum circuit is another important metric, since the current quantum technologies still struggle to achieve superposition and error-free computation for large number of qubits. T-count and qubit count are two orthogonal metrics and space-time trade off can be performed [184–186].

### 8.1.3 Synthesizing Logic Network To Quantum Circuit

Hierarchical reversible logic synthesis (LHRS) represent the state-of-the-art synthesis technique for large combinational functions [187, 188]. LHRS starts by partitioning a Boolean network $N$ into a $k$-LUT (LUT stands for Look-Up Table) network. A Boolean network is $k$-feasible, if $\delta^{-}(v) \leq k, \forall v \in V$. Homogeneous Boolean networks such as And-Inverter Graph [189] and Majority-Inverter Graphs [109], can be transformed into $k$-feasible networks by means of LUT mapping algorithms [190, 191]. An elementary method to map a LUT network to reversible circuit is by means of using a STG for each LUT in the topological ordering. The target for the STG is chosen to be an ancilla line initialized to '0'. The circuit should not have any garbage lines to allow implementation on a quantum computer. This constraint arises due to the fact that result of the calculation is entangled with the intermediate results and thus they cannot be discarded and reused without damaging the results they are entangled with [192]. To disentangle the qubits and revert the targets to their initial state (constant 0), the STGs for the LUTs computing the intermediate results should be applied in the reverse topological order. This is followed by mapping each STG to the Clifford+T library using various techniques [193, 194]. The overall process is shown in Figure 8.3.

A STG can be computed on a free qubit (constant 0), if all its predecessors have been computed.

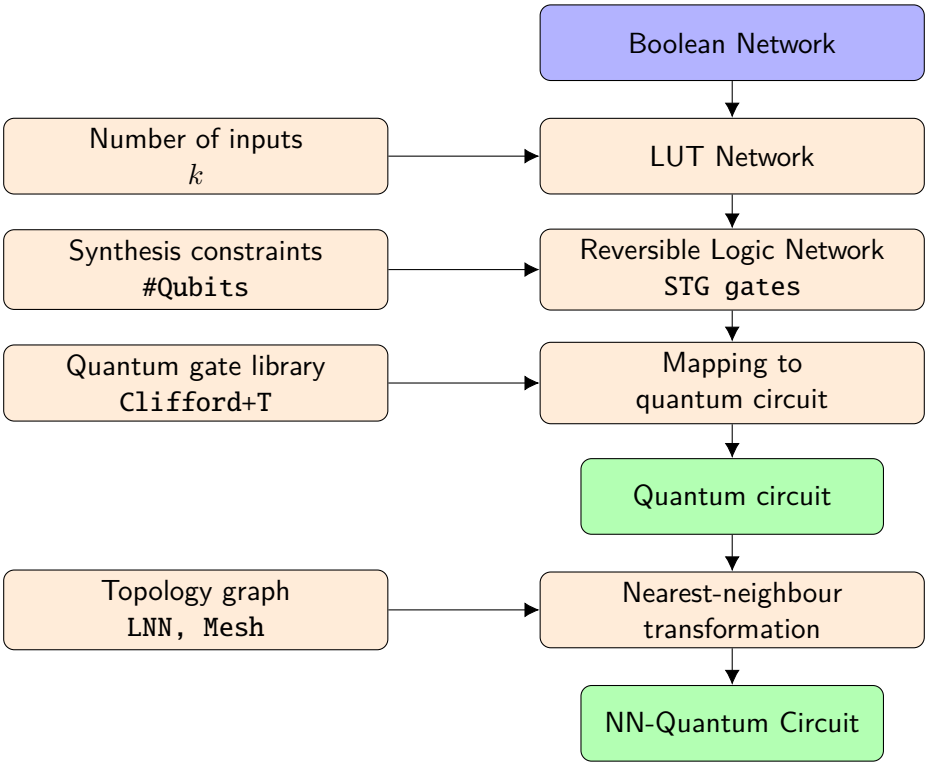



**Figure 8.3:** Hierarchical logic synthesis for quantum circuits with topology constraints.

Also, the qubit with the result of the STG can be returned to state 0, only when all the predecessors are in computed state. This is the primary constraint on *compute* and *uncompute* of STG. We express the possible operations using the following notations:

- $\texttt{PI}(x, q)$ for primary input $x \in P$ and qubit $q$: This assigns input $x$ to qubit $q$ in the circuit.
- $\texttt{COMP}(g, q)$ for gate $g \in G$ and qubit $q$: This applies a single target gate $T_{F(g)}(m(j)|j \in \delta^{-}, q)$ and sets $m(g) \leftarrow q$.
- $\texttt{UCOMP}(g, q)$ for gate $g \in G$ and qubit $q$: This acts same as $\texttt{COMP}(g, q)$ but sets $m(g) \leftarrow 0$. We represent this using $g^{\psi}$ in the circuit diagrams.
- $\texttt{PO}(x, q)$ for primary output $x \in O$ and qubit $q$: This assigns output $x$ to qubit $q$ in the circuit.

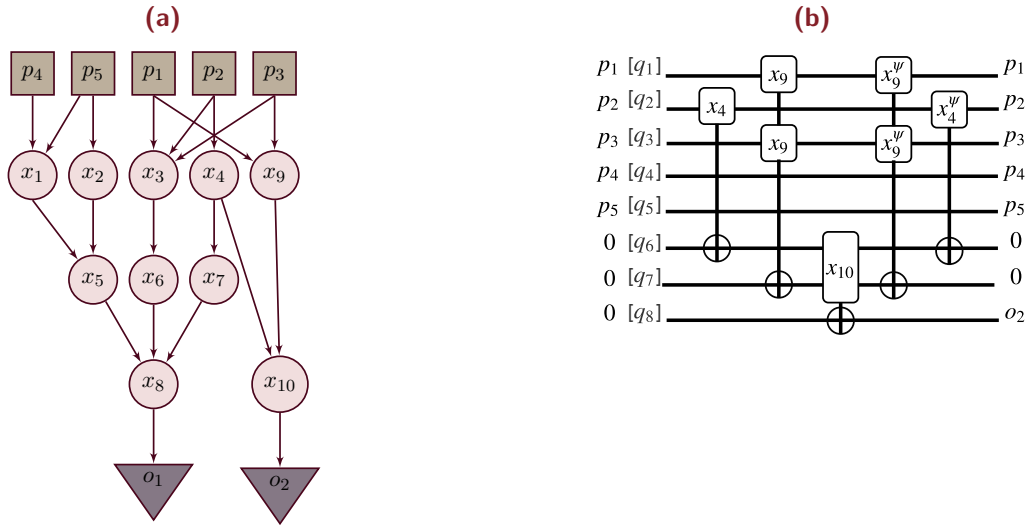



**Figure 8.4:** (a) A 3-feasible network with five inputs and two outputs. (b) Quantum circuit for computing output $o_2$ using STGs.

**Example 8.2.** *Figure 8.4a shows a 3-feasible network. The vertex set is* $V = P \cup O \cup G$ *where* $P = \{p_1, p_2, \ldots, p_5\}$, $O = \{o_1, o_2\}$ *and* $G = \{x_1, x_2, \ldots, x_{10}\}$ *while the cones corresponding to the outputs are —* $cone(o_1) = \{x_1, x_2, \ldots, x_8\}$ *and* $cone(o_2) = \{x_4, x_9, x_{10}\}$.

**Example 8.3.** *Figure 8.4b shows the circuit of mapping the cone of output* $o_2$ *using STGs. Each of the primary input is mapped to a unique qubit. Nodes* $x_4$ *and* $x_9$ *are computed first. This is followed by computing the node* $x_{10}$*, which is the output* $o_2$*. Then the intermediate nodes are uncomputed (represented by* $x_i^\psi$*). The entire circuit can be expressed in terms of operations as follows :* `PI`$(p_1, q_1)$, `PI`$(p_2, q_2)$, `PI`$(p_3, q_3)$, `PI`$(p_4, q_4)$, `PI`$(p_5, q_5)$, `COMP`$(x_4, q_6)$, `COMP`$(x_9, q_7)$, `COMP`$(x_{10}, q_8)$, `UCOMP`$(x_9, q_7)$, `UCOMP`$(x_4, q_5)$, `PO`$(o_2, q_8)$.

## 8.2 Motivation and Problem Statement

Quantum algorithms capable of running on quantum computers, have various components that are combinational functions. For example, integer multiplication is used as a sub-routine such as Shor's integer factorization [24]. Such a combinational function can be represented using traditional logic

network. This logic network can be use as input to the logic synthesis algorithm to generate quantum circuits. Each synthesis algorithm supports optimizing the output quantum circuit for a variety of performance metrics. For example, LHRS offers a good balance between the scalability, T-count and T-depth. The price for a better scalability compared to functional synthesis approaches is the requirement for many additional qubits to store temporary results of the hierarchical input representation. However, implementing a quantum circuit with large number of qubits poses a dominant problem towards the practical realization of the first batch of quantum algorithms. Optimisation of qubit count is a well-studied problem. First, there have been numerous studies on minimising the ancilla qubits [195], leading towards the ancilla-free, scalable synthesis approaches [196]. Second, by taking cue from Bennett's reversible pebble game* [198, 199], several heuristics have been proposed to include the "uncompute" stage [200–202]. However, none of these works presented a pebble game heuristic integrated with a quantum logic synthesis flow, which is also observed here [203]. The work closest to ours is by Parent et al. [200] that implemented pebbling strategies and games to demonstrate trade-off between qubit count and circuit depth. This work, however, does not explicitly address the problem of circuit synthesis. Further, the pebbling strategy needs to be closely integrated with the overall synthesis technique, which we undertook here.

In this chapter, we demonstrate and establish how reversible pebble games can be used to reduce the number of stored temporary results, thereby reducing the qubit count. Our proposed algorithm can be constrained with number of qubits, which is aimed to meet. Two optimisations have been proposed to lower the number of single target gates in the synthesized circuit. Experimental studies show that the qubit count can be significantly reduced (by up to 63.2%) compared to the state-of-the-art algorithms, at the cost of additional gate count.

*The irreversibility-space trade-off in the pebble game is also useful in other domains, such as, register allocation stage of compiler [197].

## 8.3 Qubit Count Reduction Methodology

This section presents a method to derive the upper bound on the number of qubits required for mapping a LUT network. An introduction to reversible pebble games, followed by a detailed example is presented. We then proceed to explain the proposed heuristic for reversible pebble games and two effective optimizations for the same.

### 8.3.1 Upper Bound On The Number Of Qubits Required

Given a LUT network with multiple outputs, the cone of each output can be considered separately as an LUT network. For each of the output, the output cone can be computed, followed by immediately uncomputing the intermediate results in the cone. This approach can be used to obtain an upper bound $N_{naive}$ on the number of qubits required for computing a LUT network.

**Lemma 8.1.** *A LUT graph with $N_i$ inputs, $N_o$ outputs and $N_c = max(cone(n_i))$, for $1 \leq i \leq m$, can be computed with at most $N_{naive} = N_i + N_o + N_c - 1$ qubits.*

*Proof:* Consider a LUT network with $N_i$ inputs and $N_o$ outputs such that each output cone has the same size $m$ $(= N_c)$ and do not share any logic among them. Each of the $N_i$ inputs is mapped to a unique qubit. Computation of each output cone would require $m$ qubits, with one qubit reserved for the output and the rest $m - 1$ qubits holding the intermediate results. The ancilla line with intermediate results can be uncomputed to the initial constant 0 state after an output has been computed, for reuse during computation of the next output cone. Thus, we need $N_o$ qubits for output and $m - 1$ qubits for computing the intermediate results.■

**Example 8.4.** *For the DAG shown in Figure 8.4, $N_i = 5$, $N_o = 2$ and $N_c = max(8, 3) = 8$. Therefore, the upper bound on the number of qubits required to map the graph is $N_{naive} = 14$.*

### 8.3.2 Background On Reversible Pebble Game

Bennett introduced the reversible pebble games [199] in the context of reversible computation, which is the exact problem we are addressing in this paper. Given a DAG $G$ with a unique sink node $s$, the reversible pebble games starts with no pebbles on $G$ and terminates with a pebble (*only*) on the sink $s$. Placing and removal of pebble on a node is governed by the following two rules: *(1) To pebble a node $v$, all the predecessors of $v$ must be pebbled. (2) To unpebble a node $v$, all the predecessors of $v$ must be pebbled.* In the context of computing a LUT network, the LUT network is the DAG while the pebbles represents qubits. Pebbling a node is equivalent to *computing* a node while either "unpebbling a node" or "unpebble operation" is same as *uncompute* operation of a node. The minimum number of pebbles required to pebble a DAG using reversible pebble game by Bennett is termed as the *reversible pebble number* of the graph. Determining the reversible number of a given DAG with a unique sink node (single output) is PSPACE-complete [204].

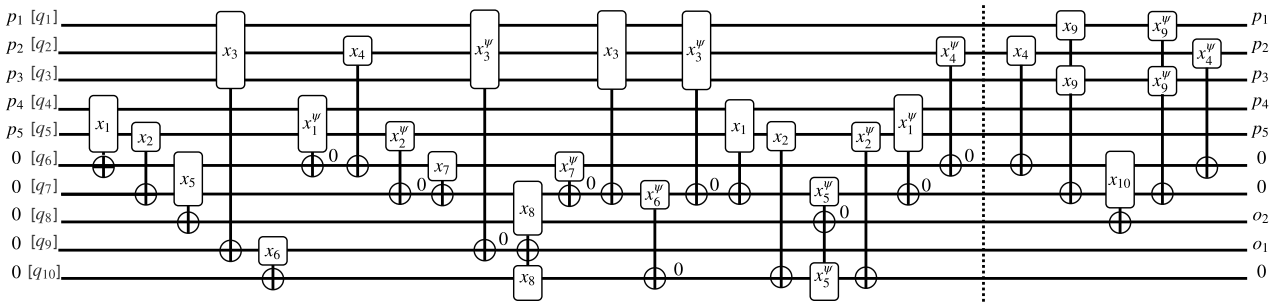



**Figure 8.5:** Mapping of the DAG shown in the Figure 8.4, using reversible pebble game. The rectangles denote the input lines of the STG, while the $\oplus$ denotes the target line of the STG. The vertical dotted line represents the completion of compute for output cone $o_1$.

**Example 8.5.** *Figure 8.5 shows the circuit mapped using reversible pebble games for the DAG in Figure 8.4. We assume that* 10 *qubits are available for mapping.* 5 *of the qubits are used for mapping each of the primary inputs $p_1, \ldots, p_5$. We consider mapping the output cone $o_1$ first, followed by $o_2$. Nodes $x_1$, $x_2$ are computed, followed by computing their successor $x_5$. Then node $x_3$ is computed, followed by $x_6$. At this point, none of the qubits are free for mapping rest of the nodes. We uncompute $x_1$ (represented by $x_1^{\psi}$ in the circuit) and use the free qubit to compute $x_4$. Similarly, $x_2$ is uncomputed*

*and $x_7$ is computed on that qubit. The qubit for $x_3$ is now uncomputed to map $x_8$ (which drives the output $o_2$).*

*Now, we begin freeing the allocated qubits in this cone. Node $x_7$ is uncomputed. Since the predecessor $x_3$ of node $x_6$ is not in computed state, $x_6$ cannot be uncomputed. This is due to rule (2) of pebble game. Therefore, $x_3$ is computed. This is followed by uncompute of $x_6$ and $x_3$. For uncomputing $x_5$, predecessors $x_1$ and $x_2$ are computed. Then, $x_5$, $x_2$, $x_1$ and $x_4$ are uncomputed. This completes computation of cone $o_1$.*

*Computation of cone $o_2$ is trivial since there are only 3 STGs to map and 4 qubits are available for mapping. The predecessors $x_4$ and $x_9$ are computed, followed by compute of node $x_{10}$. Then, the intermediate nodes $x_9$ and $x_4$ are uncomputed.*

### 8.3.3 Reversible Pebble Game Heuristic

The procedure takes a LUT-network $N = \langle V, E, F \rangle$ and number of available qubits $Q_A$ as input. The procedure returns the number of qubits required $Q_R$ and the corresponding sequence of operations $S$ to realize the LUT network. Since the procedure is a heuristic, $Q_R$ can be greater that $Q_A$ if the heuristic cannot find a feasible sequence using $Q_A$ available qubits.

Algorithm 13 presents the reversible pebble game heuristic in detail. The output cones are ordered in decreasing order of the size of their cones. Each cone is computed, followed by immediate uncompute of all the nodes, except the output node for that cone. The ELIGIBLE procedure returns $True$ for a node, if all the successors of that node are in computed state.

Algorithm 14 describes the computation of an output cone. Some of the functions used are summarily defined in Table 8.1. Initially, the nodes ($fanIn$) in the cone with only primary inputs as predecessors are eligible for computation. We determine the order of computation between eligible nodes as follows. If a node in a level $l$ in the graph is eligible for compute, then it must be computed before any node

**Algorithm 13:** Reversible Pebble Game Heuristic

```
1  Procedure Pebble(N,Q_A)
       /* Logic Network N = ⟨V = (P ∪ O ∪ G), E, F⟩                    */
       /* Q_A=Available qubits                                          */
2      Initialize empty stack A ;
3      for i in range(1, Q_A) do
4          A.push(i);
5      Q_R = Q_A;
6      Initialize empty queue S;
7      for n_i ∈ PI do
8          q = GetQubit(n_i,PI);
9          S.push_back(PI(n_i,q));
10     Initialize empty set ComputedV;
11     Sort O in descending order of cone(n_o), for n_o ∈ O;
12     for n_o ∈ PO do
13         computed = ComputeOutput(n_o);
14         UncomputeOutput(n_o, computed);
15         S.push_back(PO(n_o,map[n_o]));
       /* Synthesis sequence S, number of required qubits Q_R          */
16     return S, Q_R;
17 Procedure Eligible(node)
18     if n ∈ computed, ∀n ∈ δ⁻(node) then
19         return True;
20     else
21         return False;
```

at a lower level $l-1$. If there are multiple such eligible nodes, then we choose the node that would contribute most towards making its successor eligible for computing. We retrieve the top node for the priority queue, get a qubit to compute this node on, map this node's computation to the qubit, update the priority queue and finally add all the uncomputed successors of this node to the priority queue.

One of the key challenges is to find a free qubit (constant 0) for performing the computation, which is defined in the GetQubit procedure. If there is an available qubit (stored in stack $A$), then it can be used. Otherwise, we search topologically (from higher to lower) for eligible qubits which are not part of the current dependency (*predDep*). During compute, these include just the predecessors of the node

**Table 8.1:** Sub-functions of reversible pebble game heuristic.

| Function | Definition |
|---|---|
| $level(n)$ | $\begin{cases} 0, n \in P \\ max_{n_i \in \delta^-(n)}(level(n_i)+1), n \notin P \end{cases}$ |
| $pending(n)$ | $\lvert n_i \rvert$, $n_i \notin computedV$, $n_i \in \delta^-(n)$ |
| $contrib(n)$ | $\sum_{n_o \in \delta^+(n)} \frac{1}{pending(n_0)}$ |
| $succDep(n) = True$ | $n_o \in computed$, $\forall n_o \in \delta^+(n)$ |
| $succUComp(n) = True$ | $n_o \notin computedV$, $\forall n_o \in \delta^+(n)$ |

under consideration. Out of these eligible nodes, we should not uncompute a node for which a successor has not been computed even once (defined as *succDep* function) âĂŤ this guarantees progress of the algorithm. Moreover, it is intuitive to choose a node with all its successors uncomputed (specified by *succUComp* function) over a node that still has computed successors, since this increases the chances of more eligible nodes to be freed for reuse in the future. Finally, if a node $N$ eligible for uncompute is found, the qubit is freed, the node is removed from the computed node set and an uncompute step is added to the sequence $S$. However, it can happen that none of the computed nodes are eligible to be uncomputed. In that case, a new qubit is added.

The order of computation of a node for the first time, stored in the list *computed*, that is the input parameter to the UncomputeOutput procedure. The nodes are uncomputed in the reverse order of compute. If the node is already uncomputed, then nothing needs to be done. However, if it is computed but not eligible, then its predecessors have to be computed first. Then, the node is uncomputed and the frees qubit $q$ is added to free qubit stack $A$. Also, the node is removed from the computed node set $computedV$.

We describe the priority of computation of a node $n$ below.

1. The level of a node $n$ is defined as :-

$$level(n) = \begin{cases} 0, & n = \text{leaf node} \\ max(level(pred)+1), & \text{for other nodes} \end{cases}$$

**Algorithm 14:** Computing an output cone

```
1  Procedure ComputeOutput(out)
2      fanIn = {n} such that n_f ∈ P, ∀n_f ∈ δ⁻(n) and n ∈ cone(out);
3      pendingQ = prioQ() ;
4      for n ∈ fanIn do
5          priority = ComputePriority(n) ;
6          pendingQ.insert((n,priority));
7      computed = list() ;
8      while out ∉ computed do
9          node = pendingQ.pop() Initialize empty set predDep for n_i ∈ δ⁻(node) do
10             predDep.insert(n_i);
11         q = GetQubit(out,COMP);
12         map[node] = q;
13         computed.add(node);
14         computedV.add(node);
15         S.push_back(COMP(node,qubit));
16         UpdateQueue(pendingQ);
17         for n_o ∈ δ⁺(n) do
18             if n_o ∉ computed and Eligible(n_o) then
19                 pendingQ.insert((n_o, ComputePriority(n_o)));
20     return computed;
21 Procedure ComputePriority(node)
       /* Higher level/contribution of node indicates higher priority.   */
22     return (level(node), contrib(node));
```

2. $contrib(n) = \sum_{s \in succ(n)} \frac{1}{pending(s)}$

   $pending(s)$ = number of predecessors of node $s$ that have not been computed.

3. ComputePriority$(n) = (level(n), contrib(n))$. The higher the $level/contrib$ of a node, the higher is its priority.

### 8.3.4 Finding Available Qubit For Computation

To obtain an available qubit during computation phase of an output:-

1. If there are unused qubits ($k$), then simply allocate the unused qubits.

2. If there are no unused qubits :-

**Algorithm 15:** Algorithm to get free qubit for reversible pebble games.

```
1  Procedure GetQubit(out,type)
2      if !A.empty() then
3          q = A.pop();
4          return q;
       /* Iterate over eligible nodes in topological order (higher level considered first)
          */
5      l = max_{n∈cone(out)}(level(n));
6      while l > 0 do
7          N = None ;
8          E = {n} | ELIGIBLE(n), level(n)=l ,n ∉ predDep;
9          if type == COMP then
10             if ∃n ∈ E | succUComp(n), succDep(n) then
11                 N = n;
12             else if ∃n ∈ E | succDep(n) then
13                 N = n;
14             else if type == UCOMP then
15                 N = n|n ∈ E;
16             if N != None then
17                 q = map[N];
18                 computedV.remove(n);
19                 S.push_back(UCOMP(N,q));
20                 return q;
21         l = l − 1;
       /* No eligible nodes present with all successors computed at least once.      */
22     Q_R = Q_R + 1 ;                                          // New qubit allocated
23     return Q_R;
```

- $eligible(n) = \forall pred(n) \in computed$
- $succDep = True$, if all the successor of the node has been computed at least once.
- $succUncomp = True$, if all successors have been uncomputed.
- Node with $max(level(n))$ that are eligible, with $succDep$. If there exists a node with $succUncomp$ and $succDep$, then this node gets priority over other nodes.

In order to UNCOMPUTEOUTPUT,

1. Determine the reverse compute order —- consider the first time a node has been computed in this ordering.

**Algorithm 16:** Uncomputing an output cone.

```
1  Procedure UncomputeOutput(out, computed)
       /* Store reversed computed list in rComputed, except the output node */
2      Initialize an empty list rComputed ;
3      for i ∈ range(1, len(computed) − 1) do
4          rComputed.push_back(computed[len(computed) − i − 2]);
5      while !rComputed.empty() do
6          n = rComputed[1];
7          rComputed.delete(1);
8          if n ∉ computedV then
9              continue;
10         else if !ELIGIBLE(n) then
11             Initialize empty set predDep;
12             COMPUTEPRED(n);
13         q = map[n];
14         A.push(q);
15         S.push_back(UCOMP(n,q));
16         computedV.remove(n);
17 Procedure ComputePred(n)
18     for n_i ∈ δ⁻(n) do
19         predDep.insert(n_i);
20     for n_i ∈ δ⁻1(n)) do
21         if n_i ∈ computedV then
22             continue;
23         else if ! ELIGIBLE(n_i) then
24             COMPUTEPRED(n_i);
25         q = GETQUBIT(n_i,UCOMP);
26         map[n_i] = q;
27         S.push_back(COMP(n_i,q));
28         computedV.insert(n);
```

2. If $\forall succ(n) \notin computed$, do not uncompute — choose the qubit at the highest level.

3. If qubit is needed, i.e., the node in the compute order does not have all predecessors computed —

   - Choose computed node $n$ at highest level, for uncomputation.
   - Do not choose $n$, if it is part of the current dependency tree.

### 8.3.5 Operation Sequence Optimizations

We present two optimizations based on the outputs returned by the heuristic — number of qubits required $Q_R$ and the sequence of the operations $S$.

**Iterate Heuristic** $O_1$: The number of required qubits $Q_R$ can be higher than that the number of available qubits $Q_A$. When there are no free qubits, the heuristic aggressively tries to unpebble the graph. This leads to additional `COMP`/`UCOMP` operations. Specifically, let us consider the case when $Q_A << Q_R < N_{naive}$. Some of these steps can be avoided by running a second round of the heuristic with $Q_A = Q_R$ as input. This results in considerable reduction in the number of operations, without any significant increase in number of qubits.

**Optimize Sequence** $O_2$: The output of the sequence of step $S$ can be optimized by eliminating redundant operations. Let $S[i]$ denote the $i^{th}$ operation in the sequence. Two operations $S[i]$ and $S[i+1]$ on a node $n|n \in G$ are redundant iff

- $S[i]$ =`COMP`$(n,q)$ , $S[i+1]$ =`UCOMP`$(n,q)$
- $S[i]$ =`UCOMP`$(n,q)$ , $S[i+1]$ =`COMP`$(n,q)$

The redundant operations are repeatedly eliminated, till no further elimination is feasible.

**Example 8.6.** *In Figure 8.5, the uncomputation of node $x_4$ during computation of output cone $o_1$, followed by computation of node $x_4$ for computation of output cone $o_2$ are redundant operations. These operations can be eliminated.*

The LUT networks enforce sharing of logic to minimize the number of nodes in the network, which results in plenty of overlap between cones of the individual outputs. Also, the priority of nodes for computation remains similar, across output cones. Thus, the uncomputation order of multiple nodes for a cone is exactly reverse of the order of computation for these nodes in a different cone, with these nodes common to the cones. This enables the effectiveness of the optimization $O_2$.

**Table 8.2:** Synthesis results on the EPFL arithmetic benchmarks. $\Delta\% = \frac{(\#Q_{orig} - \#Q) * 100}{\#Q_{orig}}$, $\alpha = \frac{\#STG}{\#STG_{orig}}$.

| | | k=6 | | | k=10 | | | k=16 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $\mathbf{N_i/N_o}$ | **#LUT** | $\mathbf{Q_R}$ **(Δ%)** | **#STG(α)** | **#LUT** | $\mathbf{Q_R}$ **(Δ%)** | **#STG(α)** | **#LUT** | $\mathbf{Q_R}$ **(Δ%)** | **#STG(α)** |
| adder | 256/129 | 249 | 386 (23.6) | 14916 (40.4) | 234 | 386 (21.2) | 11608 (34.2) | 207 | 386 (16.6) | 6823 (23.9) |
| adder_bl | 256/129 | 192 | 386 (13.8) | 8190 (32.1) | 189 | 386 (13.3) | 7688 (30.9) | 187 | 386 (12.9) | 7206 (29.4) |
| bar | 135/128 | 512 | 276 (52.7) | 11932 (13.3) | 512 | 276 (52.7) | 11932 (13.3) | 510 | 275 (52.7) | 11736 (13.2) |
| bar_bl | 135/128 | 768 | 309 (63.2) | 16782 (11.9) | 668 | 286 (61.4) | 12056 (10) | 523 | 271 (54.5) | 5788 (6.3) |
| div | 128/128 | 23863 | 10273 (17.1) | 2409454 (50.6) | 23321 | 9954 (17.4) | 2345136 (50.4) | 22910 | 9691 (18.1) | 2295414 (50.2) |
| div_bl | 128/128 | 3271 | 3087 (9.2) | 634450 (98.9) | 3098 | 3091 (4.2) | 597723 (98.5) | 2889 | 2604 (13.7) | 548925 (97.2) |
| log2 | 32/32 | 7579 | 6100 (19.9) | 523327 (34.6) | 2843 | 2347 (18.4) | 185672 (32.8) | 2283 | 1925 (16.8) | 143166 (31.6) |
| log2_bl | 32/32 | 6595 | 5309 (19.9) | 460563 (35) | 3006 | 2440 (19.7) | 203898 (34.1) | 2022 | 1670 (18.7) | 136675 (34.1) |
| max | 512/130 | 721 | 1092 (11.4) | 177191 (135.1) | 389 | 825 (8.4) | 73812 (113.9) | 284 | 746 (6.3) | 44152 (100.8) |
| max_bl | 512/130 | 524 | 934 (9.8) | 113126 (123.2) | 328 | 777 (7.5) | 55765 (106) | 213 | 695 (4.1) | 23467 (79.3) |
| multiplier | 128/128 | 5678 | 4678 (19.4) | 770713 (68.6) | 2977 | 2515 (19) | 378558 (65) | 2724 | 2315 (18.8) | 343227 (64.5) |
| multiplier_bl | 128/128 | 4923 | 4068 (19.4) | 666365 (68.6) | 3291 | 2766 (19.1) | 428239 (66.4) | 2426 | 2079 (18.5) | 306224 (64.8) |
| sin | 24/25 | 1444 | 1181 (19.6) | 73093 (25.5) | 690 | 580 (18.8) | 31165 (23) | 494 | 433 (16.4) | 21330 (22.1) |
| sin_bl | 24/25 | 1262 | 1028 (19.5) | 63825 (25.5) | 534 | 453 (18.8) | 24563 (23.6) | 391 | 339 (18.3) | 17511 (23.1) |
| sqrt | 128/64 | 8084 | 6647 (19.1) | 367572 (22.8) | 7764 | 6391 (19) | 346197 (22.4) | 7688 | 6330 (19) | 337900 (22.1) |
| sqrt_bl | 128/64 | 3076 | 2640 (17.6) | 138108 (22.7) | 2746 | 2377 (17.3) | 120951 (22.3) | 2500 | 2181 (17) | 106640 (21.6) |
| square | 64/128 | 3992 | 3266 (19.5) | 526175 (67) | 3289 | 2714 (19.1) | 424555 (65.8) | 2598 | 2166 (18.7) | 331977 (65.5) |
| square_bl | 64/128 | 3244 | 2669 (19.3) | 422177 (66.4) | 2816 | 2333 (19) | 360257 (65.4) | 2232 | 1877 (18.3) | 282087 (65) |

## 8.4 Experimental Results

We have implemented Reversible Pebble Game Heuristic (RPGH) as part of the *lhrs* command in the reversible logic synthesis framework RevKit [205]. The experiments were performed on the EPFL benchmarks[†], along with their best LUT mapping results (_bl suffix). *lhrs* first derives a LUT network from an And Inverter graph (AIG) which is then mapped into a circuit of STGs using RPGH. We

[†]https://lsi.epfl.ch/benchmarks

set the number of available qubits $Q_A = 0.8(max_{n_v \in V}(cone(n_v)) + N_o) + N_i)$ for each benchmark and use it as input to RPGH. The synthesis results of RPGH are presented in Table 8.2. #LUT denotes the number of LUTs for the given cut size ($k$). $Q_R$ and #STG denote number of qubits and single target gates in the synthesized circuit obtained using RPGH respectively. We compare our results with the existing implementation results ($\#Q_{orig}$, $\#STG_{orig}$) of LHRS, available in RevKit. The RPGH algorithm manages to reduce (denoted by $\Delta\%$) the number of qubits for all the benchmarks, which is the primary goal of this paper. Naturally, the number of single-target gates (denoted by $\alpha$) significantly increases using the proposed algorithm, that leads to increased quantum costs (see, e.g., [206]).

To obtain a Clifford+T quantum circuit from the STG mapped circuit, each single-target gate has to be mapped into a Clifford+T quantum circuit. The RPGH algorithm is compatible to all varieties of STG to Clifford+T mapping algorithms (e.g., [207]). As a representative example, we show the results of mapping the synthesized STG circuit to Clifford+T circuit using various algorithms for the *sin* benchmark in Table 8.3. The T-count of the synthesized circuits are reported, along with the qubit and STG count. Any improvement in the mapping algorithm of single-target gates to quantum circuits would directly benefit our algorithm, for reducing T-count. For $k = 6$ as an example, we can see that using the *direct+def* mapping script results in lower T-count than *mindb+def_wo4* script while having the same qubit count.

The experiments were executed on a Intel(R) Xeon(R) CPU E5-1650 v2 running at 3.50 GHz with 16 GB main memory and *gcc* version 5.4.0 with Ubuntu 16.04.9 as the operating system. RPGH takes a small fraction of the execution time, while the decomposition of the STG to the Clifford+T library in the second step requires most of the overall runtime, especially for larger LUT sizes $k$.

**Table 8.3:** T-count of synthesized circuit using various STG to Clifford+T mapping algorithms [207] for *sin* benchmark.

| | | Mapping | **direct** | | **mindb** | |
|---|---|---|---|---|---|---|
| | | Script | def | def_wo4 | def | def_wo4 |
| $k$ | $\mathbf{Q_R}$ | **#STG** | **T-count** | **T-count** | **T-count** | **T-count** |
| 6 | 1181 | 73093 | 2331631 | 2318301 | 2927440 | 2850017 |
| 10 | 580 | 31165 | 9091604 | 9123544 | 14631353 | 14632203 |
| 16 | 433 | 21330 | 93594652 | 94909560 | 43922561 | 43949572 |

Using a greater cut size ($k$), the number of LUTs in the LUT network is lesser. Since the RPGH algorithm uses the LUT network as input and each LUT gets mapped to an STG, a smaller network helps RPGH to map the network using smaller number of qubits, with lower number of STGs. However, since a $k$-input LUT would map to a $k$-input STG, using a greater cut size results in an STG with greater number of inputs and thus has a higher quantum cost when mapped using the Clifford+T library. Finally, we demonstrate the space ($Q_R$) and time (T-count) trade-off feasible by using RPGH

**Table 8.4:** Qubit count ($Q_R$) and T-count of synthesized circuit for varying number of available qubits [$Q_A = 0.8(max_{n_v \in V}(cone(n_v)) + N_o) + N_i) = \varphi(1421 + 25) + 24$] for *sin* benchmark with cut size $k = 6$.

| $\varphi$ | **0.3** | **0.4** | **0.5** | **0.6** | **0.7** | **0.8** |
|---|---|---|---|---|---|---|
| $\mathbf{Q_A}$ | 458 | 603 | 747 | 892 | 1037 | 1181 |
| $\mathbf{Q_R}$ | 590 | 604 | 747 | 896 | 1037 | 1181 |
| **T-count** | 3415534 | 3413100 | 3288854 | 3147188 | 2634309 | 2331631 |

in Table 8.4. By varying $\varphi$ from 0.3 to 0.8, we vary the input number of qubits $Q_A$ to RPGH. The RPGH algorithm maps the circuit with $Q_R$ qubits, which is close to the input $Q_A$. This demonstrates the effectiveness of the algorithm in using reversible pebbles game to meet input qubit constraint. Using a lower number of qubits for mapping leads to a higher T-count and vice-versa. Similar results are observed for other benchmarks as well, but not reported due to lack of space. To the best of our knowledge, the proposed RPGH algorithm is the first one to permit a constraint on number of qubits as an input for quantum logic synthesis.

## 8.5 Discussion

Implementation of scalable quantum circuits is among the most significant scientific challenges of current times. Existing design automation flows for quantum circuits tend to emphasize on the number of gates and logical depth. In contrast, we draw attention to the reversible pebble game, which presents an opportunity to reduce the qubits. In chapter, we present a heuristic for lowering the qubits integrated within a scalable, hierarchical logic synthesis flow. This reduces the number of qubits by up to 62.3% compared to the baseline, state-of-the-art synthesis techniques.

In physical implementation of quantum circuits, a long-distance interaction between two qubits is undesirable since, it can be interpreted as a noise. Therefore, multiple quantum technologies and quantum error correcting codes strongly require the interacting qubits to be arranged in a nearest neighbor (NN) fashion. The following chapter presents an optimal algorithm to transform a given quantum circuit into a NN-complaint circuit, for any arbitrary topology of interaction.

# 9 Topology-constrained Quantum Technology mapping

A significant hurdle towards realization of practical and scalable quantum computing is to protect the quantum states from inherent noises during the computation. A major challenge towards the realization of practical and scalable quantum computing is to achieve quantum error correction [208]. Long-distance interacting qubits is particularly susceptible to noise. Therefore, prominent quantum technologies and quantum error correction codes, e.g. surface codes [209] require that the quantum gates must be formed with a nearest neighbour (NN) interaction. The current literature on converting a given quantum circuit to a NN-arranged one mainly considers chained qubit topologies or Linear Nearest Neighbor (LNN) topology. However, practical quantum circuit realizations, such as Nuclear Magnetic Resonance (NMR), may not have an LNN topology. This is the exact gap that we address in this chapter.

Conversion of a quantum circuit to a NN one can be achieved by using SWAP gates. These SWAP gates allow for making all control lines and target lines adjacent and, by this, help to convert a given quantum circuit to a nearest neighbor one. We present an Integer Linear Programming (ILP) formulation for achieving minimal logical depth while guaranteeing the nearest neighbor arrangement

between the interacting qubits for arbitrary qubit topology. We substantiate our claim with studies on diverse network topologies and prominent quantum circuit benchmarks.

## 9.1 Motivation

As noted in [210], the qubit topologies, on which the quantum circuit is to be mapped, are not necessarily of LNN structure. We provide a few examples here.

Recently, quantum error detection code is demonstrated on a square lattice [211]. It also highlights the fact that for a classical bit-flip, linear array of qubits suffices, while for general fault detection, extending to higher-dimensional lattice structures is needed.

Nuclear Magnetic Resonance (NMR) quantum computing achieved early success with realization of Shor's factorization algorithm [24]. Liquid state NMR quantum computing utilizes the atomic spin states to realize the qubit and hence, has the molecular structures as qubit topologies. Solid state NMR has been also demonstrated [212] using crystal of $NaNO_3$, essentially leading to molecular topologies.

A recent proposition for scalable quantum computer indicates that multiple, parallel quantum gates can be formed between distant qubits by controlling the lasers on Trapped Atomic Ions [213].

Harnessing atomic spins in endohedral fullerene molecules as qubits have also been reported [214]. It has been further argued that molecular structures serve as a natural candidate for quantum technology by holding superpositions for longer period and ability to scaffold multiple molecules in a larger array.

Hence, an automated algorithm for achieving nearest neighbour interactions for a given quantum circuit while mapping on diverse qubit topologies is of significant practical interest. To obtain NN quantum circuits, a cascade of adjacent SWAP gates can be inserted in front of each gate $g$ with non-adjacent circuit lines in order to shift the control line of $g$ towards the target line, or vice versa, until they are adjacent. This is shown using the following example.

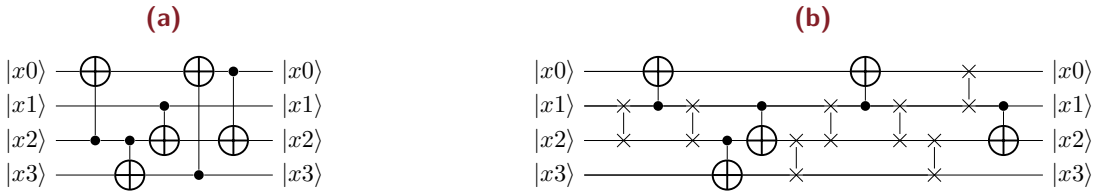



**Figure 9.1:** (a) A quantum circuit with 5 gates. We refer to the gates as $g_i$ from left to right, with the left-most gate referred by $g_i$. (b) Corresponding NN-compliant circuit obtained by inserting swap gates.

**Example 9.1.** *Consider the circuit depicted in Fig. 9.1a. As can be seen, gates $g_1$, $g_4$, and $g_5$ are non-adjacent. Thus, in order to make this circuit nearest neighbor compliant, SWAP gates in front and after all these gates are inserted as shown in Fig. 9.1b.*

In this chapter, we present an ILP-based algorithm to realize depth-optimal nearest neighbour quantum circuits for arbitrary qubit interaction topologies. Our algorithm is also applicable, naturally, to simpler structures, such as LNN.

## 9.2 Preliminaries And Problem Statement

In this section, we introduce the notations and terminologies for formally defining the nearest-neighbor optimization problem of quantum computing. Thereafter, we present three variants of the problem.

**Definition 9.1** (quantum circuit)**.** *A **quantum circuit**, defined over n-qubits $q_1$, $q_2$,...,$q_n$ is a series of levels $L_i$, where each level $L_i$ consists of a set of quantum gates $G_i^1$, $G_i^2$, $\cdots$, $G_i^k$ with each gate $G_i^j$ operating on one or more qubits. Any two pair of gates $G_i^j$ and $G_i^k$ in a level $L_i$ do not operate on any common qubit and therefore can be executed in parallel. We assume that each level $L_i$ takes one cycle to execute. A quantum circuit with k levels has a delay of k cycles.*

Given a quantum gate with $m$-control lines $l_1, ..., l_m$ and target line $l_t$, qubits $q_l$ and $q_t$ have to be nearest-neighbors, $1 \leq l \leq m$. For level $L_i$, we define **interaction** $I_i$ as the set of nearest neighbors for

*xor5_254.real file from RevLib

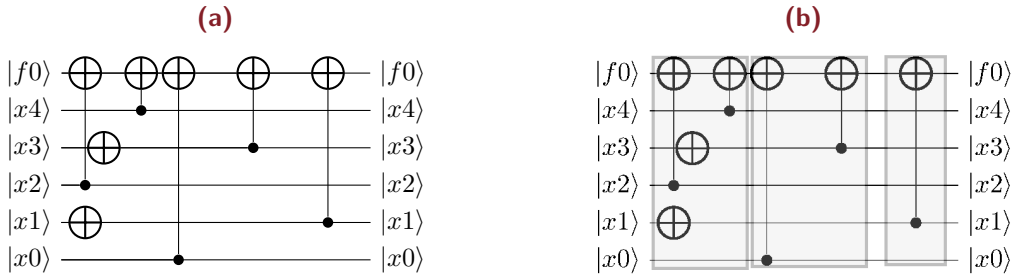



**Figure 9.2:** (a) A quantum circuit*with delay 5. (b) Interactions for block size $b = 2$.

the all the gates in $L_i$. The levels and corresponding interactions of a quantum circuit is determined using Algorithm 17.

**Example 9.2.** *Figure 9.2a shows a quantum circuit with 5 two-input Toffoli gates and 2 CNOT gates. The circuit has 5 levels and hence has a delay of 5.*

$L_1$ : *[t2 x2 f0, t1 x1, t1 x3]*

$L_2$ : *[t2 x4 f0]*

$L_3$ : *[t2 x0 f0]*

$L_4$ : *[t2 x3 f0]*

$L_5$ : *[t2 x1 f0]*

*Corresponding to level $L_1$, interaction $I_1$ is [(x2, f0), (x1), (x3)]. Similarly, $I_2, I_3, I_4$ and $I_5$ is [(x4,f0)], [(x0,f0)], [(x3,f0)] and [(x1,f0)] respectively.*

Physically, qubits can be arranged in various topologies, as discussed previously. Such topologies allow interaction between only between some pairs of qubit positions. We introduce this constraint in the form of a topology graph.

**Definition 9.2** (Topology graph)**.** *A topology graph is an ordered pair $T = \langle T_\mathcal{V}, T_\mathcal{E} \rangle$. $T_\mathcal{V}$ is the vertex set, where each vertex $v \in T_\mathcal{V}$ represents a physical location where one qubit can reside. $T_\mathcal{E}$ is the edge-set, which contains a set of edges. An edge $e_{vw} \in T_\mathcal{E}$ indicates that qubit at location/vertex $v$ and $w$ can interact. In other words, qubits at location $v$ and $w$ are nearest-neighbors (NN).*

Figure 9.3 presents various topologies. The minimum number of nodes for the smallest graph of each

**Algorithm 17:** Level Computation Algorithm

```
1  Procedure ComputeLevel(devUseTable)
2      levelList = [];
3      processedGate = set();
4      L = set();
5      Lvar = set();
6      for Gi ∈ QCkt do
7          if Gi ∉ processedGate then
8              if Gi.var ⋂ Lvar == ϕ then
9                  L.add(Gi);
10                 Lvar.add(Gi.var);
11                 processedGate.add(Gi);
12                 for Gj ∈ QCkt do
13                     if Gj.var ⋂ Lvar == ϕ then
14                         L.add(Gj);
15                         Lvar.add(Gj.var);
16                         processedGate.add(Gj)
17                 levelList.add(L);
18                 L = set();
19                 Lvar = set();
20 return reassignMap;
```

topology is presented in Table 9.1. Given a quantum circuit with $n$-qubits, and a specific topology, we use the smallest topology graph $T$ such that $T_{\mathcal{V}} \geq n$ for realizing the quantum circuit.

**Definition 9.3** (Qubit Configuration)**.** *A qubit configuration $C_t$ is the set of ordered tuples $(q_i, v)$, which indicates that in cycle $t$, qubit $q_i$, is at location $v$, $1 \leq i \leq n$ and $v \in T_{\mathcal{V}}$. Configuration $C_0$ represents the initial configuration.*

### 9.2.1 Problem Statement

We now define three variants the nearest-neighbor optimization problem of quantum circuits for arbitrary topologies and also present the relation between the variants.

**Problem $P_1$:** Given an initial configuration $C$ of $n$-inputs, an interaction $I$ and a topology graph $T$,

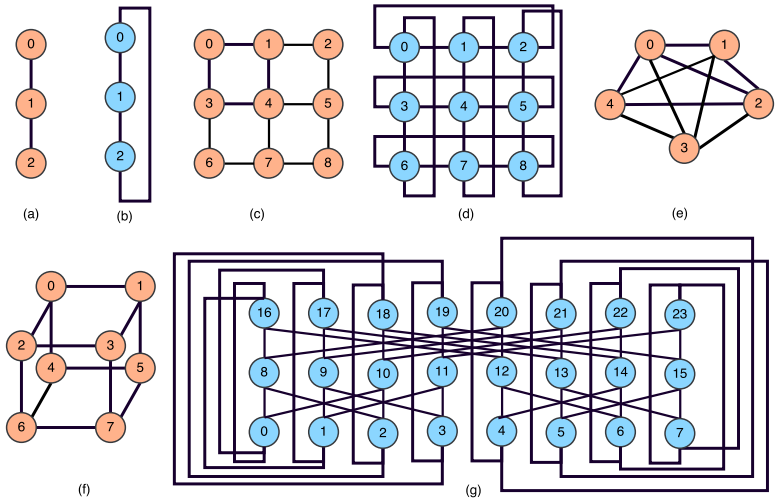



**Figure 9.3:** Some representative qubit interaction topology graphs. (a) 1D-nearest neighbor (b) Cycle (c) 2D-Mesh (d) Torus (e) Fully connected graph (f) 3D-Grid (g) Cyclic butterfly network.

**Table 9.1:** Minimum number of nodes present in the smallest graph for each topology.

| **Topology** | **Min. #Nodes** |
|---|---|
| 1D | 2 |
| Cycle | 3 |
| 2D-Mesh | 9 |
| Torus | 9 |
| 3D-Grid | 8 |
| Cyclic butterfly | 24 |

the objective is to determine the series of swap gates needed to transform the location of the qubits from configuration $C$ such that all qubit pairs in interaction I are nearest-neighbors and the delay due to insertion of swap gates is minimum.

**Problem $P_2$:** Given an initial configuration $C$ of $n$-inputs, a series of interactions $I_1, I_2, \ldots, I_k$ and a topology graph $T$, the objective is to determine the series of swap gates needed to transform the location of the qubits from configuration $C$ such that all qubits pairs in interaction $I_1$ are nearest-neighbor, and then again location of qubits are transformed to be nearest neighbors for $I_2$ and so on, till interaction $I_k$ is met and the delay due to insertion of swap gates is minimum for the overall problem.

**Table 9.2:** Depth $D$ and Space $S$ complexity of realizing arbitrary permutations using a given topology.

| Topology | Degree | D | S |
|---|---|---|---|
| Fully Connected Graph | $n$-1 | 1 | 1 |
| 1D nearest-neighbor [215] | 2 | $2n$-3 | 1 |
| 2D nearest-neighbor [216] | 4 | $O(\sqrt{n})$ | 1 |
| Cyclic butterfly network [217] | 4 | 6log $n$ | 2 |
| Hypercube [216] | log $n$ | $O(\log^2 n)$ | 1 |

**Problem $P_3$:** Given an initial configuration $C$ of $n$-inputs, a series of levels $L_1, L_2, \ldots, L_k$ and a topology graph $T$, the objective is to determine the series of swap gates needed to transform the location of the qubits from configuration $C$ such that all qubits pairs in interaction $I_1$ (corresponding on level $L_1$) are nearest-neighbor, and then again location of qubits are transformed to be nearest neighbors for $I_2$ (corresponding on level $L_2$) and so on, till interaction $I_k$ (corresponding on level $L_k$) is met and the combined delay of swap gates and gates present in the actual circuit is minimum.

The Problem formulation $P_1$ has been popularly used for showing effectiveness of various topologies to realize arbitrary permutations. Table 9.2 shows the depth and space requirements for realization of arbitrary permutations on various topologies.

Problem $P_2$ with $k = 1$ is equivalent to Problem $P_1$. Therefore, finding an optimal solution for $P_2$ with $k = 1$ is equivalent to solving $P_1$. Problem $P_2$ does not consider the scheduling of the swap gates in parallel to quantum gates present in the original circuit, if possible. $P_2$ transforms the qubit locations on the topology graph such that the interactions needed to execute a level in quantum circuit is met. Problem $P_3$ addresses this issue and considers the quantum gates as well and can find the optimal solution with minimum delay.

**Theorem 9.1.** *For a topology graph $T$ and a quantum circuit $C$ with $k+1$ levels, the delay $d_I$ of solution $S_I$ obtained by optimally solving problem $P_2$ is at most $k$-cycles more than the delay $d_O$ of optimal solution $S_O$ of problem $P_3$ i.e. $d_I - d_O \leq k$.*

*Proof.* Consider an initial configuration and a quantum circuit two levels. Let us assume that the delay of solution be $d_I$ and $d_O$ be the optimal solution. Optimal solution for $P_3$ would have been able to insert additional gates at only one level $L_0$, which was not considered by $P_2$. If $d_I - d_O > 1$, this would imply that $d_I$ is not the optimal solution for $P_2$, since there exists a solution to solve $P_2$ with $d_O + 1$ delay which is a contradiction. This idea can be extended for any number of gates to derive Theorem 9.1. ■

It is possible to split the circuits into equal size blocks, with $b$-levels in each block, except the last block which might have less than $b$ levels. Figure 9.2b shows the blocks with size $b$=2, with the last block having a single interaction. Each block can solved using $P_2$ or $P_3$ to make the qubits nearest-neighbors and the output configuration of the solution is used as input configuration for the next block. For a quantum circuit with $k+1$-levels and $b \geq k$,

- Optimal solution with minimum delay $d_O$ can be determined using $P_3$.
- A bounded delay solution with delay $d_I$ can be determined using $P_2$ such that $d_I - d_O \leq k$.

Various suboptimal solutions can be obtained using $b < k$, using both $P_2$ and $P_3$. Choosing a small block size $b$ makes it easier to solve each sub-problem and therefore it becomes feasible to solve the nearest neighbor technology mapping problem for circuits with large number of gates. Corresponding to circuit in Figure 9.2a, the 1D-nearest neighbor compliant circuit, obtained using problem formulation $P_2$ and $P_3$ for block size $b = 4$, is shown in Figure 9.4 (a) and Figure 9.4 (b) respectively.

### 9.2.2 Related Works

To the best of our knowledge, [210] and [218] were the first to look into arbitrary topologies for quantum circuits with nearest neighbour constraints. So far, most of the other works in this domain have concentrated on 1D qubit layout or 2D qubit lattice structures [219, 220]. Heuristics [221, 222]

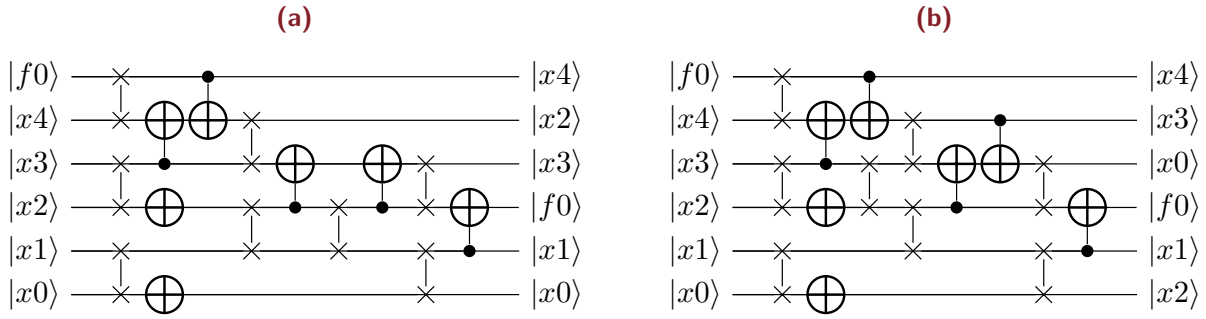



**Figure 9.4:** 1D-Nearest neighbor optimization solution. (a) $P_2$ Solution with $b = 4$. (b) $P_3$ Solution with $b = 4$.

and exact [223] solutions are proposed, which balance the LNN conversion with other performance metrics. It is pointed out in [224] that the problem of nearest neighbour quantum circuit construction is equivalent to an NP-complete problem. Hence, it is unlikely that this problem can be solved optimally for large instances.

The work presented in [218] focuses on identifying the qubit topology best suited for a given quantum circuit placement. In contrast, our focus is towards evaluating a given qubit topology and performing mapping on it. This particular problem has been dealt with in [210] with examples taken from liquid state NMR molecules as the topologies. There, a graph partitioning-based approach is proposed and it is claimed to be asymptotically optimal for the case of chain nearest neighbour architecture. We address the same problem, by formulating it as an instance of ILP and show that optimal results are achievable for a wide variety of benchmarks and different topologies.

Independently, efficient qubit topology identification and the mapping flows for specific interaction graphs have been done in [216, 217]. For example, it is proved that for cyclic butterfly topology, the depth overhead for mapping a given quantum gate to a nearest neighbour one is $6 \log n$. Subsequently, the mapping algorithm is also derived.

In parallel to the previous works, efficient LNN circuit construction has been studied for important quantum benchmarks, such as, quantum error correction [225] for Clifford+T gates [226]. In this work, we are primarily interested in the automated flow and for generic quantum circuits. Communication

**Table 9.3:** Parameters and constants used in the ILP for topology-constrained quantum technology mapping.

| Param/const. | Description |
|---|---|
| $G$ | Toplogy graph |
| $C$ | Input/start configuration |
| $n$ | Number of inputs |
| $k+1$ | Number of levels |
| $L_i$ | Number of qubit interaction pairs in level $i$ |
| $T$ | Maximum number of cycles used for the problem |

and computation over networks is of major interest in quantum networks [227] as well as for classical telecommunication networks. The problem of permutation routing on variety of graphs has been studied in the past [228–230].

## 9.3 Methodology

In this section, we initially present an ILP formulation for the problem $P_2$. Description of the variables used in the formulation is presented summarily in Table 9.4. Thereafter, we present the modified ILP for problem $P_3$.

### 9.3.1 ILP formulation for $P_2$

**Objective function:**

$$\text{Minimize } delay \tag{9.1}$$

$$\sum_{t=0}^{T} m_{k,t} - delay = 0 \tag{9.2}$$

**Chronological interaction constraints:** If an interaction is met in cycle $t$, then the status should not change to not met after that cycle. In addition, interaction $i$ must be met before $i-1^{th}$ interaction is

**Table 9.4:** Variables used in ILP formulation for NN-compliant circuit construction.

| Variable | Type | Description |
|---|---|---|
| $a_{i,t}$ | bin | 1 indicates gates in Level $i$ are scheduled in cycle $t$ |
| $b_{q,t}$ | bin | 1 indicates qubit $q$ cannot be involved in a swap in cycle $t$ |
| $b_{v,q,t}$ | bin | 1 indicates qubit $q$ in location $v$ cannot be involved in a swap in cycle $t$ |
| $c_{v,q,t}$ | bin | 1 indicates qubit $q$ will move to new location $v$ in cycle $t$ |
| $delay$ | int | Delay due to insertion of swap gates |
| $eb_{I_i,t}$ | bin | 1 indicates interaction $I_i$ has been met in cycle $t$ and gates of level $i$ can be placed in the current or following cycles. |
| $m_{i,t}$ | bin | 0 indicates Interaction $i$ met in cycle $t$ |
| $n_{p,q,t}$ | bin | 1 indicates qubit $p$ and $q$ are NN in cycle $t$ |
| $p_{(p,v),(q,w),t}$ | bin | 1 indicates qubit $p$ is in location $v$ and $q$ is in location $w$ in cycle $t$ |
| $sb_{m,n,t}$ | bin | 1 indicates swap is not permitted between locations $m$ and $n$ in cycle $t$ |
| $u_{v,q,t}$ | bin | 1 indicates qubit $q$ will remain in location $v$ in cycle $t$ |
| $x_{v,q,t}$ | bin | 1 indicates qubit $q$ is in location $v$ in cycle $t$ |

met.

$$m_{i,t+1} - m_{i,t} \geq 0 \qquad 0 \leq t \leq T-1, 0 \leq i \leq k \tag{9.3}$$

$$m_{i+1,t} - m_{i,t} \geq 0 \qquad 0 \leq t \leq T, 0 \leq i \leq k-1 \tag{9.4}$$

**Successful interaction constraints:** An interaction is met if all the qubit pairs in the interaction are nearest neighbors. If an interaction has been met in cycle $t$, then in all cycles $t' > t$, the qubit positions do not matter any longer.

$$L_i.m_{i,t} + (\sum_{(p,q)\in I_i} n_{p,q,t}) + (\sum_{t'=0}^{t-1} L_i.(1 - m_{i,t'})) \geq L_i \qquad 0 \leq t \leq T \tag{9.5}$$

**Nearest neighbor constraints:** Two qubits $p$ and $q$ are nearest neighbors if the qubits are in two locations $v$ and $w$ respectively or in $w$ and $v$ respectively, such that $(v, w) \in G_{\mathcal{E}}$.

$$p_{(p,v),(q,v),t} = x_{v,p,t} \wedge x_{w,q,t} \qquad (p,q) \in I, (v,w) \in G_{\mathcal{E}} \tag{9.6}$$

$$p_{(p,w),(q,v),t} = x_{w,p,t} \wedge x_{v,q,t}; \qquad (p,q) \in I, (v,w) \in G_{\mathcal{E}} \tag{9.7}$$

$$n_{p,q,t} = \vee_{(v,w)\in G_{\mathcal{E}}}(p_{(p,v),(q,w),t} \vee p_{(p,w),(q,v),t}) \qquad (p,q) \in I \tag{9.8}$$

**Qubit position update constraints:** A qubit $q$ is at location $v$ in cycle $t+1$ if it was in location $v$ in cycle $t$ and there were no swaps performed involving the location $v$ or if $q$ was in a location $w$ which is nearest neighbor with $v$ and a swap was performed between $v$ and $w$.

$$u_{v,q,t+1} = (\wedge_{(v,w)\in G_{\mathcal{E}}}(1 - s_{v,w,t})) \wedge x_{v,q,t}; \tag{9.9}$$

$$c_{v,q,t+1} = \vee_{(v,w)\in G_{\mathcal{E}}} s_{v,w,t} \ \wedge \ x_{w,q,t} \tag{9.10}$$

$$x_{v,q,t+1} = u_{v,q,t+1} \ \vee \ c_{v,q,t+1} \tag{9.11}$$

**Qubit location and swap constraints:** A qubit $q$ can be at exactly one position in any given cycle. In a given cycle, a location can be involved in atmost one swap.

$$\sum_{v \in G_{\mathcal{V}}} x_{v,q,t} = 1; \qquad 0 \leq t \leq T, q \in Q \tag{9.12}$$

$$\sum_{(v,w) \in G_{\mathcal{E}}} s_{v,w,t} \leq 1; \qquad 0 \leq t \leq T, v \in G_{\mathcal{V}} \tag{9.13}$$

**Initialization constraints:** A qubit $q$ is at location $v$ in cycle 0, based on input configuration $C$.

$$x_{v,q,0} = 1; \qquad (v,q) \in C \tag{9.14}$$

This concludes the description of the ILP formulation for problem $P_2$. The following subsection presents the modifications needed in the ILP for optimally solving $P_3$.

### 9.3.2 ILP formulation for $P_3$

**Objective function:**

$$\text{Minimize} \sum_{i=0}^{k} \sum_{t=0}^{T} t.a_{i,t} \tag{9.15}$$

**Level scheduling constraints:** Each level can be scheduled/activated exactly once.

$$\sum_{t=0}^{T} a_{i,t} = 1; \qquad 0 \leq i \leq k \tag{9.16}$$

Only one level can be activated per time step.

$$\sum_{i=0}^{k} a_{i,t} = 1; \qquad 0 \leq t \leq T \tag{9.17}$$

Activation for a level $i$ can happen only if corresponding interaction $i$ is met.

$$a_{i,t} + m_{i,t} \leq 1; \qquad 0 \leq t \leq T, 0 \leq i \leq k \tag{9.18}$$

**Swap blocking constraints:** If an interaction $i'$ is met and all the gates in any Level $i$ such that $(i < i')$ have been scheduled, then swaps involving the qubits in interaction $i$ cannot be performed and interaction $i'$ is blocked till Level $i$ has been scheduled. Qubit involved in an interaction $i$ cannot be swapped in the cycle, when the Level $i$ is scheduled.

$$eb_{i',t} = a_{i,t} \wedge (1 - m_{i',t}); \qquad 0 \leq i \leq k-1, i+1 \leq i' \leq k, 0 \leq t \leq T \tag{9.19}$$

$$b_{q,t} = \vee_i (a_{i,t} \vee eb_{i,t}); \qquad \forall i \; \exists \; q \in I_i, 0 \leq t \leq T \tag{9.20}$$

$$b_{v,q,t} = b_{q,t} \wedge x_{v,q,t} \qquad 0 \leq t \leq T \tag{9.21}$$

$$sb_{m,n,t} = \vee_q (b_{m,q,t} \vee b_{n,q,t}); \qquad \forall q \in Q, 0 \leq t \leq T \tag{9.22}$$

In addition to these constraints, *Chronological interaction constraints, Successful interaction constraints, Nearest neighbor constraints, Qubit position update constraints, Qubit location and swap constraints* and *Initialization constraints* presented in ILP formulation for $P_2$ are applicable to $P_3$. This completes the description of the ILP formulation of $P_3$.

## 9.4 Experimental Results

In this section, we present the benchmarking results for multiple quantum circuits from [231] for various topologies. We used Gurobi [121] as ILP solver. For all the block sizes, we set TIME_LIMIT parameter of Gurobi to 600 seconds to limit the time of execution of the solver, except for solving full circuit optimization for which we set TIME_LIMIT to 7200. We set the number of threads parameter in Gurobi to 8. For the experiments, we used 64-bit Ubuntu 14.04 running on Intel(R) Xeon(R) CPU

**Table 9.5:** Realization of all configurations of 4-qubits for 1D-topology.

| Config. | Swap Count | Delay | Config. | Swap Count | Delay | Config. | Swap Count | Delay |
|---|---|---|---|---|---|---|---|---|
| a b c d | 0 | 0 | b c a d | 2 | 2 | c d a b | 2 | 1 |
| a b d c | 1 | 1 | b c d a | 3 | 3 | c d b a | 1 | 1 |
| a c b d | 1 | 1 | b d a c | 3 | 2 | d a b c | 3 | 3 |
| a c d b | 2 | 2 | b d c a | 2 | 2 | d a c b | 2 | 2 |
| a d b c | 2 | 2 | c a b d | 2 | 2 | d b a c | 2 | 2 |
| a d c b | 3 | 3 | c a d b | 3 | 2 | d b c a | 1 | 1 |
| b a c d | 1 | 1 | c b a d | 3 | 3 | d c a b | 1 | 1 |
| b a d c | 2 | 1 | c b d a | 2 | 2 | d c b a | 0 | 0 |

E5-1650 v2@3.50GHz with 15.6 GB RAM.

Table 9.5 demonstrates realization of all possible configurations of 4-variables for 1D topology. The initial configuration is assumed to [a,b,c,d]. *Swap Count* and *Delay* is the number of swap gates required and the corresponding delay to realise the target configuration respectively. This table has been obtained using Problem formulation $P_2$ with $k$=1. We would like to highlight that configuration [a b c d] and [d c b a] are identical since for both the configuration the pair of nearest-neighbor variables is same.

Table 9.6 presents the results of 1D-Nearest neighbors for multiple block size $b$ ={1, 2, 4, 8, 16}. The column Tech. indicates whether the solution for a benchmark is obtained using the problem formulations $P_2$ or $P_3$. Using a large block size is expected to reduce overall circuit delay, since the optimization solver can search a larger solution space to obtain optimal solution in that space instead of hitting a locally optimal solution. For circuit *xor5_254* , the delay for block side $b = 1$ is 9 while that with block size $b = 4$ is 7. On the other hand, by using a smaller block, it is possible to obtain a feasible solution within the time limits specified for the solver, since the solver has to solve a smaller instance of the formulated ILP. For example, solutions could not be obtained for $b \geq 4$ within the specified time limits for circuit *alu-bdd_288*. It should be noted that for all block sizes $b < L$, where $L$ is the number of levels in the circuit, the overall circuit is not guaranteed to have least delay, even when using formulation $P_3$ since combining the optimal solutions of the subproblems does not guarantee globally optimal solution.

**Table 9.6:** Benchmarking results corresponding to various block sizes for 1D-topology. #Var : Number of variables, #Gates: Number of Gates, #L : Number of levels in the quantum circuit, Tech.: Problem formulation technique., #S : Number of Swap Gates and D: Delay incurred due to swap gates.

| **Benchmark** | **#Var** | **#Gates** | **#L** | **Tech.** | **b=1** | | **b=2** | | **b=4** | | **b=8** | | **b=16** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | #S | D | #S | D | #S | D | #S | D | #S | D |
| 3_17_14 | 3 | 6 | 6 | $P_2$ | 3 | 9 | 3 | 8 | 3 | 8 | 3 | 8 | 3 | 8 |
| | | | | $P_3$ | 3 | 9 | 3 | 8 | 3 | 8 | 3 | 8 | 3 | 8 |
| 4gt11-v1_85 | 5 | 4 | 3 | $P_2$ | 5 | 7 | 5 | 7 | 8 | 7 | 8 | 7 | 8 | 7 |
| | | | | $P_3$ | 5 | 7 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 4mod5-v1_25 | 5 | 4 | 3 | $P_2$ | 3 | 5 | 3 | 5 | 3 | 5 | 3 | 5 | 3 | 5 |
| | | | | $P_3$ | 3 | 5 | 3 | 5 | 4 | 5 | 4 | 5 | 4 | 5 |
| alu-bdd_288 | 7 | 9 | 8 | $P_2$ | 22 | 19 | 19 | 17 | — | — | — | — | — | — |
| | | | | $P_3$ | 22 | 19 | 26 | 18 | — | — | — | — | — | — |
| ex-1_166 | 3 | 4 | 4 | $P_2$ | 1 | 5 | 1 | 5 | 1 | 5 | 1 | 5 | 1 | 5 |
| | | | | $P_3$ | 1 | 5 | 1 | 5 | 1 | 5 | 1 | 5 | 1 | 5 |
| ex1_226 | 6 | 7 | 5 | $P_2$ | 7 | 9 | 7 | 9 | 8 | 9 | 8 | 9 | 8 | 9 |
| | | | | $P_3$ | 8 | 9 | 8 | 9 | 8 | 7 | 8 | 7 | 8 | 7 |
| fredkin_7 | 3 | 1 | 1 | $P_2$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | | $P_3$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| graycode6_48 | 6 | 5 | 5 | $P_2$ | 0 | 5 | 0 | 5 | 0 | 5 | 0 | 5 | 0 | 5 |
| | | | | $P_3$ | 0 | 5 | 0 | 5 | 0 | 5 | 2 | 5 | 2 | 5 |
| ham3_103 | 3 | 4 | 4 | $P_2$ | 4 | 7 | 3 | 6 | 3 | 6 | 3 | 6 | 3 | 6 |
| | | | | $P_3$ | 4 | 7 | 3 | 6 | 3 | 6 | 3 | 6 | 3 | 6 |
| mod5d2_70 | 5 | 8 | 7 | $P_2$ | 6 | 12 | 6 | | 8 | 12 | 9 | 10 | 10 | 10 |
| | | | | $P_3$ | 6 | 12 | 8 | 10 | 6 | 12 | 10 | 10 | 10 | 10 |
| one-two-three-v3_101 | 5 | 8 | 7 | $P_2$ | 11 | 13 | 11 | 13 | 10 | 13 | 10 | 13 | 10 | 13 |
| | | | | $P_3$ | 12 | 15 | 14 | 14 | 9 | 12 | 9 | 12 | 9 | 12 |
| peres_9 | 3 | 2 | 2 | $P_2$ | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 |
| | | | | $P_3$ | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 |
| rd32_272 | 5 | 6 | 5 | $P_2$ | 12 | 11 | 9 | 11 | 9 | 11 | 9 | 11 | 9 | 11 |
| | | | | $P_3$ | 10 | 11 | 12 | 11 | 9 | 11 | 9 | 11 | 9 | 11 |
| toffoli_double_4 | 4 | 2 | 2 | $P_2$ | 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |
| | | | | $P_3$ | 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |
| xor5_254 | 6 | 7 | 5 | $P_2$ | 7 | 9 | 7 | 9 | 6 | 9 | 9 | 8 | 9 | 8 |
| | | | | $P_3$ | 8 | 9 | 8 | 8 | 8 | 7 | 8 | 7 | 8 | 7 |

**Table 9.7:** Benchmarking results for *4gt10-v1_81* using $P_3$ formulation, $w = 1$. #G: Number of gates in the circuit, D: Delay of the circuit and #S: Number of swap gates.

| | #G | D | 1D | | Cycle | | 2D-Mesh | | Torus | | 3D-Grid | | CBN | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #S | D | #S | D | #S | D | #S | D | #S | D | #S | D |
| Original | 6 | 6 | NF | | NF | | 11 | 11 | 5 | 9 | 7 | 11 | — | — |
| Decomposed | 12 | 12 | 25 | 26 | 14 | 21 | 10 | 22 | 6 | 15 | 15 | 23 | — | — |

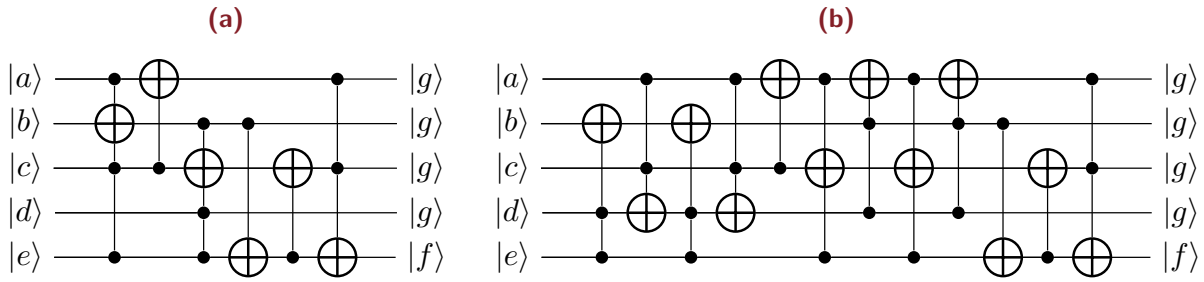



**Figure 9.5:** (a) Benchmark circuit 4gt10-v1_81. (b) The equivalent decomposed circuit.

We demonstrate the impact of topology on feasibility of nearest neighbor mapping for a given circuit. For this purpose, we used the circuit $4gt10-v1\_81$ shown in Fig. 9.5a. The circuit has a Toffoli gate with 3-control lines. This cannot be mapped using 1D-NN or cycle topology because a qubit can have at most two-neighbors in 1D or cycle topology. However, for other topologies, the mapping is feasible and the results using formulation $P_3$ are presented in Table 9.7. In order to make the nearest neighbor mapping feasible, the Toffoli gate with $n$-controls can be decomposed into a sequence of 2-control Toffoli gates [232, 233]. We used the RC-Viewer+ tool [234] to decompose the circuit as shown in Fig. 9.5b, followed by problem formulation $P_3$ to solve the nearest neighbor mapping problem for the same. As evident from the results, the decomposed circuit is now feasible to be mapped to 1D-NN and cycle topologies. For the other topologies, the mapping of the decomposed circuit has worser delay compared to the mapping of the original circuit, due the higher number of levels in the decomposed circuit.

For the first time, we report results for multiple topologies for various standard benchmark quantum circuits in Table 9.8. For each circuit, we consider the smallest topology graph with number of nodes greater than or equal to number of variables in the circuit. We have considered an arbitrary initial

placement of the qubits on the topology graph. As expected, topologies with greater number of edges have lower delay. For example, the delay obtained for *cycle* topology is less than that for $1D$ topology. Multiple benchmarks for the 3D-Grid and cyclic butterfly network (CBN) did not complete execution within the specified time limit, due to the relatively large size of the topology graphs.

Direct comparison of our method to obtain nearest-neighbor compliant circuits with existing works could not be performed for primarily three reasons. The existing works [235–238] focus on determining linear nearest neighbors (LNN), with the objective of reducing number of swap gates. Our proposed method is for obtaining the LNN circuits with minimal depth which is contrary to the goal of reducing swap gate count. Secondly, the initial placement of the qubit is assumed to be given as input to the problem, but other works consider this as part of the optimization. Finally, most of the existing works decompose the gates into two qubit gates [219, 235, 236, 239]. In our work, we used unmodified circuits from RevLib [231]. For reference of the readers, we provide a brief summary of the existing results in terms of number of swap gates against the solution of our proposed methodology using problem formulation $P_3$ with block size $b = 4$, for the decomposed circuits in Table 9.9. Due to non-availability of the depth of the transformed circuits, we *cannot* compare the performance of our method against the existing works.

## 9.5 Discussion

In this chapter, we addressed the problem of nearest-neighbor optimization for a given quantum circuit, an arbitrary topology graph and an initial configuration specifying the location of qubits in the topology graph. We formulated the problem using two ILP variants — one of the variant for obtaining the optimal solution and a simpler variant that can obtain a bounded solution. In addition, our problem formulation allows the optimization to be performed as a large set of small optimizations or a smaller set of larger optimization problems, by setting appropriate block sizes. We demonstrated the effectiveness of our approach by evaluating it on a set of benchmark circuits.

**Table 9.8:** NN-complaint circuit results for complete benchmark circuits.

| Benchmark | #Var | #Gates | #L | Tech. | 1D | | Cycle | | 2D-Mesh | | Torus | | 3D-Grid | | CBN | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | #S | D | #S | D | #S | D | #S | D | #S | D | #S | D |
| 3_17_14.real | 3 | 6 | 6 | $P_2$ | 3 | 7 | 0 | 6 | 10 | 7 | 0 | 6 | 7 | 8 | 0 | 6 |
| | | | | $P_3$ | 3 | 7 | 0 | 6 | 14 | 7 | 0 | 6 | — | — | 0 | 6 |
| 4gt11-v1_85.real | 5 | 4 | 3 | $P_2$ | 8 | 7 | 3 | 5 | 1 | 4 | 3 | 4 | 0 | 3 | 1 | 4 |
| | | | | $P_3$ | 7 | 7 | 4 | 5 | 4 | 4 | 9 | 4 | — | — | — | — |
| 4mod5-v1_25.real | 5 | 4 | 3 | $P_2$ | 3 | 5 | 2 | 4 | 3 | 5 | 2 | 4 | — | — | — | — |
| | | | | $P_3$ | 4 | 5 | 3 | 4 | 3 | 5 | 6 | 4 | — | — | — | — |
| alu-bdd_288.real | 7 | 9 | 8 | $P_2$ | 18 | 15 | — | — | — | — | — | — | — | — | — | — |
| | | | | $P_3$ | — | — | — | — | 16 | 10 | 15 | 9 | — | — | — | — |
| ex-1_166.real | 3 | 4 | 4 | $P_2$ | 1 | 4 | 0 | 3 | 1 | 4 | 0 | 3 | 1 | 4 | 0 | 3 |
| | | | | $P_3$ | 1 | 5 | 0 | 4 | 4 | 5 | 2 | 4 | 6 | 5 | 8 | 4 |
| ex1_226.real | 6 | 7 | 5 | $P_2$ | 8 | 9 | 7 | 8 | 4 | 7 | 2 | 6 | — | — | — | — |
| | | | | $P_3$ | 8 | 7 | 8 | 7 | 12 | 6 | 9 | 5 | — | — | — | — |
| fredkin_7.real | 3 | 1 | 1 | $P_2$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | | $P_3$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| graycode6_48.real | 6 | 5 | 5 | $P_2$ | 0 | 5 | 0 | 5 | 5 | 6 | 4 | 6 | — | — | 0 | 5 |
| | | | | $P_3$ | 2 | 5 | 0 | 5 | 7 | 5 | 8 | 5 | — | — | — | — |
| ham3_103.real | 3 | 4 | 4 | $P_2$ | 3 | 6 | 1 | 4 | 3 | 6 | 1 | 4 | 3 | 6 | 1 | 4 |
| | | | | $P_3$ | 3 | 6 | 1 | 4 | 8 | 6 | 2 | 4 | 11 | 6 | 21 | 4 |
| mod5mils_71.real | 5 | 5 | 5 | $P_2$ | 4 | 7 | 4 | 7 | 3 | 5 | 2 | 5 | — | — | — | — |
| | | | | $P_3$ | 6 | 7 | 4 | 6 | 5 | 5 | 6 | 5 | — | — | — | — |
| one-two-three-v3_101.real | 5 | 8 | 7 | $P_2$ | 10 | 13 | 7 | 11 | 10 | 11 | 6 | 9 | | | | |
| | | | | $P_3$ | 9 | 12 | 7 | 11 | — | — | 13 | 8 | — | — | — | — |
| peres_9.real | 3 | 2 | 2 | $P_2$ | 2 | 4 | 0 | 2 | 3 | 4 | 0 | 2 | 14 | 4 | 0 | 2 |
| | | | | $P_3$ | 2 | 4 | 0 | 2 | 9 | 4 | 2 | 2 | 2 | 4 | 0 | 2 |
| rd32_272.real | 5 | 6 | 5 | $P_2$ | 9 | 11 | 4 | 8 | 5 | 7 | 7 | 7 | | | | |
| | | | | $P_3$ | 9 | 11 | 6 | 8 | 12 | 7 | 8 | 6 | — | — | — | — |
| toffoli_double_4.real | 4 | 2 | 2 | $P_2$ | 3 | 4 | 1 | 3 | 4 | 4 | 2 | 3 | 14 | 3 | 1 | 3 |
| | | | | $P_3$ | 3 | 4 | 1 | 3 | 4 | 3 | 5 | 3 | 4 | 3 | 4 | 3 |

**Table 9.9:** Comparison with existing works on LNN.

| Benchmark | #Var | #Gates | $P_3(b=4)$ | N=4[235] | [236] | [237] |
|---|---|---|---|---|---|---|
| 3_7_13 | 3 | 14 | 7 | 6 | 6 | 4 |
| 4_49_17 | 7 | 32 | 15 | 15 | 20 | 12 |
| 4gt10-v1_81 | 5 | 36 | 33 | 22 | 30 | 20 |
| 4gt11_84 | 5 | 7 | 5 | 5 | 3 | 1 |
| 4gt13-v1_93 | 5 | 17 | 18 | 10 | 11 | 6 |
| 4gt5_75 | 5 | 22 | 25 | 15 | 17 | 12 |
| 4mod5-v1_23 | 5 | 24 | 22 | 13 | 16 | 9 |
| alu-v4_36 | 5 | 32 | 26 | 22 | 23 | 18 |
| hwb4_52 | 4 | 23 | 13 | 9 | 14 | 10 |
| ham7_104 | 7 | 87 | 140 | 83 | 84 | 68 |
| mod5adder_128 | 6 | 87 | 94 | 65 | 85 | 51 |

# 10 Conclusion And Future Work

In this thesis, we investigated new logic primitives and architectures to allow utilization of novel functionality offered by ReRAMs, which offer inherent storage and logic-in-memory capabilities. Motivated by the lack of end-of-end Electronic Design Automation (EDA) tools for emerging technologies, we investigated technology-specific constraints and developed algorithms to aid in seamless computing of arbitrary Boolean as well as multi-valued functions using memristive crossbar arrays.

In our quest to develop a programmable architecture that exploits the unique features of the mremristive crossbars, we introduced an efficient a general purpose computing platform — ReVAMP, that allows VLIW instructions to allow harnessing bit-level parallelism of the crossbar arrays (Chapter 3). We undertook a detailed case study for round function of a standard cryptographic hash algorithm known as SHA-3 or Keccak [86] using the ReVAMP architecture and compared against the state-of-the-art low power bit-serial CMOS only implementation.

From the perspective of design automation for logic realization using 1S1R devices, we addressed two variants of the technology mapping problem (Chapter 4). We proposed delay optimal technology mapping algorithm, along with heuristics to reduce device count. We introduced a novel data structure, State Dependency Graph (SDG) for addressing the problem of area-constrained technology mapping.

We formulated the problem as an Integer Linear Programming for obtaining optimal solution. We also developed scalable heuristics for solving larger instances of the problem. We proposed multi-phase framework for generation of instructions of ReVAMP architecture, starting from technology independent logic representation structure with delay as well as area constraints (Chapter 5). Also, we propose technology specific synthesis optimization techniques to improve the efficiency of the overall design automation flow (Chapter 6).

One of the outstanding features of multi-state $TaO_x$ memristive devices is the ability to store multi-bit information in a single device. This allows reduction in storage space compared to storage in binary format. However, we have investigated the characteristics of the device and found a more intriguing application — native Łukasiewicz logic implementation (Chapter 7). We demonstrated for the first time fuzzy logic control natively using the multi-state $TaO_x$ memristive devices. Furthermore, we augment an existing MVL synthesis tool with novel synthesis algorithms to support synthesis of MVL functions to Łukasiewicz logic primitives.

Quantum computing is another emerging technology that has garnered a lot of attention in recent years. Quantum computing involves computation on data by making direct use of quantum-mechanical phenomena, such as superposition and entanglement. In this thesis, we developed solutions for two challenging problems in the field of quantum computing. First, we developed heuristic algorithms to introduce the constraint of the number of qubits in the synthesis phase of automation flow for quantum circuits (Chapter 8). The second problem related to the insertion of so-called SWAP gates in-order to "move" non-adjacent qubits onto adjacent positions. The qubits can be arranged in various interaction topologies such as mesh or 1-D nearest neighbor, etc (Chapter 9). The challenge is to insert SWAP gates such that the additional delay due to the SWAP gates is minimal. We have successfully solved this challenge optimally by using an ILP formulation that permits arbitrary interaction topologies to produce nearest-neighbour complaint quantum circuits.

Our work on design automation challenges for emerging technologies have led not only to novel solutions, but to a host of interesting new problems as well. We summarily present some open challenges

and possibilities.

- **Proof of computational complexity of area-constrained technology mapping for LiM using ReRAM devices.** We have developed a polynomial time algorithm for delay-optimal technology mapping for LiM using ReRAM devices. Even though we have given a feasible solution for area-constrained technology mapping as an ILP formulation, we have not formally given reduction for the proof of the computational hardness of the problem. A feasible approach might be to try deriving a polynomial time reduction of the *one-shot pebble games* to the area-constrained technology mapping problem.

- **Design automation for sequential benchmarks.** The thesis addresses the design automation flow (including technology specific synthesis optimizations and technology mapping algorithms) for computing combinational benchmarks using crossbar arrays. However, memristive crossbar arrays are inherently sequential in nature, since each device is capable of acting as storage (register) as well as realizing some logic functionality. The problem of design automation for sequential benchmarks would be interesting to solve, especially to devise techniques that would improve mapping delay in presence of a mix of registers and logic in the input functionality specification.

- **Integrated logic-in-memory accelerators.** Similar to the proposed ReVAMP architecture, logic-in-memory accelerator could be designed for integration into an existing system. The implementation of such an accelerator block would need extensive studies on the design of the programming interface, the architecture itself and most importantly data-mapping schemes. Also, the limits of the in-memory computing acceleration block from the perspective of power dissipation, reliability and endurance are interesting directions to investigate.

- **Crossbar-constrained technology mapping for MVL.** In Chapter 7, we addressed in detail the synthesis challenges for MVL synthesis flow, but we have not considered crossbar-constraints in the algorithms. Similar to the binary counterparts discussed in Chapter 5 and Chapter 6, crossbar-aware algorithms should be investigated for leverage the inherent parallelism offered by

the arrays to improvement in delay and area of mapping MVL functions.

- **Quantum cost aware quantum circuit synthesis.** In Chapter 8, we mapped each LUT in the LUT-network to an STG. The cost of computing and uncomputing the STGs mapped to Clifford+T circuits are different. Therefore, it would be advantageous to free the qubits corresponding to the STG targets, that have a lower quantum cost. The current heuristic is not aware of this cost, while determining the qubit to free. Developing an algorithm that is aware of the costs of STGs would lead to lower costs of the resulting quantum circuits.

In conclusion, we have solved interesting and significant EDA challenges for emerging technologies, that would enable adoption of these technologies in their full potential.

# Appendices

# A ReVAMP Simulation Setup

In this appendix, we provide detailed information about the ReVAMP simulation setup. We explain the setup for a $2 \times 4$ crossbar and realize the multiplexer function described in subsection 3.2.1. We setup device-accurate simulation using Cadence AMS designer tool.

## A.1 Single Device Simulation Setup

Before describing the simulation setup of the entire system, we show the setup for a single device, with the associated peripheral circuitry, as shown in Figure A.1.

### A.1.1 1S1R device

`1S1R device` is the 1S1R ReRAM device, fitted to the parameters presented in section 2.1. `p` and `be` ports of the device represent the top and bottom electrode respectively. The port `Po` is a virtual port, that reports the instantaneous power of the device.

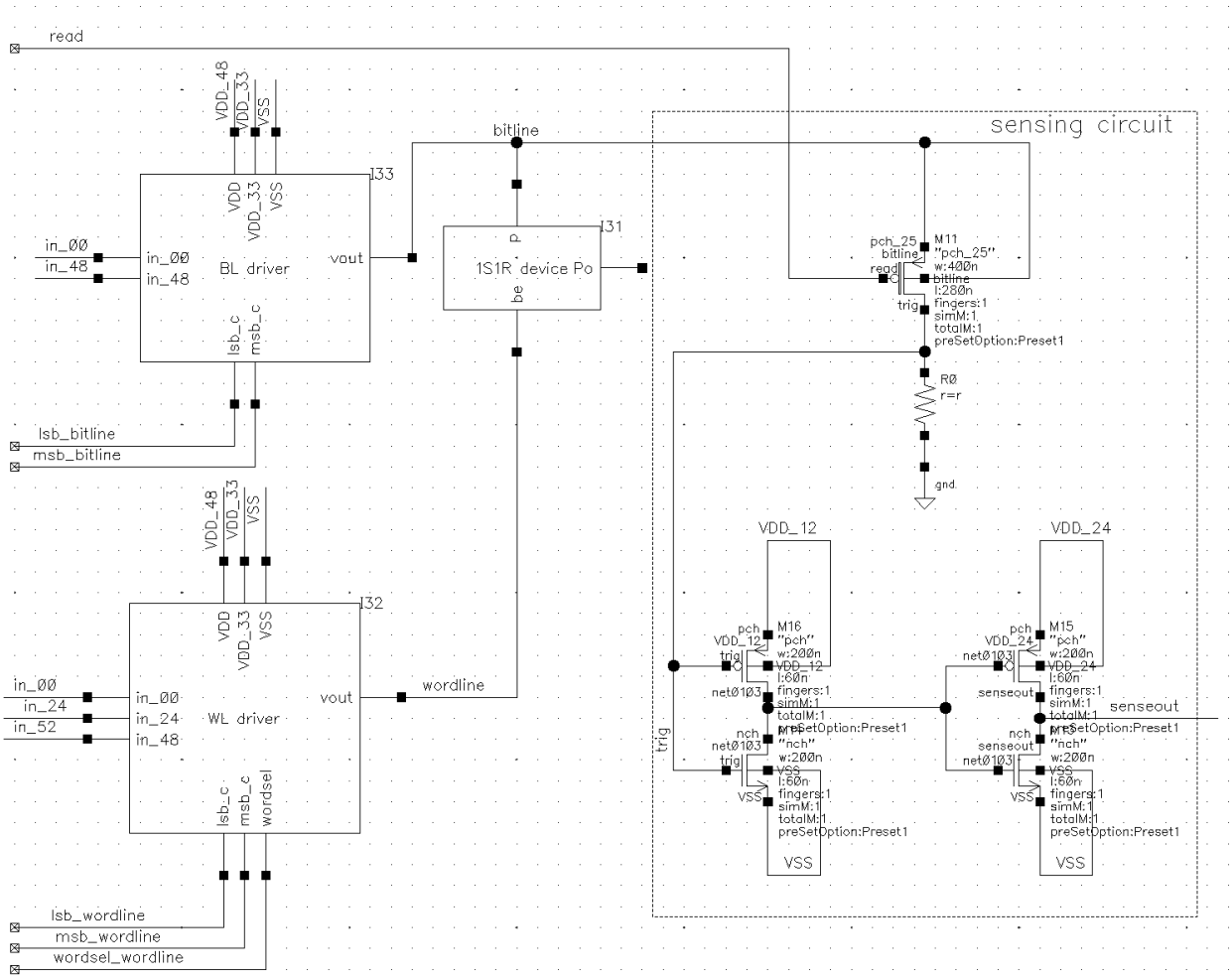



**Figure A.1:** Schematic of a single ReRAM cell with the peripheral voltage drivers and sensing circuit.

### A.1.2 Wordline and bitline drivers

The driver modules use IO3.3V transistors since a 4.8V voltage has to be driven through them. In order to driver a positive voltage (2.4V or 4.8V), a PMOS is used with body tied to VDD=4.8V while a NMOS with the body tied to VSS=0V is used for driving 0V. This is because PMOS drives strong 1's while NMOS drives strong 0's. In the driver circuits, the inputs are obtained from either a constant voltage source or an external voltage source. The control signals of each driver, generated by the digital controller block, drive the gates of transistors present in the driver. Buffers are used in the drivers to

increase drive strength of the control signals.

Figure A.2 presents the detailed schematic of the wordline driver. The output `vout` of the `WL driver` depends on the values of the three input digital ports — `wordsel`, `msb_c` and `lsb_c` as shown below.

| wordsel | msb_c | lsb_c | vout |
|---|---|---|---|
| 1 | 0 | 0 | 4.8V (‘1’) |
| 1 | 1 | 1 | 0V (‘0’) |
| 1 | 1 | 0 | Isolated (No operation) |
| 0 | 1 | 0 | 2.4V (Read) |

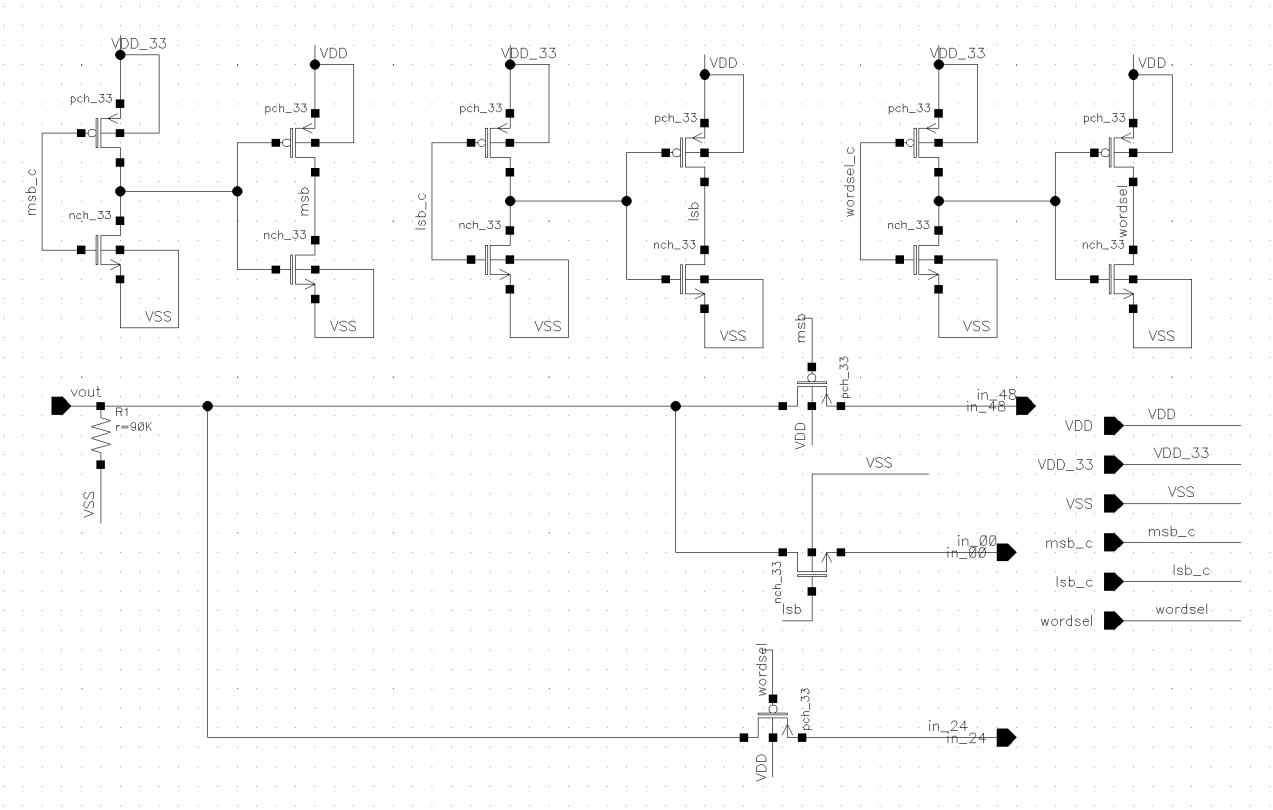



**Figure A.2:** Schematic of wordline driver.

Figure A.3 presents the detailed schematic of the bitline driver. Similar to the wordline driver, the

output `vout` of the `BL driver` is a function of the input digital signals `msb_c` and `lsb_c`.

| msb_c | lsb_c | vout |
|---|---|---|
| 0 | 0 | 4.8V (‘1’) |
| 1 | 1 | 0V (‘0’) |
| 1 | 0 | Isolated/Read |

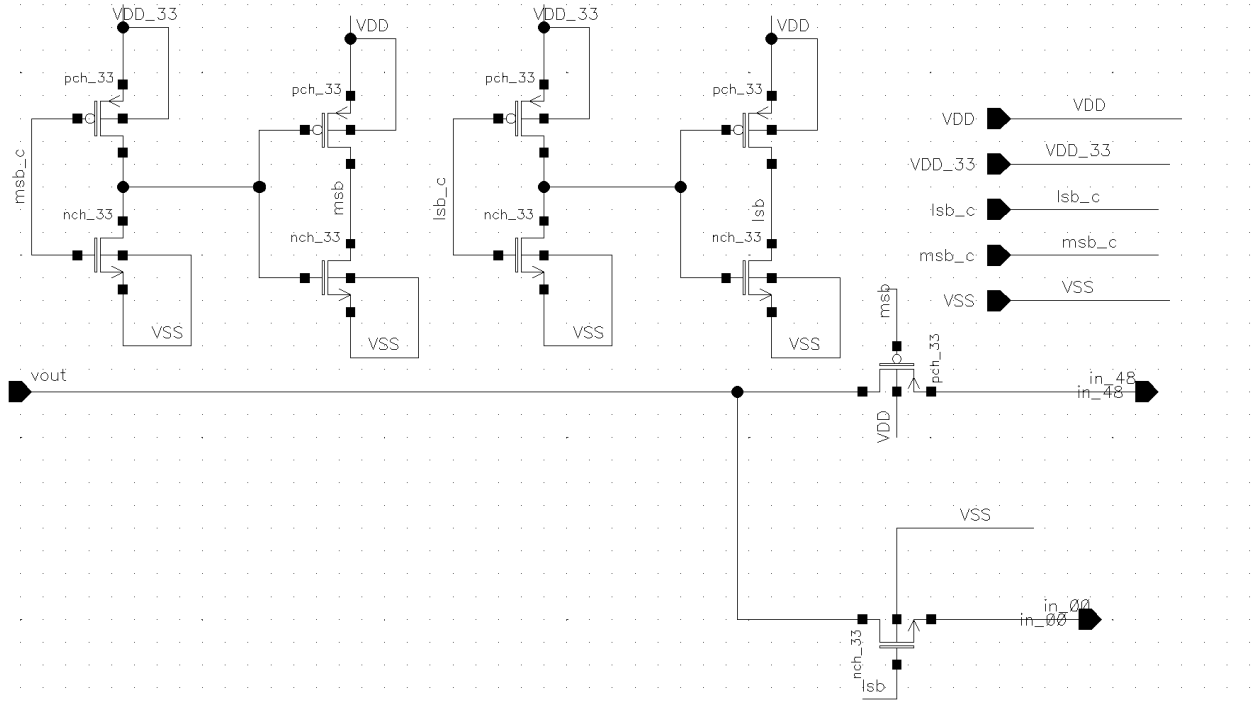



**Figure A.3:** Schematic of bitline driver.

### A.1.3 Sensing Circuit

The `sensing circuit` takes a digital active-low `read` signal and the current flowing through the `bitline` as input to generate an output `senseout` signal.

| `read` | Device state | `senseout` |
|---|---|---|
| 0 | LRS | High |
| 1 | HRS | Low |
| 1 | LRS | Isolated |
| 1 | HRS | Isolated |

A voltage divider circuit with enable is used to sense the internal resistive state of an ReRAM cell. The voltage divider employed in the setup consists of one variable resistor (ReRAM cell) and one fixed resistor (500KΩ-1MΩ) and a transistor to enable the path on a Read instruction. The sensing circuit is isolated during execution of Apply instruction. For a read voltage of 2.4V on wordline with the bitline isolated, the drop across the fixed resistor varies depending on the state of the ReRAM cell (LRS/HRS). When the device state is in LRS, the voltage drop across the resistor is ≈ 520mV, while in HRS, it is ≈ 100mV. This voltage is used for driving the next stage. The voltage probed for the next stage is across the fixed resistor. The voltage drop of 520mV is sufficient for switching the stage 1 invertor in the buffer and the following one as well. The value of the resistance for the fixed resistor must be chosen carefully, for correct read out of the device state.

### A.1.4 Simulation Waveforms

We verify the functionality of the single device setup, shown in Figure A.1. We use 50ns clock period for simulation and show the waveforms of simulation in Figure A.4.

- In the first cycle, the device is RESET to HRS state by applying '0' (0V) to the wordline with '1' (4.8V) at the bitline. In this cycle, read operation is not being performed, hence `read` signal is HIGH (2.4V). The `wordline current` signal in the figure shows the current flowing through the device during the RESET operation.
- In the next cycle, the device state is read out by applying 2.4V to the wordline with the bitline

isolated. The read signal is now set to LOW (0V). The drop across the fixed resistor in the sensing circuit shown in the `trig` signal, is 137mV. This voltage is translated to a LOW (≈0V) signal by the sensing circuit, as shown by the `sensout` signal in the figure, which verifies that the state of the device was HRS.

- In the third cycle, the device is RESET to LRS state by applying '1' (4.8V) to the wordline with '0' (0V) at the bitline.

- Similar to the second cycle, we read out the device state in this cycle. In this cycle, the `trig` signal is 592mV which is sensed as a HIGH ($approx$2.4V) signal by the sensing circuit, correctly verifying the state of the device to be LRS.

## A.2 Full System Simulation Setup

In this section, we explain the simulation setup for the complete ReVAMP architecture. Figure A.5 shows the schematic for the digital components of the architecture while Figure A.6 presents a slice of the ReRAM crossbar array acting as DCM. We explain the functionality of the individual components below.

### A.2.1 Testbench

The `Testbench` is used to load the instructions {*Read*, *Apply*} in the `Instruction Memory`. During actual fabrication of the architecture, this has to be replaced by either an SPI or UART interface to load the instructions. Once the instructions are loaded to Instruction Memory, the testbench is responsible for enabling the PC and to let the execution begin. The testbench is also used to specify the contents of the PIR, which acts an input source.

### A.2.2 Program Counter

The testbench is responsible for driving the `Program Counter` (PC), in increments of 1 on every clock cycle. On resetting, the PC is initialized to 0. We

### A.2.3 Instruction Memory

The `Instruction Memory` (IM) stores the *Read* and *Apply* instructions. In the simulation setup, a single port SRAM memory model $TS1N65LPA8192X32M16$ is used as IM. IM has 8192 memory locations with each location is of 32-bits wide. The instruction width is $max(IL_{Read}, IL_{Apply})$. However in practice, the instruction word width is rounded to power of 2 greater than or equal to required instruction width. For example, if instruction width is 94-bit, the IM width is 128-bit. As the SRAM memory model used for the simulation setup has width of 32-bit, we use 4 instances of the SRAM in parallel to form 128-bit word, with the PC being used to address each SRAM. Each instruction lies in first 94 bits (MSB) while the remaining bits are 0-filled. The `MUX` determines the value used to address the Instruction Memory — PC provided address is used during regular execution of instructions or the address supplied by the Testbench is used during loading of instructions.

### A.2.4 Crossbar Controller

The `Controller` is parameterized on number of wordlines $S_D$ and number of bitlines $w_D$ available for the DCM. The controller receives the instruction from the IM and *decodes* the instruction that are used to generate control signals for the peripherals driving the crossbar array. The controller has been implemented in Verilog and synthesized using RTL compiler. $TSMC\_CLN65LP$ PDK is used for standard cells. The controller operates at $2.4V$ as $V_{DD}$ and $0V$ as $V_{SS}$.

### A.2.5 ReRAM Crossbar Array

The ReRAM crossbar array acts as DCM. Each device is a 1S1R device, realizing the state update function. The devices are organized in crossbar organization by using wires to connect them. Figure A.6 shows the schematic for the device organization. Corresponding to each row, there is a single wordline driver that is connected to the wordlines of all devices present in that row. For each column, there is a single bitline driver connected to the bitlines of each device present in the column. Also, the devices in a column share a common sensing circuit.

### A.2.6 Simulation Waveforms

For the simulation of the entire architecture, we consider the example introduced in subsection 3.2.1. The instruction length is 19 bits, and there are 11 instructions in total. The crossbar size used is $2 \times 4$, i.e., two wordlines and four bitlines. We consider $a = 0$, $b = 1$ and $s = 1$ as inputs for computing the function $M$. The simulation waveforms are shown in Figure A.8 and Figure A.7.

- At time instant $V1$, we SET all the device 3 and 2 in wordline 1 to HRS state by applying 0V to wordline and 4.8V to the bitlines. To do so, the controller set the `wordsel`, `msb_c` and `lsb_c` to 1 that are inputs to the wordline driver 1. For the bitline driver of bitlines 3 and 2, the controller sets both `msb_c` and `lsb_c` to 0. For the bitlines 1 and 0, the bitlines are isolated by setting the corresponding `msb_c` and `lsb_c` to 1 and 0 respectively.

- At time instant $V2$, the read instruction for wordline 1 is executed. The controller sets the `read_n` signal to LOW. The wordline driver inputs corresponding to `wordsel`, `msb_c` and `lsb_c` are set to 0, 1 and 0 respectively. All the bitlines are isolated by setting the corresponding `msb_c` and `lsb_c` to 1 and 0 respectively. The sensed outputs for the device 3 and 2 in wordline 1 are 0, since we SET the devices at the time instant V1. Since we did not SET the devices 1 and 0, the sensed outputs corresponding to these devices are unpredictable.

- At time instant $V3$, the fourth instruction is executed where the wordline input is '1' (as $s = 1$) and bitline input is '0'. The wordline driver inputs corresponding to `wordsel`, `msb_c` and `lsb_c` are set to 0, 1 and 0 respectively. For the bitline driver of bitlines 3 and 2, the controller sets both `msb_c` and `lsb_c` to 1. Rest of the bitlines are isolated.

- At time instant $V4$, the second Read instruction is executed. In this cycle, the sensed out values of the devices 3 and 2 in wordline 0 are 0 and 1 respectively, which is as expected.

- At time instant $V5$, the last Read instruction is executed. In this cycle, the sensed out values of the devices 3 and 2 in wordline 1 are 1 and 0 respectively, which is as expected.

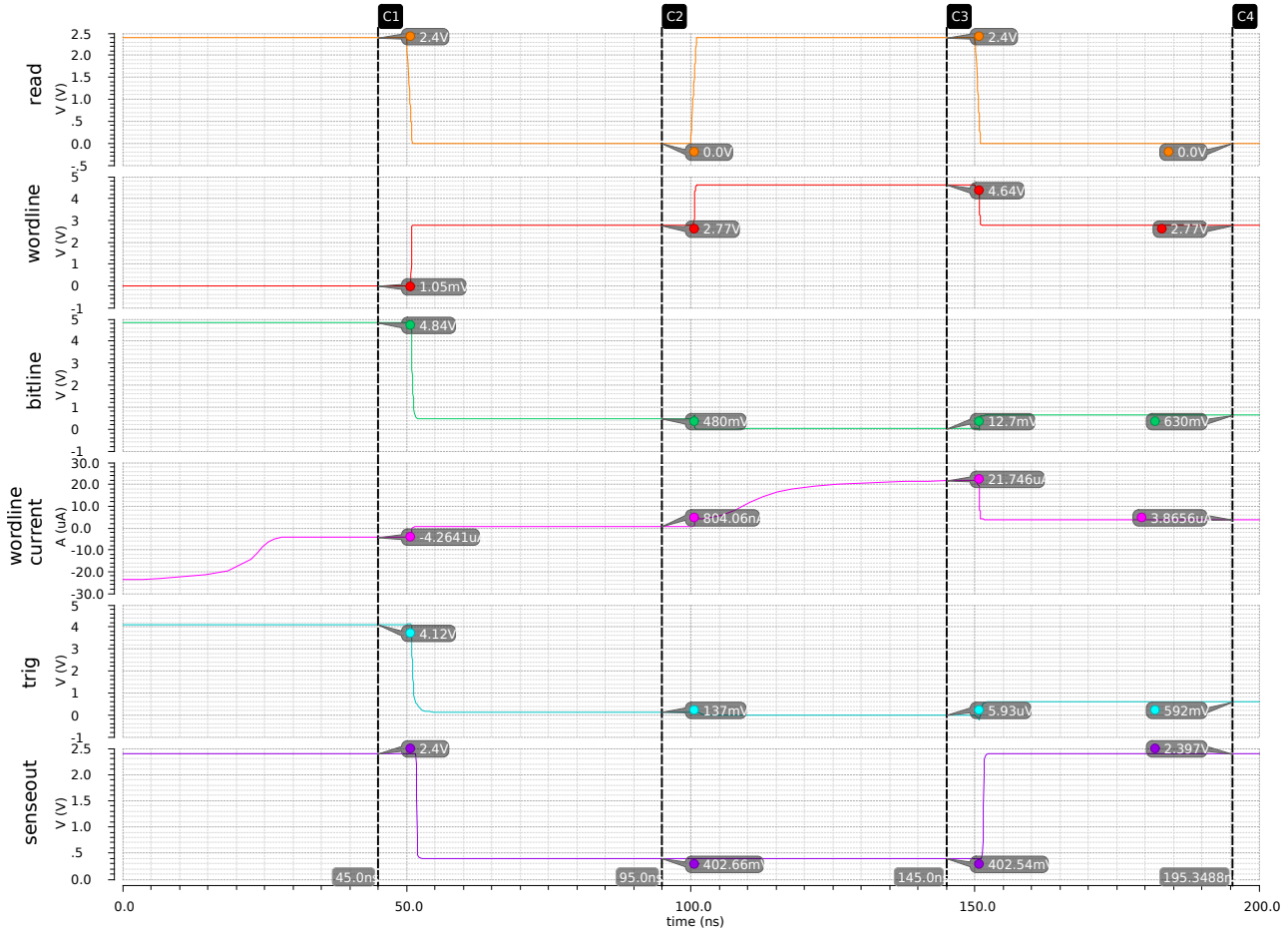



**Figure A.4:** Simulation waveforms of a single ReRAM cell. `wordline` and `bitline` show the applied voltage to the wordline and bitline of the device. `read` is an input active-low digital signal to indicate read operation. `wordline current` shows the current flowing through the device. `trig` is the voltage drop across the fixed resistor in the sensing circuit while `senseout` is the output of the sensing circuit.

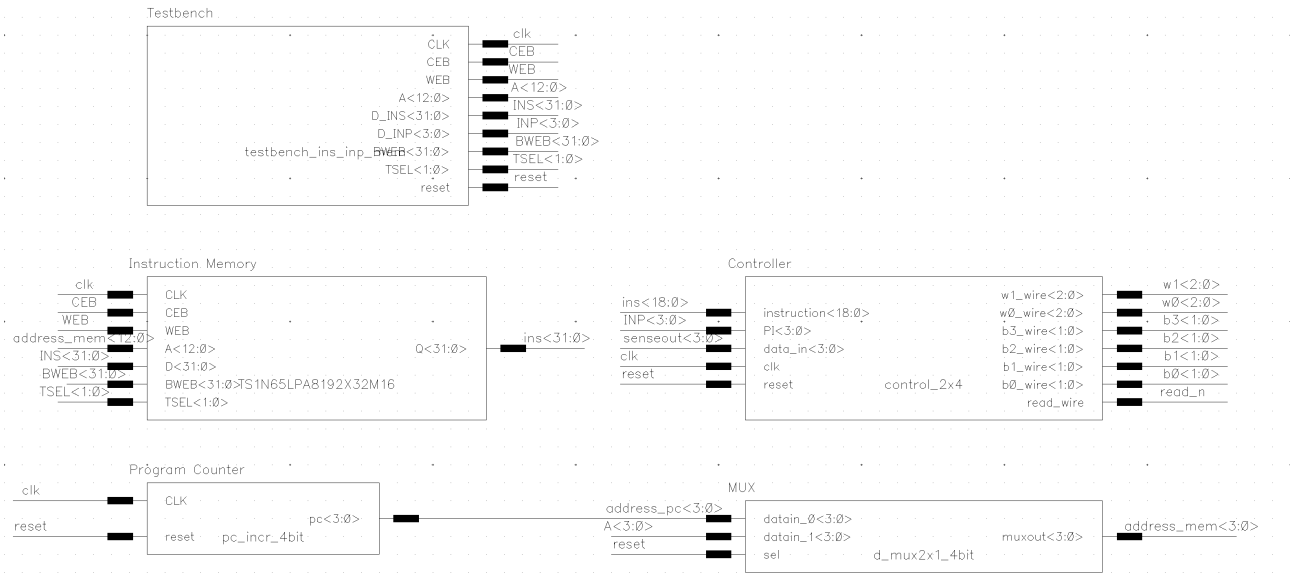



**Figure A.5:** Schema of the digital components of the ReVAMP architecture.

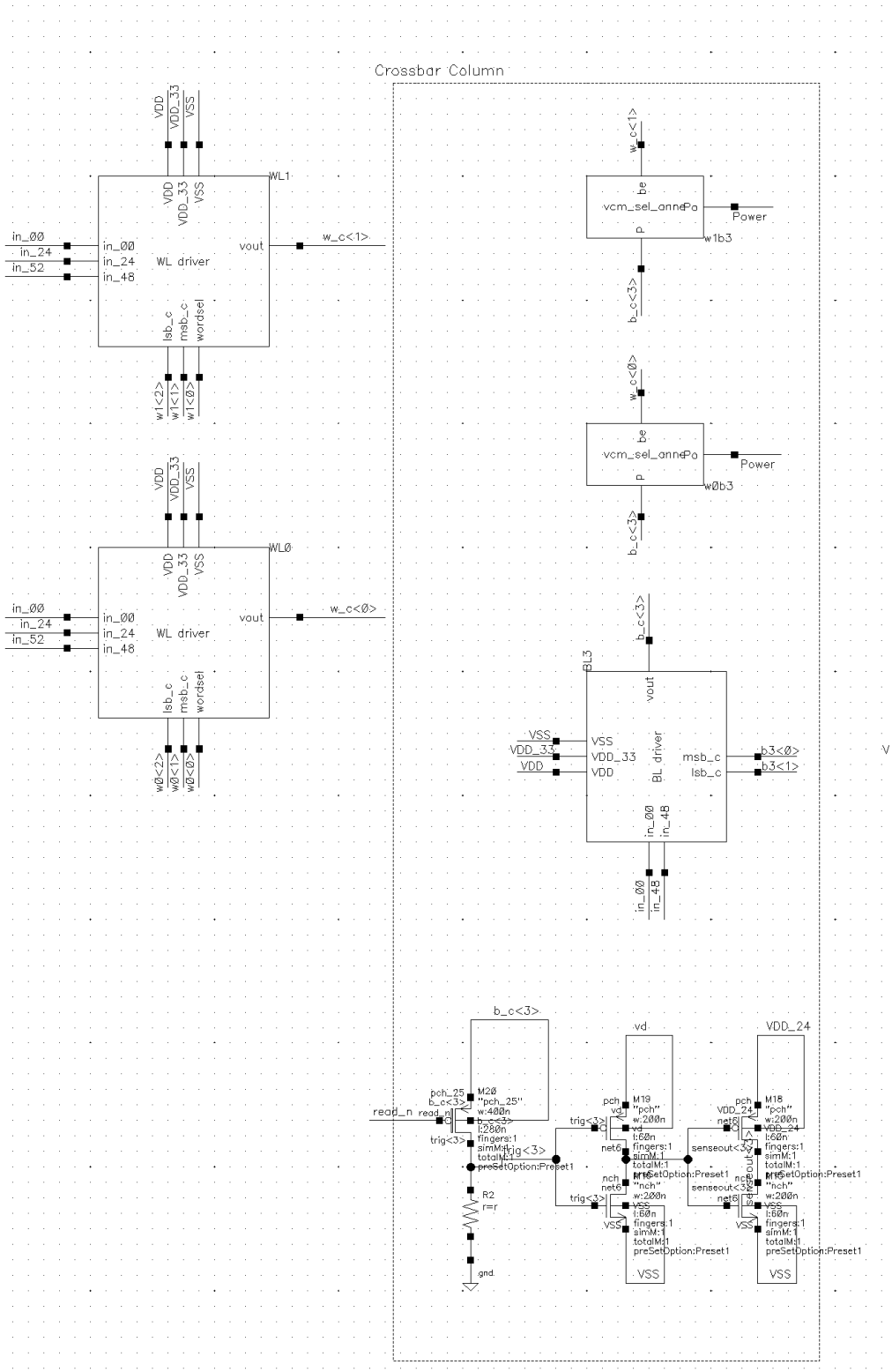



**Figure A.6:** Schema of the ReRAM crossbar and the peripherals of the crossbar. The schema shows a single column of the crossbar, with two devices.

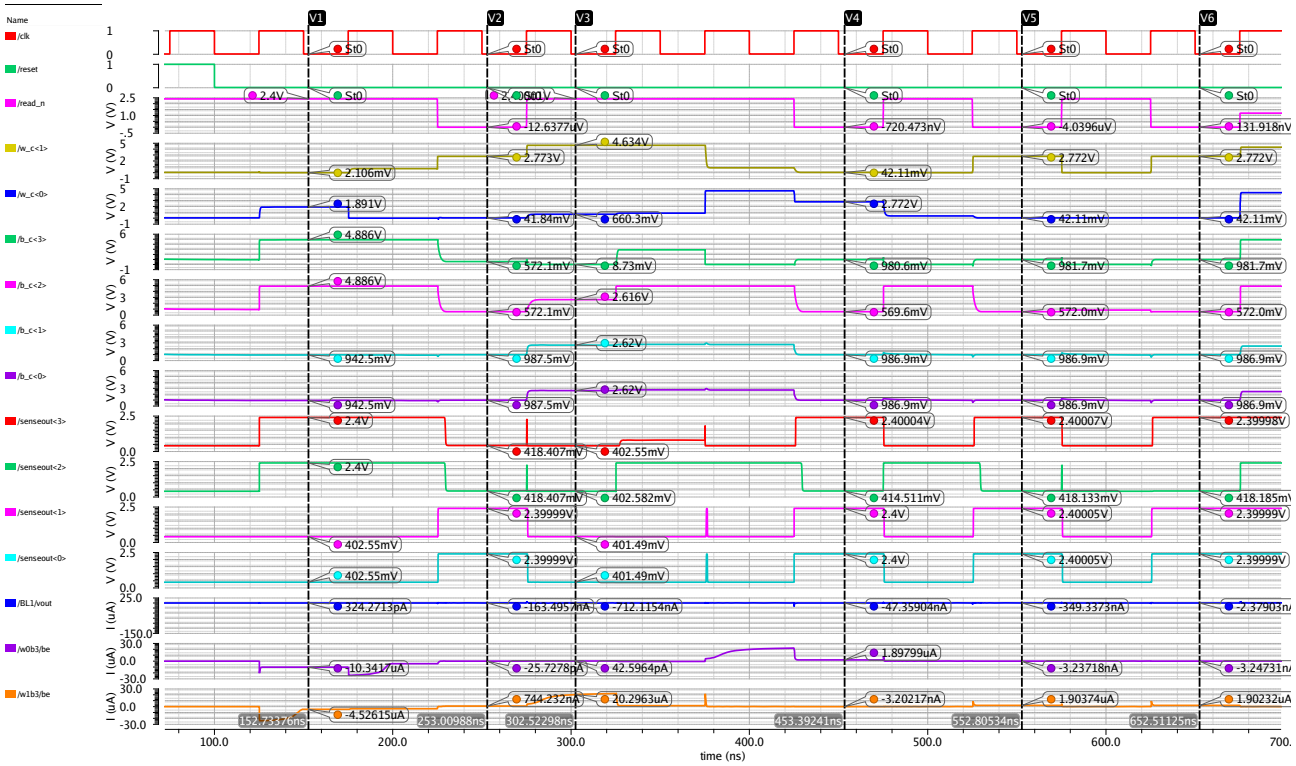



**Figure A.7:** Part 1: System Simulation waveform for multiplexer function realization with $a = 0$, $b = 1$ and $s = 1$ as inputs. `clk`: Clock signal; `reset`: Reset signal; `read_n`: Active low digital signal to enable sensing circuits; `w_c<i>`: Input voltage to wordline i; `b_c<j>`: Input voltage to bitline j; `senseout<j>`: Output of sensing circuit for bitline j; `w0b3/be,w1b3/be`: Current through the devices at wordline 0 bitline 3 and wordline 1 bitline 3 respectively.

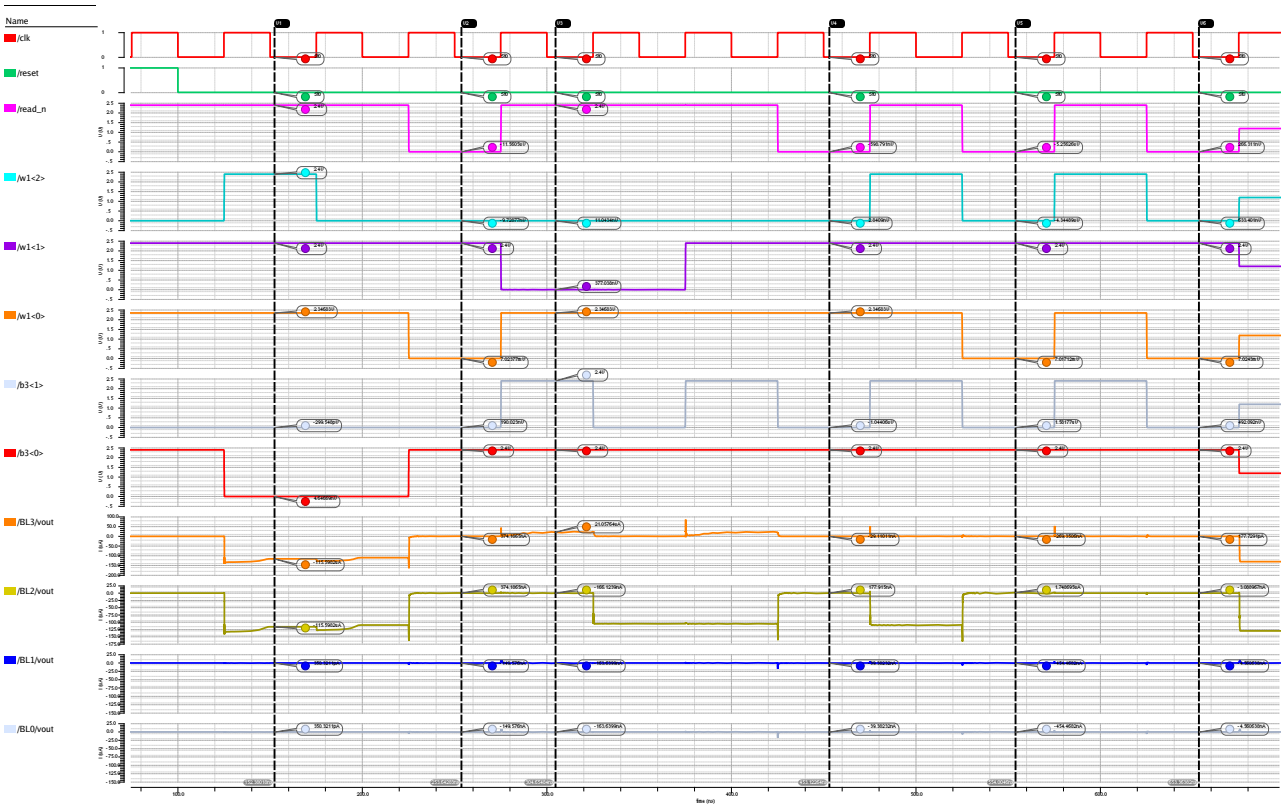



**Figure A.8:** Part 2: System Simulation waveform for multiplexer function realization with $a = 0$, $b = 1$ and $s = 1$ as inputs. `clk`: Clock signal; `reset`: Reset signal; `read_n`: Active low digital signal to enable sensing circuits; `w1<0>,w1<1>,w1<2>`: Controller input to wordline driver 1; `w3<0>,b3<1>`: Controller input to bitline driver 3; `BLj/vout`: Current at bitline driver j output.

# B Multi-state ReRAM Device Fabrication And Characteristics Study

### B.0.1 Device Fabrication

Cross-point based $Ta_2O_5$ ReRAM was fabricated on thermally grown $SiO_2$ samples. In our design, each device shares a common bottom electrode (BE). The BE was patterned in $30nm$-thick platinum ($Pt$) layers, grown by the sputtering process. After patterning the BE, switching layer of $7nm$-thick $TaO_x$, $13nm$-thick tungsten ($W$), and $25nm$-thick platinum ($Pt$) were sequentially deposited by the sputtering process. The $TaO_x$ layer was grown with reactive sputtering process with 76.6% Argon and 23.3% Oxygen at partial pressure of $2.3 \times 10^{-2} mbar$. The $W$ ohmic electrode, and the $Pt$ were grown with DC sputtering method. For the top electrode (TE) patterning, photo-lithography and reactive ion etching steps were performed. These steps lead to the $Pt/W/TaO_x/Pt$ memristive device stack. Fig. 1 shows the scanning electron microscopy (SEM) of $1 \times 3$ crossbar array with $2\mu m \times 2\mu m$ size cell with cross-sectional tunneling electron microscopy (TEM) image and its corresponding schematic diagram.

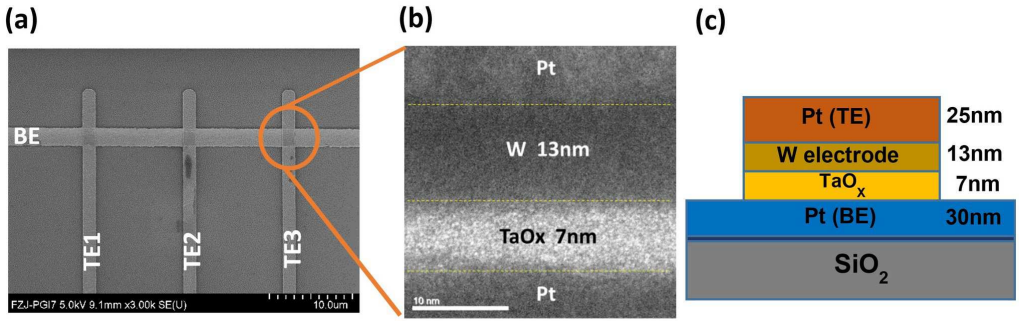



**Figure B.1:** Resistive switching device structure.s (a) Scanning electron microscopy image of $1 \times 3$ array, with $2\mu m \times 2\mu m$ device. (b) Tunneling electron microscopy image of a single device cross-section with $7nm$-thick $TaO_x$ switching layer and $13nm$-thick tungsten ohmic electrode. (c) Schematic diagram of the single resistive device.

More experimental details can be found in reference [20].

### B.0.2 Measurement Set-up

The pristine state of the memristive devices was highly resistive ($G\Omega$) and therefore an electroforming process was required. This process was carried out by applying a positive DC voltage on the TE for a given current compliance, while keeping the BE grounded. This turned the devices into low resistance state (LRS). Now, the memristive devices were sequentially switched to high resistance state (HRS) by the 'reset process' and low resistance state (LRS) by the 'set process'. In this experiment, the 'set' process is performed with DC voltage while the reset operations were performed with 200 $ns$ pulse width and 120$\mu s$ long pulse width at $0.1V$ was used to read the respective resistance states. More measurement details can be traced in reference [20].

### B.0.3 Device Properties

In this work, $2\mu m \times 2\mu m$ $Pt/W/TaO_x/Pt$ cross-point bipolar memristive devices arranged in word structure have been fabricated. Figure B.1 shows the scanning electron microscope and transmission electron microscopy image of the devices used in this experiment. The ReRAM device stack of

$25nm\ Pt/\ 13nm\ W/\ 7nm\ TaO_x\ /30nm\ Pt$ is depicted in Figire B.1 (b). Figure B.1 (c) shows the schematic of a single device along with the details of the stacked layers. Figure 7.1 (a) shows the typical $I-V$ characteristics of the ReRAM device with set current compliance of 1.0 $mA$, along with the electroforming curve. After the electroforming process, the device was toggled to high resistance state by applying the reset voltage. The maximum applied voltage $|V_{stop}|$ during RESET process defines the final resistive state of the device. This feature is also used in pulse mode operation, and can thus be used in memory and logic operations for controlling the multi-level states. To enable highly reproducible RESET operation, we have always applied a DC SET operation before each pulsed RESET operation ($200ns$). Note that a nanosecond pulsed SET operations are also feasible, but have not been applied in this work. Figure 7.1 (b) shows a very tight resistance distribution of low resistance state (LRS) and six multi-level resistive states. This confirms the excellent switching properties of these devices. For the multi-level programming, we have split the total applied amplitude across the bottom electrode and the top electrode. A fixed positive amplitude ($+0.7V$) is assigned at the bottom electrode while a varying negative amplitude ($-0.7V$ to $-1.5V$) is applied to the top electrode. Under this configuration, the total applied amplitude across the cell varies from $-1.4V$ to $-2.2V$ for the given pulse width of $200ns$. For each $V_{stop}$ voltage, the cell is toggled to a different high resistance state (HRS). The final resistance state has been read by the means of a $120\mu s$ pulse with amplitude $V_{READ} = 0.1V$. Figure 7.1 (c) shows the mean value of each resistive state from R0 to R4.

# Disseminations From This Thesis

**Book Chapter**

1. *In-memory data compression using ReRAMs.*, **Debjyoti Bhattacharjee** and Anupam Chattopadhyay, in Emerging Technology and Architecture for Big-data Analytics. Springer International Publishing, 2017.

**Journals**

1. *Multi-valued and Fuzzy Logic Realization using $TaO_x$ Memristive Devices*, **Debjyoti Bhattacharjee**, Wonjoo Kim, Anupam Chattopadhyay, Rainer Waser, and Vikas Rana, in Scientific reports 8, no. 1 (2018): 8.
2. *Quantum Circuits for Toom-Cook Multiplication*, Srijit Dutta, **Debjyoti Bhattacharjee**, and Anupam Chattopadhyay, in Phys. Rev. A, 2018.
3. *Kogge-Stone Adder Realization using 1S1R Resistive Switching Crossbar Arrays*, **Debjyoti Bhattacharjee**, Anne Siemon, Eike Linn, Stephan Menzel and Anupam Chattopadhyay. In ACM Journal on Emerging Technologies in Computing Systems (JETC), 2018.
4. *Efficient Complementary Resistive Switch-based Crossbar Array Booth Multiplier*, **Debjyoti Bhattacharjee**, Anne Siemon, Eike Linn, Anupam Chattopadhyay, in Elsevier Microelectronics Journal, 2017.

**Conferences**

1. *MAMI: Majority and Multi-Input Logic on Memristive Crossbar Array*, **Debjyoti Bhattacharjee**, Arko Dutt and Anupam Chattopadhyay, in 2018 IEEE Asia Pacific Conference on Circuits and Systems (IEEE APCCAS 2018).
2. *ReRAM-based In-Memory Computation of Galois Field Arithmetic*, Swagata Mandal, **Debjyoti Bhattacharjee**, Yaswanth Tavva and Anupam Chattopadhyay, in 26th IFIP/IEEE International Conference on Very Large Scale Integration, 2018.

3. *Single output multi-valued logic function synthesis using Łukaszewicz logic operators*, Anmol Prakash, **Debjyoti Bhattacharjee**, and Anupam Chattopadhyay. In IEEE International Symposium on Multiple-Valued Logic (ISMVL), 2018.

4. *Technology-Aware Logic Synthesis for ReRAM based In-Memory Computing*, **Debjyoti Bhattacharjee**, Luca Amaru and Anupam Chattopadhyay, in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.

5. *Floating Point Multiplication Mapping on ReRAM based In-Memory Computing Architecture*, Tarun Vatwani, Arko Dutt, **Debjyoti Bhattacharjee**, and Anupam Chattopadhyay, in VLSI Design Conference, 2018.

6. *ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting*, **Debjyoti Bhattacharjee**, Rajeshwari Devadoss, and Anupam Chattopadhyay in Design, Automation & Test, in Europe Conference & Exhibition (DATE), 2017.

7. *Area-constrained Technology Mapping for In-Memory Computing using ReRAM devices*, **Debjyoti Bhattacharjee**, Arvind Easwaran and Anupam Chattopadhyay, in Asia and South Pacific Design Automation (ASP-DAC), 2017.

8. *SHA-3 Implementation Using ReRAM based In-Memory Computing Architecture*, **Debjyoti Bhattacharjee**, Vikramkumar Pudi and Anupam Chattopadhyay, in 2017 18th International Symposium on Quality Electronic Design (ISQED).

9. *Efficient Binary Basic Linear Algebra Operations on ReRAM Crossbar Arrays*, **Debjyoti Bhattacharjee**, and Anupam Chattopadhyay, in VLSI Design Conference, 2017.

10. *Delay-Optimal Technology Mapping for In-Memory Computing using ReRAM Devices*, **Debjyoti Bhattacharjee** and Anupam Chattopadhyay, in International Conference on Computer Aided Design (ICCAD), 2016.

11. *Enabling In-Memory Computation of Binary BLAS using ReRAM Crossbar Arrays*, **Debjyoti Bhattacharjee**, Farhad Merchant and Anupam Chattopadhyay, in VLSI-SOC, 2016.

12. *Efficient Implementation of Multiplexer and Priority Multiplexer using 1S1R ReRAM Crossbar Arrays*, **Debjyoti Bhattacharjee**, Anne Siemon, Eike Linn, Stephan Menzel, Anupam Chattopadhyay, in MWSCAS, 2016.

**Pre-prints**

1. *Crossbar-Constrained Technology Mapping for ReRAM based In-Memory Computing*, **Debjyoti Bhattacharjee**, Yaswanth Tavva, Arvind Easwaran, and Anupam Chattopadhyay. arXiv preprint arXiv:1809.08195 (2018).

2. *Depth-optimal quantum circuit placement for arbitrary topologies.* Debjyoti Bhattacharjee and Anupam Chattopadhyay. arXiv preprint arXiv:1703.08540 (2017).

# Bibliography


[1] Cadence Design Systems. *GDSII Stream Format Manual, Documentation No. B97E060, Feb. 1987* (see p. 1).

[2] Carver Mead and Lynn Conway. *Introduction to VLSI systems*. Vol. 1080. Addison-Wesley Reading, MA, 1980 (see p. 1).

[3] John A Darringer et al. "LSS: A system for production logic synthesis". In: *IBM Journal of research and Development* 28.5 (1984), pp. 537–545 (see p. 1).

[4] R.K. Brayton et al. "MIS: A Multiple - Level Logic Optimization System". USenglish. In: *IEEE Trans. on Comp.* 6.6 (1987), pp. 1062–1081 (see p. 1).

[5] Patrick McGeer et al. "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions". In: *Proceedings of the 30th international Design Automation Conference*. ACM. 1993, pp. 618–624 (see pp. 1, 48).

[6] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/~alanmi/abc/. Online; accessed 28 September 2018. 2018 (see pp. 1, 47, 84, 97).

[7] Alberto Sangiovanni-Vincentelli. "The tides of EDA". In: *IEEE Design & Test of Computers* 20.6 (2003), pp. 59–75 (see p. 1).

[8] Yuan Taur. "CMOS design near the limit of scaling". In: *IBM Journal of Research and Development* 46.2.3 (2002), pp. 213–222 (see p. 1).

[9] Kerry Bernstein et al. "Device and architecture outlook for beyond CMOS switches". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2169–2184 (see pp. 1, 2).

[10] William Shockley. "The Theory of p-n Junctions in Semiconductors and p-n Junction Transistors". In: *Bell System Technical Journal* 28.3 (1949), pp. 435–489 (see p. 1).

[11] Umberto Celano. "Filamentary-Based Resistive Switching". In: *Metrology and Physical Mechanisms in New Generation Ionic Devices*. Cham: Springer International Publishing, 2016, pp. 11–45. DOI: 10.1007/978-3-319-39531-9_2 (see pp. 2, 11).

[12] Hyung Dong Lee et al. "Integration of 4F2 selector-less crossbar array 2Mb ReRAM based on transition metal oxides for high density memory applications". In: *VLSI Technology (VLSIT), 2012 Symposium on*. IEEE. 2012, pp. 151–152 (see p. 2).

[13] Myoung-Jae Lee et al. “A fast, high-endurance and scalable non-volatile memory device made from asymmetric $Ta_2O_{5-x}/TaO_{2-x}$ bilayer structures”. In: *Nature materials* 10.8 (2011), pp. 625–630 (see p. 2).

[14] Z Wei et al. “Retention model for high-density ReRAM”. In: *Memory Workshop (IMW), 2012 4th IEEE International*. IEEE. 2012, pp. 1–4 (see p. 2).

[15] W Kim et al. “3-Bit Multilevel Switching by Deep Reset Phenomenon in $Pt/W/TaO_X/Pt$-ReRAM Devices”. In: *IEEE Electron Device Letters* 37.5 (2016), pp. 564–567 (see p. 2).

[16] Antonio C Torrezan et al. “Sub-nanosecond switching of a tantalum oxide memristor”. In: *Nanotechnology* 22.48 (2011), p. 485203 (see p. 2).

[17] E Linn et al. “Beyond von Neumann-logic operations in passive crossbar arrays alongside memory operations”. In: *Nanotechnology* 23.30 (2012), p. 305205 (see pp. 2, 16, 127).

[18] Eike Linn. “Memristive nano-crossbar arrays enabling novel computing paradigms”. In: *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 2596–2599 (see p. 2).

[19] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa and W. Lu. “A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications”. In: *Nano Letters* 12.1 (2011), pp. 389–395. DOI: 10.1021/nl203687n (see p. 2).

[20] Wonjoo Kim et al. “Multistate Memristive Tantalum Oxide Devices for Ternary Arithmetic”. In: *Scientific Reports* 6 (2016) (see pp. 2, 212).

[21] Hartmut Häffner, Christian F Roos, and Rainer Blatt. “Quantum computing with trapped ions”. In: *Physics reports* 469.4 (2008), pp. 155–203 (see p. 2).

[22] William M Kaminsky, Seth Lloyd, and Terry P Orlando. “Scalable superconducting architecture for adiabatic quantum computation”. In: *arXiv preprint quant-ph/0403090* (2004) (see p. 2).

[23] Scott Aaronson and Andris Ambainis. “The need for structure in quantum speedups”. In: *arXiv preprint arXiv:0911.0996* (2009) (see p. 3).

[24] Peter W Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM review* 41.2 (1999), pp. 303–332 (see pp. 3, 156, 172).

[25] NIST. *Quantum Algorithm Zoo*. https://math.nist.gov/quantum/zoo/. Online; accessed 28 September 2018 (see p. 3).

[26] Damian S Steiger, Thomas Häner, and Matthias Troyer. “ProjectQ: an open source software framework for quantum computing”. In: *Quantum* 2 (2018), p. 49 (see p. 3).

[27] OpenQASM, IBM Q Experience (QX). *Quantum Information Science Kit (Qiskit)*. https://github.com/Qiskit/qiskit-terra. Online; accessed 28 September 2018 (see p. 3).

[28] Mathias Soeken, Thomas Häner, and Martin Roetteler. “Programming quantum computers using design automation”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. 2018, pp. 137–146 (see p. 3).

[29] L. Chua. “Memristor-The missing circuit element”. In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519. DOI: 10.1109/TCT.1971.1083337 (see p. 9).

[30] Leon O Chua and Sung Mo Kang. "Memristive devices and systems". In: *Proceedings of the IEEE* 64.2 (1976), pp. 209–223 (see p. 9).

[31] Dmitri B Strukov et al. "The missing memristor found". In: *nature* 453.7191 (2008), pp. 80–83 (see pp. 9, 11, 12).

[32] Leon Chua. "Resistance switching memories are memristors". In: *Applied Physics A* 102.4 (2011), pp. 765–783 (see p. 10).

[33] Rainer Waser. "Electrochemical and thermochemical memories". In: *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*. IEEE. 2008, pp. 1–4 (see p. 10).

[34] Rainer Waser et al. "Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges". In: *Advanced Materials* 21 (2009), pp. 2632–2663 (see pp. 10, 118).

[35] Hiroyuki Akinaga and Hisashi Shima. "Resistive random access memory (ReRAM) based on metal oxides". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2237–2251 (see p. 11).

[36] H-S Philip Wong et al. "Metal–oxide RRAM". In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970 (see p. 11).

[37] Daniele Ielmini. "Resistive switching memories based on metal oxides: mechanisms, reliability and scaling". In: *Semiconductor Science and Technology* 31.6 (2016), p. 063002 (see p. 11).

[38] Ting-Chang Chang et al. "Resistance random access memory". In: *Materials Today* 19.5 (2016), pp. 254–264 (see p. 11).

[39] B Govoreanu et al. "10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation". In: *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE. 2011, pp. 31–6 (see pp. 11, 126).

[40] Jury Sandrini et al. "Co-design of ReRAM passive crossbar arrays integrated in 180 nm CMOS technology". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.3 (2016), pp. 339–351 (see p. 11).

[41] Alexander Flocke and Tobias G Noll. "Fundamental analysis of resistive nano-crossbars for the use in hybrid Nano/CMOS-memory". In: *Solid State Circuits Conference, 2007. ESSCIRC 2007. 33rd European*. IEEE. 2007, pp. 328–331 (see p. 11).

[42] Eike Linn et al. "Complementary resistive switches for passive nanocrossbar memories". In: *Nature materials* 9.5 (2010), pp. 403–406 (see pp. 11, 12).

[43] K Tsunoda et al. "Low power and high speed switching of Ti-doped NiO ReRAM under the unipolar voltage source of less than 3 V". In: *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*. IEEE. 2007, pp. 767–770 (see p. 12).

[44] Shyh-Shyuan Sheu et al. "A 5ns fast write multi-level non-volatile 1 K bits RRAM memory with advance write scheme". In: *VLSI Circuits, 2009 Symposium on*. IEEE. 2009, pp. 82–83 (see p. 12).

[45] Cong Xu et al. "Overcoming the challenges of crossbar resistive memory architectures". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 476–488 (see p. 12).

[46] Manqing Mao et al. “Optimizing latency, energy, and reliability of 1T1R ReRAM through cross-layer techniques”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.3 (2016), pp. 352–363 (see p. 12).

[47] Simone Cortese. “Selector devices/architectures for ReRAM crossbar arrays”. PhD thesis. University of Southampton, University Library, 2017 (see p. 12).

[48] R Stanley Williams, Matthew D Pickett, and John Paul Strachan. “Physics-based memristor models”. In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*. IEEE. 2013, pp. 217–220 (see p. 12).

[49] Eike Linn et al. “Applicability of well-established memristive models for simulations of resistive switching devices”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.8 (2014), pp. 2402–2410 (see p. 12).

[50] Shahar Kvatinsky et al. “VTEAM: A general model for voltage-controlled memristors”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 62.8 (2015), pp. 786–790 (see p. 12).

[51] Anne Siemon et al. “Simulation of $TaO_x$-based complementary resistive switches by a physics-based memristive model”. In: *ISCAS*. IEEE. 2014, pp. 1420–1423 (see p. 12).

[52] Seonghyun Kim, Wootae Lee, and Hyunsang Hwang. “Selector devices for cross-point ReRAM”. In: *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*. 2012 (see pp. 12, 15).

[53] Wootae Lee et al. “High current density and nonlinearity combination of selection device based on $TaO_x/TiO_2/TaO_x$ structure for one selector–one resistor arrays”. In: *ACS nano* 6.9 (2012), pp. 8166–8172 (see p. 12).

[54] Wm A Wulf and Sally A McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24 (see p. 15).

[55] Ioannis Vourkas and Georgios Ch Sirakoulis. “A novel design and modeling paradigm for memristor-based crossbar circuits”. In: *IEEE Transactions on Nanotechnology* 11.6 (2012), pp. 1151–1159 (see p. 15).

[56] Dmitri B Strukov and Konstantin K Likharev. “CMOL FPGA: a reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices”. In: *Nanotechnology* 16.6 (2005), p. 888 (see p. 16).

[57] Wen-Pin Lin et al. “A nonvolatile look-up table using ReRAM for reconfigurable logic”. In: *Solid-State Circuits Conference (A-SSCC), 2014 IEEE Asian*. IEEE. 2014, pp. 133–136 (see p. 16).

[58] Mircea R Stan et al. “Molecular electronics: From devices and interconnect to circuits and architecture”. In: *Proceedings of the IEEE* 91.11 (2003), pp. 1940–1957 (see p. 16).

[59] Somnath Paul and Swarup Bhunia. “A scalable memory-based reconfigurable computing framework for nanoscale crossbar”. In: *IEEE Transactions on Nanotechnology* 11.3 (2012), pp. 451–462 (see p. 16).

[60] Shahar Kvatinsky et al. “MRL-memristor ratioed logic”. In: *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*. IEEE. 2012, pp. 1–6 (see p. 16).

[61] G Snider. “Computing with hysteretic resistor crossbars”. In: *Applied Physics A* 80.6 (2005), pp. 1165–1172 (see pp. 16, 22).

[62] Julien Borghetti et al. “‘Memristive’ switches enable ‘stateful’ logic operations via material implication”. In: *Nature* 464.7290 (2010), pp. 873–876 (see pp. 16, 127).

[63] Shahar Kvatinsky et al. “MAGICâĂŤMemristor-aided logic”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 61.11 (2014), pp. 895–899 (see pp. 16, 22).

[64] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger and R. Waser. “Beyond von Neumann-logic operations in passive crossbar arrays alongside memory operations”. In: *Nanotechnology* 23.30 (2012). DOI: 10.1088/0957-4484/23/30/305205 (see p. 17).

[65] Pinaki Mazumder, Sung-Mo Kang, and Rainer Waser. “Memristors: devices, models, and applications”. In: *Proceedings of the IEEE* 100.6 (2012), pp. 1911–1919 (see p. 21).

[66] Raqibul Hasan, Tarek M Taha, and Chris Yakopcic. “On-chip training of memristor crossbar based multi-layer neural networks”. In: *Microelectronics Journal* 66 (2017), pp. 31–40 (see p. 21).

[67] Mahyar Shahsavari, Pierre Falez, and Pierre Boulet. “Combining a volatile and nonvolatile memristor in artificial synapse to improve learning in Spiking Neural Networks”. In: *Nanoscale Architectures (NANOARCH), 2016 IEEE/ACM International Symposium on*. IEEE. 2016, pp. 67–72 (see p. 21).

[68] Sparsh Mittal. “A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks”. In: *Machine Learning and Knowledge Extraction* 1.1 (2018), pp. 75–114 (see p. 21).

[69] D. B. Strukov, D.R. Stewart, J. Borghetti, X. Li, M. Pickett, G. M. Ribeiro, W. Robinett, G. Snider, J. P. Strachan, W. Wu, Q. Xia, J. J. Yang and R. S. Williams. “Hybrid CMOS/memristor circuits”. In: *ISCAS*. 2010, pp. 1967–1970. DOI: 10.1109/ISCAS.2010.5537020 (see p. 21).

[70] Bernabe Linares-Barranco et al. “On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex”. In: *Frontiers in neuroscience* 5 (2011), p. 26 (see p. 21).

[71] M. P. Sah, H. Kim and L. O. Chua. “Brains are made of Memristors”. In: *IEEE Circuits and Systems Magazine* 14.1 (2014) (see p. 21).

[72] Christopher D Krieger, David J Mountain, and Mark McLean. “Relative Efficiency of Memristive and Digital Neuromorphic Crossbars”. In: *Proceedings of the International Conference on Neuromorphic Systems*. ACM. 2018, p. 3 (see p. 21).

[73] Leibin Ni et al. “An Energy-efficient Digital ReRAM-crossbar based CNN with Bitwise Parallelism”. In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* (2017) (see p. 21).

[74] Jintao Yu et al. “Memristive devices for computation-in-memory”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE. 2018, pp. 1646–1651 (see p. 21).

[75] Ameer Haj-Ali et al. “IMAGING-In-Memory AlGorithms for Image processiNG”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 99 (2018), pp. 1–14 (see p. 21).

[76] Mathias Soeken et al. “A PLiM computer for the internet of things”. In: *Computer* 50.6 (2017), pp. 35–40 (see p. 22).

[77] Said Hamdioui et al. “Memristor for computing: Myth or reality?” In: *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2017, pp. 722–731 (see p. 22).

[78] Lei Xie et al. “Fast boolean logic mapped on memristor crossbar”. In: *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE. 2015, pp. 335–342 (see pp. 22, 80).

[79] Shahar Kvatinsky et al. “Memristor-based material implication (IMPLY) logic: Design principles and methodologies”. In: *IEEE TVLSI* 22.10 (2014), pp. 2054–2066 (see pp. 22, 127).

[80] Rotem Ben Hur et al. “SIMPLE MAGIC: synthesis and in-memory mapping of logic execution for memristor-aided logic”. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press. 2017, pp. 225–232 (see pp. 22, 80).

[81] Phrangboklang L Thangkhiew and Kamalika Datta. “Scalable in-memory mapping of Boolean functions in memristive crossbar array using simulated annealing”. In: *Journal of Systems Architecture* 89 (2018), pp. 49–59 (see pp. 22, 80).

[82] Pierre-Emmanuel Gaillardon et al. “The Programmable Logic-in-Memory (PLiM) computer”. In: *DATE*. 2016, pp. 427–432 (see pp. 22, 79, 102).

[83] M. Soeken et al. “An MIG-based Compiler for Programmable Logic-in-Memory Architectures”. In: *DAC*. 2016 (see pp. 22, 49).

[84] Guido Bertoni et al. “Cryptographic sponge functions. Submission to NIST (Round 3)”. In: 2011 (see p. 29).

[85] Guido Bertoni et al. “The Keccak SHA-3 submission. Submission to NIST (Round 3)”. In: 2011 (see p. 30).

[86] “National Institute of Standards and Technology (NIST). Cryptographic Hash Algorithm Competition Website”. In: (see pp. 30, 191).

[87] “Emerging Research Devices (ERD) report, International Technology Roadmap for Semiconductors (ITRS), 2013”. In: () (see pp. 39, 118).

[88] Abdulkadir Akin et al. “Efficient hardware implementations of high throughput SHA-3 candidates keccak, luffa and blue midnight wish for single-and multi-message hashing”. In: *Proceedings of the 3rd International Conference on Security of Information and Networks*. ACM. 2010, pp. 168–177 (see pp. 40, 41).

[89] J Strömbergson. “Implementation of the Keccak hash function in FPGA devices”. In: *Techical report, InformAsic AB* (2008) (see p. 40).

[90] Stefan Tillich et al. “High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Gröstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein.” In: *IACR Cryptology ePrint Archive* 2009 (2009), p. 510 (see pp. 40, 41).

[91] Peter Pessl and Michael Hutter. “Pushing the limits of SHA-3 hardware implementations to fit on RFID”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2013, pp. 126–141 (see pp. 40, 41).

[92] Kurt Keutzer. "DAGON: technology binding and local optimization by DAG matching". In: *Proceedings of the 24th ACM/IEEE Design Automation Conference*. ACM. 1987, pp. 341–347 (see p. 43).

[93] Robert Brayton and Alan Mishchenko. "ABC: An academic industrial-strength verification tool". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40 (see p. 43).

[94] Christopher Umans, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. "Complexity of two-level logic minimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.7 (2006), pp. 1230–1246 (see p. 43).

[95] Yasuhiko Takenaga and Shuzo Yajima. "Hardness of identifying the minimum ordered binary decision diagram". In: *Discrete applied mathematics* 107.1-3 (2000), pp. 191–201 (see p. 43).

[96] Saeideh Shirinzadeh et al. "Fast Logic Synthesis for RRAM-based In-Memory Computing using Majority-Inverter Graphs". In: *DATE*. 2016 (see pp. 44, 61, 73, 79).

[97] Richard L Rudell and Alberto Sangiovanni-Vincentelli. "Multiple-valued minimization for PLA optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.5 (1987), pp. 727–750 (see p. 45).

[98] Alan Mishchenko and Marek Perkowski. "Fast heuristic minimization of exclusive-sums-of-products". In: (2001) (see pp. 45, 81).

[99] Chang-Yeong Lee. "Representation of Switching Circuits by Binary-Decision Programs". In: *Bell system Technical journal* 38.4 (1959), pp. 985–999 (see p. 46).

[100] Robert K Brayton et al. "MIS: A multiple-level logic optimization system". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.6 (1987), pp. 1062–1081 (see p. 46).

[101] Ellen M Sentovich et al. "SIS: A system for sequential circuit synthesis". In: (1992) (see pp. 46, 48).

[102] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691 (see p. 46).

[103] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "Majority-Inverter Graph: A New Paradigm for Logic Optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.5 (2016), pp. 806–819 (see pp. 47, 107–109).

[104] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994 (see p. 48).

[105] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40 (see p. 48).

[106] *Synopsys Design Compiler*. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. 2018 (see p. 48).

[107] *Xilinx Design Suite*. https://www.xilinx.com/products/design-tools/vivado.html. 2018 (see p. 48).

[108] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition". In: *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 47 (see pp. 48, 110).

[109] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "Majority-inverter graph: A new paradigm for logic optimization". In: *IEEE TCAD* 35.5 (2016), pp. 806–819 (see pp. 48, 154).

[110] Luca Amarù et al. "Majority logic synthesis". In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 79 (see p. 48).

[111] Eero Lehtonen and Mika Laiho. "Stateful implication logic with memristors". In: *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures*. 2009, pp. 33–36 (see p. 49).

[112] E. Lehtonen, J. H. Poikonen and M. Laiho. "Two memristors suffice to compute all Boolean functions". In: *Electronics Letters* 46.3 (2010), pp. 239–240. DOI: 10.1049/el.2010.3407 (see p. 49).

[113] Anika Raghuvanshi and Marek Perkowski. "Logic Synthesis and a Generalized Notation for Memristor-Realized Material Implication Gates". In: *ICCAD* (2014), pp. 470–477 (see p. 49).

[114] Anupam Chattopadhyay and Zoltan Endre Rakosi. "Combinational logic synthesis for material implication". In: *IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011, Kowloon, Hong Kong, China*. 2011, pp. 200–203 (see p. 49).

[115] Yuji Kukimoto, Robert K Brayton, and Prashant Sawkar. "Delay-optimal technology mapping by DAG covering". In: *Proceedings of the 35th annual Design Automation Conference*. ACM. 1998, pp. 348–351 (see p. 49).

[116] Kurt Keutzer and D Richards. "Computational complexity of logic synthesis and optimization". In: *Proceedings of International Workshop on Logic Synthesis*. 1989 (see p. 49).

[117] J. Cong and Y. Ding. "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs". In: *1992 IEEE/ACM International Conference on Computer-Aided Design*. 1992, pp. 48–53. DOI: 10.1109/ICCAD.1992.279398 (see p. 50).

[118] Amir H Farrahi and Majid Sarrafzadeh. "Complexity of the lookup-table minimization problem for FPGA technology mapping". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.11 (1994), pp. 1319–1332 (see p. 50).

[119] Ilan Levin and Ron Y Pinter. "Realizing expression graphs using table-lookup FPGAs". In: *Design Automation Conference, 1993, with EURO-VHDL'93. Proceedings EURO-DAC'93., European*. IEEE. 1993, pp. 306–311 (see p. 50).

[120] Thomas H.. Cormen et al. *Introduction to algorithms*. Vol. 6. MIT press Cambridge, 2001 (see p. 56).

[121] *Gurobi optimization*. http://www.gurobi.com/index (see pp. 74, 184).

[122] Tsutomu Sasao, Masahiro Fujita, et al. *Representations of discrete functions*. Vol. 242. Springer, 1996 (see p. 87).

[123] Tomasz Kozlowski, Erik L Dagless, and Jonathan M Saul. "An enhanced algorithm for the minimization of exclusive-or sum-of-products for incompletely specified functions". In: *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*. IEEE. 1995, pp. 244–249 (see p. 87).

[124] Rolf Drechsler and Bernd Becker. "Sympathy: fast exact minimization of fixed polarity Reed-Muller expressions for symmetric functions". In: *European Design and Test Conference, 1995. ED&TC 1995, Proceedings*. IEEE. 1995, pp. 91–97 (see p. 87).

[125] A Zakrevskij. "Minimum polynomial implementation of systems of incompletely specified Boolean functions". In: *Proc. Reed-Muller*. Vol. 95. 1995, pp. 250–256 (see p. 87).

[126] Rolf Drechsler. "Pseudo-Kronecker expressions for symmetric functions". In: *IEEE Transactions on Computers* 48.9 (1999), pp. 987–990 (see p. 87).

[127] M. G. A. Martins et al. "Majority-based logic synthesis for nanometric technologies". In: *14th IEEE International Conference on Nanotechnology*. 2014, pp. 256–261. DOI: 10.1109/NANO.2014.6968043 (see p. 105).

[128] K. Kong, Y. Shang, and R. Lu. "An Optimized Majority Logic Synthesis Methodology for Quantum-Dot Cellular Automata". In: *IEEE Transactions on Nanotechnology* 9.2 (2010), pp. 170–183. DOI: 10.1109/TNANO.2009.2028609 (see p. 105).

[129] S. Shirinzadeh et al. "Endurance management for resistive Logic-In-Memory computing architectures". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 1092–1097. DOI: 10.23919/DATE.2017.7927152 (see p. 105).

[130] Hans-J Bandelt and Jarmila Hedlíková. "Median algebras". In: *Discrete mathematics* 45.1 (1983), pp. 1–30 (see p. 107).

[131] Donald E Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Pearson Education India, 2011 (see p. 107).

[132] SA Kiss and Bull Amer Math. "A ternary operation in distributive lattices". In: *Selected Papers on Algebra and Topology by Garrett Birkhoff* (1987), p. 107 (see p. 108).

[133] Rajeswari Devadoss, Kolin Paul, and M Balakrishnan. "MajSynth: An n-input Majority Algebra based Logic Synthesis Tool for Quantum-dot Cellular Automata". In: *Proceedings of IWLS*. 2015 (see p. 110).

[134] VI Varshavskii. "Compatible majority decomposition". In: *Cybernetics and Systems Analysis* 4.3 (1968), pp. 75–76 (see p. 110).

[135] Claude E. Shannon. "A symbolic analysis of relay and switching circuits". In: *Electrical Engineering* 57.12 (1938), pp. 713–723 (see p. 117).

[136] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847 (see p. 117).

[137] Aristotle. *Complete Works of Aristotle, Volume 1: The Revised Oxford Translation*. Ed. by Jonathan Barnes. Princeton University Press, 2014 (see p. 117).

[138] Jan Łukasiewicz. *Selected Works*. Amsterdam: North-Holland Pub. Co., 1970 (see p. 117).

[139] Emil L. Post. "Introduction to a general theory of elementary propositions". In: *American journal of mathematics* 43.3 (1921), pp. 163–185 (see p. 118).

[140] M Wajsberg. “Axiomatization of the 3–valued sentential calculus”. In: *CR Soc. Sci. Lettr. Varsovie* 24 (1931), pp. 126–148 (see p. 118).

[141] C. C. Chang. “A New Proof of the Completeness of the Łukasiewicz Axioms”. In: *Transactions of the American Mathematical Society* 93.1 (1959), pp. 74–80 (see p. 118).

[142] Vilém Novák. “A formal theory of intermediate quantifiers”. In: *Fuzzy Sets and Systems* 159.10 (2008), pp. 1229–1246 (see p. 118).

[143] J. Paul Roth. “Diagnosis of automata failures: A calculus and a method”. In: *IBM journal of Research and Development* 10.4 (1966), pp. 278–291 (see p. 118).

[144] E. Ozer, R. Sendag, and D. Gregg. “Multiple-valued logic buses for reducing bus energy in low-power systems”. In: *IEEE Proceedings-Computers and Digital Techniques* 153.4 (2006), pp. 270–282 (see p. 118).

[145] Elena Dubrova. “Multiple-valued logic in VLSI: challenges and opportunities”. In: *Proceedings of NORCHIP*. Vol. 99. 1999, pp. 340–350 (see p. 118).

[146] Brian Hayes. “Computing Science: Third Base”. In: *American scientist* 89.6 (2001), pp. 490–494 (see p. 118).

[147] Stanley L. Hurst. “Multiple-valued logic its status and its future”. In: *IEEE Transactions on Computers* 33.12 (1984), pp. 1160–1179 (see p. 118).

[148] Mohammad Hossein Moaiyeri et al. “Design and analysis of carbon nanotube FET based quaternary full adders”. In: *Frontiers of Information Technology & Electronic Engineering* 17.10 (2016), pp. 1056–1066 (see p. 118).

[149] Takeshi Yamakawa and Tsutomu Miki. “The current mode fuzzy logic integrated circuits fabricated by the standard CMOS process”. In: *IEEE Transactions on Computers* 35.2 (1986), pp. 161–167 (see p. 118).

[150] Takeshi Yamakawa. “High-speed fuzzy controller hardware system: The mega-FIPS machine”. In: *Information Sciences* 45.2 (1988), pp. 113–128 (see p. 118).

[151] Robin Giles. “A resolution logic for fuzzy reasoning”. In: *Proceedings of IEEE 17th International Symposium on Multiple-Valued Logic*. 1985, pp. 60–67 (see p. 118).

[152] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. “Memristive devices for computing”. In: *Nature nanotechnology* 8.1 (2013), pp. 13–24 (see p. 118).

[153] Lotfi A. Zadeh. “Fuzzy sets”. In: *Information and control* 8.3 (1965), pp. 338–353 (see p. 119).

[154] Ebrahim H Mamdani. “Application of fuzzy algorithms for control of simple dynamic plant”. In: *Proceedings of the Institution of Electrical Engineers*. Vol. 121. 12. IET. 1974, pp. 1585–1588 (see pp. 119, 139).

[155] James Warren, Gleb Beliakov, and Berend Van Der Zwaag. “Fuzzy logic in clinical practice decision support systems”. In: *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*. IEEE. 2000, 10–pp (see p. 119).

[156] Soumitra Dutta. “Fuzzy logic applications: Technological and strategic issues”. In: *IEEE Transactions on Engineering Management* 40.3 (1993), pp. 237–254 (see p. 119).

[157] Ronald R Yager. "Expert systems using fuzzy logic". In: *An introduction to fuzzy logic applications in intelligent systems*. Springer, 1992, pp. 27–44 (see p. 119).

[158] Vivek G Moudgal et al. "Fuzzy learning control for a flexible-link robot". In: *IEEE Transactions on Fuzzy Systems* 3.2 (1995), pp. 199–210 (see p. 119).

[159] Lotfi A. Zadeh. "Soft computing and fuzzy logic". In: *IEEE software* 11.6 (1994), pp. 48–56 (see p. 119).

[160] Thomas Breuer et al. "Realization of Minimum and Maximum Gate Function in $Ta_2O_5$-based Memristive Devices". In: *Scientific reports* 6 (2016) (see p. 119).

[161] Yoeri van de Burgt et al. "A non-volatile organic electrochemical device as a low-voltage artificial synapse for neuromorphic computing". In: *Nature Materials* 16.4 (2017), pp. 414–418 (see p. 119).

[162] Jan Łukasiewicz. "Selected works". In: (1970) (see p. 120).

[163] Kai-Shin Li et al. "Study of sub-5 nm RRAM, tunneling selector and selector less device". In: *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 385–388 (see p. 126).

[164] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. "ReVAMP: ReRAM based VLIW architecture for in-memory computing". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 782–787. DOI: 10.23919/DATE.2017.7927095 (see p. 127).

[165] Christian Lang and Bernd Steinbach. "Bi-decomposition of function sets in multiple-valued logic for circuit design and data mining". In: *Artificial Intelligence Review* 20.3 (2003), pp. 233–267 (see pp. 127, 137, 139, 144).

[166] Alan Mishchenko, Marek Perkowski, and Bernd Steinbach. "Bi-decomposition of multi-valued relations". In: (2001) (see pp. 127, 139).

[167] Rolf Drechsler, Mitch Thornton, and David Wessels. "MDD-based synthesis of multi-valued logic networks". In: *Multiple-Valued Logic, 2000.(ISMVL 2000) Proceedings. 30th IEEE International Symposium on*. IEEE. 2000, pp. 41–46 (see pp. 127, 139).

[168] D Michael Miller and Rolf Drechsler. "On the construction of multiple-valued decision diagrams". In: *Multiple-Valued Logic, 2002. ISMVL 2002. Proceedings 32nd IEEE International Symposium on*. IEEE. 2002, pp. 245–253 (see p. 127).

[169] F Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays · trees · hypercubes*. Elsevier, 2014 (see p. 130).

[170] Lotfi A Zadeh. "Fuzzy logic and approximate reasoning". In: *Synthese* 30.3 (1975), pp. 407–428 (see pp. 139, 144).

[171] BR Gaines. "Fuzzy reasoning and the logics of uncertainty". In: *Proceedings of the sixth international symposium on Multiple-valued logic*. IEEE Computer Society Press. 1976, pp. 179–188 (see p. 139).

[172] M Katz. "Two systems of multi-valued logic for science". In: *Proceedings of the 11th International Symposium on Multiple-valued Logic*. 1981, pp. 175–182 (see p. 139).

[173] Robert McNaughton. "A theorem about infinite-valued sentential logic". In: *The Journal of Symbolic Logic* 16.1 (1951), pp. 1–13 (see p. 141).

[174] Jonathan W Mills and Charles A Daffinger. "CMOS VLSI Lukasiewicz logic arrays". In: *Application Specific Array Processors, 1990. Proceedings of the International Conference on*. IEEE. 1990, pp. 469–480 (see p. 141).

[175] Werner Van Leekwijck and Etienne E Kerre. "Defuzzification: criteria and classification". In: *Fuzzy sets and systems* 108.2 (1999), pp. 159–178 (see p. 144).

[176] Portland Logic and Optimization Group. *Polo Web Page* (see p. 144).

[177] John Preskill. "Quantum computing and the entanglement frontier". In: *arXiv preprint arXiv:1203.5813* (2012) (see p. 149).

[178] Edwin Pednault et al. "Breaking the 49-qubit barrier in the simulation of quantum circuits". In: *arXiv preprint arXiv:1710.05867* (2017) (see p. 149).

[179] *Bristlecone, Google's New Quantum Processor*. https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html. Accessed: 2018-04-21 (see p. 149).

[180] MI Dyakonov. "Is fault-tolerant quantum computation really possible?" In: *arXiv preprint quant-ph/0610117* (2006) (see p. 149).

[181] Alexis De Vos and Yvan Van Rentergem. "Young subgroups for reversible computers". In: *Advances in Mathematics of Communications* 2.2 (2008), pp. 183–200 (see p. 152).

[182] Dmitri Maslov. "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization". In: *Physical Review A* 93.2 (2016), p. 022311 (see p. 153).

[183] Matthew Amy et al. "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits". In: *IEEE TCAD* 32.6 (2013), pp. 818–830 (see pp. 153, 154).

[184] Robert Wille et al. "Trading off circuit lines and gate costs in the synthesis of reversible logic". In: *Integration, the VLSI Journal* 47.2 (2014), pp. 284–294 (see p. 154).

[185] Anupam Chattopadhyay, Nilanjan Pal, and Soumajit Majumder. "Ancilla-Quantum Cost Tradeoff during Reversible Logic Synthesis using Exclusive Sum-of-Products". In: *arXiv preprint arXiv:1405.6073* (2014) (see p. 154).

[186] Alex Parent, Martin Roetteler, and Michele Mosca. "Improved reversible and quantum circuits for Karatsuba-based integer multiplication". In: *arXiv preprint arXiv:1706.03419* (2017) (see p. 154).

[187] Mathias Soeken et al. "Hierarchical reversible logic synthesis using LUTs". In: *DAC*. IEEE. 2017, pp. 1–6 (see p. 154).

[188] Mathias Soeken and Anupam Chattopadhyay. "Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis". In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM. 2016, p. 149 (see p. 154).

[189] Andreas Kuehlmann et al. "Robust Boolean reasoning for equivalence checking and functional property verification". In: *IEEE TCAD* 21.12 (2002), pp. 1377–1394 (see p. 154).

[190] Sayak Ray et al. “Mapping into LUT structures”. In: *Proceedings of the Conference on DATE*. EDA Consortium. 2012, pp. 1579–1584 (see p. 154).

[191] Deming Chen and Jason Cong. “DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs”. In: *Proceedings of the 2004 ICCAD*. IEEE Computer Society. 2004, pp. 752–759 (see p. 154).

[192] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. 2000 (see p. 154).

[193] Nabila Abdessaied et al. “Technology mapping of reversible circuits to Clifford+ T quantum circuits”. In: *ISMVL*. IEEE. 2016, pp. 150–155 (see p. 154).

[194] Matthew Amy, Dmitri Maslov, and Michele Mosca. “Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 33.10 (2014), pp. 1476–1489 (see p. 154).

[195] Md Mazder Rahman et al. “Integrated synthesis of linear nearest neighbor ancilla-free MCT circuits”. In: *Multiple-Valued Logic (ISMVL), 2016 IEEE 46th International Symposium on*. IEEE. 2016, pp. 144–149 (see p. 157).

[196] Mathias Soeken et al. “Ancilla-free synthesis of large reversible functions using binary decision diagrams”. In: *Journal of Symbolic Computation* 73 (2016), pp. 1–26 (see p. 157).

[197] Ravi Sethi. “Complete register allocation problems”. In: *SIAM journal on Computing* 4.3 (1975), pp. 226–248 (see p. 157).

[198] Charles H Bennett. “Time/space trade-offs for reversible computation”. In: *SIAM Journal on Computing* 18.4 (1989), pp. 766–776 (see p. 157).

[199] Charles H. Bennett. “Logical reversibility of computation”. In: *IBM journal of Research and Development* 17.6 (1973), pp. 525–532 (see pp. 157, 159).

[200] Alex Parent, Martin Roetteler, and Krysta M Svore. “Reversible circuit compilation with space constraints”. In: *arXiv preprint arXiv:1510.00377* (2015) (see p. 157).

[201] Matthew Amy. “Algorithms for the Optimization of Quantum Circuits”. MA thesis. University of Waterloo, 2013 (see p. 157).

[202] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. “Reversible logic synthesis of k-input, m-output lookup tables”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2013, pp. 1235–1240 (see p. 157).

[203] Mathias Soeken et al. “Logic Synthesis for Quantum Computing”. In: *arXiv preprint arXiv:1706.02721* (2017) (see p. 157).

[204] Siu Man Chan. “Just a pebble game”. In: *Computational Complexity (CCC), 2013 IEEE Conference on*. IEEE. 2013, pp. 133–143 (see p. 159).

[205] M. Soeken et al. “RevKit: A Toolkit for Reversible Circuit Design”. In: *Proceedings of the International Symposium on Multiple-Valued Logic*. RevKit is available at http://www.revkit.org. 2008 (see p. 167).

[206] Robert Wille et al. “Trading off circuit lines and gate costs in the synthesis of reversible logic”. In: *Integration* 47.2 (2014), pp. 284–294 (see p. 168).

[207] Giulia Meuli et al. “A best-fit mapping algorithm to facilitate ESOP-decomposition in Clifford+T quantum network synthesis”. In: *ASP-DAC*. 2018, pp. 664–669 (see p. 168).

[208] Andrew W. Cross, David P. Divincenzo, and Barbara M. Terhal. “A Comparative Code Study for Quantum Fault Tolerance”. In: *Quantum Info. Comput.* 9.7 (July 2009), pp. 541–572 (see p. 171).

[209] A. G. Fowler et al. “Surface codes: Towards practical large-scale quantum computation”. In: *Physical Review A* 86.3, 032324 (Sept. 2012), p. 032324. DOI: 10.1103/PhysRevA.86.032324. arXiv: 1208.0928 [quant-ph] (see p. 171).

[210] D. Maslov, S. M. Falconer, and M. Mosca. “Quantum Circuit Placement”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.4 (2008), pp. 752–763 (see pp. 172, 178, 179).

[211] Antonio D Córcoles et al. “Demonstration of a quantum error detection code using a square lattice of four superconducting qubits”. In: *Nature communications* 6 (2015) (see p. 172).

[212] H. Kampermann and W.S. Veeman. “Quantum Computing Using Quadrupolar Spins in Solid State NMR”. In: *Quantum Information Processing* 1.5 (2002), pp. 327–344. DOI: 10.1023/A:1023461628937 (see p. 172).

[213] Kenneth R Brown, Jungsang Kim, and Christopher Monroe. “Co-designing a scalable quantum computer with trapped atomic ions”. In: *arXiv preprint arXiv:1602.02840* (2016) (see p. 172).

[214] Simon C Benjamin et al. “Towards a fullerene-based quantum computer”. In: *Journal of Physics: Condensed Matter* 18.21 (2006), S867 (see p. 172).

[215] Yuichi Hirata et al. “An efficient conversion of quantum circuits to a linear nearest neighbor architecture”. In: *Quantum Information & Computation* 11.1&2 (2011), pp. 142–166 (see p. 177).

[216] Robert Beals et al. “Efficient distributed quantum computing”. In: *Proc. R. Soc. A*. Vol. 469. 2153. The Royal Society. 2013, p. 20120686 (see pp. 177, 179).

[217] Stephen Brierley. “Efficient implementation of Quantum circuits with limited qubit interactions”. In: *CoRR* abs/1507.04263v2 (2016) (see pp. 177, 179).

[218] Mark Whitney et al. “Automated Generation of Layout and Control for Quantum Circuits”. In: *Proceedings of the 4th International Conference on Computing Frontiers*. CF ’07. 2007, pp. 83–94 (see pp. 178, 179).

[219] R. Wille et al. “Look-ahead schemes for nearest neighbor optimization of 1D and 2D quantum circuits”. In: 2016, pp. 292–297 (see pp. 178, 188).

[220] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. “Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits”. In: 2014, pp. 495–500 (see p. 178).

[221] Md. Mazder Rahman, Gerhard W. Dueck, Anupam Chattopadhyay and Robert Wille. “Integrated Synthesis of Linear Nearest Neighbor Ancilla-Free MCT Circuits”. In: 2016 (see p. 178).

[222] Md. Mazder Rahman, Gerhard W. Dueck, and Joseph D. Horton. “An Algorithm for Quantum Template Matching”. In: *J. Emerg. Technol. Comput. Syst.* 11.3 (Dec. 2014), 31:1–31:20. DOI: 10.1145/2629537 (see p. 178).

[223] A. Lye, R. Wille, and R. Drechsler. "Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits". In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 178–183. DOI: 10.1109/ASPDAC.2015.7059001 (see p. 179).

[224] Amlan Chakrabarti, Susmita Sur-Kolay, and Ayan Chaudhury. "Linear Nearest Neighbor Synthesis of Reversible Circuits by Graph Partitioning". In: *CoRR* abs/1112.0564 (2011) (see p. 179).

[225] Austin G. Fowler, Charles D. Hill, and Lloyd C. L. Hollenberg. "Quantum-error correction on linear-nearest-neighbor qubit arrays". In: *Phys. Rev. A* 69 (4 2004), p. 042314. DOI: 10.1103/PhysRevA.69.042314 (see p. 179).

[226] L. Biswal et al. "Nearest-Neighbor and Fault-Tolerant Quantum Circuit Implementation". In: *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*. 2016, pp. 156–161 (see p. 179).

[227] D. Leung, J. Oppenheim, and A. Winter. "Quantum Network Communication - The Butterfly and Beyond". In: *IEEE Transactions on Information Theory* 56.7 (2010), pp. 3478–3490 (see p. 180).

[228] A Nico Habermann. "Parallel neighbor-sort (or the glory of the induction principle)". In: (1972) (see p. 180).

[229] RA Spanke and VE Benes. "N-stage planar optical permutation network". In: *Applied Optics* 26.7 (1987), pp. 1226–1229 (see p. 180).

[230] Ignasi Sau. "Optimal permutation routing on mesh networks". In: (see p. 180).

[231] R. Wille et al. "RevLib: An Online Resource for Reversible Functions and Reversible Circuits". In: *Int'l Symp. on Multi-Valued Logic*. RevLib is available at http://www.revlib.org. 2008, pp. 220–225 (see pp. 184, 188).

[232] Adriano Barenco et al. "Elementary gates for quantum computation". In: *Physical review A* 52.5 (1995), p. 3457 (see p. 187).

[233] Dmitri Maslov et al. "Quantum circuit simplification and level compaction". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.3 (2008), pp. 436–444 (see p. 187).

[234] Arabzadeh Mona and Mehdi Saeedi. *RCViewer+, version 2.5, 2017*. http://ceit.aut.ac.ir/QDA/RCV.htm (see p. 187).

[235] Abhoy Kole, Kamalika Datta, and Indranil Sengupta. "A Heuristic for Linear Nearest Neighbor Realization of Quantum Circuits by SWAP Gate Insertion Using $N$-Gate Lookahead". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.1 (2016), pp. 62–72 (see pp. 188, 189).

[236] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. "Synthesis of quantum circuits for linear nearest neighbor architectures". In: *Quantum Information Processing* 10.3 (2011), pp. 355–377 (see pp. 188, 189).

[237] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. "Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures". In: *Proceedings of the 50th Annual Design Automation Conference*. ACM. 2013, p. 41 (see pp. 188, 189).

[238] Aaron Lye, Robert Wille, and Rolf Drechsler. "Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits". In: *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*. IEEE. 2015, pp. 178–183 (see p. 188).

[239] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. "Qubit placement to minimize communication overhead in 2D quantum architectures". In: *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE. 2014, pp. 495–500 (see p. 188).