Thèse n° 8164

Data Structures and Algorithms for Logic Synthesis in Advanced Technologies

Présentée le 7 septembre 2020

à la Faculté informatique et communications
Laboratoire des systèmes intégrés (IC/STI)
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Eleonora TESTA

Acceptée sur proposition du jury

Prof. P. Ienne, président du jury
Prof. G. De Micheli, Dr M. Soeken, directeurs de thèse
Prof. P.-E. Gaillardon, rapporteur
Prof. J.-H. R. Jiang, rapporteur
Prof. A. P. Burg, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

"Brilliant," said Hermione. "This isn't magic – it's logic – a puzzle. A lot of the greatest wizards haven't got an ounce of logic, they'd be stuck in here forever."
— Harry Potter and the Philosopher's Stone

To my beloved family

Acknowledgements

PhD is universally recognized as a very stressful and demanding period, but, for me, it has also been rewarding and full of good memories. I am grateful to many people for help and assistance in these five years.

First, I would like to express my sincere gratitude to my advisor Prof. Giovanni De Micheli (Nanni), for his constant motivation, guidance, and immense knowledge of logic synthesis. During these last years, Nanni demanded a lot, but he was also fair and understanding towards me and my colleagues. I could not have hoped for a better advisor and laboratory environment. I wish to extend my deepest appreciation to my PhD co-advisor Dr. Mathias Soeken for his bits of advice and support throughout my PhD. Thank you, Mathias. This thesis would not have been possible without your valued help and limitless patience.

Besides my advisors, I would like to recognize the invaluable assistance of Dr. Luca Amarù for allowing me to be as Synopsys Inc. and for his guidance and constant motivation. Thanks for sharing your bright ideas with me and for your trust in my abilities. I wish to extend my special thanks to Prof. Pierre-Emmanuel Gaillardon, for giving me the opportunity to start my PhD within the LSI and for all our collaborations. I would also like to express my deepest appreciation to Dr. Alan Mishchenko, Dr. Heinz Riener, and Dr. Patrick Vuillod for the great research collaborations we had. My special thanks go to Prof. Francky Catthoor, Dr. Odysseas Zografos and Dr. Julien Ryckaert for their assistance of my PhD at IMEC. I also would like to extend my gratitude to the members of my oral committee Prof. P. Ienne and Prof. A. P. Burg from EPFL, and Prof. J.-H. R. Jiang from the National Taiwan University.

My sincere thanks also go to my colleagues at the LSI. Not only you provided me with a stimulating research environment, but you have also made these five years more cheerful and carefree. I would like to recognize the invaluable assistance that you all provided during my study. A special thanks also to Francesca, Giulia, Ivan, and Simone for being not only great colleagues but dear friends. In particular, thanks to Winston, Bruno, and Giulia for putting up with me during long months of TA. I recognize that it is not always easy. I wish to show my gratitude also to Aya, Fereshte, Nadja, and Tugba, and to all the people that stopped at LSI, even for a little while: Eleonora, Lucia, and Sofia. I would like to express my deepest gratitude to Dewmini, for all her invaluable work, and in general to all my students for teaching me more than I thought them. I wish also to acknowledge the help provided by the technical IT staff, and the invaluable assistance of Carole, Chantal, and Christina. I would like to thank all my other friends in Lausanne, Leuven, and Sunnyvale; in particular, Elisa, Enrico, Martina,

Mattia, Michela, and Mish. Finally, I would like to extend my deepest and sincere gratitude to Francesca and Giovanni. You have made my PhD life easier and happier since the first day. Thank you for your priceless friendship.
Special grazie to my italian friends. For your visits to Lausanne, the holidays together, and for distracting me from my PhD life. Thanks to Carlo, Chiara, Eleonora, Luca, Marco, and Mattia. My sincere thanks also go to my lifelong friends Allegra, Elisa, Marta, and Serena. "Grazie per esserci sempre".

I would like to thank my family, in particular, Beatrice, Enrico, Milda, and Silvia. "Grazie a tutti per il costante supporto e affetto. Grazie mamma per il coraggio". Thanks also to Fabio, who makes my sister Silvia much happier and nicer. Last, but not least, my biggest thanks to the person who most of all helped, comforted, and supported me in the past five years. Thanks, Enrico, for being close to me even at 9000 km and 9 hours away.

April 3, 2020

Abstract

Logic synthesis is a key component of digital design and modern *electronic design automation* (EDA) tools; it is thus an essential instrument for the design of leading-edge chips, and to push the limits of performance (upwards) and power consumption (downwards). In the last two decades, the electronic circuits and digital systems community has evolved dramatically, facing many architectural and technological changes. Consequently, EDA and logic synthesis have adapted and changed to be able to accurately design the new generation of digital systems. In the present day, logic synthesis is an important area of research for two main reasons: (i) Many and diverse ways of computation, alternative to *complementary metal–oxide–semiconductor* (CMOS), have been presented in the last years. Post-silicon technologies (called emerging technologies) have been shown to be feasible and may provide us with better – more efficient – electronic devices. In a similar way, novel areas of applications of logic synthesis are emerging, ranging from deep learning to cryptography and security applications. (ii) The current computing and storage means make it possible to solve exactly problems that were only approximated before. Moreover, new reasoning engines, covering from deep learning to new SAT-solvers, can be used as a mean of computation, thus possibly unlocking novel optimization opportunities and enabling hardware of higher-quality.

The objective of this thesis is to advance state-of-the-art logic synthesis and present a variety of novel data structures and algorithms, addressing diverse types of applications in modern logic synthesis flows, considering standard CMOS design as well as emerging technology and cryptography.

Motivated by the many emerging technologies that implement majority gates, we first focus on majority-based logic synthesis. We present novel algorithms over the recently introduced *majority-inverter graph*s (MIGs). First, a novel optimization flow based on Boolean transformations is proposed. Then, we demonstrate how technology-dependent logic synthesis is an essential step for the abstraction and manipulation of novel and diverse majority-based emerging technologies and, more important, their technological constraints. Moreover, we advance state-of-the-art theoretical results on majority logic. In particular, we mainly focus on the problem of "how best can the n-argument majority function (majority-n) be realized with a network of 3-input majority gates?". For this purpose, we present novel general upper bounds and decompositions, together with optimum results for majority-5 and -7 and best-known results for the majority-9 function. In the second part, we shift into more pragmatic results and show practical aspects of logic synthesis, designed to be successful in modern logic synthesis flows. We focus here on XOR-based logic synthesis. Motivated by the novel computing capabilities, we propose a novel optimization flow based on the Boolean difference for area optimization of standard CMOS technologies (for *application specific integrated circuits* (ASICs) design). Moreover, we establish a novel application of logic synthesis to cryptography and security applications. It has been demonstrated that the number of AND gates in a *xor-and graph* (XAG) correlates with the degree of vulnerability (security) of cryptography benchmarks and plays an important role in high-level cryptography protocols such as *multi-party computation* (MPC) and *fully homomorphic encryption* (FHE). We introduce a complete and automatic synthesis flow which consists of the main transformations involved in logic synthesis but aims instead at minimization of the number of AND gates over XAGs. Our tool and methods obtain significant results over both EPFL and cryptography and security benchmarks.

We argue that given the progress and novel opportunities of technology, logic synthesis has to be revisited while considering the plurality of primitives and novel engines that can be of interest, and, consequently, the corresponding objective functions and optimization problems.

Keywords Electronic design automation, logic synthesis, majority logic, emerging technologies

Sommario

La sintesi logica è una componente chiave del design di circuiti elettronici digitali e dei moderni strumenti per *electronic design automation* (EDA); è quindi uno strumento essenziale per la progettazione di chip all'avanguardia e per spingere i limiti delle prestazioni (verso l'alto) e del consumo di energia (verso il basso). Negli ultimi due decenni, i circuiti elettronici e i sistemi digitali si sono evoluti notevolmente, affrontando molti cambiamenti architettonici e tecnologici. Di conseguenza, l'EDA e la sintesi logica si sono adattate e modificate per poter progettare con precisione questa nuova generazione di circuiti digitali. Al giorno d'oggi, la sintesi logica è un'importante area di ricerca per due motivi principali: (i) Negli ultimi anni sono state presentate molte e diverse technologie alternative al *complementary metal–oxide–semiconductor* (CMOS). Le tecnologie post-silicio (chiamate emerging technologies) hanno dimostrato di essere non solo realizzabili, ma potrebbero fornire dispositivi elettronici migliori e più efficienti. Allo stesso modo, stanno emergendo nuove aree di applicazione della sintesi logica, che vanno dal deep learning alla crittografia e alla sicurezza. (ii) Gli attuali mezzi di computazione e memorizzazione consentono di risolvere in maniera esatta problemi che sono stati finora risolti solo euristicamente. Inoltre, nuovi approcci come deep learning e SAT-solvers, possono essere utilizzati come mezzo di calcolo, potenzialmente sbloccando nuove opportunità di ottimizzazione e consentendo hardware di qualità superiore.

L'obiettivo di questa tesi è far avanzare la sintesi logica e presentare una varietà di nuove strutture dati e algoritmi, dedicandosi a diversi tipi di applicazioni nella sintesi logica moderna. Consideriamo quindi sia il design di circuiti elettronici basati su CMOS, nonché emerging technologies e applicazioni per la crittografia.

Motivati dalle molte emerging technologies che implementano la funzione logica del majority (funzione logica di maggioranza), ci concentriamo in primo luogo sulla sintesi logica basata sul majority e presentiamo nuovi algoritmi per *majority-inverter graph*s (MIGs). Innanzitutto, viene proposto un nuovo metodo di ottimizzazione dell'area degli MIG basato su trasformazioni Booleane. Quindi, dimostriamo come la sintesi logica debba dipendere dalla tecnologia e come questa sia un passo essenziale per l'astrazione e la manipolazione di nuove e diversificate emerging technologies e, più importante, dei loro limiti e vincoli tecnologici. Inoltre, avanziamo i risultati teorici sulle proprietá della logica a majority. In particolare, ci concentriamo principalmente sul problema di "come può essere realizzata al meglio la funzione majority con n ingressi (majority- n) con un grafo di porte a majority-3?". A tale scopo, presentiamo nuovi limiti superiori sulla loro realizzazione e nuove scomposizioni, insieme a risultati ottimi per il majority-5 e -7 e risultati migliori (conosciuti finora) per la funzione del majority-9. Nella seconda parte, passiamo a risultati più pragmatici e mostriamo aspetti pratici della sintesi logica, progettati per avere successo nei sistemi di sintesi logica industriali e moderni. Ci concentriamo qui sulla sintesi logica basata su OR esclusivo (XOR). Motivati dalle moderne capacità di computazione ed elaborazione, proponiamo un nuovo flusso di sintesi basato sulla differenza Booleana per l'ottimizzazione dell'area di circuiti basati su tecnologia CMOS (circuiti *application specific integrated circuits* (ASICs)). Inoltre, stabiliamo una nuova applicazione della sintesi logica nel campo della crittografia e della sicurezza. È stato dimostrato che il numero di porte AND in *xor-and graph* (XAG) è correlato al grado di vulnerabilità (sicurezza) dei circuiti e protocolli di crittografia e ha un ruolo centrale in applicazioni come *multi-party computation* (MPC) e *fully homomorphic encryption* (FHE). Introduciamo un nuovo metodo di sintesi completo e automatico che consiste nelle principali trasformazioni coinvolte nella sintesi logica, ma mira invece a minimizzare il numero di porte AND su XAGs. Il nostro tool e i nostri metodi ottengono risultati significativi sia sui circuiti proposti dall'EPFL che su quelli per la crittografia e la sicurezza.

Sosteniamo che alla luce dei progressi e delle nuove opportunità date dalla tecnologia, la sintesi logica debba essere rivisitata tenendo conto della pluralità di primitive Booleane e nuovi metodi di computazione, e di conseguenza, dei corrispondenti e diversi problemi di ottimizzazione.

Parole Chiave Electronic design automation, sintesi logica, funzione di maggioranza, nuove nanotecnologie

Contents

Acknowledgements **v**

Abstract (English/Italian) **vii**

List of Figures **xv**

List of Tables **xvii**

List of Acronyms **xix**

1 Introduction **1**

1.1 Electronic Design Automation 3
1.2 Research Motivation 5
1.2.1 Modern CMOS Technologies 5
1.2.2 Alternative Applications 6
1.3 Thesis Contributions 6
1.3.1 Majority-based Logic Syntesis 7
1.3.2 XOR-based Logic Synthesis 9
1.4 Thesis Organization 10

2 Background **13**

2.1 Data Structures 13
2.1.1 Truth Tables 13
2.1.2 2-level Representations 14
2.1.3 Binary Decision Diagrams 15
2.1.4 Multi-level Logic Networks 16
2.2 Algorithms 18
2.2.1 Algebraic Methods 19
2.2.2 Boolean Methods 22
2.2.3 Exact Methods 27
2.3 Efficient Circuit Partitioning 28
2.3.1 Small Scale: Cut Enumeration 29
2.3.2 Detecting All Reconvergent MFFC 30
2.3.3 Windowing 31
2.4 Summary 32

3 Majority-based Logic Synthesis **33**
- 3.1 Motivation 33
- 3.2 Majority-based Emerging Technologies 35
- 3.3 Boolean Resynthesis for MIGs 39
  - 3.3.1 Functional Reduction 39
  - 3.3.2 Boolean Resubstitution 40
  - 3.3.3 Replacement Optimization 41
  - 3.3.4 Experimental Results 42
- 3.4 Inverter Propagation and Fan-out Constraint 43
  - 3.4.1 Inversion Propagation 44
  - 3.4.2 Fan-Out Restriction 46
  - 3.4.3 Experimental Results 48
- 3.5 Exact Synthesis with Constraints 49
  - 3.5.1 Preliminaries 49
  - 3.5.2 Constraints Encoding 52
  - 3.5.3 Exact Algorithms 54
  - 3.5.4 Experimental Results 57
- 3.6 Area-Delay-Energy Product for QCA and STMG 60
- 3.7 Summary 63

4 Majority-n Logic **65**
- 4.1 Motivation 66
- 4.2 Preliminaries 68
  - 4.2.1 Binary Decision Diagrams 69
  - 4.2.2 Majority Graphs 70
- 4.3 Decomposing Majority-n into Majority-3 71
  - 4.3.1 Transforming BDDs into Majority Graphs 71
  - 4.3.2 Mapping Majority-*n* 74
- 4.4 The Complexity of Self-Dual Monotone 7-input Functions 81
  - 4.4.1 Preliminaries 82
  - 4.4.2 Length and Combinational Complexity 84
  - 4.4.3 Experimental Results 87
- 4.5 Pairs of Majority-Decomposing Functions 92
  - 4.5.1 Proof of the Main Theorem 94
  - 4.5.2 Majority-*n* Decompositions 94
  - 4.5.3 Application to Find Optimum Majority Networks 97
- 4.6 Summary 97

5 XOR-based Logic Synthesis **99**
- 5.1 Motivation 100
- 5.2 Boolean Difference Method 102
  - 5.2.1 Theory 102
  - 5.2.2 Boolean Difference Optimization 103

5.2.3 Experimental Results 106
5.3 Multiplicative Complexity and Cryptography Applications 110
5.3.1 Preliminaries 110
5.3.2 General XOR-based Resynthesis Framework 112
5.3.3 Experimental Results 118
5.4 Summary 125

6 Conclusions 127
6.1 Summary of Thesis Contributions 127
6.2 Open Problems 129

Bibliography 131

Curriculum Vitae 147

List of Figures

1.1 Novel and standard applications for logic synthesis 2
1.2 Simplified EDA flow 3
1.3 Typical logic synthesis flow 4
1.4 Explanation of the majority function 8

2.1 Example of BDD 15
2.2 Different logic networks for a 4-bit adder 17
2.3 Example of various MIG structures 18
2.4 Example of AIG rewriting 19
2.5 Example of MIG optimization using the algebraic rewriting 21
2.6 Boolean rewriting in MIGs 27
2.7 Example of 3-cut over AIG 29

3.1 Chapter organization 34
3.2 QCA layout for various gates 37
3.3 Schematic of STMG and 3 stages plasmonic gate 37
3.4 Example of MIG Boolean resubstitution 40
3.5 Example of inverter propagation on MIG 45
3.6 Example of inverter propagation with size increase 45
3.7 Example of fan-out restriction on MIG 47
3.8 Exact synthesis: 2-input and 3-input Boolean networks example 50

4.1 Chapter organization 66
4.2 Example of BDD for function clique$_{4,3}$ 69
4.3 Example of majority-graph 71
4.4 Example of leafy majority graph 73
4.5 Mapping the majority-3 from its optimal BDD 74
4.6 Decomposing majority-5 into majority-3 76
4.7 Decomposing majority-7 into majority-3 77
4.8 Optimum majority-7 78
4.9 Decomposing majority-9 into majority-3 79

5.1 Chapter organization 100
5.2 Boolean difference example 103

5.3 Example of XAG 110
5.4 Rewriting example over XAG 113
5.5 Resubstitution and refactoring example over XAG 117

List of Tables

1.1 Correspondance between chapters, contents, and publications 7

3.1 Results for size optimization over MIGs 43
3.2 Results of inversion propagation and fan-out restriction over MIGs 48
3.3 Comparison of the proposed exact synthesis to state-of-the-art methods 58
3.4 Comparison of quality of results of the proposed exact synthesis algorithms . . 59
3.5 Results of exact synthesis with constraints on LUTs of larger benchmarks 59
3.6 Technology parameters for QCA 60
3.7 Inversion-free and fan-out restriction on QCA 61
3.8 Technology parameters for STMG 62
3.9 Inversion-free and fan-out restriction on STMG 63
3.10 Results for size optimization after QCA and STMG mapping 64

4.1 Upper bounds on the number of majority-3 operations to realize majority-n . 81
4.2 $L(f)$ and $C(f)$ for self-dual monotone 7-input classes 88
4.3 $L(f)$ and $C(f)$ for self-dual monotone 7-input functions from state-of-the-art . 91
4.4 $C(f)$ for self-dual monotone 7-input functions with inverters 91

5.1 Best 6-LUT area results for the EPFL benchmarks 108
5.2 Smallest AIG results for the EPFL benchmarks 109
5.3 Post Place&Route results on 36 industrial design for ASICs 109
5.4 Experimental results for EPFL benchmarks using rewriting 120
5.5 Experimental results for EPFL benchmarks with complete flow 122
5.6 Experimental results for MPC and FHE benchmarks using the rewriting algorithm 123
5.7 Experimental results for MPC and FHE benchmarks using the complete flow . 123
5.8 Experimental results for MPC benchmarks 125

List of Acronyms

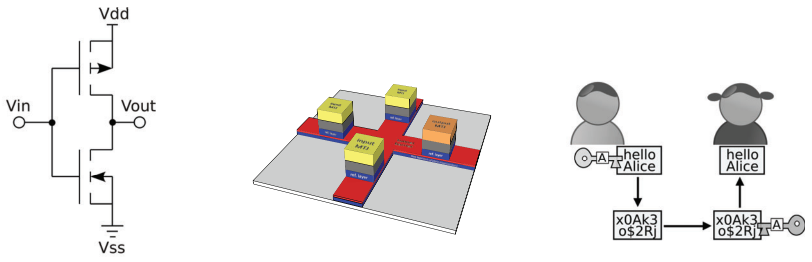
ADEP	*area-delay-energy product*
AIG	*and-inverter graph*
AOI	*and-or-inverter*
AOIG	*and-or-inverter graph*
ASICs	*application specific integrated circuits*
BDD	*binary decision diagram*
CEGAR	*counter-example-guided abstraction refinement*
CMOS	*complementary metal–oxide–semiconductor*
CNF	*conjunctive normal form*
EDA	*electronic design automation*
FHE	*fully homomorphic encryption*
FPGAs	*field-programmable gate arrays*
ICs	*integrated circuits*
MFFC	*maximum fan-out free cone*
MIG	*majority-inverter graph*
MPC	*multi-party computation*
MSPF	*maximum set of permissible functions*
MTJ	*magnetic tunnel junctions*
PI	*primary input*
PO	*primary output*

QCA	*quantum-dot cellular automata*
QoR	*quality of results*
ReRAMs	*resistive RAMs*
RTL	*register-tranfer level*
SAT	*satisfiability*
SET	*single-electron transistor*
SOP	*sum-of-product*
SPP	*surface plasmon polaritons*
STMG	*spin torque majority gate*
STT	*spin transfer torque*
TFO	*transitive fan-out*
XAG	*xor-and graph*
XMG	*xor-majority graph*

1 Introduction

Electronic circuits are omnipresent and considered an essential part of our everyday life. In the last decades, they have experienced tremendous growth that has been enabled by a wide range of tools to abstract, represent, and manipulate digital circuits as well as to optimize their realization. Such tools are called, all together, *electronic design automation* (EDA) tools and are responsible for the fast evolution of electronic circuits. This progress of electronic circuits was also due to the continuous downscaling of *complementary metal–oxide–semiconductor* (CMOS) transistors dimensions that enabled the semiconductor industry to decrease the cost and area of digital designs while increasing their performance [129]. Nowadays, transistor scaling is reaching the limit of what is physically achievable; EDA tools are thus left with the important challenge of further pushing the performances and *quality of results* (QoR) of digital circuits.

Logic synthesis is a key component of digital design and modern EDA tools [2, 3, 4]; it is thus an essential instrument for the design of leading-edge chips, and to push the limits of performance (upwards) and power consumption (downwards). In the last two decades, the electronic circuits and digital systems community has evolved dramatically, facing many architectural and technological changes. Consequently, EDA and logic synthesis have adapted and changed to be able to accurately design the new generation of digital systems. In the present days, logic synthesis is an important area of research for two main reasons:
(i) Many and diverse ways of computation, alternative to CMOS, have been presented in the last years. Post-silicon technologies (also called emerging nanotechnologies) have been shown to be feasible and may provide us with better – more efficient – electronic devices. In a similar way, new architectures such as neuromorphic [68] and quantum computing [57] are becoming more and more practical. Finally, novel areas of applications of logic synthesis are emerging, ranging from deep learning [53] to cryptography and security applications [149].
(ii) The current computing and storage means make it possible to solve exactly problems that were only approximated before. Moreover, new reasoning engines, covering from deep learning to new SAT-solvers, can be used as a means of computation, thus possibly unlocking novel optimization opportunities and enabling hardware of higher-quality [82, 193].



	CMOS	**Emerging Technologies**	**Cryptography and Security**
Abstraction	NAND/NOR	Majority	{AND,XOR,NOT}
Optimization Goal	Performance-Power-Area (PPA)	Area-Delay-Energy Product (ADEP) in the limit of the technological constraints	Minimization of AND gates

Figure 1.1 – Some examples of novel and standard paradigms of computations and applications for logic synthesis. Standard CMOS-based design uses NAND/NOR, and aims at optimizing the PPA, while emerging technologies are often based on the majority function. Cryptography applications are modeled using the basis {AND,XOR,NOT}. In this case, the minimization of the number of AND gates is desirable for applications such as *fully homomorphic encryption* (FHE) and *multi-party computation* (MPC)

As a result of the novel paradigms of computation, applications, and resources, the circuit primitives for logic design have increased and changed over the years [174]. Some examples – relevant to the rest of the thesis – are presented in Figure 1.1. CMOS technology has always favored circuits based on NORs, NANDs, and their extensions. Thus, today the majority of state-of-the-art tools handle these primitives and their extensions (e.g., AND-OR-Inverter gates) to represent and optimize logic circuits. On the other hand, new emerging technologies, such as some optical technologies [72] and *quantum-dot cellular automata* (QCA) [106], leverage majority (MAJ) and inverter (INV) gates as primitives. Neuromorphic architectures exploit threshold gates, which can be seen as majority gates with weighted inputs, and the basis {AND, XOR, NOT} [36] is often used in the modeling of cryptography circuits. In a similar way, novel cost functions and optimization goals have arisen. As an example, logic synthesis for cryptography considers as optimization goal the number of AND gates, which correlates to the degree of vulnerability of a function. Emerging technologies such as *spin torque majority gate* (STMG) [135], which can efficiently realize the majority operator, are on the contrary unable to implement an inverter. It follows that inverter-free circuits are desirable.

In view of the progress and novel opportunities of technology, logic synthesis has to be revisited while considering the plurality of primitives and novel engines that can be of interest, and, consequently, the corresponding objective functions and optimization problems. The

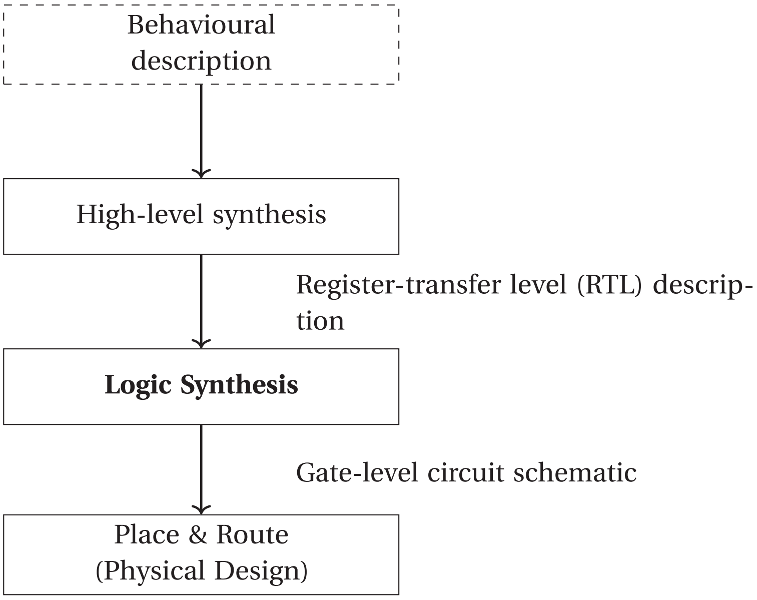

Behavioural description
High-level synthesis
Register-transfer level (RTL) description
Logic Synthesis
Gate-level circuit schematic
Place & Route (Physical Design)

Figure 1.2 – Simplified EDA flow

goal of this thesis is to present a variety of novel data structures and algorithms, addressing diverse types of applications in modern logic synthesis. We thus present several logic synthesis methods and we show their applicability to standard and emerging technology applications.

1.1 Electronic Design Automation

The term EDA refers to the tools, algorithms, and methods used to automatically design *integrated circuits* (ICs) and general electronic systems. A typical EDA flow starts from a high-level description of such electronic system and generates a final implementation in terms of technology components. This is achieved thanks to several steps of logic abstractions and algorithms, usually called: *high-level synthesis*, *logic synthesis*, and *physical design* [69].

For the sake of this thesis, we present an oversimplified EDA flow as depicted in Figure 1.2. In particular, high-level synthesis converts a programming language description (behavioural description) into a *register-tranfer level* (RTL) netlist [49]. This step determines the macroscopic structure of the circuits, and it is thus also called *structural synthesis*. On the contrary, *logic synthesis* produces the microscopic (in terms of logic gates) structure of the circuit [37, 84]. It optimizes and manipulates RTL netlists into logic models that are interconnections of logic primitives. The last step of a standard EDA flow (see Figure 1.2) is *physical design*, which consists of *placement* and *routing* (P&R) [141]. This step is responsible for generating the final layout of the chip and provides the link with the fabrication process. Placement refers to the assignment of positions to the cells while routing deals with their interconnections. Note that, modern EDA flows do not clearly separate between these three steps, but aim at more integration between them [5, 94].

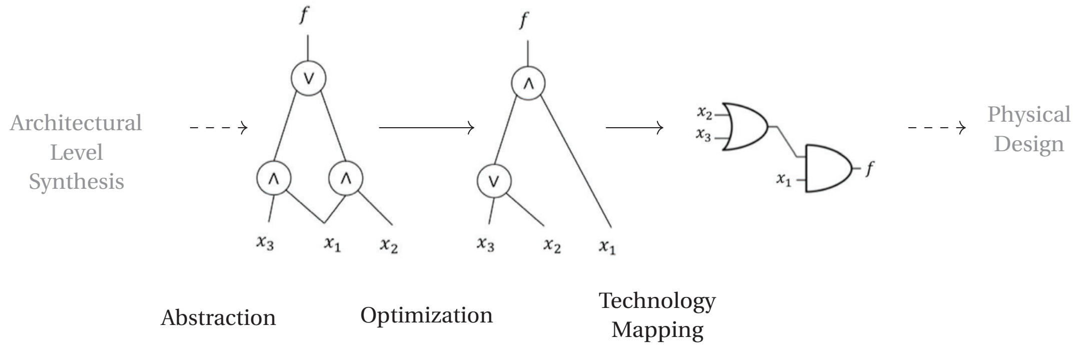


Figure 1.3 – Typical logic synthesis flow. The circuit is abstracted using *and-inverter graph*s (AIGs); the symbols ∨ and ∧ are the logic OR and AND respectively

In conjunction with synthesis, *optimization* is usually considered. It means, all the three presented steps are usually constrained to consider a given optimization metric, that could be for instance area, power, and or delay of the digital circuit. Nowadays, most synthesis and optimization algorithms are tuned to work on CMOS technology. CMOS technology provides an efficient realization of NORs, NANDs and *and-or-inverter*s (AOIs) which can be abstracted as negative unate functions. Consequently, most EDA tools use these primitives to abstract and optimize circuits. In a similar way, placement and routing algorithms changed consequently to the technological evolution of CMOS down to the nano-scale.

In this thesis, we focus on *logic synthesis*. Recall that logic synthesis is the process by which an abstract form of desired circuit behavior, typically at RTL, is turned into an optimized design implementation in terms of logic gates (depicted in detail in Figure 1.3). Common examples of this process include synthesis of hardware description languages (including VHDL and Verilog), which are specialized computer languages used to program the structure, design, and operation of digital logic circuits. Broadly speaking, the overall problem of logic synthesis is one of finding "the best implementation" of a logic function, where the term "best" may depend on goals and computational methods and it may not be unique. Thus, synthesis encompasses also logic optimization.

In logic synthesis, the logic circuit is abstracted as a logic network, which is usually defined over a set of primitive logic gates. The first approaches to logic synthesis addressed two-level *sum-of-product* (SOP) representations and attempted to reduce the cardinality of logic covers (i.e., the number of product-terms also called implicants). For example, the first logic synthesis algorithm (dated back to 1956) is the Quine-McCluskey algorithm [112], which solves the minimization of logic covers exactly. Several approaches to heuristic minimization of two-level forms [69] ended up instead in the program ESPRESSO [37, 154] that had a large impact on the design automation community of the time. Contemporary logic synthesis and its successes have risen in the 80s with the establishment of CMOS technology. In modern industrial flow, efficient multi-level logic representations are involved. Being based on CMOS, modern logic synthesis flow mainly use AND/OR primitives to abstract and optimize logic (see Figure 1.3). Note that most approaches divide synthesis into a *technology-independent* phase, where the interconnection of logic blocks is minimized independently of the library, followed by a *technology mapping* step where the instances of library elements are chosen. As an example of technology mapping, in the case of *application specific integrated circuits* (ASICs), the logic model is mapped into standard cells, while it is transformed into look-up-tables for *field-programmable gate arrays* (FPGAs) [45].

1.2 Research Motivation

In recent years, EDA tools and researches have been facing new and difficult challenges. On one side, as transistor scaling slows down at advanced technology nodes, e.g., 10 nm, 8 nm, 7 nm, EDA solutions are becoming essential to keeping up with the (expected) QoR. This motivates EDA researchers (and logic synthesis) to push further the optimization tools and revisit high-quality and high-computational-complexity optimization methods in light of modern computing capabilities. On the other hand, novel synthesis tools need to adjust and reshape to take into account the novel paradigm of computation given by modern and emerging nanotechnologies, as well as, new fields of application such as quantum computers and security.

This thesis investigates novel logic synthesis data structures and algorithms that find application in both these tasks, being thus responsible for designing competitive CMOS circuits (Section 1.2.1), but also addressing alternative applications (Section 1.2.2) as emerging technologies and cryptography and security applications.

1.2.1 Modern CMOS Technologies

Modern logic synthesis, together with the EDA community, is challenged every day. Novel logic synthesis methods and data structures need to be examined, together with a reinvestigation of already existing methods in light of the modern (and more powerful) computing potential.

As a matter of fact, standard CMOS-based logic synthesis is constrained in using mainly AND/OR primitives. For this reason, in recent years, the logic synthesis community has mainly investigated alternative and more expressive data structure and logic functions. This has resulted for instance in the *majority-inverter graph*s (MIGs) [14] or *xor-majority graph*s (XMGs) [81], which use majority and XOR as the main Boolean function for representing and optimizing logic. Even though remarkable results have been already demonstrated, much work is still needed in this regard, to build *complete* synthesis flows based on these alternative functions and to investigate theoretical properties and optimization potential. Furthermore, expensive Boolean methods (that involve expensive runtime tasks) are usually used cautiously in modern EDA flows, where faster algebraic methods are instead preferred. Boolean methods usually achieve better results as compared to algebraic methods, as they consider the Boolean nature of the functions. This leaves many possible optimization opportunities still unexplored.

1.2.2 Alternative Applications

In the last years, many and diverse emerging technologies have been presented, based on new paradigms and ways of computation. Moreover, recently, logic synthesis has been employed for the optimization of circuits in various domains, ranging from quantum computers to cryptography circuits.

Even though CMOS technology will be the core of modern digital systems for at least another decade, logic synthesis and EDA tools need to start revisiting their optimization methods in order to be able to efficiently abstract and optimize future technologies. Examples of majority-based nanotechnologies include QCA [106], superconducting electronic devices (RSFQ [107] and AQFP [169]), STMG [135] and plasmonic-based devices [72]. For example, the recently proposed plasmonic technology [72] is a promising alternative to CMOS-based design thanks to the waveguide's low power consumption and high speed of computation. Besides these qualities, this novel technology also implements different functionalities uncommon to traditional CMOS. First, plasmonic devices are intrinsically based on the majority-voter operation; moreover, they can efficiently implement threshold functions up to 27-input. On the other hand, this expressive power comes at some costs. Indeed, the 27-input block is the largest building block that can be implemented today using the mentioned plasmonic technology. This is due to the fact that the propagation losses put a limitation on the maximum number of cascaded stages (i.e., the number of levels of the circuits). Currently, it is not efficient to have more than three stages, which means that, after the third stage, either an amplifier or a converter to the voltage domain is necessary [72]. Other technologies, as for example STMG, suffer instead the lack of inverter implementation, thus inverters-free circuits are needed. QCA technology has unefficient implementation of inverters, thus benefit from their minimization [176].

Consequently, logic synthesis and EDA tools are even more important. Not only they need to be able to abstract efficiently such novel functionalities but also to investigate their design capabilities. They are left with the important task of studying and examining the design limits of such technologies in circuit implementation, in order to help in deciding for the next generation of technologies.

1.3 Thesis Contributions

This thesis is centered around logic synthesis, in particular, on novel data structures and algorithms addressing standard CMOS technologies as well as novel applications (i.e., emerging technologies and cryptography). In the following, our contributions are classified according the Boolean function involved in the optimization and abstraction. Our contributions are thus divided into two main categories: *majority-logic* and *XOR-logic*. To ease the reading of

Table 1.1 – Correspondance between chapters, contents, and publications

	Topic	Chapter	Reference
Majority-based Logic			
	Emerging Technologies	Chapter 3	[151, 180, 181]
	Theoretical Results	Chapter 4	[166, 172, 177]
XOR-based Logic			
	Standard Synthesis Flow	Chapter 5	[170, 171]
	Cryptography and Security	Chapter 5	[175, 178]

the thesis, our achievements are here presented in the same order they will appear in the chapters of the thesis. Table 1.1 summarizes the topics and their corresponding publications and chapters.

1.3.1 Majority-based Logic Syntesis

The methods and abstractions presented here are based on the majority function, denoted by $\langle x_1, x_2, \ldots, x_n \rangle$, where n is the (odd) number of inputs. The achievements and their position concerning previous works are organized and summarized according to (i) logic synthesis for emerging technologies, and (ii) theoretical results.

- **Emerging Technologies:** Motivated by the many emerging technologies that easily implement majority gates, this first part of the thesis presents novel algorithms over MIGs. First, a novel optimization flow based on Boolean transformations is proposed. The flow aims at size optimization over MIGs, next mapped into QCA and STMG. Second, novel techniques to consider constraints of emerging technologies inside MIGs state-of-the-art synthesis flows are evaluated. First, we deal with inversion and fan-out limitations of majority-based technologies such as QCA and STMG. Second, we propose an *exact synthesis* method for circuits with limited fan-out and constrained depth. Our results in logic synthesis for emerging technologies are in Chapter 3 and have been published in the following publications [151, 180, 181]. This section is the result of our fruitful collaboration with the IMEC research center in Leuven, Belgium.

Relation to Previous Works

Majority logic owes its renewed interest to many emerging nanotechnologies that naturally implement majority as their primitive building block [18]. Indeed, because Moore's law [129] will reach its end in the next decade [195], alternative devices need to be implemented and designed.

MIGs allow remarkable logic representation and optimization. A logic optimization flow that employs algebraic and Boolean methods for MIG depth reduction has been recently

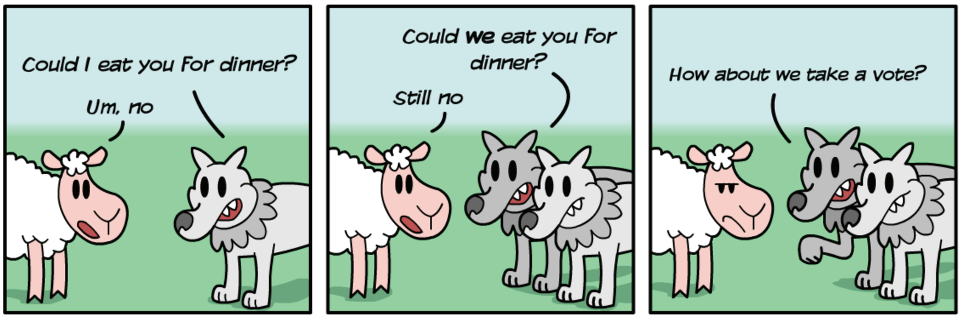

Could I eat you For dinner?
Um, no
Could we eat you For dinner?
Still no
How about we take a vote?

Figure 1.4 – Explanation of the majority function: the majority function of three inputs evaluates to true if and only if at least two of the three inputs are true. Source *http://redpanels.com/36/*

presented in [16]. However, these optimization methods are not able to obtain the same results when considering MIG size. Work on size reduction of MIGs is quite sparse [81, 101, 161] and suffers from scalability issues when exact synthesis is involved [81, 161]. Our goal is to present a flow for the size optimization of MIGs. Being Boolean, our flow has a positive effect on the quality but it is challenging in the implementation. We overcome these limits by using windowing techniques, truth tables, and Boolean filtering rules.

Many recent works have instead specifically considered logic synthesis for emerging technologies [18]. As an example, the works in [127, 194] addresses majority-based nanotechnologies by mapping the resulting networks into majority based devices such as QCA and *single-electron transistor* (SET). The work in [28] considers instead logic synthesis for emerging memories, as *resistive RAMs* (ReRAMs), and the one in [32] demonstrates how to implement the *implication* function with memristors. Finally, Zografos et al. [198] presented an MIG-based method for the synthesis of circuits based on spin-wave devices. In our work, we map majority networks into QCA and STMG by using area, delay, and power estimations obtained at IMEC research institute. Moreover, we also demonstrate how to integrate the constraints of such technologies into the synthesis flow. This is achieved by proposing novel algorithms or by adjusting already existing ones.

- **Theoretical Results:** This part of the thesis continues our investigation of majority-based logic synthesis and MIGs, but shifts to theoretical results and new identities. We present a novel *binary decision diagram* (BDD)-based method to map monotone majority functions over *n* inputs (majority-*n*) into MIGs. In particular, their optimum-size representations are discussed and novel upper bounds and decompositions are investigated. Moreover, *complexity* studies over self-dual monotone functions in terms of majority gates are studied. The results for these theoretical results are presented in Chapter 4 and published in [166, 172, 177].

Relation to Previous Works

Majority logic was intensively studied in the 1960s [8, 12, 115]. In 1964, Amarel et al. [12] proposed algorithms to rewrite majority-*n* into majority-3. At that time, many majority-based algorithms were proposed but, due to the limited available computational resources, they were not followed by implementations. Recently, a generalization to the majority operation of n inputs has been presented in [13, 55]. In [17], the authors consider majority logic decomposition based on BDDs. In [17], majority dominator nodes are used to guide the decomposition process; these nodes allow the identification of candidate functions that can be used to build the majority decomposition. Although BDDs are used in the implementation of their algorithm, it has a very different nature compared to our work, in which we equate BDDs to majority graphs in the case of monotone Boolean functions. Moreover, our method finds upper bounds and decompositions rules that put the basis for novel theoretical investigation on majority-n logic.

Concerning self-dual monotone functions, we make use of exact synthesis to study their complexity, which is defined as the minimum number of gates to realize a Boolean function over majority operators or as its shortest formula. Generally, the study of the complexity of Boolean functions deals with finding some upper bounds [78, 88] or lower bounds [139] over a set of primitives. In our case, we are instead concerned with finding exact numbers for the complexity of functions over majority operators. In [97], the complexity for all 4- and 5-input Boolean functions in terms of 2-input Boolean operators have been studied, while 3-input Boolean operators have been used in [83]. We deal instead with 7-input functions over 3-input majority operators, in particular, we investigate the positive effect of the inverters in reducing the complexity of this class of functions, and the negative impact of some constraints.

1.3.2 XOR-based Logic Synthesis

The methods and abstractions presented are based on the XOR function, denoted in the thesis with the symbol $\oplus$. The achievements and their position with respect to previous works are organized and summarized according to (i) standard logic synthesis, and (ii) logic synthesis for cryptography and security.

- **Standard Logic Synthesis:** This part presents a novel optimization flow based on the Boolean difference. Note that the Boolean difference operation is the XOR of two Boolean functions. Motivated by the novel computing capabilities, and in contrast to the previous sections, this part of the thesis shows more practical aspects of logic synthesis, designed to be successful in modern logic synthesis flows. The flow aims at area optimization addressing standard CMOS technologies (ASICs design). The results are evaluated after place & route, maintaining a reasonable runtime budget for the optimization flow. The results are computed on 36 commercial benchmarks. This section is the result of a 3-months internship at Synopsys Inc., California, US. This achievement is fully described in Chapter 5 and has resulted in the publication [171] and submission [170].

Relation to Previous Works

The interest in Boolean methods is due to the continuous need for improvement in QoR faced by the EDA community (as already stressed in the previous sections). Among all Boolean methods, a central role is played by Boolean resubstitution, which, in practice, is considered the most powerful method in terms of QoR. It takes advantage of don't cares [66] and *permissible functions* [131] to express the function of a node using other nodes already present in the logic network to achieve a more compact implementation. The recent work in [19] has demonstrated novel resubstitution techniques based on truth tables, and their efficient scalability in industrial flows. Our work continues this research study, by presenting a novel Boolean resubstitution method that uses an XOR-based technique together with BDDs to further decrease the area, but without increasing the runtime.

- **Cryptography and Security:** The last part of the thesis continues XOR-based logic synthesis, but investigates a novel application of logic synthesis for crythography and security applications. We introduce a novel complete and automatic synthesis flow which consists of the main transformations involved in logic synthesis, being rewriting, resubstitution, and refactoring. The goal is the minimization of the number of AND gates over *xor-and graph*s (XAGs), which correlates with the degree of vulnerability (security) of a cryptography benchmark. In this scenario, the XOR operation is "for free". This achievement is fully described in Chapter 5 and in the publications [175, 178].

Relations to Previous Works

Logic synthesis for cryptography has recently become subject of growing interest. In this scenario, logic synthesis uses XAGs as data structure to represent and optimize logic. Further, it aims at the minimization of the number of AND gates. Novel optimization techniques are thus needed since state-of-the-art tools [40, 190] automatically address size optimization, without precisely minimizing the number of ANDs. Recently, the works in [33, 60, 149] have started this new domain of application for logic synthesis. Unfortunately, the methods from the cryptography community rely heavily on manual decomposition and optimization strategies [33], while the method in [149] implements few changes to a commercial CMOS-based tool without explicitly designing specific algorithms. Our methods and tool focus on implementing a complete and automatic flow, to address these novel applications. Our algorithms are instead specifically designed to consider a different abstraction based on AND and XOR, and to minimize the number of ANDs, which has been demonstrated beneficial for diverse cryptography applications [64, 100, 183].

1.4 Thesis Organization

This thesis is organized as follows:

- **Chapter 2 – Background:** This chapter introduces the background needed to understand the rest of the thesis. It provides an overview of the most important data structures and algorithms involved in classical logic synthesis. It also considers recent advances in exact synthesis methods.

- **Chapter 3 – Majority-based Logic Synthesis:** In this chapter, we present novel algorithms and techniques that work over MIGs. Specifically, we propose novel Boolean methods for size optimization, demonstrating an 18% average size reduction. Particular focus is also given to the synthesis and design of majority-based logic for emerging technologies, as QCA and STMG. Such emerging technologies use the majority function as their main building block but suffer from diverse technological constraints that need to be considered in the synthesis flow. For this purpose, the chapter presents (i) techniques to limit the fan-out of the majority gates and move inverters on primary inputs; and (ii) exact synthesis methods to limit both depth and fan-out of MIGs. As already pointed out, these constraints are due to the physical limitations of the considered emerging technologies.

- **Chapter 4 – Majority-n Logic:** This chapter continues the study of majority logic, but moves to more theoretical aspects. In particular, the chapter mainly addresses the problem of "how best can the n-argument majority function (majority-n) be realized with a network of 3-input majority gates?". For this purpose, we present a novel method that directly maps BDDs of monotone functions into majority graphs. The proposed methods allow us to obtain the optimum network for the majority-5 and majority-7, and the best-known circuit for the majority-9. Novel general upper bounds and novel decompositions are also presented. The majority function is both self-dual and monotone; thus, the chapter also focuses on the study of self-dual monotone functions and their complexity. As an example, it demonstrates how the inverter can help in reducing the complexity of 7-input functions over majority graphs.

- **Chapter 5 – XOR-based Logic Synthesis:** In this chapter, we move from the majority logic to the XOR-based logic synthesis. We present two different applications: (i) CMOS-based logic synthesis and (ii) logic synthesis addressing cryptography and security applications. For the CMOS-based design, we developed a novel XOR-based resubstitution method for the size optimization of digital circuits. Our flow results in a remarkable area and power reduction after place & route. The latter considers logic synthesis for an alternative application in cryptography. For this purpose, we developed a novel framework to minimize the number od AND gates (called *multiplicative complexity*) of XAGs, which is a metric that directly correlates to the vulnerability of the circuits. The new framework is successfully tested on two sets of cryptography benchmarks.

- **Chapter 6 – Conclusions:** Finally, this chapter concludes the thesis. A summary of research accomplishments is presented, together with future perspectives.

To summarize, this thesis presents novel logic synthesis algorithms for the optimization of standard CMOS-based applications. It takes advantage of modern – more advanced – computing capabilities to push further the optimization results of existing flows. It also introduces technology-dependent logic synthesis as an essential step for the abstraction and manipulation of novel and diverse majority-based emerging technologies. Moreover, it advances state-of-the-art theoretical results on majority logic. Finally, this thesis establishes a novel field of application for logic synthesis, in the cryptography and security field.

2 Background

This thesis targets the development of novel data structures and algorithms within logic synthesis for standard and emerging technologies. This chapter is dedicated to the background and preliminaries needed through the whole thesis and aims at giving a broad overview of this field [174]. First, we introduce the data structures involved in logic synthesis to represent logic. These include *truth tables*, *binary decision diagram*s (BDDs), 2-level and multi-level logic forms. Then, we describe the algorithms for logic function optimization. Both heuristic algebraic and Boolean methods are described, followed by exact methods and synthesis. Finally, we overview methods to partition circuits into smaller sub-units for the application of runtime-expensive methods (e.g., Boolean resubstitution) in a controlled manner.

In the following, we assume that the reader is familiar with the basic concepts of Boolean algebra and Boolean functions; we refer the reader to [44, 69, 97] for further background.

2.1 Data Structures

We present various data structures that are commonly used by logic synthesis algorithms. The subsections are ordered according to the scalability of the data structures, starting from truth tables, which are suitable for functions with a small *support* (i.e., number of variables), to multi-level logic networks, which are the data structure (in various forms and shapes) to represent Boolean functions in almost all modern research and commercial tools.

2.1.1 Truth Tables

A truth table is an explicit representation where the function values are listed for all possible input combinations. Formally, a truth table for a Boolean function $f(x_1,\dots,x_n)$ is a bitstring $b_{2^n-1}b_{2^n-2}\dots b_1b_0$ of 2^n bits, where $f(x_1,\dots,x_n) = b_x$ such that $x = (x_n\dots x_1)_2$ is the integer representation of the input assignment. Consequently, we may also consider a truth table as a number in the half-open interval $[0,2^{2^n})$, for which the truth table representation is the binary expansion of that number.

Example 2.1 *The truth table for a majority-of-three (majority-3) function* $\langle x_1 x_2 x_3 \rangle = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$ *is* 1110 1000*, where* $\wedge$ *and* $\vee$ *are the AND and OR, respectively. Since the binary notation can quickly become very large, it is customary to use a hexadecimal notation, in which each block of 4 bits is represented by the corresponding hexadecimal digit. For the majority-of-three function, the hexadecimal truth table is* `e8`*.* ■

A truth table is a *canonical* (i.e., unique) representation of a function. Consequently, for small functions, truth tables can be used for a simple equivalence check of two functions, if a truth table can be efficiently derived from them.

2.1.2 2-level Representations

Logic functions can be represented in *disjunctive normal form*, also referred to as *sum-of-product* (SOP),

$$f = p_1 \vee p_2 \vee \cdots \vee p_k \tag{2.1}$$

where each

$$p_i = x_1^{q_{i,1}} \wedge x_2^{q_{i,2}} \wedge \cdots \wedge x_n^{q_{i,n}} \tag{2.2}$$

is a *product* of literals with $0 \le q_{i,j} \le 2^R - 1$ for $1 \le i \le k$ and $1 \le j \le n$ and where R is a radix. We have $R = 2$ for binary Boolean logic, $R = 3$ for ternary logic, etc. This represents the so called *positional cube* notation [69] where usually the $q_{i,j}$ are represented in binary form. Therefore, for binary-valued logic the negative and positive literals are $x^1 = x^{\{01\}} = \bar{x}$ and $x^2 = x^{\{10\}} = x$, respectively, $x^3 = x^{\{11\}}$ is a *don't care* term (i.e., both values of a variable are possible) and $x^0 = x^{\{00\}} = \emptyset$ is the empty set (i.e., no value).

Example 2.2 *Let* $f(x_1, x_2, x_3) = x_1 \,?\, x_2 : x_3$*, which is also called the* if-then-else *operator. A disjunctive normal form is* $f = x_1 x_2 \bar{x}_3 \vee x_1 x_2 x_3 \vee \bar{x}_1 x_2 x_3 \vee \bar{x}_1 \bar{x}_2 x_3$*. An alternative, shorter form is* $f = x_1 x_2 \vee \bar{x}_1 x_3$*. In general, one is interested in finding a disjunctive normal form that minimizes the number of product terms* k*.*

■

Many algorithms have been presented to find disjunctive normal forms with some minimality properties (see, e.g., [156]). Also, other 2-level representations have been investigated. Examples are *conjunctive normal form* (CNF), or *product-of-sums* (that interchange '$\vee$' and '$\wedge$' in (2.1) and (2.2)) or *exclusive sum-of-products* (which use '$\oplus$' instead of '$\vee$' in (2.1)). Conjunctive normal forms play a central role in *Boolean satisfiability* solving (see, e.g., [29, 98]) and can be seen as the dual representation of disjunctive normal forms [44]. Exclusive sum-of-product representations find extensive use in cryptography applications (see, e.g., [35, 36, 50]) and quantum computing (see, e.g., [75, 124]).

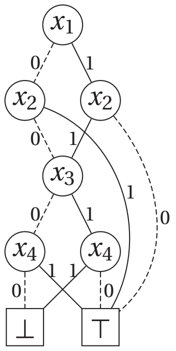


Figure 2.1 – BDD for the function $(x_1 \oplus x_2) \vee (x_3 \oplus x_4)$

2.1.3 Binary Decision Diagrams

Logic functions can be expressed by *decision diagrams* in many ways. The most common representation is the BDD [46, 97] which is a directed acyclic graph where internal nodes are associated with the Shannon expansion of the function, i.e., $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$, where f_{x_i} and $f_{\bar{x}_i}$ are the *cofactors* obtained from f when the variable x_i is assigned 1 or 0, respectively. When referring to BDDs, it is usually implicitly understood that the variables are ordered and the diagram reduced (i.e., BDD refers to ROBDDs [46]). Moreover BDDs are constructed and manipulated so that redundancy is avoided, and thus they are canonical representations of logic functions.

Example 2.3 *Figure 2.1 shows the BDD for the function* $(x_1 \oplus x_2) \vee (x_3 \oplus x_4)$. *Solid and dashed lines represent here positive and negative cofactors respectively. Two terminal nodes labeled '⊥' and '⊤' represent the constant functions* 0 *and* 1. ■

BDDs exploit the fact that for many functions of practical interest, smaller subfunctions occur repeatedly and need to be represented only once. Combined with an efficient recursive algorithm that makes use of caching techniques and hash tables to implement elementary operations, BDDs are a powerful data structure for Boolean function representation and manipulation. Indeed, algorithms for BDD manipulation have polynomial-time complexity (usually quadratic or cubic) in the number of nodes, and such a number grows mildly with the problem size (i.e., variables) in many—but not all—cases, e.g., multipliers are an exception.

The variable order in BDDs affects their size. Improving the variable ordering for BDDs (i.e., minimizing the BDD graph size) is NP-complete [31]. An exact algorithm [71] and many heuristics [76] have been presented that aim at finding a good ordering. It is easy to fit a single BDD node, which contains the variable index and pointers to its two children, into a single 64-bit unsigned integer [97]. Thus, BDDs can represent a good scalable representation for logic functions. They can cope with larger functions as compared to truth tables. When their storage becomes excessive, functions are usually decomposed into blocks forming logic networks.

2.1.4 Multi-level Logic Networks

A multi-level *logic network* is an interconnection of blocks, each implementing a logic function and whose representation style may vary. The interconnection is modeled by a directed acyclic graph where nodes represent primary inputs and outputs, as well as local functions. In most cases such functions are restricted to have a single output, by similarity to CMOS logic gates. For internal nodes, the indegree and outdegree are referred to as *fan-in* and *fan-out* respectively. Note that logic networks can be extended to deal with sequential cyclic circuits [69], but such cases are not considered here.

We use a formal notation for logic networks, that is also referred to as *Boolean chains* in the literature [97]. Given primary inputs $x_1, \dots, x_n$, a logic network consisting of r local functions is a sequence

$$x_i = f_i(x_{i_1}, x_{i_2}, \dots, x_{i_{\mathrm{ar}(f_i)}}) \qquad \text{for } n < i \le n + r, \tag{2.3}$$

where f_i is a gate function with $\mathrm{ar}(f_i)$ inputs and $0 \le i_j < i$ for $1 \le j \le \mathrm{ar}(f_i)$ are indexes to primary inputs or previous gates in the sequence. For convenience, we define $x_0 = 0$. Also, we define a sequence of primary outputs $y_1 = x_{o_1}, \dots, y_m = x_{o_m}$.

Example 2.4 *A full adder with inputs x_1, x_2, x_3 can be realized by the network*

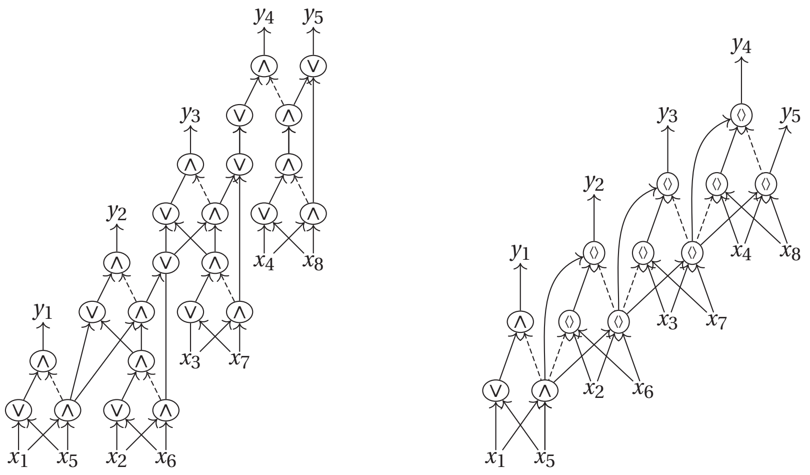
$$x_4 = x_1 \oplus x_2 \oplus x_3, \qquad x_5 = \langle x_1 x_2 x_3 \rangle$$

with outputs $y_1 = x_4$ for the sum and $y_2 = x_5$ for the carry. The network uses the parity function f_4 and the majority function f_5. ■

Logic networks can be specialized by placing restrictions on the internal nodes. A *homogeneous* logic network is one where the fan-in of each internal node is fixed. Restrictions can be applied to local functions as well (e.g., networks consisting of NANDs and/or NORs). For example, *and-inverter graph*s (AIGs), [87, 102] employ AND and Inverters (or equivalently apply AND functions to positive/negative literals). The *majority-inverter graph*s (MIGs), [14] use majority and inverter gates and *xor-majority graph*s (XMGs) [81] use majority and XOR gates. For FPGA design, bounded input look-up tables k-LUT networks are used, where $\mathrm{ar}(f) \le k$.

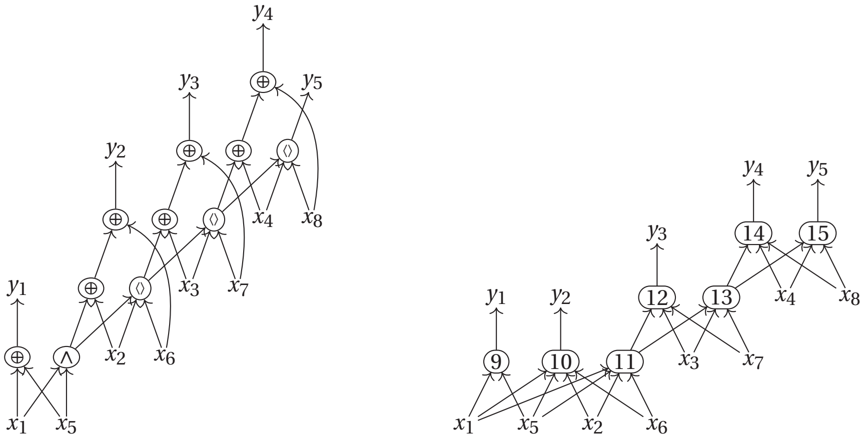
Example 2.5 *Figure 2.2 shows logic networks for a 4-bit full adder, which computes $(x_4 x_3 x_2 x_1)_2 + (x_8 x_7 x_6 x_5)_2 = (y_5 y_4 y_3 y_2 y_1)_2$. Figures 2.2(a), (b), and (c) show an AIG, and MIG, and an XMG, respectively. Inverted inputs are drawn using dashed edges. Figure 2.2(d) shows a 4-LUT network. The gate functions are $f_9 = \texttt{6}$, $f_{10} = \texttt{936c}$,$f_{11} = \texttt{137f}$, $f_{12} = \texttt{69}$, $f_{13} = \texttt{2b}$,$f_{14} = \texttt{69}$,and $f_{15} = \texttt{d4}$.*
■

Combinational logic functions can be represented by many different logic networks. A central task in logic synthesis is to optimize some figure of merit that relates to area, performance, and/or power consumption of the final implementation. Commonly-used cost functions are the *size* r of the logic network, measured in the number of nodes, the *depth* d of the logic



(a) And-inverter graph

(b) Majority-inverter graph



(c) XOR majority graph

(d) 4-LUT network

Figure 2.2 – Different logic networks for a 4-bit adder: (a) AIG, (b) MIG, (c) XMG, (d) LUT

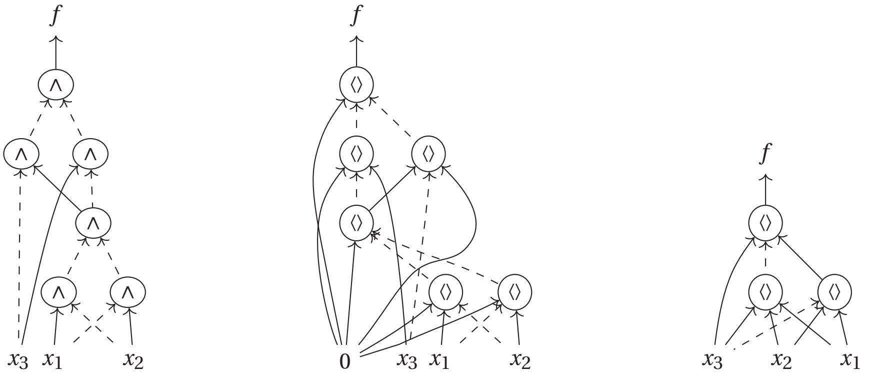
network, which is the longest path from any primary input to any primary output, and the *switching activity*.

In the thesis, we make use of MIGs as main data structure for Chapters 3 and 4. The central function in MIGs is the *majority-of-three* function. The majority function of three Boolean variables x, y, and z, denoted $\langle xyz \rangle$, evaluates to true if and only if at least two of the three inputs are true. The majority function is monotone and self-dual [97] and can be expressed in disjunctive and conjunctive normal form as

$$\langle xyz \rangle = xy \lor xz \lor yz = (x \lor y)(x \lor z)(y \lor z). \tag{2.4}$$

Setting any variable to 0 gives the conjunction of the other two variables, and analogously one obtains the disjunction by setting any variable to 1, i.e.,

$$\langle x0y \rangle = x \land y \quad \text{and} \quad \langle x1y \rangle = x \lor y. \tag{2.5}$$



(a) AOIG (b) MIG obtained from the AOIG (c) Optimized MIG

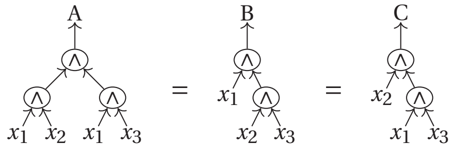
Figure 2.3 – Example (a) of *and-or-inverter graph* (AOIG) representation for $f = x \oplus y \oplus z$, and the MIG derived from its AOIG (b), complement attributes are represented by dashed lines. 3-input nodes are majority operators. Example (c) of optimized (more compact) MIG for f

MIGs are thus universal representation forms and can efficiently represent any Boolean function thanks to the expressiveness of the majority operator. As a consequence of the AND/OR inclusion by MAJ, traditional AOIGs are a special case of MIGs, and MIGs ⊃ AOIGs. It follows that MIGs can be easily derived from AOIGs. In the worst case, AND/OR operators can be replaced node-wise by majority-3 (MAJ-3) operators with a constant input, however, even smaller MIG representations arise when fully exploiting the majority functionality, i.e., with non-constant inputs. An example of MIG representation derived from its optimal AOIG is shown in Figure 2.3(b), while the optimized MIG is depicted in Figure 2.3(c). Since MIGs ⊃ AOIGs, and AOIGs ⊃ AIGs, by transitivity MIGs ⊃ AIGs.

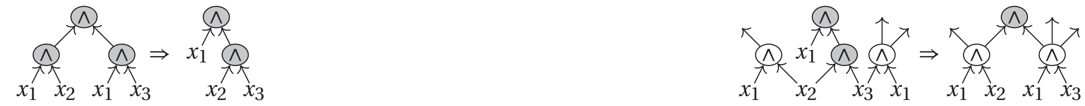
Note that the majority function can be generalized for an odd number of n variables to $\langle x_1 \dots x_n \rangle = [x_1 + \cdots + x_n > \frac{n}{2}]$.

2.2 Algorithms

We present here the underlying techniques for logic optimization algorithms. In particular, we concentrate on multi-level networks as they will be used in the rest of the thesis, and we present both algebraic and Boolean methods. We conclude with an overview on exact synthesis. We preserve here the historical names of approaches for logic network optimization, namely (i) *algebraic methods* (based on polynomial algebra) and *algebraic rewriting* (based on algebraic axioms, possibly of Boolean algebra) and (ii) *Boolean methods* (based on Boolean algebra). Heuristics are employed in these approaches to select the type and sequence of transformations.



(a) Functionally equivalent AIG structures



(b) Rewrite structure A into B

(c) Rewrite structure B into A

Figure 2.4 – Example of AIG rewriting from [121]

2.2.1 Algebraic Methods

Traditional algebraic methods represent each logic network node in SOP form and treat them as polynomials [39, 41]. This simplifying abstraction enables fast manipulation of very large logic networks. Algorithms are designed as operators that iterate one type of transformation until the logic network reaches a local minimum (w.r.t. the transformation itself). Examples of transformations are *extraction*, *substitution*, *decomposition*, and *algebraic rewriting* [39, 69]. We refer the reader to [39, 41, 69] for the details of the first three methods, that will not be discussed here; while we focus instead on algebraic rewriting as it will be important for future chapters of the thesis.

Algebraic Rewriting

The purpose of algebraic rewriting is to change and rewrite portions of a logic network in order to improve, for instance, the number of nodes and/or levels [121]. The general idea consists of applying transformation rules (based on algebraic axioms) with the objectives of improving some figure of merit. Rewriting is more effective when logic networks are homogeneous (e.g., AIGs, MIGs, and XMGs), because logic transformations can be made specific.

Algebraic rewriting has been used extensively in ABC [121]. For example, a database of pre-computed circuit structures for a function can be included, and for any subcircuit one can compute its function and check whether replacing the subcircuit by a pre-computed structure leads to an improvement. If other nodes in the circuit are reused in the rewriting, it may be beneficial to replace a smaller structure by a larger one.

Example 2.6 *An example for AIG rewriting as implemented in ABC is shown in Figure 2.4. Figure 2.4(a) shows three functionally equivalent AIGs structures. These equivalences are employed in Figure 2.4(b) and (c) to reshape the structure of AIGs into functionally equivalent ones.* ■

Refactoring is a variant of rewriting, in which large cones of logic feeding a node are iteratively selected with the aim to replace them by a factored form of the function. It resynthesizes large subnetworks in a logic network from scratch and without using existing nodes in the logic network. The change is accepted if there is an improvement in the selected cost metric (usually number of nodes) [119, 121].

Algebraic rewriting is very effective for MIGs. The related majority algebra and axiomatic system Ω have been described in [14], where it is shown that Ω is sound and complete, providing reachability in the solution space. This means that for MIGs there exists a sequence of steps – possibly with polynomial bound – leading to the optimum solution. Such a path may not exist in other representation frameworks. Indeed, experimental evidence has shown that the MIGhty program [14] implementing algebraic rewriting has outperformed other tools on several benchmarks [14].

Regarding algebraic optimization for MIGs, a dedicated Boolean algebra has been introduced in [14, 16]. The MIG Boolean algebra is defined as the axiomatic system $(\mathbb{B}, \langle\rangle, \bar{\ }, 0, 1)$, where $\langle\rangle$ is the majority operator and $\bar{\ }$ is the complementation. An axiomatic system for the MIG Boolean algebra, referred to as Ω, is introduced and defined over five primitive transformation rules. Some other additional rules can be derived from Ω; three of them, referred to as Ψ, are presented in [16]:

$$\boldsymbol{\Omega}\begin{cases}\textbf{Commutativity} - \boldsymbol{\Omega.C}\\ \langle xyz\rangle = \langle yxz\rangle = \langle zyx\rangle\\ \textbf{Majority} - \boldsymbol{\Omega.M}\\ \langle xxy\rangle = x \qquad \langle x\bar{x}y\rangle = y\\ \textbf{Associativity} - \boldsymbol{\Omega.A}\\ \langle xu\langle yuz\rangle\rangle = \langle zu\langle yux\rangle\rangle\\ \textbf{Distributivity} - \boldsymbol{\Omega.D}\\ \langle xy\langle uvz\rangle\rangle = \langle\langle xyu\rangle\langle xyv\rangle z\rangle\\ \textbf{Inverter Propagation} - \boldsymbol{\Omega.I}\\ \overline{\langle xyz\rangle} = \langle \bar{x}\bar{y}\bar{z}\rangle\end{cases} \tag{2.6}$$

$$\boldsymbol{\Psi}\begin{cases}\textbf{Relevance} - \boldsymbol{\Psi.R}\\ \langle xyz\rangle = \langle xyz_{x/\bar{y}}\rangle\\ \textbf{Complementary Associativity} - \boldsymbol{\Psi.C}\\ \langle xu\langle y\bar{u}z\rangle\rangle = \langle xu\langle yxz\rangle\rangle\\ \textbf{Substitution} - \boldsymbol{\Psi.S}\\ \langle xyz\rangle = \langle v\langle \bar{v}\langle_{v/u}xyz\rangle u\rangle\langle \bar{v}\langle_{v/\bar{u}}xyz\rangle \bar{u}\rangle\rangle\end{cases} \tag{2.7}$$

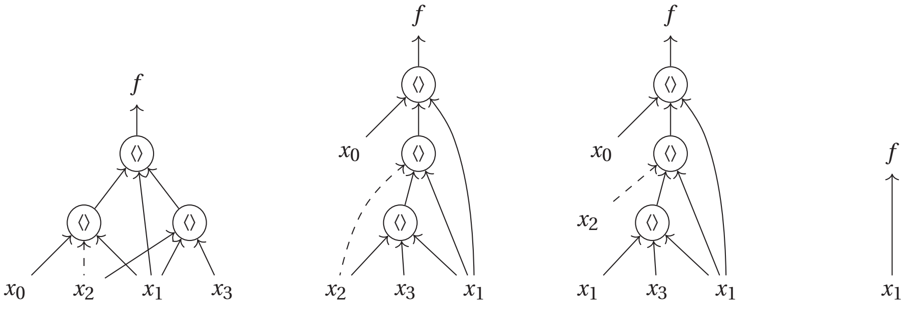
where the symbol $z_{x/y}$ is a replacement operation; that means, it replaces x with y in all its appearances in z. Some of these axioms are inspired by median algebra and others from the properties of the median operator in a distributive lattice. We like to emphasize these identities of the majority function, as they are essential for the forthcoming chapters. First, all three arguments to the majority function are *commutative*, i.e.,

$$\langle xyz\rangle = \langle yxz\rangle = \langle zxy\rangle. \tag{2.8}$$

Also, the majority function evaluates to a single argument if two arguments are equal or complement to each other, i.e.,

$$\langle xxy\rangle = x \qquad \langle x\bar{x}y\rangle = y. \tag{2.9}$$

The *associativity rule* on the majority function allows us to exchange variables if two operations



(a) Original MIG (b) Associativity (c) Relevance (d) Final MIG

Figure 2.5 – Example of MIG optimization using the algebraic rewriting. (a) is the initial MIG; (b) MIG after associativity rule; (c) relevance transformation; (d) final result after rule $\Omega.M$

are nested and share a common variable, i.e.,

$$\langle xu\langle yuz\rangle\rangle = \langle\langle xuy\rangle uz\rangle. \tag{2.10}$$

The associativity rule becomes obvious, when replacing u by some operator symbol '$\circ$' and the angular brackets by parentheses: $(x \circ (y \circ z)) = ((x \circ y) \circ z)$; as pointed out by Schensted [158]. An alternative way to think about its validity, is by setting u to 0 and 1, and noting that $\wedge$ and $\vee$ are associative [97]. Due to these properties, the set $M = \mathbb{B} = \{0, 1\}$ and the ternary operation $\langle xyz\rangle$ defined according to (2.4) are a *median algebra* [30]. A median algebra is defined as a set and a majority operator satisfying commutativity, associativity, and the first identity in (2.9).

Also a *distributivity rule* can be derived from these three rules [160]:

$$\langle xu\langle yvz\rangle\rangle = \langle\langle xuy\rangle v\langle xuz\rangle\rangle \tag{2.11}$$

To easily memorize this rule, it's handy to replace u by a symbol '$\circ$' and v by a symbol '$\times$'.

Note that in the Boolean case, in which each element has its complement, the median algebra is a complemented distributive lattice [30] and therefore a Boolean algebra (see, e.g., [156]). Since the majority function is self-dual, inverters can be *propagated* from the inputs to the outputs, i.e.,

$$\langle \bar{x}\bar{y}\bar{z}\rangle = \overline{\langle xyz\rangle}. \tag{2.12}$$

An MIG can be transformed into another MIG by just using the rules in Ω in either direction as well as additional rules. Such rules can reduce the number of nodes and depth of a logic network, or any other metric [16]. This result guarantees that the best MIG, for a given target metric, can always be reached. The general idea for logic optimization consists of applying rules and axioms to rewrite the MIG and to obtain a desired configuration. Different metrics can be optimized; here, we report the example for size optimization.

Example 2.7 *The MIG algebraic size optimization can be done by applying a sequence of the five axioms and rules both from* Left to Right *($L \rightarrow R$) and from* Right to Left *($R \rightarrow L$). For the MIG in Figure 2.5(a) the associativity rule $\Omega.A$ is used to rewrite the graph from (a) to (b), and $\Psi.R$ rewrites from (b) to (c). Figure 2.5(c) can be optimized by applying the majority rule $\Omega.M_{L\rightarrow R}$. The size of the graph is reduced from* 3 *nodes to* 0*, as shown in Figure 2.5(d).* ■

These axioms and rules have been employed in [16] to achieve significant depth optimization results over both unmapped and LUT-mapped results.

2.2.2 Boolean Methods

Boolean methods consider the true nature of logic functions by considering Boolean identities and don't cares [24, 69]. Don't care conditions relate to the embedding of a Boolean function into the environment, and are usually called *external don't cares.* They consist of both *controllability* and *observability* don't cares. The former is defined as those input patterns that are never produced by the environment, while the latter considers situations when a given output is not observed by the environment. The idea behind Boolean methods is to use the power of Boolean algebra together with the degree of freedom provided by the don't cares to construct local transformations to improve logic networks [38, 39, 69, 126]. For example, due to observability don't cares, the function at a node v may be changed to another function without changing the behavior at the primary outputs. In the transduction method proposed by Muroga [130, 131], this new function is called a *permissible function* for node v, and the set of *all* permissible functions for a node v is its *maximum set of permissible functions* (MSPF). Consequently to the use of don't cares and Boolean identities, Boolean methods usually achieve better results, but come at higher computational cost and less scalability [38].

Boolean methods evolved through time as different engines became available for detecting the existence of permissible functions. The MIS/SIS program [41] used program ESPRESSO to find permissible functions; other tools used BDDs to check if a function is a permissible replacement of another by checking the tautology of their equivalence. Fast tautology check can be provided by BDD tools [191] and thus desirable permissible replacements of a local function can be quickly evaluated. In general, Boolean methods can also be enabled by casting the search for permissible functions as a *satisfiability* (SAT) problem, and using an effective SAT solver for this task (see [118] for more details). As SAT-solvers are out of the scope of this thesis, they are not detailed in this chapter; more details on SAT-based exact synthesis can be found in Section 3.5 and in [82, 98].

Overall, Boolean methods leverage a variety of transformations that eventually resort to an engine for verifying their applicability. Examples of engines are two-level minimizers, BDD, and SAT packages. Recently, truth tables have also been used as data structure to check for permissible functions [19]. Here, we explain in detail the scenarios – as presented in [170] – in which truth tables and BDDs produce, in practice, the best results in driving Boolean methods.

Only truth tables and BDDs are considered as they will be used in the next chapters to detect don't cares and permissible functions. Afterwards, we review some Boolean transformations, while, in Section 2.3, we present various techniques to partition large logic networks into smaller units, used to apply expensive Boolean methods in an efficient way.

Truth Tables or BDDs?

Boolean methods rely on complete functional properties of a logic circuit, preferably including don't care information. In order to gather such functional properties, truth tables or expensive logic reasoning engines are required, such BDDs and SAT. The choice of the engine determines the scalability of the Boolean methods. In the last two decades, improvements in SAT solving made SAT-based methods sensibly more scalable than those based on BDDs. As a consequence, some Boolean methods based on BDDs or truth tables grew outdated. On the other hand, it appears in practice that there are still several synthesis scenarios in which BDDs or truth tables are preferable to SAT, in terms of QoR and/or runtime.

This last part of the section discusses how to automatically identify scenarios, based on circuit characteristics and optimization scope, where Boolean methods are best driven by either truth tables or BDDs.

Truth Tables: Truth tables are efficiently stored in computers as a concatenation of words. Boolean functions with n variables require 2^{n-k} words, where $k = \log_2(\text{word-size})$. It follows each 64-bit (32-bit) word can store a 6(5)-input truth table. For circuits or sub-circuits having fewer than 16 inputs, truth tables are remarkably fast to compute in practice, as they have low memory footprint and no formulation overhead. Furthermore, truth table computation may be parallelized w.r.t. words and distributed over different threads. For example, 64-bit words operating with a 16-input truth table require bit-level operations among 1024 (independent) words. Distributing such computation over 16 threads, which is common in EDA applications, reduces the latency bottleneck to just 64 consecutive bit-level word operations.

As of today, functional properties of circuits up to 16 inputs are most efficiently computed via truth tables. The overhead of formulating and solving a SAT problem, or handling a BDD manager for the same circuit usually takes considerable amount of runtime.

Example 2.8 *Consider an XOR-rich parity circuit over* 16 *variables, with many functionally identical nodes originating from partial SOP collapsing during synthesis. Depending on the depth of XOR collapsing, the circuit size can grow over several thousands of nodes. In our case, we deal with about* $10k$ *AIG nodes. Assume the goal is to merge all functionally equivalent nodes, up to complementation, in this circuit. If the task is performed using truth tables, it takes about* 1 *second of runtime. When using a SAT-based formulation of the problem, instead, it takes more than* 2 *minutes to obtain the same result. BDD-based methods take tens of seconds, ranging between* 15 *seconds and* 30 *seconds, depending on the settings for static and dynamic variable re-ordering.* ■

BDDs BDDs are a compact canonical representation form. Compared to truth tables, which are also canonical but always exponential-sized, BDDs allow polynomial-size for many functions and variable orderings of practical interest [47]. However, other functions, such as multiplication and *hidden-weighted bit* [1], have exponential-size BDDs for any variable order [47].

When dealing with a medium-large function, whose exact properties are unknown, BDDs construction time can differ sensibly. Empirically, BDDs are always constructed rapidly, compared to truth tables, for the following circuit cases:

1. Circuits with fewer than 20 primary inputs,
2. Circuits that, if decomposed into an AIG, have depth $d < 20$ levels,
3. Circuits with a small (usually smaller than 2) internal nodes over primary inputs ratio,
4. Circuits with special properties that facilitate BDD construction, e.g., symmetries.

For the cases above, where the primary inputs are fewer than 16 and truth tables are not desirable, BDDs are the fastest alternative in the majority of cases. It is worth noting that corner cases for points 3) and 4) exist, i.e., ripple carry adders whose BDDs are built with bad variable order, or symmetric functions with many variables, etc. However, these cases represent a small fraction of the ones encountered in practice and can be detected with some extra filtering.

Example 2.9 *Consider Boolean resubstitution over the MCNC circuit* $k2$ *[192]. Note that this circuit shows* 20 *levels,* 2580 *nodes, and 45/45 inputs/outputs*[1]*. Building a BDD for this circuit takes less than* $10\ ms$ *with a modern BDD package. Using SAT for each resubstitution move, spanning the whole circuit, would result in notable runtime overhead.* ■

Resubstitution

Boolean resubstitution (also called substitution [69]) aims at expressing the function of a node v using other nodes (called *divisors*) already present in the logic network. A transformation is accepted if the new implementation is more compact than the current node implementation, thus leading to size optimization. A k-resubstitution is a generalization of resubstitution, which adds exactly k new nodes and removes l nodes, where l is the number of nodes in the *maximum fan-out free cone* (MFFC) [119, 121] of v. In this case, size improvement is achieved if $l > k$. In this last scenario, resubstitution adds k new logic operators to the existing

---

[1] After decomposing it into an AIG

logic network. Note that resubstitution techniques are thus usually classified according to the number k of logic operators additionally added, i.e., 0-resubstitution does not add any new operator; 1-resubstitution expresses a logic function by adding one logic operator, and so forth. According to the type of nodes added in the logic network by resubstitution, we also refer to resubstitution as AND-resubstitution, OR-resubstitution, XOR-resubstitution, AND-OR resubstitution, etc.

Due to the use of don't cares, Boolean resubstitution finds more optimization opportunities as compared to algebraic substitution, but it is inherently more expensive [38]. Consider the following example, which shows the use of don't cares for Boolean resubstitution.

Example 2.10 *Consider the logic network [69] given by*

$$\begin{aligned} f &= x_1 \vee (x_2 \wedge x_3 \wedge x_4) \vee x_5 \\ g &= x_1 \vee (x_3 \wedge x_4) \end{aligned} \tag{2.13}$$

where $\wedge$ and $\vee$ represent the AND and OR operators, respectively. We can minimize f using g:

$$\begin{aligned} f &= x_1 \vee (x_2 \wedge g) \vee x_5 \\ g &= x_1 \vee (x_3 \wedge x_4) \end{aligned} \tag{2.14}$$

where $x_2 \wedge x_3 \wedge x_4$ can be changed into $x_2 \wedge g$ because the minterms where $x_2 \wedge x_3 \wedge x_4$ and $x_2 \wedge g$ differ are in the don't care set. ■

Different representation of don't cares and varying reasoning engines have been used in the past years to develop novel and powerful resubstitution methods. For example, in the transduction method proposed by Muroga [130, 131], resubstitution methods were applied by computing permissible functions using truth tables. Unfortunately, that tabular description was not efficient enough to be applicable to logic networks of reasonable size at that time, while modern resubstitution flows as the ones in [19] and in [151] efficiently use truth tables as reasoning engine. Permissible functions can also be quickly evaluated using BDDs. The method in [157] uses BDDs and permissible functions to build fast resubstitution techniques. More recently, Miyasaka et al. [128] have presented a method that uses a BDD-package without variable re-ordering to accelerate the computation of permissible functions. Concerning SAT-based resubstitution methods, the works in [117, 120] consider SAT-based don't cares computation aiming at resubstitution frameworks.

Although Boolean resubstitution achieves better results and is more precise than the algebraic one, it is more expensive at runtime. The right engine selection may improve its scalability, but its application is still limited to small functions. Partitioning and breaking logic networks into smaller subnetworks is thus needed to be able to efficiently compute each node's functionality and to apply Boolean resubstitution. Even when resubstitution is applied to small- (~ 15 inputs) and medium- (~ 20 inputs) size windows of logic, k-resubstitution remains intrinsically expensive due to the high amount of required equivalence checking, which are the primitive operations in all resubstitution algorithms. Consider as an example 1-resubstitution using the 2-input AND gates, applied on a small window of logic with V internal nodes. The goal is to try to express each internal node in the window as the AND of two other nodes in the window. In the worst case $O(V^2)$ equivalence checks are needed for each node, resulting in $O(V^3)$ checks for the whole window. In this scenario, Boolean filtering techniques are usually employed to strongly reduce the number of candidates for resubstitution, and at the same time without losing significant optimization opportunities. Common filtering techniques [19] include, but are not limited to, (i) structural filtering, or (ii) setting a maximum number m of candidates to be tried. Structural filtering comprises, for instance, skipping candidates in the *transitive fan-out* (TFO) cone of the current node, or skipping nodes whose level is too far away, etc. The best improvement in runtime is although given by setting a maximum number of candidates m, as the total number of equivalence checks decreases to $O(mV^2)$.

Rewriting

Rewriting aims at minimizing the size (or other metrics) of a logic network by iteratively selecting subnetworks and by replacing them with smaller (or better) precomputed subgraphs, while preserving the functionality. This is achieved by applying a Boolean equivalence check (modulo the don't cares).

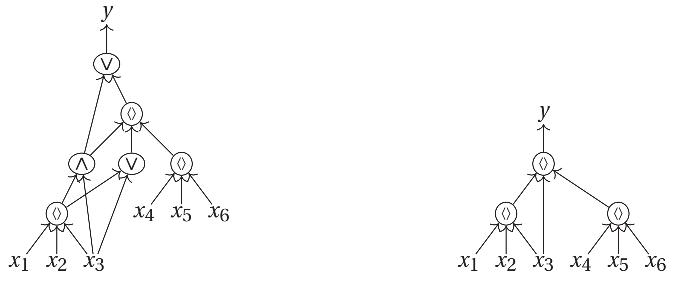
Example 2.11 *Examples of typical precomputed subnetworks are all 4 variables functions, or the 222 NPN equivalence classes [81, 121, 162]. Here, the idea is to replace 4-input subnetworks with their optimum precomputed representation. A solution can be size- or depth-optimum implementation depending on the target metric.* ■

Redundancy Removal and Rewiring

Redundancy removal is a common technique that uses *automatic test pattern generator* to detect untestable stuck-at faults in a logic network and modifies the network at the faulty net by setting it to a constant value [43, 67]. *Rewiring* improves on redundancy removal because it adds new connections in a logic network to create redundancies, that later can be removed. In practice, it adds and removes nets and it aims at removing nets related to long wires [51].

In a more general scenario, a logic network is optimized by changing a local function (to improve overall size and/or depth) by introducing errors that are then corrected by changing the local functionality somewhere else [66]. The following example shows this type of transformation for MIGs.

Example 2.12 *We show an example that makes use of an induced error correction technique*


y
∨
⟨⟩
∧
∨
⟨⟩
⟨⟩
x_1 x_2 x_3 x_4 x_5 x_6
y
⟨⟩
⟨⟩
⟨⟩
x_1 x_2 x_3 x_4 x_5 x_6

(a) Initial MIG (b) Optimized MIG

Figure 2.6 – Rewiring based on induced error correction in MIGs

for MIGs, which was first explained in [15]. The technique is based on the property that

$$y = \langle y_1 y_2 y_3 \rangle \text{ if, and only if } (y \oplus y_i)(y \oplus y_j) = 0 \text{ for all } 1 \le i < j \le 3 \quad (2.15)$$

We can think of each $y_i,\ i = 1,2,3$ *as a convenient (i.e., reduced and thus incorrect) versions of* y*. The difference between* y_i *to* y *is expressed by* $(y \oplus y_i)$ *is the local error. The condition on the right-hand side of* (2.15) *states that all three errors must be pairwise orthogonal, i.e., the pairwise differences have an empty intersection. In this condition, the majority operator restores the correct functionality. Figure 2.6(a) shows an MIG for the function* y*, which has the truth table* `f8f8f8e0f8e0e0e0`*. One can easily verify that the right-hand side condition in* (2.15) *is satisfied for* $y_1 = \langle x_1 x_2 x_3 \rangle$, $y_2 = x_3$*, and* $y_3 = \langle x_4 x_5 x_6 \rangle$*, and therefore,* $y = \langle \langle x_1 x_2 x_3 \rangle x_3 \langle x_4 x_5 x_6 \rangle \rangle$*, for which an MIG is shown in Figure 2.6(b). The optimized MIG reduces both size and depth to half of their original values. More details on this technique including methods to derive valid fault candidates are described in [15].*

■

2.2.3 Exact Methods

After the discussion on heuristic methods, we give here an overview of exact synthesis. Exact synthesis is the problem of finding the optimum logic representation for a given Boolean function with respect to some cost criterion. Usually the cost is either the number of gates (or equivalently nodes and correlated to area) or the depth of the logic network (or equivalently the critical path and correlated to delay). For example, specific instances of the problem are finding the SOP representation using the smallest number of implicants, or a 2-input gate level logic network with the fewest number of gates. For instance, a well-known exact algorithm is *FlowMap* that determines a minimum-depth mapping of a logic network into k-LUTs in polynomial time [62]. Note that an optimum circuit implementation is not necessarily unique. For example, the majority-5 function can be realized with the minimum number of majority-3 gates in more than one way, e.g.:

$$\langle\langle x_3 x_4 x_5\rangle x_2 \langle x_1 x_2 \langle x_3 x_4 x_5\rangle\rangle\rangle$$

and

$$\langle x_1 x_2 \langle\langle x_3 x_4 x_5\rangle x_2 \langle x_3 x_4 x_5\rangle\rangle\rangle$$

For theoretical purposes, one often considers the whole set of Boolean functions for a fixed number of variables n. This allows to derive lower bounds on the complexity of logic representations. For example, all 4-variable Boolean functions can be represented using SOPs with at most 8 implicants [156]. All 5-variable Boolean functions can be represented using 2-input logic networks with at most 12 gates [97]. Since the number of Boolean functions grows double-exponentially with the support size, it is hard to compute bounds for larger number of variables.

In this thesis, we are concerned with SAT-based exact synthesis. The first example of SAT-based exact synthesis can be found in [74], and successive analyses and improvements have been considered in [98, 99]. Given a Boolean formula $f(x_1,\dots,x_n)$, a Boolean satisfiability problem (or SAT problem) asks whether there exist an assignment to the variables $x_1,\dots,x_n$ such that f evaluates to true. If this is the case, such an assignment is called a *satisfying assignment* and f is called *satisfiable* (SAT). Otherwise, f is *unsatisfiable* (UNSAT). SAT solvers are software programs that receive a Boolean formula f, represented in CNF, and return a satisfying assignment, if and only if f is satisfiable. The key idea behind SAT-based exact synthesis is to verify whether it is possible to realize a function f with a logic network of size r, using a sequence of SAT formulas, i.e., encoding the problem in CNF. The size r is at first initialize at 0, or at some given value, and at each loop it is increased if a solution is not found, i.e., the result is UNSAT. If the results is SAT, an optimum size logic network can be extracted from the obtained solution. It is easy to understand that this formulation can be used to evaluate lower bounds on the complexity of logic representations. It is necessary to demonstrate that a given function cannot be realized (UNSAT) with r gates.

We refer the interested reader to [82, 164] for a more detailed review on SAT-based exact synthesis. Note also that exact synthesis finds application in many and diverse logic synthesis algorithms. For instance, logic rewriting algorithms optimize logic networks by replacing small subnetworks with optimimum logic network obtained with exact synthesis as in [81, 161].

2.3 Efficient Circuit Partitioning

Boolean methods achieve better results in average than algebraic methods, but they have worse scalability. The right data structure selection (as proposed in Section 2.2.2) determines the scalability of the Boolean methods, but their application is still limited to small functions. Partitioning and breaking down the logic network into smaller subnetworks is thus needed

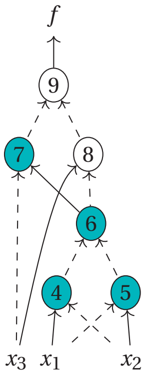


Figure 2.7 – Example of 3-cut (highlighted in blue) over AIG

to apply Boolean methods in an efficient way and on large networks. We discuss here some approaches for circuits partitioning that are usually involved in logic syntehsis flows and that will also be used in the rest of the thesis.

2.3.1 Small Scale: Cut Enumeration

Cut enumeration identifies subsets of k inputs functions inside a larger network; usually, k is in the order of 6 or 7 [122], while it is not practical for larger values. A *cut* c of a node v in the logic network is a set of nodes, called *leaves*, such that:

- every path from node v to a terminal node visits at least one leaf,
- each leaf is contained in at least one path.

Node v is called the *root* of the cut and each cut represents a subgraph that includes the root v, the leaves, and some internal nodes. A cut is k-feasible (called k-cuts), if it has a number of leaves $\leq k$.

Example 2.13 *Figure 2.7 shows an example of 3-cut over an AIG (all nodes are 2-input AND gates). The root node is node* 7*, while internal nodes are* 4, 5*, and* 6*. Leaves are* primary inputs *(PIs)* x_1, x_2, *and* x_3. ■

We summarize here two different algorithms for cut computation.

Bottom-up:

All k-feasible cuts can be generated using the recursive algorithm proposed in [63, 138]. This approach finds all k-feasible non trivial cuts of node v, denoted by $\text{cuts}_k(v)$, by *merging* the cuts of its children:

$$cuts_k(v) = cuts_k(v_1) \otimes_k cuts_k(v_2) \otimes_k \cdots \otimes_k cuts_k(v_u) \tag{2.16}$$

where $v_1, v_2, \dots, v_u$ are the u children of node v in the logic network. The operation

$$M_1 \otimes_k M_2 = \{m_1 \cup m_2 \mid m_1 \in M_1, m_2 \in M_2, \mid m_1 \cup m_2 \mid \leq k\} \tag{2.17}$$

is a saturating union over all combinations of subsets. In this scenario, each primary input has its trivial cut only and we can enumerate the k-cuts of all nodes using the recursive algorithm in a depth-first manner. Further details on the merge functions and this cut enumeration can be found in [63, 138].

Top-Down:

Another approach for cut enumeration is presented in [122]. This algorithm was proposed in 2007 to overcome runtime and memory issues of classical cut enumeration engines. It proceeds in a top-down manner, from primary outputs down to primary inputs. It makes use of cuts *factorization* using *local* cuts and *global* cuts. These are collectively known as *factor cuts*. For each node v, the method is based on the *expansion* of a factor cut with *local* cuts to obtain a larger set of cuts. A factor cut of the node v is given by:

$$cuts_k(v) = \bigcup_i c_i \tag{2.18}$$

whre c is a factor cut for node v, and c_i a local cut of a node $i \in c$. If the new cut is k-feasible, then it is called *one-step expansion* of c. The approach allows the enumeration of all k-feasible cuts, but it can be reduced to partial enumeration for memory saving. All details about local, global cuts and the top-down approach can be found in [122].

2.3.2 Detecting All Reconvergent MFFC

For the sake of completeness, we report here a re-implemented version of the state-of-the-art algorithm for MFFC computation from [121]. This algorithm will be used in the methods and tools proposed in the rest of the thesis. The reconvergent MFFC [121] of a node v is a subnetwork that contains all the logic nodes used *only* by the node v. More formally, it is defined as a subset of the fanin cone of node v such that every path from a node in the subset to the *primary output*s (POs) passes through v itself [122]. Thus, when a node v is substituted or removed, also its MFFC can be removed from the logic network. It follows that optimization methods can change and rewrite the MFFC without affecting the rest of the network. Here, we present our version of the algorithm to detect the *reconvergent* MFFC.

The pseudocode for the procedure on a node v is presented as Algorithm 2.1. The proce-

```
Function Reconvergent_MFFC($v$):
    if $marked(v) \vee (v \in PI)$ then
        return ;
    end
    Mark $v$;
    foreach children $c$ of node $v$ do
        $flag \leftarrow 0$;
        if $marked(c) \vee c \in PI$ then
            continue;
        end
        foreach parent $p$ of node $c$ do
            if $not\_marked(p)$ then
                $flag \leftarrow 1$ ;
                break;
            end
        end
        if $flag = 1$ then
            continue;
        end
        Reconvergent_MFFC($c$);
    end
End Function
```

Algorithm 2.1: Reconvergent MFFC on node v

dure *"marks"* all nodes in the MFFC of node v by recursively applying the algorithm on all its children. The algorithm stops when (i) the node is a PI, (ii) the node is already marked (i.e., already been visited once), (iii) the node has a fan-out which is not marked (i.e., a path that does not pass through v). At the end of the algorithm, all nodes in the MFFC are marked.

2.3.3 Windowing

In this section, we report a re-implementation of the *windowing* procedure presented in [119]. Windowing is an approach to limit the scope of an optimization procedure to a small fraction of a logic network that allows in many cases to drastically improve the scalability. The pseudocode of the windowing procedure applied over an MIG (used in Chapter 3) is shown as Algorithm 2.2. The procedure takes as input an MIG M and two positive integers l and s, where l denotes the maximal number of primary inputs (cut-size limit) of the window and s denotes the maximal number of nodes (node limit) of the window. As result, the procedure returns the MIG M optimized for size.

In a loop, the procedure iterates over all nodes p of M in topological order and generates for each of the nodes a *reconvergence-driven* cut C starting from p with at most l nodes (see [119] for a detailed description of the cut computation). The cut C serves as the input boundary of the window W. Starting from the nodes in C, the window W is iteratively extended by merging parent nodes if all their children are already in W. The procedure terminates if no

```
Input: MIG $M$, cut-size limit $l$, node limit $s$
Output: Optimized MIG $M$
foreach node $p$ in $M$ in topological order do
    $C$ = ComputeCut($M, \{p\}, l$);
    $W$ = ExpandToWindow($M, C, s$);
    $\hat{W}$ = OptimizationProcedure($W, p$);
    $M = M[W \leftarrow \hat{W}]$;
```

Algorithm 2.2: Windowed MIG Optimization

new parents can be merged or the number of window nodes exceeds s. The obtained window W is then locally optimized to $\hat{W}$ using an optimization procedure. Finally, the window W in M is replaced by the optimized window $\hat{W}$. It is worth mentioning that a similar procedure can be applied to diverse logic networks.

2.4 Summary

In this chapter, we presented state-of-the-art data structures and algorithms involved in logic synthesis flows. In the following discussion, truth tables, BDDs, and multi-level logic networks will be used as a data structure for optimization. We described both algebraic and Boolean heuristic methods and exact synthesis. All these algorithms will be involved, changed, and adapted for our diverse goals and optimization metrics. In the remainder of the thesis, we will mainly present new and revisited Boolean methods. All the partitioning methods discussed above together with a proper engine selection (truth tables or BDDs) allow the application of expensive Boolean methods in a controlled manner.

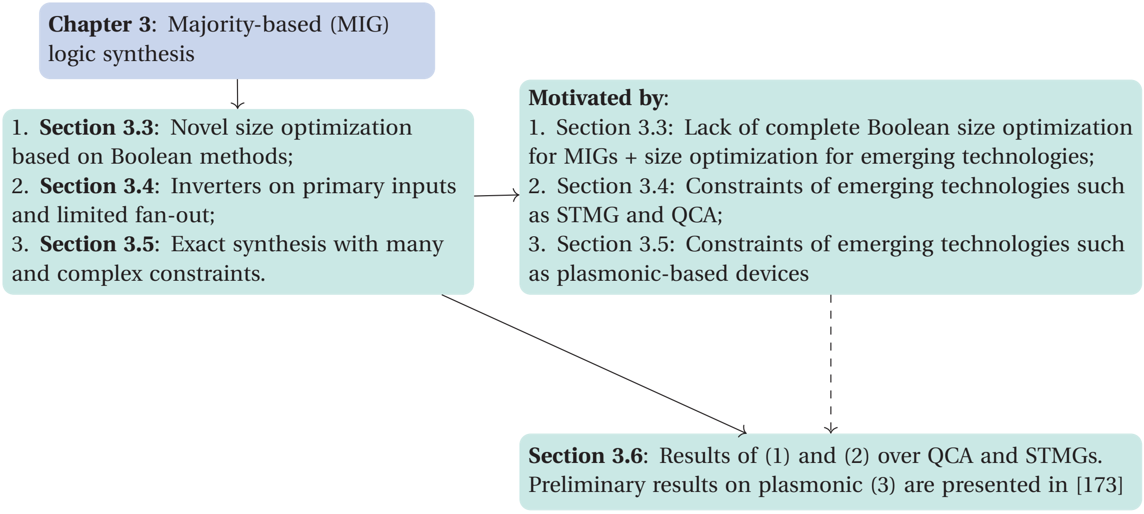
3 Majority-based Logic Synthesis

The main goal of this thesis is the development and implementation of novel logic synthesis algorithms concentrating on both standard and emerging technologies. After a brief overview of state-of-the-art synthesis methods and background in Chapter 2, we transition to the core part of the research work. In particular, this chapter focuses on novel logic synthesis techniques for majority-based logic, that work over *majority-inverter graph*s (MIGs). These techniques include: (i) novel Boolean methods for size optimization; (ii) methods to obtain inversion free MIGs with limited fan-out; and (iii) exact synthesis algorithms to deal with many and complex constraints. All methods address novel synthesis and optimization methods for emerging technologies-based design.

The remainder of this chapter is organized as follows (see also Figure 3.1). First, the motivations for this chapter are presented in Section 3.1, then, Section 3.2 introduces state-of-the-art majority-based nanotechnologies. In Section 3.3, novel Boolean methods to minimize the size of MIGs are introduced. These methods use Boolean techniques and truth tables to minimize the size of the networks, without increasing their depth. Results show that the presented methods can reduce the size of MIGs up to 18%, resulting also in depth optimization of 10.22%. This section is largely based on the publication in [151]. Section 3.4 is based instead on the publication presented in [181] and addresses novel techniques to manipulate the number of inverters and limit the fan-out of MIGs. Section 3.5 illustrates an exact method to synthesize MIGs constrained to have limited depth and limited fan-out. This last section is based on the work presented in [180]. The results of these techniques on emerging technologies such as *quantum-dot cellular automata* (QCA) and *spin torque majority gate* (STMG) are presented in Section 3.6. Note that preliminary results on plasmonic-logic are instead illustrated in [173]. Finally, this chapter is concluded and summarized in Section 3.7.

3.1 Motivation

To overcome the intrinsic scaling limitations of CMOS, emerging technologies are going to play a key role in the near future [137]. Many of today's nano-emerging technologies, including *spin-*



Chapter 3: Majority-based (MIG) logic synthesis

1. **Section 3.3**: Novel size optimization based on Boolean methods;
2. **Section 3.4**: Inverters on primary inputs and limited fan-out;
3. **Section 3.5**: Exact synthesis with many and complex constraints.

Motivated by:
1. Section 3.3: Lack of complete Boolean size optimization for MIGs + size optimization for emerging technologies;
2. Section 3.4: Constraints of emerging technologies such as STMG and QCA;
3. Section 3.5: Constraints of emerging technologies such as plasmonic-based devices

Section 3.6: Results of (1) and (2) over QCA and STMGs. Preliminary results on plasmonic (3) are presented in [173]

Figure 3.1 – Chapter organization

wave devices [95], QCA [106], and STMG[135], are inherently majority-based. As an example, the computation principle of spin-wave devices is based on the interference of propagating spin waves and the information is encoded in the phase of the waves. As a consequence, these technologies offer a particular inexpensive realization of the majority operation. For example, in the QCA technology the area requirements for the majority-of-three operation are more than 2× smaller as compared to the ones for implementing an inverter [182].

The recent progress in such nano-emerging technologies has consequently sparked considerable interest in majority-based logic synthesis optimization techniques and abstractions [18], which are fundamental in order to properly assess these post-CMOS technologies. In contrast to conventional logic synthesis algorithms–being based on logic primitives such as AND or OR–majority-based algorithms employ intermediate data-structures capable of natively representing and manipulating majority operations [18]. Recently, many and diverse works have focused on majority-based logic synthesis techniques – some examples being [14, 59, 144]. In particular, the work in [14] has introduced the MIG, which is a data structure that uses only majority-of-three and inversion in order to represent and optimize Boolean functions. While competitive solutions and significant results for majority-based delay optimization have been presented in [14, 16], as of today, there is still a lack of complete and powerful size optimization tool for MIGs. Furthermore, MIGs methods developed so far are mainly algebraic, while there is lack of a unified Boolean framework for MIGs. In Section 3.3, we introduce novel methods for size-optimization of MIGs; in particular, we focus on novel and scalable Boolean techniques, which serve two purposes: (i) they achieve size reductions when other techniques saturate; and (ii) they help to escape local minima in the logic optimization flow and thus re-enable other size optimizations. We concentrate on node replacement techniques that re-express the global function of an existing majority node using other nodes already present in the logic network. The objective is to reduce the size of the logic representation as much as possible while maintaining the global input-output functionality of the logic network (and preserving the logic network's depth).

Furthermore, the large variety of beyond-CMOS devices leads to a broad range of various technological constraints (e.g., on the fan-out load of each gate) that need to be taken into account by modern *electronic design automation* (EDA) tools. Two main drawbacks apply to several devices. First, since all devices are targeted towards ultra-low energy operation, the inherent amplification or the driving capabilities of these devices are low [137]. This leads to the need for constraining the fan-out characteristics of the implemented circuits. Second, several beyond-CMOS technologies do not offer efficient implementations of inverter [105, 134]. Therefore, it is required to minimize inversions [179] or even to eliminate them from implemented circuits. In addition, some emerging nanotechnologies require more than one constraint to be met at the same time. An example is given by the plasmonic-based devices in [72], for which both depth and fan-out need to be limited to a maximum 3.

In Section 3.4 and 3.5, we focus on majority-based logic synthesis addressing nanotechnologies as the final goal. Logic synthesis approaches should be able to take into consideration both (i) the new logic abstraction (majority-based) and (ii) the different technological constraints in order to be able to give better technology-dependent results. In Section 3.4, we present techniques to (i) eliminate inverter components, by moving them to *primary input*s (PIs); and (ii) constrain the maximum fan-out of each node to Φ (to 3 in our case). These two algorithms work on MIGs and produce networks that can be adapted for majority-based beyond-CMOS technologies, such as QCA and STMG. As a matter of fact, both these technologies suffer from fan-out limitations and have expensive or lack-of inverter cell. In Section 3.5, being faced with nanotechnologies (i.e., plasmonic-based devices) that seek for low-depth majority-based networks with limited fan-out for small functions, we demonstrate how state-of-the-art *exact synthesis* algorithms can be adapted and used to find logic networks that match these constraints. To emphasize the need for exact synthesis, we also demonstrate how conventional logic synthesis either fails to find constraint-satisfying logic networks or yields networks of inferior quality.

3.2 Majority-based Emerging Technologies

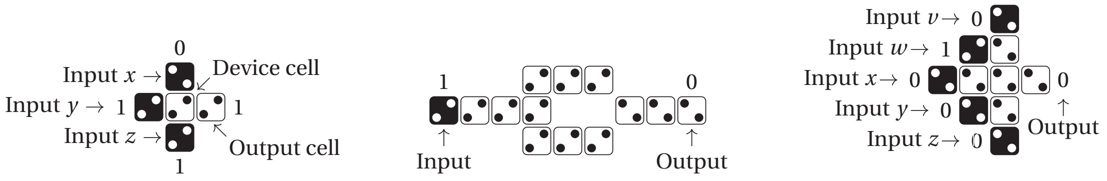
With transistor dimensions reaching their scaling limits, it is interesting to look at disruptive computation paradigms offered by emerging nanotechnologies. This section illustrates examples of state-of-the-art majority-based emerging technologies. Examples of majority-based beyond CMOS technologies include, but are not limited to, QCA [106], superconducting electronic devices (RSFQ [107] and AQFP [169]), nanomagnet logic [65], spin-based devices (e.g., spin-wave devices [198] and STMG [135]), and plasmonic-based devices [72]. Another interesting example arises from 2D materials-based devices as the ones reported in [148]. Even though these devices are not intrinsically based on the majority function, an efficient implementation of the majority-of-three-inputs gate has recently been demonstrated [147].

Although many and diverse examples of majority-based technologies exist, we discuss here in details (i) QCA, (ii) STMG, and (iii) plasmonic-based devices, as they will be used as running examples through the rest of the chapter.

Quantum-dot cellular automata

QCA technology is based on the interaction of QCA cells [106]. Each cell consists of four quantum dots and two free electrons. The free electrons can tunnel between the dots, which are coupled by tunnel barriers. Coulomb repulsion forces the electrons in opposite corners of the cell, thus producing two energetically equivalent polarizations, i.e., $P = 1$ and $P = -1$. The two polarizations are used to represent the logic values 1 and 0, respectively. QCA technology is functionally complete, and the fundamental logic element of QCA is the majority-of-three gate [182]. Figures 3.2(a) and (b) show the layout of a QCA majority gate and a QCA inverter, respectively. For the majority gate, the polarization of the central logic cell, called device cell, is the majority of the three inputs; the output cell follows the polarization of the device cell. In the inverter case, the input wire is first branched in two offset wires. Both have the same polarization as the input due to aligning effects. Anti-aligning effects at the second joint control the polarization of the next cell, causing an inversion of the input signal. In the last few years, 5-input majority gate realizations using QCA cells have been intensively studied [155]. The majority-5 is a versatile primitive and it can be employed to realize a variety of functions. Figure 3.2(c) shows one of the first implementations of the majority-5 [133]. It only requires ten QCA cells; on the other hand, the input cells are close to each other and difficult to be accessed. Improved versions of 5-input majority have been recently proposed [155, 159]. In these implementations, the 5 inputs are easier to reach, allowing single layer accessibility to the input and output cells.

Even though QCA technology enables the realization of 3- and 5-input majority gates, plus the inversion, some limitations and costs for circuits realization need to be discussed. The cost used to compare QCA blocks is the number of QCA cells [155], i.e., the area. The area of a QCA layout can be obtained by analyzing the layout with QCADesigner [186], a tool for the layout and analysis of QCA technology circuits. In this scenario, the inverter implementation is very expensive in terms of number of cells as compared to the majorities. Note that eleven QCA cells are needed for a single 5-input majority, while thirteen are necessary to change the polarity of each cell. It is thus preferable to limit the number of inversions in the circuit. Considering further constraints, each 3- and 5-input majority block has a fan-out limited to 3. This is due to the fact that in order to have the same polarization, two cells should be aligned and close to each other on one of the square borders (being 3 for each output signal). Moreover, the fabrication of interconnections between building blocks needs to be handled efficiently for better stability. Until now, an efficient and robust realization of wire crossing is not available, thus river routing is needed [155].

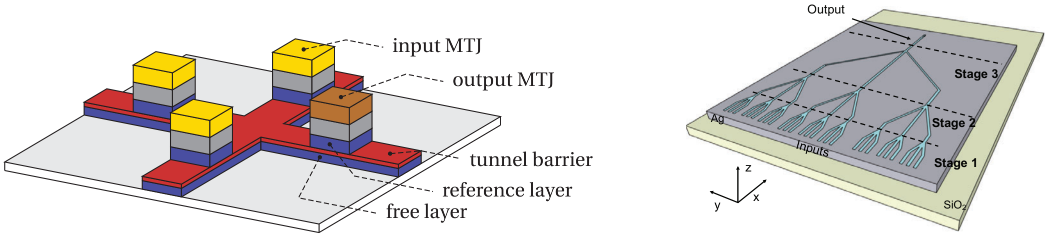


(a) 3-input majority

(b) inverter

(c) 5-input majority

Figure 3.2 – (a) QCA layout for majority, (b) inverter, and 5-input majority (c) [133]. © 2017 IEEE [181]



(a) STMG gate

(b) 3-stage plasmonic gate with 27 inputs

Figure 3.3 – (a) Schematic of STMG [135]; and (b) 3-stage cascaded plasmonic majority circuit. © 2017 IEEE [181]

Spin torque majority gates

STMG is a three-input majority gate driven by *spin transfer torque* (STT) [26] and has been proposed by Nikonov et al. [135]. It consists of a cross-shaped free layer shared between four *magnetic tunnel junctions* (MTJ) (see Figure 3.3(a)). The information (0 or 1) in the device is represented by the magnetization orientation (up or down) in the free layer. Three MTJs write the input states via STT in a current perpendicular to plane configuration. The fourth MTJ reads the output state via tunnel magnetoresistance. The magnetic domains are mainly driven by *domain wall automotion*, the transport of a magnetic domain wall under the influence of demagnetization and magnetic anisotropy [136]. The operating range has been extensively studied by micromagnetic simulations and numerical modeling in [184] and [185], respectively.

The STMG concept carries the potential of smaller area, low power, nonvolatility, reconfigurability, and radiation hardness [135]. However, there is a lack of an efficient *spin torque inverter* (STI) concept which would be necessary to implement circuits. A first inverter concept was presented in [134] where it was assumed that the functionality of an inverter is achieved through a ferromagnetic wire that connects two STMG devices and is fabricated as a slanted layer in the magnetic material stack. Due to its potential fabrication difficulties, this concept is considered unfeasible. We will describe Section 3.4 how to overcome this difficulty in implementation by producing inverter-free MIGs.

Plasmonic-based devices

Plasmonic-based devices [72] described hereafter are based on the propagation of *surface plasmon polaritons* (SPP) [23], which are electromagnetic waves propagating at the interface between a dielectric and a metal. In particular, the plasmonic-based logic considered in this thesis makes use of the phase ϕ of the SPP as logic variable. The computation is based on the interference of waves: in general, the output depends on the number of inputs with phase ϕ and $\phi+\pi$. The phase of interfering SPP waves follows the majority rule; this makes the 3-input majority function easy to realize with plasmonic-based devices [72]. Thanks to the physics of plasmonic devices that can be abstracted as multi-valued logic, it has been shown [72] that a 9-input majority gate can be easily realized using four 3-input plasmonic devices. Note that the best realization of majority-9 in binary-valued logic known so far uses 12 majority-3 gates (as demonstrated in Chapter 4), and thus plasmonic devices may be more efficient (as compared to other wave-based devices) to realize logic circuits. The wave nature of the computation allows us to easily implement the inverter by using a waveguide of half the length of the SPP wavelength. Thanks to this property, a complete set of logic primitives (INV and MAJ) can be built using plasmonic-based devices.

Plasmonic-based devices make a complete set of Boolean primitives; however, some constraints arise due to the wave nature and the physics of this device. As an example, the propagation losses of SPP put a limitation on the number of cascaded stages (i.e., the number of levels of the circuits). Currently, it is not efficient to have more than three stages, which means that after the third stage, either an amplifier or a converter to voltage domain is necessary. An example of 3-stage cascaded plasmonic majority is shown in Figure 3.3(b). The propagation losses across the first stage are around 30%, and keep increasing at every cascaded stage. The increase in propagation losses between the different stages is a direct consequence of the size difference between the devices in different stages (as shown in Figure 3.3(b)). As the size of the majority gates increases with the number of stages, also the delay of devices at different stages follows a similar trend. Furthermore, since the SPP wavelength has different values according to the stage, also the inversion cost depends on the stage at which it is implemented. It should also be noted that most emerging nanodevices target ultra-low energy operation, with an inherent low amplification and reduced driving capabilities. Thus, in addition to the constraints already considered, for this technology a strong limit exists on the maximum number of outgoing waves and, hence, on the maximum fan-out.

The presented emerging technologies naturally implement majority-based primitives, and have some potential to overcome the intrinsic scaling limitation of CMOS technologies. Nevertheless, they present some implementation limits in the depth, number of fan-outs or inverter realization. Section 3.4 and 3.5 will discuss logic synthesis algorithms to take these limitations into account.

```
ComputeTruthTable(W);
foreach node $u$ in $W$ in topological order do
    foreach node $v$ in $W \setminus \{u\}$ in topological order do
        if $v \in$ TransitiveFanout($u$) then continue;
        if $u = v$ then
            Merge($W, u, v$);
        else if $u = \bar{v}$ then
            Merge($W, u, \bar{v}$);
```

Algorithm 3.1: Functional reduction

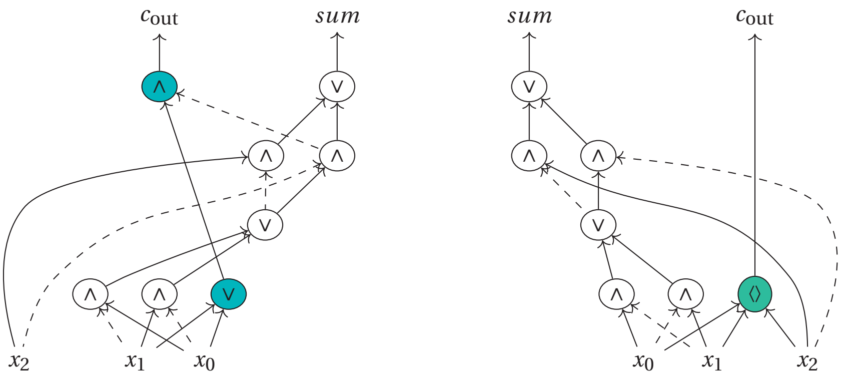
3.3 Boolean Resynthesis for MIGs

In this section, we concentrate on several methods for majority-based size optimization over MIGs. In contrast to the algebraic rewriting proposed in [14], we focus instead on Boolean methods. In particular, we revise functional reduction and introduce two new size optimization methods for MIGs: (i) Boolean resubstitution and (ii) replacement optimization. The first method is inspired by existing size optimization algorithms for non-majority-based logic networks; the second method leverages the properties of the majority function. Both methods are Boolean and make use of functional information computed for each node in the logic network. The basis for all optimization methods is the scalable logic synthesis framework described by Mishchenko and Brayton [119]: a small *window* (with restricted fan-in and unlimited fan-out) is moved over the logic network. The window is built using Algorithm 2.2 presented in Chapter 2. The Boolean function of each node within the window is computed using exhaustive simulation. The approach is fast (Boolean functions are represented as truth tables), scales well, and often outperforms computation based on *binary decision diagram*s (BDDs) [46] or Boolean satisfiability, when windows up to 16 inputs are considered.

3.3.1 Functional Reduction

Functional reduction (FR) [56, 59, 101] is an approach that identifies and merges functionally equivalent nodes in a logic network such that after its application no two nodes in the functionally reduced network represent the same logic function. In this section, we revise the basic functional reduction approach of [101] and present a scalable variant utilizing the windowing procedure from Chapter 2. The pseudocode is shown in Algorithm 3.1.

Functional reduction is applied to a window W. In an iterative process, each node $u \in W$ is checked for functionally equivalence with each node $v \in W$ not in the *transitive fan-out* (TFO) of u. If u and v (u and $\bar{v}$) represent the same logic function, i.e., $u = v$ ($u = \bar{v}$), then u and v ($\bar{v}$) are merged in W, such that the larger logic cone is replaced by the smaller logic cone and the overall size is reduced.



(a) MIG of the fulladder

(b) Optimized MIG by resubstitution

Figure 3.4 – Example of majority resubstitution. The blue nodes from (a) are substituted using the majority of already existing nodes, resulting in the network of (b)

3.3.2 Boolean Resubstitution

Boolean resubstitution (RS) expresses the logic function of a node using other nodes already present in the logic network. Resubstitution techniques are distinguished by the number k of logic nodes additionally added to the logic network when substituting a logic function, i.e., 0-resubstitution expresses a logic function by one other logic function without adding a new node; 1-resubstitution expresses a logic function by adding one logic operator, and so forth. An example of majority resubstitution is depicted in Figure 3.4. One node is added to the implementation, but the final circuit has reduced total number of nodes.

A resubstitution of a candidate node p with the logic function f is considered beneficial if the number of nodes of W decreases after substitution, i.e., if $\text{Gain}(p, f) \geq 1$ which corresponds to the number of majority operators freed. We consider 0-resubstitution and 1-resubstitution only:

```
ComputeTruthTable(W);
if TryResubstitution0($W, p$) then return ;
if TryResubstitution1($W, p$) then return ;
[...]
```

The 0-resubstitution algorithm is an asymmetric variant of functional reduction. Its pseudocode is identical to Algorithm 3.1, but instead of iterating over all nodes u (line 2) the fixed candidate node p is used.

The 1-resubstitution algorithm shown as Algorithm 3.2 searches for nodes x, y, z to replace p using one majority operator. Note that due to the inverter propagation rule $\overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle$ (see Chapter 2), it suffices to consider $\bar{x}$ as the only negated child. To further speed up the computation, we employ a Boolean filter derived from the majority law. If $x \neq y$, then $\langle xyp \rangle = p$ has to hold, i.e., after selecting nodes for x and y, one does not have to iterate over

```
foreach node $x \in W \setminus \{p\}$ in topological order do
    if $x \in$ TransitiveFanout(p) then continue;
    foreach node $y \in W \setminus \{p, x\}$ in top. order do
        if $y \in$ TransitiveFanout(p) then continue;
        if $p \neq \langle xyp \rangle$ then continue;
        foreach node $z \in W \setminus \{p, x, y\}$ in top. order do
            if $z \in$ TransitiveFanout(p) then continue;
            if Gain$(p, \langle xyz \rangle) < 1$ then continue;
            if $p = \langle xyz \rangle$ then
                $W = W[p \leftarrow \langle xyz \rangle]$;
                return true;
            else if $p = \langle \bar{x}yz \rangle$ then
                $W = W[p \leftarrow \langle \bar{x}yz \rangle]$;
                return true;
return false;
```

Algorithm 3.2: TryResubstitution1

z whenever the filter applies.

3.3.3 Replacement Optimization

In this section, we introduce *replacement optimization* (RO), a novel node replacement technique for MIGs that exploits the properties of the majority function and thus cannot be employed when restricted to gate libraries using only NOT and AND or NOT and OR.

This optimization makes use of the *replacement rule*, which describes under which condition one operand in a majority expression can be replaced by another one. A complete proof of this rule will be presented in Chapter 4. In fact, the replacement rule has been first introduced in the more general scenario of the theoretical results that will be presented in the mentioned chapter. We report the replacement rule in the followng to ease the reading of the thesis.

Theorem 3.1 (Replacement rule) *We have* $\langle xyz \rangle = \langle wyz \rangle$ *if and only if* $(x \oplus w)(y \oplus z) = 0$*, or in other words* $y \neq z \Rightarrow w = x$*.*

The replacement rule is used here to formulate an optimization procedure that replaces a child node x of a majority expression $m = \langle xyz \rangle$ with another node w if $(x \oplus w)(y \oplus z) = 0$ holds and x is not used by any other logic function in the network or as a primary output. These additional structural conditions stem from the fact that the replacement rule only enforces $x = w$ if $y \neq z$. Otherwise, if $y = z$, the result of m is determined by the majority law. However, in these cases, $x \neq w$ may hold which would affect other logic functions that use x. Further, to guarantee that the logic network stays free of cycles, the node p cannot be chosen from the transitive fan-out of m.

The replacement rule allows to reduce the complexity of a logic network in two ways: (i) If w is replaced by x, then x is no longer used in the logic network and can be removed. The size of the logic network is reduced if and only if x is not a constant. (ii) If the logic cone of w is smaller than x, the logic cone of m is reduced. Consequently, if multiple different nodes w satisfy the replacement rule, the x with the smallest logic cone is preferred. To find good candidate pairs x, w fast, we iterate over w in topological order, but over m in reverse topological order:

```
ComputeTruthTable(W);
foreach node $m = \langle xyz \rangle$ in $W$ in reverse top. order do
    foreach node $w$ in $W \setminus \{m\}$ in topological order do
        if $|\text{Fanout}(x)| > 1$ then continue;
        if $w \in \text{TransitiveFanout}(m)$ then continue;
        if $(x \oplus w)(y \oplus z) = 0$ then
            $W = W[x \leftarrow w]$;
            return ;
        else if $(x \oplus \bar{w})(y \oplus z) = 0$ then
            $W = W[x \leftarrow \bar{w}]$;
            return ;
```

All presented size optimizations can be integrated with existing large-scale logic optimization frameworks. Experimental results over unmapped network will be presented next, while mapped results over QCA and STMG are presented in Section 3.6.

3.3.4 Experimental Results

In this section, we illustrate the results for size optimization over MIGs. We implemented the presented size optimization methods in C++ and evaluated them using the EPFL combinational benchmark suite [15].[1] All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz. The windows were limited to at most 12 inputs and at most 200 nodes. We apply ABC [40] equivalence checking to ensure the correct behavior of each benchmark.

We show the size improvement obtained by applying each of the proposed techniques individually, and we compare our results over existing approaches from the state-of-the-art for size optimization [81]. We focus on reducing the size without any additional restrictions on the depth. Table 3.1 shows the results for all three optimization methods when applied individually to the benchmarks. The first column names the benchmarks, the remaining columns are organized in five blocks: the first block (**Benchmark**) lists the size and depth for the benchmarks. In the second block (**Prev. flow**), we present the size and depth of the MIGs when optimized with the best-known state-of-the-art approach. The other three blocks (**FR**, **RS**, **RO**) are structured in the same way and present the size and depth after an optimization method was applied as well as the time required for optimizing the benchmark. In the last

---

[1] Available at: *https://github.com/lsils/benchmarks*

Table 3.1 – Optimization methods applied to size optimized benchmarks

Benchmark			**Prev. flow [81]**		**FR [59]**			**RS**			**RO**		
	Size	**Depth**	**Size**	**Depth**	**Size**	**Depth**	**Time** [s]	**Size**	**Depth**	**Time** [s]	**Size**	**Depth**	**Time** [s]
ctrl	174	10	139	10	139	10	0.01	128	9	0.06	135	9	0.02
router	257	54	220	54	217	54	0.08	215	54	1.05	211	54	0.11
int2float	260	16	263	18	261	16	0.04	256	16	0.46	254	16	0.06
dec	304	3	328	4	328	4	0.00	328	4	0.26	328	4	0.01
cavlc	693	16	757	19	744	19	0.29	725	19	2.43	724	19	0.41
priority	978	250	993	245	993	245	2.15	978	239	9.95	807	125	0.92
adder	1020	255	386	129	386	129	0.08	385	129	1.29	386	129	0.07
i2c	1342	20	1329	23	1310	23	0.42	1298	23	2.78	1287	23	0.64
max	2865	287	2491	290	2428	261	4.72	2469	280	18.52	2448	279	5.70
bar	3336	12	3110	14	3110	14	2.40	3110	13	14.76	3088	14	3.75
sin	5416	225	4496	167	4480	162	4.74	4465	158	36.31	4480	170	6.08
arbiter	11839	87	8957	63	8957	63	8.82	8957	63	45.41	8957	63	11.97
voter	13758	70	7767	67	6649	59	31.90	5787	47	87.81	6537	61	45.10
square	18484	250	13671	156	13390	130	61.67	13194	128	109.84	13463	154	47.55
sqrt	24618	5058	21066	5989	21063	5989	102.62	20976	5942	624.11	21060	5988	109.43
multiplier	27062	274	19844	143	19824	143	72.16	19824	141	252.05	19804	143	122.00
log2	32060	444	25040	230	24999	230	89.96	24996	229	257.55	24977	230	109.43
mem_ctrl	46836	114	45034	144	44476	144	410.72	43305	136	1170.23	44118	143	600.50
total reduction			+15.30%	+5.59%	+16.34%	+8.18%		+17.64%	+10.56%		+18.13%	+10.22%	
improvement			0.00%	0.00%	+1.04%	+2.59%		+2.34%	+4.97%		+2.83%	+4.63%	

Table 3.1 shows the results for size improvements over MIGs. The three novel methods proposed in the chapter are tested: functional reduction (FR), Boolean resubstitution (RS) and replacement optimization (RO). The size is decreased on average of 18,13% w.r.t. the unoptimized benchmarks.

row of the table, the mean size and depth reductions are summarized for all benchmarks. The row **total reduction** shows the average reductions of the benchmarks achieved by the overall synthesis flows with respect to the unoptimized benchmarks. The row **improvement** shows the average reductions achieved by the new techniques with respect to the previous flow (**Prev. flow**).

Regarding the size-optimized MIGs, the three methods further reduced the size of the MIGs on average by 1.04%, 2.34%, and 2.83%, respectively. The size optimization also had a positive effect on the depth—the depth is reduced on average by 2.59%, 4.97%, and 4.63%, respectively. Moreover, the novel replacement optimization achieves a better size reduction than functional reduction and resubstitution, which results in an average reduction of 18.13% of nodes (up to 62.2% reduction for the *adder*) and 10.22% of levels in the MIGs. In these experiments, the optimization methods were applied only once. We argue that the presented techniques are, as in conventional logic synthesis, more powerful when applied several times interleaved with other optimization passes, e.g., rewriting, factoring.

3.4 Inverter Propagation and Fan-out Constraint

Logic synthesis addressing emerging nanotechnologies should take into consideration not only the new logic abstraction of the devices but also the different technological constraints given by the new ways of computation. In this section, we synthesize and optimize circuits by making use of a majority-based data-structure, and we introduce two algorithmic techniques

```
Input: MIG node $v$
Output: MIG node $v$ with inversions on inputs
$p \leftarrow polarity(v)$;
if $p = 1 \wedge v$ is not PI then
    apply $\Omega.I$ on $v$;
    $p \leftarrow 0$;
end
if $v$ is PI then
    return $v$;
end
if $v' \leftarrow is\_cached(v, p)$ then
    return $v'$;
end
foreach child c of v do
    $inv\_free(c)$;
end
$cache(v, p)$;
return $v$;
```

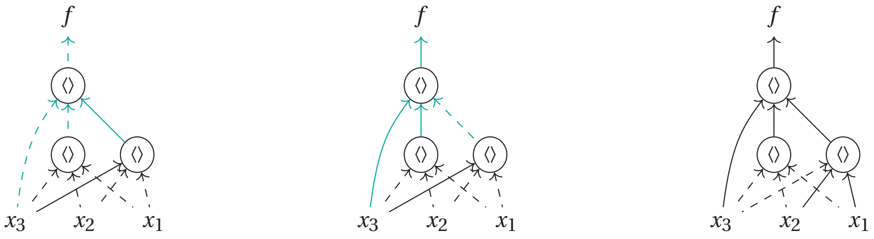
Algorithm 3.3: Inversion propagation

to rewrite MIGs for emerging technologies applications. The first algorithm, called *Inversion propagation* algorithm, propagates all inverters to the inputs in order to obtain an inversion-free MIG. The second algorithm, called *Fan-Out restriction* algorithm, limits the maximum fan-out of each node. Both methods aim at not changing the depth of the resulting graph. These two algorithms produce networks that can be adapted for majority-based beyond-CMOS technologies, such as QCA and STMG. As a matter of fact, both QCA and STMG have fan-out limitations, since their primitive logic structure relies on a cross-like shape. This means that the primitive fan-out gate for these two technologies has a fan-out of 3 ($\Phi = 3$). QCA-based circuits can realize inversion, even if not in an efficient way [105]. For STMG-based circuits, a feasible inversion implementation has not yet been proposed, so hybrid circuits need to be used.

3.4.1 Inversion Propagation

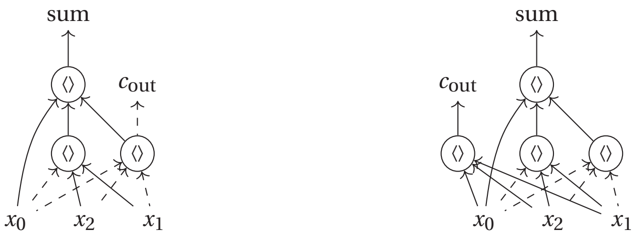
The *inversion propagation* algorithm rewrites the MIG to obtain a network where all inversions are placed on PIs. This is achieved by propagating inverters using the transformation rule $\overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle$, which is one of the axioms presented in Chapter 2 and called $\Omega.I$. The idea is to recursively apply $\Omega.I$ to move complemented edges from the outputs to the inputs. Previous works have been presented on inversion minimization [176, 179]. Here, the aim is to obtain a network with all inversions on inputs; that means, the total number of inversions may not be decreased. Our approach is similar to the one presented in [145], addressing AND/OR-based dynamic logic synthesis [142]; on the other hand, in our case, majority properties and rules are involved, aiming at emerging technologies applications.

The algorithm is based on a dynamic programming approach, and it consists of a recursive



(a) Original MIG (b) Inversion propagation (c) Final result

Figure 3.5 – Example for Algorithm 3.3. (a) is the original MIG for function $f = x \oplus y \oplus z$; (b) represents the same graph where $\Omega.I$ has been applied on the top node; (c) is the inversions free graph (i.e., inversions are on PIs)



(a) Original MIG (b) Inversion propagation

Figure 3.6 – Example for Algorithm 3.3 leading to a size increase. (a) is the original MIG; (b) represents the same graph after Algorithm 3.3 is applied

function called inv_free. The inversion propagation algorithm is shown in Algorithm 3.3. The algorithm starts by applying the function to each output of the network. If the node v is not complemented, the function inv_free is applied recursively to the children (lines 12–13 in Algorithm 3.3). If v is complemented, $\Omega.I$ is applied to the node before applying inv_free to the children (line 2 in Algorithm 3.3). In this second case, the polarities of the children are changed and the algorithm is applied taking into account the new polarities. The function is applied for each output. To avoid solving the same subgraphs more than once, all the computed solutions are cached.

Example 3.1 *An example is given in Figure 3.5. Figure 3.5(a) represents the original MIG for function $f = x \oplus y \oplus z$. Since output f is complemented, the rule $\Omega.I$ is applied on the output node f. Figure 3.5(b) shows the MIG with changed polarities. At this point, function inv_free can be applied on each child of the top node. Since one children node is complemented, $\Omega.I$ is applied. Since children of the node are all PIs, this subgraph is cached. The same procedure applies for the second and third child of the top node. The resulting MIG is shown in Figure 3.5(c). All the inversions are on PIs.* ■

The proposed algorithm does not change the depth of the graph, but it may result in an

```
   Input: MIG node $v$
   Output: MIG node $v$ with fan-out $\leq \Phi$
1  $f \leftarrow fo\_counter(v)$;
2  if $v$ is PI then
3      return $v$;
4  end
5  if $v' \leftarrow is\_cached(v, p) \wedge f < \Phi$ then
6      $fo\_counter(v)$++;
7      return $v'$;
8  end
9  if $v' \leftarrow is\_cached(v, p) \wedge f = \Phi$ then
10     remove $(v')$;
11 end
12 foreach child $c$ of $v$ do
13     $fo\_restr(c)$;
14 end
15 $fo\_counter(v) \leftarrow 1$;
16 $cache(v)$;
17 return $v$;
```

Algorithm 3.4: Fan-out restriction

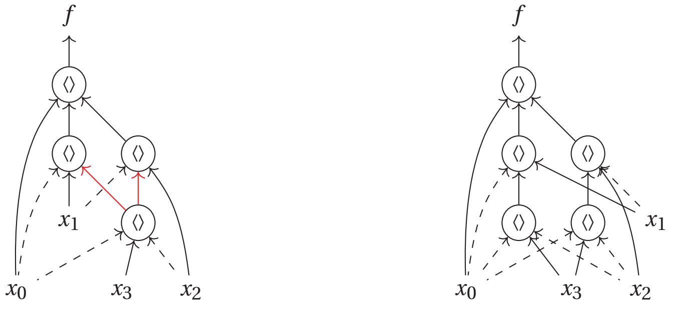
increase in the MIG's size. This happens if a node has fan-out with two different polarities. The example in Figure 3.6 shows the MIG of a full adder and it explains the size increase.

Example 3.2 *In Figure 3.6(a), there is one node with a fan-out of 2 (highlighted in red). The edge going to the top most node is not complemented, while the one going to output* c_{out} *has a complementation. To be able to move the negation on output* c_{out} *to the inputs, and without changing the polarity of the other outcoming edge, two copies of the same node are necessary. The final result is shown in Figure 3.6(b). The depth remains constant, but the size is increased.* ■

3.4.2 Fan-Out Restriction

The *fan-out restriction* algorithm rewrites the MIG such that every node has a fan-out less or equal to Φ. The main idea is to create copies of nodes with large fan-out. The algorithm is similar to the inversion propagation algorithm. It is based on a dynamic programming approach and it exploits a recursive function called fo_restr. The recursive function is shown in Algorithm 3.4. The algorithm starts by applying the function to each output. A counter called fo_restr cached the number of times node v is used (line 1 of Algorithm 3.4). For each node v, four different cases are possible:

1. v is a PI. In this case, the function returns the PI since the fan-out limit is not applied on inputs of the network;
2. The node is already cached and it has been used less than Φ times. Since the node has



(a) Original MIG

(b) MIG with restricted fan-out

Figure 3.7 – Example for the Fan-Out Restriction algorithm. (a) is the original MIG; (b) represents the same graph after fan-out restriction to 1

fan-out < Φ, the function returns the cached node. The $fo\_counter$ is updated (line 6 in Algorithm 3.4);

3. The node is already cached but it has been used Φ times (fan-out = Φ). The same node cannot be returned since the maximum fan-out is reached. In this case, a new node needs to be created. The function fo_restr is applied to the children. The cached value is updated and the counter is reset (lines 12–16);
4. The node is not cached. The recursive function fo_restr is applied to the children and the node is created and cached (lines 12–16);

It follows from this algorithm that if there is at least 1 node with fan-out > Φ, the size of the graph is increased; the depth, however, remains unchanged as we aim at keeping the same circuit speed.

Example 3.3 *Figure 3.7 shows an example with* $\Phi = 1$. *The algorithm starts from the top node and recursively reaches the PI. Figure 3.7(b) shows the final result. All nodes have fan-out* ≤ 1*; the size is increased, while the depth remains constant.* ■

It is important to highlight that this algorithm can lead to an exponential size increase. New copies of nodes lead to a fan-out increase for their children. This can result in new copies also for nodes that were not reaching the fan-out limit in the original MIG. A possible solution could be to use a method similar to the one proposed in [197]. The method consists of introducing one more level with *buffer nodes* reproducing the original node. In this way the maximum fan-out of the node is increased without introducing any new copy, hence keeping children fan-out unchanged. This alternative method could limit the size increase, but it results in a depth increase for the MIG. We applied this alternative method on the non-critical paths of some benchmarks. This mixed-method leads to small improvements as compared to Algorithm 3.4. Furthermore, for some circuits, the area increase is larger. For these reasons, we present here only the results obtained with Algorithm 3.4.

Both the inverter-free and the fan-out restriction algorithms manipulate the MIG in order to make it suitable to address nanotechnologies applications. As a matter of fact, we demonstrated inverter-free MIGs, having fan-out limited to Φ for each node. We will discuss and present results over (i) MIG (next section) and (ii) QCA and STMG mapped circuit (Section 3.6).

3.4.3 Experimental Results

This section describes results obtained with the two algorithms proposed in the previous sections. In order to obtain inversions free circuits with restricted fan-out, we developed two C++ programs to implement Algorithm 3.3 and Algorithm 3.4. We evaluate our methods on circuits from EPFL benchmarks, and we apply ABC [40] equivalence checking to ensure the correct behavior of each benchmark. First we applied Algorithm 3.3, then, on the same network, Algorithm 3.4 using $\Phi = 3$. Results are listed in Table 3.2. The depth of the circuit remains constant, while the final size is 1.9× larger on average. The column **Depth + Inv** is the path with maximum number of majority nodes and inversions. The **#INV** is the number of MIG nodes with at least one complemented fan-out. Note that the **#INV** in Table 3.2 refers to inversions on PI, since all circuits are inversion-free. The number of inversions is lower as compared to the original graph; only two circuits (*dec* and *router*) have the same number of inversions after the algorithms are applied. As expected, the algorithms result in increased size (× **Larger**), but, this is limited to only 3.3× larger network, in the worst case.

Table 3.2 – Inversion propagation and fan-out restriction algorithms © 2017 IEEE [181]

Benchmark				Original MIG			Optimized MIG				
Name	IN	OUT	Depth	Depth + Inv	Size	# INV	Depth + Inv	Size	× Larger	# INV	× Smaller
adder	256	129	12	21	2978	1449	13	6963	2.3	256	5.7
arbiter	256	129	14	27	2179	1042	15	2892	1.3	128	8.1
bar	135	128	14	28	3054	2999	15	3054	1.0	7	428.4
cavlc	10	11	11	21	745	370	12	955	1.3	10	37.0
ctrl	7	26	5	10	127	72	6	138	1.1	7	10.3
dec	8	256	4	5	420	8	5	528	1.3	8	1.0
i2c	147	142	9	18	1108	762	10	1354	1.2	93	8.2
int2float	11	7	9	15	246	121	10	279	1.1	11	11.0
log2	32	32	181	343	37582	22481	182	109839	2.9	32	702.5
max	512	130	27	53	7202	3147	28	21250	3.0	512	6.1
mem	1204	1231	18	36	10362	8519	19	14659	1.4	801	10.6
mult	128	128	111	205	41885	25594	112	113215	2.7	128	200.0
priority	128	8	10	18	498	295	11	621	1.2	122	2.4
router	60	30	9	13	113	62	10	185	1.6	60	1.0
sin	24	25	91	165	7890	3823	92	25557	3.2	24	159.3
sqrt	128	64	690	1249	52344	28734	691	174018	3.3	126	228.0
square	64	128	36	70	19200	17158	37	39234	2.0	64	268.1
voter	1001	1	60	114	14078	12064	61	31980	2.3	1001	12.1
Average									1.9		116.7

Table 3.2 shows the results for inverter propagation and fan-out restriction over MIGs. The opimized MIGs are inverter-free (with reduced inverters up to 702 ×) and have fan-out Φ limited to 3. Size increase of ~ 2 × on average is obtained.

3.5 Exact Synthesis with Constraints

The aim of exact synthesis is to find logic networks that represent given Boolean functions under a set of constraints. It indeed is a special case of the *minimum circuit size problem* [93], which asks whether a Boolean function g can be realized by a network of size at most r; it is considered an intractable problem [132]. Due to its complexity, exact synthesis is typically used to solve problems of limited size, i.e., functions with about 8 variables. Exact synthesis plays a key role in logic synthesis applications that need to take into account complex constraints. Many beyond-CMOS technologies have been studied in the last decade as replacement or enhancement for CMOS; and the broad variety of technologies has resulted in many and diversified constraints that need to be taken into account by novel logic synthesis tools. As an example, several emerging nanotechnologies do not have an efficient inversion implementation or have limited fan-out capabilities. Some technologies have more than one constraint that needs to be respected at the same time. These constraints are often present due to restrictions in the hardware primitives or the logic representations for which the synthesis has to be performed. Classical heuristic logic synthesis tools are usually not taking into account such constraints. They could be used in the optimization process, but they may lead to solutions that do not meet all the requested constraints. Moreover, no solution may exist if constraints are too tight and heuristic optimization algorithms cannot identify this.

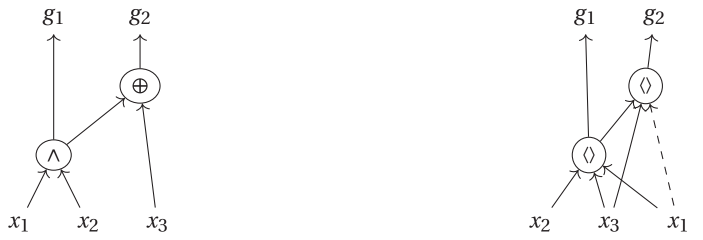
In this section, we illustrate the use of exact synthesis for logic synthesis applications that deal with many and diversified technological constraints. In the following, first, we review state-of-the-art SAT-based exact synthesis methods. Starting from Section 3.5.2, instead, we present novel methods to encode diverse constraints. In particular, we consider small multi-outputs functions (i) based on majority that necessitate (ii) limited-depth and (iii) restricted fan-out for each node. These requirements are motivated by plasmonic-based devices presented in Section 3.2, for which both depth and fan-out need to be restricted to maximum 3. We conclude the section with details on the exact synthesis algorithms and the experimental results.

3.5.1 Preliminaries

Exact methods have been briefly introduced in Chapter 2. In this section, we review methods for *SAT-based exact synthesis* based on the SAT formulation proposed by Knuth [98] to find the area-optimum normal Boolean network for functions $g_1, \dots, g_m$ which depend on n variables. This SAT encoding has been inspired by the work of Kojevnikov et al. [99] and Éen [74]. Recently, the formulation has been extended by Soeken et al. [163] for combinational delay optimization.

Recall that, given a function g of n inputs $x_1, \dots, x_n$, a Boolean network is defined as a sequence of 2-inputs gates $(x_{n+1}, \dots, x_{n+r})$, where for each gate i:

$$x_i = x_{j(i)} \circ_i x_{k(i)} \tag{3.1}$$



(a) 2-input Boolean network (b) 3-input majority Boolean network

Figure 3.8 – (a) example of 2-input Boolean network, $x_4 = x_1 \wedge x_2$ and $x_5 = x_3 \oplus x_4$. (b) example of 3-input majority Boolean network. Dashed lines are inverters

with $n < i \leq n + r$. In other words, the two inputs of each gate i are previous gates or inputs. The $\circ_i$ represents one of the 16 binary operations. A Boolean function g is called *normal* if $g(0, \ldots, 0) = 0$. If all the gates of a Boolean network are normal, then the network represents a normal Boolean function. For a normal Boolean network, each 2-input gate can represent 8 out of the 16 possible binary functions.

Knuth's idea is to verify if it is possible to realize functions $g_1, \ldots, g_m$ with a normal Boolean network of size r. In the following, variables and clauses proposed by Knuth are illustrated.

Variables

Let r be the number of gates, m be the number of outputs, and n be the number of inputs. Then, the variables used for the SAT formulation are:

$$\begin{aligned} x_{it}: &\quad t^{\text{th}}\text{bit of } x_i\text{'s truth table} \\ g_{hi}: &\quad [g_h = x_i] \\ s_{ijk}: &\quad [x_i = x_j \circ_i x_k] \text{ for } 1 \leq j < k < i \\ f_{ipq}: &\quad \circ_i (p, q) \text{ for } 0 \leq p, q \leq 1, p + q > 0 \end{aligned} \tag{3.2}$$

with $1 \leq h \leq m, n < i \leq n + r$, and $0 < t < 2^n$. For each gate x_i, the variable x_{it} represents the value of t^{th} bit in the truth table. Each output variable g_{hi} is true if the function g_h is represented by the gate x_i. The select variable s_{ijk} encodes the children of node x_i. The variable is true if gates x_j and x_k are the children of gate x_i. In this scenario, $\circ_i$ is one of the 8 normal 2-input Boolean functions. The variable f_{ipq} encodes the operation of gate x_i. This is true if for the input assignment (p, q), the operation x_i evaluates to true. It is important to highlight that this method works for normal Boolean functions. If a function is not normal, we find the optimum network for the inverted function. In the end, we invert the output node in order to obtain the original function. The normal property allows Knuth to ignore x_{i0} and f_{i00} for each i.

In the following, we illustrate an example taken from [163] to explain this SAT formulation and, in particular, the variables assignment. We refer the reader to [82, 98] for further details on SAT-based exact synthesis. We consider the network shown in Figure 3.8 (a), with inputs x_1, x_2 and x_3, therefore $n = 3$. In this example, $r = 2$, $x_4 = x_1 \wedge x_2$ and $x_5 = x_3 \oplus x_4$. The gate index i ranges from 4 to 5. Variable x_{it} encodes the truth table for each function of the multi-outputs network. Since $n = 3$ and since we know that $g(0,\dots,0) = 0$ (g is normal), the truth table bit t ranges from 1 to $2^n - 1 = 7$.

t	=	7	6	5	4	3	2	1
x_{4t}	=	1	0	0	0	1	0	0
x_{5t}	=	0	1	1	1	1	0	0

Since we are considering a multi-output network, each gate could be an output. Two out of the four output variables are assigned to 1, since $g_1 = x_4$ and $g_2 = x_5$.

$$g_{14} = 1,\ g_{15} = 0,\ g_{24} = 0,\ g_{25} = 1$$

There are three select variables for $i = 4$ and six for $i = 5$. For each gate, only one select variable is equal to 1. For instance, variable $s_{412} = 1$, since x_1 and x_2 are children of node x_4.

k	=	2	3	4
s_{41k}	=	1	0	
s_{42k}	=		0	
s_{51k}	=	0	0	0
s_{52k}	=		0	0
s_{53k}	=			1

Finally, the AND and XOR operations are encoded in the f_{ipq} variables. For this example:

p, q	=	0,1	1,0	1,1
f_{4pq}	=	0	0	1
f_{5pq}	=	1	1	0

Clauses

In order to have a working algorithm, some clauses need to be added:

- a main clause that describes how truth tables are computed for each gate, depending on children (s_{ijk}) and operation (f_{ipq}):

$$\begin{aligned}\left(s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})\right) \rightarrow (f_{ibc} \oplus \bar{a}) = \\ (\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a}))\end{aligned} \tag{3.3}$$

- a clause to constrain each output value to be the same as the one of the gate it points to;

- a clause to state that each output is realized by one gate in the network;
- and a clause to have two inputs for each gate.

In addition to the mandatory clauses listed above, some auxiliary clauses can be added to reduce the solving time of the SAT solver. More details about both clauses formalization and additional clauses can be found in [98, 163].

3.5.2 Constraints Encoding

In this section, we present our novel method to encode many constraints. In particular, we illustrate how state-of-the-art exact synthesis algorithms can be adjusted to solve constraint-problems. Knuth's algorithm is used to find the optimum normal Boolean network for functions $g_1, \dots, g_m$. In our case, we make use of MIGs as data structure for exact synthesis. Some changes to the original algorithm are then necessary in order to extend our analysis to 3-input majority gates. Further, some additional constraints need to be considered both for the maximum depth and for the maximum fan-out. Here, we demonstrate that Knuth's algorithm can be adapted and extended to work with 3-input majority gates and to limit depth and fan-out for multi-ouputs networks.

3-input Majority Gates Constraint

Here, the extension to 3-input gates and the restriction to only majority gates is illustrated.

The x_{it} and g_{hi} variables are used in the same way as proposed by Knuth. They encode the truth table and the output gates, respectively. Since we are working with 3-input gates, both the s_{ijk} and f_{ipq} need to be reexamined. Each select variable should consider three different children, here called x_j, x_k, and x_l. The select variables s_{ijkl} is true if the operands of gate x_i are x_j, x_k, and x_l. In a similar way, the function variables should take into account the 3-input operations. The variable f_{ipqu} is true if the operation of gate x_i is true under the input assignment (p, q, u).

In order to restrict the 3-input operations to only normal majority functions, a list of all 3-input majority truth table has been considered. Being p, q, and u the 3 inputs, each gate may realize $\langle pqu \rangle$, $\langle \bar{p}qu \rangle$, $\langle p\bar{q}u \rangle$ or $\langle pq\bar{u} \rangle$. Since the majority operator can behave as AND or OR using constant inputs, also all normal truth tables with constant 1 or 0 have to be considered. In this scenario, also ab, $\bar{a}b$, $a\bar{b}$, and $a+b$ have to be taken into account as possible normal majority operations. The total number of allowed operations is then equal to 8. However, the variables f_{ipqu} allow for a representation of all 128 normal 3-input functions. For each gate i, the operation variable o_{iw} is true if the operation of gate i is w, where w is one of the 8 possible normal majority operations.

Two clauses need to be added. First, a given f_{ipqu} implies a different operation w. For instance, if for gate i it holds that:

$$\begin{array}{rcccccccc} p,q,u & = & 0{,}0{,}1 & 0{,}1{,}0 & 0{,}1{,}1 & 1{,}0{,}0 & 1{,}0{,}1 & 1{,}1{,}0 & 1{,}1{,}1 \\ f_{ipqu} & = & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}$$

then the operation $\langle pqu \rangle$ is implemented. Being $\langle pqu \rangle$ the operation with $w = 1$, then the following constraint is added:

$$\begin{aligned} &(o_{i1} \rightarrow (\bar{f}_{001} \wedge \bar{f}_{010} \wedge f_{011} \wedge \bar{f}_{100} \wedge f_{101} \wedge f_{110} \wedge f_{111})) \\ = &(\bar{o}_{i1} \vee (\bar{f}_{001} \wedge \bar{f}_{010} \wedge f_{011} \wedge \bar{f}_{100} \wedge f_{101} \wedge f_{110} \wedge f_{111})) \end{aligned} \tag{3.4}$$

For each gate i, (3.4) is added for each operation w. Further, clause $\bigvee_{w=1}^{8} o_{iw}$ ensures that each gate realizes at least one of the 8 operations.

Both Knuth's algorithm and the one presented in [163] work with normal Boolean networks with 2-inputs gates. Previous work has considered 3-input gates [83]. A dedicated MIG encoding could have been considered, as in [162]. Nevertheless, here the aim is to demonstrate that existing algorithms can be adapted to solve complex constraints problem. In our case, we easily adjust existing algorithms, without changing clauses and with minor changes in the encoding of the variables.

Depth Constraint

We need to constrain the maximum depth of the network. The SAT solver should check whether there exists a MIG with r gates that can realize functions $g_1, \ldots, g_m$ with a depth less or equal to Δ. All input arrival times are considered to be 0. For each gate i, a variable d_i takes into account the depth of gate x_i with $n < i \leq n + r$. Each variable d_i has a value in the range $0 \leq d_i \leq (i - n)$. The idea is the same as the one proposed in [163], and the depth variable is encoded using the *order encoding* [98]. In this encoding, each value x in $0 \leq x \leq M$ is represented by a bitstring of length M. In particular, it is represented by x ones followed by $(M - x)$ zeros. To make use of order encoding, each depth variable is a bitstring and it is encoded as d_i^{ℓ}, where $1 \leq \ell \leq (i - n)$.

The minimum delay of gate x_i is the maximum delay of its children raised by 1. All inputs have a delay of 0, then for $j, k, l \leq n$ the d_i^{ℓ} variable has value equal to 0. The added clauses are:

$$\bigwedge_{\ell=1}^{j-n} (\bar{s}_{ijkl} \vee \bar{d}_j^{\ell} \vee d_i^{\ell+1}) \wedge \bigwedge_{\ell=1}^{k-n} (\bar{s}_{ijkl} \vee \bar{d}_k^{\ell} \vee d_i^{\ell+1}) \wedge \bigwedge_{\ell=1}^{l-n} (\bar{s}_{ijkl} \vee \bar{d}_l^{\ell} \vee d_i^{\ell+1}) \tag{3.5}$$

The clause $\bar{g}_{hi} \vee \bar{d}_i^{\Delta}$ ensures a depth $\leq \Delta$, by assigning 0 to the Δ^{th} bit in the order bitstring.

Fan-out Constraint

To constrain the maximum fan-out of each node, we make use of cardinality constraints. The cardinality constraint over a set of Boolean variables is a constraint on the number of variables that can have values equal to 1. In particular, we implement the cardinality constraint as proposed in [22].

In our case, the constraint is on the fan-out of each node to be at maximum Φ. Select variable s_{ijkl} encodes that gates x_j, x_k, and x_l are the children of node x_i. To consider the fan-out of node i, we need to take into account nodes with index larger than i: $s_{i+1jkl}, \ldots, s_{i+njkl}$. Among all these select variables we need to force a constraint on the ones that use i as children. In other words, all the select variables of index larger than i, in which j or k or l is equal to i. Also the output variables g_{hi} need to be considered for this fan-out count.

Example 3.4 *Figure 3.8 (b) shows an example of MIG exact synthesis.* x_1, x_2 *and* x_3 *are the inputs of the network, and* $n = 3$. *As in the previous example,* $r = 2$; $x_4 = \langle x_1 x_2 x_3 \rangle$ *and* $x_5 = \langle \bar{x}_1 x_3 x_4 \rangle$. *The gate index* i *ranges from 4 to 5. Variable* x_{it} *encodes the truth table for each output of the multi-outputs function. Further,*

$$g_{14} = 1,\ g_{15} = 0,\ g_{24} = 0,\ g_{25} = 1$$

There is only one select variables for $i = 4$, *which is* s_{4123}. *There are three select variables for node* 5. *For this gate, only one select variable is equal to 1:* $s_{5134} = 1$, *since* x_1, x_3, *and* x_4 *are children of node* x_5. *Finally, the two different majority operations are encoded in the* f_{ipqu} *variables. For this example:*

p,q,u	=	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
f_{4pqu}	=	0	0	1	0	1	1	1
f_{5pqu}	=	1	0	0	1	1	0	1

Only one operation variable o_{iw} *is equal to 1 for each node. For the depth, each variable* d_i^ℓ *has* $1 \leq \ell \leq (i-3)$. *It follows that for node* 4, *the depth variable consists of only 1 bit.* d_5^ℓ *is made of 2 bits, and it can have depth value of 0, 1, and 2. The fan-out constraint on node* 4 *consists of a cardinality constraint of type:*

$$s_{5124} + s_{5134} + g_{14} + g_{24} \leq \Phi \tag{3.6}$$

where all nodes that use 4 *as child are considered.* ■

3.5.3 Exact Algorithms

This section describes the implemented exact synthesis algorithm. It also illustrates three alternatives to the main algorithm.

```
1  Function $FindMIG(g, \Delta, \Phi, r)$
2      set $S \leftarrow SATSolver()$;
3      $AddVariables(S, g, \Delta, \Phi, r, )$;
4      foreach $0 < t < 2^n$ do
5          $AddMainClause(S, g, t)$;
6      end
7      $AddOtherClauses(S, r)$;
8      $AddOperationClause(S, g, r)$;
9      $AddDepthClause(S, \Delta)$;
10     foreach $n < i \leq n + r$ do
11         $AddFanOutClause(S, \Phi, i)$;
12     end
13     if $Solve(S)$ then
14         return $MIG$;
15     else
16         return $FindMIG(g, \Delta, \Phi, r + 1)$;
17     end
```

Algorithm 3.5: Function '$FindMIG()$'

The algorithm finds a MIG, if this exists, that satisfies all the constraints discussed in Section 3.5.2. The names of the variables are the ones used in previous sections. The input of the algorithm is the n-inputs m-outputs function g represented as truth tables obtained from the MIG that needs to be optimized. The algorithm starts by trying to find a solution using $r = m$ (assuming that each output represents a different function). If a solution exists for r gates, the algorithm returns a MIG that meets all the requirements, otherwise it looks for a solution with larger size. The algorithm increases the number of gates until the upper bound is reached. If no solution can be found up to the upper bound, it can be concluded that no network exists that meets all the constraints. Let m be the number of outputs, an upper bound for the number of gate r is $13m$. This result is obtained considering that each output could be represented as a tree, with no sharing edges between them. Each tree has one gate on the first level, 3 gates on the second level and 9 gates on the third one, thus 13 gates at most.

The algorithm is described as function $FindMIG()$ in Algorithm 3.5. First, the SAT solver is instantiated (line 2 in Algorithm 3.5). Then, the algorithm adds all the variables discussed in Section 3.5.1 and 3.5.2; they include the variables from Knuth's formulation, but also variables d_i^ℓ and o_{iw}. All clauses are then added (lines 4–12). The main clause is the one which encodes the truth table of the circuit (3.3); this is added for each bit t of each truth table. Other clauses consist of both necessary and additional clauses proposed in [163]; depth, operations, and fan-out clauses are the ones discussed in Section 3.5.2. The fan-out clause constraints the fan-out of each node i of the network.

Alternative implementations to Algorithm 3.5 are possible. All algorithms take into account the same constraints, however, they may show different performances. We rewrite Algorithm 3.5 by making use of *counter-example-guided abstraction refinement* (CEGAR). The idea is to overapproximate the solution space by discarding several clauses, thereby decreasing

```
1  Function $FindMIG\_CEGAR(g, \Delta, \Phi, r)$
2      set $S \leftarrow SATSolver()$;
3      $AddVariables(S, \Delta, \Phi, r)$;
4      $AddOtherClauses(S, r)$;
5      $AddOperationClause(S, g, r)$;
6      $AddDepthClause(S, \Delta, r)$;
7      foreach $n < i \leq n + r$ do
8          $AddFanOutClause(S, \Phi, i)$;
9      end
10     while $Solve(S)$ do
11         if Functionality_Respected(g) then
12             return $MIG$;
13         else
14             set $t \leftarrow$ first bit which does not respect functionality;
15             $AddMainClause(S, g, t)$;
16         end
17     end
18     return $FindMIG\_CEGAR(g, \Delta, \Phi, r + 1)$;
```

Algorithm 3.6: Function '$FindMIG\_CEGAR()$'

solving time of the SAT solver. Algorithm 3.6 illustrates one CEGAR version of Algorithm 3.5. Algorithm 3.6 does not add the main clause (3.3) which encodes the multi-output function g. In this way, the SAT solver may find a solution that does not coincide with g for all inputs assignment t. If this is the case, a refinement of the solution is pursued (lines 13–15). In order to ensure the same functionality, the main clause (3.3) is added for the first bit t of the truth table that does not agree with g. The updated problem is solved again by keeping the state of the SAT solver active (incremental SAT). This procedure is repeated until the truth tables coincide. During this refinement process, two possibilities emerge:

1. The SAT solver converges to a solution which respects the functionality;
2. The SAT solver is not able to find a solution that respects the new clauses. In this case, the size r is increased and a new solution is searched.

CEGAR can also be applied to abstract the fan-out clauses. First, a solution without fan-out constraints is found; then, the algorithm checks whether a gate i exists that does not respect the fan-out constraint. If it exists, the fan-out constraint is added only for gate i, which has fan-out $> \Phi$. The algorithm is not reported here since it is similar to Algorithm 3.6.

Both CEGAR methods can also be applied at the same time. We call this method *DoubleCEGAR* (DCEGAR) and it is shown in Algorithm 3.7. In this approach, both the truth table and fan-out clauses are not added in the main function. If a solution of size r exists, the functionality is checked (line 8). If the functionality is respected, then the algorithm ensures that also the fan-out constraint is met (line 9). If both are respected, the MIG is returned (line 10). Otherwise, first, the algorithm tries to meet the truth table constraint (lines 15–18) and then the fan-out one (lines 12–13). The algorithm works in a similar way as Algorithm 3.6; if at

```
Function $FindMIG\_DCEGAR(g, \Delta, \Phi, r)$
    set $S \leftarrow SATSolver()$;
    $AddVariables(S, \Delta, \Phi, r)$;
    $AddOtherClauses(S, r)$;
    $AddOperationClause(S, g, r)$;
    $AddDepthClause(S, \Delta, r)$;
    while $Solve(S)$ do
        if Functionality_Respected(g) then
            if $All\_Fanouts() \leq \Phi$ then
                return $MIG$;
            else
                set $i \leftarrow$ node with fan-out $> \Phi$;
                $AddFanOutClause(S, \Phi, i)$;
            end
        else
            set $t \leftarrow$ first bit which does not respect functionality;
            $AddMainClause(S, g, t)$;
        end
    end
    return $FindMIG\_DCEGAR(g, \Delta, \Phi, r+1)$;
```

Algorithm 3.7: Function '$FindMIG\_DCEGAR()$'

some point the SAT solver cannot find a solution due to the new clause, then the algorithm searches for a solution with size $r+1$.

The proposed algorithms can be employed in logic synthesis for applications that seek for low-depth majority-based networks with limited fan-out, e.g., plasmonic-based devices. We demonstrated how state-of-the-art exact synthesis algorithms can be adapted and used to find logic networks that match these constraints. To emphasize the need for exact synthesis, we will also demonstrate in the coming section how conventional logic synthesis either fails to find constraint-satisfying logic networks or yields networks of inferior quality.

3.5.4 Experimental Results

In this section, first, we demonstrate how conventional logic synthesis tools are not suitable for some complex constraints-problem; then, we illustrate results obtained with the different algorithms proposed in Section 3.5.3. Finally, we discuss the feasibility of our method on larger functions.

We developed a C++ program[2] to implement Algorithm 3.5. The implementation uses one of the SAT solvers implemented in ABC [40]. Motivated by plasmonic-based devices, for these experiments we used the maximum depth $\Delta = 3$ and the maximum fan-out $\Phi = 3$. We applied our approach to small arithmetic benchmarks and to some small *hwb* [1] circuits. The 'HWB34' benchmark is a multi-output function containing both *hwb3* and *hwb4*. To

---

[2] Available at: *github.com/eletesta/cirkit-addon-mign-sat*

emphasize the key role of exact synthesis for complex constraint-problems, we demonstrate that classical logic synthesis tools may fail in finding a solution that meets all the constraints. Results are shown in Table 3.2. We optimized circuits using ABC depth optimization (*'clp; sop; fx; strash; resyn2'*). The results are shown in the first part of Table 3.2; only two circuits out of six meet the depth constraint. The **FO viol.** column represents the number of nodes that violate the fan-out constraint, thus with fan-out > Φ; for the 'ADDER2x2' benchmark, also the fan-out constraint is not respected. The second block of Table 3.2 shows results obtained by analyzing each output separately. Each function has been depth-optimized using exact synthesis proposed in [162]. This approach leads to results that meet our depth constraint, but it is time-consuming, since all outputs are analyzed separately. Further, this does not optimize the network considering the common nodes and it does not take into consideration the fan-out constraint. Copies of nodes with fan-out > Φ have been produced. In this case, the runtime is the sum of the runtimes necessary for each output; the manual work needed to separate and reunite the whole circuit is not taken into account. The third block of Table 3.3 shows the results achieved using our exact algorithm approach, disregarding the fan-out constraint. Also, in this case, copies of nodes with fan-out > Φ are introduced. Table 3.3 shows that the better results are the ones obtained with the exact method implemented in Algorithm 3.5, therefore considering all the constraints (both depth and fan-out). In this case, there are no nodes with fan-out larger than Φ. As an example, ADDER2x2 leads to a better result when also fan-out constraint is added. For these benchmarks, both the exact solutions (disregarding and considering fan-out) lead to a depth equal to 3 and size equal to 6. But for the first case, a copy of one node needs to be introduced since its fan-out is larger than Φ; producing in this way size of 7.

Table 3.3 – Classical heuristic and exact synthesis comparison

		Classical Heuristic ABC				Exact [162] - separated outputs				Exact - no fan-out				Algorithm 3.5		
Benchmark	I/O	Depth	Size	FO viol.	Time [s]	Depth	Size	FO viol.	Time [s]	Depth	Size	FO viol.	Time [s]	Depth	Size	Time [s]
ADDER 2x2	4/3	**4**	11	**1**	0.14	3	7	**1**	0.42	3	6	**1**	1.15	3	6	0.56
MULT 2x2	4/4	3	8	—	0.14	3	12	—	0.75	3	8	—	76.84	3	8	68.92
BITCOUNT3	3/2	**4**	8	—	0.14	2	3	—	0.09	3	3	—	0.00	2	3	0.05
HWB3	3/1	2	3	—	0.13	2	3	—	0.00	2	3	—	0.00	2	3	0.05
HWB4	4/1	**4**	8	—	0.14	3	5	—	0.25	3	5	—	0.25	3	5	0.23
HWB34	4/2	**4**	11	—	0.14	3	7	—	0.84	3	6	—	2.16	3	6	1.97

Table 3.3 compares the results of exact synthesis with the state-of-the-art heuristic and exact synthesis methods. These methods do not take into account the constraints and thus produce results that do not meet all the specifications: (i) depth limited to 3; (ii) fan-out limited to 3; (iii) majority primitives.

We applied the four alternatives of Algorithm 3.5 to the same circuits; results are listed in Table 3.4. The first algorithm is the one without CEGAR approach. The second one is Algorithm 3.6, while the third one is the one in which the CEGAR method is applied not considering the fan-out clause. DOUBLE CEGAR is the last method in Table 3.4. The runtimes of the four methods are similar. This is not surprising, since the number of inputs in the considered benchmarks is small. It is important to highlight that we are not optimizing the depth, but just constrain it to be ≤ Δ. For the BITCOUNTER3, exact solutions with different depths are found. An extension that considers multi or all exact solutions can be easily included in the algorithm.

Table 3.4 – Comparison of the different exact algorithms proposed

		Algorithm 3.5 - All clauses			Truth Table CEGAR			Fan-out CEGAR			DCEGAR		
Benchmark	I/O	Depth	Size	Time [s]	Depth	Size	Time [s]	Depth	Size	Time [s]	Depth	Size	Time [s]
ADDER 2x2	4/3	3	6	0.56	3	6	0.73	3	6	1.33	3	6	0.93
MULT 2x2	4/4	3	8	68.92	3	8	77.58	3	8	76.01	3	8	94.26
BITCOUNT3	3/2	2	3	0.05	3	3	0.05	3	3	0.05	3	3	0.05
HWB3	3/1	2	3	0.05	2	3	0.05	2	3	0.05	2	3	0.05
HWB4	4/1	3	5	0.23	3	5	0.30	3	5	0.24	3	5	0.41
HWB34	4/2	3	6	1.97	3	6	4.93	3	6	2.22	3	6	2.24

Table 3.4 compares the results of the 4 **proposed exact synthesis algorithms. Similar runtimes are obtained for such small benchmarks.**

Table 3.5 – Sat-based algorithm on LUTs from EPFL benchmarks

	3-LUTs				4-LUTs				5-LUTs				6-LUTs			
Benchmark	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO
adder	41	41	100	0	185	185	100.0	0	343	342	99.7	1	399	379	95.0	20
arbiter	64	64	100	0	285	285	100.0	0	395	395	100.0	0	419	419	100.0	0
bar	8	8	100	0	8	8	100.0	0	14	14	100.0	0	13	8	61.5	5
cavlc	65	65	100	0	216	216	100.0	0	190	185	97.4	5	139	115	82.7	24
ctrl	24	24	100	0	34	34	100.0	0	26	23	88.5	3	24	19	79.2	5
dec	8	8	100	0	24	24	100.0	0	32	32	100.0	0	40	40	100.0	0
i2c	76	76	100	0	159	159	100.0	0	151	149	98.7	2	145	124	85.5	21
int2float	45	45	100	0	68	68	100.0	0	55	54	98.2	1	40	30	75.0	10
log2	166	166	100	0	986	984	99.8	2	1620	1485	91.7	135	1932	1428	73.9	504
max	47	47	100	0	115	115	100.0	0	142	142	100.0	0	155	154	99.4	1
mem_ctrl	132	132	100	0	723	721	99.7	2	927	921	99.4	6	948	857	90.4	91
mult	132	132	100	0	784	782	99.7	2	1190	1115	93.7	75	1303	1099	84.3	204
priority	30	30	100	0	53	53	100.0	0	61	61	100.0	0	61	61	100.0	0
router	21	21	100	0	25	25	100.0	0	27	27	100.0	0	25	25	100.0	0
sin	142	142	100	0	700	699	99.9	1	958	910	95.0	48	915	751	82.1	164
sqrt	192	192	100	0	1754	1752	99.9	2	4574	4394	96.1	180	5505	4654	84.5	851
square	112	112	100	0	493	491	99.6	2	615	565	91.9	50	692	571	82.5	121
voter	79	79	100	0	194	192	99.0	2	206	192	93.2	14	215	177	82.3	38
Average			100.0%				99.9%				96.8%				86.6%	

#LUTs is the total number of unique LUTs; # SAT is the number of LUTs which are satisfiable; % is the precentage of SAT over the total number of LUTs; #TO is the number of LUTs that are not finished before the timeout.

Table 3.5 shows results on larger benchmarks. When 3**-LUTs are used to map the circuit, all the functions can be realized with circuits of depth and fan-out** = 3**.** 6**-LUTs results show that on average** 86.6% **of LUTs can be realized with the given constraints.**

The constraints used so far are motivated by an industrial application based on plasmonic-based logic in which each small function is part of a larger function, and each function should meet the depth and fan-out requirements. To validate the feasibility of this method, we map networks using LUTs of different size; then we apply the SAT-based method on each LUT. We are interested in finding how many LUTs can be realized as MIGs that meet all the given constraints for depth and fan-out.

We applied this method on circuits from the EPFL benchmarks, using LUT size $k = 3, 4, 5, 6$,

Table 3.6 – Technology parameters for QCA inverter and majority gates © 2017 IEEE [181]

	Area (μm^2)	Delay (ns)	Energy (fJ)
INV	4.0×10^{-3}	1.4×10^{-2}	9.8×10^{-6}
MAJ	1.2×10^{-3}	4×10^{-3}	2.9×10^{-6}

Table 3.6 shows the technological parameters for QCA majority and inverter. These parameters can be used to evaluated area, delay, and energy of circuits. The parameters have been extracted and obtained by IMEC.

respectively. The LUTs and the functions they represent are obtained from *CirKit*[3] using the command *'xmglut'*. The SAT-based method is applied on all the unique LUT functions, using a maximum depth $\Delta = 3$ and a maximum fan-out $\Phi = 3$. The experiments were performed on a computer with Intel Xeon Processor E5-2680 v3, @ 2.5 GHz, 64 Gb RAM, using a timeout of 1 minute for each LUT function. Table 3.5 shows the results; in particular, it lists the total number of unique LUTs, the satisfiable LUTs and the total timeouts for each LUT size k. Results show that when 3-LUTs are used to map the circuit, all the functions can be realized with circuits of $\Delta \leq 3$ and $\Phi \leq 3$, and with a runtime ≤ 1 minute. For LUTs of larger size, some timeouts are present and not all the LUTs can be finished in less than 1 minute. No conclusions can be drawn on the satisfiability of functions that were not concluded in 1 minute: they could be both satisfiable or unsatisfiable. In the worst-case scenario (i.e. all timeouts are UNSAT), results for 6-LUTs show that on average 86.6% of LUTs can be realized with the given constraints. In general, a high number of LUTs from EPFL benchmarks can be realized with circuits that meet all the constraints; we are then confident that our method could produce good results when considering partitioned functions.

3.6 Area-Delay-Energy Product for QCA and STMG

In this last section, we show global trade-off results on QCA and STMG. We demonstrate how QCA-based circuits benefit from inversion-free networks with limited fan-out, and in particular, how such networks result into smaller *area-delay-energy product* (ADEP) of mapped circuits. We also demonstrate the realization of STMG-based circuits, without using inversions. Finally, we present size optimization results after QCA and STMG technology mapping compared to state-of-the-art size optimization techniques. Also in this case, size optimization achieves optimized ADEP for both QCA and STMG technologies.

First, we evaluate area, delay, and energy of QCA-based circuits using the specification from Table 3.6. These specifications have been obtained by IMEC and extracted from [86, 182]. Table 3.7 shows the experimental results comparing unoptimized MIG with our otimized one. Note that for this experimental evaluation the optimized MIG is the one without inversions and with fan-out limited to 3 (i.e., circuits from Section 3.4 and Table 3.2). Even if the optimized

---

[3] Available at: *github.com/msoeken/cirkit*

MIG has larger size (see Table 3.2), this leads to a smaller QCA area, since QCA inverters are much larger compared to majority gates. The inversion-free and fan-out restricted netlists yield on average 3.1× smaller ADEP. Only one benchmark (*dec*) has a slightly worse ADEP; all other circuits benefit from inversion-free and fan-out restriction.

Table 3.7 – Inversion-free and fan-out restriction on QCA © 2017 IEEE [181]

	Original				Optimized MIG			
Name	Area (μm^2)	Delay (ns)	Energy (fJ)	ADEP (a.u.)	Area (μm^2)	Delay (ns)	Energy (fJ)	ADEP (a.u.)
adder	9.4	1.7×10^{-1}	2.3×10^{-2}	3.7×10^{-2}	9.4	6.2×10^{-2}	2.3×10^{-2}	1.3×10^{-2}
arbiter	6.8	2.4×10^{-1}	1.7×10^{-2}	2.7×10^{-2}	4.0	7×10^{-2}	9.8×10^{-3}	2.7×10^{-3}
bar	1.6×10^{1}	2.5×10^{-1}	3.8×10^{-2}	1.5×10^{-1}	3.7	7×10^{-2}	9.0×10^{-3}	2.3×10^{-3}
cavlc	2.4	1.8×10^{-1}	5.8×10^{-3}	2.5×10^{-3}	1.2	5.8×10^{-2}	2.9×10^{-3}	2×10^{-4}
ctrl	5×10^{-1}	9×10^{-2}	1.1×10^{-3}	4.3×10^{-5}	1.9×10^{-1}	3.4×10^{-2}	4.7×10^{-4}	3.1×10^{-6}
dec	5×10^{-1}	3×10^{-2}	1.3×10^{-3}	2.1×10^{-5}	6.7×10^{-1}	3×10^{-2}	1.6×10^{-3}	3.3×10^{-5}
i2c	4.4	1.6×10^{-1}	1.1×10^{-2}	7.6×10^{-3}	2.0	5×10^{-2}	4.9×10^{-3}	4.9×10^{-4}
int2float	8×10^{-1}	1.2×10^{-1}	1.9×10^{-3}	1.8×10^{-4}	3.8×10^{-1}	5×10^{-2}	9.3×10^{-4}	1.8×10^{-5}
log2	1.4×10^{2}	3.0	3.3×10^{-1}	1.3×10^{2}	1.3×10^{2}	7.4×10^{-1}	3.2×10^{-1}	3.1×10^{1}
max	2.1×10^{1}	4.7×10^{-1}	5.2×10^{-2}	5.2×10^{-1}	2.8×10^{1}	1.2×10^{-1}	6.7×10^{-2}	2.3×10^{-1}
mem	4.7×10^{1}	3.2×10^{-1}	1.1×10^{-1}	1.7	2.1×10^{1}	8.6×10^{-2}	5.1×10^{-2}	9.1×10^{-2}
mult	1.5×10^{2}	1.8	3.7×10^{-1}	1.0×10^{2}	1.4×10^{2}	4.6×10^{-1}	3.3×10^{-1}	2.1×10^{1}
priority	1.8	1.5×10^{-1}	4.4×10^{-3}	1.2×10^{-3}	1.2	5.4×10^{-2}	3.0×10^{-3}	2.0×10^{-4}
router	4×10^{-1}	9.2×10^{-2}	9.4×10^{-4}	3.3×10^{-5}	4.6×10^{-1}	5×10^{-2}	1.1×10^{-3}	2.6×10^{-5}
sin	2.5×10^{1}	1.4	6.1×10^{-2}	2.1	3.1×10^{1}	3.8×10^{-1}	7.5×10^{-2}	8.8×10^{-1}
sqrt	1.8×10^{2}	1.1×10^{1}	4.4×10^{-1}	8.2×10^{2}	2.1×10^{2}	2.8	5.1×10^{-1}	3.0×10^{2}
square	9.2×10^{1}	6.2×10^{-1}	2.2×10^{-1}	1.3×10^{1}	4.7×10^{1}	1.6×10^{-1}	1.2×10^{-1}	8.7×10^{-1}
voter	6.5×10^{1}	10.0×10^{-1}	1.6×10^{-1}	1.0×10^{1}	4.2×10^{1}	2.5×10^{-1}	1.0×10^{-1}	1.1
Averages	4.2×10^{1}	1.1	1.0×10^{-1}	6.0×10^{1}	3.7×10^{1}	3.1×10^{-1}	9.1×10^{-2}	2.0×10^{1}

Table 3.7 shows results of mapped MIGs into QCA; optimized MIG are inverter-free and with restricted fan-out (see Table 3.2). Even though the size of our MIG is larger, the optimization technique results in 2× smaller ADEP.

We also evaluate area, delay, and energy for the STMG-based circuits. Note that the inversion-free MIGs obtained with our algorithms allow realization of STMG-based circuits, that do not involve inverters. As a matter of fact, there is a lack of an efficient STI concept which would be necessary to implement circuits. A first inverter concept was presented in [134] where it was assumed that the functionality of an inverter is achieved through a ferromagnetic wire that connects two STMG devices and is fabricated as a slanted layer in the magnetic material stack. However, this concept cannot be realized with state-of-the-art magnetic material integration technology. We can envision two possible flavors of STMG devices that can implement netlists:

1. STMG/CMOS hybrid, where each inversion is implemented by CMOS inverters. This assumes that for each inversion an MTJ is read and the next one is written with the inverted result.
2. STMG/NML hybrid, where each inversion is implemented by an out-of-plane nanomagnet, as in the NML concept. This assumes that there is no conversion to the electric domain and the inversion is implemented in the magnetic domain.

Table 3.8 – Technology parameters for STMG hybrid flavors inverter and majority gates © 2017 IEEE [181]

	Area (μm^2)	Delay (ns)	Energy (fJ)
MTJ write/read	0	4	70
MAJ	3.6×10^{-3}	1.5	0
INV-CMOS	6.0×10^{-2}	2.6×10^{-2}	4.0×10^{-1}
INV-NML	2.3×10^{-2}	10	0

Table 3.8 shows technological parameters for STMG. These parameters can be used to evaluated area, delay, and energy of circuits. The parameters have been extracted and obtained by IMEC. STMG cannot realize inverters, thus hybrid soutions are proposed using CMOS and NML.

We benchmark and compare the results of our algorithms to these two aforementioned STMG hybrid flavors. For this comparison we use the primitive area, delay, and energy constants shown in Table 3.8 (obtained and extracted at IMEC). The MTJ parameters are extracted from [91], the CMOS inverter parameters are extracted from a CMOS 7 nm node [143], and the NML inverter is assumed to be a 150 nm×150 nm nanomagnet based on [42]. In both cases CMOS and NML are used in order to obtain inversions.

Table 3.9 shows the STMG experimental results. As for the QCA case, the optimized MIG is the one obtained in Table 3.2 (presented in Section 3.4). The STMG-based circuits produce, on average, both smaller area and delay. On average, a 8.1× smaller ADEP compared to STMG/CMOS hybrids and 2.9× smaller ADEP compared to STMG/NML hybrids is obtained using only STMG. In this case, all circuits benefit from inversion-free and fan-out restriction. Note that the inversion free algorithm allows the relization of STMGs circuits, without using CMOS or NML cells.

On top of the presented results, we also test the efficiency of our size optimization methods from Section 3.3 and presented in Table 3.1, by mapping the logic networks into QCA and STMGs. We compare our results to the state-of-the-art approach presented in [187], which we reimplemented using FR from Section 3.3 and windowing to achieve larger scalability and therefore address larger benchmarks. In our experiments, we found that the results obtained with our implementation of [187] outperform the more recently presented results in [59]; this can easily be validated by comparing the numbers for size and depth in Table 3.1 to [59, Table III] for the common benchmarks (*i2c, max, square, log2, multiplier*).

Table 3.10 shows area, delay, energy, and the ADEP for each of the benchmarks when mapped to QCA and STMG technologies, respectively. Total reduction compares the results to the original EPFL benchmarks, while the improvement is evaluated with respect to [187]. In this case, since STMGs and QCA technologies have limited possibilities for inverter implementation, we always applied the Algorithm 3.3 (Section 3.4.1) in order to create inversion free circuits. For size optimization, we used the approach in [81] followed by the FR, RO, and

Table 3.9 – Inversion-free and fan-out restriction on STMG © 2017 IEEE [181]

	Original MIG — STMG/CMOS				Original MIG — STMG/NML				Optimized MIG			
Name	Area (μm^2)	Delay (ns)	Energy (nJ)	ADE (a.u.)	Area (μm^2)	Delay (ns)	Energy (nJ)	ADE (a.u.)	Area (μm^2)	Delay (ns)	Energy (nJ)	ADE (a.u.)
adder	8.7×10^{1}	9.4×10^{1}	2.5×10^{-1}	2.1×10^{3}	4.3×10^{1}	1.1×10^{2}	1.5×10^{-1}	7.2×10^{2}	2.5×10^{1}	1.8×10^{1}	4.8×10^{-1}	2.2×10^{2}
arbiter	6.3×10^{1}	1.3×10^{2}	2.7×10^{-1}	2.2×10^{3}	3.1×10^{1}	1.6×10^{2}	2.0×10^{-1}	9.5×10^{2}	1.0×10^{1}	2.1×10^{1}	2.7×10^{-1}	5.8×10^{1}
bar	1.8×10^{2}	1.4×10^{2}	6.0×10^{-1}	1.5×10^{4}	7.8×10^{1}	1.7×10^{2}	3.9×10^{-1}	5.1×10^{3}	1.1×10^{1}	2.1×10^{1}	3.9×10^{-1}	9.0×10^{1}
cavlc	2.2×10^{1}	1.0×10^{2}	1.1×10^{-1}	2.4×10^{2}	1.1×10^{1}	1.2×10^{2}	8.3×10^{-2}	1.1×10^{2}	3.4	1.7×10^{1}	1.2×10^{-1}	6.6
ctrl	4.3	5.2×10^{1}	2.2×10^{-2}	4.8	2.1	6.2×10^{1}	1.7×10^{-2}	2.1	5.0×10^{-1}	7.5	1.9×10^{-2}	7.1×10^{-2}
dec	1.5	1.8×10^{1}	5.0×10^{-2}	1.4	1.7	2.0×10^{1}	5.0×10^{-2}	1.7	1.9	6.0	6.5×10^{-2}	7.4×10^{-1}
i2c	4.6×10^{1}	9.0×10^{1}	1.9×10^{-1}	7.8×10^{2}	2.1×10^{1}	1.1×10^{2}	1.4×10^{-1}	3.1×10^{2}	5.6	1.4×10^{1}	1.7×10^{-1}	1.3×10^{1}
int2float	7.3	6.6×10^{1}	3.9×10^{-2}	1.9×10^{1}	3.6	7.8×10^{1}	3.1×10^{-2}	8.6	1.0	1.4×10^{1}	3.6×10^{-2}	5.0×10^{-1}
log2	1.3×10^{3}	1.6×10^{3}	4.2	8.9×10^{6}	6.4×10^{2}	1.9×10^{3}	2.6	3.2×10^{6}	4.0×10^{2}	2.7×10^{2}	7.9	8.5×10^{5}
max	1.9×10^{2}	2.5×10^{2}	7.8×10^{-1}	3.7×10^{4}	9.7×10^{1}	3.0×10^{2}	5.6×10^{-1}	1.6×10^{4}	7.7×10^{1}	4.1×10^{1}	1.6	5.0×10^{3}
mem	5.1×10^{2}	1.8×10^{2}	1.8	1.6×10^{5}	2.3×10^{2}	2.1×10^{2}	1.2	5.8×10^{4}	5.3×10^{1}	2.7×10^{1}	1.7	2.4×10^{3}
mult	1.5×10^{3}	9.2×10^{2}	4.9	6.9×10^{6}	7.3×10^{2}	1.1×10^{3}	3.1	2.5×10^{6}	4.1×10^{2}	1.7×10^{2}	8.2	5.6×10^{5}
priority	1.8×10^{1}	8.3×10^{1}	7.5×10^{-2}	1.1×10^{2}	8.4	9.9×10^{1}	5.4×10^{-2}	4.5×10^{1}	7.3	1.5×10^{1}	7.2×10^{-2}	7.9
router	3.7	5.0×10^{1}	1.9×10^{-2}	3.5	1.8	5.8×10^{1}	1.4×10^{-2}	1.5	3.6	1.4×10^{1}	2.4×10^{-2}	1.2
sin	2.3×10^{2}	7.3×10^{2}	7.3×10^{-1}	1.2×10^{5}	1.1×10^{2}	8.8×10^{2}	4.6×10^{-1}	4.6×10^{4}	9.2×10^{1}	1.4×10^{2}	1.6	2.0×10^{4}
sqrt	1.7×10^{3}	5.5×10^{3}	5.3	5.0×10^{7}	8.3×10^{2}	6.6×10^{3}	3.2	1.8×10^{7}	6.3×10^{2}	1.0×10^{3}	1.1×10^{1}	7.0×10^{6}
square	1.0×10^{3}	3.3×10^{2}	2.8	9.4×10^{5}	4.6×10^{2}	4.0×10^{2}	1.5	2.8×10^{5}	1.4×10^{2}	5.4×10^{1}	3.2	2.5×10^{4}
voter	7.2×10^{2}	5.3×10^{2}	2.0	7.5×10^{5}	3.2×10^{2}	6.3×10^{2}	1.1	2.3×10^{5}	1.2×10^{2}	9.0×10^{1}	2.5	2.6×10^{4}
Averages	4.3×10^{2}	6.0×10^{2}	1.3	3.8×10^{6}	2.0×10^{2}	7.2×10^{2}	8.3×10^{-1}	1.3×10^{6}	1.1×10^{2}	1.1×10^{2}	2.2	4.7×10^{5}

Table 3.9 shows results of mapped MIG into STMG when optimized MIG are inverter-free and with restricted fan-out from Table 3.2. Inverter-free MIGs allow the realization of entirely STMG-based circuit which do not use hybrid solutions for inversion.

RS optimization techniques. To obtain area, delay, and energy, we use the same specifications as Tables 3.8 and 3.6. Our optimization method is able to further reduce the ADEP by 2.31% for QCA and by 2.07% for STMGs. Overall, the MIG-based synthesis flow obtains an average improvement of 20.81% and 55.63% for QCA and STMG, respectively.

3.7 Summary

Motivated by the requirements set by emerging nanotechnologies, we presented diverse algorithms for logic synthesis of MIGs. First, we introduced novel methods for size optimization of majority logic, focusing on both technology-independent results and mapped circuits. The flow uses three Boolean methods to reduce the size of MIG, obtaining a total reduction of the size of the MIG up to 18.13%, on average. Second, we focused on novel algorithms to synthesize inversion-free circuits, with limited fan-out. Such constraints are set by emerging nanotechnologies such as QCA and STMG. Our algorithms demonstrated the realization of inversion-free circuits and limited fan-out, with only 2× size increase of the MIG. Finally, we showed that existing exact synthesis algorithms can be adjusted to solve complex constraints problems. We implemented an algorithm that takes into account all the different constraints and we demonstrated that classical heuristic logic synthesis tools may lead to a solution that does not meet all the requirements or give results of inferior quality. In this scenario, plasmonic-based logic is responsible for this set of constraints. Our exact synthesis method illustrated that, on average, more than 86% of practical 6-LUTs can be efficiently realized with these constraints.

In the last part of the chapter, we mapped MIG networks into STMG and QCA to explore global trade-offs. Technology-dependent logic synthesis approaches can lead to further

Table 3.10 – Size optimization techniques (after QCA and STMG technology mapping)

Benchmark	Baseline [187] QCA				Opt. QCA				Baseline [187] STMG				Opt. STMG			
	Area [μm^2]	**Delay** [ns]	**Energy** [J]	**ADEP**	**Area** [μm^2]	**Delay** [ns]	**Energy** [J]	**ADEP**	**Area** [μm^2]	**Delay** [ns]	**Energy** [J]	**ADEP**	**Area** [μm^2]	**Delay** [ns]	**Energy** [J]	**ADEP**
adder	1.6	0.5	4.0E-18	3.5E-18	1.6	0.5	4.0E-18	3.5E-18	15.4	193.5	5.4E-11	1.6E-07	15.4	193.5	5.4E-11	1.6E-07
arbiter	12.8	0.3	3.1E-17	1.1E-16	12.8	0.3	3.1E-17	1.1E-16	35.2	94.5	5.9E-10	2.0E-06	35.2	94.5	5.9E-10	2.0E-06
bar	7.9	0.1	1.9E-17	1.1E-17	7.8	0.1	1.9E-17	1.0E-17	21.9	21.0	7.2E-10	3.3E-07	21.9	19.5	7.1E-10	3.0E-07
cavlc	1.4	0.1	3.3E-18	4.1E-19	1.3	0.1	3.2E-18	3.7E-19	4.0	28.5	1.3E-10	1.5E-08	3.8	28.5	1.2E-10	1.3E-08
ctrl	0.3	0.1	6.2E-19	8.6E-21	0.2	0.1	5.7E-19	6.6E-21	0.7	15.0	2.3E-11	2.4E-10	0.6	13.5	2.1E-11	1.8E-10
dec	0.4	0.0	1.1E-18	1.4E-20	0.4	0.0	1.1E-18	1.4E-20	1.2	6.0	2.8E-11	2.0E-10	1.2	6.0	2.8E-11	2.0E-10
i2c	2.6	0.1	6.5E-18	1.8E-18	2.5	0.1	6.1E-18	1.6E-18	6.9	34.5	2.1E-10	5.0E-08	6.6	34.5	2.0E-10	4.6E-08
int2float	0.5	0.1	1.2E-18	5.0E-20	0.5	0.1	1.2E-18	4.3E-20	1.4	24.0	4.8E-11	1.6E-09	1.3	24.0	4.4E-11	1.4E-09
log2	60.0	0.9	1.5E-16	8.2E-15	59.9	0.9	1.5E-16	8.2E-15	179.6	345.0	2.1E-09	1.3E-04	179.3	343.5	2.0E-09	1.2E-04
max	7.4	1.1	1.8E-17	1.4E-16	7.3	1.1	1.8E-17	1.4E-16	30.7	391.5	4.3E-10	5.2E-06	30.7	390.0	4.3E-10	5.1E-06
mem	92.4	0.6	2.3E-16	1.2E-14	86.3	0.6	2.1E-16	1.0E-14	265.2	216.0	6.5E-09	3.7E-04	247.0	204.0	6.0E-09	3.0E-04
mult	47.7	0.6	1.2E-16	3.3E-15	47.7	0.6	1.2E-16	3.2E-15	141.6	214.5	2.2E-09	6.6E-05	141.6	211.5	2.2E-09	6.5E-05
priority	2.6	1.0	6.5E-18	1.7E-17	2.6	1.0	6.3E-18	1.6E-17	7.7	367.5	1.8E-10	5.0E-07	7.7	358.5	1.7E-10	4.8E-07
router	0.7	0.2	1.8E-18	3.1E-19	0.7	0.2	1.8E-18	2.9E-19	3.6	81.0	4.2E-11	1.2E-08	3.6	81.0	4.1E-11	1.2E-08
sin	10.7	0.7	2.6E-17	1.9E-16	10.6	0.7	2.6E-17	1.8E-16	31.8	243.0	3.5E-10	2.7E-06	31.6	240.0	3.3E-10	2.5E-06
sqrt	51.0	24.0	1.3E-16	1.5E-13	50.8	23.8	1.2E-16	1.5E-13	151.6	8983.5	1.6E-09	2.2E-03	150.9	8911.5	1.6E-09	2.2E-03
square	32.0	0.5	7.8E-17	1.3E-15	31.0	0.5	7.6E-17	1.3E-15	95.1	195.0	1.6E-09	3.0E-05	92.3	193.5	1.6E-09	2.8E-05
voter	19.9	0.3	4.9E-17	2.4E-16	17.6	0.2	4.3E-17	1.6E-16	60.1	88.5	6.7E-10	3.6E-06	60.1	73.5	5.8E-10	2.6E-06
total reduction				+18.50%				+20.81%				+53.56%				+55.63%
improvement				0.00%				+2.31%				0.00%				+2.07%

Table 3.10 shows results of mapped MIG into QCA and STMG when MIG are size-optimized. Inversion-free technique is applied to remove inversions (and thus realize STMG circuits). The ADEP is reduced for both technologies.

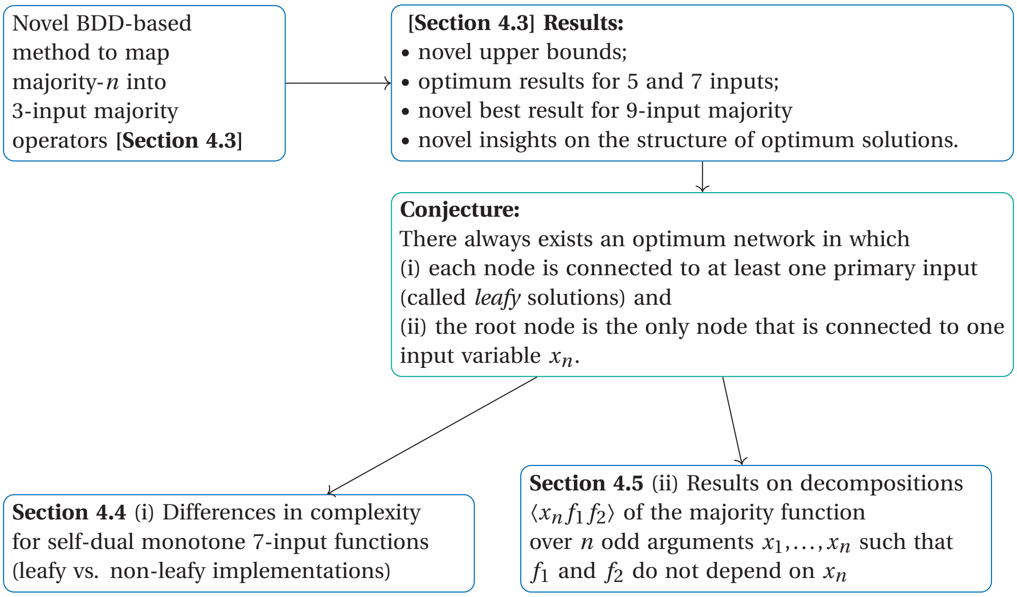
improvements for many emerging nanotechnologies. Inversion-free MIGs with limited fan-out result in a better ADEP for both STMG and QCA. In particular, for the STMG technology which cannot realize inversion gates, our synthesis method for inversion-free networks demonstrated the realization of circuits entirely composed of STMG cells, i.e., they do not need CMOS logic to implement the complementation. The ADEP is decreased of 3.1× on average also in the QCA case. As a last result, we presented mapped results of size-optimized MIG over QCA and STMG, demonstrating a reduction in the ADEP also for this class of algorithms which do not specifically take into account emerging technologies requirements, while reducing the number of nodes in technology-independent MIGs.

4 Majority-n Logic

> *I think most of the important things that I know have been the result of learning from mistakes.*
> — Private email from D. E. Knuth

Chapter 3 was dedicated to logic synthesis techniques over *majority-inverter graph*s (MIGs). In this chapter, we continue the study of majority logic, but we shift into more theoretical results. In particular, the chapter concentrates on size-optimum networks for monotone functions, and principally majority-n functions (i.e., majority with n inputs), using majority-of-three gates. The chapter is divided into three parts addressing different and relevant questions on such topic. First, we present a novel method to map majority-n into MIG, together with novel upper bounds on the number of gates. Then, we study the *complexity* of 7-input self-dual monotone functions, to give more insights on the characteristic of such class of functions. Note that the majority function is both self-dual and monotone. Finally, we conclude with more theoretical results on the decomposition of majority-n into majority-3.

The remainder of this chapter is organized as follows. Section 4.1 illustrates the motivation for this work, while Section 4.2 provides the notation used in this chapter on *binary decision diagram*s (BDDs) and majority graphs. Note that both topics have been previously introduced in Chapter 2. This section aims instead at explaining the notation and highlighting important features needed in the chapter. In Section 4.3, a novel BDD-based method to map majority-n functions into 3-input majority is presented, together with novel upper bounds. In particular, our method is able to obtain the optimum results for the majority-5 and the majority-7; while demonstrating the best-known result for the majority-9. The section is largely based on the journal publication in [177]. Section 4.4 addresses theoretical study of self-dual monotone functions over 7-input. This class of function is important as it includes the majority-n functions. Both the complexity and the length are studied in detail as already presented in the paper [172]. Section 4.5 is largely based on the publication in [166] and proposes novel results on the decomposition of majority-n into majority-3. This chapter is summarized in Section 4.6. An overview of the chapter is shown in Figure 4.1.


Novel BDD-based method to map majority-n into 3-input majority operators **[Section 4.3]**

[Section 4.3] Results:
- novel upper bounds;
- optimum results for 5 and 7 inputs;
- novel best result for 9-input majority
- novel insights on the structure of optimum solutions.

Conjecture:
There always exists an optimum network in which
(i) each node is connected to at least one primary input (called *leafy* solutions) and
(ii) the root node is the only node that is connected to one input variable x_n.

Section 4.4 (i) Differences in complexity for self-dual monotone 7-input functions (leafy vs. non-leafy implementations)

Section 4.5 (ii) Results on decompositions $\langle x_n f_1 f_2 \rangle$ of the majority function over n odd arguments $x_1, \ldots, x_n$ such that f_1 and f_2 do not depend on x_n

Figure 4.1 – Chapter organization

4.1 Motivation

The majority-of-three function $\langle xyz \rangle$, which is true if and only if at least two of its inputs are true, plays an important role in the digital design of circuits using emerging nanotechnologies, as extensely demonstrated in Chapter 3. Moreover, it has been shown that many arithmetic and Boolean operations, such as addition, multiplication, and division, are contained in the complexity class TC_0, which means that they have an efficient realization with polynomial size and constant depth networks, using unbounded majority gates [88, 146]. For instance, it is possible to build a n-bit adder with depth = 2 [11, 146] and iterated multiplication of two numbers with depth = 4 [88, 90]. It is then possible to realize arithmetic addition, division, and multiplication with polynomial size and constant depth, built using only unbounded-fanin majority gates and inverters [7]. Note that the arity of each majority is unbounded. These results may serve as a good starting point for circuit realizations when using majority-based nanotechnologies, provided that one is equipped with a technique to express majority-n functions in terms of majority-of-three. In other words, as many emerging technologies only provide the realization of 3-input majority, an effective realization of majority-n functions in terms of MIG is required. Moreover, the study of majority-n functions properties and decompositions into smaller-input blocks play also a fundamental role for the development of plasmonic-based technologies (as the ones in [72] and Chapter 3) that can efficiently implement 3-, 9-, and 27-input majority functions.

The problem of expressing majority-n using majority-of-three has already been studied in the 1960s [12, 189]. In [12], Amarel et al. investigated "*how best can the 5-argument majority function be realized with a network of 3-input majority gates?*". The focus was on finding the minimum number of 3-input majorities to build majority-5, but larger n were also considered [12]. In the remainder, we will refer to the minimum number of majority-of-three to build a majority-n as $M(n)$. It is surprising that, as of today, $M(n)$ is known only for 5- and 7-input majorities, while the minimum realization of majority-9 (and larger n) in terms of majority-3 is still under investigation. Other works have focused on $M(n)$, but they have only considered its asymptotic bounds [103]. The asymptotic complexity for $M(n)$ is linear, since one can reduce it to median selection [70]. However, when applying the construction to small values for n, the resulting majority graphs are still very large. For example, the majority-7 function can be constructed using 42 majority-3 operations according to median selection construction, while it is known that $M(7) = 7$. Sorter networks provide an alternative construction that provides a quasi-linear bound [61]. However, for small n the construction can yield better results compared to median selection. To follow up with the previous example, the majority-7 function can be constructed using 32 majority-3 operations based on the sorter network construction.

In Section 4.3, we focus on finding the minimum number of majority-3 operations to express majority-n. In particular, an alternative construction based on BDDs [46] is proposed and we show that for monotone Boolean functions the Shannon decomposition, which is used in the construction of BDDs, can be expressed using the majority function. This leads to a one-to-one translation of BDDs into majority graphs, which are logic networks in which all operations are majority-3 functions. Since majority-n is monotone, we can use the construction as an upper bound on the number of majority-3 operations. This bound is asymptotically quadratic, however, for small n it leads to much smaller values compared to the constructions based on median selection and sorter networks. For instance, for majority-7 the construction leads to a realization with 15 majority-3 operations. Therefore, in order to find small realizations for majority-n networks, for small n, the proposed BDD based approach can be a more effective starting point. Further, we can derive the known optimum results for majority-5 and majority-7 starting from the proposed BDD construction by applying well-known algebraic properties of the majority function and two new identities. We apply the same procedure to majority-9, and it is shown that significant optimization can be obtained leading to a new best-known solution with 12 majority-3 operations. Majority-7 is the largest majority-n function for which a minimum size solution is known. The exact solution was found by using exhaustive search using a SAT-solver (see, e.g., [162]). Here, the entire derivation is explained in detail; this can provide insight into the decomposition of larger majority functions and may help to find values for $M(n)$ where $n \geq 9$.

The structure of the optimum majority networks obtained using the BDD method has led to the following conjecture, which does not predict the number of minimum operations, but predicts a common structure.

Conjecture 4.1 *There always exists an optimum network in which (i) each node is connected to at least one* primary input *(PI) (*leafy *solution) and (ii) the root node is the only node that is connected to x_n.*

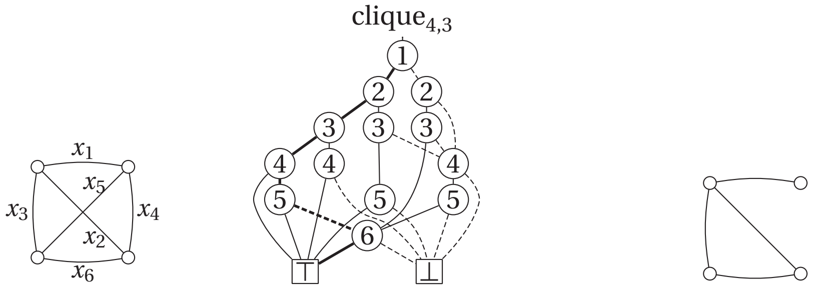
In the second part of the chapter, we thus augment the previously described results with other two sets of results: (i) a study of self-dual monotone 7-input functions over 3-input majority operators, focusing on the comparison of their complexity for the leafy and the non-leafy case; and (ii) novel results on decompositions $\langle x_n f_1 f_2 \rangle$ of the majority function over n odd arguments $x_1, \ldots, x_n$ such that f_1 and f_2 do not depend on x_n.

In Section 4.4, all self-dual monotone 7-input functions are enumerated and classified according to their *length* $L(f)$ and their *combinational complexity* $C(f)$ over different majority Boolean chains (detailed in Section 4.4.1). The former is defined as the minimum number of operators in the shortest formula of a Boolean function, while the latter is the minimum length of its Boolean chain. There are 1,422,564 7-input self-dual monotone functions, distributed over 716 classes according to input permutation (P-equivalence). The classification over majority network has first been presented by Knuth in [97], both according to the complexity and the length. Here, we use our own implementation of the algorithm from [97] to enumerate all 1,422,564 functions and to reproduce their classification according to their length. On the other hand, we propose our own strategy to compute the combinational complexity for all 716 classes, which is a SAT-based exact synthesis method. By comparing our results with the ones in [97], we confirm that, when using majority Boolean chains, the largest $L(f)$ is 11, while the largest $C(f)$ is 8. Moreover, we also present results on leafy majority Boolean chains, focusing on the differences with respect to the majority case. We demonstrate an increase in the maximum length, while we show that the worst combinational complexity remains unaffected. However, the combinational complexity does not remain unchanged for all functions, i.e., 40 functions have increased combinational complexity when built using leafy majority chains. As a last result, we show that inverters have an impact on the minimum majority-network of self-dual monotone functions. It is theoretically known that inverters can decrease complexity even in monotone functions [89], here we demonstrate and give concrete examples for the complexity of 7-input functions.

In Section 4.5, we present results on decompositions $\langle x_n f_1 f_2 \rangle$ of the majority function over n odd arguments $x_1, \ldots, x_n$ such that f_1 and f_2 do not depend on x_n. In particular, we derive the conditions for f_1 and f_2 that satisfy the decomposition. Such decompositions play a central role in finding optimum majority-3 networks for the majority-n function and integrate the previous set of theoretical results.

4.2 Preliminaries

This section describes the terminology used in the chapter for the BDDs and the majority-graphs. We refer the reader to Chapter 2 for further background and to [97] for more details on the notation.

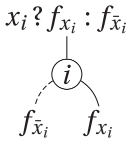


(a) K_4 graph (b) BDD (c) Graph instance with 3-clique

Figure 4.2 – Diagrammatic notation of a binary decision diagram for the function $\text{clique}_{4,3}$. (a) is the graph for K_4, while (b) is the BDD. The '⊥' and '⊤' represent the constant functions 0 and 1 [97]. (c) is the graph instance with $x_4 = x_5 = 0$. © 2018 IEEE [177]

4.2.1 Binary Decision Diagrams

Recall that BDDs [46] are connected directed acyclic graphs, in which each node represents a Boolean function. Two terminal nodes labeled '⊥' and '⊤' represent the constant functions 0 and 1, and all nonterminal nodes are labeled 'i' for some $1 \le i \le n$ and connect their two successor nodes f_{x_i} and $f_{\bar{x}_i}$



(4.1)

using Shannon's decomposition:

$$f = x_i \mathbin{?} f_{x_i} : f_{\bar{x}_i} = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i} \tag{4.2}$$

The cofactors f_{x_i} and $f_{\bar{x}_i}$ are the functions that are obtained by replacing x_i with 1 and 0 in f, respectively. Each BDD has one node without parents representing the function, called the root. Note that in the following, we use the term BDD to refer to an ordered and reduced BDD.

We will use the Boolean function $\text{clique}_{n,k}$ (see for example [92]) as a running example throughout the section. The function has $\binom{n}{k}$ variables, each of which representing an edge in the complete graph K_n over n variables. For example, the 6 variables representing edges in K_4 are shown in Figure 4.2(a). A variable assignment corresponds to a particular graph instance, in which an edge exists if the corresponding variable is set to true, and is absent otherwise. We have $\text{clique}_{n,k}(x) = 1$, if and only if the graph instance corresponding to the assignment x contains a k-clique, which is a fully connected sub-graph of k vertices.

Example 4.1 *Figure. 4.2(b) shows a BDD for* $\text{clique}_{4,3}$ *with the variable ordering* $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. *The highlighted path represents the graph instance which assigns* $x_1 = x_2 = x_3 = x_6 = 1$ *and* $x_4 = x_5 = 0$ *(see Figure 4.2(c)). This graph instance contains a* 3*-clique on edges* x_2,

x_3, and x_6, thus the function is equal to 1, i.e., the BDD path terminates in node '$\top$'. ■

When working with BDDs, it is often more convenient to represent a BDD of a Boolean function over n variables as a sequential list of branch instructions $I_{s-1}, I_{s-2}, \ldots, I_1, I_0$, where each I_k has the form $(v_k ? h_k : l_k)$ [97]. In this notation, v_k is the label of the node, and $h_k < k$ and $l_k < k$ are indexes to other branch instructions, called *high* and *low*, respectively. Instructions $I_1 = (n+1 ? 1 : 1)$ and $I_0 = (n+1 ? 0 : 0)$ are special instructions to represent the terminal nodes. They have as vertex labels the "*impossible*" value $n+1$, which is not used in any of the other steps.

Example 4.2 *The BDD in Figure 4.2(b) has the following sequential list of branch instructions:*

$$\begin{aligned}
I_{14} &= (1?13:12), & I_{13} &= (2?11:10), & I_{12} &= (2?9:6),\\
I_{11} &= (3?8:7), & I_{10} &= (3?4:6), & I_9 &= (3?2:6),\\
I_8 &= (4?1:5), & I_7 &= (4?1:0), & I_6 &= (4?3:0),\\
I_5 &= (5?1:2), & I_4 &= (5?1:0), & I_3 &= (5?2:0),\\
I_2 &= (6?1:0), & I_1 &= (7?1:1), & I_0 &= (7?0:0).
\end{aligned}$$

For example, I_{14} refers to the root node labeled 1, and I_2 refers to the only node labeled 6. ■

4.2.2 Majority Graphs

The mapping approach, which we will present in Section 4.3.1, changes BDDs into majority graphs. We discuss here the notation used in the rest of the chapter.

A *majority graph* is a directed acyclic graph in which each terminal node is a PI or a constant and each nonterminal node represents a majority operation with three incoming edges. Formally, given a Boolean function $f(x_1, \ldots, x_n)$, it is most convenient to represent a majority graph with r majority gates as a chain $x_{n+1}, \ldots, x_{n+r}$, where

$$x_i = \langle x_{j(i)} x_{k(i)} x_{l(i)} \rangle \qquad \text{for } n < i \leq n+r, \tag{4.3}$$

with $-1 \leq j(i) < i$, $-1 \leq k(i) < i$, and $-1 \leq l(i) < i$. We also define $x_0 = \bot$ and $x_{-1} = \top$.

We call a majority graph *leafy* if $j(i) \leq n$, $k(i) \leq n$, or $l(i) \leq n$ for all $n < i \leq n+r$. In other words, in each majority operation at least one operand is an input variable.

Example 4.3 *Figure 4.3 shows the majority graph for the Boolean function* $\text{clique}_{4,3}$. *$x_1, \ldots, x_6$ are the PIs, while each node $x_7, \ldots, x_{15}$ represents the majority-of-three function. The majority*

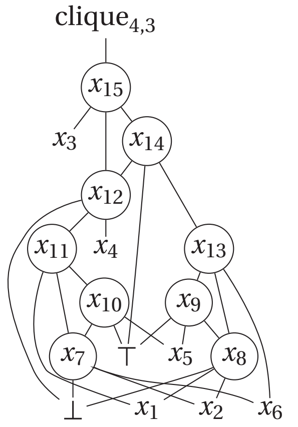


Figure 4.3 – Majority graph for the function clique$_{4,3}$ © 2018 IEEE [177]

nodes are given by:

$$\begin{aligned} x_7 &= \langle x_0 x_2 x_6 \rangle, & x_8 &= \langle x_0 x_1 x_2 \rangle, & x_9 &= \langle x_{-1} x_5 x_8 \rangle, \\ x_{10} &= \langle x_{-1} x_5 x_7 \rangle, & x_{11} &= \langle x_1 x_7 x_{10} \rangle, & x_{12} &= \langle x_0 x_4 x_{11} \rangle, \\ x_{13} &= \langle x_6 x_8 x_9 \rangle, & x_{14} &= \langle x_{-1} x_{12} x_{13} \rangle, & x_{15} &= \langle x_3 x_{12} x_{14} \rangle. \end{aligned}$$

■

4.3 Decomposing Majority-n into Majority-3

In this section, first, we present a novel method based on BDDs to map majority-n functions into majority-3 and illustrate how the proposed construction can be used as an upper bound on the number of majority-3 operations. Then, we show that we can derive the known optimum results for majority-5 and majority-7 and a novel best-known result for the majority-9, when starting from the proposed BDD construction.

4.3.1 Transforming BDDs into Majority Graphs

As discussed in Section 4.2, one can express any Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ in terms of its cofactors using (4.2). Recall that (4.2) states:

$$f = x_i \,?\, f_{x_i} : f_{\bar{x}_i} = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$$

If f is monotone, i.e., $f_{\bar{x}_i} \bar{f}_{x_i} = 0$ for all i, it is possible to use the majority function for decomposition [9]:

$$f = \langle x_i f_{x_i} f_{\bar{x}_i} \rangle = x_i f_{x_i} \oplus x_i f_{\bar{x}_i} \oplus f_{x_i} f_{\bar{x}_i} \tag{4.4}$$

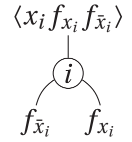
Remark 4.1 *The cofactors* f_{x_i} *and* $f_{\bar{x}_i}$ *commute in* (4.4)*, but they do not in* (4.2)*.*

Remark 4.2 *The cofactor operation preserves monotonicity in* (4.2)*. Thus,* (4.4) *can iteratively be applied to the whole BDD.*

Due to these properties,

$$f = x_i \,?\, f_{x_i} : f_{\bar{x}_i} = \langle x_i f_{x_i} f_{\bar{x}_i} \rangle \tag{4.5}$$

and one can replace each "Shannon node" by a "majority node" in the BDD of monotone functions:



(4.6)

Remark 4.3 *Since* f_{x_i} *and* $f_{\bar{x}_i}$ *commute, it is no further necessary to distinguish between the two successors in the BDD node. Also, since the resulting majority graph is leafy, we can use a notation analogous to the BDD notation, in which the node is labeled by the variable operand* x_i*. Note that the nodes in BDD and majority graphs use the same notation and have the same shape, but they differ in the operation they implement (i.e., if-then-else and majority, respectively). They also differ in how terminal nodes are drawn.*

The replacement allows us to generate majority graphs for monotone functions $f(x_1, \dots, x_n)$ directly from their BDDs using the following simple transformation rule. Let the BDD for f be represented using a sequential list of branch instructions $I_{s-1}, \dots, I_0$ as described in Example 4.2. Then a majority graph for f is

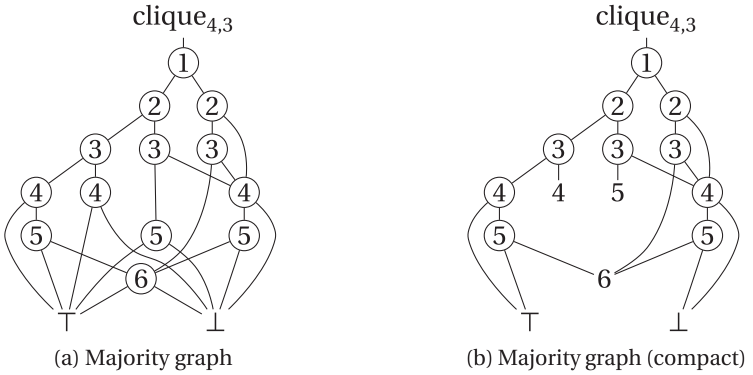
$$x_{n+k-1} = \langle x_{v_k} x_{t(h_k)} x_{t(l_k)} \rangle \quad \text{where } I_k = (v_k \,?\, h_k : l_k) \tag{4.7}$$

for $2 \le k < s$. In (4.7) the index transformation function $t : [0, s-1] \mapsto [-1, n+s-2]$ is defined as

$$t(i) = \begin{cases} 0 & \text{if } i = 0, \\ -1 & \text{if } i = 1, \\ n+i-1 & \text{otherwise.} \end{cases} \tag{4.8}$$

Example 4.4 *We show how to apply the transformation to the BDD in Figure 4.2(b). By just translating every BDD node into a MAJ node as in* (4.6)*, one obtains the majority graph depicted in Figure 4.4(a). Note that we removed the boxes around the terminal nodes, mainly to further help distinguishing the two representations.*

Since $\langle x_i 01 \rangle = x_i$*, we write nodes that have two constant successors also as terminal nodes. This applies to three nodes in the graph. Figure 4.4(b) shows the compact version.* ■



(a) Majority graph (b) Majority graph (compact)

Figure 4.4 – Diagrammatic notation of a leafy majority graph for the function clique$_{4,3}$ © 2018 IEEE [177]

We prove some properties of the transformation.

Theorem 4.1 *The sequence of majority operations derived from* (4.7) *is a leafy majority graph.*

Proof 4.1 *By construction, we have* $v_k \le n < n+k-1$. *This implies that the graph is leafy. Also,* $n+h_k-1 < n+k-1$, *since* $h_k < k$. *The same applies to* l_k. ■

Theorem 4.2 *The majority graph represents* f.

Proof 4.2 *The last step in the majority graph is* x_{n+s-2} *which corresponds to branch instruction* I_{s-1} *of the BDD. Therefore, due to* (4.5),

$$f = x_{n+s-2} \overset{(4.5)}{=} \langle x_{v_{s-1}} x_{t(h_{s-1})} x_{t(l_{s-1})} \rangle$$

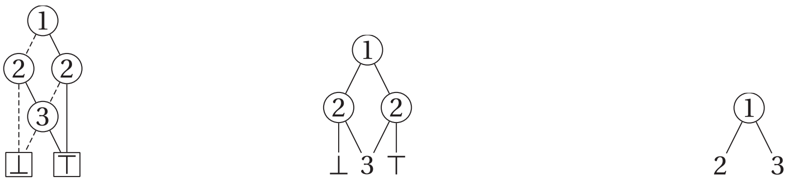
where

$$I_{s-1} = (v_{s-1}\,?\,h_{s-1} : l_{s-1})$$

The rest follows from induction. When $i = 1, 0$, *the instructions are* $I_1 = (n+1\,?\,1 : 1)$ *and* $I_0 = (n+1\,?\,0 : 0)$, *respectively, which hold by construction. Let us suppose it holds for all* I_i, *with* $2 \le i \le k$. I_{k+1} *has the form* $(v_{k+1}\,?\,h_{k+1} : l_{k+1})$. v_{k+1} *is the label of the node, and* $h_{k+1} < k+1$ *and* $l_{k+1} < k+1$ *are indexes to other branch instructions, which can only be instructions* I_i *with* $2 \le i \le k$. *Then it follows that also* I_{k+1} *holds, and this concludes the proof.* ■

Discussion on complementation

When reducing the size of the resulting majority graph (which has not complemented edges), some transformation rules may introduce complemented edges in the majority graph and transform it into a MIG [16]. MIGs are majority graphs which make use of complemented



(a) BDD (b) Not optimized majority-3 (c) Optimum result

Figure 4.5 – Majority-3 from its optimal BDD. (a) is the BDD, while (b) is the network obtained by the direct mapping of the BDD into the MIG. The optimum result is shown in (c). © 2018 IEEE [177]

edges to represent inversions. In the next section, which deals with deriving optimum majority graphs for majority-5 and majority-7, we do make use of complemented edges to PIs and we call them optimum MIGs. However, we also present solutions of the same size that do not require complemented edges.

Discussion on optimality

In this section, we answer the question whether a size-optimum BDD produces a size-optimum MIG. Using a simple counter-example, the majority-3 function, we show that this is not the case. In general, the size of a BDD depends solely on the variable order. Therefore, there are one or more variable orders that lead to the smallest BDD. The next section shows that the optimum BDD for the majority-3 function requires 4 nodes, and therefore the direct transformation into a majority graph requires 3 majority gates. This simple example shows that this MIG is far from the optimal; and since the majority-3 function is symmetric (see, e.g., [97]), it is not affected by the variable order. Optimal MIG representations can be achieved by making use of algebraic transformation rules and identities presented in Chapter 2. It has been proven in [16] that any other functionally equivalent MIG can be reached using a finite sequence of transformation rules. It is worth noting that these identities and rules reshape the MIG in a way such that the inverse transformation (from MIG to BDD) is not always possible (see, e.g., majority-5 optimum result given in the next section).

4.3.2 Mapping Majority-n

In this section, we illustrate how to use the proposed approach to map the majority-n function into MIGs with majority-3 operations. For example, for the majority-3 function $\langle x_1 x_2 x_3 \rangle$, the BDD and its corresponding majority graph are shown in Figure 4.5(a) and (b), respectively. The majority expression that Figure 4.5(b) represents is $\langle x_1 \langle x_2 0 x_3 \rangle \langle x_2 1 x_3 \rangle \rangle$, which is of course far from optimal. The distributivity rule applies to this expression, giving $\langle \langle x_1 0 1 \rangle x_2 x_3 \rangle = \langle x_1 x_2 x_3 \rangle$; its diagrammatic notation is shown in Figure 4.5(c).

This example is rather trivial. Next, we show that we can rewrite majority-5 and majority-7 into their optimum MIGs using the identities presented in Chapter 2 and new identities which will be described next. We present majority-5 and majority-7 optimization in detail in order to (i) demonstrate that our method leads to the optimum known results, and (ii) to illustrate the complete optimization procedure. Then, we show that the optimization procedure can be generalized for larger *n* by showing the optimized majority-9, leading to a new best solution with 12 majority-3 operations.

Replacement rule and swapping rule

The replacement rule describes under which condition one operand in a majority expression can be replaced by another one. Note that this rule was used in Chapter 3 for size optimization of MIGs.

Theorem 4.3 (Replacement rule) *We have*

$$\langle xyz\rangle = \langle wyz\rangle \quad \textit{if and only if } (y\oplus z)(w\oplus x) = 0,$$

or in other words $y \neq z \Rightarrow w = x$.

Proof 4.3 *First note that* $\langle xyz\rangle = x(y\oplus z)\oplus yz$. *Then*

$$\begin{aligned} 0 &= \langle xyz\rangle \oplus \langle wyz\rangle \\ &= x(y\oplus z)\oplus yz \oplus w(y\oplus z)\oplus yz \\ &= x(y\oplus z)\oplus w(y\oplus z) = (w\oplus x)(y\oplus z), \end{aligned}$$

which concludes the proof. An alternative way to see that the theorem is true, is by applying the majority law to y *and* z. ■

One can readily verify that the relevance rule presented in [14] is a special case of the replacement rule.

Corollary 4.1 (Relevance rule) *We have* $\langle xyz\rangle = \langle x_{y/\bar{z}}yz\rangle$, *where* $x_{y/\bar{z}}$ *is obtained by replacing all occurrences of* y *with* $\bar{z}$ *in* x.

Relevance rule is a special case of the replacement rule when $w = x_{y/\bar{z}}$. It can be easily shown that the condition $(y\oplus z)(w\oplus x) = 0$ is always met. In fact, when $y = z$, $(y\oplus z) = 0$; when $y \neq z = \bar{z}$, $(w\oplus x) = (x_{y/\bar{z}}\oplus x) = (x\oplus x) = 0$.

The swapping rule describes when two operands in a majority expression can be swapped between them.

Theorem 4.4 (Swapping rule) *Let* v_1, v_2, w_1, w_2 *not depend on* x *and* y. *We have* $\langle x\langle yv_1w_1\rangle\langle yv_2w_2\rangle\rangle = \langle x\langle yv_2w_1\rangle\langle yv_1w_2\rangle\rangle$, *if* $(v_1\oplus v_2)(w_1\oplus w_2) = 0$.

In other words, the swapping rule describes a condition in which the subfunctions v_1 and v_2 can be swapped. Due to commutativity, one can also swap w_1 with w_2, or both.

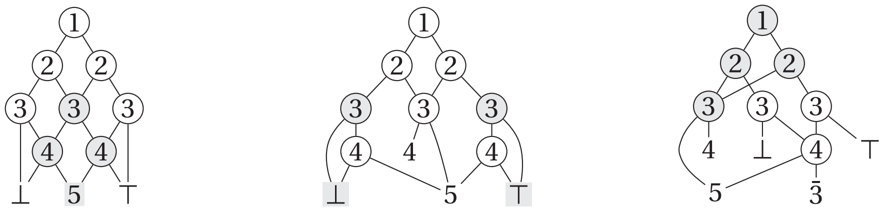
Proof 4.4 *From the condition in Theorem 4.4* $(v_1 \oplus v_2)(w_1 \oplus w_2) = 0$*, it follows that we have either* $(v_1 = v_2)$ *or* $(w_1 = w_2)$*. Therefore, one of the following cases is true.*

Case $(v_1 = v_2)$*: Then* $\langle x \langle y v_1 w_1 \rangle \langle y v_1 w_2 \rangle \rangle = \langle x \langle y v_1 w_1 \rangle \langle y v_1 w_2 \rangle \rangle$ *is trivially true.*

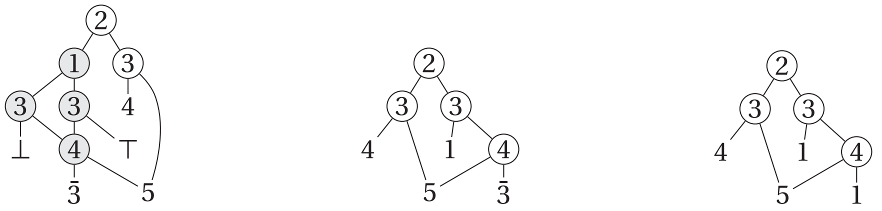
Case $(w_1 = w_2)$*: Then* $\langle x \langle y v_1 w_1 \rangle \langle y v_2 w_1 \rangle \rangle = \langle x \langle y v_2 w_1 \rangle \langle y v_1 w_1 \rangle \rangle$ *due to commutativity.* ■

These new rules will be used in the next section to derive the optimum solution for majority-5 and majority-7. However, they can be employed in more general majority-based optimization.

Mapping majority-5



(a) Distributivity (b) Relevance (c) Distributivity



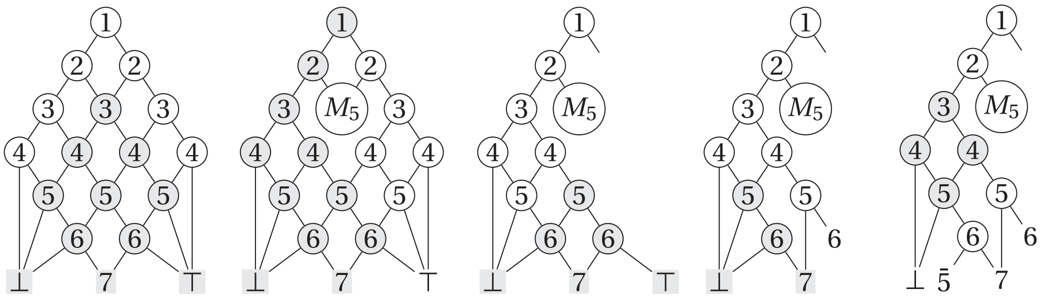
(d) Distributivity (e) Optimum result with complemented edges (f) Optimum result without complemented edges

Figure 4.6 – Decomposing majority-5 into majority-3. (a) is the optimal BDD, the final result (optimum) is obtained by applying distributivity (a), relevance (b), distributivity 2× (c)-(d). Both optimum results with (e) and without (f) complemented edges are shown. © 2018 IEEE [177]

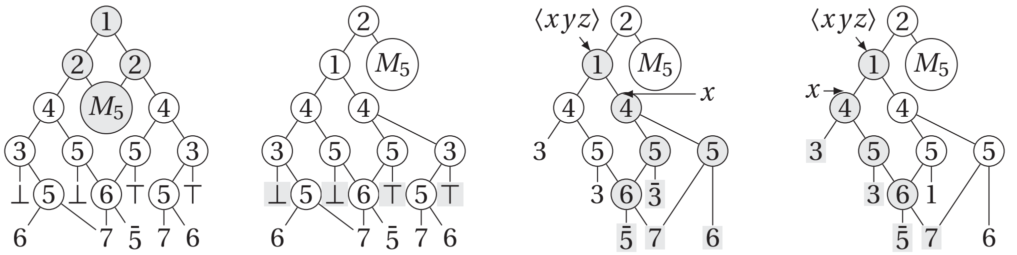
We now show how to use these diagrams for the majority decomposition of the majority-5 expression $\langle x_1 x_2 x_3 x_4 x_5 \rangle$. The starting point derived from a BDD similar to the majority-3 case is shown in Figure 4.6(a). Recall also that the distributivity rule states:

$$\langle x u \langle y v z \rangle \rangle = \langle \langle x u y \rangle v \langle x u z \rangle \rangle \tag{4.9}$$

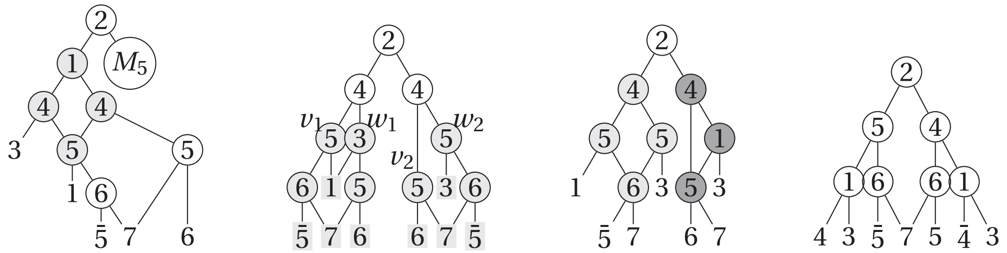
First, distributivity is applied to the gray nodes to change $\langle x_3 \langle x_4 0 x_5 \rangle \langle x_4 1 x_5 \rangle \rangle$ into $\langle x_3 x_4 \langle x_5 0 1 \rangle \rangle = \langle x_3 x_4 x_5 \rangle$. However, since the nodes labeled '4' have other ingoing edges, these nodes need to be preserved. The resulting network is shown in Figure 4.6(b). Relevance rule is applied on the gray nodes: $\langle x_3 0 \langle x_4 0 x_5\rangle\rangle = \langle x_3 0 \langle \bar{x}_3 x_4 x_5\rangle\rangle$ and $\langle x_3 1 \langle x_4 1 x_5\rangle\rangle = \langle x_3 1 \langle \bar{x}_3 x_4 x_5\rangle\rangle$. This allows us to replace $\perp$ and $\top$ by $\bar{x}_3$ for the gray nodes, thereby making the two nodes labeled '4' structurally equal, as shown in Figure 4.6(c). After again applying the distributivity law twice, one obtains the final network in Figure 4.6(e). This network has 4 nodes, which is optimum. This result has been demonstrated to be optimum using SAT-based exact synthesis (see Chapter 2). In particular, by demonstrating that a network for the majority-5 with 3 gates is unfeasible (UNSAT).



(a) Identify majority-5 (b) Left branch (c) Identify majority-3 (d) Relevance (e) Distributivity



(f) Distributivity (g) Remove ⊤ and ⊥ (h) Replacement rule (i) Replacement rule



(j) Distributivity + M_5 optimum (k) Swapping rule (l) Distributivity (gray) and relevance (dark gray) (m) Optimum result

Figure 4.7 – Decomposing majority-7 into majority-3. After identifying the majority-3 and -5 (a-c), the graph is optimized by doing: relevance (d), distributivity ×2 (e)-(f), removing ⊤ and ⊥ (g), replacement rule ×2 (h)-(i), distributivity and changing the M_5 with its optimum realization from Figure 4.6 (j), swapping (k), distributivity and relevance (l). The optimum result is shown in (m) © 2018 IEEE [177]

The final expression is

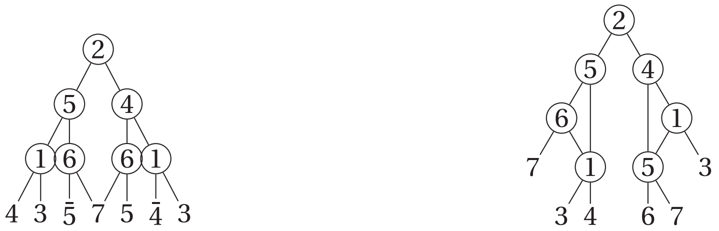
$$\langle x_2 \langle x_3 x_4 x_5\rangle \langle x_1 x_3 \langle \bar{x}_3 x_4 x_5\rangle\rangle\rangle \tag{4.10}$$

Note that an alternative expression without complemented edges can be obtained by applying the relevance rule on Figure 4.6(e). The final expression without complemented edges is

$$\langle x_2 \langle x_3 x_4 x_5 \rangle \langle x_1 x_3 \langle x_1 x_4 x_5 \rangle\rangle\rangle \tag{4.11}$$

This is shown in Figure 4.6(f).

Mapping majority-7

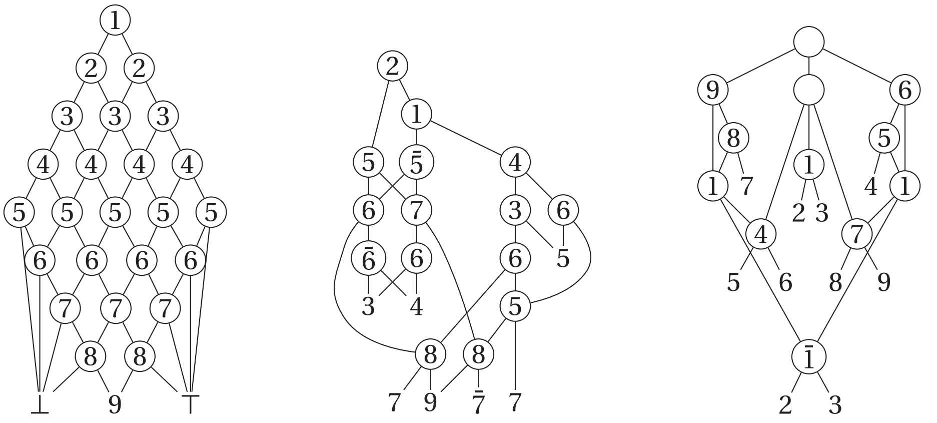


(a) Optimum majority-7, with complemented edges

(b) Optimum majority-7, without complemented edges

Figure 4.8 – Optimum majority-7 with (a) and without (b) complemented edges, respectively © 2018 IEEE [177]

Here, we decompose the majority-7 expression $\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 \rangle$ using an approach similar to the one employed for majority-5, and we demonstrate that the optimum known result can be obtained with our methodology. Figure 4.7(a) shows the starting point, derived from its BDD as done for the majority-3 in Figure 4.5. First, the majority-5 graph is identified (gray in Figure 4.7(a)) and written as node $M_5 = \langle x_3 x_4 x_5 x_6 x_7 \rangle$; its expression will be used again later. From Figure 4.7(b) to 4.7(e), only the left branch of node labeled '1' is considered, but it is worth noting that all steps are applied in the same way also to its right branch. First, the majority-3 highlighted in Figure 4.7(c) is changed into its optimum result, then the relevance rule is applied on the gray colored nodes of Figure 4.7(d), allowing the replacement of 0 with $\bar{x}_5$. Further, the distributivity rule is applied on Figure 4.7(e). The same procedure (Figure 4.7(b) to 4.7(e)) works for the right branch and the complete graph is shown in Figure 4.7(f). Here, the distributivity rule is applied on the topmost nodes. In Figure 4.7(g), the two nodes labeled '4' are almost identical; they only differ in $\perp$ and $\top$. In this scenario, $\perp$ and $\top$ are interchangeable, and this allows us to replace $\perp$ and $\top$ with two signals of opposite polarities. Figure 4.7(h) shows the resulting network, where $\perp$ is replaced by the input x_3, and $\top$ by the input $\bar{x}_3$. Further, the replacement rule is applied on node labeled '1'. The highlighted branch (being x here) is substituted with a new sub-graph w, resulting in Figure 4.7(i). The replacement rule can also be applied in a similar way on the left branch of node labeled '1'; the resulting graph is Figure 4.7(j). The replacement rule allows us to apply the distributivity rule on the graph (highlighted in gray). The new graph is shown in Figure 4.7(k); the node M_5 has been changed into its optimum expression from Figure 4.6(e). The swapping rule is next applied on the graph shown in Figure 4.7(k). Since $(v_1 \oplus v_2)(w_1 \oplus w_2) = 0$,



(a) Not optimized majority-9: 24 nodes

(b) Manually optimized majority-9: 15 nodes

(c) Optimized majority-9: 12 nodes

Figure 4.9 – Decomposing majority-9 into majority-3. (a) is the original graph mapped form the BDD. Results in (b) is obtained by manually optimized (a), while (c) is obtained starting from (a) and using the rewriting rules in an automatic implementation © 2018 IEEE [177]

then $\langle x\langle y v_1 w_1\rangle\langle y v_2 w_2\rangle\rangle = \langle x\langle y v_2 w_1\rangle\langle y v_1 w_2\rangle\rangle$ and branches w_1 and w_2 can be exchanged between them, resulting in Figure 4.7(l). Distributivity and relevance are applied (highlighted in Figure 4.7(l)) to obtain the network of Figure 4.7(m). This is the final network, which has 7 nodes and which corresponds to the optimum solution. As in the previous case, the optimality of the result has been demonstrated using SAT-based exact synthesis and by showing that a solution with 6 gates is unfeasible. The final expression is

$$\langle x_2\langle x_5\langle x_1x_3x_4\rangle\langle\bar{x}_5x_6x_7\rangle\rangle\langle x_4\langle x_5x_6x_7\rangle\langle x_1x_3\bar{x}_4\rangle\rangle\rangle \tag{4.12}$$

Note that an alternative expression without complemented edges is

$$\langle x_2\langle x_5\langle x_1x_3x_4\rangle\langle x_6x_7\langle x_1x_3x_4\rangle\rangle\rangle\langle x_4\langle x_5x_6x_7\rangle\langle x_1x_3\langle x_5x_6x_7\rangle\rangle\rangle\rangle \tag{4.13}$$

This expression can be obtained by applying the relevance rule on the highlighted nodes in Figure 4.8(a). The optimum result without complemented edges is shown in Figure 4.8(b). Further, it is worth noting that to find the same solution, SAT-based exact synthesis (see, e.g., [162]) takes around 0.5 seconds; here, the entire derivation is provided.

We have applied the same rules for the decomposition of the majority-9 expression $\langle x_1x_2x_3x_4x_5x_6x_7x_8x_9\rangle$, using an approach similar to the one employed for majority-5 and majority-7, and we have demonstrated the realization of majority-9 using 15 nodes. The initial (not optimized) majority graph is the one shown in Figure 4.9(a). As of today, $M(9)$ is still unknown. We applied similar reduction and rewriting techniques as in the case of majority-7 and were able to reduce the number of majority-3 operations to 15 (Figure 4.9(b)). At the time of this work, this was the smallest realization of majority-9 in terms of majority-3. At the time of this writing, we report also a majority-9 realization using 12 nodes (shown in Figure 4.9(c)) [6]. This has been obtained using algebraic rewriting (replacement rule) on the majority-9 obtained from the BDD in an automatic way. As shown in Figure 4.9(c), this graph is not leafy and contains complemented edges, but it is highly symmetric. This demonstrates that our BDD method results into an advantageous starting point for finding optimum or close-to-optimum implementations. State-of-the-art exact synthesis is still not able to find an optimum representation for majority-9. The best-known lower bound is 10; this was derived by showing, using exact synthesis, that no majority graph can be found with 9 nodes or less.

Upper bounds

In this section, we compare upper bounds of the proposed method with the state-of-the-art. Further, we show that our method leads to a tighter upper bound for majority-9.

Our proposed synthesis method from a BDD suggests an upper bound $u_{\mathrm{B}}(n)$ for the majority-n function.

Theorem 4.5 *The majority-n function $\langle x_1 \dots x_n \rangle$ can be realized using a majority graph with at most $u_{\mathrm{B}}(n) \le \left(\lceil \frac{n}{2} \rceil\right)^2 - 1$ majority operations.*

Proof 4.5 *Let $n = 2k+1$, i.e., $k = \lfloor \frac{n}{2} \rfloor$. The BDD to represent $\langle x_1 \dots x_n \rangle$ has a diamond shape with 1 node at the first level, 2 nodes at the second level, until $k+1$ nodes at level $k+1$. Then, the number of nodes per level decreases: it has k nodes on level $k+2$, $k-1$ nodes on level $k+3$, until 1 node on the last level. Further, this node in the last level will be a leaf in the majority graph (thus the -1). In summary, this leads to:*

$$u_{\mathrm{B}} \le \sum_{i=1}^{k+1} i + \sum_{i=1}^{k} i - 1.$$

From this, we can derive

$$\begin{aligned} \sum_{i=1}^{k+1} i + \sum_{i=1}^{k} i - 1 &= \frac{(k+1)(2k+2)}{2} - 1 = (k+1)^2 - 1 \\ &= \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)^2 - 1 = \left(\left\lceil \frac{n}{2} \right\rceil\right)^2 - 1. \end{aligned}$$

■

Theorem 4.5 yields a quadratic upper bound, but as discussed in the introduction, it is possible to do much better. A quasi-linear construction follows from sorter networks [96]. We simply sort the n bits and pick the one that ends up in the middle position. Sorter networks consist of comparators, which are functions that map a pair of numbers $x, y \mapsto \min(x, y), \max(x, y)$. For Boolean numbers we have $\min(x, y) = x \wedge y$ and $\max(x, y) = x \vee y$, i.e., each comparator in a sorter network can be composed with 2 majority-3 operations. Let $S(n)$ be the optimum number of comparators in a sorter network that sorts n elements. Then an upper bound on

Table 4.1 – Upper bounds on the number of majority-3 operations to realize majority-n © 2018 IEEE [177]

n	3	5	7	9	11	13	15	17
Optimum ($M(n)$)	1	4	7					
Optimized BDDs	1	4	7	**12**				
BDDs	3	8	15	24	35	48	63	80
Sorter networks	6	18	32	50	70	90	112	142
Median selection*	18	30	42	53	65	77	89	101

* These numbers are based on the number of comparators in the construction of [70], but do not take other operations into account.

Table 4.1 shows the upper bounds on majority functions with various inputs n. The optimum $M(n)$ is known for n up to 7. Even though the BDD method leads to a quadratic upper bound, for small n (up to 17) the upper bound is better than state-of-the-art. The BDD method – followed by optimization – leads to the best-known majority-9 with 12 gates.

the number of majority-3 operations is $u_S(n) \leq 2S(n)$. From [61], it is known that $S(10) = 29$. Using, e.g., the systematic construction from Batcher [25], one can derive upper bounds for larger n.

Dor and Zwick showed that less than $2.942n$ comparisons are necessary to select the median value from a set of n numbers, without the need to sort them [70]. Applying it directly, it would lead to an upper bound of $5.884n$ majority-3 operations to decompose majority-n. However, this number needs to be treated more carefully, since their analysis is based on the *comparison model* in which *only* the number of comparators are counted and all other operations are considered free.

Both the construction on sorter networks and median selection have asymptotically better upper bounds compared to the quadradic bound from the BDD construction. However, when actually calculating the numbers for small instances with $n \leq 17$, the BDD approach yields the smallest values (see Table 4.1, which also shows the known optimum results up to $n = 7$ that were confirmed using exhaustive enumeration [162]). The approach is therefore a good starting point for finding compact majority-n realizations for small n. The results obtained using BDD + optimization rules are listed in Table 4.1 as **Optimized BDD**. Our method is able to obtain (i) the optimum known results for majority-5 and -7, and (ii) the best-known result for majority-9. One may be able to derive a general derivation procedure to obtain optimum or close to optimum majority-n realizations for $n \geq 9$.

4.4 The Complexity of Self-Dual Monotone 7-input Functions

In the previous section, we described a mapping method for monotone Boolean functions, based on the transformation of BDDs into majority graphs. We have used the proposed method to map the majority-n functions into MIGs with majority-3 operations, obtaining optimum solutions in which each node is connected to at least one PI (*leafy* solution). In other words, we demonstrated that for the majority-n functions up to 7 inputs, the *complexity* is invariant when considering majority or leafy majority graphs. Further, it has been proved that all majority-n functions can recursively be constructed from self-dual monotone functions.

In this section, thus, we study the class of functions that includes self-dual monotone functions, in particular, focusing on their implementation as multi-level logic networks that only use (i) majority and (ii) majority-leafy operators. We use enumeration-based and SAT-based exact synthesis methods to find (i) the minimum number of operators in the shortest formula of a Boolean function, and (ii) the minimum length of its Boolean chain. First, we introduce the background and the notation useful to understand the rest of the chapter, this includes: majority Boolean chains and their complexity and P equivalence classes; then, we present the proposed algorithms and the results.

4.4.1 Preliminaries

In this section, we focus on self-dual monotone functions and in particular on their complexity [188]. In order to define the complexity of a logic Boolean function, we introduce the concept of majority Boolean chain.[1] The complexity of a Boolean function depends on the way used to represent the function over a given set of primitives. Given a Boolean function $f(x_1,\ldots,x_n)$ of n input variables, a *majority Boolean chain* [97] is a way of representing functions defined as a sequence $(x_{n+1},...,x_{n+r})$, with the property that each step i in the chain combines 3 previous steps or inputs using a 3-input majority operator, such that for $n+1 \le i \le n+r$:

$$\begin{aligned} x_i &= \langle x_{1(i)} x_{2(i)} x_{3(i)} \rangle \\ x_{1(i)} &< x_{2(i)} < x_{3(i)} < x_i \end{aligned} \tag{4.14}$$

A *leafy* majority Boolean chain is a majority Boolean chain for which $x_{1(i)} \le n$ for all steps i. In other words, each step in such Boolean chain is constrained to have at least one previous step which is an input variable. The inherent complexity of a Boolean function is studied here according to two different measures, being (i) the *combinational complexity*, and (ii) the *length*. The combinational complexity of a Boolean function f, denoted as $C(f)$, is defined [97] as the minimum length r of the majority or leafy majority Boolean chain such that $x_{n+r} = f(x_1,\ldots,x_n)$. Note that the definition of Boolean chain allows for multiple fanouts: multiple distinct steps in the chain may refer to the same input or step x_i. On the other hand, the length $L(f)$ is defined as the number of 3-input majority operators (leafy or not leafy) in the shortest formula for f. It can be easily verified that $L(f) = C(f)$ for $n \le 3$, and that $L(f) \ge C(f)$ [97].

Generally, the study of the complexity of Boolean functions considers the problem of

---

[1] In [97] Knuth called them *median* Boolean chains.

finding some upper bounds [78, 88] or lower bounds [139] over a set of primitives. In our case, we are instead interested in finding exact numbers both for the length and the combinational complexity over majority operators. Similar problems have already found application in the logic synthesis and optimization field [81, 161, 162], as, for instance, logic rewriting algorithms optimize logic networks by replacing small subnetworks with optimized Boolean chains [81, 161]. In [97], the complexity for all 4- and 5-input Boolean functions in terms of 2-input Boolean operators have been studied, while 3-input Boolean operators have been used in [83]. Having exact numbers for the combinational complexity of some small functions can help to find tighter upper bounds for larger functions by using arguments from Boolean decomposition.

A Boolean function $f(x_1, x_2, \ldots, x_n)$ is *monotone* if and only if $f(x) \le f(y)$ whenever $x \subseteq y$. This means that for the bitstrings $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_m$, it follows $x_i \le y_i$ for all i. A monotone Boolean function can be expressed using only AND ($\wedge$) and OR ($\vee$) operators, without using complementation [97]. Being $\langle xyz \rangle$ the majority-of-three-input (majority-3) operator and considering that $\langle x0y \rangle = x \wedge y$ and $\langle x1y \rangle = x \vee y$, it follows that any monotone Boolean functions can be written using only majority-3 operators and constants, without using inverters.

A Boolean function $f(x_1, x_2, \ldots, x_n)$ is *self-dual* if it satisfies

$$\bar{f}(x_1, x_2, \ldots, x_n) = f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$$

where the $\bar{}$ is used to represent the complementation. In other words, it states that negations can be freely propagated from the inputs to the output. A monotone Boolean function expressed over $\wedge$ and $\vee$ operators is self-dual if the symbols $\wedge$ and $\vee$ can be interchanged without affecting the value of the function. The majority-3 is an example of self-dual function.

Another useful definition is the one of *normal* Boolean function, also called *0-preserving* function [140]. A Boolean function $f(x_1, x_2, \ldots, x_n)$ is normal if

$$f(0, 0, \ldots, 0) = 0$$

More generally, a Boolean chain is normal if and only if all its operators are normal (see also Chapter 3).

The majority operator is monotone, self-dual and normal. When considering functions over 7 inputs, there are 1,422,564 self-dual monotone functions. A majority chain always computes a monotone and self-dual function. Also, for each monotone and self-dual function, there exists a majority Boolean chain that computes it.

Input Permutation Equivalence

Two Boolean functions are *P-equivalent* if they are equivalent up to permutation of their inputs. As an example, functions $f = a \wedge \bar{b}$ and $g = \bar{a} \wedge b$ are equal if we swap the input a with input b and thus are said to be P-equivalent. P-equivalence is used to group functions into P-equivalent classes which consist of all functions equivalent up to permutation of the inputs. A class for a function f is denoted here as $[f]$. When two function f and g belong to the same class, i.e., $g \in [f]$, they are P-equivalent. Each class can be represented using the *canonical representative* $\hat{f}$ of the class, which is the function $f \in [f]$ that has the truth table corresponding to the smallest integer value. More details about efficient exact and heuristic algorithms for P classification can be found in [97].

In this section, we are interested in studying the complexity of Boolean functions. A key property is that all P-equivalent functions, i.e., functions which are in the same class, have the same combinational complexity $C(f)$ and the same length $L(f)$, which means that when $g \in [f]$, $C(f) = C(g)$ and $L(f) = L(g)$. All self-dual monotone 7-input Boolean functions can be grouped into 716 classes according to the permutation of their inputs. The key idea is that, thanks to P-equivalence, we can study the complexity of self-dual monotone 7-input functions by looking at $C(f)$ and $L(f)$ for only 716 functions, instead of for all 1,422,564 functions. This is preferable, since the number of P classes is significantly smaller than the number of functions. Thanks to this property, P-equivalence and its generalization to NPN [79] are largely used in logic synthesis, for instance in logic rewriting and exact synthesis [81, 121].

4.4.2 Length and Combinational Complexity

In this section, we describe novel algorithms to enumerate and classify self-dual monotone 7-input functions with respect to their length and their combinational complexity over majority operators. First, we illustrate the implementation of an algorithm to classify functions according to their $L(f)$. The same algorithm allowed us to obtain the truth table for all the 716 self-dual monotone 7-input functions. Then, we propose a SAT-based exact synthesis method to classify the obtained 7-input functions according to their combinational complexity. Indeed, the combinational complexity of Boolean functions can be extracted directly from an optimum size Boolean chain obtained with SAT-based exact synthesis, as it corresponds to the number of steps in the optimum solution. Both the majority and the leafy majority Boolean chains are considered.

Length L(f): Algorithm L

This section describes an exact algorithm to evaluate the length of self-dual monotone functions in terms of majority operators. We start by describing the algorithm for majority operators, which is inspired by the one in [97]. In particular, it is a 3-input majority-based version of "Algorithm L" presented in Section 7.1.2 of [97]. Finally, we address the changes necessary to

```
   Input: Truth tables of input variables $x_k$
   Output: $function\_to\_length$
 1 $count \leftarrow 1{,}422{,}564$ ;
 2 $current\_length = 0$ ;
 3 foreach $variable \in x_k$ do
 4     $function\_to\_length(variable) \leftarrow current\_length$ ;
 5 end
 6 while $count > 0$ do
 7     $current\_length = current\_length + 1$ ;
 8     $j, k = 0$ $l = current\_length - 1$ while $l > 0$ do
 9         foreach combination of $g, h, i \in function\_to\_length$ with length $j, k, l$ respectively do
10             $f \leftarrow \langle ghi \rangle$ ;
11             if $f \not\in function\_to\_length$ then
12                 $function\_to\_length(f) \leftarrow current\_length$ ;
13                 $count \leftarrow count - 1$ ;
14                 if $count = 0$ then
15                     return $function\_to\_length$;
16                 end
17             end
18         end
19         if $j + k = current\_length - 1$ then
20             $j = j + 1$ ;
21             $k = j$ ;
22         end
23         else
24             $k = k + 1$ ;
25         end
26         $l = current\_length - 1 - j - k$;
27     end
28 end
29 return $function\_to\_length$ ;
```

Algorithm 4.1: Algorithm L to compute $L(f)$

make the algorithm work for the leafy majority case.

The idea is to compute the length of all 1,422,564 functions by enumerating all functions with length $0, 1, 2, \ldots r$. Each function f with $L(f) = r$ is built as $\langle ghi \rangle$, where g, h, and i are three functions already enumerated and whose sum of lengths is equal to $r - 1$. This is practically obtained by enumerating and storing all self-dual monotone functions from previous lengths. As we are using only majority operators, each function built using previously obtained self-dual monotone functions is self-dual and monotone and can be added to the list itself, if not already there. The algorithm is also called [97] *Find normal length* as it works only on normal Boolean chains.

The pseudocode is depicted in Algorithm 4.1. The input is a vector containing the truth tables for the 7-input variables. The first part of the procedure initializes the count to the total number of functions (line 1 in Algorithm 4.1) and the length for the input variables to 0 (lines [3–5] in Algorithm 4.1). The table $function\_to\_length$ maps each function, represented as truth table, to its length. The algorithm's outer loop takes into account the total number of functions, and it ends once the counter hits 0. The inner while loop considers different values for j, k, and l, which are the lengths of functions g, h, and i, respectively. Function f is computed as the majority of functions g, h, and i, using all combinations over the three functions whose lengths sum is equal to $current\_length - 1$ (lines [21–27] of Algorithm 4.1). The length of f is equal to $current\_length$, further, as all functions from previous steps are self-dual and monotone, it follows also function f is self-dual and monotone. Thus, if the function is not already present in the $function\_to\_length$ map, it is saved in the map together with its length. The algorithm ends when all functions have been computed, and it returns all functions and their corresponding lengths.

In practice, few changes to Algorithm 4.1 allowed us to save not only all 1,422,564 self-dual monotone 7-input functions, but also the 716 representatives of the P-classes. The for loop in line 11 of Algorithm 4.1 consists practically of 3 loops over g, h, and i. The algorithm consists thus of 5 nested loops, whose complexity grows with $current\_length$. Experimentally, to save runtime, we used the map $function\_to\_length$ to search for the existence of function f (line 13 of Algorithm 4.1), while the loops over g, h, and i have been performed using vectors of truth tables. Further runtime has been saved by using the commutativity property of majority, thus by disregarding all combinations of g, h, and i already considered in different orders.

Algorithm 4.1 works over majority operators. We also designed a second algorithm to work on leafy majority formulas. To constraint Algorithm 4.1 to work with leafy operators, we constrained variable j to be always equal to 0, thus to consider only input variables. It means function g loops over all input variables x_k, while h and i can consider functions with larger lengths. We do not report the leafy-algorithm here as, apart from fixing $j = 0$, it remains the same as Algorithm 4.1.

Combinational Complexity C(f): Exact Synthesis

In this section, we present a SAT-based exact method to evaluate the combinational complexity of self-dual monotone 7-input functions in terms of majority operators. The combinational complexity is invariant under input permutation, thus we apply the SAT-based exact method only to the 716 P-classes, whose number is significantly smaller than the total number of functions. The truth tables for the 716 functions have been obtained by using Algorithm L in 4.1. As in the previous section, we first present the general method that works over majority Boolean chains, we then describe the differences to the implementation in order to consider leafy majority Boolean chains.

The implemented exact synthesis algorithm starts by trying to find a Boolean chain for function f using $r = 0$. If a solution exists with r steps, the algorithm returns a majority Boolean chain that implements function f, otherwise it searches a solution with larger size $(r + 1)$. The algorithm increases the number of steps until a solution is found.

```
synthesize_maj($f, r, S$)
$S \leftarrow$ Restart SATSolver ;
$CreateVariables(S, f, r)$ ;
$AddMainClause(S, f, r)$ ;
$AddFanInClauses(S, r)$ ;
$AddOtherClauses(S, f, r)$ ;
if $Solve(S)$ then
    return Majority Boolean chain ;
else
    return synthesize_maj($f, r+1, S$);
```

Algorithm 4.2: SAT-based exact method to compute $C(f)$

This idea is described in the recursive procedure depicted in Algorithm 4.2, which is applied to each function f separately. The inputs of the algorithm are (i) the function specification f (for instance, given as truth table), (ii) the number of steps r, and (iii) the SAT solver S. For the majority Boolean chain, we used the same encoding presented in [162, 180] and used in Chapter 3. This is an extension that works over 3-input Boolean operators of the encoding first proposed by Knuth [98] for 2-input Boolean operators. In our case, the operations of each step are limited to the majority operator, without allowing inversions. The clauses are the same as discussed in [98]. The main clause is the one which encodes the truth table of the circuit, while the fanin clause assures that each step has exactly 3 distinct inputs. $AddOtherClauses$ consists of both necessary and additional clauses proposed in [163], which can be added to reduce the solving time of the SAT solver. More details about both clauses formalization and additional clauses can be found in [82, 98, 163]. Algorithm 4.2 is first applied to each function using $r = 0$, r is then increased until a solution is found. It is worth noting that this method allows to not only count the number of functions for each combinational complexity, but also to obtain their implementation in terms of majority operators.

In order to constrain Algorithm 4.2 to work only with leafy majority Boolean chains, we changed the $AddFanInClauses$ in order to constrain the first input of each step to be one of the input variables. The fanin of each step i is encoded in the *select variable* s_{ijkl}, which is true if steps x_j, x_k and x_l are the children of step x_i. In the leafy case, the fanin clause is changed to ensure $x_j \leq n$. The rest of the algorithm remains the same as Algorithm 4.2.

4.4.3 Experimental Results

In this section, we describe the experimental results both for the length and the combinational complexity of self-dual monotone 7-input functions. First, we present our results for the majority case. Then, we give a comparison between majority and leafy majority Boolean chains.

We have implemented the proposed algorithms using the open source EPFL Logic Synthe-

Table 4.2 – $L(f)$ and $C(f)$ for all 716 self-dual monotone 7-input classes

	Majority Computation					Leafy Majority Computation			
$L(f)$	Classes	Functions	$C(f)$	Classes	$L_{\text{leafy}}(f)$	Classes	Functions	$C_{\text{leafy}}(f)$	Classes
0	1	7	0	1	0	1	7	0	1
1	1	35	1	1	1	1	35	1	1
2	2	350	2	2	2	2	350	2	2
3	8	3745	3	9	3	8	3745	3	9
4	38	35203	4	48	4	35	33628	4	45
5	139	270830	5	201	5	123	233660	5	191
6	313	699377	**6**	**354**	6	272	600887	6	347
7	176	367542	**7**	**98**	7	210	449673	7	114
8	34	43135	8	2	8	50	84519	8	6
9	3	2310	9	0	9	12	14770	9	0
10	0	0	10	0	10	1	1260	10	0
11	1	30	11	0	11	0	0	11	0
12	0	0	12	0	12	1	30	12	0

Table 4.2 shows the length $L(f)$ and combinational complexity $C(f)$ for self-dual monotone 7-input functions, comparing the majority operators with the leafy majority operators. In the leafy case, $L(f)$ is increased from 11 to 12 for one class of function; while the maximum $C(f)$ in unaffected.

sis Libraries [165]. Algorithm L has been implemented using the truth table library *kitty*.[2] The SAT-based exact method has been implemented using the exact logic synthesis library *percy*.[3] We have used the "maj_encoder" and the `Glucose` SAT solver [20, 21]. All the experiments have been carried out on Intel Xeon E5-2680 CPU with 2.5 GHz and with 256 GB of main memory.

Regarding the majority Boolean chains, the results of classification have already been presented in [97]. Our results have been obtained using Algorithm L (Algorithm 4.1) and are shown in the first part of Table 4.2. The first three columns of Table 4.2 show both the number of classes and the number of functions for each length. The largest length is equal to 11 (in agreement with the results from [97]). Consider as an example function $f = \langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 \rangle$, which is the majority-of-seven-inputs (majority-7). Its $L(f)$ has been demonstrated both here and in [97] equal to 8, given by:

$$\langle x_1 \langle x_2 \langle x_3 x_4 x_5 \rangle \langle x_3 x_6 x_7 \rangle \rangle \langle x_4 \langle x_2 x_6 x_7 \rangle \langle x_3 x_5 \langle x_5 x_6 x_7 \rangle \rangle \rangle \rangle \tag{4.15}$$

While for the length we have implemented the same algorithm presented in [97] to obtain

---

[2] Available at: *https://github.com/msoeken/kitty*

[3] Available at: *https://github.com/whaaswijk/percy*

our results, for the combinational complexity we have used an alternative approach, i.e., SAT-based exact synthesis. The results are obtained applying Algorithm 4.2 on all 716 functions, with a total runtime of 4038 seconds and 2997 seconds for the majority and leafy majority Boolean chains, respectively. The runtime for the method presented in [97] on majority Boolean chains is 6894 seconds. Note also that while Knuth's algorithm only counts the number of functions for each class, in our case extra memory and runtime are necessary in order to also get the majority networks implementations.[4] Our results for the combinational complexity are shown in columns 4 and 5 of Table 4.2. As we used a SAT-based exact synthesis method, we have computed the combinational complexity only for the 716 classes. The maximum combinational complexity is 8 (in agreement with the results in Table 4.3). The shortest chain for function $f = \langle x_1x_2x_3x_4x_5x_6x_7\rangle$ needs 7 steps and it is given by:

$$\begin{gathered}x_8 = \langle x_1x_3x_4\rangle,\ \ x_9 = \langle x_6x_7x_8\rangle,\ \ x_{10} = \langle x_5x_8x_9\rangle\\ x_{11} = \langle x_5x_6x_7\rangle,\ \ x_{12} = \langle x_1x_3x_{11}\rangle,\ \ x_{13} = \langle x_4x_{11}x_{12}\rangle\\ x_{14} = \langle x_2x_{10}x_{13}\rangle\end{gathered} \tag{4.16}$$

This last result matches the one demonstrated both in [97] and in Section 4.3.

The second half of Table 4.2 shows the leafy majority results. The worst $L(f)$ is increased to 12, while the largest combinational complexity is still 8. As a general trend, functions need more steps when the majority operators are constrained to have at least one PI. For example, note that the number of chains with 8 steps is increased from 2 to 6. By comparing the majority results with the leafy ones, we can conclude that:

1) The class with the largest length is the same both in majority and leafy majority computation, and it is the function with truth table (in hexadecimal form):

`fefefeaafeccf080fef0cc80aa808080`

Its combinational complexity in both majority and leafy is equal to 8 steps, obtained using the Boolean chain given by:

$$\begin{gathered}x_8 = \langle x_5x_6x_7\rangle,\ \ x_9 = \langle x_2x_3x_5\rangle,\ \ x_{10} = \langle x_1x_8x_9\rangle\\ x_{11} = \langle x_3x_7x_{10}\rangle,\ \ x_{12} = \langle x_2x_6x_{10}\rangle,\ \ x_{13} = \langle x_4x_{11}x_{12}\rangle\\ x_{14} = \langle x_1x_4x_{13}\rangle,\ \ x_{15} = \langle x_5x_{13}x_{14}\rangle\end{gathered} \tag{4.17}$$

2) Regarding the length, 98 functions out of 716 have different length when comparing majority with leafy majority formulas. The maximum difference in length is equal to 3. This is true for functions:

---

[4] Available at: *https://github.com/eletesta/7input_classification*

`feeaeaeaeeaaaaa0faaaaa88a8a8a880`
`feeeeeeaeeeaecc0fcc8a888a8888880`
`feeaece0faeaa8a0faeaa8a0f8c8a880`

3) For the combinational complexity, 40 functions out of 716 have a different combinational complexity when comparing majority with leafy majority Boolean chains. In this case, the difference is equal to 1 for all the functions.

4) Up to 3 steps, the results are the same both for majority and leafy majority case. This was expected, if we consider that we are not allowing steps in the Boolean chains to have two equal inputs.

5) For the majority-7, the length in the majority case is equal to 8 and its complexity is 7. The combinational complexity in the leafy case is unchanged (w.r.t. the majority case) and equal to 7, as it has been demonstrated in Section 4.3 and seen in (4.16). Further, we prove that also its length is unchanged when considering leafy majority operators.

To further stress the difference between majority and leafy majority Boolean chains, consider function `feeaeaeaeeaaaaa0faaaaa88a8a8a880` as an example. When considering its length, the difference between majority and leafy majority is equal to 3. i.e., $L(f) = 4$ and $L_{\text{leafy}}(f) = 7$. The smallest majority Boolean chain needs 4 steps:

$$\begin{aligned} x_8 &= \langle x_1 x_3 x_6 \rangle, \quad x_9 = \langle x_1 x_4 x_5 \rangle, \\ x_{10} &= \langle x_1 x_2 x_7 \rangle, \quad x_{11} = \langle x_8 x_9 x_{10} \rangle \end{aligned} \tag{4.18}$$

In the leafy case, the same Boolean chain cannot be used, as the last step x_{11} is the majority of three majority operators in the previous steps. The function needs 5 steps in the leafy majority Boolean chain, given by:

$$\begin{gathered} x_8 = \langle x_1 x_3 x_6 \rangle, \quad x_9 = \langle x_1 x_2 x_7 \rangle, \\ x_{10} = \langle x_1 x_8 x_9 \rangle, \quad x_{11} = \langle x_5 x_8 x_9 \rangle, \quad x_{12} = \langle x_4 x_{10} x_{11} \rangle \end{gathered} \tag{4.19}$$

As a last result, it is worth mentioning that our SAT-based exact synthesis method was able to demonstrate better combinational complexity for one of the 716 classes with respect to the original results in [97]. The discrepancy was due to a bug in the original version of the algorithm in [97], which has been found and solved as a result of this work. The original results obtained in [97] are listed in Table 4.3. In Table 4.3, the number of functions with complexity 6 and 7 respectively are not in agreement with our results in Table 4.2 (see highlighted numbers). In particular, for one function, the SAT-based synthesis method was generating a smaller combinational complexity. The discrepancy has been discussed with the author, and corrected in the most recent version of [97].

Table 4.3 – $L(f)$ and $C(f)$ for all self-dual monotone 7-input functions taken from Section 7.1.2 in [97]

$L(f)$	Classes	Functions	$C(f)$	Classes	Functions
0	1	7	0	1	7
1	1	35	1	1	35
2	2	350	2	2	350
3	8	3745	3	9	3885
4	38	35203	4	48	42483
5	139	270830	5	201	406945
6	313	699377	**6**	**353**	798686
7	176	367542	**7**	**99**	169891
8	34	43135	8	2	282
9	3	2310	9	0	0
10	0	0	10	0	0
11	1	30	11	0	0

Table 4.3 shows the length $L(f)$ and combinational complexity $C(f)$ for self-dual monotone 7-input functions over majority operators. The discrepancy in this table as compared to our results in Table 4.2 is due to a bug in the original version of [97].

Table 4.4 – $C(f)$ for 715 self-dual monotone 7-input functions over majority Boolean chains with inverters

$C(f)$	Classes
0	1
1	1
2	2
3	9
4	48
5	**210**
6	**374**
7	**70**
8	**0**

Table 4.4 shows the combinational complexity $C(f)$ for self-dual monotone 7-input functions over majority operators with inverters. The table is not complete (one function is still running), but preliminary results show that inverters have a positive effect in decreasing the complexity of such functions.

Concerning future developments of this section results, a key role is played by the use of inverters. It is already known from complexity theory that the inverters could help to reduce asymptotic bounds of size and depth of logic networks for monotone functions [89]. Future work may include studying the effect of inversions both on the length and the combinational complexity of self-dual monotone 7-input functions. Also for this case, leafy majority Boolean chains could be considered. Towards this direction, we show in Table 4.4 some preliminary results for the combinational complexity of the 716 self-dual monotone Boolean functions when using majority Boolean chains and inverters. The results are obtained using a SAT-based exact synthesis method for MIGs [14], as presented in [162]. This method is similar to the one presented in Algorithm 4.2. At the time of writing the thesis, 715 functions out of 716 were synthesized by our exact synthesis method. [5] Even if not complete, Table 4.4 shows that the combinational complexity of 39 functions is decreased thanks to the use of inverters. For instance, consider function `feeeeee8faa8aaa0faaaeaa0e8888880`. The combinational complexity over majority Boolean chain (with no inversion) is equal to 7, given by the Boolean chain:

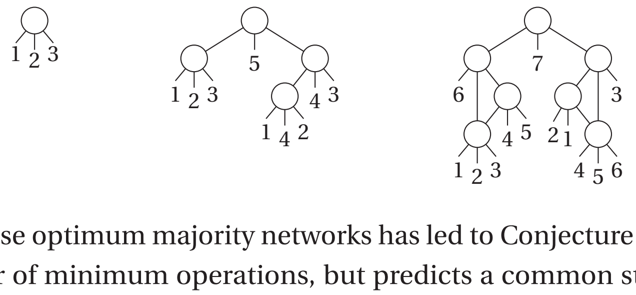
$$\begin{aligned} x_8 =& \langle x_1 x_3 x_4 \rangle, \;\; x_9 = \langle x_2 x_6 x_8 \rangle, \\ x_{10} =& \langle x_7 x_8 x_9 \rangle, \;\; x_{11} = \langle x_3 x_4 x_{10} \rangle, \\ x_{12} = \langle x_5 x_6 x_{11} \rangle, \;\; & x_{13} = \langle x_2 x_5 x_{12} \rangle, \;\; x_{14} = \langle x_1 x_{10} x_{13} \rangle \end{aligned} \tag{4.20}$$

The same function can be synthesized using only 6 steps, if we allow inverters:

$$\begin{aligned} & x_8 = \langle x_1 \bar{x}_5 x_7 \rangle, \;\; x_9 = \langle x_1 x_5 \bar{x}_7 \rangle, \;\; x_{10} = \langle x_3 x_4 x_8 \rangle, \\ & x_{11} = \langle x_2 x_6 x_9 \rangle \;\; x_{12} = \langle x_1 x_5 \bar{x}_9 \rangle, \;\; x_{13} = \langle x_{10} x_{11} x_{12} \rangle \end{aligned} \tag{4.21}$$

4.5 Pairs of Majority-Decomposing Functions

In this section, we present further theoretical results on top of those presented in Section 4.3. Ultimately, we are again driven by the question of how many majority-3 operations are sufficient to realize the majority-n function. Recall that we address the minimum number of majority-of-three to build a majority-n as $M(n)$. Optimum majority networks obtained with the BDD method for $n = 3$, 5, and 7 are demonstrated in Section 4.3 and reported here:



(4.22)

The structure of these optimum majority networks has led to Conjecture 4.1, which does not predict the number of minimum operations, but predicts a common structure. We report again the conjecture here to help the reading of the rest of the chapter.

---

[5] Unfortunately, after 10 months of computation, one function is still running. It is not surprising that the "missing" function is the one with largest lenght and combinational complexity.

Conjecture 4.1 *There always exists an optimum network in which (i) each node is connected to at least one PI and (ii) the root node is the only node that is connected to* x_n.

For example, variables x_3, x_5, and x_7 only appear at the root nodes of the optimum majority networks for $n = 3,5$, and 7 in (4.22). In order to derive further knowledge from the optimum majority networks for $n = 3,5$, and 7, and motivated by Conjecture 4.1, in this section, we investigate decompositions of the majority-n function $\langle x_1 \dots x_n \rangle$ into the majority-3 expression $\langle x_n f_1 f_2 \rangle$, such that f_1 and f_2 are Boolean functions over $n-1$ variables that do not depend on x_n.

The main result of this section is the following:

Theorem 4.6 *For* $k \geq 1$, *let* $n = 2k+1$, *and* f_1 *and* f_2 *two* $(n-1)$*-variable Boolean functions. Then*

$$\langle x_1 \dots x_n \rangle = \langle x_n f_1 f_2 \rangle,$$

if, and only if

(a) $f_1(x_1,\dots,x_{2k}) = f_2(x_1,\dots,x_{2k}) = 1$, *if* $x_1 + \cdots + x_{2k} > k$,

(b) $f_1(x_1,\dots,x_{2k}) \oplus f_2(x_1,\dots,x_{2k}) = 1$, *if* $x_1 + \cdots + x_{2k} = k$, *and*

(c) $f_1(x_1,\dots,x_{2k}) = f_2(x_1,\dots,x_{2k}) = 0$, *if* $x_1 + \cdots + x_{2k} < k$.

In other words, if the number of ones in the input pattern is less than k, then both functions must evaluate to 0, and if the number of ones is larger than k, then both functions must evaluate to 1. Only in the case where the number of ones equals k, one has the freedom to select the output of one function to be 1, if the other function outputs 0.

Moreover, a second conjecture states:

Conjecture 4.2 $M(n) = \frac{3(n-3)}{2} + 1$, *for odd* $n \geq 3$.

Note that $M(n)$ is the minimum number of majority-of-three to build a majority-n. Consequently, $M(9) = 10$. However, neither a witness nor a proof excluding the existence of a network with 10 majority operations has been found. We made use of our findings in an exhaustive search algorithm and were able to find experimentally that Conjecture 4.2 and Conjecture 4.1 cannot *both* be true. More precisely, there cannot be a majority network for majority-9 with 10 majority operations (as predicted by Conjecture 4.2) adhering to a structure as described by Conjecture 4.1.

The remainder of the section is organized as follows. First, we give the proof for Theorem 4.6. Then, after introducing threshold functions in Section 4.5.2, we review two results from the literature as special cases of Theorem 4.6, together with a new decomposition, which is also a special case of Theorem 4.6 and can be used as an explanation for the optimum majority network for $n = 7$.

4.5.1 Proof of the Main Theorem

In this section, we give the proof for Theorem 4.6.

Proof of Theorem 4.6 *The theorem is proved by case distinction on* x_n. *If* $x_n = 0$, *then the result of the majority-n must be true only if more than k of the arguments* $x_1, \dots, x_{2k}$ *are true. Case (a) yields* $\langle 011 \rangle = 1$; *case (b) yields* $\langle 001 \rangle = \langle 010 \rangle = 0$; *case (c) yields* $\langle 000 \rangle = 0$.

If $x_n = 1$, *then the result of the majority-n must be true only if at least k of the arguments are true. Case (a) yields* $\langle 111 \rangle = 1$; *case (b) yields* $\langle 101 \rangle = \langle 110 \rangle = 1$; *case (c) yields* $\langle 100 \rangle = 0$. ■

4.5.2 Majority-n Decompositions

In this section, we describe two majority-n decompositions from state-of-the-art and introduce a novel type of decomposition called “Parity-splitting Decomposition”. The described decompositions are special cases of our main result from Theorem 4.6. In order to understand the following discussion, we first illustrate some important symmetric Boolean functions called *threshold functions*, which are a generalization of majority functions. Let

$$S_{>k}(x_1, \dots, x_n) = [x_1 + \cdots + x_n > k] \tag{4.23}$$

be the function that is true, if *more than* k of the input arguments are true. Also, for $k > 0$ let

$$S_{=k}(x_1, \dots, x_n) = S_{>k-1}(x_1, \dots, x_n) \wedge \overline{S_{>k}(x_1, \dots, x_n)} \tag{4.24}$$

be the function that is true, if *exactly* k of the input arguments are true.

Let $n = 2k + 1$ for some integer $k \geq 1$. Then the majority-n function can also be written as

$$\langle x_1 \dots x_n \rangle = S_{>k}(x_1, \dots, x_n). \tag{4.25}$$

Co-factor Decomposition

The first decomposition was discovered by Akers in the early 1960s [9]. It is a simple decomposition in which f_1 and f_2 are the positive and negative co-factor of the majority-n function, respectively. One obtains the positive or negative co-factor of a function f with respect to a variable x_i, by setting x_i to 1 or 0, respectively:

$$\langle x_1 \dots x_n \rangle = \langle x_n \langle x_1 \dots x_{n-1} 1 \rangle \langle x_1 \dots x_{n-1} 0 \rangle \rangle \tag{4.26}$$

Theorem 4.7 *For* $k \geq 1$ *and* $n = 2k+1$, *the functions* $f_1^k = \langle x_1 \dots x_{n-1} 0\rangle$ *and* $f_2^k = \langle x_1 \dots x_{n-1} 1\rangle$ *are a pair of majority-decomposing functions.*

Proof 4.7 *It follows easily from noting that* $f_1^k = S_{>k}(x_1, \dots, x_{2k})$ *and* $f_2^k = S_{\geq k}(x_1, \dots, x_{2k})$. ■

Example 4.5 *We use the co-factor decomposition to derive an expression for majority*-3. *In this case, we get* $n = 3$, $f_1^1 = \langle x_1 x_2 0\rangle = x_1 \wedge x_2$, *and* $f_2^1 = \langle x_1 x_2 1\rangle = x_1 \vee x_2$. *Hence, the decomposition leads to the expression* $\langle x_3 (x_1 \wedge x_2)(x_1 \vee x_2)\rangle$ *with 3 majority*-3 *operations to express a single majority*-3 *operation.* ■

Majority-reducing Decomposition

In this section, we review a decomposition from Amarel, Cooke, and Winder [12] that sets $f_1^k = \langle x_1 \dots x_{2k-1}\rangle$. In other words, the majority-n function is decomposed in terms of the smaller majority-$(n-2)$ function.

Theorem 4.8 (See [12] for the proof) *For* $k \geq 1$ *and* $n = 2k+1$, *the functions* $f_1^k = \langle x_1 \dots x_{2k-1}\rangle$ *and* $f_2^k = S_{>k}(x_1, \dots, x_{2k-1}) \vee x_{2k} S_{>k-2}(x_1, \dots, x_{2k-1})$ *are a pair of majority-decomposing functions.*

Example 4.6 *We use Theorem 4.8 to find a decomposition for majority*-3. *Then,* $f_1^1 = S_{>0}(x_1) = x_1$ *and* $f_2^1 = S_{>1}(x_1) \vee x_2 S_{>-1}(x_1) = x_2$. *Hence* $\langle x_1 x_2 x_3\rangle = \langle x_3 x_1 x_2\rangle$. ■

The optimum network for majority-5 can be directly derived from this decomposition as illustrated by the following example.

Example 4.7 *For* $k = 2$, *we have* $f_1^2 = \langle x_1 x_2 x_3\rangle$, *corresponding to the subnetwork on the left-hand side in* (4.22), *and*

$$f_2^2 = S_{>2}(x_1, x_2, x_3) \vee x_4 S_{>0}(x_1, x_2, x_3) = x_1 x_2 x_3 \vee x_4 (x_1 \vee x_2 \vee x_3).$$

One can readily verify that $f_2^2 = \langle x_1 x_4 \langle x_2 x_3 x_4\rangle\rangle$, *which corresponds to the subnetwork on the right-hand side in the optimum network for majority*-5. ■

It is worth noting that in [12], the authors also show how to describe f_2^k in terms of k majority-3 operations and $k+1$ majority-5 operations.

Parity-splitting Decomposition

In this section, we introduce a new decomposition, which we will use to explain the optimum majority network for majority-7. Let $n = 2k+1$, as in the previous sections. The decomposition, by construction, is not applicable to all odd n, but only when k is odd, e.g., $n = 3$, $n = 7$, $n = 11$, and so on. We first define a function

$$g_k(x_1, \dots, x_k, x_{k+1}, \dots, x_{2k}) = \\ S_{>k}(x_1, \dots, x_{2k}) \oplus S_{=k}(x_1, \dots, x_{2k})(x_1 \oplus \dots \oplus x_k), \quad (4.27)$$

which is true, if (i) either more than k arguments are true, or if (ii) exactly k arguments are true while an odd number of these k arguments must be from the first arguments $x_1, \dots, x_k$. We can use this function to describe a pair of majority-decomposing functions.

Theorem 4.9 *Let $k \geq 1$, and k be odd. Then*
$f_1^k = g_k(x_1, \dots, x_k, x_{k+1}, \dots, x_{2k})$
and
$f_2^k = g_k(x_{k+1}, \dots, x_{2k}, x_1, \dots, x_k)$
are a pair of majority-decomposing functions.

Proof 4.9 *It is easy to see that case (a) and (c) of Theorem 4.6 are true from the definition of g_k.*

In the case of (b), the functions simplify to $f_1^k = x_1 \oplus \dots \oplus x_k$ and $f_2^k = x_{k+1} \oplus \dots \oplus x_{2k}$. Since k is odd, we have $f_1^k \oplus f_2^k = x_1 \oplus \dots \oplus x_{2k} = 1$. Note that in the case k even, the $x_1 \oplus \dots \oplus x_{2k}$ would be equal to 0, and the theorem would not be valid. ■

Example 4.8 *Let $k = 3$, i.e., $n = 7$. Then one can verify that*

$$f_1^3 = S_{>3}(x_1, x_2, x_3, x_4, x_5, x_6) \oplus (x_1 \oplus x_2 \oplus x_3) S_{=3}(x_1, x_2, x_3, x_4, x_5, x_6)$$

can be expressed as

$$\langle x_3 \langle x_4 x_5 x_6 \rangle \langle x_1 x_2 \langle x_4 x_5 x_6 \rangle \rangle \rangle,$$

which corresponds to the subnetwork on the right-hand side in the optimum network for majority-7 in (4.22). Similarly, f_2^3 corresponds to the subnetwork on the left-hand side, as it is obtained by simply swapping x_1, x_2, x_3 with x_4, x_5, x_6. In fact, it is quite surprising that in the optimum network for majority-7, there is no sharing between the networks for f_1^3 and f_2^3, although their expressions are very similar. ■

Also the optimum network for $k = 1$, i.e., majority-3, can be derived from the parity-splitting decomposition.

Example 4.9 *Let $k = 1$. Then we have*

$$f_1^1 = S_{>1}(x_1, x_2) \oplus x_1 S_{=1}(x_1, x_2) = x_1 x_2 \oplus x_1 (x_1 \oplus x_2) = x_1.$$

Analogously, we find $f_2^1 = x_2$. ■

4.5.3 Application to Find Optimum Majority Networks

Having found that the reviewed and proposed decompositions can explain the optimum results for $n \leq 7$, we investigate whether they help to find optimum networks for larger n.

Theorem 4.6 describes a large set of pairs of majority-decomposing functions. Case (a) and (c) fix the output for f_1^k and f_2^k for all input patterns with less or more than k ones. But for the $\binom{2k}{k} = \frac{(2k)!}{k!^2}$ input patterns that have exactly k ones, one can decide whether to assign f_1^k to 1 or 0. This leads to $2^{\frac{(2k)!}{k!^2}}$ different pairs of decomposing functions. Concretely, these are $4, 64, 2^{20}, 2^{70}$, and 2^{252} for $k = 1, 2, 3, 4$, and 5.

We first show that Conjecture 4.2 and Conjecture 4.1 cannot both hold for $n = 9$, i.e., $k = 4$. In other words, there exists no majority network for majority-9 with 10 gates in which each gate points to a PI and only the top-most gate points to x_9. Instead of finding a majority network for majority-9, we tried to find a majority network for a pair of functions (f_1^4, f_2^4) with 8 inputs and 9 gates. We leave f_1^4 and f_2^4 unspecified, but only constrain them to adhere to the conditions from Theorem 4.6. This allows to explore the full space of all $2^{\frac{8!}{4!^2}} = 2^{70}$ possible decompositions. We expressed this problem using a SAT solver similar to the encoding proposed in [99, 162]. On a MacBook computer using a 2.7 GHz Intel Core i5 processor with 8 GB memory, we are able to show that the problem is unsatisfiable within about 5 minutes. Since no network with 9 gates exists to compute any pair (f_1^4, f_2^4), there cannot be a majority network to compute majority-9 with 10 gates that follows the structure described in Conjecture 4.1. However, there still may exist a majority network for majority-9 with 10 gates, but if so, it cannot have a decomposition structure similar to those found for $n = 3, 5$ and 7. Or, the optimum network requires more than 10 gates but can still have a structure as described by Conjecture 4.1.

Based on Conjecture 4.2, majority-11 might be realizable using 13 operations. Inspired by the structure for majority-7 in (4.22), we tried to realize $f_2^5 = S_{>5}(x_1, \dots, x_{10}) \oplus (x_1 \oplus \cdots \oplus x_5) S_{=5}(x_1, \dots, x_{10})$ from the parity-splitting decomposition using 6 gates. We can show with exhaustive search that this is not possible. This implies that the structure for majority-7 in (4.22) cannot trivially be extended for majority-11, with two disjoint subnetworks for f_1^5 and f_2^5. However, this result does not imply that there is no 13-operation majority network for majority-11 that used the parity-splitting decomposition, since there may be shared nodes for f_1^5 and f_2^5.

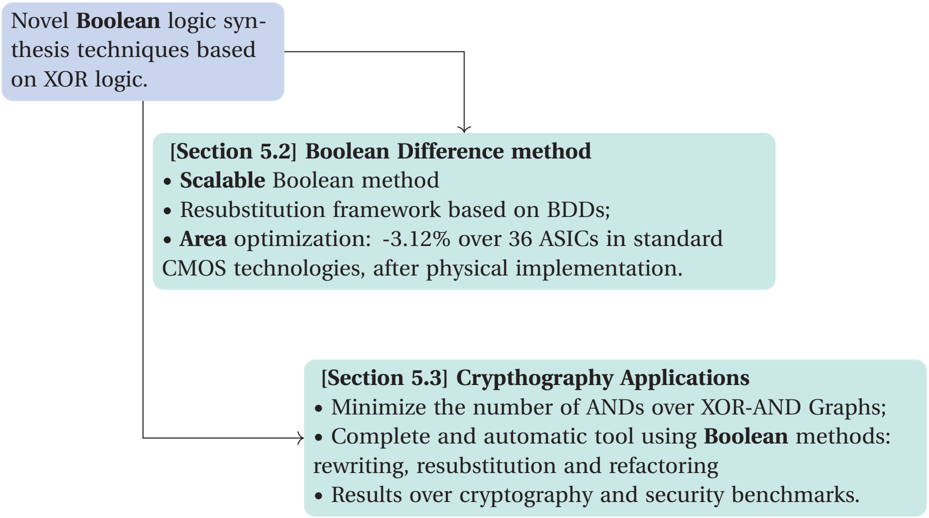
4.6 Summary

In this chapter, we illustrated many and diverse theoretical results on majority logic. We addressed two main topics, being (i) answering the question "how best can the n-argument majority function be realized with a network of 3-input majority gates?"; and (ii) the complexity of self-dual monotone functions, and its dependancy on inversions and leafy constraint. Regarding the first topic, we described a mapping method for monotone Boolean functions, based on the transformation of BDDs into majority graphs. We used the proposed method to map the majority-n function into majority graphs. Due to the more compact initial representation, the approach is favorable as compared to methods based on median selection and sorter networks, for small n. In particular, we were able to derive the optimum MIG for majority-5 and majority-7 functions, and the best-known results for majority-9. For their optimization, we introduced two useful new identities for majority-based function optimization. Moreover, we presented a novel decomposition, called "parity-splitting decomposition". This novel decomposition contributed in giving more insights on the optimum results of large n. We also studied the complexity of self-dual monotone 7-input functions in terms of 3-input majority operators. Finding minimum chains is not only of interest from a theoretical point of view, but it also has practical application. For example, they can be used in logic optimization and technology mapping. Our method uses both state-of-the-art algorithms and exact synthesis, based on P classification, to compute the complexity of Boolean functions according to their (i) length, and (ii) combinational complexity. We demonstrated how inverters can positively contribute in reducing the complexity for this class of functions, while constraints on the inputs (i.e., leafy majority nodes) of each node can instead have a negative effect. As a final result, for the majority function of 7 inputs, nor the inverters neither the leafy constraint change the complexity of the function, when realized using 3-input majority operators.

5 XOR-based Logic Synthesis

Chapter 3 was dedicated to majority logic synthesis mainly addressing emerging technologies, while Chapter 4 considered theoretical results of majority-n Boolean functions. This chapter focuses instead on the study of XOR-based logic synthesis, i.e., techniques that use the XOR operator as a key part of the optimization. The chapter consists of two main sections, concentrating on different aspects and optimization goals. First, we present a novel Boolean resubstitution method based on the *Boolean difference*, for the area optimization of standard CMOS-based designs. Then, we describe a complete and automatic framework that works over *xor-and graph*s (XAGs), to minimize the number of AND gates for cryptography and security applications.

An overview of the chapter is presented hereafter and shown in Figure 5.1. First, the motivations are detailed in Section 5.1. Then, we present two XOR-based optimizations in Section 5.2 and Section 5.3, respectively. In particular, in Section 5.2, we illustrate a novel Boolean method addressing size optimization of *and-inverter graph*s (AIGs). The method is a resubstitution algorithm, based on computing the Boolean difference between two nodes of the network. Note that the Boolean difference is the XOR operation between two functions. Our method uses BDDs as underlying datastructure for optimization, and allows us to reduce the area of 36 industrial benchmarks of −3.12% on average, after place & route, at limited runtime increase. The section is largely based on the publications in [170, 171]. In Section 5.3, we present three different methods concentrating on the minimization of the number of AND gates in an XAG. The three methods are based on classical resubstitution, rewriting, and refactoring techniques of standard logic synthesis. The presented tool is fully automatic, and achieves remarkable results over both EPFL benchmarks and two sets of cryptography benchmarks. This last section is based on the publications presented in [175, 178]. Finally, the chapter is concluded in Section 5.4 with a summary.



Novel **Boolean** logic synthesis techniques based on XOR logic.

[Section 5.2] Boolean Difference method
- **Scalable** Boolean method
- Resubstitution framework based on BDDs;
- **Area** optimization: -3.12% over 36 ASICs in standard CMOS technologies, after physical implementation.

[Section 5.3] Crypthography Applications
- Minimize the number of ANDs over XOR-AND Graphs;
- Complete and automatic tool using **Boolean** methods: rewriting, resubstitution and refactoring
- Results over cryptography and security benchmarks.

Figure 5.1 – Chapter organization

5.1 Motivation

This section discusses the motivations of the chapter. Compared to the previous chapter, which concentrates on theoretical results, this chapter shows a more pragmatic approach to logic synthesis, and thus considers practical aspects to make algorithms and methods successful in logic syntehsis complex tools. This chapter focuses entirely on XOR-based logic synthesis methods, and experimental results address both classical CMOS-based design, and cryptography and security benchmarks.

The benefit of the XOR operator in representing and optimizing logic has already been demonstrated in [58, 81, 85]. For instance, the work in [81] proved how majority graphs enriched with the XOR operator (*xor-majority graph*s (XMGs)) are more compact representations and result in more efficient exact synthesis-based rewriting than MIGs; while the work in [85] has presented the benefit of the XOR operator in AIG-based rewriting. The renewed interest in Boolean methods in logic synthesis is due to the continuous push to improve *quality of results* (QoR) faced by the *electronic design automation* (EDA) community. In particular, and in light of modern computing capabilities, Boolean synthesis methods have experienced revived attention in the last few years [19, 171]. Boolean methods use *don't cares* as degree of freedom for optimization and rely on complete functional properties of Boolean functions. Compared to algebraic methods, which treat a Boolean function as a polynomial, Boolean methods achieve better QoR but imply higher computational cost and consequently have been used cautiously in EDA flows [38, 69]. For instance, the recent work in [19] showed improvements to Boolean resynthesis, enabling some high-quality Boolean methods to be runtime affordable. This motivates our research to revisit high-quality and high-computational-complexity optimization methods. In Section 5.2, we present a novel Boolean resubstitution method (based on the Boolean difference), specifically designed to be scalable (runtime-effcient) and to unveil further optimization opportunities in modern synthesis flows. The XOR operator is the key in identifying the *difference* between two functions, which is defined as the ⊕ of two Boolean functions.

Our research in XOR-based Boolean methods has resulted in a novel interest for such techniques, and, in the second half of the chapter, we specifically extend XOR-based logic synthesis to consider an alternative optimization objective for cryptography and security applications. Recently, the works in [33, 60, 149, 175] have started this new domain of application for logic synthesis. In this scenario, logic synthesis makes use of XAGs [175] as data structure for optimization, because they efficiently abstract cryptography circuits over the basis {AND, XOR, NOT} [33]. Further, logic synthesis focuses on the minimization of the number of AND gates as its main target metric for optimization. The minimization of the number of AND gates for cryptography is fundamental for two main reasons. First, the number of AND gates correlates to the degree of vulnerability of a circuit [183]. The minimum number of AND gates sufficient to implement a Boolean function as an XAG is called *multiplicative complexity of the function* [183], while the *multiplicative complexity of the logic network* is defined as the actual number of AND gates used in the network representation of the function [80, 175]. The multiplicative complexity of a function directly correlates to the resistance of the function against algebraic attacks [64], while the multiplicative complexity of a logic network implementing that function only provides an upper bound. Consequently, minimizing the multiplicative complexity of a network is important to assess the real multiplicative complexity of the function, and therefore its vulnerability. Second, the number of AND gates plays an important role in high-level cryptography protocols such as *zero-knowledge protocols*, *fully homomorphic encryption* (FHE), and secure *multi-party computation* (MPC) [10, 52, 149]. For example, the size of the signature in post-quantum zero-knowledge signatures based on “MPC-in-the-head” [77] depends on the multiplicative complexity in the underlying block cipher [52]. Moreover, the number of computations in MPC protocols based on Yao’s garbled circuits [167] with the free XOR technique [100] is proportional to the number of AND gates. Regarding FHE, XOR gates are considered cheaper and less noisy compared to AND gates. To further motivate our work, it is worth mentioning that in techniques to protect against side-channel attacks, the cost of general-purpose protections grows with the number of AND gates [64]. Moreover, in a different domain, the work in [113] has recently demonstrated the positive effect of the minimization of AND gates on the number of qubits and expensive quantum operations (T gates) in fault-tolerant quantum circuits.

While it is clear that the multiplicative complexity has a key role in cryptography and that logic synthesis can have a strong impact in its optimization, so far, there are no fully automatic logic synthesis tools able to address the optimization of the number of AND gates in a network as their main goal for optimization. State-of-the-art tools [40, 190] automatically address size optimization, without precisely minimizing the number of ANDs, and methods from the cryptography community rely heavily on manual decomposition and optimization strategies [33]. Thus, in Section 5.3, we propose a fully *automatic* logic synthesis toolbox for cryptography applications. The proposed tool presents a *complete* synthesis flow that interchanges various logic synthesis techniques able to find different optimization opportunities on the same network.

5.2 Boolean Difference Method

This section presents a novel Boolean resubstitution framework based on the Boolean difference computation and implementation, aiming at optimizing the size of circuits. We focus on techniques to make our Boolean resubstitution methods efficient and applicable to large functions. We thus discuss novel Boolean filtering and partitioning techniques to break the logic into smaller subnetworks. In the remainder of the section, we first present the theory on the Boolean difference; then, we describe the algorithm implementation and the complete flow. Finally, we present the experimental results over both academic and industrial benchmarks.

5.2.1 Theory

The *Boolean difference* [38] of a Boolean function $f(x_1, x_2, \ldots, x_n)$ w.r.t. an input variable x_i is defined as:

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i} \tag{5.1}$$

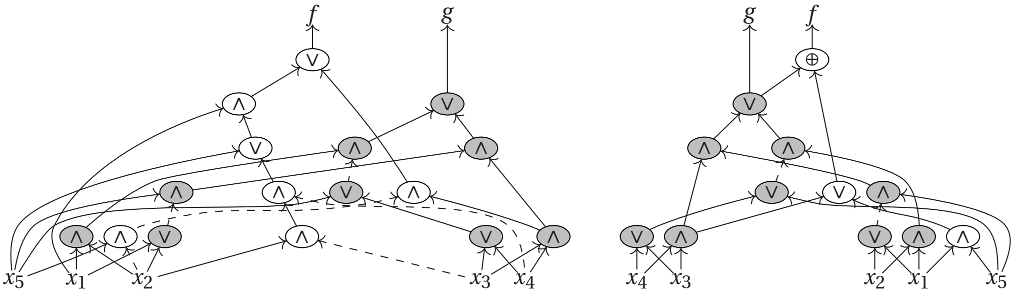
where f_{x_i} and $f_{\bar{x}_i}$ are the two cofactors and $\oplus$ is the XOR operator. It states whether function f is sensitive to any change in input x_i. In a similar way, the Boolean *difference* of two Boolean functions $f(x_1, x_2, \ldots, x_n)$ and $g(x_1, x_2, \ldots, x_n)$ is defined as:

$$\frac{\partial f}{\partial g} = f \oplus g, \tag{5.2}$$

It indicates whether the two functions are functionally equivalent (i.e., the Boolean difference value w.r.t. inputs assignments is 0) or not (i.e., they have a Boolean difference equal to 1). Note that the Boolean difference is a function, and it takes value 0 and 1 with respect to some input assignments.

According to Boolean difference, each function f can be written as $f = \frac{\partial f}{\partial g} \oplus g$. While this approach reminds general XOR bi-decomposition algorithms [125, 168], in our case, the main advantage of the resubstitution method comes from the synthesis of the Boolean difference $\frac{\partial f}{\partial g}$. Since g is a node already in the logic network, the cost of implementing f is given entirely by the Boolean difference $\frac{\partial f}{\partial g}$. A small Boolean difference implementation could thus result in size reduction for the logic network.

Example 5.1 *Consider as an example the logic network for function f and g in Figure 5.2(a). Each node in Figure 5.2(a) is a 2-input gate, and dashed edges represent inverters. Function f is rewritten as $\frac{\partial f}{\partial g} \oplus g$ in Figure 5.2(b). The function g is the one highlighted in gray in both Figure 5.2(a) and (b). The small size of the Boolean difference network results in a decreased total number of nodes (from 16 to 12).* ■



(a) Logic network for functions f and g (in gray)

(b) Function f rewritten as $f = \frac{\partial f}{\partial g} \oplus g$

Figure 5.2 – Boolean difference example: (a) is the original AIG of f and g, while (b) represents the network optimized using the Boolean difference – thus introducing a XOR node. © 2019 IEEE [171]

We refer to function f and g as *candidates* for Boolean difference, and to the inputs variables $x_1, x_2, \ldots, x_n$ as their support. We use f and g both for the corresponding nodes in the logic network and for the function they represent.

In the following, we first present the selection of the candidates and their support, and we discuss algorithms aiming at computing the Boolean difference between two nodes. Then, the resubstitution flow based on the Boolean difference is presented. Both the use of BDDs and partition methods are discussed.

5.2.2 Boolean Difference Optimization

Identifying Viable Candidates

To ensure the scalability of this Boolean method, we evaluate and apply the Boolean difference locally on limited size *circuit partitions*. The partitions are created by collecting all the nodes in topological order and by sorting them according to the similarity of their structural support. Each partition respects some predefined characteristic, e.g., maximum number of *primary input*s (PIs), maximum number of internal nodes V, maximum number of levels, etc. In our implementation, we give priority to the limit on the maximum number of levels, as they correlate with the complexity of the reasoning engine selected for the Boolean difference computation. Nevertheless, we also ensure partitions to have limited size and limited number of PIs. Experimentally, we found promising bounds on the number of levels ranging from 5 to 30, resulting in partitions with controlled maximum size of 1000 nodes.

In order to find good candidates f and g, all pairs of nodes inside each partition are considered. The supports for the computation are the PIs of the partition itself. As this requires the evaluation of V pairs of nodes for each node, in the worst case, the time complexity of the resubstitution framework is quadratic w.r.t. the partition size V. Experimentally, to reduce the time complexity, we fixed the maximum number m of pairs to be tried. Structural filtering can

```
Input: Two nodes $f$ and $g$, $xor\_cost$, $all\_bdds$
Output: A new node $boolean\_diff$ equal to $\frac{\partial f}{\partial g} \oplus g$
$boolean\_diff \leftarrow 0$;
$bddf \leftarrow all\_bdds(f)$;
$bddg \leftarrow all\_bdds(g)$;
$bdd\_diff \leftarrow bddf \oplus bddg$;
if $bdd\_diff$ already exists in $all\_bdds()$ then
    return corresponding node;
end
if (size($bdd\_diff$) $>$ $threshold$) then
    return $null$;
end
$saving \leftarrow$ mffc($f$) $+$ $nodes\_sharing$;
if (size($bdd\_diff$) $+$ $xor\_cost > saving$) then
    return $null$;
end
$bdiff\_node \leftarrow$ bdd_to_node($bdd\_diff$);
$boolean\_diff \leftarrow bdiff\_node \oplus g$;
return $boolean\_diff$;
```

Algorithm 5.1: Boolean difference computation and implementation using BDDs

also accelerate the computation. For example, the algorithm does not consider nodes with less than one element in their shared support, and it also neglects cases where f is completely included in g, or partially included up to a certain threshold. Functional filtering similar to the one in [19] also helps speeding up the computation. After all structural and functional filtering, we can apply the method to EPFL *i2c* and *cavlc* benchmarks with a runtime of 2.3 and 1.2 seconds, respectively.

Computing and Implementing The Difference

BDDs are the selected data structure to compute and implement the Boolean difference. BDDs result into fast evaluation of the Boolean difference (as XOR of the two BDDs) and can be easily “strashed” [1] into their corresponding AIG. The pseudocode is depicted in Algorithm 5.1.

Recall that f and g are two nodes belonging to the same partition. The BDDs for all nodes in the partition are precomputed and stored in the hashtable all_bdds. The algorithm computes the BDD of the Boolean difference as XOR of the two BDDs. Thanks to the limited size of the partition, BDDs allow fast Boolean difference computation. If the BDD of the difference already exists in the hashtable, the corresponding node is returned. In our implementation, we did not perform any BDD variables ordering, as we are dealing with small BDDs. This saves runtime, but it requires a higher amount of memory to be used by the BDD package. The memory usage plays a critical role. For instance, for the EPFL *cavlc* benchmark, the algorithm does not converge in a reasonable amount of time unless the memory used for the BDD of the difference is freed at each iteration. In this last case, the algorithm was applied on the whole

---

[1] strashing refers to structural hashing, defined in [119]

network, which has 10 inputs and more than 600 nodes. To further prevent memory issues, we set a maximum memory limit for the employed BDD package. The BDD computation is bailed out if the maximum memory limit is hit. This case results into a BDD of size 0 for the given node, which will be disregarded in the next steps of the algorithm.

Afterwards, structural filtering is applied on the BDD. In case the BDD does not meet the size requirements, Algorithm 5.1 returns $null$, which means the current pair of nodes can be skipped. First, we limit the size of the BDD (lines 8–10 in Algorithm 5.1) to consequently limit the size of the difference network once its BDD is merged into the AIG. This usually ensures a limited size implementation for the Boolean difference, but it may overlook some optimization opportunities. Empirically, we found 10 to be a suitable tradeoff to have good QoR and feasible runtime. The second filter skips pairs of nodes that could result in a larger network implementation. Experimentally, we skip nodes whose $saving$ is smaller than the empirical threshold set by the BDD size and the xor_cost. The saving resulting from the Boolean difference is the sum of the size of the MFFC of f and the total sharing of nodes between the Boolean difference implementation and the existing network. The size of the BDD sets a lower bound on the number of AIG nodes to implement the Boolean difference. The xor_cost is the number of AIG nodes needed to implement the functionality of a two-input XOR. According to the specific technology involved, the XOR node has a different area ratio as compared to AND/OR nodes, so the xor_cost can have a different value.

The algorithm concludes with the implementation of the Boolean difference node (lines 15 in Algorithm 5.1) as an AIG, obtained using strashing on the corresponding BDD. Optimization algorithms from the state-of-the-art are applied on the AIG to guarantee an optimized implementation.

Global Resubstitution Flow

We integrate the candidates selection and the Boolean difference computation into a resubstitution framework. Algorithm 5.2 depicts the pseudocode. The flow applies the resubstitution framework to each partition N of the entire network. The partitions can be chosen to be distinct or overlapping to cover more optimization opportunities. The algorithm precomputes and stores all BDDs in the hashtable, and considers all nodes in topological order. Trivial pairs of nodes are skipped according to criteria discussed above. Thanks to the use of BDDs, information needed for functional filtering of pairs are immediately available. Algorithm 5.1 is used to achieve the new implementation of f using the Boolean difference. Algorithm 5.2 accepts a new implementation of f only if (i) it leads to size minimization, or (ii) it does not increase the number of nodes. This second case could reshape the network, open new optimization opportunities and help escaping local minima.

```
Input: Network $N$, $xor\_cost$
Output: Optimized network
$lists \leftarrow$ topological_sorted_partitions($N$);
foreach $list$ in $lists$ do
    $all\_bdds \leftarrow$ BDDs for all nodes in $list$;
    for nodes $f$ in $list$ do
        for nodes $g$ in $list$ do
            if $f = g$ then
                continue;
            end
            if $f$ and $g$ are not good candidates then
                continue;
            end
            $diff \leftarrow$ Boolean_difference($f, g, xor\_cost, all\_bdds$);
            if size($diff$) <= size($f$) then
                Change $f$ with $diff$ in $N$;
            end
        end
    end
end
return $N$;
```

Algorithm 5.2: Resubstitution flow based on Boolean difference

5.2.3 Experimental Results

In this section, we evaluate the efficacy of the Boolean difference method for synthesis. First, we consider the EPFL logic synthesis competition[2]. In this scenario, we outperform previous EPFL best results coming from various research groups in industry and academia. Finally, we integrate our Boolean method in an industrial EDA flow for *application specific integrated circuits* (ASICs), and show sensible QoR gains post place & route.

Methodology

We implemented our scalable Boolean method as part of a commercial design automation solution. We target size reduction of logic networks, as, in the EDA flow, Boolean methods are frequently called during logic structuring, which mainly aims at reducing area. Nevertheless, we enforced a tight control on the number of levels and the number of nets during the synthesis flow for ASIC results, as this is known to correlate with delay and congestion later on in the flow.

We have integrated our Boolean difference method in an industrial logic synthesis tool, together with other novel methods we presented in [170, 171]. We created a global Boolean resynthesis flow which runs the following optimizations:

1. *Gradient-Based AIG Minimization*, which revisits classical AIG optimization by using a

---

[2] Available at: *https://github.com/lsils/benchmarks*

technique that adapts online to apply the most effective AIG transformations [171];

2. *Novel Resubstitution Methods*, which includes the novel resubstitution techniques from [19, 170];
3. *Resubstitution with Boolean Difference*, which consists of our novel resubstitution flow based on Boolean difference from Algorithm 5.2;
4. *Heterogeneous Elimination for Kernel Extraction* to enhance division and logic sharing to work on heterogeneous thresholds within the same network [171];
5. *SAT Resubstitution*, which includes the resubstitution techniques with Boolean filtering and windowing, implemented using SAT [170].

The optimization flow is run once. However, note that this Boolean resynthesis flow may produce better results when iterated multiple times, e.g., 2 to 5 times, depending on the specific runtime budget. Further, after each transformation, the logic network is translated into an AIG in order to have a consistent interface and costing between the various steps of the flow. More details on the implementation of the resynthesis flow can be found in our work on Boolean methods [170].

EPFL Benchmarks

In this section, we use our global synthesis flow to improve (decrease) the size of the EPFL benchmarks [15]. In particular, we challenge the area (i.e., number of LUTs) category within the EPFL competition,[3] which records the best synthesis results in term of size and number of levels obtained by mapping the optimized EPFL benchmarks into LUT-6.

In order to compare our results to the baseline, we apply our resynthesis flow followed by the ABC [40] command *"if -K 6 -a"* to map our AIGs into LUT-6. Since LUT-6 minimization does not follow strictly AIG minimization, we adapted our tool to work in general for the LUT-6 experiment in the following way: We inserted selective structural hashing of LUTs, over previous best results, with optimization and remapping on smaller partitions, in order to preserve some of the good LUT-6 structures.

Table 5.1 summarizes our results. At the time of evaluation of this work, we improved 12 (out of 20) of the previous best size (area) results of the EPFL benchmarks,[4] advancing the size results coming from [19], [108], and [110]. Our improvements range from a few LUTs to several (tens) for larger circuits. It is worth mentioning that the EPFL benchmarks have been optimized various times in the last 3 years by the most advanced techniques both from industry and academia, thus each improvement (even if relatively small), is highly significant.

---

[3] The best results for the EPFL benchmarks are available at: *https://github.com/lsils/benchmarks*

[4] We compare our results to commit 87cf8ec in *https://github.com/lsils/benchmarks*, reported here as **Baseline LUT-6 count** in Table 5.1

Table 5.1 – Best 6-LUT area results for the EPFL benchmarks © 2019 IEEE [171]

Benchmark	I/O	Baseline LUT-6 count	LUT-6 count	Level count
arbiter	256/129	403	**365**	117
div	128/128	3268	**3267**	1211
hypotenuse	256/128	40385	**40377**	4530
i2c	147/142	211	**207**	15
log2	32/32	6570	**6567**	119
max	512/130	523	**522**	189
mem_ctrl	1204/1231	2117	**2086**	23
mult	128/128	4923	**4920**	93
priority	128/8	106	**103**	26
sin	24/25	1228	**1227**	55
sqrt	128/64	3076	**3075**	1106
square	64/128	3244	**3242**	76

Table 5.1 shows the best-area results for the EPFL benchmarks, mapped into 6-LUTs. We report only the improved benchmarks. We optimized 12 of the previous best results in the EPFL logic synthesis competition.

Note also that some benchmarks (e.g., *max*) are mostly improved by our Boolean difference method, which is capable of resolving convergent logic not distinguished by other techniques. Furthermore, recently, new best-size results have been presented by Machado et al. in [109]. Even though additional size optimizations have been applied on top of our best results, we still hold the best area results for 5 out of the 12 results from Table 5.1, that is, no further optimizations were found for these 5 cases.

As already pointed out, a smaller AIG was not resulting in the best LUT-6 result for some of the benchmarks. Nevertheless, our global resynthesis flow allows us to obtain the smallest-known AIGs compared to the state-of-the-art,[5] which are reported in Table 5.2. For some benchmarks, this result is much smaller than the AIG size leading to the best LUT-6 results. As an example, we show 1.3× size reduction in the smallest known AIG for the EPFL *voter* benchmark.

ASIC Results

We tested a commercial EDA flow, empowered with our global resynthesis flow, on 36 state-of-the-art ASICs, coming from major electronics industries. Due to non-disclosure agreements, we cannot provide details on each ASIC benchmark. However, we present average results w.r.t. a baseline flow without our resynthesis methods. The post place & route results are summarized in Table 5.3. All benchmarks are verified with an industrial formal equivalence checking flow. In Table 5.3, the baseline is a complete industrial EDA flow from register transfer level to GDSII, without any of the mentioned techniques. The first experiment presents results when the gradient-based AIG minimization is included in the EDA flow. On average,

---

[5] The smallest known AIG from state-of-the-art has been computed by structural hashing the current best LUT-6 result and running *resyn2rs* from ABC [40] until no improvement is seen

Table 5.2 – Smallest AIG results for the EPFL benchmarks. Submitted to IEEE TCAD 2019

Benchmark	I/O	Best-known size AIG	Opt. size AIG
cavlc	10/11	584	483
div	128/128	22206	19250
hypotenuse	256/128	226220	209460
i2c	147/142	769	710
log2	32/32	33213	30522
mem_ctrl	1204/1231	7986	7644
mult	128/128	29885	25371
router	60/30	99	96
sin	24/25	6300	4987
sqrt	128/64	22569	19706
square	64/128	18187	17010
voter	1001/1	13272	9817

Table 5.2 shows the best-size AIGs for some of the EPFL benchmarks. For instance, it presents a 1.3× size reduction in the smallest known AIG for the EPFL *voter* benchmark.

Table 5.3 – Post Place&Route results on 36 industrial design for ASICs. Submitted to IEEE TCAD 2019

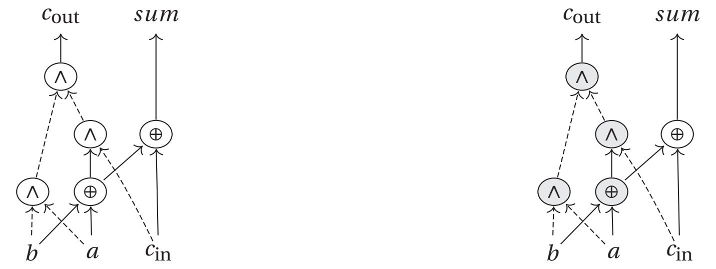
Flow	Comb. Area	N-C. Area	WNS	TNS	Comb. SW P.	Comb. Leak P.	# Cells	Runtime
Baseline	-	-	-	-	-	-	-	-
Ex1: optimized AIG [171]	-0.48%	+0.02%	+0.02%	+1.09%	-0.68%	-0.22%	-0.45%	+0.2%
Ex2: Ex1 + resub [170]	-1.31%	+0.04%	-0.65%	-0.24%	-1.29%	-0.45%	-0.95%	+0.5%
Ex3: Ex2 + Bdiff resub	-1.87%	+0.01%	+0.11%	+0.87%	-1.36%	-0.36%	-1.24%	+1.2%
Ex4: Ex3 + het. kernel [171]	-2.47%	-0.01%	-0.80%	-2.43%	-2.18%	-0.90%	-2.07%	+1.8%
Ex5: Ex4 + SAT resub [170]	**-3.12%**	**-0.02%**	**-1.34%**	**-0.82%**	**-2.49%**	**-0.98%**	**-2.25%**	+2.2%

“Comb. Area" and “N-C. Area” are the combinational and non-combintional area respectively. “WNS" is the worst negative slack, and “TNS" is the total negative slack; “Comb. SW P." is the dynamic power dissipation due to switching activity, while “Comb. Leak P. ” is the static power dissipation due to leakage. “ # Cells” is the total cell count in the design, and the “Runtime” is the total runtime post P&R. All results are averaged over 36 designs.

Table 5.3 shows average results of our flow over 36 industrial benchmarks for ASICs. We demonstrate an area reduction of -3.12%, and power of -2.49% after place and route, with limited runtime increase.

this technique decreases the combinational area, which is our target metric, of −0.48%, with negligible increase in runtime and WNS/TNS. The second experiment enriches experiment 1 with novel resubstitution methods from [19]. These techniques allow a further combinational area decrease, resulting in improvements also for WNS and TNS. Experiment 3 uses the resubstitution with Boolean difference presented in this thesis to further improve the combinational area, while experiment 4 enhances the previous steps by using heterogeneous elimination for kernel extraction. The last experiment presents results when the complete flow is applied, i.e., experiment 4 enriched with SAT resubstitution.

Our complete design flow embedding our new optimization, and highlighted in green, enables sensible combinational area and dynamic power (without considering the clock network) reductions, −3.12% and −2.49%, respectively, after physical implementation. On average, we also achieve WNS/TNS improvements, with a runtime increase of only +2.2%.



(a) XAG implementation of the full adder (b) One cut of the full adder highlighted in gray

Figure 5.3 – (a) XAG representation of the full adder. The signal c_{out} in the output carry. Dashed lines represent complemented edges. (b) highlights a subgraph described by the cut for the output c_{out} with leaves a, b and c_{in}. A cut is k-feasible (denoted here as k-cut), if it has at most k leaves (see Chapter 2)

5.3 Multiplicative Complexity and Cryptography Applications

In this section we present a XOR-based logic synthesis framework, aiming instead at the minimization of the number of AND gates in an XAG. As already pointed out, the minimization of the number of AND gates plays an important role for many and diverse cryptography and security applications. Our logic synthesis tool is complete and automatic, and it consists of (i) rewriting, (ii) resubstitution, and (iii) refactoring. The remainder of the section is organized as follows: We first discuss the preliminaries necessary to understand the rest of the chapter; then, we describe the complete flow and the three methods. We conclude with the implementation details and the experimental results over both EPFL benchmarks and benchmarks from the cryptography community.

5.3.1 Preliminaries

In this section, we review XAGs and the multiplicative complexity. We also discuss a Boolean functions classification method based on affine-equivalence of Boolean functions. This classification is used in the rewriting algorithm.

Xor-And Graphs and Multiplicative Complexity

In many cryptographic applications, Boolean functions are usually represented over the basis {AND, XOR, NOT} [36]. In analogy with the data structures usually involved in logic synthesis, e.g., AIGs [121], or *majority-inverter graph*s (MIGs) [14], in this thesis we represent logic networks from cryptographic applications in terms of XAG. We define an XAG as a logic network in which each gate corresponds to either an AND or an XOR operator. Both regular and complemented edges can be used to connect the gates, where a complemented edge indicates the inversion of the signal. Figure 5.3(a) shows an XAG representation of the full adder, which uses two XOR gates, denoted by $\oplus$, and three AND gates, denoted by $\land$. Inversions are represented as dashed lines. Previous works in logic synthesis have considered XOR-AND logic networks, called *xor-and-inverter graphs* (XAIGs) [85]. Even if our work and the one in [85] use the same data structure, XAIGs have been exploited to perform a different task. Indeed, the work in [85] focuses on LUT mapping, and considers XOR and AND gates to have the same cost.

As stated in the introduction, the multiplicative complexity *of a Boolean function* is defined as the minimum number of AND gates sufficient to implement it over the basis {AND, XOR, NOT} [33, 183]. More general, we also call the multiplicative complexity *of a logic network* the actual number of AND gates to implement the circuit [80]. For example, the full-adder in Figure 5.3 has a multiplicative complexity equal to 3.

Affine Functions Classification

This section reviews affine function classification, which is a strong Boolean function classification technique based on affine operations.

Definition 5.1 (Affine operations [73]) *The following set of five affine operations on a Boolean function can be used to partition all Boolean functions into equivalence classes [73].*

1. Swapping two variables. *From* $f(x_1,\dots,x_i,\dots,x_j,\dots,x_n)$, *one obtains* $g = f(x_1,\dots,x_j,\dots,x_i,\dots,x_n)$ *by swapping variables* x_i *and* x_j. *We denote this operation as* $f \xrightarrow{x_i \leftrightarrow x_j} g$.
2. Complementing a variable. *From* $f(x_1,\dots,x_i,\dots,x_n)$, *one obtains* $g = f(x_1,\dots,\bar{x}_i,\dots,x_n)$ *by complementing variable* x_i. *We denote this operation as* $f \xrightarrow{\bar{x}_i} g$.
3. Complementing the function. *One obtains* $g = \bar{f}$ *from* f *by complementing the whole function. We denote this operation as* $f \xrightarrow{\neg} g$.
4. Translational operation. *One obtains* $g = f(x_1,\dots,x_i \oplus x_j,\dots,x_n)$ *from* $f(x_1,\dots,x_i,\dots,x_n)$ *by replacing* x_i *with* $x_i \oplus x_j$. *We denote this operation as* $f \xrightarrow{x_i \oplus x_j} g$.
5. Disjoint translational operation. *One obtains* $g = x_i \oplus f$ *from* f *by XOR-ing it with input* x_i. *We denote this operation as* $f \xrightarrow{\oplus x_i} g$.

These operations partition all n-variable Boolean functions into equivalence classes by means of the following equivalence relation.

Definition 5.2 (Affine equivalence [104]) *We say that two* n*-variable Boolean functions* f *and* g *are* affine-equivalent, *if there exist operations* $o_1,\dots,o_k$ *from Definition 5.1 such that*

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} g.$$

One can readily verify that affine equivalence is an equivalence relation. In the remainder, we write $f \doteq g$*, if* f *is affine-equivalent to* g*. Further, we refer to the equivalence class of* f *as* $[f] = \{g \mid f \doteq g\}$.

We can define one element of $[f]$ to be the *representative function* of that class. In an abuse of notation, we use $[f]$ both as the set of all Boolean functions in the equivalence class, and also to denote the representative itself. Note that $f \doteq g$, if and only if $[f] = [g]$.

Example 5.2 *We can show that* $\langle x_1 x_2 x_3 \rangle \doteq x_1 \land x_2$*, where in this case* $x_1 \land x_2$ *is considered a 3-variable Boolean function in which* x_3 *is a don't care input.*

$$x_1 \land x_2 \xrightarrow{\bar{x}_2} x_1 \land \bar{x}_2 \xrightarrow{x_2 \oplus x_3} x_1 \land (\bar{x}_2 \oplus x_3) \xrightarrow{x_1 \oplus x_2}$$
$$(x_1 \oplus x_2) \land (\bar{x}_2 \oplus x_3) = x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 \xrightarrow{\oplus x_1}$$
$$x_1 \oplus x_1\bar{x}_2 \oplus x_1x_3 \oplus x_2x_3 = x_1x_2 \oplus x_1x_3 \oplus x_2x_3 = \langle x_1x_2x_3 \rangle$$

■

Using this equivalence relation the set of all n-variable Boolean functions for $n = 1, 2, 3, 4, 5, 6$ collapses into just $1, 2, 3, 8, 48, 150\,357$ equivalence classes [27, 183], respectively. An algorithm to compute the representative of each class and the set of operations $o_1 \dots o_k$ has been recently proposed in [114, 116].

5.3.2 General XOR-based Resynthesis Framework

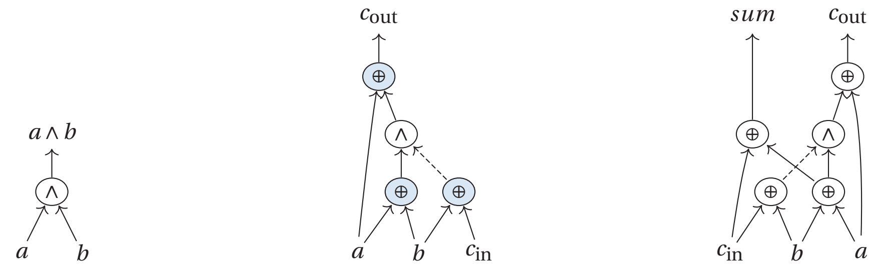
In this section, we present XOR-oriented *rewriting*, *resubstitution*, and *refactoring* as three new algorithms to create a logic synthesis flow for cryptography and security applications. The presented algorithms modify state-of-the-art logic synthesis optimization techniques by considering the minimization of the number of AND gates as their primary goal.

Rewriting

Rewriting is a technique largely used in logic synthesis, and allows to replace parts of a logic network with optimized (e.g., in the number of nodes or levels) subnetworks. These subnetworks can be precomputed, as done in [121, 175], or computed on-the-fly with *exact synthesis* as presented in [150]. This section presents first the general idea and illustrates the proposed rewriting method using an example, then it describes the optimization algorithm.

Our rewriting optimization method is based on two major considerations:

1. The multiplicative complexity of a function is invariant under affine operations. Each affine operation can be realized by (i) an XOR gate, (ii) an inversion or (iii) a permutation



(a) Circuit of the representative (b) Circuit of the representative and operations (c) Final full adder XAG

Figure 5.4 – Rewriting example: (a) is the circuit of the representative; (b) is the circuit of the representative plus the operations, and (c) is final circuit after rewriting

of two inputs. All of them do not affect the number of AND gates in an XAG. Thus, to find the multiplicative complexity of a Boolean function, it is enough to know the multiplicative complexity of the representative of its equivalence class. In other words, each function can be written as an XAG using the same number of AND gates of its representative.

2. As the number of affine classes is orders of magnitudes smaller than the number of functions, a minimum circuit implementation over {AND, XOR, NOT} is known [48, 183] for each representative up to 6-input functions. In this scenario, minimum means minimum in terms of AND gates. The minimum XAG implementation of each Boolean function (up to 6-input) can thus be obtained by the XAG of its representative. This is obtained by adding XOR gates, inverters, and input permutation in accordance with the operations from Definition 5.1. As stated above, this will not influence the number of AND gates.

These two considerations allow us to optimize the number of AND gates of a Boolean function by (i) using the minimum XAG of the representative and (ii) augmenting it by the gates required for each transformations. In the following, we use the full adder from Figure 5.3 as an example.

Example 5.3 *Consider the full adder in Figure 5.3(a), which has three AND gates. The objective is the minimization of the number of such gates. Let us focus on the* c_{out} *output, which has the subgraph highlighted in Figure 5.3(b). The subgraph implements the majority of three inputs* $\langle abc_{in} \rangle$ *which has truth table (in hexadecimal form) equal to* $0xe8$*. The representative of the class is the function* $0x88$*, which is the AND gate represented in Figure 5.4(a). As in Example 5.2,* $a \wedge b$ *is considered a 3-variable Boolean function in which* c_{in} *is a don't care input. This means that the full adder can be built using one AND gate together with some of the operations from Definition 5.1. The operations* $o_1 \dots o_k$ *to transform a majority gate into a AND gate are the ones from Example 5.2:* $\bar{b}, b \oplus c_{in}, a \oplus b, c_{out} \oplus a$*. These add three XOR gates to the circuit in*

```
Input: XAG of the cut $c$ of node $v$, DB_representative_to_xag
Output: Optimized XAG for cut $c$
$f \leftarrow$ Boolean function of $v$ with respect to the leaves ;
$representative \leftarrow$ representative of the equivalence class of $f$ ;
$operations \leftarrow$ operations to go from $f$ to $representative$ ;
if $representative \in DB\_representative\_to\_xag$ then
    $repr\_circuit \leftarrow DB\_representative\_to\_xag[representative]$;
else
    return $c$;
end
$new\_cut\_circuit \leftarrow repr\_circuit +$ gates corresponding to operations on inputs and outputs ;
return $new\_cut\_circuit$;
```

Algorithm 5.3: Cut-rewriting to minimize the number of AND gates (multiplicative complexity) of an XAG

Figure 5.4(a), and one inversion. The gates introduced are highlighted in Figure 5.4(b). The final XAG of the full adder in shown in Figure 5.4(c). We can conclude that the full adder has a multiplicative complexity of at most 1. ■

To sum up, we minimized the number of AND gates of a full adder by (i) using the minimum XAG of the representative (Figure 5.4(a)) and (ii) by adding to it the gates corresponding to each operation (Figure 5.4(b)).

The proposed algorithm is based on cut rewriting and is a general version of the DAG-aware AIG rewriting presented in [121]. The work in [121] aims at minimizing the AIG size by iteratively selecting AIG subgraphs and replacing them with smaller pre-computed subgraphs. Our algorithm implements a similar approach, based on cut enumeration [138]. The idea is to replace XAG subgraphs with new graphs which have smaller multiplicative complexity.

For each cut, the minimum representation in term of AND gates can be computed as described in Example 5.3. Algorithm 5.3 presents the pseudocode. The minimum representations over the basis {AND, XOR, NOT} for all affine class representatives up to 6 inputs[6] are used to create a database mapping all representatives up to 6 inputs to their minimum XAG representation. Further, as optimum results are known for functions with up to 6 inputs, the cut enumeration has been restricted to 6-cut. First, the Boolean function of the cut with respect to its leaves is computed. The work presented in [116] is then used to compute the affine class representative and the operations. The XAG of the representatives is retrieved from the database previously stored, and XOR gates, inverters and permutations according to the different operations are added in order to obtain the XAG implementing the correct function. Once the circuit for the cut is obtained, the algorithm continues as in [121]. In our case, the gain is evaluated considering the reduction in the number of AND gates.

---

[6] Available at: *https://github.com/usnistgov/Circuits/tree/master/data/slp*

```
Input: Logic network $N$, cut-size, max_div
Output: Resynthesized logic network
list ← topological-sort-network($N$);
foreach node $v$ in list do
    cut ← find-reconvergent-cut($v$, cut-size);
    mffc ← computeMFFC($v$);
    if $|mffc| > 0$ then
        $div$ ← collect-divisors($list$, $v$, $max\_div$) ;
        compute-truth-tables(cut);
        compute-satisfiability-DC(cut);
        if $v' \leftarrow$ 0-resub($list$, $v$, $div$) then
            continue;
        end
        $and\_mffc$ ← AND-in-MFFC($mffc$);
        if $and\_mffc = 0$ then
            continue;
        end
        if $and\_mffc > 0$ then
            if $v' \leftarrow$ xor-resub($list$, $v$, $div$) then
                continue ;
            end
            if $v' \leftarrow$ xx-resub($list$, $v$, $div$) then
                continue ;
            end
            if $v' \leftarrow$ and-resub($list$, $v$, $div$, $and\_mffc$) then
                continue;
            end
            if $v' \leftarrow$ aa-resub($list$, $v$, $div$, $and\_mffc$) then
                continue;
            end
            if $v' \leftarrow$ ao-resub($list$, $v$, $div$, $and\_mffc$) then
                continue;
            end
        end
    end
end
network-cleanup-and-sweeping($N$);
```

Algorithm 5.4: Resubstitution to reduce the number of ANDs

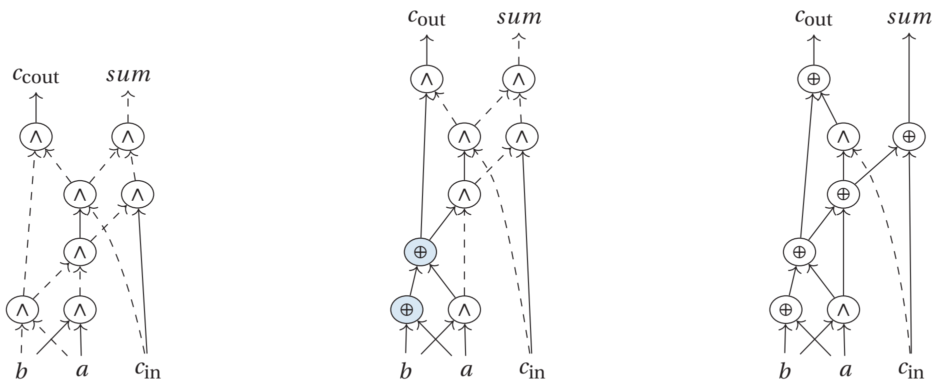
Resubstitution

The second transformation of our synthesis tool is resubstitution. Recall that resubstitution is a method adopted in many logic synthesis flows [19] to express the function of a node v using other nodes (called *divisors*) which are already present in the logic network. A resubstitution is accepted if the new implementation is more compact (e.g., in the number of nodes) than the current one, thus resulting in size optimization. Resubstitution is usually classified according to the number of operators that it adds to the logic network, i.e., 0-resubstitution, if it does not add any new operator; 1-resubstitution if it expresses a logic function by adding one logic operator, and so forth. When k nodes are added by resubstitution, size improvement is obtained if $l > k$, where l is the number of nodes in the *maximum fan-out free cone* (MFFC) of node v. We also refer to “AND-resubstitution”, “OR-resubstitution”, etc., depending on the type of operators added to the network.

Our tool minimizes the number of AND gates in the logic network, independently from the number of XOR gates and inverters. Thus, state-of-the-art resubstitution algorithms need to be re-investigated to take this new cost into account. First, XOR gates do not take part in the total cost and saving, and only the number of AND gates in the MFFC, called hereafter and_mffc, are considered in the global saving for resubstitution. It means that XOR-resubstitution is always advantageous when the number of AND gates in the MFFC is larger than 0. On the other hand, in the case $and\_mffc = 0$, resubstitution is never leading to any AND optimization. Regarding AND/OR resubstitutions, classical implementations can be used, paying attention to evaluate the gain as and_mffc.

The resubstitution procedure is depicted in Algorithm 5.4. For each node in the network (in topological order), the procedure computes a reconvergent-driven cut and the MFFC of v as implemented in [151]. k-resubstitution is intrinsically an expensive task, and it is thus applied to small partitions of the whole network. To accelerate the computation, the divisors are collected by setting a maximum number of nodes max_div. As resubstitution may result in more optimization opportunities when enriched with don’t cares, we allow the use of *satisfiability don’t cares* in the algorithm. Truth tables are used as underlying data structure for the computation of both the Boolean functionality and the don’t cares (lines 7–8 in Algorithm 5.4). First 0-resubstitution is attempted. Due to the use of don’t cares, the algorithm looks for a divisor d_1 such that $DC(v) \vee v = DC(v) \vee d_1$. If this is successful, resubstitution is applied and the procedure moves to the next node; otherwise, more complex types of resubstitution are tried, depending on the number of AND gates in the MFFC. If $and\_mffc = 0$, the procedure jumps to the next node, as resubstitution is not leading to any optimization. In the opposite case, i.e., $and\_mffc > 0$, any XOR-resubstitution is successful independently on the number of increased XORs. Two XOR resubstitutions are implemented: XOR-resubstitution (*xor-resub*) and XOR-XOR-resubstitution (*xx-resub*). In the case *xor-resub* and *xx-resub* are not applicable, standard AND resubstitutions are attempted in increasing complexity order. The tried resubstitutions are: AND-resubstitution (*and-resub*), AND-AND resubstitution (*aa-resub*), AND-OR resubstitution (*ao-resub*). In this scenario, the resubstitution is successful if the number of k added AND nodes is smaller than and_mffc. To accelerate the computation, Boolean filtering rules from [19] have been applied for the AND-resubstitution.

Example 5.4 *As an example, consider the logic network from Figure 5.5(a), which is the XAG of the full adder. By applying XOR-XOR-resubstitution, one AND gate can be written using two XORs (highlighted in Figure 5.5(b)). This example intentionally shows that the algorithm does not consider the increase in the number of nodes as the main cost for optimization, while only the AND gates are accounted for in the optimization process (decreased from* 7 *to* 6*).* ■



(a) Logic network for the full adder (b) Logic network rewritten using resubstitution (c) Logic network rewritten using refactoring

Figure 5.5 – Resubstitution and refactoring example. (b) shows the network after XOR-XOR resubstitution is applied on (a); while (c) shows the result after refactoring. © 2020 IEEE [178]

Refactoring

Refactoring is an effective technique often used to overcome local minima that can be encountered during optimization. As a matter of fact, refactoring resynthesizes large subnetworks in a logic network from scratch and without using existing nodes in the logic network. Depending on the optimization needs and the data structure, different logic synthesis algorithms can be used for this purpose, e.g., [8, 125].

In the presented flow, we aim at minimizing the number of AND gates over an XAG, consequently, a refactoring technique that works over 2-input XOR/AND operators is needed. For this purpose, the algorithm for bi-decomposition proposed by Mishchenko et al. in [125] is used. The algorithm synthesizes a function using OR, AND and XOR gates, together with internal don't cares to allow a better quality of results. The primary goal of optimization in [125] is to obtain a "balanced" network; it means that, when more than one bi-decomposition exists, the algorithm chooses the type of operator (i.e., AND, OR, XOR) that leads to the most-balanced result in terms of the size of the support. In our optimization, we consider a different operator-selection and change the original algorithm to always (when possible) choose the XOR operator over AND and ORs, independently on the size of the supports. This is because the XOR operator does not take part in the total cost and, in this way, the algorithm always picks the XOR operator when more than one bi-decomposition exists.

The refactoring procedure to minimize the number of AND gates is depicted in Algorithm 5.5. It has been implemented following the general guidelines in [152]. For each node in the network, the MFFC is evaluated by setting a limit on the maximum number of PIs (max_fanin), and truth tables are used to compute Boolean functions and satisfiability don't cares. The function is synthesized (line 5 in Algorithm 5.5) by using a modified version of the bi-decomposition from [125], in which the selection of the operators is changed to prefer the

```
Input: Logic network $N$, $max\_fanin$
Output: Resynthesized logic network
foreach node $v$ in $N$ do
    mffc ← computeMFFC($v, max\_fanin$);
    $f$ ← compute-truth-tables($mffc$);
    $dc$ ← compute-satisfiability-DC($mffc$);
    $new\_mffc$ ← synthesize($f, dc$) ;
    if AND-in-MFFC($new\_mffc$) < AND-in-MFFC($mffc$) then
        Substitute($new\_mffc$, $mffc$);
    end
end
network-cleanup-and-sweeping($N$);
```

Algorithm 5.5: Refactoring to reduce the number of ANDs

XOR operator, when a XOR-bi-decomposition exists. If the new implementation of the MFFC has less AND gates, the new MFFC is substituted to the previous one, resulting in a network with reduced multiplicative complexity.

Example 5.5 *As an example, consider the logic network from Figure 5.5(b), which is the XAG implementation of the full adder, obtained after resubstitution. By applying refactoring on each primary output, the network can be factorized as presented in Figure 5.5(c). The new implementation has a smaller number of AND gates, which are decreased to only* 2 *gates.* ■

5.3.3 Experimental Results

The three aforementioned techniques have been implemented to create a complete and automatic logic synthesis toolbox that minimizes the number of AND gates. In this section, first, we detail the implementation of the proposed algorithms, then the experimental results are presented. We test the efficacy of the algorithms on state-of-the-art best-known results both on the EPFL and cryptography and security benchmarks.

Implementation Details

The proposed algorithms have been implemented as part of the open-source logic synthesis framework *mockturtle*, which is part of the EPFL logic synthesis libraries [165].[7] All the experiments have been carried out on an Intel Xeon E5-2680 CPU with 2.5 GHz and with 256 GB of main memory.

Regarding the rewriting algorithm, the maximum number of leaves for each cut is equal to 6 (Algorithm 5.3). Thus, as we are dealing with 6-input functions, we make use of truth tables to represent the Boolean functions. Truth tables for 6-input functions can be efficiently

---

[7] Available at: *github.com/lsils/mockturtle*. Experiments are available at: *github.com/lsils/date2020_experiments* and *https://github.com/eletesta/dac19-experiments*

stored in computers as a single 64-bit unsigned integer, and are fast to compute. Further, our algorithm allows us to limit the maximum number of cuts computed for each node. In our experimental evaluation, we found that a cut limit of 12 leads to a good trade-off between runtime and quality.

The database stores the XAGs for each representative. In practice, this can be stored as one "large" XAG, called hereafter XAG_DB. XAG_DB has 6 inputs, and 147 998 outputs (this number is explained below). Each output is the XAG of one representative. The total size of this XAG is 2 339 563. XAG_DB is created once and can be reused for several rewriting calls. The *database_to_xag* function in Algorithm 5.3 maps the truth table of each representative to its corresponding output in XAG_DB.

The work presented in [116] is used to calculate the affine representative and the required operations. The classification is performed by rearranging the coefficients of the function's Rademacher-Walsh spectrum [73] based on their magnitudes. Depending on the distribution of coefficients, the number of iterations to reach the representative can vary significantly among different functions. In most cases, a representative is found very quickly, but for some functions this computation can be inefficient. We address this problem using two techniques. First, we maintain a cache of computed representatives and affine operations for all considered Boolean functions during rewriting. Therefore, no Boolean function needs to be classified twice. Also, we put an iteration limit on the classification routine, which causes us to omit some Boolean functions from rewriting. In our experiments, we consider 147 998 of all 150 357 affine equivalence classes.

For resubstitution, we fixed the maximum number of divisors to 100, and the maximum number of inputs for computing a cut to 8. Don't cares may be used to trade off runtime and quality of results. In our case, we always use don't cares within resubstitution. The maximum number of PIs for the refactoring MFFC was set to 15; as in the previous case, don't cares are also used for refactoring.

The three presented algorithms can be applied separately or in a global flow, which alternates between the three proposed techniques. In the following experiments, we optimize benchmarks using the following heuristic: rewriting method until saturation of the baseline results, followed by resubstitution, refactoring and rewriting interleaved in this order until further saturation of the results. All optimized benchmarks are verified to be formally equivalent to the original ones. It is worth mentioning that we do not apply any XOR optimization; nevertheless, in some protocols, XORs involve a communication overhead [10]. An algorithm to minimize the number of XORs for crypthography applications can be found in [33].

Results over EPFL Benchmarks

In this experiment, we demonstrate that our method decreases the number of AND gates when applied to benchmarks optimized using state-of-the-art generic size optimization.

Table 5.4 – Experimental results for EPFL benchmarks using rewriting

Name	Inputs	Outputs	Baseline*		Rewriting				Repeat until convergence			
			AND	XOR	AND	XOR	time [s]	impr.	AND	XOR	time [s]	impr.
Adder	256	129	550	255	318	529	3.74	42 %	128	549	5.36	77 %
Barrel shifter	135	128	2688	0	896	1728	15.41	67 %	832	1728	16.65	69 %
Divisor	128	128	12001	3897	6378	8779	100.83	47 %	6060	8994	1132.23	50 %
Log2	32	32	24941	3592	19942	8583	327.34	20 %	19436	9371	11988.6	22 %
Max	512	130	2687	0	1471	1387	17.36	45 %	931	1479	81.82	65 %
Multiplier	128	128	16119	4301	12209	8122	169.97	24 %	11940	8614	9202.11	26 %
Sine	24	25	4937	519	4194	1572	56.76	15 %	4075	1770	405.47	17 %
Square-root	128	64	12336	3746	7101	9122	103.35	42 %	6244	9640	418.98	49 %
Square	64	128	9225	3850	5323	7984	34.34	42 %	5181	8084	158.92	44 %
Norm. geom. mean			1		**0.60**				**0.49**			
Round-robin arbiter	256	129	1181	0	1181	0	17.85	0 %	//	//	//	0 %
Alu control unit	7	26	86	2	85	8	0.7	1 %	85	8	1.22	1 %
Coding-cavlc	10	11	536	16	507	152	10.05	5 %	494	197	23.86	8 %
Decoder	8	256	341	0	341	0	0	0 %	//	//	//	0 %
i2c controller	147	142	823	15	659	342	17.22	20 %	623	502	109.12	24 %
int to float converter	11	7	133	13	112	76	1.82	16 %	100	101	4.66	25 %
Memory controller	1204	1231	7418	361	5393	3165	89.62	27 %	5113	4168	2592.97	31 %
Priority encoder	128	8	368	0	327	158	4.12	11 %	327	158	8.1	11 %
Lookahead XY router	60	30	96	0	96	0	1.6	0 %	//	//	//	0 %
Voter	1001	1	7308	1833	6046	4917	55.74	17 %	5651	6066	262.21	23 %
Norm. geom. mean			1		**0.90**				**0.87**			

*Note that the baseline here is obtained using ABC. “Norm. geom. mean” is the normalized geometric mean.

Table 5.4 shows the results for AND minimization over the EPFL benchmarks, using the rewriting algorithm. On average, we decrease the number of AND gates of 34% using only rewriting. The arithmetic benchmarks benefit more from our method and are optimized up to 77% in the number of AND gates.

We present our results on the EPFL benchmark suite [15], and we use the synthesis package ABC [40] as baseline for our comparison. In case of the EPFL benchmarks, our starting points are the best-known size-optimized 6-LUT benchmarks.[8] As state-of-the-art size optimization, we apply a synthesis script that interleaves priority-cut-based 2-LUT mapping (*&if*) [123], structural choices (*&dch* and *&synch2*) [54, 122], and Boolean network optimization and resynthesis (*&mfs*) [120]. We apply the synthesis script

```
&st; &synch2; &if -m -a -K 2; &mfs -W 10;
  &st; &dch; &if -m -a -K 2; &mfs -W 10
```

ten times, and we pick the final result as our baseline. The result is a 2-LUT network, i.e., a logic network in which each gate corresponds to an arbitrary 2-input function. Note that a 2-LUT network can be directly translated into an XAG without increasing the number of gates by choosing inverters appropriately. Therefore, it provides us with a good starting point, despite the fact that it uses a unit cost model that accounts the same cost for both AND and XOR gates.

---

[8] See version v2018.1 on *https://github.com/lsils/benchmarks*

The results for the rewriting algorithm are shown in Table 5.4. The initial benchmarks are generated as previously discussed. The **Rewriting** results are obtained by applying one iteration of our proposed rewriting method (see Algorithm 5.3), while the **Repeat until convergence** results show the number of AND and XOR gates after more iterations of our rewriting algorithm. In this last case, the algorithm is run until no further improvement is obtained by rewriting. A ' // ' entry in Table 5.4 indicates that no improvement was possible even with applying a single iteration of our proposed method. On average, 15 iterations are needed before convergence. The maximum number of iterations encountered by our rewriting algorithm is equal to 58 (*multiplier* benchmark). The experiments show that the number of AND gates reaches a local minimum for all benchmarks, and the normalized geometric mean decreases both for arithmetic and random-control benchmarks. The total improvement is shown in the last column of both the **One round** and **Repeat until convergence** results. On average, we decrease the number of AND gates of 34% using only rewriting. The arithmetic benchmarks benefit more from our method and are optimized up to 77% in the number of AND gates. On the contrary, the random-control benchmarks are optimized 23% on average.

The experiments of resubstitution and refactoring, followed by the complete flow, are shown in Table 5.5. As baseline, we use the results presented in Table 5.4, that are obtained applying the rewriting algorithm until convergence of the results is achieved. In the following, we thus present separately the results of resubstitution and refactoring, as the rewriting algorithm is not leading to any further optimization if applied separately (i.e., not as part of the complete flow) on the baseline. The column **Resubstitution** presents results when Algorithm 5.4 is applied once on top of the results obtained after rewriting, while **Refactoring** shows the improvements achieved by applying once Algorithm 5.5. The complete flow shows the results when the three techniques (i.e., rewriting, refactoring, and resubstitution) are applied in the given order until convergence is reached. It means, no further optimization is achieved with any of the proposed algorithms. The runtime of the complete flow is evaluated as an average runtime obtained dividing the total runtime by the number of iterations, where each iteration consists of the three presented techniques. Even though, as a general trend, the results in Table 5.5 show that resubstitution achieves better optimization compared to refactoring, for few benchmarks (e.g., *mult, sin, log2*) refactoring largely overcomes the results achieved with resubstitution. As expected, for the *adder* benchmark no further optimization is obtained as the rewriting result is optimum [34]. More interestingly, none of the presented techniques manages to optimize the *bar* benchmark. On average, the complete flow optimizes the best-known results up to 47%, with a geometric mean of 0.81 and 0.85 for the arithmetic and random-control benchmarks, respectively. For the *max*, *arbiter*, *decoder*, and *router* benchmarks, the results of the complete flow are entirely obtained by resubstitution. On average, 4 iterations are needed to reach the convergence of the results for the complete flow.

Table 5.5 – Experimental results for EPFL benchmarks using (i) resubstitution, (ii) refactoring and (iii) the complete flow. © 2020 IEEE [178]

Benchmark	Baselineˆ		Resubstitution			Refactoring			Complete flow			
	AND	XOR	AND	XOR	impr.	AND	XOR	impr.	AND	XOR	impr.	time [s]
Adder *	128	549	//	//	0%	//	//	0%	//	//	0%	1.76
Barrel shifter	832	1728	//	//	0%	//	//	0%	//	//	0%	8.38
Divisor	6060	8994	5844	9053	4%	5691	8562	6%	5291	8678	13%	129.60
Log2	19436	9371	17240	10644	11%	12360	15633	36%	10913	15923	44%	732.69
Max	931	1479	890	1520	4%	//	//	0%	890	1520	4%	8.48
Multiplier	11940	8614	11623	8120	3%	7941	12272	33%	7653	11855	36%	233.45
Sine	4075	1770	3390	2157	17%	3236	2444	21%	2603	2709	36%	70.00
Square-root	6244	9640	5927	9562	5%	6023	9148	4%	5381	9260	14%	148.38
Square	5181	8084	4929	8140	5%	5011	8076	3%	4672	8198	10%	84.57
Norm. geom. mean	1		**0.94**			**0.87**			**0.81**			
Round-robin arbiter	1181	0	1174	7	1%	//	//	0%	1174	7	1%	21.29
Alu control unit	85	8	53	36	38%	69	24	19%	45	49	47%	1.62
Coding-cavlc	494	197	414	246	16%	476	215	4%	394	267	20%	17.20
Decoder	341	0	328	13	4%	//	//	0%	328	13	4%	15.16
i2c controller	623	502	586	345	6%	588	535	6%	557	375	11%	11.99
int to float converter	100	101	91	85	9%	99	102	1%	85	88	15%	2.20
Memory controller	5113	4168	4923	3262	4%	4893	4328	4%	4695	3401	8%	135.07
Priority encoder	327	158	326	159	0.3%	324	161	1%	323	162	1%	7.36
Lookahead XY router	96	0	93	6	3%	//	//	0%	93	6	3%	2.81
Voter	5651	6066	4802	5759	15%	5257	6160	7%	4257	5990	25%	92.61
Norm. geom. mean	1		**0.90**			**0.96**			**0.85**			

ˆ Note that the baseline here are the results obtained in Table 5.4. *These results are known to be optimum [34], we thus do not expect any further optimization for the number of AND gates. "Norm. geom. mean" is the normalized geometric mean.

Table 5.5 shows the results for AND minimization over the EPFL benchmarks, using the resubstitution, refactoring, and the complete flow. On average, the complete flow further optimizes the best-known results up to 47%**, with a geometric mean of** 0.81 **and** 0.85 **for the arithmetic and random-control benchmarks, respectively.**

MPC and FHE Benchmarks for Cryptographic Applications

In this section, we demonstrate our approach in the context of MPC and FHE, by optimizing the number of AND gates for best-known reported benchmarks.[9] Both these cryptographic applications benefit from AND gates minimization. XOR gates and inverters are almost for free, while AND gates are considered more expensive in both cases [10].

The results are presented as in the previous section. It means, the first part of the results are shown in Table 5.6. As in the previous case, we distinguish between **Rewriting** results and **Repeat until convergence**. The first four benchmarks are block ciphers, followed by three hash functions, and seven arithmetic functions. Of most interest is the improvements in the block ciphers and hash functions. No improvement is possible with our rewriting technique in both variants of the AES block cipher, which indicates that the reported number of AND gates may be close to the multiplicative complexity of the function. An improvement of 17%

---

[9] Available at: https://web.archive.org/web/20190105040458/homes.esat.kuleuven.be/~nsmart/MPC/

Table 5.6 – Experimental results for MPC and FHE benchmarks using the rewriting algorithm

Name	Inputs	Outputs	Baseline*		Rewriting				Repeat until convergence			
			AND	XOR	AND	XOR	time [s]	impr.	AND	XOR	time [s]	impr.
AES (No Key Expansion)	256	128	6800	25124	6800	25124	37.48	0 %	//	//	//	0 %
AES (Key Expansion)	1536	128	5440	20325	5440	20325	27.32	0 %	//	//	//	0 %
DES (No Key Expansion)	128	64	18124	1337	17404	4096	251.57	4 %	15093	11105	8876.11	17 %
DES (Key Expansion)	832	64	18175	1348	17403	4168	256.69	4 %	15126	11263	9262.73	17 %
MD5	512	128	29084	14133	12300	29270	101.53	58 %	9381	30325	145.44	68 %
SHA-1	512	160	37172	24166	17141	42415	114.55	54 %	11820	44311	293.8	68 %
SHA-256	512	256	89478	42024	52921	86304	311.68	41 %	30201	91278	12562.8	66 %
32-bit Adder	64	33	127	61	38	146	0.83	70 %	32	150	0.98	75 %
64-bit Adder	128	65	265	115	100	260	2.06	62 %	64	284	2.61	76 %
32x32-bit Multiplier	64	64	5926	1069	4290	2351	57.19	28 %	4107	2473	135.02	31 %
Comp. 32-bit Signed LTEQ	64	1	150	0	121	69	3.65	19 %	114	89	6.3	24 %
Comp. 32-bit Signed LT	64	1	150	0	129	74	3.9	14 %	108	116	10.17	28 %
Comp. 32-bit Uns. LTEQ	64	1	150	0	121	69	3.23	19 %	114	89	6.38	24 %
Comp. 32-bit Uns. LT	64	1	150	0	129	74	4.04	14 %	108	116	10.59	28 %
Norm. geom. mean			1		**0.68**				**0.56**			

*Note that the baseline here is obtained using ABC. “Norm. geom. mean” is the normalized geometric mean.

Table 5.6 shows the results for AND minimization over the cryptography benchmarks, using the rewriting algorithm. Our rewriting method optimizes the number of AND gates needed to implement the 32- and 64-bit adders down to 32 and 64, respectively. These are known to be optimum [34] in the number of AND gates.

Table 5.7 – Experimental results for MPC and FHE benchmarks using (i) resubstitution, (ii) refactoring and (iii) the complete flow © 2020 IEEE [178]

Benchmark	Baseline^		Resubstitution			Refactoring			Complete flow			
	AND	XOR	AND	XOR	impr.	AND	XOR	impr.	AND	XOR	impr.	time [s]
AES (No Key Expansion)	6800	25124	//	//	0%	//	//	0%	//	//	0%	383.87
AES (Key Expansion)	5440	20325	//	//	0%	//	//	0%	//	//	0%	274.50
DES (No Key Expansion)	15093	11105	9840	12347	35%	11413	14658	24%	9048	13092	40%	532.85
DES (Key Expansion)	15126	11263	10078	12291	33%	11595	14691	23%	9205	13136	39%	536.07
MD5	9381	30325	9374	29743	0.1%	9380	30326	0.01%	9367	29729	0.1%	520.94
SHA-1	11820	44311	11684	44355	1%	11776	44302	0.4%	11515	44358	3%	996.70
SHA-256	30201	91278	29145	91578	3%	29933	91254	1%	26927	91495	11%	4316.69
32-bit Adder *	32	150	//	//	0%	//	//	0%	//	//	0%	0.41
64-bit Adder *	64	284	//	//	0%	//	//	0%	//	//	0%	0.92
32x32-bit Multiplier	4107	2473	4060	2456	1%	2650	3866	35%	1689	3861	59%	27.78
Comp. 32-bit Signed LTEQ	114	89	98	98	14%	114	89	0%	92	97	19%	3.25
Comp. 32-bit Signed LT	108	116	96	100	11%	108	116	0%	92	95	15%	4.13
Comp. 32-bit Uns. LTEQ	114	89	98	98	14%	114	89	0%	92	97	19%	3.26
Comp. 32-bit Uns. LT	108	116	96	100	11%	108	116	0%	92	95	15%	4.11
Norm. geom. mean	1		**0.90**			**0.93**			**0.82**			

^ Note that the baseline here are the results obtained in Table 5.6. *These results are known to be optimum [34], we thus do not expect any further optimization for the number of AND gates. “Norm. geom. mean” is the normalized geometric mean.

Table 5.7 shows the results for AND minimization over the cryptography benchmarks, using the resubstitution, refactoring, and the complete flow. Our method further optimizes the number of AND gates up to 59%, reducing it 2.4× **(*multiplier* benchmark).**

was possible in the case of the DES cipher, while a much larger improvement was possible for all three hash functions, with more than 66% improvement after repeating the proposed approach until convergence. It is worth noticing that the our rewriting method optimizes the number of AND gates needed to implement the 32-bit adder down to 32, which is known to be optimum [34] in the number of AND gates. The same applies for the 64-bit case.

The experimental results for refactoring, resubstitution and the complete flow are shown in Table 5.7. As in the previous case, results are obtained with (i) resubstitution, (ii) refactoring, and (iii) a complete flow that alternates between rewriting, refactoring, and resubstitution until convergence of the results. The runtime is evaluated as discussed in the previous section. These algorithms are applied on top of the results presented in Table 5.6 (used as **Baseline**). Both the 32- and 64-adder cannot be further optimized because their number of AND gates is optimum [34]. For the AES benchmarks, which were not optimized by the previous method, the global flow does not find any optimization opportunity. On the other hand, the tool can further optimize most of the previous best results. In particular, for the multiplier, it further optimizes the number of AND gates up to 59%, reducing it 2.4×. As for the EPFL benchmarks, this optimization mostly comes from refactoring, while resubstitution does not find many opportunities (1%) for the multiplier. As a general trend, resubstitution obtains better results compared to refactoring on most of the cryptography benchmarks (e.g., on the comparators). On average, the complete flow achieves substantial optimizations, with a geometric mean of 0.82. For the *SHA-256* benchmark, 21 iterations are needed for the saturation of the results.

Our tool's main goal is to optimize the number of AND gates in cryptography applications that do not account for XORs and inverters in their cost function. We exercised the proposed flow on a second group of cryptography benchmarks in the context of MPC presented in [149].[10] This second set consists of practical MPC problems. We compared the proposed flow to the flow in [149] on the MPC circuits. Table 5.8 lists the results. Beside the name of the benchmark, the table shows the number of AND gates obtained using the original approach as reported in [149]. In this case, we did not apply the proposed approach on top of their optimized benchmarks, but compared the flows directly. The benchmarks in [149] are specified in register-transfer-level Verilog code, which we transformed using *Yosys* [190] and *ABC* [40] into a format that can be read by our tool. The proposed complete flow applies rewriting, refactoring, and resubstitution until convergence. The resulting number of AND gates are reported in the third column of the table, with the percentage improvement compared to the original flow given by the last column. We do not report the number of XORs because the results reported in [149] includes both XOR and inverters, while in our case it only includes XORs. The results in Table 5.8 show that the proposed flow achieves significant improvements. For one benchmark, our result is far beyond the flow in [149], meaning that some optimization opportunities are not found by the proposed method. On the *auction* benchmarks, results are close to the ones obtained running the flow in [149]; on the other hand, for the *voting* and *secure k-nearest neighbor search* (knn) benchmarks, results are improved by 25% and 17% on

---

[10] Available at: *github.com/sadeghriazi/MPCircuits*

Table 5.8 – Experimental results for MPC benchmarks [149] © 2020 IEEE [178]

Benchmark	Flow in [149]	Proposed complete flow	impr.
	AND	AND	impr.
knn_comb_K_2_N_16	2370	1919	19.0%
knn_comb_K_1_N_16	1160	1162	-0.2%
knn_comb_K_1_N_8	556	554	0.4%
knn_comb_K_2_N_8	1080	881	18.4%
knn_comb_K_3_N_8	1520	1060	30.3%
knn_comb_K_3_N_16	3500	2394	31.6%
voting_N_1_M_3	8	7	12.5%
voting_N_3_M_4	388	275	29.1%
voting_N_2_M_4	147	104	29.3%
voting_N_2_M_3	79	55	30.4%
voting_N_1_M_4	16	15	6.3%
voting_N_2_M_2	37	21	43.2%
auction_N_3_W_16	228	232	-1.8%
auction_N_2_W_16	97	97	0.0%
auction_N_3_W_32	454	456	-0.4%
auction_N_4_W_16	492	495	-0.6%
auction_N_4_W_32	975	975	0.0%
auction_N_2_W_32	194	193	0.5%
secure_s_m_M_4_N_16	16700	16001	4.2%
secure_s_m_M_8_N_16	46660	58723	-26.0%
Normalized geom. mean		**0.87**	

Table 5.8 shows the results for AND minimization over MPC benchmarks, using the complete flow. For the *voting* and *knn* benchmarks, results are improved by 25% **and** 17% **on average, respectively.**

average, respectively. Results for the *private set intersection (bitwise-AND)* are not reported as neither the flow in [149] nor the complete flow managed to optimize them.

5.4 Summary

This chapter focused on more practical aspects of logic synthesis, concentrating on XOR-logic. First, we presented a practical algorithm to synthesize CMOS-based circuits with better area and performances, without increasing the runtime budget of the logic synthesis flow. The algorithm uses BDDs and the Boolean difference between two functions to decrease the size of the underlying AIG. We showed significant synthesis results. We obtained remarkable improvements of the smallest known AIGs for EPFL benchmarks, and we improved 12 of the best-known area results in the EPFL synthesis competition. We demonstrated -3.12% combinational area savings and -2.49% dynamic power reduction, after physical implementation, at contained runtime cost for a commercial EDA flow over 36 ASICs designs. In the second part, we addressed XOR-based logic synthesis for cryptography and security. We proposed different algorithms to reduce the number of AND gates (called multiplicative complexity) in an XAG, which is a logic network composed of AND, XOR, and inverter gates. Such XAG optimization plays a central role in cryptography applications such as FHE and MPC. For both these applications, XOR gates and inverters are for free, while AND gates are considered slower and more expensive. Nevertheless recent work in this field, there is still a lack of logic synthesis tools to automatically and efficiently optimize the multiplicative complexity of cryptography circuits. In this thesis, we have presented a complete and automatic logic synthesis toolbox to address these alternative applications. The tool alternates between rewriting, refactoring, and resubstitution techniques, precisely modified to focus on the minimization of the number of AND gates in an XAG. Our tool achieves significant results over both EPFL and cryptography best-known results. Our rewriting experiments show that we can reduce the number of AND gates by 34% on average when compared to generic size optimization. We demonstrate an average further improvement of 15% for the EPFL benchmarks when using refactoring and resubstitution. We also demonstrate improvement in best-known benchmarks for MPC and FHE applications.

6 Conclusions

In this thesis, we investigated novel data structures and algorithms for logic synthesis, focusing on both standard and emerging technologies. Motivated by the (i) many and diverse ways of computation, alternative to CMOS, presented in the last years; and (ii) the modern computing and memory means, we extended logic synthesis to abstract the constraints given by modern nanotechnologies, as well as optimized standard logic synthesis flow in light of modern computing capabilities. We also augmented standard logic synthesis algorithms to consider novel applications in cryptography and security.

The results presented in this thesis demonstrate the need for logic synthesis to be revisited to consider the variety of modern primitives and novel engines that can be of interest, and, consequently, the corresponding objective metrics and optimization goals. In the following, we present the main thesis contributions and directions for future work.

6.1 Summary of Thesis Contributions

We introduce the main contributions of this thesis in the order of the chapters.

- **Chapter 3 – Majority Logic for Emerging Technologies [151, 180, 181].** Motivated by the many emerging technologies that naturally implement the 3-input majority as the main building block, we demonstrated novel optimization methods over MIGs. We introduced novel logic synthesis size optimizations that achieve more than 18% reduction in the number of MIG gates, as well as remarkable results after technology mapping to *quantum-dot cellular automata* (QCA) and *spin torque majority gate* (STMG). We further demonstrated how technology-dependent optimization steps are needed to investigate the limits and capabilities of emerging technologies. For instance, since STMG technology can efficiently implement a majority gate, but it cannot realize inverters, we proposed an algorithm to move inversions to PI. This achieves an inversion-free circuit, that can be fully implemented using STMG gates. We further studied novel exact synthesis algorithms to abstract problems with many and complex constraints that need to be met at the same time. We demonstrated that tailored changes to existing algorithms allow us to fully abstract the constraints given by emerging technologies such as plasmonic devices.

- **Chapter 4 – Theoretical Results for Majority-*n* Logic [166, 172, 177].** Motivated by our research on majority logic, we presented novel theoretical results on the decomposition of majority-*n* functions into 3-input majority gates. We introduced a novel method to directly map *binary decision diagram*s (BDDs) of monotone functions into 3-input majority graphs. Our method allows us to find novel upper bounds on the size-optimum results for these functions, as well as novel decompositions. We proposed the optimum implementation of majority-5 and majority-7, showing the whole derivation by using algebraic transformations of MIGs. Moreover, we demonstrated the best-known majority-9 implementation with 12 nodes. This is a remarkable result if you consider that the optimum realization of the majority-9 is still unknown. Being the majority both self-dual and monotone, we investigated the complexity of such class of functions over 7 inputs. Our results demonstrated that for the majority function over 7 inputs, the complexity is invariant when (i) inverters, or (ii) leafy constraints are considered. We also showed that this is not the case for all self-dual monotone 7-input functions, as the inverters have in general a positive effect in decreasing their complexity.
- **Chapter 5 – XOR-logic for Standard Logic Synthesis Flows [170, 171].** We introduced a novel size optimization method for standard CMOS technologies, that takes advantage of the novel and modern computing capabilities of industrial logic synthesis flows. In contrast to the theoretical results of the previous part, this section aims at building a functional and efficient flow for industrial purposes. We demonstrated how XOR-based techniques based on the Boolean difference, together with a modern BDD package allow us to obtain significant size and power optimization over 36 ASIC design, evaluated after P&R. We showed that many unexplored optimizations opportunities can still be revealed by using Boolean methods and pushing the computing performances of modern flows. Embedding our algorithm in an industrial flow achieved a total area reduction of -3.12% and switching power of -2.49%, on average, after physical design.
- **Chapter 5 – XOR-logic for Cryptography and Security Appications [175, 178].** Motivated by the remarkable results given by XOR optimization for standard CMOS circuits, we further investigated XOR-based logic synthesis for cryptography and security applications. We proposed a complete and automatic flow that works over XAGs to reduce the number of AND gates in the network. Heuristic state-of-the-art methods are not able to address this alternative cost, which is fundamental for cryptography applications such as MPC and FHE. We presented a tool consisting of standard logic synthesis transformations, i.e., rewriting, resubstitution, and refactoring, specifically changed and designed to consider the minimization of ANDs as the primary goal in the optimization. We presented remarkable results over both EPFL benchmarks and best-known results from the cryptography community. As an example, our tool obtained the optimum result for 32- and 64-bit adders.

6.2 Open Problems

Here, we give some ideas for future research and perspectives.

- **Logic Synthesis Flow for MIGs** In Chapter 3, we presented a size optimization flow for MIGs. In this direction, much work is still needed to be able for MIG-based tools to be fully competitive in the number of optimization opportunities and quality of results with AIGs state-of-the-art tools such as ABC. Further direction for research could continue the work started by the *EPFL Logic Synthesis Libraries* in the direction of building a complete flow for MIGs [153].

  Future work in the same direction could also include an automatic and complete flow for size optimization of majority-3 graphs obtained with the BDDs method presented in Chapter 4. Indeed, preliminary results on the majority-9 demonstrated that promising results are obtained by the automated flow as compared to the ones of the manual decomposition strategy. How to automate the process of optimizing the baseline majority-3 network derived from its BDD remains still to be studied.

- **Mapping of Emerging Technologies** While we focused on some of the constraints set by emerging technologies in Chapter 3, many other constraints and technology mapping requirements need to be addressed. For instance, QCA technology has strict constraints on the wiring and timing of the cells. Logic synthesis and EDA tools are far from being able to abstract these circuits and efficiently work over emerging technologies. Considering the large variety of different emerging technologies, this leaves many research opportunities in this direction still unexplored.

- **Plasmonic-based Logic**: For the plasmonic-based logic presented in Chapter 3, we mainly took into consideration small circuits. The techniques shown in Chapter 3 can be used and extended to consider this technology and its constraints. At the time of this thesis, plasmonic-logic circuits are still in their infancy, and detailed results on area, delay, and energy still under investigation [196]. Preliminary results on plasmonic-based logic for multiplier architectures have been recently proposed in [173]. Future work on logic synthesis for plasmonic-based logic should investigate technology mapping algorithms using area, delay, and energy results to unlock and study the possibilities given by this alternative way of computation.

- **Task-specific MIG Encoding**: In this thesis, we presented exact synthesis methods for three different tasks: (i) abstract the constraints of emerging technologies (Chapter 3), (ii) find upper bounds and optimum solutions for majority-n (Chapter 4), (iii) study the complexity of self-dual monotone 7-input functions (Chapter 4). In all these cases, we used and modified existing MIG encodings for our needs. While this allowed us to obtain important results, it was not always the best choice for runtime requirements. As an example, the complexity of a 7-input function over majority Boolean chain with inverters is still running after 10 months. Future work may include revisiting them and build encoding specifically for our requirements. This could also help in finding the size-optimum implementation for the majority-9 function over MIGs.

- **Majority-n Logic for Arithmetic Circuits**: An interesting application of majority-n logic worth being explored is arithmetic circuits. In particular, the depth and size properties of arithmetic circuits could be addressed. It has been demonstrated that many arithmetic Boolean operations, such as addition, multiplication, and division, are contained in the complexity class TC_0, which means that they have an efficient realization with polynomial-size and constant depth networks, using unbounded threshold gates [88, 146]. The possibility to built such arithmetic components using majority-n graphs with polynomial depth, and restricted number of inputs is a possible future direction for research. Novel bounds on the number of inputs for polynomial depth are worth being considered.
- **New Logic Primitives**: Our thesis focused on majority- and XOR-based logic synthesis. Studies to understand which elementary functions have a major impact and can lead to higher improvements in logic synthesis should also be conducted. This is also important in light of future emerging technologies that could be built upon different primitives. Preliminary results in this direction will be submitted for publications soon [111].

In this thesis, we extended logic synthesis to approach standard as well as unconventional applications. Our synthesis methods demonstrated that many optimization opportunities are still unexplored for conventional logic synthesis flows. Besides, we believe that the material presented in this thesis opens novel research paths in logic synthesis for emerging technologies and cryptography and security applications.

Bibliography

[1] Hidden-weighted bit (hwb) functions. *IEEE Trans.*, (C-40):208–210, 1991.

[2] Cadence Genus Synthesis Solution, datasheet available at https://www.cadence.com/content/dam/cadence-www/global/en_us/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf, 2018 release.

[3] Mentor Graphics Oasys-RTL, datasheet available at http://s3.mentor.com/public_documents/datasheet/products/ic_nanometer_design/place-route/realtime-designer/realttime-designer.pdf, 2018 release.

[4] Synopsys Design Compiler Graphical, datasheet available at https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/dc-graphical-ds.pdf, 2018 release.

[5] Synopsys Fusion Compiler, datasheet available at https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/fusion-compiler-ds.pdf, 2018 release.

[6] Dewmini Sudara Marakallage, Private communication, 2019.

[7] M. Agrawal, E. Allender, and S. Datta. On TC0, AC0, and arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395 – 421, 2000.

[8] S. B. Akers. Synthesis of combinational logic using three-input majority gates. In *Annual Symposium on Switching Circuit Theory and Logical Design*, pages 149–158, 1962.

[9] S. B. Akers Jr. A truth table method for the synthesis of combinational logic. *IEEE Trans. Electronic Computers*, 10(4):604–615, 1961.

[10] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454, 2015.

[11] N. Alon. and J. Bruck. Explicit constructions of depth-2 majority circuits for comparison and addition. *SIAM J. Discrete Math.*, 7(1):1–8, 1994.

[12] S. Amarel, G. E. Cooke, and R. O. Winder. Majority gate networks. *IEEE Trans. Electronic Computers*, 13(1):4–13, 1964.

[13] L. G. Amarù, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli. A sound and complete axiomatization of majority-n logic. *IEEE Trans. on Computers*, 65(9):2889–2895, 2016.

[14] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *Design Automation Conference*, pages 194:1–194:6, 2014.

[15] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. The EPFL combinational benchmark suite. In *Int'l Workshop on Logic and Synthesis*, 2015.

[16] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A new paradigm for logic optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(5):806–819, 2016.

[17] L. G. Amarù, P.-E. Gaillardon, and G. D. Micheli. BDS-MAJ: a BDD-based logic synthesis tool exploiting majority logic decomposition. In *Design Automation Conference*, pages 47:1–47:6, 2013.

[18] L. G. Amarù, P.-E. Gaillardon, S. Mitra, and G. De Micheli. New logic synthesis as nanotechnology enabler. *Proceedings of the IEEE*, 103(11):2168–2195, 2015.

[19] L. G. Amarù, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. De Micheli. Improvements to Boolean resynthesis. In *Design, Automation and Test in Europe*, pages 755–760, 2018.

[20] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, 2009.

[21] G. Audemard and L. Simon. Glucose and Syrup in the SAT Race 2015. In *Reports on the SAT 2015 Competition*, 2015.

[22] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Constraint Programming*, pages 108–122, 2003.

[23] W. L. Barnes, A. Dereux, and T. W. Ebbesen. Surface plasmon subwavelength optics. *Nature*, 424(6950):824–830, 2003.

[24] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(6):723–740, 1988.

[25] K. E. Batcher. Sorting networks and their applications. In *AFIPS Sprint Joint Computing Conference*, pages 307–314, 1968.

[26] L. Berger. Emission of spin waves by a magnetic multilayer traversed by a current. *Physical Review B*, 54(13):9353–8, 1996.

[27] E. R. Berlekamp and L. R. Welch. Weight distributions of the cosets of the (32,6) Reed-Muller code. *IEEE Trans. on Information Theory*, 18(1):203–207, 1972.

[28] D. Bhattacharjee and A. Chattopadhyay. Synthesis, technology mapping and optimization for emerging technologies. In *Annual Symp. on VLSI*, pages 369–374, 2018.

[29] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[30] G. Birkhoff and S. A. Kiss. A ternary operation in distributive lattices. *Bulletin of the American Mathematical Society*, 53(8):749–752, 1947.

[31] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Computers*, 45(9):993–1002, 1996.

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

[33] J. Boyar, P. Matthews, and R. Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, 2013.

[34] J. Boyar and R. Peralta. Tight bounds for the multiplicative complexity of symmetric functions. *Theoretical Computer Science*, 396(1–3):223–246, 2008.

[35] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Int'l Symp. on Experimental Algorithms*, pages 178–189, 2010.

[36] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-Box. In *International Information Security Conference*, pages 287–298, 2012.

[37] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

[38] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[39] R. K. Brayton and C. T. McMullen. The decomposition and factorization of Boolean expressions. In *Int'l Symp. on Circuits and Systems*, pages 49–54, 1982.

[40] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40, 2010.

[41] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(6):1062–1081, 1987.

[42] S. Breitkreutz, J. Kiermaier, I. Eichwald, C. Hildbrand, G. Csaba, et al. Experimental demonstration of a 1-bit full adder in perpendicular nanomagnetic logic. *IEEE Trans. on Magnetics*, 49(7):4464–4467, 2013.

[43] F. Brglez, D. Bryan, J. Calhoun, G. Kedem, and R. Lisanke. Automated synthesis for testability. *IEEE Transactions on Industrial Electronics*, 36(2):263–277, 1989.

[44] F. M. Brown. *Boolean reasoning: the logic of Boolean equations*. Springer Science & Business Media, 2012.

[45] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[46] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

[47] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40(2):205–213, 1991.

[48] C. Calik, M. S. Turan, and R. Peralta. The multiplicative complexity of 6-variable Boolean functions. Cryptology ePrint Archive, Report 2018/002, 2018.

[49] R. Camposano and W. Wolf. *High-level VLSI synthesis*, volume 136. Springer Science & Business Media, 2012.

[50] D. Canright and L. Batina. A very compact "Perfectly Masked" S-Box for AES. In *Int'l Conf. on Applied Cryptography and Network Security*, pages 446–459, 2008.

[51] S.-C. Chang, L. P. Van Ginneken, and M. Marek-Sadowska. Circuit optimization by rewiring. *IEEE Trans. on Computers*, 48(9):962–970, 1999.

[52] M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Conference on Computer and Communications Security*, pages 1825–1842, 2017.

[53] S. Chatterjee and A. Mishchenko. Circuit-based intrinsic methods to detect overfitting. *arXiv preprint arXiv:1907.01991*, 2019.

[54] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(12):2894–2903, 2006.

[55] A. Chattopadhyay, L. Amarú, M. Soeken, P.-E. Gaillardon, and G. De Micheli. Notes on majority boolean algebra. In *Int'l Symp. on Multiple-Valued Logic*, pages 50–55, 2016.

[56] Y. Chen and C. Wang. Fast detection of node mergers using logic implications. In *Int'l Conf. on Computer-Aided Design*, pages 785–788, 2009.

[57] F. T. Chong, D. Franklin, and M. Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, 2017.

[58] Z. Chu, L. Shi, L. Wang, and Y. Xia. Multi-objective algebraic rewriting in XOR-majority graphs. *Integration*, 69:40–49, 2019.

[59] C. Chung, Y. Chen, C. Wang, and C. Wu. Majority logic circuits optimisation by node merging. In *Asia and South Pacific Design Automation Conference*, pages 714–719, 2017.

[60] S. Cimato, V. Ciriani, E. Damiani, and M. Ehsanpour. An OBDD-based technique for the efficient synthesis of garbled circuits. In *International Workshop on Security and Trust Management*, pages 158–167, 2019.

[61] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *Int'l Conf. on Tools with Artificial Intelligence*, pages 186–193, 2014.

[62] J. Cong and Y. Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(1):1–12, 1994.

[63] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 29–35, 1999.

[64] N. Courtois, D. Hulme, and T. Mourouzis. Solving circuit optimisation problems in cryptography and cryptanalysis. *IACR Cryptology ePrint Archive*, 2011:475, 2011.

[65] G. Csaba, A. Imre, G. H. Bernstein, W. Porod, and V. Metlushko. Nanocomputing by field-coupled nanomagnets. *IEEE Trans. on Nanotechnology*, 99(4):209, 2002.

[66] M. Damiani, J.-Y. Yang, and G. De Micheli. Optimization of combinational logic circuits based on compatible gates. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 14(11):1316–1327, 1995.

[67] F. David Bryan, Brglez and R. Lisanke. Redundancy identification and removal. *IWLS*, 1991.

[68] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[69] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[70] D. Dor and U. Zwick. Selecting the median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999.

[71] R. Drechsler, N. Drechsler, and W. Günther. Fast exact minimization of BDDs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(3):384–389, 2000.

[72] S. Dutta, O. Zografos, S. Gurunarayanan, I. Radu, B. Soree, et al. Proposal for nanoscale cascaded plasmonic majority gates for non-Boolean computation. *Scientific reports*, 7(1):17866, 2017.

[73] C. R. Edwards. The application of the Rademacher-Walsh transform to Boolean function classification and threshold logic synthesis. *IEEE Trans. on Computers*, 24(1):48–62, 1975.

[74] N. Éen. Practical SAT - a tutorial on applied satisfiability solving, 2007. slides of invited talk at FMCAD.

[75] K. Fazel, M. A. Thornton, and J. E. Rice. ESOP-based Toffoli gate cascade generation. In *Pacific Rim Conference on Communications, Computers and Signal Processing*, 2007.

[76] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the conference on European design automation*, pages 50–54, 1991.

[77] I. Giacomelli, J. Madsen, and C. Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *Security Symposium*, pages 1069–1083, 2016.

[78] M. Goldmann, J. Håstad, and A. Razborov. Majority gates vs. general weighted threshold gates. *Computational Complexity*, 2(4):277–300, 1992.

[79] E. Goto and H. Takahasi. Some theorems useful in threshold logic for enumerating Boolean functions. In *IFIP Congress*, pages 747–752, 1962.

[80] D. Goudarzi and M. Rivain. On the multiplicative complexity of Boolean functions and bitsliced higher-order masking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 457–478, 2016.

[81] W. Haaswijk, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. A novel basis for logic optimization. In *Asia and South Pacific Design Automation Conference*, pages 151–156, 2017.

[82] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli. SAT-based exact synthesis: Encodings, topology families, and parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019.

[83] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli. Classifying functions with exact synthesis. In *Int'l Symp. on Multiple-Valued Logic*, pages 272–277, 2017.

[84] G. D. Hachtel and F. Somenzi. *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2006.

[85] I. Háleček, P. Fišer, and J. Schmidt. Are XORs in logic synthesis really necessary? In *Design and Diagnostics of Electronic Circuits & Systems, International Symposium on*, pages 134–139, 2017.

[86] I. Hanninen and J. Takala. Pipelined array multiplier based on quantum-dot cellular automata. In *European Conf. on Circuit Theory and Design*, pages 938–941, 2007.

[87] L. Hellerman. A catalog of three-variable Or-invert and And-invert logical circuits. *IEEE Trans. Electronic Computers*, 12(3):198–223, 1963.

[88] W. Hesse, E. Allender, and D. A. M. Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695 – 716, 2002.

[89] T. Hofmeister. The power of negative thinking in constructing threshold circuits for addition. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 20–26, 1992.

[90] T. Hofmeister, W. Hohberg, and S. Köhling. Some notes on threshold circuits, and multiplication in depth 4. *Inf. Process. Lett.*, 39(4):219–225, 1991.

[91] K. Ikegami, H. Noguchi, C. Kamata, M. Amano, K. Abe, , et al. Low power and high density STT-MRAM for embedded cache memory using advanced perpendicular MTJ integrations and asymmetric compensation techniques. In *IEEE Int'l Electron Devices Meeting*, pages 28–1, 2014.

[92] S. Jukna. *Boolean Function Complexity*. Springer, 2012.

[93] V. Kabanets and J. Cai. Circuit minimization problem. *Symposium on Theory and Computing*, pages 73 –79, 2000.

[94] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[95] A. Khitun and K. L. Wang. Non-volatile magnonic logic circuits engineering. *Journal of Applied Physics*, 110(034306), 2011.

[96] D. E. Knuth. *The Art of Computer Programming, Volume 3, Second Edition*. Addison-Wesley, 1998.

[97] D. E. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[98] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.

[99] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev. Finding efficient circuits using SAT-solvers. In *Int'l Conf. on Theory and Applications of Satisfiability Testing*, pages 32–44, 2009.

[100] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498, 2008.

[101] K. Kong, Y. Shang, and R. Lu. An optimized majority logic synthesis methodology for quantum-dot cellular automata. *IEEE Trans. on Nanotechnology*, 9(2):170–183, 2010.

[102] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[103] A. S. Kulikov and V. V. Podolskii. Computing majority by constant depth majority circuits with low fan-in gates. In *Symposium on Theoretical Aspects of Computer Science*, pages 49:1–49:14, 2017.

[104] R. J. Lechner. Harmonic analysis of switching functions. In *Recent Developments in Switching Theory*, pages 121–228. Academic Press, 1971.

[105] C. S. Lent and P. D. Tougaw. A device architecture for computing with quantum dots. *Proceedings of the IEEE*, 85(4):541–557, 1997.

[106] C. S. Lent, P. D. Tougaw, W. Porod, and G. H. Bernstein. Quantum cellular automata. *Nanotechnology*, 4(1):49–57, 1993.

[107] K. K. Likharev and V. K. Semenov. RSFQ logic/memory family: A new Josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Transactions on Applied Superconductivity*, 1(1):3–28, 1991.

[108] G. Liu and Z. Zhang. A parallelized iterative improvement approach to area optimization for LUT-based technology mapping. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 147–156, 2017.

[109] L. Machado and J. Cortadella. Support-reducing decomposition for FPGA mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, pages 1–1, 2018.

[110] L. Machado and J. Cortadella. Support-reducing functional decomposition for FPGA technology mapping. *Int'l Workshop on Logic and Synthesis*, 2018.

[111] D. S. Marakallage, E. Testa, H. Riener, M. Soeken, and G. De Micheli. Semester project on 3-input gates for logic synthesis, 2020.

[112] E. J. McCluskey. Minimization of Boolean functions. *Bell Labs Technical Journal*, 35(6):1417–1444, 1956.

[113] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. D. Micheli. The role of multiplicative complexity in compiling low T-count oracle circuits. In *Int'l Conf. on Computer-Aided Design*, 2019.

[114] D. M. Miller and M. Soeken. A spectral algorithm for ternary function classification. In *Int'l Symp. on Multiple-Valued Logic*, pages 198–203, 2018.

[115] H. S. Miller and R. O. Winder. Majority-logic synthesis by geometric methods. *IRE Trans. Electronic Computers*, 11(1):89–90, 1962.

[116] M. Miller and M. Soeken. An algorithm for linear, affine and spectral classification of Boolean functions. *International Workshop on Boolean Problems*, pages 237–254, 2018.

[117] A. Mishchenko, R. Brayton, A. Petkovska, and M. Soeken. SAT-based optimization with dont-cares revisited. *Int'l Workshop on Logic and Synthesis*, 2017.

[118] A. Mishchenko and R. K. Brayton. SAT-based complete don't-care computation for network optimization. In *Design, Automation and Test in Europe*, pages 412–417, 2005.

[119] A. Mishchenko and R. K. Brayton. Scalable logic synthesis using a simple circuit structure. In *Int'l Workshop on Logic and Synthesis*, pages 15–22, 2006.

[120] A. Mishchenko, R. K. Brayton, J. R. Jiang, and S. Jang. Scalable don't-care-based logic optimization and resynthesis. *ACM Trans. on Reconfigurable Technology and Systems*, 4(4):34:1–34:23, 2011.

[121] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Design Automation Conference*, pages 532–535, 2006.

[122] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):240–253, 2007.

[123] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton. Combinational and sequential mapping with priority cuts. In *Int'l Conf. on Computer-Aided Design*, pages 354–361, 2007.

[124] A. Mishchenko and M. A. Perkowski. Logic synthesis of reversible wave cascades. In *Int'l Workshop on Logic and Synthesis*, 2002.

[125] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Design Automation Conference*, pages 103–108, 2001.

[126] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. K. Brayton, and M. Chrzanowska-Jeske. Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):743–755, 2006.

[127] V. K. Mishra and H. Thapliyal. Heuristic based majority/minority logic synthesis for emerging technologies. In *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, pages 295–300, 2017.

[128] Y. Miyasaka, A. Mishchenko, and M. Fujita. A simple BDD package without variable reordering and its application to logic optimization with permissible functions. In *Int'l Workshop on Logic and Synthesis*, 2019.

[129] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[130] S. Muroga. Logic synthesizers, the transduction method and its extension, sylon. In *Logic Synthesis and Optimization*, pages 59–86. Springer, 1993.

[131] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method-design of logic networks based on permissible functions. *IEEE Trans. on Computers*, (10):1404–1424, 1989.

[132] C. D. Murray and R. R. Williams. On the (non) NP-hardness of computing circuit complexity. *Conference on Computational Complexity*, pages 365–380, 2015.

[133] K. Navi, R. Farazkish, S. Sayedsalehi, and M. R. Azghadi. A new quantum-dot cellular automata full-adder. *Microelectronics Journal*, 41(12):820–826, 2010.

[134] D. E. Nikonov, G. I. Bourianoff, and T. Ghani. Nanomagnetic circuits with spin torque majority gates. In *IEEE-NANO*, pages 1384–1388, 2011.

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

[136] D. E. Nikonov, S. Manipatruni, and I. A. Young. Automotion of domain walls for spintronic interconnects. *Journal of Applied Physics*, 115(21):213902, 1996.

[137] D. E. Nikonov and I. A. Young. Overview of beyond-CMOS devices and a uniform methodology for their benchmarking. *Proceedings of the IEEE*, 101(12):2498–2533, 2013.

[138] P. Pan and C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 35–42, 1998.

[139] W. J. Paul. A 2.5n-lower bound on the combinational complexity of Boolean functions. *SIAM Journal on Computing*, 6(3):427–443, 1977.

[140] E. L. Post. *The Two-Valued Iterative Systems of Mathematical Logic*, volume 5. Princeton University Press, 2016.

[141] B. T. Preas, M. J. Lorenzetti, and B. D. Ackland. *Physical design automation of VLSI systems*. Benjamin-Cummings Pub Co, 1988.

[142] R. Puri, A. Bjorksten, and T. E. Rosser. Logic optimization by output phase assignment in dynamic logic synthesis. In *Int'l Conf. on Computer-Aided Design*, pages 2–8, 1996.

[143] P. Raghavan, M. G. Bardon, D. Jang, P. Schuddinck, D. Yakimets, et al. Holisitic device exploration for 7nm node. In *IEEE Custom Integrated Circuits Conf.*, pages 1–5, 2015.

[144] S. Ray, A. Mishchenko, N. Eén, R. K. Brayton, S. Jang, and C. Chen. Mapping into LUT structures. In *Design, Automation and Test in Europe*, pages 1579–1584, 2012.

[145] S. M. Reddy. Complete test sets for logic functions. *IEEE Trans. on Computers*, C-22(11):1016–1020, 1973.

[146] J. H. Reif and S. R. Tate. On threshold circuits and polynomial computation. *SIAM J. Comput.*, 21(5):896–908, 1992.

[147] G. V. Resta, Y. Balaji, D. Lin, I. P. Radu, F. Catthoor, et al. Doping-free complementary logic gates enabled by two-dimensional polarity-controllable transistors. *ACS Nano*, 12(7):7039–7047, 2018.

[148] G. V. Resta, A. Leonhardt, Y. Balaji, S. De Gendt, P. Gaillardon, and G. De Micheli. Devices and circuits using novel 2-D materials: A perspective for future VLSI systems. *IEEE Trans. VLSI Syst.*, 27(7):1486–1503, 2019.

[149] M. S. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar. MPCircuits: Optimized circuit generation for secure multi-party computation. *Int'l Symp. on Hardware-Oriented Security and Trust*, pages 198–207, 2019.

[150] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken. On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis. In *Design, Automation and Test in Europe*, pages 1649–1654, 2019.

[151] H. Riener, E. Testa, L. G. Amarù, M. Soeken, and G. De Micheli. Size optimization of MIGs with an application to QCA and STMG technologies. In *Int'l Symp. on Nanoscale Architectures*, pages 157–162, 2018.

[152] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarù, G. De Micheli, and M. Soeken. Scalable generic logic synthesis: One approach to rule them all. In *Design Automation Conference*, 2019.

[153] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amaru, et al. Logic optimization of majority-inverter graphs. In *Workshop-Methods and Description Languages for Modelling and Verification of Circuits and Systems*, pages 1–4, 2019.

[154] R. Rudell and A. Sangiovanni-Vincentelli. *Logic synthesis for VLSI design.* PhD thesis, University of California, Berkeley, 1989.

[155] T. N. Sasamal, A. K. Singh, and A. Mohan. An optimal design of full adder based on 5-input majority gate in coplanar quantum-dot cellular automata. *Int.'l Journal for Light and Electron Optics*, 127(20):8576–8591, 2016.

[156] T. Sasao. *Switching Theory for Logic Synthesis.* Springer, 1999.

[157] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita. Boolean resubstitution with permissible functions and binary decision diagrams. In *Design Automation Conference*, pages 284–289, 1991.

[158] C. Schensted, 1978. A letter to Martin Gartner from December 9, 1978.

[159] S. Sheikhfaal, S. Angizi, S. Sarmadi, M. H. Moaiyeri, and S. Sayedsalehi. Designing efficient QCA logical circuits with power dissipation analysis. *Microelectronics Journal*, 46(6):462–471, 2015.

[160] M. Sholander. Medians and betweenness. *Proceedings of the American Mathematical Society*, 5:801–807, 1954.

[161] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Optimizing majority-inverter graphs with functional hashing. In *Design, Automation and Test in Europe*, pages 1030–1035, 2016.

[162] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Exact synthesis of majority-inverter graphs and its applications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(11):1842–1855, 2017.

[163] M. Soeken, G. De Micheli, and A. Mishchenko. Busy man's synthesis: Combinational delay optimization with SAT. In *Design, Automation and Test in Europe*, pages 830–835, 2017.

[164] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. G. Amarù, et al. Practical exact synthesis. In *Design, Automation and Test in Europe*, pages 309–314, 2018.

[165] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, et al. The EPFL logic synthesis libraries, Nov. 2019. arXiv:1805.05121v2.

[166] M. Soeken, E. Testa, A. Mishchenko, and G. De Micheli. Pairs of majority-decomposing functions. *Information Processing Letters*, 139:35–38, 2018.

[167] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *Symposium on Security and Privacy*, pages 411–428, 2015.

[168] B. Steinbach and C. Posthoff. Compact XOR bi-decomposition for generalized lattices of Boolean functions. *Reed-Muller Workshop*, 2017.

[169] N. Takeuchi, D. Ozawa, Y. Yamanashi, and N. Yoshikawa. An adiabatic quantum flux parametron as an ultra-low-power logic device. *Superconductor Science and Technology*, 26(3):035010, 2013.

[170] E. Testa, L. Amarù, M. Soeken, A. Mishchenko, P. Vuillod, P.-E. Gaillardon, and G. D. Micheli. Extending Boolean methods for scalable logic synthesis. *Submitted to IEEE Trans. on CAD of Integrated Circuits and Systems*, 2020.

[171] E. Testa, L. Amarù, M. Soeken, A. Mishchenko, P. Vuillod, J. Luo, C. Casares, P.-E. Gaillardon, and G. De Micheli. Scalable boolean methods in a modern synthesis flow. In *Design, Automation and Test in Europe*, pages 1643–1648, 2019.

[172] E. Testa, W. J. Haaswijk, M. Soeken, and G. De Micheli. The complexity of self-dual monotone 7-input functions. In *Int'l Workshop on Logic and Synthesis*, 2019.

[173] E. Testa, S. L. Noor, O. Zografos, M. Soeken, F. Catthoor, et al. Multiplier architectures: Challenges and opportunities with plasmonic-based logic. In *Design, Automation and Test in Europe*, 2020.

[174] E. Testa, M. Soeken, L. Amarù, and G. De Micheli. Logic synthesis for established and emerging computing. *Proceedings of the IEEE*, 107(1):165–184, 2018.

[175] E. Testa, M. Soeken, L. Amarù, and G. De Micheli. Reducing the multiplicative complexity in logic networks for cryptography and security applications. In *Design Automation Conference*, 2019.

[176] E. Testa, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Inversion minimization in majority-inverter graphs. In *Int'l Workshop on Logic and Synthesis*, 2016.

[177] E. Testa, M. Soeken, L. G. Amarù, W. Haaswijk, and G. De Micheli. Mapping monotone Boolean functions into majority. *IEEE Trans. on Computers*, 68(5):791–797, 2018.

[178] E. Testa, M. Soeken, H. Riener, L. Amarù, and G. De Micheli. A logic synthesis toolbox for reducing the multiplicative complexity in logic networks. In *Design, Automation and Test in Europe*, 2020.

[179] E. Testa, M. Soeken, O. Zografos, L. G. Amarù, P. Raghavan, et al. Inversion optimization in majority-inverter graphs. In *Int'l Symp. on Nanoscale Architectures*, pages 15–20, 2016.

[180] E. Testa, M. Soeken, O. Zografos, F. Catthoor, and G. De Micheli. Exact synthesis for logic synthesis applications with complex constraints. In *Int'l Workshop on Logic and Synthesis*, 2017.

[181] E. Testa, O. Zografos, M. Soeken, A. Vaysset, M. Manfrini, et al. Inverter propagation and fan-out constraints for beyond-CMOS majority-based technologies. In *Annual Symp. on VLSI*, pages 164–169, 2017.

[182] P. D. Tougaw and C. S. Lent. Logical devices implemented using quantum cellular automata. *Journal of Applied Physics*, 75(3):1818–1825, 1993.

[183] M. Turan Sönmez and R. Peralta. The multiplicative complexity of Boolean functions on four and five variables. In *Lightweight Cryptography for Security and Privacy*, pages 21–33, Cham, 2015.

[184] A. Vaysset, M. Manfrini, D. E. Nikonov, S. Manipatruni, I. A. Young, et al. Toward error-free scaled spin torque majority gates. *AIP Advances*, 6(6):065304, 2016.

[185] A. Vaysset, M. Manfrini, D. E. Nikonov, S. Manipatruni, I. A. Young, et al. Operating conditions and stability of spin torque majority gates: Analytical understanding and numerical evidence. *Journal of Applied Physics*, 121(4):043902, 2017.

[186] K. Walus, T. J. Dysart, G. A. Jullien, and R. A. Budiman. QCADesigner: A rapid design and simulation tool for quantum-dot cellular automata. *IEEE Trans. on Nanotechnology*, 3(1):26–31, 2004.

[187] P. Wang, M. Y. Niamat, S. R. Vemuru, M. Alam, and T. Killian. Synthesis of majority/minority logic networks. *IEEE Trans. on Nanotechnology*, 14(3):473–483, 2015.

[188] I. Wegener. *The complexity of Boolean functions*. John Wiley, 1987.

[189] R. L. Wigington. A new concept in computing. *Proceedings of the IRE*, 47(4):516–523, 1959.

[190] C. Wolf. Yosys Open SYnthesis Suite. *http://www.clifford.at/yosys/*.

[191] C. Yang and M. Ciesielski. BDS: A BDD-based logic optimization system. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(7):866–876, 2002.

[192] S. Yang. Logic synthesis and optimization Version 3.0, 1991.

[193] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Int'l Symp. on Field Programmable Gate Arrays*, pages 161–170, 2015.

[194] R. Zhang, P. Gupta, and N. K. Jha. Majority and minority network synthesis with application to QCA-, SET-, and TPL-Based nanotechnologies. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(7):1233–1245, 2007.

[195] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff. Limits to binary logic switch scaling - a gedanken model. *Proceedings of the IEEE*, 91(11):1934–1939, 2003.

[196] O. Zografos, F. Catthoor, S. Dutta, and A. Naeemi. *US Patent Application US20190064438A1*, 2019.

[197] O. Zografos, A. De Meester, E. Testa, M. Soeken, P.-E. Gaillardon, and *et al.* Wave pipelining for majority-based beyond-CMOS technologies. *Design, Automation and Test in Europe*, 2017.

[198] O. Zografos, B. Sorée, A. Vaysset, S. Cosemans, L. G. Amarù, et al. Design and benchmarking of hybrid CMOS-spin wave device circuits compared to 10nm CMOS. In *Int'l Conference on Nanotechnology*, pages 686–689, 2015.

Eleonora Testa

PhD Student at EPFL

Rue de Crissier 9 B
1020 Renens, Switzerland
☎ *(+41) 77 9851412*
✉ *eleonora.testa@epfl.ch*

Research Interests: Logic synthesis, EDA, majority-based logic synthesis, logic synthesis for nanotechnologies, emerging nanotechnologies, new data structures for Boolean functions optimization, algorithms, threshold logic.

Education

Sept. 2016 - to date	**PhD in Computer and Communication Sciences**, *Institute: École Polytechnique Fédérale de Lausanne (CH)*, Thesis advisor: Prof. Giovanni De Micheli, Thesis co-advisor: Dr. Mathias Soeken. Thesis title: Data Structures and Algorithms for Logic Synthesis in Advanced Technologies
2013 - 2015	**Master of Science in Nanotechnologies for ICTs**, *Institute: École Polytechnique Fédérale de Lausanne (CH), Grenoble INP (FR), Politecnico di Torino (IT)*, Final Mark: 110 con Lode/110 (**full marks with honors**).
2010 - 2013	**Bachelor Degree in Physics Engineering**, *Institute: Politecnico di Torino (IT)*, Final Mark: 109/110.

Research Experience

June - Aug. 2018	**Internship**, *Synopsys Inc. (USA)*, Supervisor: Dr. Luca Amarù.
Sept. 2015 - Aug. 2016	**Research Assistant**, *École Polytechnique Fédérale de Lausanne (CH)*, Supervisor: Prof. Giovanni De Micheli.
June - Aug. 2016	**Visiting Researcher**, *IMEC (BE)*, Supervisors: Dr. Julien Ryckaert and Prof. Francky Catthoor.
June - Aug. 2014	**Internship**, *Istituto Italiano di Tecnologia (IT)*, Supervisors: Dr. Giancarlo Canavese and Prof. Fabrizio Pirri.

Publications

Conference & Workshop Papers:

- L. Amarù, F. Marranghello, **E. Testa**, C. Casares, V. Possani, *et al.*:"SAT-Sweeping Enhanced for Logic Synthesis". DAC, 2020.
- **E. Testa**, S. L. Noor, O. Zografos, M. Soeken, F. Catthoor, *et al.*: "Multiplier Architectures: Challenges and Opportunities with Plasmonic-based Logic (special session)". DATE, 2020.
- **E. Testa**, M. Soeken, H. Riener, L. Amarù, G. De Micheli: "A Logic Synthesis Toolbox for Reducing the Multiplicative Complexity in Logic Networks". DATE, 2020.
- H. Riener, M. Soeken, **E. Testa**, G. De Micheli: "Generic Logic Synthesis Meets RTL Synthesis". WOSET, 2019.
- M. Soeken, **E. Testa**, M. Miller: "A Hybrid Method for Spectral Translation Equivalent Boolean Functions". PACRIM, 2019.
- **E. Testa**, W. Haaswijk, M. Soeken, G. De Micheli: "The Complexity of Self-Dual Monotone 7-Input Functions". IWLS, 2019.
- M. Soeken, **E. Testa**, M. Miller: "A Hybrid Spectral Method for Checking Boolean Function Equivalence". RM, 2019.
- H. Riener, **E. Testa**, W. Haaswijk, A. Mishchenko, L. Amarù, *et al.*: "Logic Optimization of Majority-Inverter Graphs". MBMV, 2019.
- H. Riener, **E. Testa**, W. Haaswijk, A. Mishchenko, L. Amarù, *et al.*: "Scalable Generic Logic Synthesis: One Approach to Rule Them All". DAC, 2019.

- **E. Testa**, M. Soeken, L. Amarù, G. De Micheli: “Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications”. DAC, 2019.
- **E. Testa**, L. Amarù, M. Soeken, A. Mishchenko, P. Vuillod, *et al.*: “Scalable Boolean Methods in a Modern Synthesis Flow”. DATE, 2019.
- M. Soeken, H. Riener, W. Haaswijk, **E. Testa**, G. De Micheli: “The EPFL Logic Synthesis Libraries". WOSET, 2018.
- L. Amarù, **E. Testa**, M. Couceiro, O. Zografos, G. De Micheli, *et al.*: “Majority Logic Synthesis (embedded tutorial)". ICCAD, 2018.
- H. Riener, **E. Testa**, L. Amarù, M. Soeken, G. De Micheli: “Size Optimization of MIGs with an Application to QCA and STMG Technologies". NANOARCH, 2018.
- M. Soeken, W. J. Haaswijk, **E. Testa**, A. Mishchenko, L. Amarù, *et al.*:“Practical Exact Synthesis (special session)". DATE, 2018.
- **E. Testa**, O. Zografos, M. Soeken, A. Vaysset, M. Manfrini, *et al.*: “Inverter Propagation and Fan-out Constraints for Beyond-CMOS Majority-based Technologies". ISVLSI, 2017. **Best Poster Award.**
- **E. Testa**, M. Soeken, O. Zografos, F. Catthoor, G. De Micheli: “Exact Synthesis for Logic Synthesis Applications with Complex Constraints". IWLS, 2017.
- W. Haaswijk, **E. Testa**, M. Soeken, and G. De Micheli: “Classifying Functions with Exact Synthesis". ISMVL, 2017.
- O. Zografos, A. De Meester, **E. Testa**, M. Soeken, P.-E. Gaillardon, *et al.*: “Wave Pipelining for Majority-based Beyond-CMOS Technologies (special session)". DATE, 2017.
- L. Amarù, M. Soeken, W. Haaswijk, **E. Testa**, P. Vuillod, *et al.*: “Multi-level Logic Benchmarks: An Exactness Study". ASPDAC, 2017.
- **E. Testa**, M. Soeken, O. Zografos, L. Amarù, P. Raghavan, *et al.*: “Inversion Optimization in Majority-Inverter Graphs". NANOARCH, 2016.
- **E. Testa**, M. Soeken, L. Amarù, P.-E. Gaillardon, G. De Micheli: “Inversion Minimization in Majority-Inverter Graphs". IWLS, 2016.

Journal Papers:

- H. Riener, G. Meuli, **E. Testa**, M. Soeken, V. N. Kravets, *et al.*: “Open-Source EDA Benchmarks”. Submitted to IEEE Design & Test, 2020.
- **E. Testa**, L. Amarù, M. Soeken, A. Mishchenko, P. Vuillod, *et al.*: “Extending Boolean Methods for Scalable Logic Synthesis”. Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.
- **E. Testa**, M. Soeken, L. Amarù, W. Haaswijk, G. De Micheli: “Mapping Monotone Boolean Functions into Majority”. IEEE Transactions on Computers, 2018.
- **E. Testa**, M. Soeken, L. Amarù, G. De Micheli: “Logic Synthesis for Established and Emerging Computing". Proceedings of the IEEE, 2018.
- M. Soeken, **E. Testa**, A. Mishchenko, G. De Micheli: “Pairs of Majority-decomposing Functions". Information Processing Letters, 2018.

Preprints:

- M. Soeken, H. Riener, W. Haaswijk, **E. Testa**, B. Schmitt, G. Meuli, F. Mozafari, G. De Micheli: “The EPFL Logic Synthesis Libraries”. arXiv:1805.05121v2, subject: cs.LO, cs.MS, November, 2019.

Invited and Conference Talks

Invited Talks:

- “Decomposing n-ary Majority Functions”: ICCAD Embedded Tutorial, 2018.
- “Logic Synthesis for Post-CMOS Technologies”: Dagstuhl Reports, Vol. 7, Issue 2 ISSN 2192-5283, 2017

Conference Talks:

- “The Complexity of Self-Dual Monotone 7-Input Functions”. IWLS, 2019.
- “Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications”. DAC, 2019.
- “Scalable Boolean Methods in a Modern Synthesis Flow”. DATE, 2019.

- “The EPFL Logic Synthesis Libraries". WOSET, 2018.
- “Inverter Propagation and Fan-out Constraints for Beyond-CMOS Majority-based Technologies". ISVLSI, 2017.
- “Exact Synthesis for Logic Synthesis Applications with Complex Constraints". IWLS, 2017.
- “Inversion Optimization in Majority-Inverter Graphs". NANOARCH, 2016.
- “Inversion Minimization in Majority-Inverter Graphs". IWLS, 2016.

Awards and Honors

- O-1 U.S. work visa - *individuals with an extraordinary ability in sciences*.
- Best poster award at the IEEE Computer Society Annual Symposium on VLSI (ISVLSI) received on the 3rd July 2017 for the presentation of the paper “Inverter Propagation and Fan-out Constraints for Beyond-CMOS Majority-based Technologies”.
- EPFL, I&C School PhD Fellowship, 2016.

Teaching Assistantships

- Design Technologies for Integrated Systems, M.Sc. course, Fall 2019, EPFL.
- Analog Circuits for Biochips, M.Sc. course, Spring 2019, EPFL.
- Design Technologies for Integrated Systems, M.Sc. course, Fall 2018, EPFL.
- Analog Circuits for Biochips, M.Sc. course, Spring 2018, EPFL.
- Design Technologies for Integrated Systems, M.Sc. course, Fall 2017, EPFL.
- Mathematical Analysis II, B.Sc. course, Spring 2017, EPFL.

Patents

- Patent Application with Synopsys Inc. on “Scalable Boolean Methods in a Modern Synthesis Flow".

Professional Service

- Reviewer for the journal ELSEVIER Integration, the VLSI Journal.
- Reviewer for the journal ELSEVIER Microprocessors and Microsystems.
- Reviewer for the journal IEEE Transactions on Very Large Scale Integration Systems.
- Reviewer for the journal IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
- Reviewer for the Design, Automation & Test in Europe Conference (DATE), 2020.
- Reviewer for the Design Automation Conference (DAC), 2019.
- Reviewer for the IEEE International Symposium on Multiple-Valued Logic (ISMVL), 2018-2019.
- Reviewer for the Workshop on Design Automation for Understanding Hardware Designs (DUHDe), 2017.
- Member of the IEEE.

Competences

Software Skills: LaTeX, MATLAB; COMSOL; LabView; Cadence Virtuoso, Encounter; Synopsys Design Compiler, Formality; Mentor Modelsim; Altera Quartus.

Technical Skills: Cleanroom experience (6 months): Optical lithography; Dry - Wet etching of Si, metals and high-ks; Microscopy; SEM; FIB; Electrical characterization.

Programming Languages: C, C++, Python.

Languages

- Italian, first language.
- English, fluent (IELTS 7.5).
- French, A2/B1.

Ce document a été imprimé au Centre d’impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.