Thèse n° 10 876

Technology Mapping and Optimization Algorithms for Logic Synthesis of Advanced Technologies

Présentée le 4 octobre 2024

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

Alessandro TEMPIA CALVINO

Acceptée sur proposition du jury

Prof. V. Kuncak, président du jury
Prof. G. De Micheli, directeur de thèse
Prof. M. Ciesielski, rapporteur
Prof. J. Cortadella, rapporteur
Prof. P. Ienne, rapporteur

École
polytechnique
fédérale
de Lausanne

2024

To my family and to all those who have taught me

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Giovanni De Micheli, for warmly welcoming me into his laboratory of brilliant researchers. His invaluable guidance, expertise, motivation, and unwavering support were crucial to the realization of this thesis.

Besides my advisor, I would like to extend my sincere thanks to Dr. Alan Mishchenko for the fruitful collaboration and the many insightful discussions we shared over the past two years, both on research and life. His enthusiasm and creative mind, constantly generating a wealth of research ideas, were instrumental in shaping multiple parts of this thesis.

I am also deeply thankful to Dr. Luca Amarù and Dr. Patrick Vuillod, who, during my internship at Synopsys, guided me toward pursuing a PhD with Prof. Giovanni De Micheli and introduced me to research in the field of logic synthesis. Without their encouragement and influence, I might not have embarked on this journey.

Furthermore, I am grateful to Dr. Heinz Riener for his guidance at the beginning of the PhD.

I extend my special thanks to Dr. Xiaoqing Xu and the entire Bodger team at Google for giving me the opportunity to intern and collaborate on groundbreaking research projects.

I would like to deeply thank all the members of my oral committee - Prof. Maciej Ciesielski, Prof. Jordi Cortadella, Prof. Paolo Ienne, and Prof. Viktor Kunčak - for their time, valuable feedback, and insightful questions while reviewing this thesis.

I also wish to thank Prof. Jason Cong, Prof. Wayne Burleson, Prof. Robert Wille, and Dr. Victor Kravets for their support.

Additionally, I would like to acknowledge Prof. Ludovic Apvrille and Prof. Renaud Pacalet for their guidance and introduction to research during my master's studies.

My sincere thanks also go to my colleagues at the Integrated Systems Laboratory (LSI), in particular to Andrea, Sonia, Mingfei, Giulia, and Rassul, for our fruitful collaborations. I am also grateful to the rest of my labmates for creating a supportive research environment, as well as to the visitors and students who enriched the LSI during their time with us. A special thanks goes to Chantal Demont for her kind help and support throughout my PhD journey.

I would like to thank all my friends in Lausanne, especially Marija, Marco, Andrea, and Martina, for their support and friendship, as well as my friends in Italy, the United States, and France, and those who have moved abroad but remained close. Though not everyone is mentioned by name, please accept my heartfelt appreciation to each of you whom I have had the pleasure of

meeting over these past four years, both in my academic and personal life.
Last but not least, I would like to thank my mother and my grandparents for supporting me throughout my PhD studies and my life in general.

Lausanne, 16 Sep 2024 A. T.C.

Abstract

Logic synthesis is a key component of *electronic design automation* (EDA) tools, essential for designing high-performance, compact, and power-efficient integrated circuits. The continuous downscaling of *complementary metal-oxide-semiconductor* (CMOS) technology has led to remarkable improvements in performance, power efficiency, and density of integrated circuits. However, as CMOS technology faces challenges in further downscaling transistor dimensions, with only marginal improvements at smaller nodes, logic optimization becomes even more vital for enhancing *power-performance-area* (PPA) metrics. Additionally, many potential alternative technologies to CMOS are emerging, offering significant advantages in power efficiency and performance. Nevertheless, existing EDA tools for CMOS are often not well-suited for these new technologies, necessitating the development of specialized synthesis techniques.

This thesis focuses on developing state-of-the-art logic synthesis methods for advanced technologies. These technologies include conventional CMOS for *field-programmable gate arrays* (FPGAs) and *standard-cell-based designs*, as well as *superconducting electronics* (SCE). In particular, we concentrate on the technology mapping problem, which involves translating a technology-independent circuit description into an interconnection of gates specific to a technology library.

Novel contributions are organized into four parts. First, we improve performance-driven technology mapping for FPGAs. We introduce powerful and runtime-efficient algorithms to decompose functions into *lookup tables* (LUTs). Then, we develop a LUT mapper that utilizes this decomposition to minimize delay. Our results show substantial advancements over existing methods, including some of the best public results for combinational benchmark circuits. We also propose a LUT mapper that exploits non-routable connections in FPGAs to minimize the circuit delay. Second, we enhance technology mapping for standard-cell-based design. We develop algorithms that address the *matching* and *covering* problems, fully leveraging standard cells in modern libraries, including large-input and multiple-output cells. We demonstrate significant improvements compared to the state of the art. Third, motivated by multiple logic representations available in logic synthesis, we propose a technique to translate circuits between different logic representations and perform circuit optimization. Then, we show how to leverage efficiently *don't care* conditions in logic rewriting. Our methods contribute to obtaining the best-known results in *majority-inverter graphs* (MIGs) size. Next, we present practical algorithms for factored form literal optimization in modern logic synthesis based on *and-inverter graphs* (AIGs), demonstrating applications in standard-cell-based design and transistor-level synthesis. Fourth and last, we research synthesis solutions for the two most mature logic families in SCE: the *adiabatic quantum-flux parametron* (AQFP) and the *single-flux quantum* (SFQ). For AQFP circuits, we demonstrate that depth-optimal technology mapping is a tractable problem and propose scalable algorithms for mapping and post-mapping area reduction. Finally, we introduce a synthesis framework for SFQ circuits. We show strong results in both logic families.

Considering the increasing difficulties in meeting the design objectives of modern ICs, we argue that innovative research in EDA solutions is of extreme importance.

Key words: Electronic design automation, logic synthesis, technology mapping, FPGA, ASIC, emerging technologies, superconducting electronics

Sommario

La *sintesi logica* è un componente chiave degli strumenti di *electronic design automation* (EDA), essenziale per la progettazione di chip prestanti, compatti ed a basso consumo energetico. La continua miniaturizzazione della tecnologia CMOS ha portato a notevoli miglioramenti nelle prestazioni, nell'efficienza energetica e nella densità dei circuiti integrati. Tuttavia, poiché la tecnologia CMOS fatica nel ridurre le dimensioni dei transistor, con miglioramenti marginali ai nodi più recenti, l'ottimizzazione logica diventa cruciale per migliorare la metrica *power-performance-area* (PPA). Inoltre, stanno emergendo numerose tecnologie alternative a CMOS, con vantaggi in termini di efficienza energetica e prestazioni. Tuttavia, gli strumenti EDA esistenti per CMOS spesso non sono adatti a queste nuove tecnologie, rendendo necessario lo sviluppo di tecniche di sintesi specializzate.

Questa tesi si concentra sullo sviluppo di nuovi metodi di sintesi logica per tecnologie avanzate. Queste tecnologie includono CMOS per *field-programmable gate array* (FPGA) e *standard-cell-based designs*, nonché *superconducting electronics* (SCE). In particolare, ci concentriamo sul problema della mappatura tecnologica, che implica la traduzione di una descrizione di circuito indipendente dalla tecnologia in un'interconnessione di porte logiche specifiche.

La tesi è organizzata in quattro parti. In primo luogo, miglioriamo la mappatura tecnologica per FPGA. Proponiamo algoritmi efficienti per decomporre funzioni in *lookup tables* (LUTs). Sviluppiamo poi un mapper per LUTs che utilizza questa decomposizione per migliorare le prestazioni. I nostri risultati mostrano progressi sostanziali rispetto ai metodi esistenti e alcuni dei migliori risultati pubblici per circuiti combinatori. Proponiamo anche un mapper per LUTs che sfrutta le *non-routable connections* in FPGA per migliorare ulteriormente le prestazioni. In secondo luogo, miglioriamo la mappatura tecnologica per standard-cell-based designs. Sviluppiamo algoritmi che affrontano i problemi di *matching* e *covering*, sfruttando appieno le librerie di standard cells, comprese le porte con molti ingressi e più uscite. Dimostriamo miglioramenti significativi rispetto allo stato dell'arte. In terzo luogo, motivati dalle molteplici rappresentazioni logiche utilizzate in sintesi logica, proponiamo una tecnica per navigare tra diverse rappresentazioni ed ottimizzarle. Poi, mostriamo come sfruttare le condizioni *don't care* nella riscrittura logica. I nostri metodi contribuiscono ad ottenere i migliori risultati conosciuti finora in area per *majority-inverter graph* (MIGs). Presentiamo inoltre algoritmi per l’ottimizzazione di factored form literals sugli *and-inverter graphs* (AIGs). Dimostriamo applicazioni in standard-cell-based design e nella sintesi a livello di transistor. Infine, ci concentriamo sulla sintesi per le due famiglie di SCE più mature: l’*adiabatic quantum-flux parametron* (AQFP) e il *single-flux quantum* (SFQ). Per i circuiti AQFP, dimostriamo che la mappatura tecnologica ottimale per prestazioni è un problema trattabile e proponiamo algoritmi efficienti per eseguire la mappatura e la riduzione dell’area post-mappatura. Introduciamo poi un framework di sintesi per i circuiti SFQ. Mostriamo risultati significativi in entrambe le famiglie logiche.

Considerando le crescenti difficoltà nel soddisfare gli obiettivi di progettazione dei moderni circuiti integrati, sosteniamo che la ricerca in EDA sia di estrema importanza.

Parole chiave: Electronic design automation, sintesi logica, mappatura tecnologica, FPGA, ASIC, nuove nanotecnologie, elettronica a superconduttore

Contents

List of Figures

List of Tables

List of Acronyms

AIG	*and-inverter graph*
ACD	*Ashenhurst-Curtis decomposition*
ALAP	*as late as possible*
AQFP	*adiabatic quantum-flux parametron*
ASAP	*as soon as possible*
ASIC	*application-specific integrated circuit*
BDD	*binary decision diagram*
CAD	*computer-aided design*
CMOS	*complementary metal-oxide-semiconductor*
DAG	*directed acyclic graph*
EDA	*electronic design automation*
FFL	*factored form literal*
FFLC	*factored form literal count*
FPGA	*field-programmable gate array*
IC	*integrated circuits*
JJ	*Josephson junction*
LUT	*lookup table*
MFFC	*maximum fan-out-free cone*
MIG	*majority-inverter graph*
MSPF	*maximum set of permissible functions*
NLDM	*non-linear delay model*
PI	*primary input*
PO	*primary output*
PPA	*power-performance-area*

QCA	*quantum-dot cellular automata*
QoR	*quality of results*
RSFQ	*rapid single-flux parametron*
RTL	*register-transfer level*
SAT	*satisfiability*
SFQ	*single-flux parametron*
SOP	*sum-of-products*
TFI	*transitive fan-in*
TFO	*transitive fan-out*
XAG	*xor-and graph*
XAIG	*xor-and-inverter graph*
XMG	*xor-majority graph*

1 Introduction

The modern era of digital electronics is marked by an unprecedented level of complexity in the design and manufacturing of integrated circuits (ICs). Central to managing this complexity is the field of *computer-aided design* (CAD), which provides the tools and methodologies to conceptualize, model, and verify large-scale and high-performance circuits for a wide range of applications, including information processing (e.g., computers, data centers, smartphones), and telecommunication. Computer-aided design for electronics is more known as *electronic design automation* (EDA).

The continuous downscaling of *complementary metal-oxide-semiconductor* (CMOS) has led to remarkable improvements in the performance, power efficiency, and density of integrated circuits. However, this relentless miniaturization has also introduced significant challenges in EDA as integrated circuits became more complex. Furthermore, the recent slowdown in transistor scaling, which has traditionally followed Moore's Law, is shifting the main driving force of power-performance-area (PPA) improvements from physical scaling to EDA, requiring more sophisticated designs and optimization techniques.

Logic synthesis is an essential phase of EDA tools aimed at improving the implementation cost of integrated circuits in terms of PPA. This process is critical for transforming a high-level description of a circuit into an optimized implementable gate-level representation suitable for fabrication. The process of logic synthesis typically begins with an abstraction of the circuit, described in terms of generic logic gates or Boolean functions. The initial objective is to optimize the logic using several technology-independent techniques to reduce its complexity. Then, the circuit is transformed into a netlist of gates that can be implemented using a specific semiconductor technology. Here, more accurate and technology-dependent estimations of delay and area replace the initial, simpler estimations. After technology mapping, further optimizations can be performed using technology-dependent techniques.

Research in logic synthesis is critically important for several reasons. First, faster and more power-efficient circuits are required for high-performance computing, data centers, artificial intelligence, and mobile and consumer applications. As CMOS technology faces numerous challenges in further downscaling the transistor dimensions, with only marginal improvements at smaller technology nodes, logic optimization becomes crucial for improving PPA. Additionally, the exceedingly high costs associated with manufacturing at the latest technology nodes, now at 2 nanometers, mean that even a 1% reduction in area can translate into significant cost savings in large-scale chip production. Second, many alternative technologies are emerging as future potential alternatives to CMOS. Various designs realized in post-silicon technologies have demonstrated advantages in power efficiency and performance [7, 14, 77, 166]. However, the scope of these emerging nanotechnologies remains limited compared to their potential. Due to differing technological constraints and physical properties, existing EDA tools for CMOS are often not well-suited for these new technologies, necessitating the development of specialized synthesis techniques. Furthermore, alternative computing paradigms, such as quantum computing, and applications in cryptography and security, have shown significant benefits from advancements in logic synthesis. Third, progress in related fields, such as Boolean satisfiability and machine learning, enables the development of novel synthesis methodologies to achieve higher PPA.

This thesis concentrates on developing novel logic synthesis techniques to improve the quality of modern ICs. The first part proposes improved technology mapping algorithms for field-programmable gate arrays and standard-cell designs. The second part focuses on technology-independent logic synthesis. The third part proposes synthesis methods to realize efficient circuits in superconducting technology.

1.1 Electronic Design Automation

EDA is a category of software tools, algorithms, and methods used to automatically design, analyze, verify, and optimize electronic systems, particularly integrated circuits. The evolution of EDA tools is closely linked with the history of semiconductor technology. In the early days of semiconductor design, engineers manually designed circuit layouts and performed simulations by hand or with simple software tools. As ICs grew in complexity and the industry transitioned from small-scale integration (SSI) to very large-scale integration (VLSI), manual design methods became impractical. The advent of CAD techniques laid the groundwork for EDA, with early tools providing basic schematic capture and layout capabilities. Research in academic institutions and companies founded in the mid-70s and 80s drove innovation in logic synthesis, physical design, and verification. As semiconductor technology advanced, EDA tools evolved to address the challenges of deep submicron and nanometer-scale designs, incorporating features to manage PPA and manufacturability constraints.

The realization of ICs is structured as a *flow* of well-defined steps. An EDA flow starts from a high-level behavioral description of ICs and generates a detailed, manufacturable implementation using specific technology components. In this section, we provide a largely simplified overview of the main steps involved in an EDA flow, illustrated in Figure 1.1. The flow starts from a behavioral description of the IC in the form of a hardware description

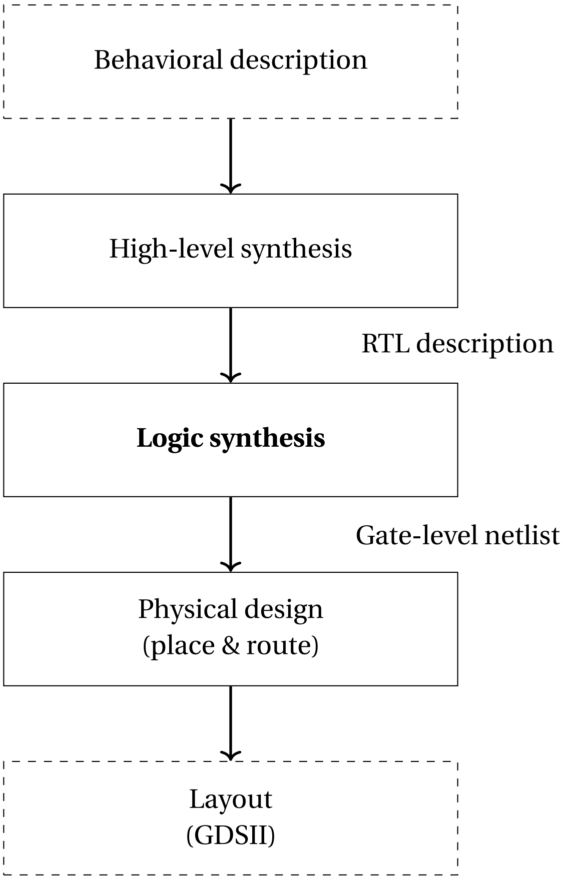


Figure 1.1: Simplified EDA flow.

language (e.g., Verilog, VHDL) or a programming language (e.g., SystemC, C/C++). High-level synthesis converts the behavioral description into a *register-transfer level* (RTL) description, which is an abstraction that models digital circuits using digital signals (data), registers, and logic operations. This step is also named *architectural-level synthesis* as it determines the macroscopic (block-level) structures of the circuit. *Logic synthesis*, instead, generates a gate-level (or logic-level) model, consisting of an interconnection of logic primitives. In logic synthesis, *technology mapping* transforms the logic model into an interconnection of library cells reflecting the target technology. A simplified flow of the steps in logic synthesis is depicted in Figure 1.2. Finally, there is *physical design*, primarily consisting of *placement* and *routing* (P&R), where placement assigns a physical position to the cells while routing generates the interconnections. Physical design translates the gate-level model into geometrical patterns defining the physical layout of the chip. The final layout is typically described in the GDSII format, which is handed over to IC foundries for fabrication.

Figure 1.1 is oversimplified in the following aspects. First, there are no clear boundaries separating the various phases in an EDA flow nowadays. For instance, logic synthesis is also performed during physical design to further optimize logic under a more accurate model. Moreover, some parts of physical design, such as *floorplanning*, are performed at the very

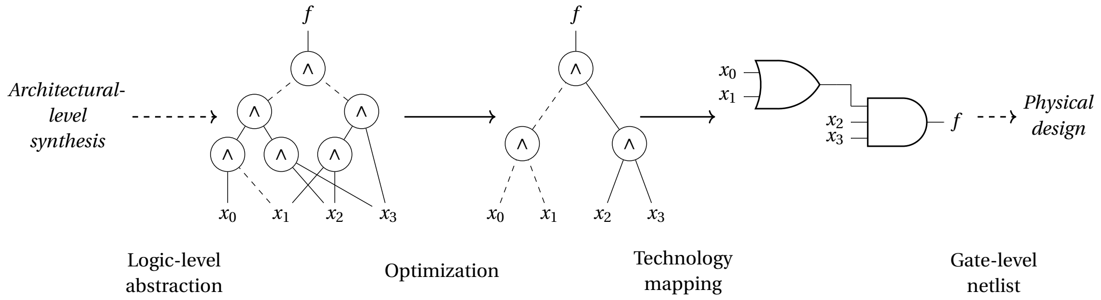


Figure 1.2: Simplified logic synthesis flow. Initially, the circuit is abstracted as a logic network of simple primitives, such as 2-input ANDs, with symbol ∧, and inverters, represented with dashed edges. Then, optimization steps reduce the complexity of the circuit. Finally, technology mapping translates the circuit into a mapped gate-level netlist using library cells (here an OR2 cell and an AND3 cell).

beginning of the flow. Second, the flow is not a straight line but consists of several cycles. Third, an EDA flow includes multiple steps of formal verification and simulation.

During logic synthesis, and in general during the whole design flow, *circuit optimization* is performed to maximize the circuit quality, for instance, in terms of performance, area, and power consumption, and to meet the target constraints. Without optimization, modern ICs would not be competitive or even feasible due to their complexity. In this thesis, we mainly concentrate on *area* and *delay* (performance) improvements.

As shown in Figure 1.1, EDA flows represent ICs using several levels of abstractions that grow in complexity after each step. Abstraction is necessary to simplify the optimization problems to a tractable level and synthesize circuits in a reasonable run time. Furthermore, without abstraction, modern circuit design would not be feasible. As a consequence, target metrics such as power, performance, and area have to be estimated during the flow since they are accurately known only after physical design. Abstraction led to the creation of different estimation models, cost metrics, and data structures to represent and manipulate circuits.

In logic synthesis for CMOS design, circuits are abstracted as logic networks defined over a set of logic primitives. Originally, logic synthesis utilized NANDs and NORs, together with inverters, as primitives in graph representations thanks to their universality. At this time, many approaches addressed two-level synthesis problems with Boolean functions represented as *sum-of-products* (SOPs). Optimization was focused on reducing the number of *implicants* and *literals*. Then, circuits were modeled as multi-level logic networks, where nodes are represented in SOP form. The number of literals in the factored forms of SOPs became the standard metric for area in technology-independent synthesis, leading to the development of many optimization methods for this metric [28]. As logic synthesis evolved and integrated circuits became larger and more complex, scalability emerged as a crucial issue. This led to the adoption of *and-inverter graphs* (AIGs) [75, 97], consisting of 2-input AND gates and inverters, and *and-or-inverter graphs* (AOIGs), consisting of 2-input AND gates, 2-input OR gates, and inverters, as the most common technology-independent representations. The gate count and levels of logic became the most common metrics for area and performance, respectively. These abstractions focus on technology-independent logic synthesis prior to technology mapping. After technology mapping, circuits are represented as logic networks of library cells, providing a more accurate and technology-dependent area and delay estimations. Technology mapping also addresses optimization problems. It translates a technology-independent circuit representation into an interconnection of library cells corresponding to a technology while minimizing cost metrics, such as area under delay constraints.

This thesis primarily addresses the technology mapping problem, with a secondary focus on logic synthesis. Throughout this thesis, we present advanced technology mapping algorithms for *field-programmable gate arrays* (FPGAs), standard-cell-based designs (including general-purpose processors and *application-specific integrated circuits*), and *superconducting electronics* (SCE). Since these technologies have distinct libraries of gates and constraints, competitive technology mapping algorithms necessitate specialized approaches. Additionally, we investigate technology-independent logic optimization with methods inspired by technology mapping and physical implementations in various technologies.

1.2 Research Motivation

Over the past few decades, integrated circuits have undergone remarkable advancements in complexity and performance, a trend required to continue to drive future innovations in many applications, such as high-performance computing, artificial intelligence, and low-power devices, and in various fields, such as weather forecasting and computational chemistry. However, as CMOS technology faces numerous challenges in further downscaling the transistor dimensions, now at 2 nanometers, EDA tools become increasingly more important for sustaining this progress. Additionally, extremely high manufacturing costs at the latest technology nodes significantly motivate EDA solutions for area minimization, which has often been overshadowed by the focus on performance. Furthermore, emerging technologies are demonstrating advantages in power efficiency and performance, positioning themselves as potential future alternatives to CMOS. However, the scope of these emerging nanotechnologies remains limited. Due to differing technological constraints and physical properties, existing EDA tools for CMOS are often not well-suited for these new technologies, necessitating the development of specialized synthesis techniques. Nowadays, the development of novel synthesis techniques is highly important.

This thesis investigates advanced logic synthesis algorithms to address these challenges. We first discuss research motivations related to conventional CMOS technologies in Section 1.2.1. Subsequently, we continue with EDA applications for superconducting electronics as a potential post-silicon technology in Section 1.2.2.

1.2.1 Conventional CMOS Technologies

CMOS is the most advanced technology for integrated circuits. Despite the physical scaling limitations and the gradual slowdown of Moore's Law, CMOS-based chips maintain a significant lead over emerging technologies in terms of scalability, cost-effectiveness, and maturity of the manufacturing process. Moreover, ongoing research and development efforts continue to push the boundaries of CMOS technology to its limit. However, numerous challenges in further downscaling the transistor dimensions require increased efforts in researching novel EDA techniques to achieve better *quality of results* (QoR).

Modern microelectronic circuits typically comply with two design styles based on the target implementation, namely *standard-cell-based design* and *field-programmable gate arrays* (FPGA). In this thesis, we address both.

Standard-cell-based design

Standard-cell design is a semi-custom design methodology based on *standard cells* for dense and efficient integrating circuits, including general-purpose processors and application-specific integrated circuits (ASICs), such as GPUs. Standard cells are pre-designed, pre-characterized blocks of transistors configured to perform specific logic functions and serve as the fundamental building blocks for creating complex integrated circuits. Utilizing standard cells offers multiple key advantages. They increase the level of abstraction in the design flow, eliminating the need to design complex circuits at the transistor level, which enables tools to manage billions of transistors and synthesize circuits in a reasonable time. Moreover, the design flow becomes more versatile and capable of operating across various fabrication technologies and design rules. Standard cells also facilitate efficient design synthesis, placement, and routing by allowing automated tools to leverage pre-characterized data to optimize and compute power, performance, and area.

Traditional logic synthesis for standard-cell-based design was mainly based on NAND-NOR (AND-OR) primitives because they closely abstract simple gates realizable in CMOS technology. In the last decade, other data structures incorporating XORs, multiplexers, and majorities have demonstrated remarkable results in leveraging new optimization opportunities [5, 9, 145]. This led to the development of novel algebraic and Boolean techniques. The ongoing research for logic synthesis methods continues to uncover new opportunities, but many remain unexplored.

Technology mapping for standard-cell-based designs is the phase in which a synthesized logic network is transformed into an interconnection of standard cells. This process is formulated as an optimization problem, with optimum-cost technology mapping classified as an NP-hard problem. Furthermore, mapping includes many sub-problems, such as *matching*, *covering*, *gate sizing*, *buffering*, etc. In Chapter 4, our focus is primarily on the sub-problems of matching and covering, which are essential for achieving efficient and effective technology mapping. Generally, technology mapping is crucial for good QoR. Consequently, it is important to research and develop algorithms to further improve area, delay, power consumption, and mapping runtime.

In addition to the technology mapping problem itself, the physical downscaling limitations of transistors have led to the evolution of standard cell libraries to be more comprehensive. Consequently, technology mapping needs to evolve to better leverage these advanced libraries.

Field-programmable gate arrays

Field-Programmable Gate Arrays (FPGAs) are integrated circuits with *configurable logic blocks* (CLBs) and programmable interconnect. Unlike standard-cell-based designs, which follow a semi-custom design methodology and have a fixed configuration, FPGAs can be programmed many times after manufacturing. This flexibility comes at the cost of lower power-performance-area (PPA) efficiency. Nevertheless, FPGAs offer sufficient speed for various applications, including rapid prototyping, low to medium-volume products, custom hardware accelerators, telecommunications equipment, and applications requiring frequent updates or feature enhancements. Furthermore, FPGAs offer lower initial production costs due to the absence of *non-recurring engineering* (NRE) expenses, even if their per-unit costs remain relatively high regardless of volume.

The EDA flow for FPGA designs consists of modeling hardware designs to make them run efficiently on FPGA architectures. EDA design flows for FPGAs share similarities with those for standard-cell-based design, but they focus on using configurable logic blocks to implement logic operations and programmable switch boxes and routing channels for interconnects. EDA for FPGA design researches new solutions to improve performance and scalability in two directions: (i) by improving the FPGA architecture; and (ii) by enhancing the tools. In the former case, architectural improvements in FPGAs aim to reduce the implementation cost, in PPA metric, of generic or specific classes of designs. For instance, modern architectures include higher-level functionality fixed in silicon, such as *digital signal processor* (DSP) blocks to accelerate signal processing applications. Furthermore, updates to the configurable logic blocks (CLBs) and their interconnections can significantly enhance performance. A recent innovation by FPGA vendors involves supplementing programmable interconnects with fast, non-routable connections between LUTs to reduce the interconnect delay. In the latter case, tools must continually evolve to support new FPGA architectures and fully exploit their capabilities. This involves not only keeping pace with architectural innovations but also rethinking existing problems with modern techniques to unlock significant improvements.

In the logic synthesis phase, the main logic primitive implemented in CLBs is the k-input *lookup table* (LUT), which functions like a memory and can realize any k-input Boolean function. Typically, FPGAs feature LUTs with 6 or 4 inputs. The technology mapping problem involves the transformation of a synthesized logic network into an interconnection of LUTs. As for technology mapping for standard cells, this process is formulated as an optimization problem, with optimum-cost technology mapping being classified as an NP-hard problem.

In Chapter 3, we address the technology mapping problem for designs meant to run on FPGAs (*LUT mapping*). We propose efficient Boolean decomposition techniques integrated into a structural mapper to enhance circuit performance. Additionally, we introduce a method to leverage non-routable connections between LUTs in recent FPGAs, further optimizing for performance.

1.2.2 Superconducting Electronics

The increasing interest in *superconducting electronics* (SCE) is related to the search for a computing technology that could match or surpass the current performance of CMOS while achieving lower energy consumption. Among the various emerging nanotechnologies introduced over the past few decades, SCE has demonstrated a considerable level of maturity, with several examples of medium-size processors already realized [2, 11, 14, 148]. SCE works at temperatures near absolute zero (typically at 4K), where resistive effects can be neglected, and uses *Josephson junctions* (JJs) as switching elements. SCE systems can achieve up to 100 times lower operating power, including refrigeration power, and 1-100 times higher clock frequencies than CMOS [78, 89]. More complex circuits have demonstrated clock frequencies between 1 to 10 times those of conventional CMOS [11, 14, 148]. The two main logic families of SCE are the *single-flux quantum* (SFQ) [119] and the *adiabatic quantum-flux parametron* (AQFP) [191].

Despite successful applications, the scope of SCE applications remains narrow compared to its potential. Conventional EDA tools for CMOS are not well-suited for SCE due to fundamental differences between the two technologies. One key difference is the unique nature of representing zeros and ones. SFQ uses the presence or absence of a voltage pulse, while AQFP uses the current direction. To distinguish logic zero from logic one, the evaluation at each gate is triggered by a clock signal. Consequently, data at a gate's inputs must be available in specific timeframes for the computation to be correct. This often requires to use delaying registers on certain circuit paths. In literature, this problem is referred to as *path balancing*. Another major difference is the poor driving capacity of SCE gates, due to the small currents involved. This limitation requires the addition of special gates called *splitters* to the logic, which distribute signals to multiple destinations without degrading the signal integrity. In literature, this problem is referred to as *fan-out branching*. The number of delaying registers required for path balancing and splitters can be prohibitively large, often contributing to 50% of the total area and energy consumption [13, 35, 88, 148]. This significantly degrades the efficiency and yield of superconducting integrated systems.

These complications necessitate the development of EDA tools tailored for SCE. Current EDA tools for superconducting electronics can handle only limited-scale designs with less than a million cells and often require manual intervention. Additionally, optimization methods for SCE often come from a straightforward adaptation of CMOS techniques, which are not fully suited to the unique requirements of SCE. Significant progress can be made in design optimization. Logic synthesis for SCE demands modern abstractions and optimization methods. For example, AQFP logic is majority-based, while SFQ logic efficiently implements XORs. Furthermore, the technology mapping problem for these technologies is particularly challenging, as it must address path balancing and fan-out branching requirements.

1.3 Thesis Contributions

This thesis focuses on logic synthesis and, more specifically, on technology mapping algorithms for conventional CMOS technologies and superconducting electronics. In this section, our contributions are classified according to the underlying technology. First, we present our contributions to technology mapping for CMOS, targeting the two main types of semiconductor devices, namely *field-programmable gate arrays* (FPGAs) and standard-cell-based designs. Then, we introduce our contributions to technology-independent logic synthesis. Finally, we present our contributions in logic synthesis and technology mapping for *superconducting electronics* (SCE). For ease of reading, our achievements are presented in the same order they appear in the chapters of the thesis.

1.3.1 Technology Mapping

We develop novel technology mapping techniques to improve the quality of CMOS-based designs. The contributions are classified based on the target technology. We first present our contributions for *field-programmable gate array* (FPGA)-based designs to later transition to *standard cell*-based designs.

Field-programmable gate array design flow

We study novel technology mapping algorithms to enhance the performance of designs meant to run on FPGAs. First, we propose techniques to efficiently decompose Boolean functions into a flexible number of k-input LUTs. Second, we propose an advanced performance-driven technology mapping algorithm that integrates Boolean decomposition on the fly. Finally, we present a technology mapper that leverages non-routable LUT connections available in FPGAs to optimize for performance.

State-of-the-art technology mapping into LUTs is performed through local substitutions applied to an initial graph representation derived by technology-independent logic synthesis. The drawback of this approach is that the technology-independent optimization step and the technology mapping step are separated. Consequently, the impact of optimization on the quality of the final LUT network is hard to predict before mapping. Specifically, the structure of the subject graph highly influences the mapping quality. This is known as *structural bias.* To mitigate structural bias, the known methods compute structural choices for the subject graph and use them during mapping [37, 114], or collapse and decompose parts of the graph during mapping [41, 60, 112]. However, exact area and delay optimization during LUT mapping remain NP-hard [46, 124].

Our contributions focus on developing advanced techniques to mitigate the structural bias during LUT mapping. We first study the *Ashenhurst-Curtis decomposition* (ACD) [12, 49], also known as *Roth-Karp decomposition* [170], which is the most generic Boolean decomposition formulation. Current state-of-the-art ACD techniques suffer from slow run times or limited performance due to constraints aimed at reducing the complexity. We address these limitations by proposing a revised formulation of ACD for LUT mappers and post-mapping resynthesis engines that perform delay optimization. We provide algorithms to minimize the decomposition cost in terms of the number of LUTs, edges, and delay, considering input arrival times. Then, we present a delay-driven LUT mapping algorithm that integrates ACD on the fly to reduce the structural bias. To our knowledge, this is the first practical and scalable work using ACD for delay-driven LUT mapping. We demonstrate remarkable improvements in performance compared to state-of-the-art LUT mapping. Additionally, we focus on methods to leverage non-routable connections between LUTs, forming LUT structures, in modern FPGA [10]. These connections reduce the need for signal routing through multiple switch boxes and routing channels. However, placement algorithms struggle to utilize these connections effectively for delay optimization. We address this problem during technology mapping by using our ACD formulation to extract decompositions into LUT structures. We show significant improvements compared to previous approaches.

Standard cell design flow

We study methods to improve technology mapping for standard-cell-based design. We propose novel *matching* techniques to increase the support of large and multiple-output cells during mapping. Then, we present *covering* algorithms to achieve better QoR and support multiple-output cells.

Standard cells define a set of pre-designed and pre-characterized logic primitives that are used as building blocks to create digital circuits. The problem of optimally mapping Boolean functions to standard cells is known to be intractable. Therefore, technology mapping is generally formulated as a series of local substitutions applied to a simple multi-level graph representation obtained from technology-independent logic synthesis. Given the complexity of the technology mapping problem, numerous heuristics and solutions have been proposed in the literature to address delay and area optimization (possibly under delay constraints) [37, 38, 65, 79, 84, 87, 90, 98, 114, 123, 146, 186]. All these works focus on technology mapping for single-output cells. Mapping mainly addresses two sub-problems: *matching* and *covering*. Matching involves associating sections of the subject graph with a list of cells that are functionally equivalent and capable of implementing those sections. Covering selects appropriate cells to cover the graph, such that the target cost function is minimized.

Our contribution introduces novel techniques to enhance the quality of results achievable by technology mappers, with a primary focus on area optimization. We begin by investigating the matching problem during technology mapping. Current state-of-the-art techniques often face a trade-off between matching quality and support for large cells. To overcome this limitation, we propose a fast matching approach that combines pattern matching and Boolean matching, yielding significant improvements in both area and runtime compared to existing methods. Next, we focus on supporting multiple-output library cells during technology mapping. We present innovative techniques for detecting and matching multiple-output cells, and we introduce the first technology mapping algorithm that fully integrates support for multiple-output cells. This advancement enables more efficient utilization of available standard cells and further optimizes area. We also revisit heuristic algorithms for covering, aiming to achieve better area optimization under delay constraints. Overall, our contributions offer significant advancements in technology mapping, particularly in terms of area optimization, by addressing key limitations in matching quality, multiple-output cell support, and covering algorithms.

1.3.2 Mapping for Logic Synthesis

We investigate methods to improve technology-independent logic synthesis. We propose algorithms influenced by technology mapping to optimize and convert logic in various representations. Then, we develop methods to efficiently leverage don't care conditions during logic rewriting in various representations. Finally, we revisit AIG-based logic optimization for factored form literals, with applications in standard cell design flows and transistor-level synthesis.

Modern state-of-the-art logic synthesis tools leverage multiple homogeneous logic networks for the representation, manipulation, and optimization of logic. Homogeneous logic networks are directed acyclic graphs that use restrictions on the type of Boolean functions and fan-in of logic blocks. The *and-inverter graph* (AIG) [75, 97], consisting of 2-input AND gates and inverters, is the most common representation. The *majority-inverter graph* (MIG) [5, 6], consisting of 3-input majority gates and inverters, has shown to further unlock optimization opportunities in arithmetic-intensive designs. Furthermore, many emerging nanotechnologies are majority-based and benefit from majority-based logic optimization (e.g., AQFP [191], QDCA [116], and spin-wave devices [91]). Other examples of useful homogeneous logic networks include the *xor-and graphs* (XAGs) [72] and *xor-majority graphs* (XMGs) [68], for their compactness and efficiency in arithmetic-intensive designs. XAGs are also the natural abstraction for emerging nanotechnologies, and cryptography or security applications.

Our contribution focuses primarily on optimization for homogeneous logic networks. Since multiple graph representations are available to support logic synthesis for conventional and emerging nanotechnologies, we first propose a versatile technique called *graph mapping* to convert a homogeneous logic network into another while performing global Boolean optimization. When the destination representation matches the starting one, graph mapping performs logic rewriting. This approach facilitates the use of multiple logic representations in logic synthesis and their conversion. Furthermore, it provides a valuable optimization technique for less mature graph representations, such as MIGs and XMG. This method has been demonstrated to be one of the most effective logic optimization approaches over MIGs, XAGs, and XMGs. Then, we study how to leverage Boolean *don't care* conditions in logic rewriting. We provide scalable Boolean matching algorithms to achieve local size-optimum results for incompletely specified Boolean functions extracted during rewriting and graph mapping. Finally, we focus on *factored form literal count* (FFLC) optimization, a traditional cost function used to drive area optimization in networks of *sum-of-products* (SOPs). FFLC is a critical metric as it correlates strongly with the number of transistors required in CMOS implementation. We present a comprehensive set of algebraic and Boolean methods to optimize for FFLC directly on the AIG. The applications of FFLC optimization are twofold: it can enhance a standard-cell design flow, and it is applicable in transistor-level synthesis and auto-creation of custom standard cells, as FFLC closely approximates transistor count. This latter project is the result of an internship at Google LLC (X division), which led to the publication [200] and patents [203, 213].

1.3.3 Synthesis for Superconducting Electronics

We research logic synthesis techniques tailored for the two main logic families of superconducting electronics (SCE), namely the *adiabatic quantum-flux parametron* (AQFP) and the *single-flux quantum* (SFQ). First, we consider the technology mapping problem for AQFP involving the path-balancing and fan-out-branching requirements. We propose a depth-optimal technology mapping algorithm and a post-mapping area-oriented optimization method. Second, we describe the synthesis problem for SFQ over the primitives AND and XOR and propose an automatic logic synthesis toolbox.

Due to the path-balancing and fan-out-branching requirements, technology mapping for SCE needs to insert delay registers and splitters. The number of delaying registers required for path balancing and the number of splitters can be prohibitively large, often contributing to 50% of the total area and energy consumption [13, 35, 88, 148]. Existing work considered reducing imbalances and high-fan-outs during logic optimization as a proxy for reducing the path-balancing and fan-out-branching costs [33, 126, 158, 204]. Other previous work developed techniques to insert and minimize delay registers and splitters after logic synthesis [35, 80, 88, 107]. A key distinction characterizes the path-balancing and fan-out-branching requirement between AQFP and SFQ technologies. In AQFP, splitters are clocked and considered in path balancing, so path balancing and fan-out branching have to be addressed together. The interplay between buffers and splitters makes the optimization of delay registers and splitters for AQFP a challenging problem. Conversely, in SFQ, splitters are not clocked, thus the two constraints can be considered separately, thereby simplifying the technology mapping problem.

Our contributions focus on minimizing the area and power overhead derived from path-balancing and fan-out-branching requirements. First, we propose technology mapping algorithms to satisfy the technological constraints of AQFP circuits, addressing the buffer (delay register for AQFP) and splitter (B/S) insertion problem. We prove that the depth-optimal B/S insertion problem is tractable with polynomial complexity, and we provide two depth-optimal algorithms based on the *as-late-as-possible* (ALAP) and *as-soon-as-possible* (ASAP) strategies. Following this, we present a post-mapping area-oriented B/S optimization algorithm based on minimum-register retiming. Finally, we develop a logic synthesis framework for SFQ. We focus on delay optimization, which is key to synthesizing efficient SFQ circuits. Optimization is carried out on an XAG representation. We present multiple algebraic and Boolean techniques, and we demonstrate how to efficiently perform technology mapping for SFQ.

1.4 Thesis Organization

The goal of this thesis is to develop and enhance technology mapping algorithms for multiple technologies and, more broadly, for logic synthesis. Each technology presents unique constraints and requires specialized techniques, which are addressed in different chapters of this thesis. This thesis is organized as follows:

- **Chapter 2 - Background:** This chapter introduces the necessary background needed to understand the thesis. It provides an overview of state-of-the-art data structures and algorithms used in logic synthesis, along with a discussion of various algebraic, Boolean, and exact optimization methods. Additionally, it introduces the benchmark suites employed throughout the thesis for the experimental evaluation of the proposed algorithms.

- **Chapter 3 - Technology Mapping for FPGAs:** This chapter presents algorithms to improve performance-driven technology mapping for FPGAs. Specifically, this chapter proposes: (i) practical algorithms for generic Boolean decomposition of functions into LUTs; (ii) an advanced delay-driven technology mapping algorithm that integrates Boolean decomposition to reduce the structural bias; (iii) a technology mapper that leverages non-routable connections in FPGAs to reduce the worst-case delay. We demonstrate that our Boolean decomposition method outperforms the state-of-the-art techniques in generality, decomposition success, quality, and run time. We show that our LUT mapping algorithm with Boolean decomposition achieves a significant 12.39% average depth reduction and 2.20% average area reduction compared to the state of the art. Additionally, we demonstrate remarkable results in performance-driven technology mapping leveraging non-routable LUT connections.

- **Chapter 4 - Technology Mapping for Standard Cells:** This chapter focuses on methods to improve the quality of technology mapping for standard cells. Specifically, this chapter proposes: (i) a novel technique to perform high-quality and scalable matching;

(ii) algorithms for technology mapping using multiple-output cells; (iii) advanced technology mapping covering algorithms; and (iv) a technology mapper that integrates the techniques discussed in the chapter. We present the first technology mapper that can support multiple-output cells. We demonstrate remarkable results, especially in area reduction, before and after buffering and gate sizing.

- **Chapter 5 - Mapping for Logic Synthesis:** This chapter studies how approaches similar to technology mapping and innovations in logic rewriting can enhance technology-independent logic synthesis. Specifically, it presents: (i) a versatile mapping approach for graph mapping and logic rewriting of technology-independent graph representations; (ii) algorithms to efficiently leverage don't care conditions in graph mapping and logic rewriting; (iii) novel methods to optimize the factored form literal count in large multi-level Boolean networks, with applications in standard-cell design flows and transistor-level synthesis. For graph mapping and logic rewriting, we developed a versatile technique that can map from and to various logic network representations. Additionally, we present a global logic rewriting technique based on mapping with significant results in optimization over xor-and graphs (XAGs), majority-inverter graphs (MIGs), and xor-majority graphs (XMGs). The latter topic investigates AIG-based optimization of the number of factored form literals, traditionally used as a cost function for synthesis due to its correlation with the number of transistors needed to implement a circuit in CMOS technology. We show remarkable results in standard-cell design flows and discuss applications in transistor-level synthesis and standard-cell design.

- **Chapter 6 - Specializing Synthesis for Superconducting Technologies:** This chapter focuses on logic synthesis and technology mapping techniques specialized for superconducting electronics. We introduce the two most mature logic families, namely the single-flux quantum (SFQ) and the adiabatic quantum-flux parametron (AQFP). We discuss the technological constraints and differences compared to CMOS that make the synthesis for these logic families challenging. Specifically, we present: (i) depth-optimal technology mapping algorithms for AQFP circuits; (ii) a post-mapping optimization algorithm for AQFP circuits based on minimum-register retiming; (iii) a logic synthesis and technology mapping framework for SFQ circuits based on the *xor-and graph* (XAG). In the first part, we proposed the first technology mapping algorithm for AQFP with depth optimality guarantees. We demonstrate significant results in AQFP mapping and scalability to designs that are 10 to 100 times larger than those any related work could handle. In the second part, we propose a synthesis flow for SFQ over the xor-and graph (XAG) representation. We show remarkable results in area and delay reduction compared to the state of the art.

- **Chapter 7 - Conclusions:** This chapter concludes the thesis. We summarize the research accomplishments and present the remaining challenging problems related to the topics of this thesis.

2 Background

This thesis focuses on developing algorithms and data structures for the logic synthesis of established and emerging technologies. This chapter serves as the preliminary background for the entire thesis and provides a comprehensive overview of key concepts in logic synthesis. First, we introduce the most common Boolean operations based on Boolean algebra. Then, we discuss data structures to represent Boolean logic. These include *truth tables*, *binary decision diagrams*, two-level representations, and multi-level Boolean networks. Next, we describe methods to extract circuit sub-networks, which are at the core of technology mapping and run-time-intensive algorithms, such as *Boolean resubstitution*, which often requires a *peephole optimization* approach. Then, we review *matching* techniques, which are essential in technology mappers and many logic optimization methods. Subsequently, we present the principal algorithms at the base of logic optimization. This includes a variety of methods, usually classified into algebraic and Boolean methods. Finally, we introduce the benchmark suites used throughout the thesis for the experimental evaluation of the proposed algorithms.

2.1 Boolean Algebra

A *Boolean function* is a mapping from a k-dimensional Boolean space $\mathbb{B}^k = \{0,1\}^k$ into a 1-dimensional one: $\mathbb{B}^k \to \mathbb{B}$. A *multiple-output Boolean function* is mapping from a k-dimensional Boolean space into a m-dimensional one $\mathbb{B}^k \to \mathbb{B}^m$. Multiple-output Boolean function can be seen as an array of m Boolean functions over the same domain.

This definition refers to the *completely specified Boolean function*. An *incompletely specified Boolean function* is defined over a subset of $\mathbb{B}^k$. The points were the function is not defined are called *don't care* conditions. Don't care conditions arise when input combinations never occur (controllability conditions) or when output value do not matter for some input combinations (observability conditions). Don't care conditions may emerge when a Boolean function is a part of a larger system, such a Boolean networks, and are crucial in logic synthesis because they offer additional flexibility when performing Boolean simplification. Typically, incompletely specified Boolean functions are define as a mapping from $\mathbb{B}^k \to \{0,1,*\}$, where $*$ denotes a don't care condition.

The *positive cofactor* of a Boolean function $f(x_0,\ldots,x_i,\ldots,x_{k-1})$ with respect to a variable x_i, represented as f_{x_i}, is the Boolean function obtained by setting $x_i = 1$. Similarly, the *negative cofactor* $f_{\bar{x}_i}$ is the Boolean function obtained by setting $x_i = 0$.

The *Boolean difference* of a Boolean function $f(x_0,\ldots,x_i,\ldots,x_{k-1})$ is $\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i}$. This operator reveals whether f is sensitive to a change in value of input x_i.

The Boole's expansion of a function f, often called Shannon's expansion or Shannon's decomposition, states that:

$$f(x_0,\ldots,x_i,\ldots,x_{k-1}) = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}. \qquad (2.1)$$

Alternatively, the Boole's expansion can be expressed as $f(x_0,\ldots,x_i,\ldots,x_{k-1}) = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$.

The positive Reed-Muller expansion, or positive Davio expansion, states that:

$$f(x_0,\ldots,x_i,\ldots,x_{k-1}) = f_{\bar{x}_i} \oplus x_i \frac{\partial f}{\partial x_i}. \qquad (2.2)$$

Similarly, the negative Reed-Muller expansion, or negative Davio expansion, states that:

$$f(x_0,\ldots,x_i,\ldots,x_{k-1}) = f_{x_i} \oplus \bar{x}_i \frac{\partial f}{\partial x_i}. \qquad (2.3)$$

A completely specified Boolean function f *essentially depends* on a variable x_i if there exists an input combination, such that the value of the function changes when the variable is toggled, i.e., $\frac{\partial f}{\partial x_i} = 1$. The *support* of f is the set of all variables on which function f essentially depends.

In the following, we assume that the reader is familiar with other basic concepts on Boolean algebra and Boolean operations. We refer the reader to [31, 52, 71] for further background.

2.2 Data Structures

Logic representations are key for developing robust EDA tools. They enable compact data storage in memory and efficient implementation of optimization algorithms. The choice of the most suitable data structure depends on the complexity and size of the problem being represented, as well as on the specific operations to be performed on the data. Over the years, numerous data structures have been proposed to enhance the efficiency and effectiveness of logic synthesis. Each data structure offers unique advantages and is tailored for specific types of operations, such as logic simplification, equivalence checking, or data extraction. In this section, we provide an overview of the principal data structures for logic synthesis and their applications.

2.2.1 Truth Tables

A *truth table* representation of a k-input Boolean function $f : \{0,1\}^k \to \{0,1\}$ can be encoded as a bit string $b = b_{2^k-1}b_{2^k-2}\dots b_0$, i.e., a sequence of bits, of length 2^k. A bit $b_i \in \{0,1\}$ at position $0 \le i < 2^k$ is equal to the value taken by f under the input assignment $\vec{x} = (x_0,\dots,x_{k-1})$ where

$$2^{k-1} \cdot x_{k-1} + 2^{k-2} \cdot x_{k-2} + \dots + 2^0 \cdot x_0 = i. \tag{2.4}$$

Example 2.2.1. *The truth table of a* 3*-input AND* $f(x_0, x_1, x_2) = x_0 \wedge x_1 \wedge x_2$ *is* 10000000*. Typically, truth tables are represented using the hexadecimal notation to reduce the size of the representation by a factor of* 4 *(i.e., an hexadecimal digit represents* 4 *bits). For the 3-input AND function, the hexadecimal truth table is* 0x80. ▲

The truth table is a *canonical* representation, meaning that it is unique. As a result, it can efficiently verify the Boolean equivalence of two functions (theoretically in constant time), provided that truth tables can be derived from them.

Commonly, a truth table representation is effective for representing up to 16-input functions. Beyond this value, the exponential growth in size hinders its practical utility. In a 64-bit machine, truth tables for functions up to 6 inputs fit into one machine word. Larger truth tables are typically represented as vectors (or arrays) of 2^{k-6} machine words.

It is common to refer to the leftmost input column of a truth table as the *most significant variable* (x_{k-1}) and the rightmost input column as the *least significant variable* (x_0). A *swap* of two variables alters the truth table by exchanging the location of the corresponding two-variable cofactors.

Example 2.2.2. *The Boolean implication function* $x_1 \to x_0 = \bar{x}_1 \vee x_0$*, represented in binary format as* 1011*, changes to* 1101 *after the variable swap.* ▲

Figure 2.1 depicts two truth tables represented as bit strings, one in binary and one in hexadecimal. Notably, the rightmost truth table can be derived from the leftmost one by swapping variables x_0 and x_2. Marked next to both truth tables are the cofactors with respect to two most significant variables.

A truth table t_1 is said to *imply*, or *cover*, another truth table t_2 if each bit of t_1 is true also in t_2. This relationship is denoted as $t_1 \le t_2$. Similarly, t_2 is said to be *implied* by t_1, denoted as $t_2 \ge t_1$. For instance, $1000 \le 1001$.

2.2.2 Two-level Representations

Boolean functions can be represented by expressions of literals linked by the AND ($\wedge$ or $\cdot$) and OR ($\vee$ or $+$) operations. Note that the AND operator $\cdot$ can be omitted. Generally, the *level* of a representation refers to the number of operators applied to two or more arguments.

x_2	x_1	x_0	f	
0	0	0	1	$f_{\bar{x}_1\bar{x}_2}$
0	0	1	0	
0	1	0	1	$f_{x_1\bar{x}_2}$
0	1	1	0	
1	0	0	1	$f_{\bar{x}_1 x_2}$
1	0	1	1	
1	1	0	0	$f_{x_1 x_2}$
1	1	1	1	

$f = 10110101$

$\xrightarrow{x_0 \leftrightarrow x_2}$

x_0	x_1	x_2	f	
0	0	0	1	$f_{\bar{x}_0\bar{x}_1}$
0	0	1	1	
0	1	0	1	$f_{\bar{x}_0 x_1}$
0	1	1	0	
1	0	0	0	$f_{x_0\bar{x}_1}$
1	0	1	1	
1	1	0	0	$f_{x_0 x_1}$
1	1	1	1	

$f = \text{0xA7}$

Figure 2.1: Truth table representation in the classical tabular form and as bit string (binary and hexadecimal), cofactor extraction w.r.t. the two most significant variables, and variable swapping of x_0 with x_2.

One of the first representations of Boolean logic was the *sum-of-products* (SOP) [28], also referred to as *disjunctive normal form* (DNF). An SOP is a two-level representation consisting of the logic OR (disjunction) of *product terms*, which are logic ANDs (conjunction) of *literals* (variables or their complements). This representation was originally motivated in EDA by *programmable logic arrays* (PLAs) whose primitives are modeled directly using SOPs. Because of the simple structure of a two-level circuit, the optimization problems for SOPs are well understood, which led to the development of efficient heuristic and exact minimization methods, such as *espresso* [171].

Another representation is the *product-of-sums* (POS), also referred to as *conjunctive normal form* (CNF). POSs can be seen as the *dual* of SOPs, and consist of the logic AND (conjunction) of *sum terms*, which are logic ORs (disconjunction) of *literals*. CNFs play a central role in Boolean satisfiability solving [93].

Example 2.2.3. *Let us consider the majority-of-*3 *function. It can be expressed as a sum-of-products $f(a,b,c) = ab + ac + bc$ and as a product-of-sums $f(a,b,c) = (a+b)(a+c)(b+c)$.* ▲

An additional useful two-level representation is the *exclusive sum-of-products* (ESOP), which replaces the OR operator with a XOR operator (⊕). For many Boolean functions, the number of cubes in minimal ESOPs is lower than the number of cubes in minimal SOPs [175]. ESOPs play an important role in applications where the XOR operator is particularly efficient, such as in emerging technologies (see, e.g., [50, 194]), quantum computing (see, e.g., [130]), and cryptography applications (see, e.g., [205, 219]).

2.2.3 Binary Decision Diagrams

The *binary decision diagram* (BDD) [1, 102] is one of the most common logic representation of Boolean functions. The BDD is a directed-acyclic graph based on the *if-then-else* operator. Each node in a BDD is associated with a variable x_i and implements a cofactoring step resulting in a Shannon expansion, i.e., $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$. Hence, each node is connected to two

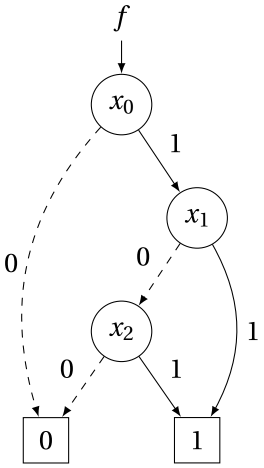


Figure 2.2: BDD (reduced and ordered) for the function $x_0(x_1 + x_2)$.

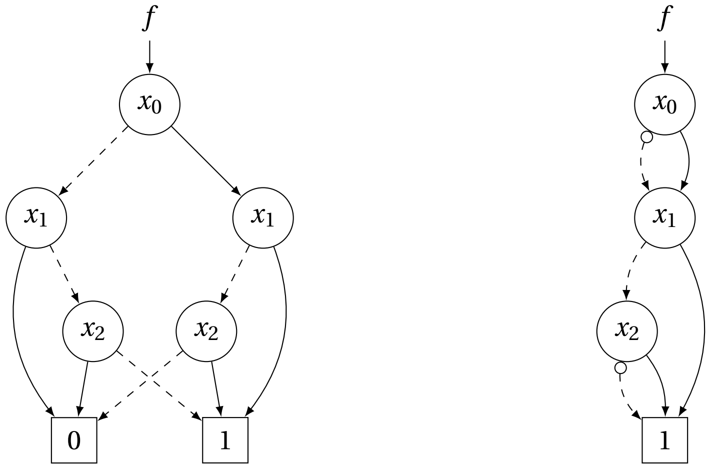
other nodes whose functions represent cofactors of the given function. The root of a BDD is a node representing the given function, and the leaves are constant functions *true* and *false*. The term BDD typically refers to *reduced ordered BDD* (ROBDD) [32], which is a canonical representation for a given variable order and a set of reduction rules.

Example 2.2.4. *Figure 2.2 shows the BDD for the function $x_0(x_1 + x_2)$. Solid and dashed edges, along with the one or zero weights, represent the positive and negative cofactors, respectively.* ▲

BDDs are particularly useful for representing medium-size Boolean functions. Thanks, to reduction rules, the BDD size can be minimized during its construction by merging equivalent cofactors. An important factor controlling the size of a BDD is the variable ordering. For instance, adder functions are sensible to the variable ordering and the corresponding BDD has exponential size in the worst case and linear size in the best case, with respect to the number of variables. Exact [55] and multiple heuristic [63] algorithms have been proposed to minimize the size of a BDD searching for a good ordering. Although, in the worst-case scenario the BDD size is exponential, for many practical functions BDDs remains compact, scaling considerably better than truth tables. For instance, the BDD of the parity function (i.e., multiple-input XOR) grows linearly in size compared to the number of input variables. When the size of the BDD becomes too large, the function is decomposed into logic blocks and represented as a *multi-level Boolean network*.

Nowadays, BDDs are typically implemented using complemented edges to reduce their memory usage. The complementation of an edge takes constant time and may reduce the number of BDD nodes. To preserve the canonicity, BDDs with complemented edges can only complement the zero-cofactor edges and have a single terminal node implementing logic 1.

Example 2.2.5. *Figure 2.3 shows two BDDs for the function $x_0(x_1 + x_2) + \bar{x}_0\bar{x}_1\bar{x}_2$. The one on the left does not use complemented edges. The one on the right uses complemented edges, which are represented by circles on the zero-cofactor edges. This example shows how BDDs with*



(a) Without complemented edges

(b) With complemented edges

Figure 2.3: BDD (reduced and ordered) with and without complemented edges for the function $x_0(x_1 + x_2) + \bar{x}_0\bar{x}_1\bar{x}_2$.

complemented edges may reduce the number of required BDD nodes. ▲

Besides the traditional BDD based on Shannon expansion, multiple other variations have been proposed, such as ZDDs [131] for an efficient manipulation of combinatorial problems, ADDs [15] to symbolically represent Boolean functions whose codomain is a finite set of constants, and BBDDs [4] that use the biconditional expansion instead of the Shannon expansion.

2.2.4 Multi-level Logic Networks

A *multi-level logic network* is an interconnection of logic blocks modeled as a directed acyclic graph (DAG), having nodes associated with Boolean functions. Generally, multi-level circuits for both standard-cell and FPGA design tend to be much smaller, power efficient, and faster, compared to the two-level counterparts. The sources of the graph are the *primary inputs* (PIs), the sinks are the *primary outputs* (POs). For any node *n*, the *fan-ins* of *n* is a set of nodes driving *n*, i.e. nodes that have an outgoing edge towards *n*. Similarly, the *fan-outs* of *n* is a set of nodes driven by node *n*, i.e., nodes that have an incoming edge from *n*. If there is a path from node *a* to node *b*, then *a* is in the *transitive fan-in* (TFI) of *b*, and *b* is said to be in the *transitive fan-out* (TFO) of *a*. The transitive fan-in of *b* includes node *b* and the nodes in its transitive fan-in, including the PIs. The *transitive fan-out* of *b* includes *b* and all the nodes in its transitive fan-out including the POs. Note that logic networks can be extended to represent sequential circuits. In this thesis, we focus on combinational circuits.

A natural extension of SOPs into a multi-level representation are *factored forms* [28]. A factored form is defined recursively as follows. A literal is a factored form, and the logic OR or logic AND of two factored forms is a factored form. Informally, a factored form is an SOP whose inputs are other SOPs, etc. Factored forms are an interesting type of multi-level representation because they efficiently abstract CMOS transistor networks composed of transistors connected in series and in parallel. Moreover, they provide a method to reduce the number of literals of a SOP.

Example 2.2.6. *A sum-of-products $ab + ac + ad + cbd$ can be extended to factored form by sharing the literal a among multiple cubes, obtaining $a(b + c + d) + cbd$.* ▲

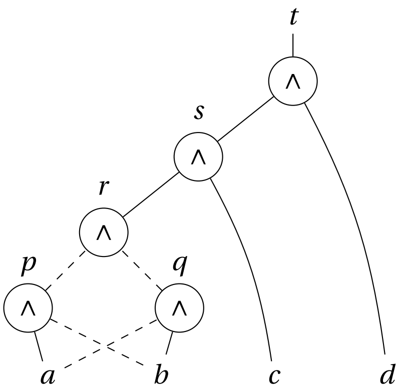
Logic synthesis often uses *homogeneous* logic networks to represent Boolean circuits, which use restrictions on the type of Boolean functions and fan-in of logic blocks. Using a small set of primitives makes DAGs easy to manipulate, for example, through algebraic rules, and improves the memory efficiency. Originally, 2-input NANDs and NORs, together with inverters, were used as primitives in graph representations thanks to their universality. As logic synthesis evolved, the *and-inverter graph* (AIG) [75, 97], consisting of 2-input AND gates and inverters, became the most common technology-independent representation. Other practical graph representations are the *majority-inverter graph* (MIG) [5, 6], using 3-input majority and inverters, the *and-xor graph* (XAG) [72] (often called *and-xor-inverter graph* (XAIG)), using 2-input AND, 2-input XOR, and inverters, and the *xor-majority graph* (XMG) [68], using 3-input majority, 3-input XORs, and inverters. A *k-LUT network* is a logic network used in FPGA designs and composed of k-input *lookup tables* (k-LUTs), capable of realizing any k-input Boolean function. In logic networks for standard-cell design flows each internal node, excluding the primary inputs and primary outputs, represents a standard cell.

2.3 Cuts and Partitions

Due to the intractability of most logic synthesis problems, optimization is often performed on small sub-networks, known as *windows* or *peephole*. Consequently, it is crucial to develop efficient techniques to extract sub-networks or partition a Boolean network. In this section, we first present *cuts*, which are an essential part of technology mapping and many optimization algorithms, and a method to enumerate them. Then, we describe *windowing* as a method for peephole optimization, such as Boolean resubstitution.

2.3.1 Cuts and Cut Enumeration

A *cut C* of a Boolean network is a pair $(n, \mathscr{K})$, where n is a node called *root*, and $\mathscr{K}$ is a set of nodes, called *leaves*, such that 1) every path from any PI to node n passes through at least one leaf; and 2) for each leaf $v \in \mathscr{K}$, there is at least one path from a PI to n passing through v and not through any other leaf. The *size* of a cut is defined as the number of leaves. A cut is k-feasible if its size does not exceed k. A cut *covers* (i.e., includes) all the nodes encountered on the paths between the leaves and the root, including the root and excluding the leaves. A *multiple-output cut* is an extension of a cut defined over a set of roots $\mathscr{L}$. A cut rooted in a



$$
\begin{aligned}
\Phi(a) &= \{\{a\}\}\\
\Phi(b) &= \{\{b\}\}\\
\Phi(c) &= \{\{c\}\}\\
\Phi(d) &= \{\{d\}\}\\
\Phi(p) &= \{\{a,b\},\{p\}\}\\
\Phi(q) &= \{\{a,b\},\{q\}\}\\
\Phi(r) &= \{\{a,b\},\{a,b,p\},\{a,b,q\},\{p,q\},\{r\}\}\\
\Phi(s) &= \{\{a,b,c\},\{p,q,c\},\{r,c\},\{s\}\}\\
\Phi(t) &= \{\{r,c,d\},\{s,d\},\{t\}\}
\end{aligned}
$$

Figure 2.4: Example of a Boolean network with the corresponding 3-feasible cuts.

node n is named *trivial* if it consists of only node n itself

The *maximum fan-out free cone* (MFFC) of a node n is a subset of the transitive fan-in of n such that every path from the nodes in the MFFC to the POs passes through n. Informally, the MFFC of a node contains the node itself and all the logic exclusively used by the node. When a node is removed (or substituted) the logic in the MFFC can also be removed. The MFFC can be extended to operate on a set of roots $\mathscr{L}$ such that every path from the nodes in the MFFC to the POs passes through at least one node in $\mathscr{L}$.

The enumeration of cuts is an essential part of technology mappers, and optimization algorithms, such as *rewriting*. Here we show the traditional method based on dynamic programming to enumerate k-feasible cuts in a logic network [47, 155]. The computation proceeds in topological order from the primary inputs (PIs) to the primary outputs (POs).

Let N be a logic network where V is the set of nodes, and O is the set of POs. Let $I \in V$ be the set of PIs of the network, and $G \in V$ the set of internal nodes ($V \setminus (I \cup \{0,1\})$). Let $FI(n)$ be the set of immediate fan-in nodes of node $n \in V$ and $\Phi(n)$ represent the set of *k-feasible* cuts at node $n \in V$. We define recursively Φ as:

$$
\begin{aligned}
&\Phi(0) = \Phi(1) = \{\{\}\} && \\
&\Phi(x) = \{\{x\}\} && \text{for } x \in I\\
&\Phi(n) = \{\{n\}\} \cup \Big(\bigotimes_{i \in FI(n)} \Phi(i)\Big) && \text{for } n \in G
\end{aligned}
\tag{2.5}
$$

where $FI(n)$ indicates the set of direct fan-in nodes of n, and the *merging* operation $\otimes$ is defined as:

$$
A \otimes B = \{u \cup v \mid u \in A, v \in B, |u \cup v| \le k\}. \tag{2.6}
$$

Example 2.3.1. *Figure 2.4 shows a Boolean network and the* 3*-feasible cuts for each node enumerated using the method in Equations 2.5 and 2.6. For instance, cut* $(s, \{a, b, c\})$*, rooted at node* s*, covers the nodes* p*,* q*,* r*, and* s*.* ▲

Another approach for cut computation was proposed in [39] to overcome memory issues when enumerating large cuts (for $k > 7$) because the number of cuts is simply too high. The idea is to "factor" cuts in two sets called *global* and *local cuts*. Cut enumeration computes and saves only local and global cuts at each node. Then, a larger set of cuts can be extracted by expanding factor cuts with respect to local cuts on the fly, without a need of storing them. The method allows for both complete and partial enumeration of cuts to save memory. For further details, we refer the reader to [39].

Related to the concept of cuts, there is a notion of *cover* of a logic network given a set of computed cuts, with applications in technology mapping. A *cover* of a logic network is a set of cuts such that 1) each node in the network is covered by at least one cut; and 2) the root of each cut in the cover is either a PO of the network or a leaf of one or more cuts in the cover. A cover can be extracted in reverse topological order by selecting cuts rooted in the POs and recurring on the leaves.

2.3.2 Windowing

In a Boolean network, a path is a finite sequence of connected nodes $v_0 \to \cdots \to v_l$ where (v_i, v_{i+1}) are connected with an edge. Two paths are *reconvergent* if they start at the same node v_0 and join at the same node v_l arriving from two different fan-ins of v_l. Identifying reconvergent paths is crucial in logic optimization because reconvergence enables don't care conditions. A *reconvergence-driven cut* [132] is a type of cut constructed to include reconvergence paths. This type of cut is used in Boolean methods such as Boolean resubstitution to leverage don't cares. A reconvergence-driven cut with multiple outputs is referred to as a *reconvergence-driven window*. For additional details and algorithms to extract windows, we refer the reader to [132].

2.4 Matching

Matching is the process of recognizing the equivalence of pairs of functions under certain conditions. Typically, matching is defined over a set of *equivalence classes*. For example, matching may determine whether there exists an input permutation of variables that make two functions Boolean equivalent. Matching plays a pivotal role in many parts of logic synthesis. In technology mapping algorithms, matching binds Boolean functions to library cells that can implement the function under some configuration (e.g., input permutations and negations). In rewriting algorithms, matching binds Boolean functions to pre-computed cost-optimum graphs implementing their function. In this section, we first present useful *equivalence classes* for synthesis and technology mapping. Then, we describe two main techniques for matching, namely, *pattern matching* and *Boolean matching*. Additionally, we provide an overview of *generalized matching*.

2.4.1 Equivalence Classes

Consider two functions $f(x_0, \ldots, x_{k-1})$ and $g(x_0, \ldots, x_{k-1})$, defined over the same variable set $\vec{x}$. The two functions are $\mathcal{NPN}$-equivalent if there exists an inversion of the inputs $\mathcal{N}_I : (x_i \rightarrow \bar{x}_i)$, a permutation of the inputs $\mathcal{P}_I : (x_i x_j \rightarrow x_j x_i)$, or an inversion of the output $\mathcal{N}_O : (f \rightarrow \bar{f})$ such that f and g can be made Boolean equivalent [23]. Similarly, $\mathcal{N}$-, $\mathcal{P}$-, and, $\mathcal{NP}$-equivalence classes are defined considering input negations, input permutations, and both input negations and permutations, respectively.

Matching is commonly defined in terms of $\mathcal{N}$-, $\mathcal{P}$-, $\mathcal{NP}$-, or $\mathcal{NPN}$-equivalence classes. For instance, over $\mathcal{P}$-equivalence classes, matching of two functions f and g searches for a permutation operator $\mathcal{P}_I$ such that $f(\vec{x}) \bar{\oplus} g(\mathcal{P}_I \vec{x})$ is a tautology.

Example 2.4.1. *Let us consider two functions $f = ab + cd$ and $g = \bar{a}c + b\bar{d}$. The two functions are $\mathcal{NPN}$-equivalent under the input operator $\mathcal{N}_I \mathcal{P}_I = \{abcd \rightarrow \bar{a}cb\bar{d}\}$.* ▲

Equivalence classes are used as solutions to reduce the memory footprint of cell libraries and databases of functions. This process avoids saving all the configurations of the cells based on permutations and negations. For k-inputs, 2^{2^k} different Boolean functions exist. Using $\mathcal{NPN}$ classes reduces the number of k-input Boolean functions to $14, 222$ and 616126 classes, for $k = 3, 4, 5$, respectively.

For additional details on how to perform the enumeration of equivalence classes for small and large functions, we refer the reader to [82, 182].

2.4.2 Pattern Matching

Structural approaches were the first adopted methods to bind Boolean functions to cells, for instance those contained in standard cell libraries. Cells were represented using a pattern (graph), typically in the form of a 2-input NAND decomposition of the Boolean functionality. The matching task was then formulated as a (sub)graph isomorphism problem, particularly efficient when the decomposition graph is a tree. A cell can be associated with a sub-graph if one of its patterns matches the sub-graph, i.e., they are structurally equivalent. This form of matching became known as *pattern matching* [90]. However, while being simple, this approach has several limitations. The most important one is that the graph decomposition is not unique. Consequently, the number of possible graph decompositions can grow exponentially large for some functions, making it challenging to detect potential matches. Moreover, the matching process is significantly more involved when the decomposition graph is not a tree, such as in the case of XOR gates, because the decomposition has reconvergence paths.

Generally, a database contains a family of patterns for each cell. Initially, potential decompositions for cells into 2-input NANDs were manually created and stored in a database. However, as more robust algebraic and Boolean methods emerged, decompositions began to be automatically generated. Technology mapping algorithms relying on pattern matching, are termed rule-based, as the pattern databases contained rules for matching [65, 84, 90].

2.4.3 Boolean Matching

Another approach to matching, called *Boolean matching* [123], uses a canonical Boolean representation of functions to address the non-canonicity problem of pattern matching. Boolean matching inherently solves a tautology problem between a target Boolean function and a set of functions representing library cells. Additionally, Boolean matching may consider variable permutation, phase assignment, and don't care conditions along with the matching problem.

Example 2.4.2. *Let us consider two Boolean functions $f = x_0x_1 + \bar{x}_0\bar{x}_1 + \bar{x}_1x_2$ and $g = x_0x_1 + \bar{x}_0\bar{x}_1 + x_0x_2$. Although the two functions differ from the last term, they are Boolean equivalent and Boolean matching can detect the equivalence. However, since the f and g are different and have different decomposition trees, pattern matching might not detect the equivalence.* ▲

Original approaches to solve Boolean matching were based on recursive Shannon decomposition and on filters based on unate/binate variable matching and symmetry properties. While these methods are still in use for large functions, when small functions up to 6 inputs are involved, modern approaches use truth tables as a canonical data structures. Boolean matching is used in many state-of-the-art technology mappers and supports the majority of the cells present in standard cell libraries.

2.4.4 Generalized Matching

Generalized matching (GM) [23] is a multiple-output Boolean matching technique that supports matching multiple single-output cells or one multiple-output cell with a multiple-output Boolean function expressed as a Boolean relation [184]. Generalized matching has two main advantages compared to standard matching methods. First, it supports mapping to multiple-output cells, which are commonly available in standard libraries, such as full adders. Second, by leveraging Boolean relations, GM can achieve lower-cost mappings compared to traditional matching algorithms, even when considering don't care conditions. However, the significantly higher computational complexity of GM makes it impractical for use in global technology mappers.

Generalized matching has been used to iterative remap logic circuits by locally replacing clusters of two or more cells with a more efficient implementation [22]. Since GM is used during the remapping phase, accurate cost functions to minimize delay, dynamic power, or area can be considered. For more details on GM, we refer the reader to [22, 23].

2.5 Algorithms

In this section, we provide an overview of logic optimization algorithms for multi-level networks. We first present *algebraic* methods, which are based on polynomial algebra. Then, we introduce *Boolean* methods, which are based on Boolean algebra. Finally, we describe exact synthesis methods.

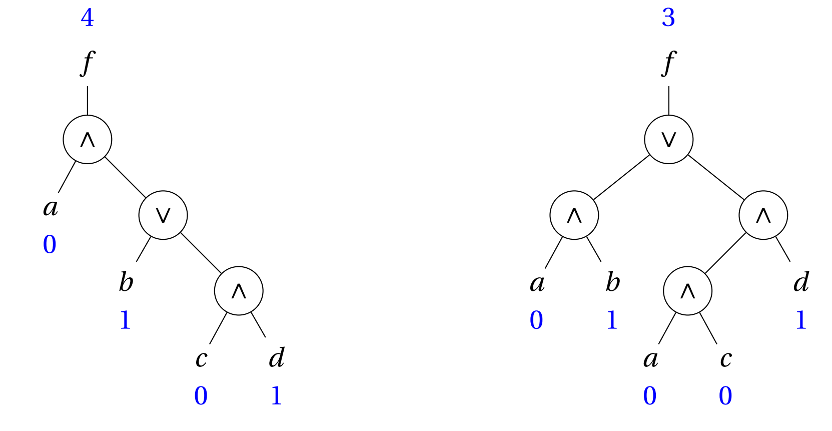
2.5.1 Algebraic Methods

Algebraic methods leverage the algebraic model to represent Boolean functions as algebraic expressions (or polynomials) [52]. This abstraction simplifies the manipulation of large networks by neglecting Boolean properties, thus enabling the development of fast optimization techniques. Traditionally, these methods were designed for multi-level networks where each node is described in SOP form. Various optimization techniques were developed, including *factoring*, *substitution*, *extraction*, *decomposition*, and *algebraic rewriting* (see, e.g., [28, 29, 52]). The effectiveness and scalability of these methods were supported by theories on weak-division and kernel extraction.

One important algebraic method is *factoring*, which involves transforming a sum-of-products (SOP) form into a compact factored form by identifying and extracting common sub-expressions. The primary objective is to find a factorization that minimizes the number of literals. Traditionally, the number of factored form literals in a logic network served as a key metric for optimization, leading to the development of many algorithms to compute minimal factored forms. While exact algorithms have been created to determine the optimal factored form [100, 101], their high computational complexity often makes them impractical in logic manipulation. Consequently, multiple heuristic algorithms relying on kernel extraction and algebraic division have been proposed [28]. These algorithms have proven to be highly effective and still remain in use in modern logic synthesis tools.

Algebraic methods for homogeneous logic networks, such as AIGs, XAGs, and MIGs, primarily utilize algebraic transformation rules based on algebraic axioms. For example, *balancing* uses the associative property to reduce circuit depth [143]. Algebraic rewriting employs additional rules to replace small cones of logic with improved implementations, enhancing circuit size or depth (see, e.g., [5, 194]).

Example 2.5.1. *Figure 2.5 shows an algebraic rewriting transformation over an AND-OR graph using the rule* $a \wedge (b \vee (c \wedge d)) \rightarrow (a \wedge b) \vee ((a \wedge c) \wedge d)$*. This rule first applies the distributive property to* a *and then balances the graph. Considering the input arrival times shown in blue, this transformation reduces the circuit depth from* 4 *to* 3 *while increasing the node count by one.* ▲



(a) Before algebraic rewriting

(b) After algebraic rewriting

Figure 2.5: Rewriting with AND-OR distributive rule $a \wedge (b \vee (c \wedge d)) \rightarrow (a \wedge b) \vee ((a \wedge c) \wedge d)$.

2.5.2 Boolean Methods

Contrarily to algebraic methods, Boolean methods fully leverage the power of the Boolean model, using Boolean identities and *don't care* conditions [52]. Don't care conditions are functional flexibilities related to the environment around a Boolean function and play a crucial role in logic synthesis. The don't care conditions at the primary inputs and primary outputs of a local function are referred to as *external* and consist of the *controllability* and *observability* components. Controllability don't cares are input patterns that are never produced by the environment at the function's inputs. Observability don't cares are input patterns that produce outputs that are not observed by the environment and, thus, can be ignored. Boolean methods leverage the additional degrees of freedom given by Boolean algebra and don't care conditions to achieve better solutions, in general, compared to algebraic methods. However, these methods have also a higher computational complexity and inferior scalability. Traditional Boolean methods include *Boolean simplification* and *Boolean substitution* (or *resubstitution*) (see, e.g., [28, 52]).

Example 2.5.2. *In Boolean simplification, a local Boolean function f can use the external don't care conditions (DC_{ext}) to minimize its implementation cost, thereby obtaining a new Boolean function f' that is locally different but globally equivalent in the environment to the original, as long as the difference is contained in its don't care set ($f \oplus f' \subseteq DC_{ext}$).* ▲

For a local Boolean function, functional flexibilities due to don't care conditions define a set of *permissible functions*, such as f' in Example 2.5.2. The set of all permissible functions it is called *maximum set of permissible functions* [151]. Boolean methods rely on an efficient computation of don't care conditions, for which many algorithms exist based on Boolean algebra, image computation, BDDs, and *satisfiability* (SAT) solving (see, e.g., [52, 136, 176]).

Multiple Boolean methods have been proposed for homogeneous logic networks, including *Boolean resubstitution*, *Boolean rewriting*, *Boolean factoring*. In this section, we review

Boolean resubstitution and Boolean rewriting.

Boolean resubstitution

Resubstitution [132], often shortened to *resub*, (re)expresses the function of a node using other nodes, called *divisors*, that are already present in the network. The transformation is accepted if the new implementation of a node is better, according to a target metric (e.g., size), compared to the current implementation of the node in terms of its immediate fan-ins. This approach generalizes to *k-resubstitution*, which adds k new nodes and removes at least $k+1$ nodes. The removed nodes are the ones present in the *maximum fan-out free cone* (MFFC) [132] of the node. The functionality of the new nodes is derived from a library of primitives used for resubstitution or a composition of primitives. For instance, in an AIG, added gates are 2-input ANDs with optional inverters at the inputs/outputs. More complex primitives, such as XORs and MUXes, have also been used for resubstitution [9]. Boolean resubstitution leverages don't care conditions to find additional opportunities beyond those available through algebraic substitution. The quality of resubstitution largely depends on the collection of divisors and the adopted resynthesis heuristics.

Modern scalable resubstitution is achieved using windows or simulation signatures, to overcome the limitations in the number input variables and divisors. In *window-based resubstitution* [132], a window (or logic region), typically up to 12 or 16 inputs, is structurally built around the target node. Resubstitution is then performed by treating the inputs (outputs) of the window as primary inputs (outputs) of the circuit. This methods allows for a feasible computation of don't cares and a manageable number of collected divisors. In *simulation-guided resubstitution* [108], the entire circuit is considered, but the nodes' functionality is extracted from simulation using a limited number of patterns, typically up to 1024. This has the advantage of considering global controllability don't cares. Divisors are extracted using large windows that are not limited by the number of inputs. Then, resubstitution is performed approximately on the simulation patterns and subsequently verified for functional correctness using SAT-based equivalence checking.

Boolean rewriting

Rewriting [137] is a fast greedy algorithm that aims at minimizing the size of a logic network. It does so by iteratively replacing sub-graphs identified by structural cuts rooted in a node with smaller pre-computed structures while preserving the functionality at the root node. Typically, pre-computed structures cover all the 4-variable functions, which are classified into the NPN equivalence classes for compactness and extracted using exact synthesis. Rewriting has been implemented for many homogeneous logic networks representations, including AIGs, XAGs, MIGs, k-LUTs, and XMGs (see, e.g., [68, 137, 167, 169, 196, 202]).

2.5.3 Exact Synthesis Methods

Most of the methods reviewed in this section are heuristic, reflecting the inherent intractability of many synthesis problems. Here, we provide an overview of exact synthesis methods, which aim to find the minimum-cost logic representation for a given Boolean function. Typically, the cost criterion is the number of gates (correlated with the area) or the logic network depth (correlated with the delay). Examples of exact methods in the literature include finding the minimum SOP with respect to the number of implicants (Quine-McCluskey, Espresso exact [171]), the minimum factored form with respect to the number of literals [101], and the minimum-size logic network consisting using 2-input gates or other primitives [69]. Exact methods for multi-level networks generally involve techniques such as enumeration or SAT solving.

The *Boolean satisfiability* (SAT) problem asks if there exists a value assignment to the input variables of a formula that makes it *satifiable* (SAT, i.e., evaluates to true). This assignment is called a *satisfying assignment.* SAT solvers are programs that receive a Boolean formula, typically represented in CNF, and return a satisfying assignment, if one exists. SAT-based exact synthesis for minimum size encodes the problem as a CNF formula and asks whether there exists a multi-level logic network involving r gates. Initially, r is set to 0, and the value is incremented in a loop until the problem becomes satisfiable, indicating that an optimum-size implementation has been found. SAT-based exact synthesis has been shown to be practical for computing minimum-size implementations up to 5-input functions using 2-input gates. In this context, NPN classification significantly reduces the number of functions that need to be synthesized and stored, enabling the creation of a database of optimum implementations for Boolean rewriting. For more details on SAT-based exact synthesis and its encoding, we refer the reader to [69, 180, 181].

2.6 Benchmark suites

In this section, we present the benchmark sets used throughout the experiments in this thesis. For each circuit, we provide its name, the number of inputs and outputs, as well as its size and depth in an unoptimized AIG representation.

EPFL combinational benchmark suite

The EPFL combinational benchmark suite [3][1], introduced in 2015, was designed to establish a new comparative standard for the logic optimization and synthesis community. It originally comprised twenty combinational circuits intended to challenge modern logic optimization tools. The suite is categorized into arithmetic and random or control circuits. Additionally, it includes three *more than a million* (MtM) circuits, which are not utilized in the experiments of this thesis. Table 2.1 shows the characteristics of the EPFL circuits.

[1] The EPFL circuits are available at the following link: https://github.com/lsils/benchmarks.

Table 2.1: The EPFL combinational benchmark suite.

Type	Benchmark	Inputs	Outputs	AND nodes	Levels
Arithmetic	adder	256	129	1020	255
	bar	135	128	3336	12
	div	128	128	57247	4372
	hyp	256	128	214335	24801
	log2	32	32	32060	444
	max	512	130	2865	287
	multiplier	128	128	27062	274
	sin	24	25	5416	225
	sqrt	128	64	24618	5058
	square	64	128	18484	250
Random/control	arbiter	256	129	11839	87
	cavlc	10	11	693	16
	ctrl	7	26	174	10
	dec	8	256	304	3
	i2c	147	142	1342	20
	int2float	11	7	260	16
	mem_ctrl	1204	1231	46836	114
	priority	128	8	978	250
	router	60	30	257	54
	voter	1001	1	13758	70

IWLS 2005 benchmark suite

The IWLS 2005 benchmark suite [85] includes 84 open-source circuits, featuring up to 185,000 registers and 900,000 cells. This suite comprises circuits from OpenCores, Gaisler Research, Faraday Technology Corporation, ITC 99, and ISCAS 85 and 89. In this thesis, we utilize a subset of 27 of these circuits, specifically from the first three collections, represented as AIGs[2]. The circuits were originally provided mapped to a 180nm technology. The characteristics of the circuits are shown in Table 2.2.

Superconducting benchmark suite

The superconducting benchmark suite includes circuits from ISCAS 85, represented as MIGs instead of AIGs, provided by the authors of [35][3]. The characteristics of these circuits are detailed in Table 2.3. Additionally, we present the maximum fan-out of nodes, including primary inputs, as this parameter has important implications in superconducting electronics.

[2] The IWLS 2005 circuits are available in the logic synthesis tool Mockturtle at the following link: https://github.com/lsils/mockturtle/tree/master/experiments/benchmarks.

[3] The superconducting circuits are available at the following link: https://github.com/lsils/SCE-benchmarks/tree/main/ISCAS.

Table 2.2: The IWLS 2005 benchmark suite.

Benchmark	Inputs	Outputs	AND nodes	Levels
ac97_ctrl	4482	2251	14268	12
aes_core	1319	668	21522	26
des_area	496	72	4857	33
des_perf	17850	9038	82650	20
DMA	5070	2559	24393	27
DSP	7835	3954	45420	63
ethernet	21216	10698	86726	32
iwls05_i2c	275	144	1166	14
leon2	298888	291880	789647	58
leon3_opt	370159	252691	974977	54
leon3	370159	252691	1088122	59
leon3mp	217858	142925	652353	55
iwls05_mem_ctrl	2281	1226	15337	36
netcard	195730	97805	803848	40
pci_bridge32	6880	3533	22806	30
RISC	15678	8111	75613	40
sasc	250	132	773	9
simple_spi	280	147	1053	12
spi	505	277	3808	32
ss_pcm	193	98	405	7
systemcaes	1600	819	12384	46
systemcdes	512	258	2999	27
tv80	732	404	9647	52
usb_funct	3620	1858	15894	27
usb_phy	211	111	460	10
vga_lcd	34247	21412	126708	24
wb_conmax	2670	2189	47853	27

2.7 Summary

In this chapter, we introduced state-of-the-art data structure and algorithms used in logic synthesis. We also explored various optimization methods, including algebraic, Boolean, and exact approaches. These methods form the foundation of modern logic synthesis, enabling efficient manipulation and optimization of logic circuits. In the remainder of the thesis, we will use truth tables, BDDs, and logic networks to represent Boolean functions and to perform circuit transformations. Additionally, we will extensively refer to partitioning and matching algorithms, which are essential in technology mapping. This chapter provides the fundamental background needed to understand the algorithms and methodologies proposed in the rest of the thesis.

Table 2.3: The superconducting benchmark suite.

Benchmark	Inputs	Outputs	3-MAJ nodes	Levels	Max fan-out
adder1	3	2	7	4	2
adder8	17	9	77	17	3
mult8	16	16	439	35	9
counter16	16	5	29	9	4
counter32	32	6	82	13	4
counter64	64	7	195	17	4
counter128	128	8	428	22	4
c17	5	2	6	3	2
c432	36	7	121	26	10
c499	41	32	387	18	8
c880	60	26	306	27	9
c1355	41	32	389	18	9
c1908	33	25	289	21	14
c2670	157	64	368	21	32
c3540	50	22	794	32	38
c5315	178	123	1302	26	41
c6288	32	32	1870	89	17
c7552	207	108	1394	33	170
sorter32	32	32	480	15	2
sorter48	48	48	880	20	3
alu32	68	65	1513	100	128

3 Technology Mapping for FPGAs

The goal of this thesis is to develop and enhance technology mapping algorithms for multiple technologies and, more broadly, for logic synthesis. Each technology presents unique constraints and requires specialized techniques, which are addressed in different chapters of this thesis. Following a brief introduction on state-of-the-art synthesis methods and relevant background in Chapter 2, we transition into the first technical section of this research work. This chapter focuses on novel technology mapping algorithms for field-programmable gate arrays (FPGAs), where a synthesized logic network is transformed into an interconnection of *lookup tables* (LUTs). Specifically, this chapter analyzes: (i) practical algorithms for the Boolean decomposition of large functions into LUTs; (ii) an advanced delay-driven technology mapping algorithm that integrates Boolean decomposition to reduce the structural bias; (iii) a technology mapper that leverages non-routable connections in FPGAs to reduce the worst-case delay. The content of this chapter is largely based on the publications in [144, 198, 199].

The remainder of this chapter is organized as follows. First, we present the motivations of this chapter in Section 3.1 and the relevant background on technology mapping for FPGAs and Boolean decomposition in Section 3.2. Next, Section 3.3 proposes an efficient algorithm to compute the Ashenhurst-Curtis decomposition (ACD) of functions into LUTs. We propose several improvements that make ACD applicable to LUT mappers and resynthesis engines. We provide algorithms to minimize the decomposition cost in terms of the number of LUTs, edges, and delay, considering input arrival times. The experimental results show that our approach runs up to 80 times faster compared to state-of-the-art Boolean decomposition methods while achieving the decomposition success of an optimum SAT-based implementation. Then, Section 3.4 presents a technology mapping algorithm that integrates the ACD decomposition of Section 3.3 in a delay-driven LUT mapper to achieve better delay results. The experimental results show that LUT mapping with ACD improves the delay of circuits in the EPFL benchmark suite by 12.39%, on average, compared to the state-of-the-art mapper with choices. Additionally, it discovers new best implementations in the EPFL competition. Next, Section 3.5 describes methods to leverage fast (non-routable) connections between adjacent LUTs in FPGAs [10] to minimize the delay. The proposed approach is based on Boolean decomposition into structures of 2 LUTs arranged in cascade. We show that our implementation outperforms the state-of-the-art methods in delay, area, edge count, and run time by 6.22%, 3.82%, 3.09%, and 20%, respectively. Finally, Section 3.6 concludes and summarizes this chapter, highlighting the key findings and contributions.

3.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are integrated circuits with configurable logic blocks and programmable interconnects. Unlike standard-cell-based designs, which follow a semi-custom design methodology and have a fixed configuration, FPGAs can be programmed many times, which comes at the cost of lower power-performance-area (PPA) metric. FPGAs are widely used for rapid prototyping, in low-volume applications, and for hardware acceleration of specific tasks.

Logic synthesis for hardware designs intended to run on FPGAs shares similarities with those for standard-cell-based design, but the target primitive is a k-input *lookup table* (LUT), capable of implementing any Boolean function up to k inputs. Specifically, this chapter focuses on mapping technology-independent combinational logic into networks composed of k-LUTs.

State-of-the-art technology mapping into LUTs is performed through local substitutions applied to an initial graph representation, called the *subject graph*. The drawback of this approach is that the technology-independent optimization step and the technology mapping step are separated. Consequently, the impact of optimization on the quality of the final LUT network is hard to predict before mapping. Delay-optimal mapping for a fixed subject graph is feasible in polynomial time [45]. Area-optimal mapping is NP-hard [59]. Specifically, the structure of the subject graph highly influences the mapping quality. This is known as *structural bias*. To mitigate structural bias, the known methods compute structural choices for the subject graph and use them during mapping [37, 114], or collapse and decompose parts of the graph during mapping [41, 60, 112]. However, exact area and delay optimization during LUT mapping remain NP-hard [46, 124]. This chapter explores the use of Boolean decomposition to enhance delay-driven LUT mapping.

On another note, the performance of modern FPGAs is limited by programmable interconnect. Specifically, the interconnect delay can be five times or more higher than the intrinsic delay of a LUT because wires are routed through multiple switch boxes and routing channels. One solution adopted by FPGA vendors is to supplement programmable interconnect with non-routable (fixed) connections between adjacent LUTs within a slice, creating LUT structures such as LUT cascades [10]. However, existing placement algorithms struggle to effectively utilize these connections because this requires introducing LUT structures after LUT mapping. Alternatively, Boolean decomposition has emerged as an efficient way of generating LUT structures during mapping [165].

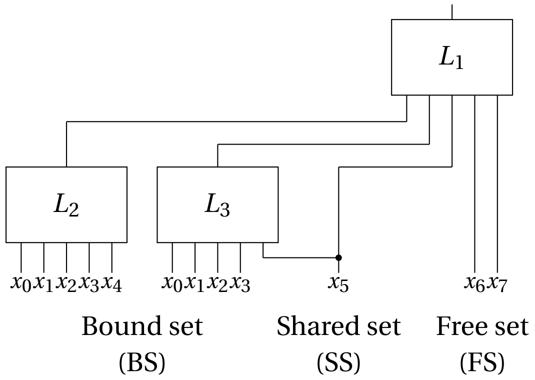


Figure 3.1: ACD of an 8-input Boolean function into three 5-input LUTs with a 5-variable *bound set* (BS), a 1-variable *shared set* (SS), and a 2-variable *free set* (FS).

The Ashenhurst-Curtis decomposition (ACD) [12, 49], also known as Roth-Karp decomposition [170], is a powerful technique to decompose a Boolean function into a set of sub-functions and a composition function with reduced support. ACD finds applications in logic optimization and technology mapping. The traditional formulation of ACD breaks the input variables into two groups: the bound set (BS) and the free set (FS). Other approaches to ACD [113] allow for a shared set (SS) when some functions in terms of the BS variables are buffers. The larger the SS size, the fewer sub-functions are required. For instance, Figure 3.1 shows an ACD of a function with BS, FS, and SS, resulting in three 5-input LUTs. Conventional methods leverage *binary decision diagrams* (BDDs) [32] to perform ACD [113, 161, 208]. More recent approaches use truth tables for functions up to 11 or 16 inputs [133, 165].

This chapter has three main contributions. First, we revisit the formulation of ACD with shared set to enhance its computationally efficiency in LUT mappers and post-mapping resynthesis engines performing delay optimization. Our algorithm is truth-table-based and flexible in the number of FS, BS, and SS variables, and in the number of BS functions. Our ACD runs up to 2x faster, compared to [165], and up to 80x faster, compared to [133], when performing decompositions into the LUT structure "66" composed of two 6-LUTs. Furthermore, the proposed method finds considerably more solutions, which translates into better quality of results.

Second, we use ACD for the delay optimization of LUT networks. The idea is to compute functional decompositions using timing-critical variables in the FS and the rest of the variables in the BS and SS. This method is more general than cofactoring w.r.t. late arriving variables using Shannon expansion [134] and leads to improved quality of results. We integrate our ACD into the state-of-the-art LUT mapper for delay optimization. To our knowledge, this is the first practical and scalable work that uses ACD for delay-driven LUT mapping.

Third, we propose a technology mapper that uses ACD to leverage the non-routable cascade connections between LUTs for delay optimization.

We experimentally evaluate the use of ACD for LUT mapping by comparing the results with state-of-the-art methods:

1. We show that the proposed ACD method has a higher decomposition success ratio, up to 32.58% more than state-of-the-art, and a better or competitive run time.

2. We demonstrate that mapping with ACD can efficiently mitigate the structural bias and considerably reduce the delay. We compare the traditional LUT mapper in ABC, the LUT-structure mapper in ABC, and the proposed mapper with integrated ACD. We show that mapping with ACD notably outperforms the other mappers in delay by 7.52% on average, also when using structural choices [37]. Moreover, we show that an additional mapping round using the network obtained by ACD as a structural choice can further improve the delay, compared to the baseline, by 12.39%, with a surprising area reduction of 2.20%.

3. We present four new best results in the EPFL competition. These results have been obtained using delay-oriented mapping with ACD and without employing design-space exploration (DSE) methods. Hence, we expect even better results by using LUT mapping with ACD in a DSE tool.

4. We use this new ACD formulation to compute mappings into LUT structures composed of 2 LUTs with a non-routable connection between them. Compared to the state-of-the-art approach [165], our method reduces the average delay, area, and edge count by 6.22%, 3.82%, and 3.09%, respectively, with better run time. In particular, this new formulation is exact, i.e., it always guarantees a solution for functions decomposable into 2 LUTs.

3.2 Preliminaries

In this chapter, we research algorithms to solve the technology mapping problem for combinational networks targeting FPGAs. Our focus is on Boolean decomposition methods to improve mapping. In the following sub-sections, we introduce the basic notations, background, and related work on Boolean decomposition and LUT mapping.

3.2.1 Boolean Decomposition

Boolean decomposition refers to the process of breaking down a Boolean function into simpler components. Boolean decomposition produces a Boolean network with POs functionally equivalent to the original function. The most generic decomposition is the Ashenhurst-Curtis decomposition (ACD) [12, 49, 170]. The ACD of a single-output Boolean function f can be expressed as follows:

$$f(\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}) = g(\vec{h}(\vec{x}_{bs}, \vec{x}_{ss}), \vec{x}_{ss}, \vec{x}_{fs}), \tag{3.1}$$

where $\vec{x}_{bs}$ is the *bound set* (BS), $\vec{x}_{ss}$ is *shared set* (SS), and $\vec{x}_{fs}$ is the *free set* (FS). These sets are disjoint variable subsets, which together form the support of f. The function $\vec{h}$ may be multiple output with the number of outputs less than the BS size. The single-output functions in $\vec{h}$ are referred to as BS functions. The function g is referred to as the *composition function*. When decomposing into k-LUTs, the composition function is typically chosen to fit into one k-input LUT.

Example 3.2.1. *Figure 3.1 shows an ACD of an* 8*-input function into three* 5*-input LUTs with a* 5*-variable BS, a* 1*-variable SS, and a* 2*-variable FS. The decomposition generates two BS functions (*L_2*,* L_3*) and a composition function (*L_1*).* ▲

The *disjoint-support decomposition* (DSD) [25] is a decomposition where the set of nodes have disjoint support. Hence, the Boolean network generated from DSD is always a tree. ACD generates a DSD decomposition when $\vec{x}_{ss} = \emptyset$ and BS functions have disjoint support.

The *Shannon decomposition* is a Boolean decomposition based on the Shannon expansion:

$$f = x f_x + \bar{x} f_{\bar{x}}. \tag{3.2}$$

The result of applying the Shannon decomposition to all variables and merging identical cofactors, is a BDD.

Related works

Traditionally, Boolean decomposition is implemented using BDDs [95, 113, 161, 208], derived by applying the Shannon decomposition to all variables in a given order and using reduction rules. Typically, multiple variable orderings are explored to find a partition of variables into *bound set* (BS) and *free set* (FS) and perform a support-reducing ACD [208]. However, algorithms that perform ACD suffer from slow run time and poor performance on large functions. To enhance efficiency, conventional methods often restrict decomposition to a limited set of primitives, such as 2-input operators and multiplexers [185, 214], and compute only *disjoint support decompositions* [24, 25, 36]. For instance, the logic optimization system BDS [214] can perform decomposition and optimization using AND, OR, XOR, and MUX operators over BDDs. Additionally, the tool BDS-pga [208] extended BDS to map circuits to LUTs.

Recent advancements have leveraged truth tables for ACD up to 16 variables, either by replicating variable re-ordering and size minimization of BDDs without explicitly constructing one or by computing a DSD that minimizes the required number of LUTs. Specifically, in [165], the authors use DSD and a heuristic variable re-ordering to find an ACD into a structure of 2 or 3 LUTs with non-routable connections. This method limits the shared set to at most one variable. In [133], the authors use ACD in post-mapping resynthesis when logic cones composed of several LUTs are collapsed into single-output Boolean functions and re-expressed using fewer LUTs by DSD and the Shannon expansion.

In Section 3.3, we address the limitations of the previous ACD methods. Our method is based on truth tables and does not have limitations on the number of LUTs and the size of SS. It produces better quality of results and runs up to 80x faster than other more constrained implementations in ABC [133]. Moreover, our ACD does not rely on BDD-related heuristics and is not limited to primitive gates used in DSD, but performs a more complete search.

3.2.2 FPGA Technology Mapping

LUT mapping is the process of expressing a Boolean network in terms of k-input lookup tables (k-LUTs). Before mapping, the network is represented as a *k-bounded* Boolean network called the *subject graph*, which contains nodes with a maximum fan-in size of k. The AIG is the most common subject graph representation. The subject graph is transformed into a mapped network by applying local substitutions to sections of the circuit defined by cuts computed using cut enumeration [47]. A LUT mapper computes a mapping solution, called *cover*, by selecting a subset of the cuts that cover the subject graph while minimizing a cost function. State-of-the-art LUT mappers compute cuts and refine the cover in several mapping passes using heuristics based on delay, area, and edge count. For further details on LUT mapping, we refer the reader to [141].

Related works

State-of-the-art LUT mapping for FPGAs relies on cut enumeration [47] followed by graph covering [45, 141]. Depth-optimal mapping for a k-bounded network is solvable in polynomial time [45], while area-optimal mapping is proven to be NP-hard [59, 117]. However, the structure of the subject graph influences the structure of the mapped network to a large extent. This is known as *structural bias*. Mitigating structural bias is essential to improve the mapping quality.

Several methods derive an LUT network by applying flavors of Boolean decomposition to the BDD of the original function [99, 112, 208]. Despite having a lower structural bias, these approaches are run-time intensive and limited to small functions, for which BDDs can be constructed. In practice, they rarely work well for functions with more than 16 inputs.

To scalably reduce structural bias, previous work adopted different techniques. In [37, 114], structural bias is reduced by accumulating *structural choices* for the subject graph and using them during mapping. In [41, 60, 165], decomposition into k-LUTs is performed during technology mapping. In particular, the Chortle mapper [60] uses a structural decomposition based on bin-packing techniques to map logic into LUTs wherever the associative property holds. The method in [165] integrates Boolean decomposition, based on an heuristic ACD algorithm, into k-LUT mapping to map logic into non-routable LUT structures composed of 2 or 3 LUTs. The approach extracts combinational logic cones with more than k inputs and decomposes them on the fly.

Often, reducing structural bias during technology mapping is not enough to achieve good quality of results. For instance, in [121], the authors perform a flow consisting of several iterations of remapping and support-reducing decomposition to reduce structural bias.

In Sections 3.4 and 3.5, we perform on-the-fly decomposition similar to [165] but with two main differences. First, we utilize a more flexible and expressive ACD formulation. Second, our method can be customized for delay minimization.

3.3 Boolean Decomposition into LUTs

This section discusses a fast and versatile truth-table-based implementation of ACD with shared set for single-output functions. We propose several enhancements that make ACD readily applicable in LUT mappers and resynthesis methods. Figure 3.2 illustrates the ACD computation. The BS, SS, FS, and the number of BS functions used are flexible and determined during the decomposition. The composition function (L_1) is implemented as a multiplexer controlled by the outputs of the BS functions and the shared set. The FS functions, FS (g_i), drive the data inputs of the multiplexer. These functions become part of the composition function.

In this section, we first review the properties of the proposed ACD, showing that it is as generic as the original definition in [12, 49, 170] (Section 3.3.1). Second, we show how to efficiently check the existence of a feasible ACD and divide variables into three sets: FS, BS, and SS (Section 3.3.2). Third, we show how to compute the decomposition while minimizing the number of BS functions and their support (Section 3.3.3). Fourth, we discuss an alternative method to maximize the number of variables in the shared set (Section 3.3.4). Fifth, we present an efficient algorithm for decomposing functions into two LUTs (Section 3.3.5) and a cascade of LUTs (Section 3.3.6). Finally, we present the experimental results of our ACD formulation and algorithms. We show that the proposed ACD method has a higher decomposition success ratio, up to 32.58% more than state-of-the-art, and a better or competitive run time while being more generic.

3.3.1 Theory

First, we formalize the definition of ACD and discuss its properties. Given the ACD shown in Figure 3.2 and the disjoint sets of variables $\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}$, we name

$$\vec{h}(\vec{x}_{bs}, \vec{x}_{ss}) = (h_0(\vec{x}_{bs}, \vec{x}_{ss}), \ldots, h_{v-1}(\vec{x}_{bs}, \vec{x}_{ss})) \tag{3.3}$$

the set of bound set functions of size $|\vec{h}| = v$. In Figure 3.2, $\vec{h}$ has size $v = 1$ and is represented by L_2. In Figure 3.1, $\vec{h}$ has size $v = 2$ and is represented by L_2 and L_3. An ACD can be expressed by Equation 3.1. In Figure 3.2, L_1 implements function g as a multiplexer with M select lines connected to functions in $\vec{h}$ and variables in $\vec{x}_{ss}$, such that $M = v + |\vec{x}_{ss}|$. An input assignment

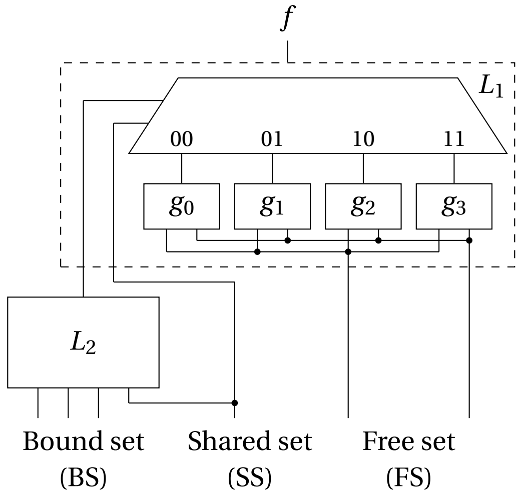


Figure 3.2: Illustrating the AC decomposition of a Boolean function

to the select lines of g selects a function $g_i(\vec{x}_{fs})$ where $0 \leq i < 2^M$.

We demonstrate that our ACD decomposition is generic and includes other formulations, such as the Shannon decomposition. Let us represent the function g as a ROBDD ordered with variables $\vec{h}$ and $\vec{x}_{ss}$ located close to the root, while variables $\vec{x}_{fs}$ are found close to the leaves. Let us draw a cut line in the ROBDD, such that nodes are partitioned into two disjoint sections: one dependent on $\vec{h} \cup \vec{x}_{ss}$ variables (denoted by α), and one dependent on $\vec{x}_{fs}$ variables (denoted by β). In our decomposition, α is implemented by the multiplexer of g, and β is implemented by the FS functions g_i. In particular, the number of nodes in β at the interface of the cut is equivalent to the number of unique g_i functions. Notably, we can extract β by drawing a cut in the ROBDD of f, with $\vec{x}_{fs}$ variables close to the leaves, separating $\vec{x}_{fs}$ from $\vec{x}_{bs} \cup \vec{x}_{ss}$ [99, 142].

Example 3.3.1. *Figure 3.3 shows the BDD of a decomposable* 6*-input function f and a partition into bound set and free set variables when targeting* 4*-input LUTs. The bound set contains variables x_0, x_1, x_2, and x_3, while the free set contains variables x_4 and x_5. The partition draws a cut, in blue, that divides the BDD into the β and γ sections. The unique cofactors that are reachable from the cut in β are $g_0 = 0$ and $g_1 = x_4 \oplus x_5$. Hence, the multiplexer needs $M = 1$ select line. The bound set function and the multiplexer are extracted from γ using the methods in Section 3.3.3.* ▲

It follows that g implements a partitioned BDD. Hence, our ACD formulation can implement any decomposable function. Moreover, the Shannon expansion (Equation 3.2) where x is a control input of the multiplexer, can be represented by ACD as follows:

$$f = f_x f_{\bar{x}} 1 + f_x \bar{f}_{\bar{x}} x + \bar{f}_x f_{\bar{x}} \bar{x} + \bar{f}_x \bar{f}_{\bar{x}} 0,$$

where x is a FS variable, f_x and $f_{\bar{x}}$ are BS functions, and FS functions g_i are 1, x, $\bar{x}$, and 0.

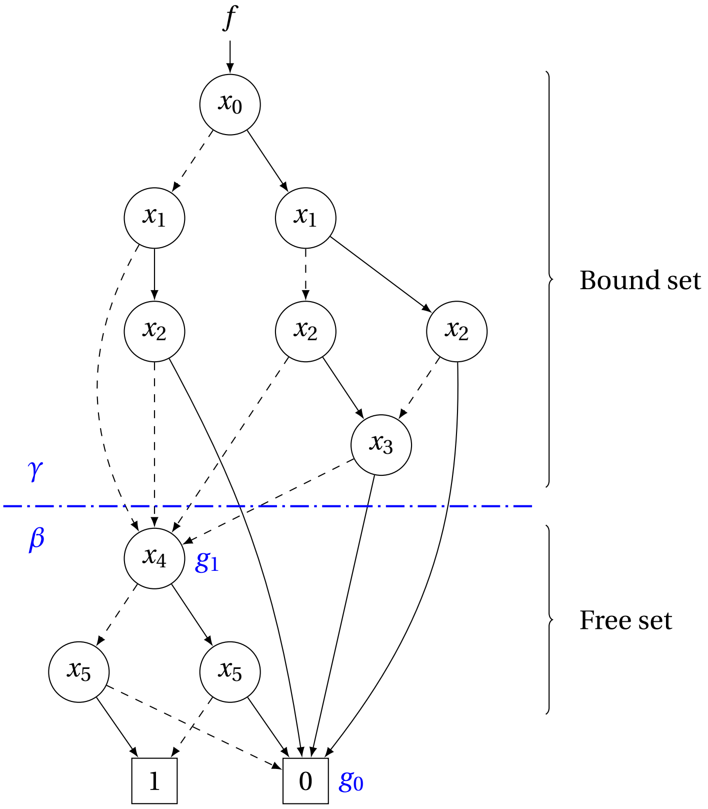


Figure 3.3: ACD decomposition over a BDD with a partition of the variables into free set and bound set.

Definition 1: Variables in the SS that are not used by BS functions are called *independent shared set variables* (ISS variables). Conversely, those that are used by BS functions are defined as *dependent shared set variables* (DSS variables).

According to the ACD definition in Equation 3.1, ISS variables would belong to the FS rather than the SS, since they are not in the support of functions in $\vec{h}$. However, in our decomposition, ISS variables serve as controls for a multiplexer, while the FS variables provide support for the FS functions, which feed into the data inputs for the multiplexer. We demonstrate that our definition is equivalent to the original one, i.e., if a decomposition with ISS variables in the SS exists, it also exists with ISS variables in the FS.

Theorem 3.3.2. *Let $\vec{x}_{ss} = \vec{x}_{iss} \cup \vec{x}_{dss}$ be an SS defined as the union of two disjoint sets: one of independent ($\vec{x}_{iss}$) and one not independent ($\vec{x}_{dss}$) SS variables. Then, $f(\vec{x}_{bs}, \vec{x}_{ss}, \vec{x}_{fs}) = g(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{iss} \cup \vec{x}_{dss}, \vec{x}_{fs})$ can be written as $g'(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{dss}, \vec{x}_{fs} \cup \vec{x}_{iss})$.*

Proof. Let us suppose that $\vec{x}_{iss}$ contains a single variable a. Function g is implemented as a multiplexer of M select lines connected to $\vec{h}$, $\vec{x}_{dss}$, and a, and 2^M data inputs functions $\{g_0, \cdots, g_{2^M-1}\}$. Then, each cofactor of g with respect to $\vec{h} \cup \vec{x}_{dss}$ variables is a function in the form $\dot{g}(a, \vec{x}_{fs}) = a \cdot g_i(\vec{x}_{fs}) + \bar{a} \cdot g_j(\vec{x}_{fs})$ with $0 \leq i < j \leq 2^M - 1$. Since the number of $\dot{g}$ cofactors cannot be larger than 2^{M-1}, f can be decomposed into the form $f = g'(\vec{h}(\vec{x}_{bs}, \vec{x}_{dss}), \vec{x}_{dss}, \vec{x}_{fs} \cup \{a\})$ with variable a in the free set. The generic case is proved by induction. ■

Finally, we state a theorem used in Section 3.4 to conduct the search for a feasible decomposition.

Theorem 3.3.3. *If a decomposition of function f into two levels of k-LUTs with P variables in the free set does not exist, f cannot be decomposed with $P' > P$ variables in the free set.*

Proof. Let us suppose that a decomposition exists for $P' > P$ and does not exist for P. The decomposition with P' involves at most $k - P' + 1 < k - P + 1$ LUTs. This is a contradiction of the principles of information theory since a decomposition using P' has less information encoding than the one using P. ■

3.3.2 Finding a Feasible Variable Partition

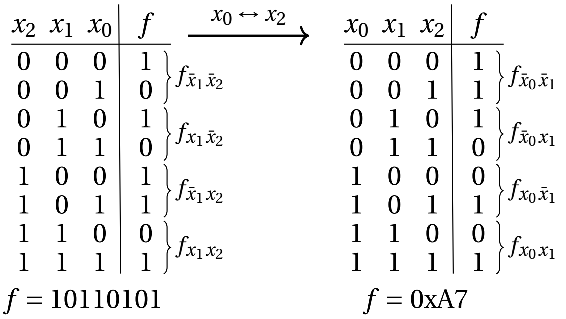
After defining the properties of ACD, in this section we present an efficient method to check the existence of a Boolean decomposition and find an assignment of support variables to the FS and the BS (and SS). In particular, we focus on decomposition into a two-level k-input LUT structure. For simplicity, in this section, we include the SS variables in the BS.

The first step to derive a decomposition is to partition variables into FS and BS. Given a truth table, our approach enumerates different free sets. Let N be the number of variables in the support of the function to decompose. Let P be the number of variables to consider in the FS. The remaining $N - P$ variables are considered in the BS. The number of different free sets is $\binom{N}{P}$. Regarding the choice of value P when searching for a feasible two-level decomposition, for an N-input function and k-input LUTs, it is convenient to consider $(N - k)$ variables in the FS, so that at most k variables are considered in the BS.

Example 3.3.4. *When $N = 8$ and $k = 6$, we can choose $P = 2$ to make $N - P$ fit in a k-LUT and evaluate $8 \cdot 7/2 = 28$ different 2-variable free sets.* ▲

For each FS, the truth table is transformed to have the FS variables as the least significant ones. The variable reordering is performed using a dedicated procedure, which swaps two variables at a time. Note that when enumerating all the free sets, the first FS composed of the P least significant variables in the support of the function does not need variable swapping, since the original truth table already reflects this order. Then, every consecutive FS can be derived from a previous FS by swapping one variable in $\vec{x}_{fs}$ with one in $\vec{x}_{bs}$. The complexity to explore all the FS is of $\binom{N}{P}$ swap operations. Figure 3.4 shows how a variable swap affects the truth table.

Each input assignment to the BS variables selects one P-input function in terms of the FS variables. Specifically, each P-input function is a cofactor with respect to variables in $\vec{x}_{bs}$.



x_2	x_1	x_0	f	
0	0	0	1	$f_{\bar{x}_1\bar{x}_2}$
0	0	1	0	
0	1	0	1	$f_{x_1\bar{x}_2}$
0	1	1	0	
1	0	0	1	$f_{\bar{x}_1 x_2}$
1	0	1	1	
1	1	0	0	$f_{x_1 x_2}$
1	1	1	1	

$f = 10110101$

$\xrightarrow{x_0 \leftrightarrow x_2}$

x_0	x_1	x_2	f	
0	0	0	1	$f_{\bar{x}_0\bar{x}_1}$
0	0	1	1	
0	1	0	1	$f_{\bar{x}_0 x_1}$
0	1	1	0	
1	0	0	0	$f_{x_0\bar{x}_1}$
1	0	1	1	
1	1	0	0	$f_{x_0 x_1}$
1	1	1	1	

$f = $ 0xA7

Figure 3.4: Truth table representation in the classical tabular form and as bit string (binary and hexadecimal), cofactor extraction w.r.t. the two most significant variables, and variable swapping of x_0 with x_2.

Given a truth table with this variable ordering, FS functions are easily computed by extracting groups of 2^P bits at $i \cdot 2^P$ offsets with $i \in [0, 2^{(N-P)})$. Informally, FS functions are bit-strings positioned next to each other in the bit-string of the truth tables. Figure 3.4 graphically depicts the extraction of cofactors with respect to the two most significant variables.

Example 3.3.5. *Consider a* 6*-variable function represented in hexadecimal as the truth table* $f =$ 0x8804800184148111. *Assume that the FS variables are the two least significant variables and the BS variables are the four most significant ones. The functions in terms of FS variables have truth tables with* $2^P = 4$ *bits. There are* $2^{(N-P)} = 16$ *of these functions, corresponding to hexadecimal digits in the truth table (*0x8, 0x8, 0x0, 0x4*, etc).* ▲

The target function can be realized using M bound set functions if the number of unique FS functions, known as column multiplicity μ, does not exceed 2^M, hence $M \geq \lceil \log_2(\mu) \rceil$. If $P + M \leq k$, the composition function fits into one k-LUT.

Example 3.3.6. *Continuing Example 3.3.5, there are* 16 *FS functions, of which only* 4 *are unique. The FS functions are* 0x8, 0x0, 0x4, *and* 0x1. *Hence, the column multiplicity* $\mu = 4$*, which requires* $M = \lceil \log_2(4) \rceil = 2$ *or more BS functions. Hence, this partition of variables into FS and BS produces a feasible support-reducing decomposition into* 4*-input LUTs. Using Figure 3.2, ACD assigns FS functions to* g_i*. Then, two BS functions of at most* 4 *inputs are necessary to select the correct FS function.* ▲

Example 3.3.7. *To illustrate how partitioning variables into FS and BS affects the truth table, consider* $f' =$ 0xA010800190148105. *This function is derived by swapping one FS variable with one BS variable from function* f *in Example 3.3.5. In this case, for* $P = 2$*,* $\mu = 7$*, requiring* $M = \lceil \log_2(7) \rceil = 3$ *or more BS functions. Therefore, this partition of variables into FS and BS does not produce a feasible support-reducing decomposition into* 4*-input LUTs.* ▲

We employ the enumeration of free sets while counting the number of unique cofactors to check if a support-reducing decomposition exists. In practice, a sufficient condition for a 2-level decomposition to exist, is to have $M + P \leq k$ and $N - P \leq k$, i.e., the composition function is k-feasible, and the number of remaining variables in the BS does not exceed k. However, a decomposition could have $N - P > k$ and k-feasible BS functions, as shown in Figure 3.1. In this case, it is not sufficient to partition variables into FS and BS to guarantee a 2-level decomposition (unless there are ISS variables that can be moved to the FS, by Theorem 3.3.2, to make $N - P \leq k$ true). Consequently, each potential decomposition with $N - P > k$ and $P + M \leq k$, similar to the one in Figure 3.1, must be checked to be 2-level decomposable by computing minimal-support BS functions as shown in Section 3.3.3. Due to this additional computation, the latter ACD is often too slow to be used in mainstream LUT mappers or resynthesis engines.

After partitioning variables into FS and BS and computing the corresponding unique FS functions, our method uses the techniques in Section 3.3.3 to produce a decomposition while minimizing the number of BS functions and their support.

3.3.3 Functional Encoding and Support Minimization

Once a partition of variables into FS and BS with a feasible decomposition is found, the BS functions are extracted by assigning each FS function a code. Informally, an encoding represents the assignment of FS functions to the data inputs of the MUX of Figure 3.2 (e.g., the encoding of g_1 is 01). While any encoding that distinguishes FS functions is a valid solution, a good encoding also minimizes the number of BS functions and their support. It is crucial to find an encoding that minimizes the support for three reasons. First, if $N - P > k$, by minimizing the support, each BS function may ideally fit into a k-LUT, allowing for a two-level decomposition. Second, minimizing the support maximizes the shared set (an SS variable is a BS function represented by a buffer), reducing the number of required LUTs. Third, the number of edges is reduced, helping routability. Finding a feasible encoding is similar to solving constrained encoding problems [51, 209, 215].

An encoding assigns a code $T = t_{M-1} \dots t_0$ of length M to each of the FS function. A variable t_i takes value 1, 0, or $-$, indicating the ON-set, OFF-set, and DC-set, respectively. A *minimum-length encoding* is an encoding of length $M = \lceil \log_2(\mu) \rceil$. An encoding is *strict* if a unique term T is assigned to each FS function, resulting in a Boolean function. An encoding is *non-strict* if multiple pair-wise disjoint terms T can be assigned to each FS function, resulting in a Boolean relation. For instance, given $M = 2$ and $\mu = 3$, an assignment "1$-$" to a FS function is strict, while "01 $\vee$ 10" is non-strict. In this work, we only utilize strict encodings since non-strict encodings are too many to be efficiently enumerated in a fast ACD implementation. Moreover, experimental evaluations on practical functions suggest that non-strict encodings do not improve the quality of the decomposition. For further details on the number of possible encodings, refer to [142].

Let *i-sets* be the set of μ Boolean functions in terms of the BS variables encoding FS functions using one-hot encoding. Specifically, an i-set represents one FS function and takes value 1 when an input assignment to the BS variables selects the corresponding FS function.

Example 3.3.8. *Continuing Example 3.3.6, the i-set corresponding to the FS function* 0x8 *is* 1100100010001000 *in binary format. Note that the truth table depends on $N - P$ variables and has value* 1 *when the original function is* 0x8 *or* 0 *otherwise.* ▲

I-sets are used to derive a more compact encoding with a two-step procedure. The first step enumerates *candidate BS functions*. The second one solves a unate covering problem, in which columns are candidate BS functions and rows are pairs of FS functions to be distinguished.

Candidate BS functions are a set of functions depending on BS variables used as t_i signals encoding FSs. They are enumerated by combining i-sets. To leverage all the functional degrees of freedom of a strict encoding, i-sets in a BS candidate can be either in the *ON-set*, *OFF-set*, or *don't-care* (DC) set. Since candidate BS functions are used as control inputs of a multiplexer, they can distinguish elements in the ON-set (takes value 1) against elements in the OFF-set (takes value 0). In encoding problems, BS functions are called *dichotomies*, while pairs of functions to be distinguished can be interpreted as *seed dichotomies* [215]. Don't-cares are also important to minimize the support, which translates into fewer LUT fan-ins.

Example 3.3.9. *Continuing Example 3.3.8, let us consider the candidate bound set function h that has the i-sets* {0x8, 0x1} *in the ON-set, the i-set* {0x4} *in the OFF-set, and the i-set* {0x0} *in the DC-set. Its function in the binary format is h =*11-01--110101111 *where "-" is a don't care. When $h = 1$, either* 0x8 *or* 0x1 *are selected. When $h = 0$,* 0x4 *is selected. The corresponding dichotomy is* {{0x8, 0x1},{0x4}}. *In this case, function h distinguishes* 0x8 *from* 0x4 *and* 0x1 *from* 0x4*, covering two seed dichotomies* {{0x8},{0x4}} *(or* {{0x4},{0x8}}*) and* {{0x1},{0x4}} *(or* {{0x4},{0x1}}*).* ▲

A candidate BS function is generated by assigning each i-set to the ON-set, OFF-set, or DC-set. Hence, the total number of possible BS candidate functions is 3^{μ}. Nonetheless, this bound can be greatly reduced. Some BS candidate functions are interchangeable, i.e., one candidate can be obtained by swapping the ON-set and the OFF-set of another candidate. Our enumeration removes these symmetries. Moreover, in a minimum-length encoding, each candidate must have at least r i-sets in the ON-set and r i-sets in the OFF-set, where r is defined as:

$$r(\mu) = \mu - 2^{\lfloor \log_2(\mu-1) \rfloor}. \tag{3.4}$$

Candidates that do not satisfy this constraint are eliminated as they cannot encode the FS functions. For instance, if μ is a power of 2, then $r = \mu/2$, implying that the FS functions must be evenly distributed between ON-set and OFF-set, i.e., each candidate must distinguish half of the FS functions against the other half. In a non-minimum encoding, $r(\mu) = 1$, such that each BS candidate function has some distinguishing power. The number of possible BS candidate functions is given by the following formula depending on μ:

$$\mathscr{E}(\mu) = \frac{1}{2} \cdot \sum_{i=0}^{\mu-2r(\mu)} \left[\binom{\mu}{i} \cdot \sum_{j=r(\mu)}^{\mu-i-r(\mu)} \binom{\mu-i}{j} \right]. \tag{3.5}$$

Note that when μ is a power of 2, the number of possible BS candidate functions reduces to

Table 3.1: Comparison on decomposing 8-input practical functions using different ACD settings for the encoding problem.

ACD mode	Success rate (%)	#LUTs	#Edges	Time (s)
No encoding (only partitioning)	100	277411	1399908	1.61
No don't cares	100	277007	1331527	7.10
Default (DCs for $\mu < 8$)	100	268954	1220771	7.15
Always don't cares	100	268954	1220437	82.48

$\binom{\mu}{\mu/2}/2$.

A limitation of this method is that the number of candidates grows rapidly with increasing column multiplicity. However, we may further reduce the number of BS candidate functions when it is too large. In particular, for an ACD into 6-LUTs the maximum column multiplicity to support is 16. In fact, for $\mu > 16$ the function would require at least 5 BS functions, remaining with one variable in the free set. But the number of unique functions that can be realized using one variable is $2^2 = 4$. Hence, the maximum column multiplicity for a 1 FS variable is $\mu = 4$. Equation 3.5 is maximized for $\mu = 13$ with 91,377 candidate BS functions. To maintain a reasonable number of candidates and to significantly reduce run time, our method does not use the DC-set for problems with $\mu > 8$, lowering the maximum number of candidates to 6,435. This simplification removes the leftmost sum and fixes $i = 0$ in Equation 3.5, resulting in:

$$\mathscr{E}'(\mu) = \frac{1}{2} \cdot \sum_{j=r(\mu)}^{\mu-r(\mu)} \binom{\mu}{j}. \tag{3.6}$$

Experiments show that this restriction tends to improve run time without significantly impacting the encoding quality, but using it for lower values of column multiplicity noticeably compromises the quality.

To support this choice, we evaluated the decomposition quality for various configurations of ACD on decomposing 107466 8-input practical functions extracted from the EPFL benchmark suite [3]. Generally, 8-input practical functions have a quite simple variable partitioning problem but complex encoding since μ can reach value 16. The results are available in Table 3.1, which shows the success rate, the total number of LUTs used, the total number of edges, and the time to compute the decomposition. Mode *no encoding* shows the number of LUTs without solving the encoding problem. Mode *no don't cares* assigns i-sets only to the ON-set or the OFF-set during the encoding. Mode *default* uses the DC-set for $\mu < 8$. Lastly, mode *always don't care* always uses don't care assignments. Table 3.1 shows that the default mode has the best compromise between quality and run time. Mode *always don't care* is slower for a limited improvement in the number of edges. Moreover, the *always don't care* mode obtains exactly the same number of LUTs of *default* in practical 8-input functions, showing that don't cares are important for small μ values but not so much for higher values.

Each BS candidate function is associated with a cost that depends on the number of

	4	3	3
	C9AF	1177	2727
{{0x8}, {0x0}}	1	0	1
{{0x8}, {0x4}}	1	1	0
{{0x8}, {0x1}}	0	1	1
{{0x0}, {0x4}}	0	1	1
{{0x0}, {0x1}}	1	1	0
{{0x4}, {0x1}}	1	0	1

Figure 3.5: Covering table to solve the encoding problem.

variables in its support. The number of variables is computed using a special procedure that considers don't cares. Each variable is checked individually. If the incompletely specified BS candidate function remains equivalent when a variable is assigned both constant 0 and constant 1, that variable is not in the functional support and can be removed. Then, a covering table is constructed by having all the pairs of FS functions to be distinguished (seed dichotomies) as rows and the BS candidates as columns. A row-column entry (i, j) is 1 if the BS candidate function of column j distinguishes the seed dichotomy i. A support-minimum solution is computed by solving a minimum-cost covering problem [215]. The solution must cover all the rows while minimizing the cost. We use greedy covering followed by local search to compute a minimum-cost cover. A single iteration of greedy covering extracts one column covering the most non-covered rows while minimizing the cost. The process is iterated until a solution is found. Then, the solution is iteratively improved by replacing one column with another column having a lower cost.

Example 3.3.10. *Figure 3.5 shows a covering table reflecting the examples in this section. Each column is a candidate BS function shown as a truth table in hexadecimal format on* 4 *variables. Each BS candidate function has a cost based on the number of variables on its support (shown in the figure above the BS function). Each row is a seed dichotomy. An element* (i, j) *in the table is* 1 *if the* BS_j *distinguishes the seed dichotomy* i. *The best solution with cost* 6 *takes the second and third columns and leads to two BS functions depending on* 3 *variables.* ▲

Given a solution, an encoding of the FS functions is obtained by assigning a code $T = t_{M-1} \dots t_0$, in which each signals t_i corresponds to a selected BS$_i$ candidate.

Example 3.3.11. *Continuing Example 3.3.10, a minimum cover results in* $BS_0 = 0x1177$, *by putting* 0x4 *and* 0x1 *in the ON-set, and* $BS_1 = 0x2727$ *by putting* 0x0 *and* 0x1 *in the ON-set. Both bound sets depend only on* 3 *variables. Given the BS functions, the encoding of the FS functions assigns the following codes to* g_i *in Figure 3.2:* $T_{0x8} = 00$, $T_{0x4} = 01$, $T_{0x0} = 10$, *and* $T_{0x1} = 11$. *Finally, the composition function is computed using the FS functions and the selected encoding, resulting in function* 0x1048, *in hexadecimal format. Consequently, the function has been successfully decomposed using three* 4*-LUTs. The final result of decomposition is shown in Figure 3.6, after minimizing the support of BS functions.* ▲

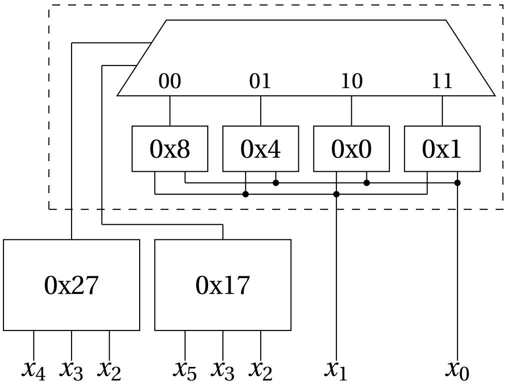


Figure 3.6: AC decomposition of Boolean function 0x880480018414811.

3.3.4 Maximizing the Shared Set

The number of LUTs required to implement the BS functions can be minimized using the shared set. In Section 3.3.3, we presented a generic method to find an encoding that minimizes the LUT count and the support size. Alternatively, to check whether a decomposition with $L \in [0, M)$ single-variable functions (or buffers) and $M - L$ non-buffer BS functions exists, our method may enumerate subsets of L out of $N - P$ variables, with a total of $\binom{N-P}{L}$ subsets. For each subset, the method checks if the number of unique FS functions in each cofactor with respect to L variables does not exceed 2^{M-L}. If this is the case, a decomposition with L variables in the shared set exists.

Example 3.3.12. *Consider the truth table* 0xffff0880ffff0000 *with* $P = 2$ *and unique FS functions* 0xf, 0x0, *and* 0x8. *Let us check the existence for a shared set when* $M = 2$ *using* $L = 1$. *If the most significant variable is in the SS, the truth table is divided into two cofactors* 0xffff0880 *and* 0xffff0000. *The number of unique FS functions in the first cofactor exceeds* $2^{2-1} = 2$. *Hence, the most significant variable cannot be shared. However, the second most significant variable, with cofactors* 0xffffffff *and* 0x08800000, *can be shared.* ▲

3.3.5 Boolean Decomposition into Two LUTs

A decomposition into two LUTs is a special type of ACD with a single BS function and possibly multiple SS variables. Since BS functions are limited to one, the problem has a lower complexity than the generic case. Here we propose a dedicated algorithm to solve this problem more efficiently.

For a truth table on N variables, a "kk" decomposition may exist for $N < 2 \cdot k$. According to Theorems 3.3.2 and 3.3.3, it is sufficient to test the decomposition for $P = N - k$, when allowing for multiple variables in the shared set. Specifically, this is the minimum number of variables to have a k-feasible bound set and a decomposition. Note that a decomposition with $P < N - k$ (or $N - P > k$) may exist only if there are at least y independent variables in the shared set, such that $P + y = N - k$. Since, by Theorem 3.3.2, ISS variables can always be

Algorithm 3.1: ACD into two LUTs

```
   Input: Truth table $tt$, number of variables $N$, LUT size $k$
   Output: Decomposition if it exists
1  $P \leftarrow N - k$
2  $Perm \leftarrow \{0, 1, 2, \ldots, N-1\}$
3  for $\binom{N}{P}$ iterations do
4      $\mu \leftarrow$ compute_multiplicity($tt$, $P$)
5      $L_{min} \leftarrow \lceil \log_2 \mu \rceil - 1$;                ▷ Required variables in SS
6      if $P + L_{min} < k$ then
7          $\vec{x}_{ss} \leftarrow$ compute_shared_set($tt$, $N$, $P$, $k$, $L_{min}$)
8          if $P + |\vec{x}_{ss}| < k$ then
9              return decompose($tt$, $N$, $P$, $k$, $Perm$, $\vec{x}_{ss}$)
10     $tt \leftarrow$ next_combination( $tt$, $N$, $P$, $Perm$ )
11 return not decomposable
```

moved into the free set, and, by Theorem 3.3.3 a smaller free set has more solutions than a larger one, $P = N - k$ is the only necessary FS size to check.

Algorithm 3.1 shows a sequence of steps to perform a decomposition into two LUTs. The algorithm takes as inputs a truth table tt, the number of its support variables N, and the LUT size k. First, P and the permutation vector $Perm$ are initialized. Vector $Perm$ is necessary to track the order of the variables during the enumeration of combinations, compared to the original one, and to compute the next combination. A loop iterates over all the possible P combinations of N. The method *next_combination* (at line 10) computes a new combination from the previous one by swapping one variable in the FS with one in the BS. The returned truth table reflects the new variable order. The column multiplicity μ is computed for the truth table tt (at line 4). If $\mu = 2$, a decomposition exists with one BS function. Since the structure is limited to one BS function, for $\mu > 2$ the method searches for SS variables. First, L_{min} is computed to minimize the number of shared variables. Then, the algorithm searches for a shared set of L elements, employing the techniques of Section 3.3.4. The search for a shared set is performed for $L_{min} \leq L < k - P$, which also allows for non-minimum-length encodings. If a shared set exists, the corresponding decomposition is returned. Otherwise, if the conditions in the for loop are not met, the function is not decomposable into 2 LUTs.

In case of an implementation constraining the maximum number of variables in the SS, Algorithm 3.1 is modified to additionally explore different sizes P, similarly to Algorithm 3.3. This is because Theorem 3.3.3 is not valid when limiting the maximum number of BS functions and SS variables because it constraints the maximum value of encoding M. Hence, when $\lceil \log_2(\mu) \rceil > M_{max}$ there might be ISS variables to include in the FS to make $\lceil \log_2(\mu') \rceil \leq M_{max}$.

3.3.6 Boolean Decomposition Beyond 2 Levels

Previous subsections discuss methods for partitioning variables into bound and free sets, solving the functional encoding problem, and extracting the shared set. These algorithms primarily address the case when Boolean functions can be decomposed into two levels of logic. This subsection provides a preliminary exploration of methods to perform ACD to multiple logic levels. Specifically, we first propose a recursive formulation. Then, we discuss possible improvements for scalability based on BDDs.

In the cases when Boolean function cannot be decomposed into two levels of LUTs, a recursive approach may be employed. When decomposing a function into k-input LUTs, a bound set of size exceeding k may help to simplify the function by removing the dependency on some variables for another round of ACD. In practice, multi-level ACD performs a recursive support-reducing decomposition until a feasible k-feasible bound set with a solution exists. In this subsection, we propose a method that extends Algorithm 3.1 for multi-level ACD, resulting in a cascade structure of LUTs. This approach is heuristic and not exact, as it selects only one feasible support-reducing decomposition among the possible options.

Algorithm 3.2 illustrates a possible implementation of the ACD into k-input LUTs resulting in a cascade structure. Initially, procedure `ACD` takes the truth table of the function to be decomposed, the number of variables N, and the LUT size k. The first step attempts a decomposition into 2 levels of LUTs using Algorithm 3.1 (lines 3 to 6). In this case, a solution exists only if $N < 2 \cdot k$. If this decomposition is feasible, it is returned, otherwise, the algorithm tries a generic support-reducing step. The support-reducing step decomposes the function by having more than k variables in the bound set. To maximize the number of variables in the free set, as a way of heuristically simplifying the bound set function, the search starts with $i = k - 1$ variables in the FS and decreases to $i = 1$ (lines 8 to 12). Algorithm 3.1 is employed again to find a solution with an $(N - i)$-variable BS (line 9). If a solution exists, the remainder BS function is further decomposed by recursively calling function `ACD`. If no solution is found, the algorithm terminates without a successful decomposition.

Algorithm 3.2 demonstrates how the algorithms presented in this chapter can be used to decompose functions into multiple levels of LUTs. However, the proposed method has high computational complexity and may suffer from long run times, making it impractical in large designs, even when the method operates on functions of 16 variables (the upper limit for a truth table implementation). Developing more efficient algorithms is an area for future work.

One potential way to improve the scalability of the method is by using BDDs. Many implementations of Boolean decomposition were based on BDDs since they are easy to manipulate and they can be constructed for most of the practical function up to 50 input variables. In this chapter, we presented more efficient and powerful algorithms to compute ACD based on truth tables. However, these method are often impractical when the truth table depends on many variables (> 16) or when the decomposition requires multiple levels of logic. In such cases, a BDD-based implementation may offer advantages. For instance,

Algorithm 3.2: Recursive ACD cascade into multiple logic levels

```
Input: Truth table $tt$, number of variables $N$, LUT size $k$
Output: Decomposition if it exists
Algorithm ACD($tt$, $N$, $k$)
    $D \leftarrow \Lambda$
    if $N < 2 \cdot k$ then
        $D \leftarrow$ acd_two_luts($tt$, $N$, $k$)                ▷ Run Algorithm 3.1
        if $D$ is feasible then
            return $D$
    $P \leftarrow \infty$
    foreach Free set size $i$ from $k-1$ to $1$ do
        $D \leftarrow$ acd_two_luts($tt$, $N$, $N-i$)              ▷ Run Algorithm 3.1
        if $D$ is feasible then
            $P \leftarrow$ free_set_size($D$)
            break
    if $P = \infty$ then
        return not decomposable
    $R \leftarrow$ get_reminder_function($D$)
    $R_d \leftarrow$ ACD($R$, $N-P$, $k$)
    if $R_d$ is decomposable then
        return compose_decomposition($D$, $R_d$)
    return not decomposable
```

if a decomposition into a cascade of k-LUTs exists, the BDD should be “slim” so that it is possible to draw variable partitions that satisfy the ACD conditions (as a proxy for low column multiplicity). Instead of evaluating every possible variable ordering, ACD can leverage BDD heuristic re-ordering to minimize the number of BDD nodes and use a filter on the relationship between the number of variables and the number of BDD nodes to estimate the feasibility of the decomposition. If the input delay profile is available, it can also be used to fix the ordering of the variables. This approach can improve the practicality and efficiency of multiple-level ACD.

3.3.7 Experimental Results

This section presents an experimental evaluation of the proposed Ashenhurst-Curtis Decomposition (ACD) algorithms. We evaluate the performance of our ACD methods in decomposing functions by comparing them against other implementations of Boolean decomposition in ABC. Specifically, we test the number of functions that can be successfully decomposed and the run time needed. We run this experiment on *practical functions*, i.e., functions collected in hardware designs and benchmark suits, which include fully-decomposable, partially-decomposable, and non-decomposable functions over the bases AND, XOR, and MUX. A set of N-input practical functions tends to be much smaller than a set containing all N-input functions since designs are never completely random. We extract practical functions from the EPFL benchmarks [3] by recording all the functions encountered during cut enumeration in a technology mapper. Since the number of practical functions can be large, we classify them into $\mathscr{NPN}$-equivalence classes employing the heuristic sifting algorithm [82, 182]. Note that if a function in an $\mathscr{NPN}$-equivalence class is decomposable, all the functions in the same class are also decomposable (the $\mathscr{NPN}$ transformation can be applied after decomposition).

Decomposition success rate

Table 3.2 and Table 3.3 show the percentage of decomposable functions and the run time for different methods and support sizes. For instance, the first column of Table 3.2 contains results for decomposing practical 5-input functions, where (1233) indicates the number of unique functions collected after computing $\mathscr{NPN}$ canonical forms. Each row of the tables show one ACD method. Row DSD computes a decomposition graph of the function into AND and XOR primitives, with possible inverters. Then, the graph is mapped to 4-input (in Table 3.2) or 6-input (in Table 3.3) LUTs using a structural LUT mapper. The decomposition graph employs top and bottom disjoint-support decomposition (see, e.g., Section 4.3.1 in Chapter 4 or [25, 36]) and Shannon decomposition, when the function is not fully-DSD-decomposable. The DSD method is used to evaluate the ability of classical AIG and XAG logic synthesis methods to find a minimum-size implementation. Row S44 presents the state-of-the-art method in [165] to decompose into a LUT structure composed of two 4-LUTs. Similarly, S66 performs the same approach for 6-LUTs. Note that S44 and S66 support no more than one variable in the shared set. The next approach *lutpack* [133][1] performs a heuristic ACD using DSD and the Shannon expansion, supporting up to 3 variables in the shared set. Then, we present two variants of the ACD method of Section 3.3.5 to find decompositions into LUT structures composed of two 4-input (in Table 3.2) or 6-input (in Table 3.3) LUTs, denoted by J44 and J66, respectively. The 1-SS version uses up to one variable in the SS to better compare against S44 and S66. Meanwhile, the M-SS version has no restrictions on the number of SS variables. Additionally, Table 3.2 shows the results of a SAT-based exact synthesis formulation [69] to compute the minimum-size decompositions. Note that this latter method is restricted to 4-input LUTs due to its poor scalability.

Table 3.2 shows that the approaches described in this paper significantly outperform all the previous state-of-the-art methods while achieving proven optimum results in decomposition success. In particular, J44 1-SS has a significantly better success rate in all columns and better run time, compared to S44. This underlines the limitation of the heuristics used in S44. Moreover, the experiment shows that heuristics take more run time than our exact formulation. J44 M-SS further improves the results for 5-input functions while maintaining dominant run time. Note that J44 M-SS matches the success rate of SAT while being up to 7661 times faster.

[1] We modified lutpack in ABC to perform only the decomposition required by the experiment without the overhead of the resynthesis engine.

Table 3.2: Decomposition success ratio into two 4-LUTs for practical functions using different ACD methods.

ACD type	**5 vars** (1233)		**6 vars** (7351)		**7 vars** (41071)	
	Success (%)	Time (s)	Success (%)	Time (s)	Success (%)	Time (s)
DSD-44	55.31%	0.25	23.30%	1.81	16.52%	11.8
S44 [165]	83.94%	0.01	56.09%	0.05	16.36%	0.27
lutpack [133]	91.08%	0.34	45.65%	2.11	18.70%	12.72
J44 1-SS	88.00%	0.00	64.22%	0.02	20.86%	0.10
J44 M-SS	96.67%	0.00	64.22%	0.02	20.86%	0.10
SAT	96.67%	1.47	64.22%	34.98	20.86%	766.11

Table 3.3: Decomposition success ratio into two 6-LUTs for practical functions using different ACD methods.

ACD type	**7 vars** (41071)		**8 vars** (107466)		**9 vars** (195602)		**10 vars** (313649)		**11 vars** (404991)	
	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)	Success (%)	Time(s)
DSD-66	85.52%	12.86	54.12%	48.35	42.16%	104.32	32.72%	214.86	21.95%	373.63
S66 [165]	84.18%	0.60	69.24%	2.57	52.13%	4.99	37.36%	6.99	19.14%	9.79
lutpack [133]	98.34%	20.39	83.47%	64.37	69.92%	154.38	48.95%	334.79	26.87%	897.55
J66 1-SS	97.30%	0.28	82.23%	1.41	74.24%	4.20	63.06%	9.39	32.88%	16.43
J66 M-SS	99.82%	0.30	92.94%	3.08	84.71%	9.92	63.06%	9.73	32.88%	16.58

Table 3.3 further shows that the approaches described in this paper outperform state-of-the-art. This experiment is even more significant than the previous one due to the additional complexity of the problem. In particular, J66 1-SS has a significantly better success rate in all columns and better run time up to 9-input functions, compared to S66. Notably, while searching for a decomposition with the same characteristics, J66 1-SS always finds a solution if it exists (under the 1-SS limitation), while S66 does not always find it because it uses heuristics. This leads to an improvement in success rate that peaks at 25.7%. This table shows the potential of the methods proposed in this work, which can outperform state-of-the-art in quality and run time. J66 M-SS further improves the results for functions between 7 and 9 inputs, with an improvement that peaks at 32.58%, compared to S66. Regarding the run time, while Table 3.3 shows that S66 is generally faster than J66, J66 is, on average, faster for decomposable functions and considerably slower for non-decomposable ones. In fact, J66 enumerates all the possible free sets to find a solution if it exists, while S66 limits the exploration to a smaller subspace.

Decomposition success rate for delay optimization

We extend the previous experiment for 6-input LUTs to evaluate delay minimization using the proposed ACD methods. This experiment tests the success rate of a delay-minimal decomposition for practical functions given delay-critical variables required to be in the free set. Informally, for delay-critical variables with delay D, this experiment checks the existence of a decomposition with delay $D+1$. The other variables are considered to have delay $D-1$. We only consider J66 M-SS and generic ACD, which integrates the search for a feasible free set (in

Table 3.4: Decomposition success ratio into 2 levels of 6-LUTs for practical functions given late arriving variables.

N late	ACD type	7 vars	8 vars	9 vars	10 vars	11 vars
0	lutpack [133]	99.01%	88.25%	84.65%	75.25%	26.87%
	J66 M-SS	99.82%	92.94%	84.71%	63.06%	32.88%
	Generic	100.00%	100.00%	98.05%	90.20%	32.88%
1	J66 M-SS	96.59%	79.60%	61.51%	37.35%	16.54%
	Generic	100.00%	100.00%	97.57%	83.23%	16.54%
2	J66 M-SS	86.22%	59.78%	39.28%	23.74%	10.95%
	Generic	100.00%	100.00%	94.19%	66.56%	10.95%
3	J66 M-SS	65.11%	36.37%	21.25%	13.78%	6.96%
	Generic	93.78%	86.03%	76.82%	44.51%	6.96%
4	J66 M-SS	36.96%	17.00%	8.62%	7.21%	4.43%
	Generic	54.55%	40.42%	25.45%	23.70%	4.43%
5	J66 M-SS	14.52%	5.42%	2.96%	2.84%	2.61%
	Generic	14.52%	5.42%	2.96%	2.84%	2.61%

Section 3.3.2) and the generic functional encoding problem (in Section 3.3.3) to compute the decomposition, since other known methods do not perform delay minimization using input arrival times. We show *lutpack* [133] only for the first row to perform a 2-level decomposition, without limiting the number of LUTs. For each function, we randomly generate up to 10 unique sets of delay-critical variables and test the decomposition for each one of them.

Table 3.4 shows the success rate of decomposing practical functions based on the number of delay-critical variables, shown in column “N late”. Generic ACD has a high success rate in most cases. Limitations occur when the number of delay-critical variables exceeds 3 or the number of variables in the support is 10 or more. Generally, the decomposition of 11-input functions is rare. However, many 10 input functions are still decomposable. Furthermore, the table highlights the advantages of using multiple BS functions, with a success rate difference between J66 and generic that peaks at 55.57% for 9-input functions, given 3 delay-critical variables. Thus, in this case, it is 55% more likely to find a solution to a delay-driven decomposition problem if we consider the most general two-level ACD formulation, compared to the case when only J66 is used.

Decomposition success rate on multiple levels

In this experiment, we evaluate the recursive ACD algorithm, shown in Algorithm 3.2, to find decompositions of Boolean functions into at a cascade of LUTs of at most three levels (3 LUTs in 3 levels). We use the same practical functions of previous experiments, but up to 15 inputs, to map into 6-input LUTs. For this experiment, we use up to one variable in the shared-set. Similarly to Table 3.3, we report the success rate for practical functions depending on a variable number of inputs.

Table 3.5: Decomposition success ratio into two or three levels of 6-LUTs for practical functions using our ACD methods.

Num Vars.	**J66 1-SS**		**J666 1-SS**	
	Success (%)	Time (s)	Success (%)	Time (s)
7 (39964)	97.30%	0.28	97.30%	0.37
8 (92897)	82.23%	1.41	86.44%	1.82
9 (161642)	74.24%	4.20	82.64%	6.83
10 (240531)	63.06%	9.39	76.69%	19.88
11 (290166)	32.88%	16.43	71.65%	57.64
12 (305480)	-	-	65.40%	154.75
13 (283457)	-	-	53.84%	328.30
14 (248555)	-	-	43.31%	697.80
15 (169998)	-	-	28.70%	1522.06

Table 3.5 shows the results comparing ACD into two levels of 6-LUTs to the recursive ACD to three levels of 6-LUTs, named J666. The proposed ACD on three levels works well and scales decently for functions up to 13 inputs. For instance, the decomposition time 12-input variable functions the decomposition time per function is approximately of 500 microseconds. We speculate that there is a good margin of improvement by not considering all the possible P-variable partitions at each level, but using a heuristic method (e.g., BDD variable reordering). Future work will consider researching more sophisticated and scalable methods to address this problem while maintaining a similar success rate.

3.4 Technology Mapping with Boolean Decomposition

In this section, we leverage the Ashenhurst-Curtis decomposition (ACD) methods described in Section 3.3 to improve the delay of LUT networks. ACD can be used in two ways: 1) as part of LUT mapping, or 2) as a post-mapping resynthesis method to compact logic and decrease the delay. In this work, we focus on the former usage since it has more flexibility and offers good optimization opportunities. While this work does not cover post-mapping resynthesis, its implementation would involve extracting cuts consisting of a few LUTs, computing the cut function as a truth table, and finally performing ACD. If the new implementation is better, it replaces the old one. For an example of how this resynthesis engine could be implemented, we refer the reader to [138]. First, this section discusses how to perform delay-oriented functional decomposition for any number FS variables and BS functions. Then, it describes the integration of ACD in a technology mapper. Finally, we present the experimental results of the performance-driven technology mapper with ACD. We demonstrate that our technology mapping approach improves the state-of-the-art LUT mapper in ABC with choices by 12.39% in delay and 2.20% in area, on average. Additionally, we present 4 new best results in the EPFL synthesis competition.

3.4.1 Delay-oriented ACD

Let us consider a node n in a k-LUT network and a cut C rooted in n that contains leaves in the input sub-network of n. Among all the leaves, some are timing-critical and some are not. Let D be the latest arrival time of a leaf in C. We use ACD to find an implementation that realizes the function of cut C with delay $D+1$, when $|C| > k$, assuming a unit-delay model. Specifically, we put the timing-critical leaves of C into the FS and other non-critical ones into the BS or SS. This transformation, when applied on the critical path, may reduce the worst delay of a LUT network.

The ACD-based transformation is performed in two steps. First, our method verifies the existence of a delay-minimizing decomposition. Second, if a decomposition exists, it solves the encoding problem and returns a solution.

Checking the existence of a decomposition

Algorithm 3.3 shows the procedure *evaluate* used to check the existence of an ACD. The algorithm receives the function represented as a truth table tt of a large cut of size N where $N > k$. Set S contains a list of timing-critical variables with delay D. First, the truth table is transformed to have critical variables as the least significant ones since they must be in the FS (at line 1). The proposed approach limits $N - P \leq k$ targetting a two-level decomposition without solving the encoding problem. Hence, the number of variables in the FS must be at least $P \geq N - k$, and $P \geq |S|$ to include all the delay-critical variables (in line 4). For each FS of P_i variables, the column multiplicity value is computed using the method described in Section 3.3.2, and the smallest one is returned (at line 5). In this case, since delay-critical variables are always part of the FS, $\binom{N}{P_i - |S|}$ different combinations are enumerated. If the configuration with the smallest column multiplicity is implementable using at most $k - P_i$ BS functions, a delay-minimizing ACD exists. In this case, variables in the FS have the delay increase of 1 while other variables have the delay increase of 2 (at line 12). If, on the other hand, a decomposition with P_i does not exist, the function is not decomposable.

The loop in line 4 checks the existence of a decomposition starting with a smaller value of P. Notably, if a decomposition with P does not exist, neither does it exist with $P+1$, according to Theorem 3.3.3. Then, if a decomposition exists, the loop attempts to identify independent shared-set variables (ISS) to add to the free set, according to Theorem 3.3.2. Specifically, maximizing the free set to include non-critical variables has multiple benefits. First of all, the decomposition would have a reduced column multiplicity, which simplifies the encoding problem. Additionally, including ISS in the FS may reduce the required time of the associated non-critical signals, facilitating area recovery during technology mapping.

Algorithm 3.3: ACD evaluation

```
Input: Truth table $tt$, LUT size $k$, Late vars set $S$
Output: Propagation delay
reorder_variables($tt$, $S$)
$\mu_{best} \leftarrow \infty$
$\vec{x}_{fs} \leftarrow \emptyset$
for $P_i \leftarrow \max(num\_vars(tt) - k, |S|)$ to $k - 1$ do
    $\{\mu, \vec{x}'_{fs}\} \leftarrow$ compute_smallest_multiplicity($tt$, $P_i$, $|S|$)
    if $\mu \leq 2^{k-P_i}$ and $\mu < \mu_{best}$ then
        $\mu_{best} \leftarrow \mu$
        $\vec{x}_{fs} \leftarrow \vec{x}'_{fs}$
        continue
    break
if $\mu_{best} \neq \infty$ then
    return compute_propagation_delay($tt$, $\vec{x}_{fs}$)
return infinite_propagation_delay()
```

Computing the decomposition

After applying *evaluate*, another procedure *decompose* computes the actual decomposition, as described in Section 3.3.3.

3.4.2 Technology Mapping Algorithm with ACD

The methods described in Section 3.4.1 have been integrated into an LUT mapping algorithm. State-of-the-art technology mapping typically performs delay minimization followed by multiple iterations to recover area [141]. Each mapping iteration computes k-feasible cuts rooted in nodes of the subject graphs and selects one best cut for each node based on the cost function and slack. Typically, enumerated cuts are k-feasible, that is, can be implemented by a k-LUT. In our implementation, cut enumeration computes large cuts up to size $k < l \leq 11$, where l is provided by the user. During cut enumeration, the mapper computes cut functions as truth tables. For the non-k-feasible computed cuts, the mapper uses Algorithm 3.3 to check the existence of a delay-minimizing decomposition into k-LUTs. If a decomposition does not exist, the cut is discarded. If a decomposition exists, the cut delay is computed using the propagation delay returned by Algorithm 3.3. The area is estimated using column multiplicity. Specifically, to have precise area information, i.e., the number of required LUTs, ACD has to solve the encoding problem and compute the decomposition. However, experimentally, not running the decomposition on the fly reduces the run time considerably with a negligible impact on the final area. The area is estimated conservatively, neglecting the existence of a shared set, i.e., $Area = \lceil \log_2 \mu \rceil + 1$.

The mapper uses l-feasible cuts with ACD in the delay mapping pass, while it uses k-

feasible cuts in the following area recovery. Note that area recovery aims at improving the solution over non-critical paths and can re-use the best cuts from the previous passes, while assuring that the required times are met. After the last mapping pass, a cover is generated consisting of k- and l-feasible cuts. At this stage, the mapper decomposes non-k-feasible cuts into k-LUTs.

3.4.3 Experimental Results

This section presents an experimental evaluation of the proposed delay-driven LUT mapping with ACD. First, we evaluate our mapper comparing it with the state-of-the-art mapper in ABC. Second, we present new best results in the EPFL synthesis competition [57]. While the experiments are reported for 6-input LUTs, similar improvements have been obtained for 4-input LUTs. For our experiments, we use the EPFL combinational benchmark suite [3] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained using the following script "`dfraig; resyn; resyn2; resyn2rs; if -y -K 6; resyn2rs;`" in ABC, which perform a high-effort size and depth AIG optimization. In particular, it combines SAT sweeping [139], scripts for delay-oriented AIG optimization [132], and lazy man's logic synthesis [216], which is the most aggressive depth minimization for AIGs in ABC. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. The results have been verified using combinational equivalent checkering in ABC.

The proposed methods have been implemented and are available in the open-source logic synthesis framework *ABC* [30]. We extended the LUT mapper *if* to perform ACD, as discussed in Sections 3.4 and 3.5. The following commands are used in the experiments:

- `dch (-f)`: computes structural choices used to mitigate the structural bias [37], where `-f` stands for "fast";
- `if -K 6`: performs delay-oriented technology mapping with choices into 6-LUTs using 6-feasible cuts;
- `if -s -S 66 -K 8`: performs delay-oriented technology mapping using 8-feasible cuts and decomposes logic for minimal delay into two 6-LUTs using a SAT-based formulation;
- `if -Z 6 -K 8`: performs technology mapping into 6-LUTs using the proposed delay-oriented implementation of ACD described in Section 3.4 on 8-feasible cuts;
- `if -S 66`: performs technology mapping based on a given LUT library and packs logic into a structure composed of two 6-LUTs using the ACD method from [165];
- `if -J 66`: performs technology mapping based on a given LUT library and packs logic into a structure composed of two 6-LUTs using the ACD method described in Section 3.5;
- `st`: derives an AIG from an LUT network.

Delay-driven LUT mapping

Table 3.6 compares four technology mapping strategies for delay minimization during mapping into 6-LUTs, assuming a unit-delay model. Each strategy takes the baseline as an input and computes structural choices before mapping. Structural choices have not been used for the benchmark *hyp* due to a known bug in ABC. The proposed method is compared against standard LUT mapping and mapping into LUT structures. In the rightmost column, command *ACD* denotes the sequence "`dch; if -Z 6 -K 8`". We do not compare against [165] and [133] because those methods perform only area-oriented ACD. Furthermore, we do not compare against the recent mapper with gate decomposition based on bin-backing [58] because it can improve the delay of standard ABC `if` by only 0.31% on average.

Mapping into LUT structures "66" composed of two 6-LUTs, which is based on a limited version of structural ACD, reduces depth by 1.04% and the area by 2.57% on average, at the cost of increasing the number of edges by 2.57%. The proposed LUT mapping with ACD improves the depth of the LUT network by 7.52% on average while increasing the number of LUTs and edges by 8.13% and 7.87%, respectively.

Note that most of the improvements are due to the first 10 benchmarks since others are already close to their optimal depth. For 4 of them, the delay reduction exceeds 20% and is up to 27.27%. Practically, part of the area increase can be reduced by area recovery [133, 135, 177], using delay relaxation, or by an additional mapping step applied after ACD. The rightmost strategy performs the latter option. The LUT count and edge count are reduced considerably, leading to an area improvement of 2.20%, compared to traditional technology mapping with choices. Also, the logical depth further decreases up to 54.55%. To achieve this, the LUT network after ACD is used as a structural choice to improve the next round of mapping because choices extracted from mapping with ACD are more structurally suited to delay-oriented mapping, compared to the original AIG. Moreover, structural choices help reduce the area on the non-critical paths. Note that a second mapping round does not give practical benefits if applied after the default LUT mapper (leftmost column) since the network after deriving the AIG is structurally similar to the baseline. Furthermore, benchmark *hyp* is noticeably improved by remapping both in area and delay, although it does not use structural choices. Regarding the run time, mapping with ACD is much faster than mapping into LUT structures while being more general.

EPFL synthesis competition

In Table 3.7, we show that ACD-based LUT mapping can improve well optimized LUT networks, resulting in best known results for 4 benchmarks in the ongoing EPFL synthesis competition. The previous best results were obtained using a portfolio of heavy logic optimization applied to various representations, such as AIGs and LUT networks. In recent years, results have been further improved using *design-space exploration* (DSE) techniques that incrementally generate optimization scripts and visit multiple points of the design space. Examples of these methods

Table 3.6: Comparison of delay-driven LUT mapping, LUT mapping to “66” structure, and LUT mapping using ACD.

Benchmark	**dch; if -K 6**				**dch; if -s -S 66 -K 8**				**dch; if -Z 6 -K 8**				**ACD; st; dch -f; if -K 6**			
	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)	LUTs	Edges	Depth	Time (s)
adder	363	1433	22	0.18	362	1465	20	0.28	383	1519	16	0.20	353	1518	10	0.39
bar	1664	9344	4	0.44	1664	9344	4	0.57	1664	9344	4	0.47	1006	5274	4	0.76
div	8618	32394	406	6.62	9107	33665	397	13.42	11644	44496	326	7.16	9068	39167	271	21.19
hyp	58393	239097	1864	5.43	61701	247699	1840	31.82	65615	264998	1396	11.13	61769	263254	1034	19.76
log2	9712	43562	58	17.05	10172	44943	58	30.06	10313	46365	56	17.81	9429	42533	57	39.09
max	831	3804	14	0.37	840	3668	14	0.63	1211	5578	12	0.42	871	4277	11	1.39
multiplier	7383	34137	36	6.01	7334	32781	36	12.11	7693	35798	33	6.82	6800	31705	31	13.32
sin	1928	8445	30	1.31	1948	8463	30	4.94	2052	8913	29	1.50	1830	8178	30	2.91
sqrt	7515	29573	663	4.17	7972	30610	638	12.66	10156	38558	519	4.73	9292	36030	476	8.77
square	4122	17319	23	1.98	4165	17547	22	3.91	4107	17924	18	2.22	4118	18285	14	5.15
arbiter	1833	8982	6	1.64	1879	8836	6	2.02	1850	8987	6	1.70	2037	8780	6	3.33
cavlc	137	707	4	0.13	104	491	4	0.56	137	707	4	0.15	123	655	4	0.20
ctrl	30	133	2	0.07	28	127	2	0.08	30	133	2	0.08	29	126	2	0.08
dec	287	684	2	0.09	287	1404	2	0.1	287	684	2	0.10	284	816	2	0.12
i2c	312	1360	3	0.16	306	1316	3	0.36	319	1378	3	0.19	297	1329	3	0.27
int2float	52	258	3	0.08	46	205	3	0.18	52	258	3	0.09	50	251	3	0.11
mem_ctrl	11037	48812	18	10.24	10830	46368	18	31.67	11232	49483	17	11.40	10398	45793	16	20.57
priority	178	725	6	0.11	182	736	6	0.18	185	736	6	0.12	171	698	6	0.17
router	89	285	4	0.09	61	283	4	0.14	92	290	4	0.09	89	279	4	0.12
voter	1838	8596	13	2.23	1784	8624	13	4.14	1838	8583	13	2.32	1777	8426	13	4.82
Improvement					2.57%	-2.57%	1.04%		-8.13%	-7.87%	7.52%		2.20%	-0.30%	12.39%	
Total				58.40				149.83				68.70				142.52

are: Bayesian optimization [66], reinforcement learning [154, 220], machine learning, and other heuristic approaches.

We compete in the best delay competition by using standard delay-oriented scripts in ABC and LUT mapping with ACD. We do not use DSE to show that the proposed method outperforms or gets close to the best results in the competition. We obtain the optimized AIGs by repeatedly running the script used in the baseline of Table 3.6 along with additional delay-oriented AIG commands in ABC. For the resulting AIGs, we compare traditional LUT mapping with choices and LUT mapping with ACD. The results are shown in Table 3.7. Notably, results by the traditional mapper are quite far from the best results. This observation shows that our technology-independent optimization finds worse AIGs than those used to obtain the best results, as expected. However, LUT mapping with ACD matches or improves the depth for almost all the benchmarks. The improved benchmarks are *hyp*, *log2*, *multiplier*, and *square*. Remarkably, our method reduces the depth of *hyp* by 10 levels, compared to state-of-the-art while also reducing area by 15%. In the benchmark *multiplier*, our result matches the depth but improves the number of LUTs. Benchmark *sin* is the only one where there is a large gap compared to the best result. The best result for *sin* requires brute-force cofactoring, which is not performed in our synthesis flow.

Unlike many other methods used to produce the best results, our results in Table 3.7 are obtained directly by LUT mapping without post-mapping optimization. For instance, if we use LUT resubstitution, the area of *multiplier* is further reduced to 6499 nodes. Even better results are expected by integrating ACD-based LUT mapping into a DSE flow.

Table 3.7: LUT mapping in the EPFL synthesis competition.

Benchmark	**Best [57]**		**dch -f; if -K 6**		**dch -f; if -Z 6 -K 10**	
	LUTs	Depth	LUTs	Depth	LUTs	Depth
adder	347	5	360	6	445	5
bar	512	4	512	4	512	4
div	25318	175	23461	192	31526	175
hyp	182723	483	122394	511	154903	473
log2	8617	52	8778	60	9613	51
max	1114	6	1113	7	1250	6
multiplier	7785	25	6839	28	6903	25
sin	680530	10	1820	33	2379	27
sqrt	29593	162	30945	172	41626	156
square	3732	10	4189	11	4275	10

3.5 Improving Delay Leveraging Non-routable FPGA Connections

As mentioned in Section 3.1, the delay in the modern FPGAs is often dominated by that of programmable interconnect. To reduce the need for signal routing, one approach modifies the FPGA architecture to include non-routable connections between adjacent LUTs. For instance, recent FPGAs produced by AMD have configurable logic blocks (CLBs) divided into *slices*. A slice contains 8 LUTs that can be connected independently, with external routing, or in a *cascade* fashion using internal direct connections [10]. Specifically, a slice LUT LUT_i, with $0 \leq i < 8$, may connect one of its 6 inputs to LUT_{i-1}, forming a cascade structure. An in-slice cascade connection is 10 to 40 times faster than standard interconnect, which helps delay optimization.

Although in-slice non-routable connections are available, LUT networks generated by the traditional LUT mapping do not use them efficiently. This is because a placement algorithm may fully leverage non-routable connections only for LUTs on the critical path with one critical fan-in. In practice, however, LUTs on the critical path tend to have multiple critical fan-ins, making it hard for the placement algorithm to utilize cascade structures.

An efficient way to leverage cascade connections is to generate mappings of LUTs into cascades during technology mapping. LUT cascades can be generated by decomposing large non-k-feasible functions. In this implementation, we use the ACD method of Section 3.3.5 (Algorithm 3.1) to compute decompositions into specific structures of two LUTs, called "kk" decomposition. Contrary to previous approaches [165], our approach is not based on a heuristic and may support more than one variable in the shared set. Specifically, it always finds a solution if it exists. As previously shown in Table 3.2, its success rate matches the one of a SAT implementation.

3.5.1 Technology Mapping Algorithm

We follow the method proposed in [165] for mapping into LUT structures. Specifically, the LUT mapper performs cut enumeration using cuts up to size l with $k < l \leq 2 \times k$, derives their functions as truth tables, and checks if the functions are decomposable into a "kk" structure. If a function is decomposable, the area and delay are assigned based on a given LUT library. If the function is not decomposable, the cut is ignored. An LUT library specifies the area and delay of an LUT based on its size. Similarly to Section 3.4.2, the mapper begins by minimizing delay, followed by several iterations of area recovery. Contrarily to Section 3.4.2, the mapper uses ACD decomposition of l-feasible cuts during all mapping iterations.

3.5.2 Experimental Results

In this experiment, we perform technology mapping into LUT structures by leveraging non-routable cascade connections of LUTs in FPGA architectures. We use the same baseline of Section 3.4.3. Motivated by the high cost of routing, we assume that a 6-LUT and a cascade of two 6-LUTs both have unit delay. A more precise model would assign propagation delay of about 1.2 to the signals in the bound set of a LUT cascade and unit delay to the signals connected to the composition function. However, the mapper in [165] only supports a fixed delay assignment to all the signals. Hence, we assume the delay of a cascade to be unitary to not penalize the quality of mapping into LUT structures. We run all the mappers with the same parameters to perform minimal-delay mapping. Mappers running ACD use cuts up to 10 inputs.

Table 3.8 compares traditional LUT mapping with choices, the LUT structure mapping [165], and the proposed method described in Section 3.3.5 supporting 1 (1-SS) or multiple (M-SS) shared set variables. S66 improves the traditional mapper by 30.74% in delay while increasing area and the number of edges. For many benchmarks, the area increases due to logic duplication to minimize delay. Notably, J66 1-SS considerably improves all the metrics, compared to S66. The improvement comes from the better success rate of the decomposition shown in Table 3.3. Moreover, J66 M-SS achieves further improvement, compared to S66, reducing the average delay, area, and edge count by 6.22%, 3.82%, and 3.09%, respectively, with a faster run time. Remarkably, for designs with a similar delay to the traditional mapper, J66 achieves a large reduction in the number of LUTs and edges. This is because J66 successfully mitigates structural bias. For instance, for benchmark *int2float*, J66 M-SS reduces the number of LUTs by 27%. For the same benchmark, S66 reduces the number of LUTs only by 1.92%. Similar improvements are also observed for all the benchmarks when performing area-oriented mapping, instead of delay-oriented mapping. Another interesting benchmark is *cavlc*, where multiple shared set variables significantly improve the delay, area, and edge count.

While S66 is generally faster than J66 for large functions, the mapping time of J66 is better than that of S66. This is because J66 is faster when applied to frequently appearing decomposable functions and slower when applied to non-decomposable functions. After all, it uses

Table 3.8: Comparison of delay-driven LUT mapping and multiple ACD-based mapping into “66” cascade structures.

Benchmark	**dch; if -K 6**				**dch; if -S 66** [165]				**dch; if -J 66 1-SS**				**dch; if -J 66 M-SS**			
	LUTs	Edges	Delay	Time (s)	LUTs	Edges	Delay	Time(s)	LUTs	Edges	Delay	Time(s)	LUTs	Edges	Delay	Time(s)
adder	363	1433	22	0.18	352	1521	13	0.85	356	1552	13	0.52	354	1550	13	0.83
bar	1664	9344	4	0.44	1664	8320	3	1.34	1664	8320	3	0.75	1664	8320	3	0.78
div	8618	32394	406	6.62	11555	46558	266	34.87	11071	45711	251	27.54	11298	47587	248	30.87
hyp	58393	239097	1864	5.43	65987	274992	1144	270.03	65352	274000	1082	161.75	65175	273434	1076	183.77
log2	9712	43562	58	17.05	12813	59950	42	73.19	12526	58798	40	54.03	12409	59528	39	71.87
max	831	3804	14	0.37	1177	6162	9	1.77	1113	5448	9	1.31	1113	5448	9	1.44
multiplier	7383	34137	36	6.01	8898	42566	25	46.10	8861	42005	25	31.27	8645	43556	24	36.15
sin	1928	8445	30	1.31	2620	12074	22	12.45	2461	11125	21	9.39	2400	10977	21	13.17
sqrt	7515	29573	663	4.17	9510	37809	423	42.21	9109	37396	403	24.86	9441	38373	398	32.31
square	4122	17319	23	1.98	4299	19677	15	11.75	4290	19843	14	8.07	4299	19972	14	12.65
arbiter	1833	8982	6	1.64	2000	9481	4	2.71	1992	9834	4	2.53	1992	9834	4	2.50
cavlc	137	707	4	0.13	125	645	3	0.49	124	639	3	0.43	110	565	2	0.54
ctrl	30	133	2	0.07	28	131	2	0.08	28	133	1	0.09	28	133	1	0.09
dec	287	684	2	0.09	512	2304	1	0.11	512	2304	1	0.12	512	2304	1	0.12
i2c	312	1360	3	0.16	327	1530	2	0.49	319	1478	2	0.41	306	1433	2	0.45
int2float	52	258	3	0.08	51	257	2	0.17	42	216	2	0.17	38	191	2	0.20
mem_ctrl	11037	48812	18	10.24	11666	52725	13	59.20	11247	51109	13	48.04	11019	50726	13	56.43
priority	178	725	6	0.11	175	761	4	0.21	176	768	4	0.28	176	768	4	0.28
router	89	285	4	0.09	65	305	3	0.15	65	306	3	0.19	65	303	3	0.21
voter	1838	8596	13	2.23	2133	10793	10	7.22	2068	10082	10	5.58	2053	10081	10	7.87
Improvement					-13.65%	-27.62%	30.74%		-10.73%	-24.52%	34.29%		-9.44%	-24.00%	35.86%	
Total				58.40				565.39				377.33				452.53

more effort to find a solution. For instance, on the benchmark *sqrt*, which has a considerable run time difference between S66 and J66, only 2.93% of all cuts are not decomposable by J66, compared to 11.45% by S66. Moreover, only 10.35% of 10-input cuts are not decomposable by J66 M-SS, while 39.48% are not decomposable by S66. Run time could be further reduced by taking advantage of GPU-based LUT mapping implementations [120].

Although not deeply studied in this chapter, our ACD-based LUT mapping approach shows great potential in area optimization. For example, we optimized the *int2float* benchmark the following script from our experiments: `dfraig; resyn; resyn2; resyn2rs; if -y -K 6; resyn2rs;`. In one flow, we ran `&if -K 6 -a` for area-oriented mapping, followed by *mfs* and *lutpack* until saturation for post-mapping area optimization. This resulted in a network of 47 LUTs. In the other flow, we used only our ACD method for area-oriented mapping by running `&if -J 66 -K 10 -a -z`[2]. This approach resulted in a network of 34 LUTs, marking a substantial reduction in the number of LUTs by 27.6%. Similarly, the first flow obtains a network of 135 LUTs for benchmark *cavlc*. Our mapper finds a network with 102 LUTs, achieving a remarkable reduction in number of LUTs by 24.4%. Therefore, the methods discussed in this chapter have the potential to offer significant area reductions.

3.6 Summary

In this chapter, we proposed two enhancements to Boolean decomposition into LUTs and performance-driven technology mapping for FPGAs.

First, we presented efficient methods to compute Ashenhurst-Curtis decomposition (ACD). We provided a theoretical base and practical algorithms to solve its sub-problems, namely the feasible variable partition, functional encoding, and the shared set maximization. Our

[2] Option `-z` decomposes the LUT structures created by J66 into LUTs cascades.

algorithm is truth-table-based and it is the first of its kind for its flexibility in the number of free set, bound set, and shared set variables, and the number resulting LUTs. We provided algorithms to optimally solve the decomposition of functions into two LUTs, as well as heuristics for generating two-level or multi-level LUT structures. Our approach outperforms all of the previous state-of-the-art methods in their ability to derive a decomposition by a large margin, up to 35.58% with a better run time.

Second, we proposed a LUT mapping algorithm that integrates ACD on-the-fly to improve performance-driven technology mapping. Experimental results show an impressive 12.39% delay improvement compared to the state-of-the-art LUT mapper in ABC with choices, with a surprising area reduction of 2.20%, and with competitive run time. Additionally, we presented four new best results in the EPFL synthesis competition. These results were obtained without using design-space exploration (DSE) methods. Hence, we expect even better results by using LUT mapping with ACD in a DSE tool. Third, we focused on improving the performance of FPGA by leveraging non-routable connections. We used our formulation of ACD to compute mappings of Boolean functions into LUT structures composed of two LUTs connected by a non-routable connection. We integrated this version of ACD in a technology mapper. Compared to the state of the art, we showed that this method reduces the delay, area, and edge count by 6.22%, 3.82%, and 3.09%, respectively, with better run time. The overall improvement in delay for the selected benchmarks, compared to not using non-routable connections, is 35.86%. Additionally, we showed the great potential of these methods in area-oriented mapping.

The findings of this work have impact beyond technology mapping. LUT mappers are key in design-space exploration engines and in various optimization flows, for example, in those used for standard cells [153]. Hence, the methods proposed in this chapter may significantly improve the quality of logic synthesis tools, especially for delay optimization.

4 Technology Mapping for Standard Cells

Chapter 3 was dedicated to studying technology mapping algorithms for field-programmable gate arrays (FPGAs). In contrast, this chapter explores novel technology mapping methods designed to improve the quality of results (QoR) of standard-cell-based designs, which include general-purpose processors and application-specific integrated circuits (ASICs). The focus is on algorithms that transform a synthesized logic network into an interconnection of standard cells described by standard-cell libraries. Specifically, these methods include: (i) novel techniques to perform high-quality and scalable matching; (ii) algorithms for technology mapping using multiple-output cells; and (iii) advanced technology mapping covering algorithms. The content of this chapter is largely based on the publications in [163, 197, 202].

The remainder of this chapter is organized as follows. First, we present the motivations of this chapter in Section 4.1 and the relevant background on technology mapping for standard cells in Section 4.2. Next, Section 4.3, based on the publication in [163], presents a matching technique called *hybrid matching* to improve over Boolean matching by supporting large cells and leveraging structural redundancies. Circuits mapped using *hybrid matching* typically show a 6.5% average reduction in area compared to Boolean matching, for similar delay. Then, Section 4.4, based on the publication in [197], describes methods to support multiple-output cells during technology mapping. This section addresses the problem by proposing Boolean matching for multiple-output cells and introducing the first selection and covering algorithm for multiple-output cells. In the experiments, we compare our mapper against ABC showing a 7.48% area reduction on average when mapping arithmetic circuits for the minimal delay. Furthermore, our mapping algorithm reduces the area by 5%, on average, compared to the mapping method of Yosys in which adder cells are considered as white boxes during technology mapping. Next, Section 4.5 studies technology mapping covering algorithms, which are used to select a subset of cells that cover a logic network while minimizing cost metrics and/or meeting target constraints. The experimental results show that the proposed mapping heuristics can improve area by 4.66%, on average, compared to other mappers in ABC when mapping for best delay. Following this, Section 4.6 presents *emap*: a technology mapper that incorporates the algorithms discussed in this chapter. This section includes several experiments with multiple technology libraries and benchmarks. Moreover, we show results after buffering and gate sizing. In the experimental results, we show that *emap* achieves better average area and run time than the mappers in ABC. Finally, Section 4.7 concludes and summarizes this chapter, highlighting the key findings and contributions.

4.1 Motivation

CMOS is the most advanced technology for integrated circuits offering unparalleled performance, power consumption, area, and reliability. Despite the physical scaling limitations and the gradual slowdown of Moore's Law, CMOS-based chips still maintain a significant lead over emerging technologies in terms of scalability, cost-effectiveness, and maturity of the manufacturing process. Moreover, ongoing research and development efforts continue to push the boundaries of CMOS technology to its limit, ensuring its continued relevance and superiority.

Technology mapping is one of the fundamental steps in the realization of integrated circuits. It consists of translating a technology-independent representation of digital hardware into a connection of technology-specific components. Standard-cell-based design utilize a semi-custom design methodology based on *standard cells*. Standard cells are pre-designed, pre-characterized blocks of transistors configured to perform specific logic functions and serve as the fundamental building blocks for creating complex integrated circuits. This chapter focuses on technology mapping algorithms for standard cells-based design, which translate a technology-independent representation of digital hardware, modeled as a multi-level logic networks, into a network of standard cells.

The problem of optimally mapping Boolean functions to a cell library is known to be intractable. Therefore, technology mapping is generally formulated as a series of local substitutions applied to a simple multi-level representation of logic called the *subject graph*. The goal of technology-independent logic synthesis is to create a compact subject graph in terms of size and depth to facilitate high-quality technology mapping, as a compact subject graph correlates with a high-quality mapped circuit. This chapter discusses the typical challenges of technology mapping, such as delay minimization and area minimization (possibly under delay constraints). Given the complexity of the technology mapping problem, numerous heuristics and solutions have been proposed in the literature [37, 38, 65, 79, 84, 87, 90, 98, 114, 123, 146, 186]. This chapter revisits technology mapping for standard cells, introducing new heuristics and methods to enhance the quality of results.

In addition to addressing the technology mapping problem itself, the increasing physical scaling limitations of transistors have led to the evolution of standard cell libraries to be more comprehensive. For example, libraries at the 7nm technology node include cells capable of realizing more complex functionalities compared to older nodes, such as large AND-OR cells with up to 9 inputs. Moreover, libraries include multiple-output cells, such as *half adders* and *full adders*, which are often neglected by technology mappers. Consequently, technology mapping needs to evolve to better leverage these advanced libraries.

This chapter has three main research contributions. First, motivated by technology libraries with large cells, we propose *hybrid matching* as a new method to solve the matching problem during technology mapping. The main advantage of hybrid matching is its ability to support large-input cells and its scalability. This method overcomes the quality limitations of pattern matching and the cell-size limitations of Boolean matching. Second, we present algorithms to increase the support of multiple-output cells in technology mapping, addressing the following problems: (i) multiple-output cell detection; (ii) Boolean matching for multiple-output cells; and (iii) cell selection and covering algorithms with multiple-output cell support. Third and last, we revisit heuristic algorithms for technology mapping to achieve better area under delay constraints. Specifically, we analyze methods for cell selection (covering) and inverter insertion.

Based on these contributions, we developed *emap*: a technology mapper for standard cells that includes the algorithms discussed in this chapter. The mapper is available in the open-source logic synthesis library *Mockturtle*[1] [183].

We experimentally evaluate the proposed methods by comparing the results with state-of-the-art technology mappers. A summary of the experimental results with the ASAP 7nm standard cell library [44] is as follows:

- We show that *hybrid matching* reduces the area under delay constraints up to 39% and by 6.5% on average compared to Boolean matching. Additionally, hybrid matching reduces the run time of technology mapping by 25%, on average, compared to Boolean matching.

- We show that multiple-output technology mapping with *half adders* and *full adders* achieves a 7.48% area reduction on average when mapping for minimal delay in arithmetic circuits. Additionally, we show that our method reduces the area by 5% on average compared to considering adder cells as white boxes during technology mapping.

- We show that the proposed cell selection methods reduce the average area by 4.66% when mapping for best delay compared to ABC.

- We evaluate *emap* against the state-of-the-art mappers in ABC before and after buffering and gate sizing. When mapping for best delay, we show that *emap* achieves 9.16% better area and 2.95% better delay after technology mapping, and 9.22% better area and 2.59% better delay after buffering and gate sizing.

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

4.2 Preliminaries

In this chapter, we research algorithms to solve the technology mapping problem for combinational logic networks. While digital circuits are inherently sequential, the choice of implementation of registers, I/O circuits and drives is often accomplished through direct replacement. Sequential circuits can be transformed into combinational ones by treating register and box outputs as additional inputs of the network and register and box inputs as additional outputs of the network.

Minimum-cost technology mapping, defined in general terms as finding a set of cells realizing a circuit specification while minimizing cost, is an intractable problem. It is a form of minimum-cost satisfiability, whose exact solution has time complexity higher than polynomial, as for exact synthesis [152]. Hence, this problem is approached as series of local substitutions applied to a multi-level logic network implementing the specifications called the *subject graph*. The key objective of technology-independent logic synthesis is to achieve a compact subject graph, both in terms of size and depth, to facilitate technology mapping and enhance quality. This approach is often referred to as *structural* [90].

This idea assumes that a good structure for a Boolean network, achieved through technology-independent synthesis, correlates with good QoR after technology mapping. While this assumption generally holds true for technology mapping using standard cells, it introduces a bias, known as *structural bias*. This bias occurs because the network after technology mapping is structurally similar to the subject graph, limiting the exploration of alternative configurations that might offer better quality.

Standard cell libraries define a set of pre-designed and pre-characterized logic primitives that are used as building blocks to create digital circuits. Technology mapping uses these blocks to cover the subject graph while minimizing a cost function, typically based on power, delay, and area. Mapping addresses two sub-problems: *matching* and *covering* (also known as *selection*). Matching involves associating sections of the subject graph with a list of cells that are functionally equivalent and capable of implementing those sections. Covering chooses a set of cells to cover the graph such that the target cost function is minimized.

In the following sub-sections, we further present the terminology and the state-of-the art methods for estimating delay and solving the matching and selection problems.

4.2.1 Delay Models in Technology Mapping

Accurate cost estimations are crucial in technology mapping. However, costs such as delay and switching power depend on factors such as the load of the cells and the input transition time, which remain unknown until a mapping solution is extracted. Moreover, interconnects also play an important role. Hence, correctly estimating the delay and identifying optimal matching is challenging. Additionally, standard cell libraries define multiple discrete sizes for each cell, based on the load they can sustain. For a cell n, increasing the size enables driving additional load while maintaining similar delay, but also increases the load on the cells in n's fan-in. In this section, we focus on delay models, but the considerations apply for switching power.

Standard cell libraries typically use the non-linear delay model (NLDM), which accurately describes the response of cells depending on their operative environment. The NLDM format provides look-up tables that describe the delay response of cells based on input transition time (slew) and output load. The delay response is characterized by the transition time and the propagation time components, which typically depend on the input values. Standard cell libraries are often described in *liberty* format. While essential for accurate static timing analysis, the NLDM's complexity makes it impractical for direct use in the technology mapping process. Thus, various approximations to the delay model have been adopted. In this section, we present the most used models for technology mapping: the *constant* and *linear* delay models.

Constant delay model

A constant delay model assumes that the delay of a cell is independent of its load and the input transition time. The benefits of this method are its simplicity and the computational efficiency, which enable the use of more advanced match selection algorithms. Moreover, this delay model can be used as an initial estimation of the delay of a network. Predictably, the drawbacks of this method is its inaccuracy, since the delay can strongly vary based on the load and transition time. Typically, mapping based on the constant delay model uses only cells with smaller sizes. Subsequently, the delay is more accurately computed and minimized during the buffering and gate sizing phase.

Given a standard cell library in NLDM format, extracting load-independent propagation delay is not straightforward. Modern methods are based on the concept of *logical effort* [188] to model the load-independent delay of a cell:

$$d = e \cdot g + p. \tag{4.1}$$

The effort delay depends on the load ($e \cdot g$) and properties of the cell driving the load (p). In Equation 4.1, p is the load-independent parasitic delay, caused primarily by the parasitic capacitance of the cell, e is the *logical effort*, which depends on the cell's topology and is independent of the load capacity, and g is the *electrical effort*, or *gain*, which describes how the electrical environment affects the performance of the cell. The gain g is described by:

$$g = \frac{C_{out}}{C_{in}}, \tag{4.2}$$

where C_{out} is the load at the output terminal of the cell, and C_{in} is the load at the input terminals of the cell.

From this model, it can be seen that the delay is independent of the load as long as the gain g is known. Moreover, logical effort states that optimal delay is achieved when the effort delay ($e \cdot g$) is balanced for all the stages on the critical paths. Since e varies depending on the type of cell, g is the parameter to tune. The greater the logical effort, the smaller the load a cell can drive while maintaining the same effort delay. Additionally, more realistic delay models also include the transition delay (slew rate) [179].

If the standard cells are continuously sizeable, it is possible to find a cell such that the gain is balanced, meeting the target delay. Hence, a quite accurate constant delay model is achieved. However, cell libraries contain a limited number of sizes. Consequently, a cell might not be able to satisfy the load requirement to maintain constant delay. Nevertheless, a range of loads for which the constant delay model holds can be determined. A method to characterize the constant delay model given a cell library of discrete sizes is proposed in [79]. This approach requires to fix a target slew rate τ over the critical path to extract the delay model of Equation 4.1 and a global gain G to determine the target gain at each stage. Parameter τ can be estimated as the average slew of an inverter. Parameter G is related to the inverter in the library. Parameters e and p in Equation 4.1 are computed for each pin of the cells via linear regression over the NLDM delay values. Then, the method computes a *gain-based cell* for each functionality group (i.e., it groups multiple cell sizes) such that the constant delay model holds. We refer the reader to [79] for further details of the methodology.

Technology mappers using a delay model based on logical effort are often referred to as *gain-based* [79, 87, 186]. Gain-based mappers have been shown to be as accurate as more sophisticated mappers using load-based delay models in many cases and, currently, are the most common [21, 38, 79, 146]. Additionally, gate sizing is separated from technology mapping, as the mapping process only involves gain-based cells.

Linear delay model

A linear delay model estimates the delay of a cell as follows:

$$d = c \cdot l + p, \tag{4.3}$$

where p is the load-independent propagation delay, c is a (slope) constant, and l is the load of the gate. While the format resembles Equation 4.1, the equation describes a different relation. Constant c represents the delay induced by units of load driven by the cell. Values for c and p are computed and assigned to each input pin of a cell. Contrarily to the gain-based method, technology mappers using the linear delay model perform gate sizing during mapping. Often this model is further approximated. The load l is replaced by the fan-out number and c becomes the delay contribution of each fan-out. This model is referred to as *nominal delay model.* The nominal delay model is simpler since it does not compute an accurate load. The technology mapper in SIS [178] adopts this latter model. A common library format that describes cells using linear or nominal delay model is the *genlib* format[2].

Key points

The mapping algorithms in this thesis adopt a constant delay model. We argue that the constant delay model is a good choice considering that technology mapping is performed before buffering, sizing, placement, and routing, stages where more realistic delay estimations are available. Consequently, the delays computed by more complex models would still be large approximations at this stage. Additionally, the constant delay model simplifies technology mapping algorithms, making them more efficient and capable of exploring a broader solution space. More precise delay models, such as those accounting for interconnect delays [56, 67], can be considered during technology-dependent logic synthesis and incremental remapping [22]. Alternatively, the constant delay model can be refined through multiple delay-oriented mapping iterations to enhance accuracy and better reflect the effort required by the critical path. Despite this, the algorithms proposed in this thesis can be extended to accommodate different delay models.

4.2.2 Related Works

Algorithms for technology mapping where pioneered by Keutzer [90], who addressed both the matching and the selection problem. In his approach, the subject graph is partitioned into trees and matches are extracted for each tree using pattern matching. Keutzer proposed an efficient algorithm based on dynamic programming to combine matching and covering. He showed that minimum-area and minimum-delay mapping is solvable in linear time for trees when using a constant delay model. For instance, the area of a tree can be optimally computed as follows. The nodes in a tree are processed in topological order, from the inputs to the output. The area contribution of match at a node is computed as the the area of the corresponding cell plus the area contribution of the best matches at the leaves of the match. The best match at a node is the match minimizing the area contribution. By induction, the set of matches reachable from the output of the tree represent an area-optimal mapping.

The optimality results in [90] are weakened by the nature of logic networks being *directed acyclic graphs* (DAGs). Consequently, optimality on the entire subject graph is not guaranteed. Moreover, optimality is assured only if the pin-to-pin rise and fall delays are identical; otherwise the problem becomes NP-hard [150].

Rudell [172] proposed an extension over Keutzer's work with a linear-time algorithm to optimally solve the tree mapping problem under a linear delay model. He realized that standard cell libraries have a fixed and small set of sizes. Consequently, he proposed to store for each node multiple best matches for each possible capacitive load. If enough matches are used to cover all the possible loads, the algorithm returns an optimal solution. An improvement

[2] The genlib format is also used for libraries following the constant delay model by setting c to zero.

over this work consisted of using a piece-wise linear curve to model the optimal arrival times as a function of the load, where each line represents a different cell [206]. Additionally, the mapper in MIS [53] extended pattern matching to DAGs, eliminating the initial partitioning of the subject graph into trees.

While previous mappers relied on pattern matching techniques, Mailhot [123] expanded on Keutzer's work by introducing Boolean matching based on Shannon decomposition. This approach addressed the challenges of pattern matching for logic functions with multiple occurrences of the same variable, such as XORs or majority functions. Furthermore, Boolean matching could potentially leverage *don't care* conditions to optimize the circuit for size or speed.

Up to this point, optimal algorithms for technology mapping had been presented only for trees. The first work to propose an optimal algorithm mapping for DAGs was *FlowMap* [45]. With FlowMap, Cong and Ding demonstrated that there is a polynomial time algorithm to solve the delay-optimal match selection problem under the unit delay model. The algorithm is based on network flow computations. This algorithm maintains the optimality property under static net delay models for FPGAs [48], which can be seen as a special case of the constant delay model. Kukimoto et al. used the FlowMap idea in standard cell technology mapping, showing that the optimality holds under a constant delay model [98]. When considering load-dependent delay models, the delay-optimal mapping is an intractable problem.

Regarding area optimality on DAGs, Levin and Pinter [117], and Farrahi and Sarrafzadeh [59] proved that it is an NP-complete problem by showing that a 3-SAT problem can be transformed into a k-LUT minimization problem. Furthermore, they show that the problem is still NP-complete for single-output networks and circuits with bounded fan-ins and fan-outs. While their proof is for FPGA mapping, the same argument holds for standard cell mapping.

Due to the intractability of area-optimum mapping, many heuristics have been proposed for FPGA mapping [40, 124], which were later extended for standard cell mapping [38]. These heuristics aim to provide practical solutions to the mapping problem by balancing the trade-offs between computational complexity and the quality of the resulting mappings.

All these works focus on technology mapping for single-output cells. In Section 4.4, we propose algorithms to extend the technology problem to multiple-output cells.

4.2.3 Covering

The covering problem, often referred as *match selection* or simply as *selection*, chooses a set of the matches to cover the subject graph and produce a mapped network minimizing a cost function or satisfying the given constraints. Selection is generally divided into two sub-problems: *match evaluation* and *cover extraction*.

Match evaluation is responsible for selecting the best matches rooted in a node based on specific cost functions, such as delay and area. Typically, the match evaluation is conducted in topological order, from the primary inputs to the primary outputs, so that the cost of a match includes the cost of the best matches at its inputs. This enables a correct propagation of delays and area estimations.

After match evaluation, cover extraction computes a mapped network by selecting the best matches reachable from the primary outputs of the network. For each *primary output* (PO), the algorithm adds the best match to the cover. Then, recursively, each match at the leaves (inputs) of matches in the cover is also included in the cover until the primary inputs are reached.

In the remainder of this section, we present the state-of-the-art methods addressing the selection problem to minimize delay and area.

Covering for delay

Under the constant delay model, it is possible to extract a delay-optimal mapping with respect to the subject graph using the algorithm from [45, 98]. The approach for standard cells can be formulated as follows. The algorithm proceeds in topological order. Let $M(n)$ be a set of matches rooted in a node n of the subject graph. We then define the delay of a match to be D_m and the best delay of a node n to be $D(n)$. The delay D_m can be computed as:

$$D_m = \max_{l \in leaves(m)} (D(l) + p^m(l)), \tag{4.4}$$

where $p^m(l)$ represent the propagation delay of m from the input pin connected to l to the output pin of the match. Consequently, the best delay at a node can be computed as follows:

$$D(n) = \min_{m \in M(n)} D_m. \tag{4.5}$$

In [98], all the matches at a node are computed using a pattern matching algorithm for DAGs. The complexity of the delay-optimal mapping algorithm is $\mathcal{O}(p \cdot N)$, where N is the number of nodes in the network and p is the total number of nodes in all pattern graphs. Given that modern cell libraries contain a large number of cells, and thus patterns, p can be quite large number. Consequently, state-of-the-art standard cell mappers generally sacrifice delay optimality for improved runtime by selecting a limited number of matches (or cuts) at each node using cut prioritization algorithms [47, 141]. This approach also facilitates scalable usage of Boolean matching.

Covering for area

Due to the NP-completeness of the optimal-area mapping problem, area-oriented covering is approached similarly to delay-oriented covering using heuristics. In this section, we review the two most important heuristics, namely, *area flow* and *exact area*.

The *area flow* [124], or *effective area* [40], of a match m rooted in node n is computed as follows:

$$AF_m = a_m + \sum_{l \in leaves(m)} \frac{AF(l)}{f(l)}, \tag{4.6}$$

where AF_m is the area flow of match m, a_m is the area of m, leaves l are the input nodes of the match, $AF(l)$ is the area flow of the best match at node l, and $f(l)$ is the fan-out number of node l, which describes the fan-out of matches rooted in l. For primary inputs or constants, the area flow is considered to be zero. The area flow uniformly distributes the area contribution at l over the fan-out nodes of l. This has the benefit of accounting for the multiple fan-outs during the estimation of the area in the TFI cone of m. Similarly to delay covering, area flow covering selects in topological order the match minimizing area flow at the node:

$$AF(n) = \min_{m \in M(n)} AF_m. \tag{4.7}$$

For an efficient computation in technology mappers, area flow at a node is often expressed as:

$$AF'_m = \frac{AF_m}{f(m)} = \frac{a_m + \sum_{l \in leaves(m)} AF'(l)}{f(m)}. \tag{4.8}$$

In this chapter, we use this latter formulation.

If accurate fan-out numbers $f(l)$ are known, the sum of the area flow over the POs of a network results in the area of its cover. However, during covering the actual fan-out numbers are not known and have to be estimated. Specifically, fan-out references are available only after the cover extraction phase while area flow is used in the match selection flow (the cells in the cover are extracted using a reachability analysis from the POs; during match selection the cells in the cover are not known). The approaches in [38, 140] initially estimate the fan-out reference of a node to be the one of the subject graph. In this way, matches at nodes with higher fan-out are more likely to be selected since they better distribute the area flow to the fan-out. If area flow covering is performed for multiple iterations, to incrementally improve the cover, fan-out references are updated as follows. If a node is in the cover, its fan-out reference number matches the one in the cover. Otherwise, its fan-out reference number is assumed to be one. However, this approach is often too aggressive and quickly leads to a local minima. A better approach generates references using a linear combination between the previous estimations and the current cover [30, 201]. At round j:

$$f(n)^j = \max(\alpha \times f(n)^{j-1} + (1-\alpha) \times ref(n), 1), \tag{4.9}$$

where $f(n)^j$ are the estimations at round j, $f(n)^{j-1}$ are the estimations at the previous round $j-1$, $ref(f)$ are the actual references in the cover, and $\alpha \in [0,1]$ is parameter. Parameter α can be considered as a temperature controlling the convergence rate to the reference value of the cover and is computed experimentally. While having a dependency on the circuit structure,

Algorithm 4.1: Recursive dereferencing and referencing for exact area

```
Input: node $n$, match $m$
Output: exact area
Algorithm dereference($n$, $m$)
    $area \leftarrow a_m$
    foreach node $l \in leaves(m)$ do
        $f(l) \leftarrow f(l) - 1$
        if $f(l) = 0$ then
            $area \leftarrow area +$ dereference($l$, $best\_match(l)$)
    return $area$
Algorithm reference($n$, $m$)
    $area \leftarrow a_m$
    foreach node $l \in leaves(m)$ do
        if $f(l) = 0$ then
            $area \leftarrow area +$ reference($l$, $best\_match(l)$)
        $f(l) \leftarrow f(l) + 1$
    return $area$
```

we empirically find that for a convergence within 2 or 3 rounds of area flow:

$$\alpha^j = \frac{1}{(j+1)^2}, \tag{4.10}$$

is a good metric. Parameter α starts to be used from the second mapping round when $j = 1$. In the following rounds, α is decreased to converge to a solution.

Another powerful method for minimizing area is *exact area* [38, 140]. Similarly to local search in set covering problems, exact area aims at iteratively improving an existing cover by first removing a match from it, together with its dependencies in the MFFC, and then adding an alternative match that has lower area than the removed one. Algorithm 4.1 shows the two methods, *dereference* and *reference*, to remove a match from the cover and add a match to the cover, respectively, while computing the local change in the area. The fan-out number $f(l)$ for a node l counts the fan-out of the best match at l in the cover. Note that methods in Algorithm 4.1 are recursive, since the dependencies in the MFFC are considered.

4.3 Hybrid Matching

This section contains the first technical contribution of this chapter and focuses on the matching problem. In early approaches, cells were represented using a graph, typically in the form of a NAND decomposition of the Boolean function. The matching task was then formulated as a (sub)graph isomorphism problem, particularly efficient when the decomposition graph is a tree. This form of matching is referred to as *pattern matching* [90]. However, this approach has several drawbacks. The most important one is that the graph decomposition is not canonical.

Consequently, the number of possible graph decompositions can grow exponentially large for some cells, making it challenging to detect potential matches. Moreover, the matching process is significantly more complex when the decomposition graph is not a tree, such as in the case of XOR cells or majority cells. Later approaches used *Boolean matching* [123], which is based on a canonical Boolean representation of the function, to mitigate the limitations of pattern matching. Boolean matching inherently solves a tautology problem. Typically, this technique scales to cells up to 6 inputs, covering the majority of the cells present in standard cell libraries. Modern approaches use truth tables or BDDs as a canonical data structure.

In this section, we propose *hybrid matching* for technology mapping: an approach that combines the strengths of pattern and Boolean matching to achieve better quality. On the one hand, pattern matching finds application for many cells that are decomposable into NAND trees, such as AND-ORs. Furthermore, modern cell libraries contain cells of 7 or more inputs, beyond Boolean matching capabilities (for a reasonable run time). On the other hand, Boolean matching generally offers better quality for cells up to 6 inputs. By leveraging the benefits of both techniques, hybrid matching computes both pattern and Boolean matches and strategically combines them to achieve better quality and run time speedup.

We have integrated hybrid matching in the technology mapper *emap* in the logic synthesis library Mockturtle [183]. Although we work with a gain-based delay model, the methods discussed in this section are compatible with physical-aware mappers since they are related to matching and not to covering. In the experiments, we compare hybrid matching to both Boolean and pattern matching showing that:

- Hybrid matching reduces the area up to 39% compared to Boolean matching, making efficient use of large gates. On average, it reduces the area by 6.5% while maintaining a competitive worst-case delay.
- Hybrid matching reduces the area by 4.6%, on average, compared to pattern matching for better delay.
- Hybrid matching reduces the run time of technology mapping by 25% compared to Boolean matching.
- Hybrid matching achieves comparable average delay performance to Boolean matching.

The remainder of this section is organized as follows. We first present the background on Boolean decomposition, used to automatically generate decomposition patterns for standard cells in Subsection 4.3.1. Then, we describe our methodology for Boolean matching and pattern matching in Subsection 4.3.2. Next, we present *hybrid matching* in Subsection 4.3.2. Finally, Subsection 4.3.4 presents the experimental results.

4.3.1 Preliminaries

In the preliminaries, we introduce the *disjoint-support decomposition* (DSD) as an efficient method to derive a decomposition graph of a Boolean function. In hybrid matching, we use DSD to derive decomposition patterns for each library cell.

The DSD is a special case of Boolean decomposition. A function f has is disjoint-support decomposable if it can be decomposed so that:

$$f(\vec{x}) = h(\vec{x}_1, g(\vec{x}_2)), \quad \text{with} \quad \vec{x}_1 \cup \vec{x}_2 = \vec{x} \quad \text{and} \quad \vec{x}_1 \cap \vec{x}_2 = \emptyset. \tag{4.11}$$

A Boolean function is called *full-DSD* if function g is recursively decomposable with disjoint support. Two efficient procedures can be used to find DSD decompositions using 2-input operators, namely *top-down decomposition* [25] and *bottom-up decomposition* [36].

Top-down decomposition finds a decomposition using a 2-input operator $\odot$ applied to a support variable x_i and a remainder function g:

$$f(\vec{x}) = x_i \odot g(\vec{x} \setminus \{x_i\}), \quad x_i \in \vec{x}. \tag{4.12}$$

Bottom-up decomposition finds two variables x_i and x_j that uniquely influence f through a 2-input operator:

$$f(\vec{x}) = h(x_i \odot x_j, g(\vec{x} \setminus \{x_i, x_j\})), \quad x_i, x_j \in \vec{x}. \tag{4.13}$$

Example 4.3.1. *The function $f = ((a \vee b) \wedge (c \vee d)) \wedge e$ is top-down decomposable for variable e and it is bottom-up decomposable for variables (a, b) and (c, d).* ▲

4.3.2 Boolean and Pattern Matching

In this section, we present our methodology for Boolean and pattern matching. These two techniques are then combined to obtain a hybrid mapper (in Subsection 4.3.3). Throughout this section, we utilize the term *Boolean mapping* to denote a mapping algorithm based on Boolean matching and the term *structural mapping* for a mapping procedure based on pattern matching.

Boolean matching

In technology mapping, delay, power, and area can be minimized by exploiting different configurations of cells based on the $\mathcal{NPN}$-equivalence classes [23, 202]. Specifically, permutations increase the number of matches and negations play a crucial role in the insertion of inverters. Boolean matching relates a canonical representation of a Boolean function to a list of cells that can implement it and is typically defined over $\mathcal{NP}(\mathcal{N})$-equivalence. In a Boolean mapper, Boolean matching is performed during the cut enumeration phase of mapping where a set of cells are associated to a cut given its function.

In line with previous work [202], we define a data structure that facilitates Boolean matching, called *Boolean matching library*. Such a library is a hash table that relates the functionality, represented as a truth table, to a set of cells that can implement it. For each cell, the library precomputes all its $\mathscr{NP}-$configurations. Given a function f of a cell, its $\mathscr{NP}-$configurations are all the input permutations and inversions applicable to f that generate functions in the $\mathscr{NP}-$equivalence class of f. Specifically, given a Boolean function to match, the library returns a set of cells in the $\mathscr{NP}-$equivalence class of the function along with their $\mathscr{NP}-$configurations.

Pattern matching

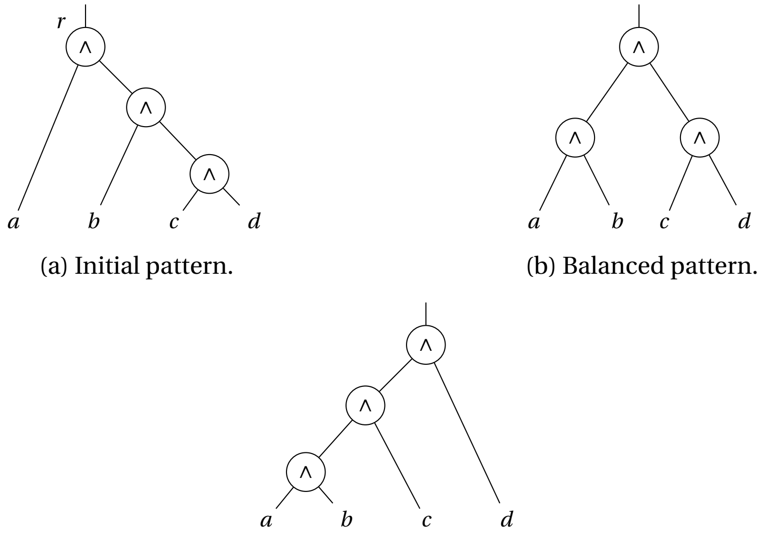
Pattern matching associates a set of cells to a (sub)graph by solving a graph isomorphism problem. A database contains a family of patterns for each cell. In modern technology mapping, a pattern is an AIG representation of a cell's function. A cell can be associated with a sub-graph in the subject graph if one of its patterns matches the sub-graph, i.e., it is structurally equivalent.

In technology mapping, sub-graphs to match are described by cuts and extracted using cut enumeration. When cell's patterns are only trees, pattern matching integrates readily with cut enumeration since the pattern identification algorithm is based on dynamic programming. Specifically, patterns at a 2-input node n can be identified during the cut merging operation by linking two input cut patterns using the top operation of node n. This operation is efficient if the support of the two cuts is considered as disjoint. Considering non-disjoint supports would require generating many complex patterns and checking for reconvergences during the cut merging operation, significantly increasing the computational complexity of pattern matching. This latter complication is not addressed by our work, as Boolean matching compensates for these scenarios.

To enable pattern matching, we first present a method to derive the pattern database. Then, we show how to identify patterns during cut enumeration using pattern indexing.

Structural patterns derivation

In this work, we define patterns of a cell, called *structural patterns*, as *and-inverter trees* representing the disjoint support decompositions of the cell's function. We restrict patterns to include only tree-like structures because this significantly enhances the efficiency of the matching procedure. Moreover, most of the cells in technology libraries are full-DSD, such as multiple-input ANDs and AND-OR gates. A few exceptions include gates whose functions are not full-DSD, such as XORs, MUXs, and majority gates. For these gates, it is impossible to produce a structural pattern since the decomposition inevitably results in a DAG structure. However, non-full-DSD functions can be specifically identified in the subject graph via structural analysis before mapping, a feature not included in our implementation since it is



(a) Initial pattern.

(b) Balanced pattern.

(c) Other redundant pattern.

Figure 4.1: Possible structural patterns for AND4.

addressed by Boolean matching.

Initially, one structural pattern is derived for each full-DSD cell by recursively applying top-down decomposition and bottom-up decomposition for the AND operator, with possible input and output negations. A key observation is that decomposition trees are not canonical, meaning that the same function can be represented by different structural patterns.

Example 4.3.2. *A* 4*-input AND gate can be expressed as* $(a \wedge (b \wedge (c \wedge d)))$*,* $((a \wedge b) \wedge (c \wedge d))$*, or* $(((a \wedge b) \wedge c) \wedge d)$ *resulting in the three structural patterns depicted in Figure 4.1.* ▲

Since matching is performed by comparing derived patterns with sub-graphs extracted from the subject graph, generating multiple patterns for each cells is crucial as it translates into additional match opportunities. Therefore, we employ an algorithm to derive additional patterns from the initial one obtained using DSD. The algorithm generates new structural patterns by applying 3-input associative transformations to the initial pattern. An associative move is a tree rotation applied to 2 neighboring nodes that exhibit the associative property. Algorithm 4.2 outlines the steps to derive multiple patterns. The algorithm recursively applies associative moves to every node of each computed structural pattern and terminates when no new patterns are generated.

Example 4.3.3. *Suppose that the pattern in Figure 4.1a is the one initially generated by the decomposition. Pattern in Figure 4.1b can be obtained from the first one by applying a left rotation to node* r*.* ▲

In order to minimize the number of derived structural patterns, hence limiting the run time for pattern matching, the algorithm filters structural patterns that are symmetric to

Algorithm 4.2: Pattern Derivation

```
Input: Pattern $og\_pat$, Node $start\_node$, Pattern_set $set\_pat$
Algorithm derive($og\_pat$, $start\_node$, $set\_pat$)
    if is_pi($start\_node$) then
        return
    $l \leftarrow$ left_fan-in($start\_node$)
    $r \leftarrow$ right_fan-in($start\_node$)
    if not_negated(l) AND not_pi(l) then
        $r\_pat \leftarrow$ right_move($og\_pat$, $start\_node$)
        if check_symm($r\_pat$, $set\_pat$) then
            add_pattern($r\_pat$, $set\_pat$)
            derive($r\_pat$, $start\_node$, $set\_pat$)
    if not_negated(r) AND not_pi(r) then
        $l\_pat \leftarrow$ left_move($og\_pat$, $start\_node$)
        if check_symm($l\_pat$, $set\_pat$) then
            add_pattern($l\_pat$, $set\_pat$)
            derive($l\_pat$, $start\_node$, $set\_pat$)
    derive($og\_pat$, $l$, $set\_pat$)
    derive($og\_pat$, $r$, $set\_pat$)
```

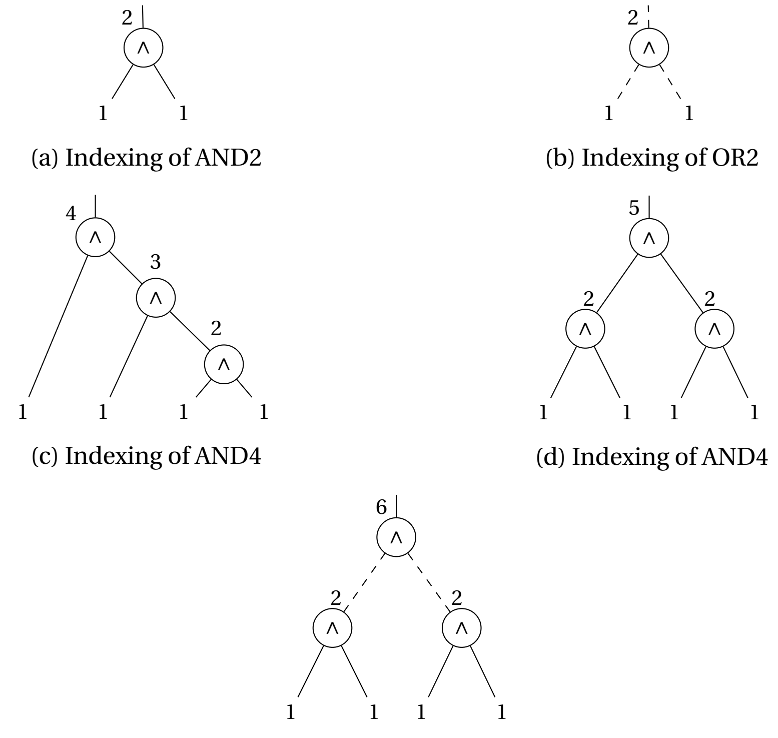
others already found by canonicalizing the order of PIs and AND's inputs. Thus, in Figure 4.1, the pattern of Figure 4.1c would not be produced as it is symmetric to the one of Figure 4.1a (under input permutation).

Pattern indexing and pattern table generation

To identify isomorphic patterns, each node in the structural patterns is assigned to an index. This index serves as a unique identifier for sub-patterns. In practice, if two pattern roots have the same index, it indicates that the patterns are isomorphic.

Patterns are processed in ascending order of size. The procedure indexes nodes of a pattern in topological order. Specifically, each PI is assigned to an index of 1. To other nodes, the index is uniquely assigned based on the input indexes and polarities (structural hashing). The order of the inputs is canonicalized for permutation to remove symmetries (e.g., an AND between 1 and 2 is equivalent to an AND between 2 and 1). This procedure neglects the presence of negations and permutations at the PIs and the PO of a pattern. Hence, isomorphic structures in an $\mathcal{NPN}$-equivalence class share the same root index. For the rest on internal connections, a negative polarity edge in the graph negates the index of the substructure.

Example 4.3.4. *To illustrate the indexing procedure, let us consider the simple cell library shown in Figure 4.2, composed of* 2*-input AND (AND2),* 2*-input OR (OR2),* 4*-input AND (AND4), and* 4*-input AND-OR-Inverted (AOI22) cells. The first pattern to be indexed is the one associated with the AND2 cell. The index* 2 *is assigned to its only node. Afterward, index* 2 *is also assigned to the pattern corresponding to the OR2 cell since the same structure has already been observed (the*



(a) Indexing of AND2

(b) Indexing of OR2

(c) Indexing of AND4

(d) Indexing of AND4

(e) Indexing of AOI22

Figure 4.2: Indexing of patterns.

input and output negations are ignored). Next, the pattern in Figure 4.2c is processed assigning new indexes for AND3 (index 3*) and AND4 (index* 4*). Then, in Figure 4.2d another pattern of the AND4 cell is elaborated leading to index 5. Finally, in Figure 4.2e the pattern for the AOI22 is indexed. Unlike the pattern in Figure 4.2d, the AOI22 pattern features a top AND gate with negated input edges, resulting in a different index.* ▲

This procedure naturally exposes the relationship between structures and sub-structures. For instance, from the previous example, we can conclude that an AND4 can be described as the AND between two AND2s or a PI and an AND3. From this information, we generate a hash table that expresses for each structure and substructure how they can be obtained by ANDing smaller substructures. Such a table is called *pattern table*. An (i, j) entry in the table identifies the index of a pattern having pattern i and j (with $i \geq j$ for canonicity) connected to an AND at the top. The indexes i and j can be negative, representing a connection with negated polarity such that, for instance, function $a \wedge (b \vee c)$ is recognised as different from $a \wedge (b \wedge c)$ even if $b \vee c$ and $(b \wedge c)$ have the same pattern index. Thus, each entry depends on the index and polarity of the left node, and index and polarity of the right one, except for PIs. Additionally, we generate another hash table, named the *index table*, which relates the pattern indexes representing structural patterns to corresponding cells. Tables 4.1a and 4.1b show the pattern table and index table for the patterns shown in Figure 4.2.

Indexes	-2	1	2	3	4	5	6
-2	6	-	-	-	-	-	-
1	-	2	3	4	-	-	-
2	-	3	5	-	-	-	-
3	-	4	-	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-

(a) Pattern Table

Indexes	**Gates**
2	AND2; OR2
4	AND4
5	AND4
6	AOI22

(b) Index Table

Table 4.1: Pattern and Index Table

Pattern matching in cut enumeration

During cut enumeration, each cut is assigned to a pattern index that identifies the underlying pattern covered by the cut. Differently from Boolean matching, this operation does not require computing the function of the cut. Instead, the pattern index is computed during the cut merging using the pattern table. Specifically, the pattern index of a cut c, obtained by merging two cuts u and v, is assigned by looking in the pattern table for an entry with the pattern of u and pattern v as fan-ins along with the polarity of the connection. Initially, a trivial cut is associated with the pattern index 1. Then, cuts are computed in topological order and patterns are assigned.

Given a cut and its associated pattern index, pattern matching retrieves the set of cells using the index table. Since the cells are in the $\mathscr{NPN}-$equivalence class, correct permutation, and negations are applied during mapping.

4.3.3 Mapping with Hybrid Matching

In this subsection, we present our main contribution: *hybrid matching*, a matching algorithm that combines the Boolean and pattern matching techniques presented in Section 4.3.2, addressing the shortcomings of both strategies and achieving better quality-of-results. We first summarize the advantages and disadvantages of Boolean and pattern matching to then present the algorithm for hybrid matching.

Boolean vs. structural

Boolean matching typically yields results of superior quality compared to pattern matching for a few reasons. First, Boolean matching is not restricted to full-DSD functions. Second, Boolean matching inherently removes structural redundancies present in the subject graph. However, Boolean matching is also typically slower because it requires computing the Boolean function and matching in the $\mathscr{NP}-$equivalence class. Boolean matching in $\mathscr{NP}-$equivalence is typically addressed by enumerating all the $\mathscr{NP}-$configurations of the cells as explained in

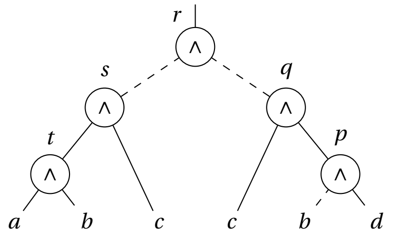


Figure 4.3: Subject graph with functional redundancy.

Section 4.3.2. This procedure may generate up to $n! \cdot 2^n$ configurations for a cell with n inputs. This has two consequences. First, a Boolean library contains many configurations, leading to more matches and larger mapping time. Second, $\mathcal{NP}$−matching scalability is limited to cells up to 6-inputs.

Although pattern-matching results are generally of inferior quality, the matching time is significantly shorter. Indeed, pattern matching supports larger cells because its matching complexity depends on the number of patterns generated which is typically small for full-DSD functions after filtering for symmetries. Informally, the number of minimum-size patterns for large full-DSD cells is significantly lower than the $\mathcal{NP}$-configurations needed by Boolean matching. Additionally, run time also benefits from the reduced number of matches per cut compared to Boolean matching. Because of this, the technology mapping algorithm is faster during selection. Moreover, despite Boolean matching's capability to detect functional redundancy in the subject graph, pattern matching can sometimes produce better results.

Example 4.3.5. *Using an example observed in our experiments, let us suppose that the subject graph presents the structure of Figure 4.3 and that the ASAP 7nm cell library [44] is employed. The structure presents functional redundancies and implements the following Boolean function:* $(\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{c} \vee b \vee \overline{d})$. *Since pattern matching is purely based on the structure, it ignores redundancies and matches the subject graph to the 6-input OA33 cell, which implements the following Boolean expression:* $(a \vee b \vee c) \wedge (d \vee e \vee f)$. *Conversely, Boolean matching detects a functional support of* 4 *variables and consequentially searches for 4-input cells to match. However, due to variable b being binate (present in two polarities), such a cell cannot be found, leading to a mapping that employs an AND3 and an AOI31 cell. This causes lower quality of results for area and potentially for delay compared to pattern matching.* ▲

Integrating Boolean and pattern matching

Initially, both pattern and Boolean libraries are generated as described in Section 4.3.2. In hybrid matching, two distinct phases of cut enumeration are performed, one for pattern matching and another for Boolean matching. Algorithm 4.3 shows the algorithm to compute cuts and matches. First, cuts for pattern matching are enumerated and matched for every node of the subject graph. Subsequently, for every node of the subject graph, cuts for Boolean matching are enumerated and matched. However, at this step, the algorithm joins the set of

Algorithm 4.3: Hybrid Matching

```
Input: Boolean library $bool\_lib$, pattern library $pat\_lib$, Boolean network $N$, cut size
       Boolean $k$, cut size pattern $l$, bool $do\_pattern$, bool $do\_bool$, Cut limit $p$
Output: Match Set $match\_set$
if $do\_pattern$ then
    foreach $node\ n \in N$ do
        $cut\_pat[n] \leftarrow$ pattern_cuts_merge($cut\_pat, n, l, p$)
        $match\_pat[n] \leftarrow$ pattern_match($cut\_pat[n], pat\_lib, p$)
if $do\_bool$ then
    foreach $node\ n \in N$ do
        $cut\_bool[n] \leftarrow$ bool_cut_merge($match\_set, n, k, p$)
        $match\_bool[n] \leftarrow$ bool_match($cut\_bool[n], bool\_lib, p$)
        $match\_set[n] \leftarrow$ union($match\_pat[n], match\_bool[n], p$)
return $match\_set$
```

Boolean and structural cuts together with their matches. The cut merge operation follows the recursive definition of Equations 2.5 and 2.6 to generate new cuts. The collected cuts are organized according to the priority cuts paradigm [47] for which a limited number of cuts are saved for each node and are sorted according to a cost function depending on delay, area, and size. Only a limited number of p best cuts are selected based on their delay and area flow under the unit model. This is a crucial point in the algorithm, as this procedure allows us to combine the best results of both Boolean and pattern matching, overcoming the respective shortcomings. Note that Boolean cuts are computed starting from the merged cut sets. Additionally, hybrid matching can optionally perform only Boolean or pattern matching, by selecting the parameters do_bool and do_pat.

Algorithm 4.3 requires two distinct phases of cut enumeration and matching since cuts for Boolean or pattern matching have different characteristics. Cuts for Boolean matching require the computation of the truth table, are limited to 6 inputs, and have redundancies removed in the support. Contrarily, cuts for pattern matching need only a pattern index, are canonicalized on symmetries, and keep functional redundancies. Given these differences, the two types of cuts are incompatible during the cut merging operation.

4.3.4 Experimental Results

In this subsection, we evaluate the performance of hybrid matching in a technology mapper. We compare it to the state-of-the-art Boolean mapper *map* implemented in *ABC*[3], which represents the baseline of our experiments. Additionally, we show the results of our mapper in three settings: (i) using only Boolean matching; (ii) using only pattern matching; and (iii) using hybrid matching.

The Boolean, structural, and hybrid matching methods have been implemented in C++17

[3] Available at: https://github.com/berkeley-abc/abc

Table 4.2: Results for Boolean, structural, and hybrid matching for delay-oriented technology mapping and comparison against ABC *map*.

Benchmark	**ABC *map***		**Boolean**		**Structural**		**Hybrid**	
	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)
adder	92.53	2577.43	84.03	2573.43	87.96	2573.79	84.03	2573.43
bar	325.47	168.08	356.23	168.08	218.84	157.72	216.32	172.61
div	5336.99	43765.09	4982.71	43765.25	4410.34	44048.23	4673.59	43836.68
hyp	16395.1	195822.72	14958.07	195345.77	15620.63	195788.75	15237.73	195276.66
log2	2177.57	3955.63	1908.24	3848.17	2022.71	4109.63	1737.34	3916.17
max	226.59	2213.23	209.22	2213.23	192.53	2257.11	172.77	2228.59
multiplier	1954.48	2738.95	1916.27	2667.36	1881.62	2726.37	1654.58	2649.57
sin	431.22	1814.85	446.42	1760.82	433.41	1852.61	407.40	1792.56
sqrt	1767.89	47442.32	1681.05	47438.44	1762.06	47769.50	1683.20	47438.60
square	1193.12	2516.39	1070.08	2505.10	1151.12	2521.89	1084.48	2503.20
arbiter	766.68	898.75	766.50	898.75	766.50	898.75	766.32	898.75
cavlc	41.57	187.04	38.80	187.04	37.32	186.44	37.52	186.07
ctrl	8.58	102.49	7.86	102.49	9.97	120.31	7.85	101.21
dec	30.83	65.72	30.83	65.72	30.11	66.19	27.44	66.15
i2c	78.65	182.65	73.69	182.65	69.11	184.57	69.38	182.65
int2float	13.93	181	12.95	181.00	11.84	181.17	11.31	182.22
mem_ctrl	2763.83	1103.42	2595.75	1100.46	2439.27	1130.11	2427.28	1104.37
priority	87.28	2501.95	82.37	2501.95	82.08	2510.03	82.25	2501.95
router	19.59	280.01	18.57	274.05	20.17	278.83	18.41	274.05
voter	1506.14	804.96	1519.56	749.36	1447.60	790.84	1538.53	755.33
Total time (s)		32.19		20.08		4.15		14.95
Ratio	1.000	1.000	0.953	0.991	0.934	1.011	**0.889**	0.994

and are available in the open-source logic synthesis framework *Mockturtle*[4] [183] in the command `emap`. For the experiments, we use the EPFL combinational benchmark suite [3] containing combinational circuits in the form of AIGs. All the results were verified using the combinational equivalent checker in *ABC*. We employ the ASAP7 7nm cell library [44], pre-processed by OpenLane[5]. For every benchmark, we provide the area and delay results and the total run time. The baseline has been obtained by applying the area-driven balancing algorithm available in *Mockturtle*. The maximum cut size for Boolean matching is 6, and for pattern matching is 9. Moreover, a maximum of 16 cuts are stored for each node.

The results for delay-oriented mapping are shown in Table 4.2. The hybrid mapper achieves an average area improvement of approximately 11.1% compared to the mapper in ABC, 6.5% compared to our Boolean mapper, and 4.5% compared to our structural mapper. This improvement is due to the use of both Boolean and pattern-matching cuts during mapping. For instance, in benchmarks such as *bar*, *max*, and *i2c*, large cuts extracted using pattern matching significantly reduce the area compared to the Boolean approach. Conversely, for benchmarks such as *hyp* and *voter*, the employment of XOR and Majority cells extracted by Boolean matching results in considerably better delay outcomes compared to the structural mapper. The delay results are generally similar between the hybrid and Boolean mappers for most benchmarks. However, for a few benchmarks, the hybrid mapper experiences slightly worse delay values, leading to an small average delay increase over the Boolean mapper. This is primarily because the hybrid mapper generates more cuts through pattern matching, most

[4] Available at: https://github.com/lsils/mockturtle

[5] Available at: https://github.com/The-OpenROAD-Project/OpenLane

Table 4.3: Results for Boolean, structural, and hybrid matching for area-oriented technology mapping and comparison against ABC *map -a*.

Benchmark	**ABC *map -a***		**Boolean**		**Structural**		**Hybrid**	
	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)
adder	57.4	3548.84	57.4	3548.84	71.39	2813.73	57.4	3548.84
bar	191.81	238.53	155.19	228.93	128.74	200.57	128.74	199.39
div	3427.1	66322.78	3232.36	52243.49	3106.12	55875.19	3079.58	53745.74
hyp	14378.94	300243.28	13221.39	244958.55	14303.22	405054.5	13167.04	272523.78
log2	1643.99	6734.09	1499.35	5196.26	1619.63	7944.92	1501.99	5409.33
max	166.18	2862.23	152.23	2956.82	145.55	3830.87	139.06	2952.58
multiplier	1495.54	4961.61	1283.65	3302.57	1421.02	5675.77	1284.5	3448
sin	309.11	3056.93	277.14	2743.47	286.86	3545.27	270.05	2782.29
sqrt	1461.52	106239.08	1343.39	102465.89	1337.51	102273.54	1336.59	102003.83
square	1168.02	3711.58	1052.22	3483.91	1108	3079.15	1048.11	3508.29
arbiter	569.4	1018.69	557.72	999.87	557.72	1015.47	557.72	999.87
cavlc	39.01	221.56	35.53	223.11	34.16	247.51	34.13	263.16
ctrl	8.07	131.57	7.65	125.09	9.3	154.49	7.2	127.51
dec	27.5	85.83	27.73	90.42	28	90.46	27.06	86.33
i2c	77.9	214.29	71.42	258.99	67.33	269.73	67.16	267.86
int2float	13	205.02	12.05	200.71	10.99	202.35	11.04	197.6
mem_ctrl	2673.88	1799.5	2452.88	1834.53	2280.18	1724.13	2278.47	1706.2
priority	62.56	2795.01	57.65	4153.73	51.45	2943.53	51.17	2918.99
router	15.01	448.06	12.78	406.72	13.65	394.49	12.96	400.89
voter	851.87	1297.25	792.49	1101.06	840	1265.82	802.13	1185.61
Total time (s)		31.39		17.04		3.98		14.90
Ratio	1.000	1.000	0.919	0.964	0.936	1.045	**0.885**	**0.953**

of which are area oriented. As a result, cut filtering rules may more likely discard cuts that are useful for delay optimization. Despite this minor negative impact, the overall area reduction, which peaks at 39% in the case of *bar*, significantly outweighs it. Regarding run time, the hybrid mapper achieves an average speedup of 1.25× compared to the Boolean mapper. This improvement is mainly due to the use of structural cuts, which limits the number of Boolean cuts computed at each node. Additionally, larger structural cuts generally present fewer matches than smaller ones, thus reducing the number of matches per node and decreasing the runtime during *selection*. Indeed, the structural mapper achieves the lowest run time.

The results for area-oriented mapping are shown in Table 4.3. Similar considerations apply with the only difference of an area and delay improvement of 3.8% and 0.1%, respectively, compared to the Boolean mapper.

4.4 Technology Mapping Using Multiple-output Cells

Cell libraries, such as *standard cells*, define a set of pre-designed and pre-characterized primitives that are used as building blocks to create digital circuits. Typically, libraries contain simple cells (e.g., a NAND2), complex cells (e.g., a XOR3), multiple-output cells (e.g., a full adder), and sequential elements. Commonly, technology mapping algorithms efficiently exploit technology libraries. However, multiple-output cells are often neglected due to the complexity of detecting and evaluating them in the optimization loops of technology mapping. Only a few frequent multiple-output cells, such as half adders and full adders, have partial support in industrial tools. These common elements are generally identified in ordinary logic blocks, such as adders, which can be extracted from a *register-transfer level* (RTL) description of digital hardware for which the mapping is known. Nevertheless, synthesis flows often decompose these cells to meet timing constraints. Consequently, it is crucial to re-detect and map multiple-output cells starting from a gate-level description to recover area and power consumption. While practical solutions based on LUT merging techniques have been proposed for FPGAs [81, 128, 210], mapping to multiple-output cells for standard-cell-based designs remains an open problem.

Generalized matching (GM) [23] has been introduced as a multiple-output matching technique that supports concurrent matching to multiple single-output cells or a multiple-output cell. In [22], the authors propose an incremental remapping technique that utilizes GM on local small windows of already mapped logic. Their method supports multiple-output cells and evaluates substitutions symbolically by solving a minimum-cost GM problem using *binary decision diagrams* (BDDs) [32] and *algebraic decision diagrams* (ADDs) [15].

The tool Yosys [211], which is part of the RTL to GDSII toolchain OpenLane, integrates a limited support of half and full adder cells. Yosys identifies adder cells from RTL or by performing circuit analysis. Adder cells are kept as *don't touch white boxes* during logic optimization and technology mapping. Hence, technology mapping is oblivious to multiple-output cells resulting in a degradation of the potential delay, power, and area of the design.

In this work, we describe an alternative method to increase the support of multiple-output cells in the optimization loops of a technology mapping algorithm. Primarily, we address scalability since multiple-output cells substantially increase the complexity of mapping to an intractable level. Our contributions include a fast multiple-output cell detection methodology, an extension of Boolean matching, and a formulation of global and local area recovery heuristics that supports multiple-output cells. In contrast to [22], we tackle the global technology mapping problem instead of local remapping. This approach has the advantage of selecting cells and optimizing for inverters globally while meeting delay constraints without iterating through many incremental steps. Instead of BDDs and ADDs, we pre-compute a library to facilitate Boolean matching that can be rapidly accessed through canonicalization and hashing. Moreover, we use a cut-based method to enumerate multiple mapping options and cell selections. In area recovery, we employ a generalization of *area flow* [40, 124] and *exact area* [140] to evaluate single- and multiple-output cells.

Our implementation is open-source and available in the library *Mockturtle*. To the best of our knowledge, this is the first open-source implementation of a technology mapper for *standard cells* that integrates the support for multiple-output cells.

In the experiments, we evaluate our approach using the ASAP7 cell library [44], which contains the *half adder* and *full adder* cell. We compare our mapper against ABC showing a 7.48% area reduction on average when mapping for the minimal delay. When the delay is not constrained, our approach obtains a considerable area reduction of 7.42%. Furthermore, our method reduces the area by 5% on average compared to the method in Yosys in which adder cells are considered as white boxes during technology mapping. Our approach demonstrates scalability to large networks with an average run time increase of 8%. We consider this overhead fairly limited since large circuits are mapped in a few seconds. Finally, we discuss the usage of other multiple-output cells.

4.4.1 Extraction of Multiple-output Cells

In this subsection, we present a method to identify multiple-output cells in a Boolean network. This process requires: (i) an elaboration of the cell library to be suitable for Boolean matching; (ii) a multiple-output Boolean matching method; and iii) a multiple-output cut computation procedure.

Matching library generation

Given a cell library, we define a data structure, called *matching library*, that facilitates fast Boolean matching. The matching library links a Boolean function represented as a truth table to a set of cells that implements it.

In technology mapping, delay, power, and area can be minimized by exploiting different configurations of cells based on the $\mathscr{N}\mathscr{P}\mathscr{N}$-equivalence classes [38, 202]. Specifically, permutations increase the number of matches, and negations play a crucial role in the insertion of inverters.

For single-output library cells, the matching library is generated similarly to [202] by enumerating cell configurations based on $\mathscr{N}\mathscr{P}$-equivalence classes. Hence, given a function f to match, the matching library returns a set of cells in the $\mathscr{N}\mathscr{P}-$equivalence class of f. Each one of them has attached an $\mathscr{N}\mathscr{P}$-configuration so that its functionality matches f. Since our implementation matches in two polarities (complemented and uncomplemented), the output inversion is not considered at this stage.

For multiple-output cells, the procedure is more involved. The functionality of a multiple-output cell is defined through a set $\mathscr{R}$ of functions, one for each output pin. Due to this higher degree of freedom compared to single-output cells, the multiple-output matching library utilizes two additional operators $\mathscr{N}_O$, representing output negations, and $\mathscr{P}_O$, representing output permutations. First, the classification of cells in $\mathscr{N}\mathscr{P}-$equivalence classes is extended such that each configuration of input negations and permutations is applied concurrently to the functions in $\mathscr{R}$. Second, to have a fast matching, the $\mathscr{N}\mathscr{P}-$configuration of the multiple-output cell is canonicalized. This is achieved in two steps. First, each individual output function is negated to be *normal*, i.e., a Boolean function f is *normal* if $f(0,0,\ldots,0) = 0$. Second, a multiple-output function is canonicalized by sorting output functions in lexicographical order.

Example 4.4.1. *Let us consider a canonicalization example on a half adder cell described by the functionality set* $R = \{$“0110",“1000"$\}$, *which consists of a XOR2 and an AND2 function and is represented using truth tables. Let us select a configuration of the cell composed of negations over the input variables* $\mathcal{N}_I = \{x_0 x_1 \rightarrow \bar{x}_0 x_1\}$ *and no permutations* $\mathcal{P}_I$. *After applying the input* $\mathcal{N}\mathcal{P}$ *operators, the resulting function is* $\{$“1001",“0100"$\}$. *Finally, the configuration is canonicalized leading to the multiple-output function* $\{$“0100",“0110"$\}$ *where the XOR2 has been normalized and the outputs have been permuted in lexicographical order. The corresponding output operators are* $\mathcal{N}_O = \{o_0 o_1 \rightarrow \overline{o}_0 o_1\}$ *and* $\mathcal{P}_O = \{o_0 o_1 \rightarrow o_1 o_0\}$ *(the negation is applied before the permutation).* ▲

multiple-output cells are represented as a set of single-output cells each corresponding to an individual output pin. We refer to these cells as *virtual output-pin* (VOP) cells. VOP cells describe the pin-to-pin delay relation and the functionality of each output pin. In our method, we compute an area contribution associated with each VOP cell. Typically, functions of VOP cells are also offered by single-output cells in a technology library. For instance, a full adder cell is covered by a MAJ3 and a XOR3 cell. In this case, the area contribution of a VOP cell is computed by proportionally scaling the area of each VOP cell implemented as a single-output cell of the library such that the sum over all VOP cells equals the area of the multiple-output gate. If some VOP cells are not offered by the cell library, a fixed scaling factor is employed. The area contribution is used for area estimations during technology mapping, in particular during the exact-area phase.

Example 4.4.2. *Given the cells HA (half adder), AND2, and XOR2 with an area of* 2.3, 1.3, *and* 2.0, *respectively, the VOP cells area contribution is of* 0.9 *for the AND2 pin and* 1.4 *for the XOR2 pin.* ▲

Multiple-output Boolean matching

Boolean matching assigns to a function a set of gates that can implement it. In technology mapping, Boolean matching is performed on local regions of a network defined by *cuts*. For single-output cells, matching consists of a simple look-up of the cut function in the matching library. Similarly to [38, 202], our approach matches while considering two polarities for each gate (uncomplemented, complemented) to enable better logic sharing of inverters or to avoid additional inverter delay costs.

For multiple-output cells, matching assigns a multiple-output cut to a set of VOP cells. The matching is achieved in two steps. First, the functions of the cut roots are normalized and permuted to be canonical according to the matching library rules. Second, the canonical function is looked up in the matching library. The output negations and permutations are then adjusted to assign the individual VOP cells to the corresponding cut roots.

Algorithm 4.4: Multiple-output cells detection

```
Input: Subject graph N, Maximum k, Maximum l, Matching library lib
Output: Set of multiple-output cuts multi_cuts
cuts ← enumerate_cuts(N, k);
/* filter cuts based on individual multiple-output cells */
cuts ← filter_match_cuts(cuts, lib);
/* hash the cuts based on the leaves */
cuts_h ← hash_cuts(cuts);
/* combine cuts sharing the same leaves and match */
multi_cuts ← combine_cuts(cuts_h, l, lib);
/* remove incompatible kl-cuts */
multi_cuts ← filter_multi_cuts(multi_cuts);
return multi_cuts
```

Multiple-output cut computation

The multiple-output cut computation may require significant run time since the number of cuts grows significantly with respect to the number of nodes in a Boolean network. KL-cuts [128] is an algorithm that can be used to generate generic multiple-output cuts. However, some specific cells, such as adder cells, can be identified using a much simpler methodology. Hence, in this section, we propose a multiple-output cut enumeration that considers a class of cells in which each output has all the cut leaves in its support. These cuts describe cells such as half adder and full adder and can be extracted very rapidly throughout a Boolean network.

Algorithm 4.4 presents a high-level view of the steps to enumerate and match multiple-output cuts. The inputs are a subject graph N, a maximum cut size k, a maximum cut merging value l, and the matching library lib. First, k-feasible cuts are enumerated for every node of the network and the associated function is computed. Second, filtering rules are applied to reduce the number of cuts to combine. Specifically, we select only cuts whose function is $\mathcal{NPN}$-equivalent to a VOP cell function, i.e., the cut may be part of a matchable multiple-output cut. Next, cuts are arranged in groups such that each cut in a group shares the same leaves. This is achieved using a fast algorithm that hashes the leaves of cuts. Then, cuts in each group are combined up to l outputs and directly matched. Matched multiple-output cuts are added to a list. Alternatively to cut hashing, kl-cuts [128] can be used to generate multiple-output cuts.

While combining cuts, filtering rules are employed to remove *partially dangling* multiple-output cuts. Specifically, a multiple-output cut with a set of roots $\mathcal{L}$ is partially dangling if $\exists n \in \mathcal{L}$ s.t. $n \in \text{MFFC}(\mathcal{L} \setminus n)$. Informally, a partially dangling multiple-output cut has an output pin that cannot be connected externally since it is only used internally in the cut.

The last step of Algorithm 4.4 further filters cuts to be *compatible*. Two multiple-output cuts C_i and C_j having the set of roots $\mathcal{L}_i$ and $\mathcal{L}_j$ are *incompatible* if $\mathcal{L}_i \cap \mathcal{L}_j \neq \emptyset$ and $\mathcal{L}_i \neq \mathcal{L}_j$. This filtering rule selects multiple-output cells making sure they do not overlap if they do not share the same outputs. This constraint is crucial to limit the run time of technology mapping,

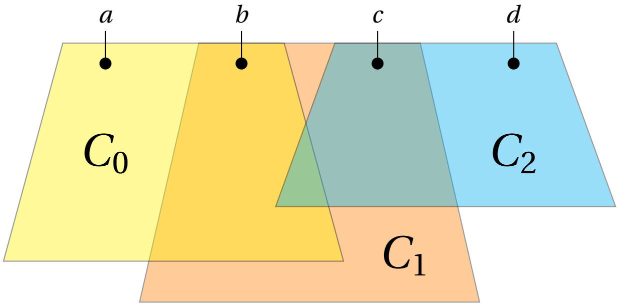


Figure 4.4: Example of compatible and incompatible cuts

which would require undoing and re-evaluating many mapping choices with an exponential increase.

Example 4.4.3. *Figure 4.4 shows three multiple-output cuts C_0, C_1, and C_2 with outputs $\mathscr{L}_0 = \{a, b\}$, $\mathscr{L}_1 = \{b, c\}$, and $\mathscr{L}_2 = \{c, d\}$. Cuts C_0 and C_2 are compatible since $\mathscr{L}_0 \cap \mathscr{L}_2 = \emptyset$. Cuts C_0 and C_1, and also C_1 and C_2, are instead incompatible. Hence, our algorithm removes, for instance, C_1 from the list of multiple-output cuts.* ▲

Since cells that can be matched to incompatible cuts are not very common in designs for typical cell libraries (that contain very few multiple-output cells), we argue that the loss in quality deriving this filtering rule is limited. On the other hand, if we extend cell libraries to contain multiple-output cells derived from the compression of random logic, the presented methodology would remove many possible matches affecting the potential quality. However, when multiple-output mapping is used for re-mapping a small window of logic, the incompatible multiple-output cell choices can be enumerated with a limited run time overhead.

4.4.2 Technology Mapping using Multiple-output Cells

In this subsection, we present how a technology mapping algorithm can be extended to support multiple-output cells. Specifically, our method maps to multiple-output cells only when the design cost improves compared to using single-output cells. Moreover, it handles the optimization of inverter cells across multiple-output gates and delay minimization.

Algorithm 4.5 shows the high-level pseudo-code of the mapper. All the steps are analyzed in detail in this subsection. First, the multiple-output cuts are computed and matched using Algorithm 4.4. If multiple-output cells have been already detected, e.g., they have been extracted from a *register-transfer level* (RTL) hardware description, they can be added at this step as multiple-output cell choices for the mapper. Next, the network is sorted in a specific topological order guided by multiple-output cuts. This is necessary to map the whole network in one forward and backward pass. Then, single-output cuts are computed using priority cuts [141]. At this step, our algorithm checks that the enumerated cuts contain also the single-output sub-cuts of the multiple-output ones. This is desirable in some cases to reduce the

Algorithm 4.5: Technology mapping algorithm

```
Input: Subject graph $N$, Maximum $k$, Maximum $l$, Matching library $lib$, cost function $C$
Output: Mapped network $M$
/* enumerate and match multiple-output cuts */
$multi\_cuts \leftarrow$ compute_multioutput_cuts($N$, $k$, $l$ $lib$);
/* compute the constrained topological order */
$topo\_order \leftarrow$ constrained_topo_order($N$, $multi\_cuts$);
/* compute and match single-output cuts */
$cuts \leftarrow$ compute_cuts($N$, $k$, $lib$, $multi\_cuts$, $C$);
/* cover the network and refine the mapping */
$M \leftarrow$ cover($N$, $topo\_order$, $cuts$, $multi\_cuts$, $lib$, $C$);
return $M$
```

impact of unmapping multiple-output cells (a trivial decomposition can be used). Finally, the mapper covers the network by selecting a subset of cuts and associated cells.

Technology mapping consists of several iterations of mapping and covering. A mapping pass selects a candidate cell based on a cost function at each node. Covering extracts a complete mapping solution selecting a reachable subset of the cells starting from the POs. First, our implementation performs a round of delay-oriented mapping that selects for each node in topological order the cell with the smallest arrival time. This round finds the worst-case delay at the outputs and identifies critical paths. Following iterations have the objective of reducing area and/or power subjected to the delay constraints. In our technology mapper, we employ *area flow* to globally optimize for the area and *exact area* to locally refine the cover for area or power. While delay and *area flow* rounds are carried bottom-up (in topological order), exact area is carried top-down (in reverse topological order). Instead of propagating arrival times forward, exact area rounds propagate required times backward to select the candidate cells.

Constrained topological order

Technology mapping is generally a fast algorithm with a time complexity that is linear with respect to the number of nodes in the network (under the priority cuts paradigm [47]). Ideally, we want to maintain the same complexity also when mapping multiple-output gates. In technology mapping, as for many other synthesis algorithms, networks are stored in topological order to guarantee that when a node is processed, the nodes in its *transitive fan-in* (TFI) have already been processed. For instance, this supports efficient propagation of arrival times while mapping. Our algorithm, presented in the next sub-section, follows this practice, i.e., nodes are mapped in topological and reversed topological order.

Example 4.4.4. *Let us consider a network and a topological order $\mathcal{T}$. Let us select three arbitrary nodes p, q, and t in $\mathcal{T} = \{0,\dots,p,\dots,q,\dots,t,\dots,m\}$ such that $p \in TFI(q)$, $q \notin TFI(t)$, and there is a 2-output cell that can implement a multiple-output cut rooted in p and t. Initially, all the nodes preceding t in the topological order, including p and q, are mapped using single-output* *cells. Next, p and t are mapped using the 2-output cell. Consequently, the previous arrival time computed at q and used at q to compute the best match may be invalid since node p in q's TFI changed the mapping. Moreover, the new mapping may unlock a different and more suitable mapping choice at node q. Thus, an algorithm that maps to multiple-output gates might have to re-map some of the nodes between roots of multiple-output cells in the topological order. However, we could avoid re-processing node q by picking a different topological order $\mathcal{T} = \{0, \ldots, p, \ldots, t, \ldots, q, \ldots, m\}$, where t precedes q.* ▲

Our algorithm performs the topological ordering by positioning multiple-output roots "close" to each other. In particular, given two roots p and t of a multiple-output cut with $t > p$ in the topological order, it is always possible to have them topologically next to each other if $p \notin \text{TFI}(t)$ or $p \in \text{fan-ins}(t)$. Informally, if there is a path longer than 1 that connects two roots, there is at least a node that is between p and t in the topological order. In our topological sorting algorithm, when a node is a root of a multiple-output cut, first the TFI of all the roots is visited, then the roots are stored close to each other. This algorithm has the same computational complexity as a depth-first search.

Node mapping

Typically, technology mapping is carried out for several rounds to refine the cover according to specific cost functions. The first round is usually delay oriented with the objective of identifying the worst-case delay and critical paths. The following rounds optimize for area and/or power and are constrained on the maximum allowed delay. Mapping passes are commonly carried out in a bottom-up fashion, while the cover selection and the required time propagation are carried out in a top-down fashion.

Algorithm 4.6 shows a forward pass to map nodes that can be found in a delay or area flow round. First, the required times at each node are computed from the previous round if available. Then, each node is mapped in topological order. Node mapping works by initially covering the two polarities (complemented and uncomplemented) using single-output cuts. The best-fitting cell is chosen based on the selected cost function (e.g., delay) and the required time. If the latter allows it, only one polarity is implemented and the other is realized through an output inverter. Finally, multiple-output-cell mapping is performed on the nodes that belong to a multiple-output cut. However, multiple-output mapping is evaluated for all the roots only when the highest-index root (with respect to the topological order) is processed. In this way, we ensure that all the roots of the cut have previously been mapped using single-output cells. This is necessary to compute accurate costs by comparing multiple-output cell implementations to single-output ones. A multiple-output cell is selected if it improves the cost function. If the multiple-output mapping is successful, some nodes in the topological order among the roots of the multiple-output cut might need to be remapped. This is to ensure that correct arrival times are propagated and that the best cell choices are made. Anyway, this step is often unnecessary if the multiple-output roots are not leaves of a selected cut in the

Algorithm 4.6: Node mapping pass

```
   Input: Subject graph $N$, Mapping $M$, Topological order $topo\_order$, Cuts $cuts$, Multiple-output
          cuts $multi\_cuts$, Matching library $lib$, cost function $C$
   Output: New mapping $M'$
 1 /* compute the required time */
 2 $req \leftarrow$ compute_required($N$, $M$, $lib$)
 3 $M' \leftarrow$ empty_mapping($N$);
 4 foreach Node $n \in topo\_order$ do
 5     /* map node using single-output cells */
 6     map_positive_polarity($M'$, $n$, $cuts(n)$, $lib$, $C$, $req$);
 7     map_negative_polarity($M'$, $n$, $cuts(n)$, $lib$, $C$, $req$);
 8     /* try to drop one polarity if there is enough slack */
 9     select_polarity($M'$, $n$, $req$, $C$)
10     /* highest index output-pin of a multiple-output cut */
11     if $multioutput\_root(n)$ then
12         map_multioutput($M'$, $n$, $multi\_cuts$, $cuts$, $lib$, $C$, $req$)
13         remap_conflicts($N$, $M'$, $n$, $cuts$, $lib$, $C$, $req$)
14 return extract_cover($M'$)
```

current mapping. Finally, a cover is extracted starting from the POs in reverse topological order and the mapping is returned.

During the delay and area flow rounds, the mapping is improved by picking the best cell implementation at each node according to the selected cost function. The cover is only known after a top-down computation that selects the best cuts starting from the POs and recurs on the leaves. Consequently, after a bottom-up round not all the outputs of the multiple-output cells are guaranteed to be used in the cover. This issue is not addressed during these iterations. Nevertheless, during exact area rounds the cover is known and is locally improved to guarantee that all the output pins of the selected cells are used, else nodes are re-mapped for a more suitable single-output cell implementation. Moreover, exact area rounds are carried out in reverse topological order to increase the likelihood of connecting all the output pins of multiple-output cells. Contrarily to Algorithm 4.6, for exact area rounds, *multioutput_root*(n) is true for the lowest index output-pin of a multiple-output cut.

The node mapping pass complexity is generally linear with respect to the number of nodes in the network. However, conflicts (line 15 of Algorithm 4.6) increase the complexity by having to re-process some nodes in the topological order among the multiple outputs. Anyhow, in practice no more than 0.5% of the nodes are ever re-mapped over the EPFL benchmarks [3].

Multiple-output cell mapping evaluation

We evaluate multiple-output cells by re-mapping simultaneously roots of a multiple-output cut. The pseudo-code during an area flow round is shown in Algorithm 4.7. Initially, all the roots are assigned to single-output cells. We evaluate a multiple-output cell G by assigning to

Algorithm 4.7: Map to multiple-output cells

```
Input: Mapping $M$, Root $n$, Multiple-output cuts multi_cuts, Cuts cuts, Matching library lib,
       cost function $C$, Required time req
Output: Modified mapping $M$
foreach Multiple-output cut $L \in$ multi_cuts(n) do
    foreach Cell configuration $G \in$ lib(L) do
        /* Evaluate multiple-output cell */
        next_cell ← false
        foreach Root $p \in L.roots$ do
            if compute_arrival(M, L, $G(p)$) > req(p) then
                next_cell ← true
                break
        if next_cell then
            continue
        /* Evaluate improvement: area flow and delay */
        if improvement(M, G, L, C) then
            commit_cell($M$, $G$, $L$)
return $M$
```

each root p the corresponding VOP cell ($G(p)$) and individually evaluating them. In particular, the arrival time is checked against the required time. After the arrival time is checked, delay and *area flow* are used to evaluate the improvement and commit a multiple-output cell if it has a better cost. A commit assigns VOP cells to each root. This process is repeated for all available cuts and cells.

The area flow (AF') of a node n involved in a multiple-output cell G and a multiple-output cut L is generalized as follows:

$$AF'(n) = \frac{Area(G) + |L.roots| \times \sum_{l \in L.leaves} AF'(l)}{\sum_{p \in L.roots} Refs(p)} \tag{4.14}$$

where *Refs* represent the estimated references of nodes in the cover. Initially, during the first round of the mapper, *Refs(n)* takes the fan-out count of node n. Every following round it is updated using a linear combination of the previous value and the actual referencing in the cover. Note that if the generalized area flow formula is applied to a single-output gate, the equation reduces to the known formulation. In our implementation, each root of a multiple-output cut references the leaves. This is why the area flow of a multiple-output cut is multiplied by the number of roots. If AF is computed and exact references are known, the sum of AF over the POs gives the area of the cover.

In an *exact area* iteration, the cover is known from previous passes and it is locally improved. Specifically, for each node in the cover exact area chooses a cell such that the area in the MFFC of the node is minimized. This measure is extracted using a recursive cut referencing/dereferencing (Algorithm 4.1).

In exact area passes, a multiple-output cut is evaluated only if all the output pins are referenced in the cover. This removes partially-connected multiple-output cells originating from area flow iterations. Initially, all the roots of a multiple-output cut are assigned to single-output cells. The roots are then recursively dereferenced to measure the exact area of the roots and remove them from the current mapping. A multiple-output cell is then evaluated by checking the delay against the required time and measuring its MFFC area. If the cell replacement is accepted, the multiple-output cut is referenced for each root, or else the previous implementation is restored.

4.4.3 Experimental Results

In this subsection, we present experimental results on using multiple-output cells during technology mapping. We evaluate our method using the ASAP7 cell library [44], which defines two multiple-output cells, namely the *half adder* and *full adder* with inverted outputs. The gain-based delay model, considers these cells slightly slower than the single-output counterparts. For our experiments, we use the EPFL combinational benchmark suite [3] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained by applying the area-driven balancing algorithm available in *Mockturtle*[6] [183].

The mapper has been implemented in C++17 in the open-source logic synthesis framework *Mockturtle* as a command *emap*. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in *ABC*[7].

Comparing against ABC

In this experiment, we test *emap* for delay-oriented mapping against the default technology mapper in ABC (command *&nf -p*). We enable the use of multiple-output cells in our mapper. We use the same settings for cut size and cut limit per node for both mappers.

Table 4.4 shows the results. We evaluate our mapper in terms of the area reduction and the geometric mean of the area and delay with respect to ABC. Our implementation effectively finds multiple-output cells. Specifically, our mapper selects multiple-output gates over non-critical paths to guarantee the minimal arrival time found during the first delay pass. Yosys and state-of-the-art mappers do not support this feature. In some benchmarks such as *adder*, no multiple-output cells are used due to the delay constraints. In fact, multiple-output cells have a higher delay than individual single-output cells in the ASAP7 library. While having a similar or better delay than ABC, our mapper has an average area reduction of 7.48%. For the *hyp* and *square* benchmarks, our mapper improves the area result of ABC by 20%. Our mapper has worse results than ABC for only the *bar* benchmark. This is mainly due to worse

[6] Available at: https://github.com/lsils/mockturtle

[7] Available at: https://github.com/berkeley-abc/abc

Table 4.4: Comparing our mapper against ABC for minimal delay

Benchmark	**Baseline**		**ABC *&nf -p***			**Multiple-output cell mapping**			
	Size	Depth	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Multiple-output	Time (s)
adder	1020	255	1087.54	2520.72	0.06	1052.78	2514.41	0	0.03
bar	3336	12	2512.67	147.76	0.11	2835.74	151.76	0	0.08
div	45050	4405	46156.14	42422.38	1.82	41191.66	41964.47	4	2.11
hyp	214335	24801	197464.66	176397.09	8.16	157949.52	175299.08	12335	14.28
log2	31881	410	21997.83	3589.91	6.46	21612.64	3567.67	507	4.27
max	2865	229	2382.70	2114.45	0.19	2361.95	2106.08	0	0.11
multiplier	26943	266	24013.34	2596.60	1.04	19935.90	2576.03	652	1.56
sin	5383	186	5127.26	1614.09	0.71	5085.07	1583.31	49	0.74
sqrt	18372	6049	20576.44	43721.45	0.60	18066.04	43010.26	513	1.27
square	18264	250	13670.21	2446.85	0.64	10794.21	2440.58	1178	1.06
Reduction						**7.48%**	**0.48%**		-16.92%
Geomean			7614.05			**7284.50**			

cuts at the nodes. In fact, the benchmark does not contain any multiple-output cells defined in the ASAP7 cell library. The experiment shows a 16.92% run time overhead compared to ABC which is reasonable considering that our implementation is generally 9% slower than ABC even when multiple-output cell mapping is disabled.

Comparing to a two-step approach

In this experiment, we consider area-oriented mapping without delay constraints. We propose three flows. The first one performs a vanilla area-oriented mapping without mapping to multiple-output cells. The second one emulates the approach in Yosys[8]. It is a two-step approach that detects full adders and half adders (command *extract_fa*), saves them as *don't touch white boxes*, and then maps the rest of the logic. We re-implemented the Yosys extraction algorithm following the detection algorithm presented in Section 4.4.1, which offers significantly better run time, scalability to large designs, and results. Additionally, we use our mapper instead of ABC since our implementation typically provides 7% better area compared to *map -a* in ABC for area-oriented mapping (when not using multiple-output cells). The third flow integrates multiple-output detection and selection in technology mapping. We show that managing multiple-output cells in a global technology mapping algorithm outperforms the two-step approach.

Table 4.5 shows the results. We use the same baseline as Table 4.4 to carry out the experiments. We evaluate the flows in terms of area and delay reduction with respect to the vanilla implementation. The two-step approach reduces the area of the vanilla implementation by 2.41% on average. This method is particularly run time efficient since technology mapping is decomposed into two independent steps. The multiple-output approach is the best one with an average area reduction of 7.42%, outperforming the two-step approach. The total run time increase is about 8% compared to the vanilla implementation which is reasonable considering the improvement in quality. Furthermore, if we consider only the arithmetic benchmarks (the first 10), our mapper improves the area by 12.7% against the 4.42% of the two-step approach.

[8] Available at: https://github.com/YosysHQ/yosys

Table 4.5: Area-oriented mapping in different settings

Benchmark	**Vanilla mapping**			**Multiple-output detection + mapping**				**Multiple-output cell mapping**			
	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Multiple-output	Time (s)	Area (μm^2)	Delay (ps)	Multiple-output	Time (s)
adder	743.92	3910.79	0.03	596.02	7046.61	128	0.01	551.23	5376.32	128	0.03
bar	2062.66	209.60	0.08	2062.66	209.60	0	0.17	2062.66	209.60	0	0.10
div	29346.85	45188.06	1.91	28251.16	112512.28	3911	1.16	24486.23	90780.33	3904	2.02
hyp	157306.39	225518.08	11.73	156490.92	360506.91	20927	7.93	134586.34	218205.08	15853	13.65
log2	19031.21	4742.11	4.25	17561.92	8680.94	1461	4.34	15915.06	7393.68	1359	4.31
max	1860.41	2410.55	0.38	1860.41	2410.55	0	0.44	1860.41	2410.55	0	0.38
multiplier	16760.62	3638.24	1.68	15009.51	7701.35	1511	1.35	13219.61	6724.49	1467	1.66
sin	3585.97	2410.42	0.94	3361.07	3636.69	265	0.85	3132.99	3194.23	204	0.88
sqrt	12676.20	55493.75	1.42	14564.95	86093.17	1830	1.53	13107.36	64646.94	804	1.71
square	12966.38	3065.54	1.06	11600.65	7304.42	1471	0.56	9940.72	6974.56	1322	1.06
arbiter	7401.25	923.08	2.27	7401.25	923.08	0	2.80	7401.25	923.08	0	2.35
cavlc	441.37	221.24	0.03	446.03	217.10	7	0.04	446.50	223.54	2	0.02
ctrl	99.84	115.60	0.00	101.01	130.63	1	0.01	100.78	115.60	0	0.00
dec	289.26	56.63	0.16	289.26	56.63	0	0.18	289.26	56.63	0	0.16
i2c	911.19	210.04	0.09	911.42	210.04	2	0.12	911.19	210.04	0	0.10
int2float	146.50	187.80	0.01	146.50	187.80	1	0.01	146.50	187.80	0	0.01
mem_ctrl	30658.97	1530.06	3.23	30650.35	1512.75	45	5.27	30598.68	1512.75	5	3.47
priority	661.11	2475.18	0.07	661.11	2475.18	0	0.08	661.11	2475.18	0	0.07
router	171.69	374.68	0.09	178.22	708.09	38	0.02	171.21	522.58	15	0.07
voter	7190.60	895.47	0.59	6468.51	1372.72	1257	0.13	5534.16	1197.18	1183	0.62
Reduction				2.41%	-44.02%			**7.42%**	-26.27%		
Total			30.02			32855	27.00			26246	32.67

The multiple-output mapping generally uses fewer multiple-output cells than the two-step approach while outperforming it. Our implementation effectively leverages mapping heuristics to select the proper cells based on the global context. Moreover, even if the delay is not constrained, our approach often finds solutions with both better area and delay. Only the *sqrt* benchmark has 3.4% worse area when mapping using multiple-output gates. Nevertheless, this is due to area recovery heuristics. Specifically, the multiple-output mapper has 5.2% better area until the end of *area flow* rounds. However, subsequent local rounds using *exact area* are considerably more effective on the vanilla flow (with a 13.37% area reduction) resulting in the mentioned quality difference. An interesting result is obtained on the *adder* benchmark. Although the two multiple-output flows use the same number of multiple-output cells, our proposed method achieves better area and delay by selecting different configurations of the multiple-output cells (input and output negations) such that the number of inverters is globally minimized (192 instead of 256).

In this experiment, we have not constrained the delay during mapping for two reasons: (i) Yosys' method does not support effective delay minimization when adder cells are used (since they are considered as white boxes); and (ii) we want to test how much additional area is recovered with the proposed method against the state-of-the-art approach and Yosys. In addition, multiple-output cells have higher propagation delay compared to single-output cell realizations. Hence, compared to the vanilla flow case, delay increase is justified.

4.4.4 Discussion

In the previous subsection, we showed the potential of supporting multiple-output cells in a technology mapper. Our method effectively maps to multiple-output gates when convenient, achieving a substantial area reduction both in area-oriented and delay-oriented mapping. However, the ASAP7 technology library, like other open-source libraries such as SKY130, contains only half and full adders. We predict that by notably increasing the number of multiple-output cells the mapper would be less efficient at handling them. This is mainly due to a very large number of matches and *incompatible* cuts (explained in Section 10) that must be filtered to achieve scalability to large designs.

We perform an experiment to test the limits of our mapping algorithm and to identify which multiple-output standard cells are worth designing to enhance the quality of results. In this experiment, we generated 120 two-output cells by merging single-output cells of the same support size from the ASAP7 library, assuming a 20% reduction in area. Our mapper correctly matches some of the generated multiple-output cells. However, circuits rarely contain multiple-output functions (with shared support) that differ from common cells, such as the $\mathcal{NPN}$ classes XOR2-AND2 (half adder) or XOR3-MAJ3 (full adder), and that can be extracted structurally. Consequently, the number of matches for the other generated two-output cells is low. Additionally, these cells are often not used to produce a mapping solution. Most of the generated multiple-output cells represent the compression of random logic. As such, these cells do not lead to any structural advantage during global technology mapping. Additionally, due to cut filters and the algorithm design for scalability, a library with many multiple-output cells can limit the effectiveness of the proposed approach. In this work, we addressed scalability primarily. In particular, we obtained a quick algorithm with a similar run time to state-of-the-art mappers.

To mitigate the decrease in quality when many multiple-output cells are defined, we propose to limit the number of cells handled by a global technology mapper. The selected ones should have specific regular structures able to affect the global mapping. In the experiments, we showed the potential of integrating half and full adders. Other appearing cells in our experiments are: (i) $O_1 = abc\ O_2 = ab + ac$; (ii) $O_1 = abc\ O_2 = c(\bar{a} + \bar{b}) + ab\bar{c}$. Additionally, we noticed that it is more effective to handle cells that compress random logic later in the flow through incremental local remapping steps as performed in [22].

Despite these findings, the algorithms presented in this paper can support multiple multiple-output cells and are not restricted to only half adders or full adders. If we limit the mapper to work on a small logic cone (e.g., in the order of tens of nodes), cut filters can be relaxed to include incompatible cuts. KL-cuts [128] can also be used to enumerate more multiple-output cuts. Conflicting configurations of multiple-output cells can be enumerated and mapping can be rapidly performed for each one of them, also in parallel. Then, the best solution is committed. This method would be a part of a re-mapping engine that extracts windows of mapped logic and re-maps them using a better implementation. Moreover, this approach would extend the remapping algorithm in [22] that focuses only on one multiple-output cell at a time. The integration of this mapper into a remapping engine is out of the scope of this work.

4.5 Improving Covering Algorithms for Technology Mapping

In this section, we propose new algorithms to improve the covering for standard cell networks. Rather than proposing new heuristics, this section focuses on methods to better estimate the fan-out references of nodes during mapping. Thus, we study approaches to better estimate the value $f(l)$ of Equation 4.6. In particular, we first review the state-of-the-art method to estimate the fan-out references. Then, we propose two new approaches to improve covering: (i) a policy for fan-out estimation and match evaluation; and (ii) a method to improve area under delay constraints using alternative matches.

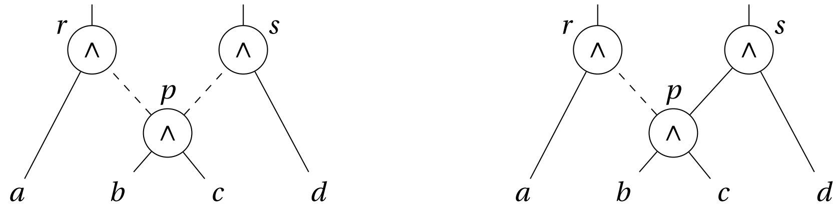
In the experimental results, we show that the fan-out estimation policy and the match evaluation methods reduce the average area by 3.78% when mapping for best delay. Alternative matches further improve the area by an additional 0.88%. The impact of alternative matches becomes evident when mapping under delay constraints. When fixing the delay constraint to be 5% higher than the best found, alternative matches reduce the area by 2.04% on average.

4.5.1 Related Works

As introduced in Section 4.2.3, during the covering phase, the best match for each node is extracted in topological order. In standard cell mapping, it is convenient to select two best matches at each node: one representing the uncomplemented (positive) polarity, and one representing the complemented (negative) polarity [38, 202]. Considering both polarities at each node allows for accurate area estimations by considering inverter sharing. Furthermore, it helps leverage logic duplication to avoid additional inverter delay costs. We explain the advantages of covering using two polarities using Figure 4.5. We represent complemented connections using dashed edges and uncomplemented connections using continuous edges.

In Figure 4.5a, we show how two polarities support better area estimations.

Example 4.5.1. *Let us consider node p with two negated outputs and a cell library that contains a 2-input AND cell (AND2) with area 2 and an inverter with area 1. Let us first consider a single match model for covering. The only possible match rooted in p is an AND2 with area flow 2. At node r a match would pick an AND2 with an additional inverter over the edge (p,r), resulting in area flow $3+2/2=4$. Similarly, at node s a match would pick an AND2 with an additional inverter over the edge (p,s), with area flow of 4. The total area estimation at p and s is 8 even if the cover has a minimum area of 7 because the inverter can be shared. This is the consequence of mapping using a single match, which creates an unnecessary inverter duplication in the area estimation. Although these redundancies could be removed with a circuit analysis after mapping, the mapper would be affected by wrong area estimations during the covering phase worsening the potential quality of results. The solve this problem, we consider two matches at each node for both polarities. At node p we consider an AND2, with area flow of 2, for its uncomplemented match and an AND2 with an output inverter, with area flow of 3, for its complemented match. The uncomplemented match at node r considers an AND2 cell connected*



(a) Inverter sharing

(b) Optimal delay

Figure 4.5: Advantages of covering using two polarities

to the complemented match at node p, with area flow of $2+3/2=3.5$*. The matches at node s area computed similarly, resulting in an area flow of* 3.5 *for the uncomplemented match. The total area estimation at p and s is now accurate.* ▲

In Figure 4.5b, we show how matching in two polarities may lead to better delay results.

Example 4.5.2. *Let us consider a unitary delay model (i.e., a constant delay model where cells have unitary propagation delays) and the previous cell library with the addition of a* 2*-input NAND cell (NAND2). When considering single match covering, p is arbitrary mapped using one cell between AND2 and NAND2. In this example, let us consider an AND2 cell. Due to this choice, the best delay at node r is* 3 *since an inverter is needed on the edge* (p,r)*. When considering matching in two polarities, node p is assigned to both an AND2, for the uncomplemented polarity, and a NAND2, for the complemented one. In this case, node r uses the complemented polarity match of p (NAND2) to achieve a delay of* 2*. Instead, node s uses the uncomplemented polarity match of p (AND2). This operation duplicates node p to minimize the delay. Generally, this operation is evaluated in terms of delay gain and area increase.* ▲

4.5.2 Referencing Policy and Gate Selection

We investigate methods to estimate the fan-out references for area flow computation. This subsection delves quite deeply into technology mapping algorithms. As described in Section 4.5.1, our method uses two best matches at each node to model the complemented and uncomplemented behavior. Hence, each node is associated with two fan-out estimation and two area flow values, for the complemented and uncomplemented match. However, coordinating these two values to make correct local cell selections is not an easy task. We address these critical aspects proposing a policy to update the references.

Algorithm 4.8 outlines the steps performed for one mapping round using two matches for the positive and negative polarity. Nodes are processed in topological order, ensuring that the arrival times and area flow of each node's transitive fan-ins are known. For each internal node, the algorithm extracts the best positive polarity match and the best negative polarity match according to the current cost metric C (delay or area flow). Then, `select_polarity` (at line 8) decides whether to use both matches, which would result in node duplication as demonstrated

Algorithm 4.8: Node mapping in two polarities during delay or area flow rounds

```
Input: Subject graph $N$, Mapping $M$, Cuts $cuts$, Matching library $lib$, cost function $C$,
       estimated fan-out references $ref$
Output: New mapping $M'$
req ← compute_required($N$, $M$, $lib$)
$M'$ ← empty_mapping($N$);
foreach Node $n \in N$ in topological order do
    if $n \in PI(N)$ then
        continue
    select_positive_polarity($M'$, $n$, $cuts(n)$, $lib$, $C$, $req$, $ref$)
    select_negative_polarity($M'$, $n$, $cuts(n)$, $lib$, $C$, $req$, $ref$)
    select_polarity($M'$, $n$, $req$, $ref$, $C$);          ▷ Utilize one or both polarities
$C$ ← extract_cover($M'$)
return $C$
```

in Example 4.5.2, or to select only one match between the positive and the negative polarity, implementing the other polarity using an output inverter as shown in Example 4.5.1.

Selecting to implement both matches or only one requires defining heuristic strategies, also depending on the cost metric C. Simple cases include: (i) if a polarity has no selected match (e.g., a match does not exist) then the other is forcefully selected; (ii) if the best match for a polarity *dominates* the other, i.e., it can realize a lower-cost implementation of the other polarity through an inverter while respecting the required times, then it is selected to replace the other polarity. During the delay mapping round matches for both phases are normally selected to minimize delay (as in Example 4.5.2), unless one of the previous two conditions holds. Similarly, during the area flow rounds, if the area flow of one polarity does not dominate the other, both are selected. However, reference estimations change every mapping round affecting area flow values and the choice between keeping both phases or selecting one of the two. Consequently, developing good policies to update references is crucial for area optimization.

Among the possible policies that can be designed, we discuss two that we name *collective reference*[9] and *distributed reference*. Let us consider a node n with two fan-out estimations $f^p(n)$ and $f^n(n)$ for the uncomplemented (positive) and complemented (negative) polarity, respectively. If a node selects both polarities, meaning that both polarities are used in the cover (the node is duplicated as in Example 4.5.2), $f^p(n)$ and $f^n(n)$ both reflect the fan-out of the corresponding polarity. Specifically, the area flow of each phase for a match m at a node n

[9] This strategy is adopted by mapper *map* in ABC.

is computed as follows:

$$AF^{p'}{}_m = \frac{a_m + \sum_{(l,\phi)\in leaves(m)} AF^{\phi'}(l)}{f^p(n)} \tag{4.15}$$

$$AF^{n'}{}_m = \frac{a_m + \sum_{(l,\phi)\in leaves(m)} AF^{\phi'}(l)}{f^n(n)}. \tag{4.16}$$

Moreover, the estimations are updated as follows for each iteration as follows:

$$f^p(n)^j = \max(\alpha \times f^p(n)^{j-1} + (1-\alpha) \times ref^p(n), 1) \tag{4.17}$$

$$f^n(n)^j = \max(\alpha \times f^n(n)^{j-1} + (1-\alpha) \times ref^n(n), 1), \tag{4.18}$$

where ref^p (ref^n) is the actual reference in the cover for the positive (negative) polarity. Both policies behave in the same way when a node implements selected matches for both polarities. Differences arise when a node implements only one polarity and the other one is used through an inverter.

The *collective reference* assumes that references should be summed (collected) together when phases are shared through an inverter (see Example 4.5.1), as the match inherits fan-out from both phases. This method uses a single collecting reference, $f^\star(n) = f^p(n) + f^n(n)$, to model a single match polarity usage. Consequently, area flow uses $f^\star(n)$ as an estimation of the area flow contribution of the match when evaluating to use a single match:

$$AF'{}_m = \frac{a_m + \sum_{(l,\phi)\in leaves(m)} AF^{\phi'}(l)}{f^\star(n)} \tag{4.19}$$

Thus, the collective reference favors solutions that share phases if the required time allows for it. To update the reference estimations, this approach employs Equations 4.17 and 4.18.

Conversely, the *distributed reference* approach maintains separate fan-out estimations, even when the match is shared. As a result, area flow is computed as in Equations 4.15 and 4.16. However, during cover extraction references are updated differently: the implemented phase considers direct connections to that phase and the inverter as references, while the not implemented phase considers only the connections to the inverter as references. Compared to the classical method, the implemented phase includes a reference from the inverter.

Generally, collective reference tends to take more aggressive choices than the distributed reference, aiming to implement nodes using one phase whenever the required time allows for it. This results in a faster convergence of area flow but also performs less well in inverter minimization (or inverter sweeping). In contrast, the distributed reference approach requires more iterations of area flow to decide which phase to implement due to the small bias from the inverter reference to the implemented phase. This approach tends to obtain better quality solution. Ultimately, the best approach depends on the specific design. In our implementation, we utilize the distributed reference method as it tends to produce designs with lower area and fewer inverters.

4.5.3 Alternative Matches

Technology mapping is achieved through several incremental mapping and refinement rounds. Each round uses the previous generated cover or reference estimations to tune the heuristics and improve the cover. As described in Section 4.2.3, reference estimations are key for a good area recovery during area flow rounds. In this subsection, we propose a method based on alternative matches to improve the reference estimations of area flow to achieve better area recovery.

As mentioned in Subsection 4.2.3, during covering the actual fan-out numbers are not known and have to be estimated. The state-of-the-art-approach estimates the fan-out reference of a node to be the one of the subject graph. When using two fan-out estimations, one for each polarity, both are initialized to this value. Then, the following iterations estimate the fan-out reference to be the a linear combination of the one of the previous round and the current reference in the cover (see Equation 4.9). This approach allows for a gradual improvement of the cover without quickly leading to a local minima after the first round of covering. Although Equation 4.9 mitigates the dependency of the fan-out estimations from the first rounds of covering, this dependency can decrease the quality of results when the first round is delay-oriented. As discussed in Section 4.2.3, delay-driven or delay-constrained technology mapping involves an initial covering round that focuses on minimizing delay. Subsequent rounds aim to optimize the area while meeting these delay constraints. However, the first round biases fan-out references towards a delay-oriented configuration rather than an area-oriented one. While this approach benefits critical or near-critical paths, it leads to sub-optimal area results for non-critical paths. To overcome this limitation, we propose to compute alternative area-driven matches during the delay-driven round and dynamically select them during the cover extraction phase.

The alternative matches approach works as follows. During the delay-oriented round, matches for best delay and matches for best area flow - referred to as *alternatives* - are computed and collected. This is equivalent to combining one round of delay-oriented match evaluation and one round of area-oriented match evaluation, both utilizing the initial fan-out estimations derived from the subject graph. Then, during the cover extraction phase, when delays and required times are known, an algorithm dynamically selects which implementation to use based on the slack.

Algorithm 4.9 shows the selection match process during cover extraction. To explain the general idea, we illustrate the procedure for matching in one polarity. While using two polarities involves additional considerations, the process can be addressed in a similar manner. The inputs to the algorithm are the subject graph (N), the best delay matches (M^D) and the best area matches (M^A) for each node, and the corresponding arrival times (D^D and D^A). Initially, the cover is empty (line 1). Next, references and required times for each node are set

Algorithm 4.9: Cover extraction with alternative matches

```
    Input: Subject graph $N$, Collected delay matches $M^D$, Collected area matches $M^A$, Arrival
           times delay matches $D^D$, Arrival times area matches $D^A$
    Output: Cover $C$
 1  $C \leftarrow \emptyset$
 2  foreach Node $n \in N$ do
 3      $Ref(n) \leftarrow 0$
 4      $R(n) \leftarrow \infty$
 5  $R \leftarrow$ set_output_required_time($D^D$)
 6  foreach Node $n \in PO(N)$ do
 7      $Ref(n) \leftarrow Ref(n) + 1$
 8  foreach Node $n \in N$ in reversed topological order do
 9      if $Ref(n) = 0$ then
10          continue
11      Match $m \leftarrow \Lambda$
12      if $D^A(n) \leq R(n)$ then
13          $m \leftarrow M^A(n)$
14      else
15          $m \leftarrow M^D(n)$
16      $C \leftarrow C \cup \{m\}$
17      foreach Node $l \in leaves(m)$ do
18          $Ref(l) \leftarrow Ref(l) + 1$
19          $R(l) \leftarrow \min(R(l), R(n) - p^m(l))$
20  return $C$
```

to zero and infinite, respectively. At line 5, the required time for the output nodes is set based on external constraints or the computed delays D^D in the case of minimal-delay mapping. Next, the PO nodes are referenced since they are always part of the cover. Afterwards, every node that is a leaf of at least one match in C is processed in reverse topological order (line 8 to 19). The best area match is selected for the cover if its delay satisfies the required time. Otherwise, the best delay match is selected. Finally, the references and the required time for the leaves of the selected match are updated accordingly.

4.5.4 Experimental Results

In this subsection, we present experimental results on using alternative matches during technology mapping and, generally, of our mapping covering algorithm. First, we evaluate the usage of alternative matches in delay-driven technology mapping for best possible delay. Then, we test mapping under delay constraints. We use the ASAP7 7nm cell library [44] as a standard cell library. For our experiments, we use the EPFL combinational benchmark suite [3] containing several circuits provided as *and-inverter graphs* (AIGs). The baseline has been obtained by applying the area-driven balancing algorithm available in *Mockturtle*[10] [183].

[10] Available at: https://github.com/lsils/mockturtle

Table 4.6: Delay-oriented technology mapping for best delay with and without alternative matches and a comparison against ABC *map*.

Benchmark	**Baseline**		**ABC *map***			**Without alternatives**			**With alternatives**		
	Size	Depth	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
adder	1020	255	92.53	2577.43	0.13	83.59	2573.43	0.02	84.03	2573.43	0.02
bar	3336	12	325.47	168.08	0.17	353.63	168.08	0.05	356.23	168.08	0.05
div	51725	4372	5336.99	43765.09	2.95	4988.74	43765.25	1.67	4982.71	43765.25	1.63
hyp	214335	24801	16395.10	195822.72	13.53	15185.09	195345.77	7.42	14958.07	195345.77	8.07
log2	31950	410	2177.57	3955.63	3.72	1915.88	3848.17	2.33	1908.24	3848.17	2.28
max	2865	229	226.59	2213.23	0.24	209.46	2213.23	0.10	209.22	2213.23	0.11
multiplier	26953	266	1954.48	2738.95	2.01	1925.57	2667.36	0.99	1916.27	2667.36	1.03
sin	5399	186	431.22	1814.85	0.74	444.33	1760.82	0.49	446.42	1760.82	0.45
sqrt	24216	5058	1767.89	47442.32	1.70	1681.12	47438.44	0.85	1681.05	47438.44	0.81
square	18302	250	1193.12	2516.39	1.37	1104.92	2505.10	0.72	1070.08	2505.10	0.76
arbiter	11839	87	766.68	898.75	1.20	766.50	898.75	0.65	766.50	898.75	0.67
cavlc	690	16	41.57	187.04	0.10	39.58	187.04	0.02	38.80	187.04	0.01
ctrl	170	10	8.58	102.49	0.08	7.96	102.49	0.00	7.86	102.49	0.00
dec	304	3	30.83	65.72	0.12	30.83	65.72	0.05	30.83	65.72	0.04
i2c	1275	16	78.65	182.65	0.14	74.51	182.65	0.06	73.69	182.65	0.05
int2float	235	15	13.93	181.00	0.08	13.21	181.00	0.01	12.95	181.00	0.01
mem_ctrl	46820	114	2763.83	1103.42	2.32	2670.08	1100.46	1.79	2595.75	1100.46	1.90
priority	863	249	87.28	2501.95	0.12	84.20	2501.95	0.04	82.37	2501.95	0.04
router	257	27	19.59	280.01	0.10	18.89	274.05	0.02	18.57	274.05	0.02
voter	13029	71	1506.14	804.96	1.37	1546.35	749.36	1.92	1519.56	749.36	2.13
Total					32.19			19.20			20.08
Reduction						3.78%	0.92%		**4.66%**	**0.92%**	

Distributed references and alternative matches have been implemented in the open-source technology mapper *emap* in the logic synthesis framework *Mockturtle*. The experiments use *emap* with Boolean matching and compare against ABC (command *map*), which uses collective references. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in *ABC*[11].

Delay-oriented technology mapping

In this experiment, we test *emap* to map for minimal delay in two modes: one utilizing alternative matches and Algorithm 4.9 during the cover extraction phase, and the other using the standard algorithm without alternative matches. To further demonstrate the quality of the mapped circuits found by *emap*, we compare its performance against the default technology mapper in ABC (command *map*).

Table 4.6 presents the results, which are evaluated in terms of area reduction and delay reduction compared to ABC. The new referencing policy and gate selection method contribute to achieve an average area reduction of 3.78% in the default version of *emap*. Instead, the delay reduction by 0.92% is mainly due to different cut prioritization heuristics. When mapping with alternatives, the average area is further reduced by an additional 0.88%. The impact of alternative matches varies depending on the benchmark. For instance, in the circuit *square*, the area reduction peaks at 3.15% compared to the mapper without alternative matches, while the largest area degradation is −0.74% in the circuit *bar*. Although the area reduction achieved by alternative matches is modest, the primary advantage of using alternative matches becomes evident when mapping under delay constraints, as shown in the subsequent experiment.

[11] Available at: https://github.com/berkeley-abc/abc

Table 4.7: Delay-oriented technology mapping with and without alternative matches given a worst-delay constraint 5% higher than the best one found by the mapper.

Benchmark	**Without alternatives**			**With alternatives**		
	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
adder	80.14	2699.27	0.02	80.92	2699.83	0.02
bar	278.35	176.28	0.05	245.97	176.48	0.05
div	4716.73	45822.60	1.47	4323.53	45910.42	1.62
hyp	14996.99	202567.03	6.79	14596.83	204537.44	6.67
log2	1802.42	4036.25	2.11	1780.54	4030.47	2.13
max	207.17	2323.43	0.10	202.40	2323.43	0.10
multiplier	1558.88	2799.09	0.88	1588.74	2800.08	0.89
sin	425.18	1848.58	0.41	429.03	1848.49	0.41
sqrt	1663.31	49790.13	0.72	1657.69	49806.60	0.73
square	1098.55	2588.13	0.67	1063.26	2587.45	0.68
arbiter	685.82	935.93	0.61	652.05	941.52	0.63
cavlc	37.86	195.47	0.02	37.12	195.47	0.01
ctrl	7.97	102.49	0.00	7.90	102.49	0.00
dec	28.16	68.67	0.04	28.16	68.67	0.04
i2c	74.07	191.03	0.05	73.10	190.87	0.05
int2float	12.59	190.04	0.00	12.24	189.88	0.01
mem_ctrl	2663.66	1145.53	1.65	2587.68	1153.64	1.64
priority	61.78	2551.81	0.03	60.19	2589.50	0.04
router	18.14	287.61	0.02	18.46	287.61	0.02
voter	1244.41	786.81	1.26	1252.78	786.78	1.36
Total			16.90			17.10
Reduction				**2.04%**		

Technology mapping under delay constraints

In this experiment, we test mapping with and without alternative matches under delay constraints. Specifically, the delay constraints are assigned by allowing the best delay found by *emap* in Table 4.6 to increase by up to 5%. This relaxation permits the mapper to increase the delay by up to 5% when necessary to minimize the area.

Table 4.7 shows the results, which are evaluated in terms of area reduction and delay reduction compared to mapping without alternative matches. The experiment demonstrates that using alternative matches further reduces the area by 2.04% on average. The area reduction peaks at 11.63% in benchmark *bar*, while the largest area degradation is −1.92% in circuit *max*. Although there are 5 cases out of 20 where mapping with alternative matches performs slightly worse due to heuristic choices during covering, the experiment highlights that alternative matches are generally an effective solution for reducing area when mapping under delay constraints.

4.6 Extended Mapping (*emap*)

In this section, we present *emap*, an extended technology mapping algorithm that implements the techniques discussed in this chapter to achieve better quality of results. Emap is implemented in the open-source logic synthesis library Mockturtle [183]. Compared to other mappers, it implements the novel methods presented in this chapter:

- Hybrid matching presented in Section 4.3
- Support of multiple-output cells presented in Section 4.4.
- Enhanced covering heuristics presented in Section 4.5.

In the experiments we show that:

- *emap* achieves better area and delay than the mappers in ABC when mapping for best delay with the ASAP7 library. For instance it obtains 9.16% better area and 2.95% better delay after technology mapping, and 9.22% better area and 2.59% better delay after buffering and gate sizing, compared to ABC *&nf*.
- *emap* achieves 5.75% better area and 1.38% better delay than mapper *&nf* in ABC when mapping for best delay with a commercial 28nm library. However, the results slightly degrade after buffering and gate sizing with a 2.77% better area and 0.63% worse delay, due to the poor accuracy of the gain-based delay model. Nevertheless, *emap* achieves delay-area Pareto point solutions on 20 out of 21 benchmarks, while *&nf* in only 15 out of 21.
- *emap* achieves better area than the mappers in ABC when mapping for best area with the ASAP7 library. For instance, it obtains 15.87% better area compared to ABC *map*.
- *emap* achieves better area than the mappers in ABC when mapping for best area with a commercial 28nm library. For instance it obtains 10.24% better area compared to ABC *map*.
- *emap* has better run time than other mappers in ABC. However, it has also a higher memory usage due to alternative matches, hybrid matching, and multiple-output cell support.

4.6.1 Experimental Results

In the experimental results section, we evaluate the techniques proposed in this chapter, which have been implemented in the command *emap* in *Mockturtle*, against the other open-source state-of-the-art technology mappers in ABC, namely command *map* and command *&nf -p*. The mappers share similar settings for a fair comparison. While the mappers in ABC use

Boolean matching, mapper *emap* uses hybrid matching (discussed in Section 4.3). For our experiments, we use the EPFL combinational benchmark suite [3], containing several circuits provided as *and-inverter graphs* (AIGs), and the IWLS 2005 benchmark suite [85]. The baseline has been obtained by optimizing the benchmarks using the area-oriented AIG balancing algorithm in *Mockturtle*. We employ the gain-based delay model generated by ABC. Hence, all mappers share the same delay model and delay estimations. We perform buffering and gate sizing using ABC with the script `buffer; upsize; dnsize`, and we report timing and area after buffering and gate sizing using the static timing analysis command `stime` in ABC. The experiments do not use multiple-output cells with *emap* since they are not supported by the buffering and gate sizing commands in ABC. In this section, we perform the experiments using two standard cell libraries:

- ASAP7 7.5-track standard cell library [44]: an open-source standard cell library to predict the behavior of advanced nodes at $7nm$. Besides the common library cells, it contains large AND-OR cells, up to 9 inputs, and half and full adder cells. This library has been pre-processed by the open-source EDA tool-chain OpenLane[12] to remove unnecessary cells, such as filler cells and clock-gating cells. Since the library is open source, the experimental results using this library are reproducible. The adopted library with gain-based delay model is available in the tool *Mockturtle*.

- A commercial $28nm$ standard cell library: commercial library containing various cells up to 6 inputs and half and full adders.

Mapping for best delay using the ASAP 7*nm* standard cell library

In this experiment, we compare *emap* against the open-source mappers in ABC for minimal-delay mapping by setting the required time to zero. We use the ASAP $7nm$ cell library, which contains large cells up to 9 inputs. Hence, this library highlights the advantages of hybrid matching against Boolean matching. In this subsection, we show the experimental results before buffering and gate sizing in Table 4.8 and after buffering and gate sizing in Table 4.9.

Table 4.8 presents the results of our evaluations, comparing area reduction and delay reduction after technology mapping with the ABC command *map* of ABC, using the gain-based delay model of the library. The table is divided into two sections: the first section displays results for the EPFL benchmarks, while the second section shows results for the IWLS benchmarks. Our mapper demonstrates a significant improvement in average area, achieving a reduction of 13.93%, and an average delay reduction of 2.13% compared to command *map* of ABC. Additionally, when compared to *&nf*, our mapper achieves an average area reduction of 9.61% and a delay reduction of 2.95%. Furthermore, *emap* has a better run time than the mappers in ABC. These improvements are mainly attributed to the use of large gates thanks to *hybrid matching* (in Section 4.3.2) and the enhanced heuristics guiding the mapping

[12] Available at: https://github.com/The-OpenROAD-Project/OpenLane

Table 4.8: Delay-oriented technology mapping comparing mappers in ABC and a mapper implementing the algorithms proposed in this chapter using the ASAP7 library.

Benchmark	**Baseline**		**ABC *map***			**ABC *&nf -p***			**Our work *emap***		
	Size	Depth	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
adder	1020	84.03	92.53	2577.43	0.13	100.62	2577.43	0.19	84.03	2573.43	0.03
bar	3336	216.32	325.47	168.08	0.17	355.51	168.08	0.11	216.32	172.61	0.05
div	51725	4673.59	5336.99	43765.09	2.95	5044.47	44201.14	1.67	4673.59	43836.68	1.29
hyp	214335	15237.73	16395.1	195822.72	13.53	15539.4	196733.14	6.98	15237.73	195276.66	6.89
log2	31950	1737.34	2177.57	3955.63	3.72	2023	3941.43	4.84	1737.34	3916.17	1.44
max	2865	172.77	226.59	2213.23	0.24	224.39	2213.23	0.17	172.77	2228.59	0.08
multiplier	26953	1654.58	1954.48	2738.95	2.01	1850.21	2738.95	1.04	1654.58	2649.57	0.75
sin	5399	407.4	431.22	1814.85	0.74	435.46	1811.8	0.54	407.4	1792.56	0.27
sqrt	24216	1683.2	1767.89	47442.32	1.70	1831.71	48222.85	2.26	1683.2	47438.60	0.69
square	18302	1084.48	1193.12	2516.39	1.37	1143.31	2510.24	0.62	1084.48	2503.20	0.54
arbiter	11839	766.32	766.68	898.75	1.20	779.3	898.75	0.26	766.32	898.75	0.7
cavlc	690	37.52	41.57	187.04	0.10	39.47	187.04	0.09	37.52	186.07	0.02
ctrl	170	7.85	8.58	102.49	0.08	8.51	102.49	0.09	7.85	101.21	0
dec	304	27.44	30.83	65.72	0.12	30.29	65.72	0.11	27.44	66.15	0.02
i2c	1275	69.38	78.65	182.65	0.14	76.24	182.65	0.14	69.38	182.65	0.04
int2float	235	11.31	13.93	181	0.08	13.46	181	0.08	11.31	182.22	0.01
mem_ctrl	46820	2427.28	2763.83	1103.42	2.32	2702.08	1100.46	1.79	2427.28	1104.37	1.5
priority	863	82.25	87.28	2501.95	0.12	79.77	2501.95	0.12	82.25	2501.95	0.03
router	257	18.41	19.59	280.01	0.10	18.87	278.53	0.09	18.41	274.05	0.02
voter	13029	1538.53	1506.14	804.96	1.37	1496.84	783.85	0.46	1538.53	755.33	1.04
ac97_ctrl	14241	11	947.27	138.37	0.18	912.23	138.37	0.28	735.18	127.9	0.18
aes_core	21376	21	1655.13	260.41	0.67	1647.58	259.77	0.54	1642.25	249.71	0.6
des_area	4808	27	445.02	341.32	0.24	439.05	352.38	0.26	399.81	321.46	0.21
des_perf	80101	17	7642.30	219.48	3.33	7060.1	218.54	3.03	6829.56	197.23	3.44
DMA	24278	21	1588.15	234.47	0.98	1514.84	258.56	0.93	1315.11	230.92	0.84
DSP	45004	52	2842.72	562.36	2.3	2732.09	562.36	2.83	2414.51	534.42	1.81
ethernet	86576	27	5139.95	311.82	5.06	4838.34	302.14	5.16	3737.37	310.15	3.56
iwls05_i2c	1132	12	72.32	141.10	0.04	68.86	137.36	0.11	61.49	145.74	0.03
leon2	788859	45	78019.13	500.90	94.92	52230.6	513.74	100.18	51641.42	484.87	62.65
leon3_opt	973338	51	67090.78	529.32	130.45	55784.53	541.43	163.19	54906.4	510.13	80.28
leon3	1087244	47	70703.28	514.19	113.4	64647.32	520.81	85.22	58696.03	487.25	71.64
leon3mp	651721	47	43116.87	495.61	59.33	37401.58	511.1	57.01	37149.92	485.16	37.16
iwls05_mem_ctrl	15048	29	804.64	306.03	1.44	757.61	339.04	1.36	681.39	325.07	0.77
netcard	803417	36	50457.91	396.18	68.87	45761.73	398.9	59.02	44035.52	388.81	40.43
pci_bridge32	22705	22	1488.73	262.18	0.85	1355.34	292.43	0.86	1170.23	256.27	0.59
RISC	74651	36	4361.92	414.12	2.94	4145.32	414.12	3.42	3629.55	408.4	2.01
sasc	770	8	49.37	108.32	0.01	46.54	108.32	0.09	36.32	106.74	0.01
simple_spi	1039	10	64.97	127.66	0.02	64.05	127.66	0.10	54.08	124.01	0.02
spi	3760	31	240.29	360.00	0.17	228.54	360	0.28	216.95	346.5	0.15
ss_pcm	405	7	32.59	97.50	0	31.31	97.5	0.08	33.04	86.27	0
systemcaes	12242	44	750.50	545.99	0.32	730	543.72	0.42	597.31	493.88	0.31
systemcdes	2877	23	263.51	289.52	0.11	271.87	284.21	0.18	215.38	275.59	0.1
tv80	9461	43	602.11	469.78	0.49	597.76	467.13	0.57	549.9	463.7	0.37
usb_funct	15715	23	950.22	300.59	0.49	910.65	301.55	0.56	803.98	277.49	0.39
usb_phy	452	9	32.04	112.55	0.01	30.51	112.46	0.08	29.17	112.46	0.01
vga_lcd	126636	18	9195.92	238.60	8.3	7733.28	243.57	7.97	6476.49	250.16	6.02
wb_conmax	47520	18	2428.06	248.82	3.19	2242.56	261.53	3.31	2139.92	235.33	2.11
Total					530.3			518.69			**331.1**
Reduction						4.58%	-0.91%		**13.93%**	**2.13%**	

process (in Section 4.2.3). For instance, when *emap* uses Boolean matching instead of hybrid matching, it achieves an average area reduction of 7.82% and an average delay reduction of 1.81% compared to ABC *map*.

Table 4.9 presents the results for the IWLS 2005 benchmarks with less than 100K gates after technology mapping, buffering, and gate sizing. We compare *emap* against the ABC command *&nf* using the technology mapping results from Table 4.8. After technology mapping under the gain-based delay model, we perform buffering and gate sizing with ABC. After gate sizing, our mapper achieves an average area and delay reduction by 9.22% and 2.59%, respectively, compared to the mapper in ABC. Hence, the improvement shown in Table 4.8 is preserved under a more precise delay model (that uses NLDM).

Table 4.9: Comparing our technology mapper *emap* against ABC *&nf* after performance-driven buffering and gate sizing using the ASAP7 library.

Benchmark	**Baseline**		**ABC *nf*-*p***		**Our work *emap***	
	Size	Depth	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)
ac97_ctrl	14241	11	947.86	126.61	785.69	117.96
aes_core	21376	21	1674.44	273.42	1697.16	252.39
des_area	4808	27	470.15	372.71	433.14	353.17
des_perf	80101	17	7213.21	237.51	7156.42	232.15
DMA	24278	21	1581.92	271.38	1394.36	249.12
DSP	45004	52	2813.2	577.41	2529.13	556.61
ethernet	86576	27	5164.59	335.3	4219.71	354.08
iwls05_i2c	1132	12	69.34	160.69	62.52	162.4
iwls05_mem_ctrl	15048	29	790.94	356.72	721.45	344.57
pci_bridge32	22705	22	1421.13	294.12	1278.46	296.03
RISC	74651	36	4298.5	404.78	3838.19	417.24
sasc	770	8	46.9	101.19	37.69	109.13
simple_spi	1039	10	64.97	134.48	55.94	131.12
spi	3760	31	241.09	322.71	229.46	327.47
ss_pcm	405	7	32.75	90.55	35.87	65.24
systemcaes	12242	44	770.63	508.21	641.51	474.88
systemcdes	2877	23	283.16	303.6	228.79	308.51
tv80	9461	43	619.61	509.44	575.3	483.14
usb_funct	15715	23	926.56	277.43	823.67	283.81
usb_phy	452	9	30.47	90.78	29.48	92.9
wb_conmax	47520	18	2360.05	279.26	2281.22	276.46
Reduction					**9.22%**	**2.59%**

Mapping for best delay using a commercial 28nm standard cell library

Similarly to the previous experiment, we compare *emap* against ABC *&nf* for delay-driven technology mapping. We employ a commercial 28nm standard cell library for these experiments. First, we perform mapping under the gain-based delay model, then we perform buffering and gate sizing using ABC. This experiment uses IWLS 2005 benchmarks containing less than 100K nodes. Since the standard cell library does not contain cells with more than 6-inputs hybrid matching performs similarly to Boolean matching.

Table 4.10 shows the experimental results, comparing area reduction and delay reduction after technology mapping, and after buffering and gate sizing. The columns labeled “TM” present the results after technology mapping, while the columns labeled “SN” present the results reported by static timing analysis after buffering and gate sizing. Our mapper achieves 5.75% better average area and 1.38% better delay after technology mapping compared to ABC. After sizing, the area improvement over ABC is reduced to 2.77%, and there is a minor delay degradation of 0.63%. The primary reason for this discrepancy, compared to Table 4.9, is the low accuracy of the gain-based delay model with respect to the static timing analysis results. The sizer fails to meet the effort delay in nearly every benchmark for both *&nf* and *emap*. As a result, *emap* is slightly penalized because it tends to share more logic to achieve

Table 4.10: Comparing our technology mapper *emap* against ABC *&nf* before and after performance-driven buffering and gate sizing using a 28nm commercial library.

Benchmark	**Baseline**		**ABC *nf -p***				**Our work *emap***			
	Size	Depth	Area TM (μm^2)	Delay TM (ps)	Area SN (μm^2)	Delay SN (ps)	Area TM (μm^2)	Delay TM (ps)	Area SN (μm^2)	Delay SN (ps)
ac97_ctrl	14241	11	4255.32	58.27	4603.54	86.05	3902.05	58.27	4367.92	88.00
aes_core	21376	21	8781.12	107.27	10100.92	169.98	8795.96	107.27	10244.30	169.92
des_area	4808	27	1956.05	146.78	2498.58	243.03	1868.93	141.09	2357.21	246.74
des_perf	80101	17	34735.35	88.45	38872.01	166.38	35915.25	88.45	38430.50	195.86
DMA	24278	21	7402.01	107.89	8037.67	186.97	7103.5	101.68	7876.76	176.07
DSP	45004	52	13817.81	246.79	14924.83	378.61	12685.95	241.17	14578.07	369.52
ethernet	86576	27	23899.69	123.14	25703.62	190.85	20419.45	123.14	22862.57	196.56
iwls05_i2c	1132	12	345.95	61.66	392.74	82.33	321.38	61.66	378.00	83.48
iwls05_mem_ctrl	15048	29	4071.39	143.87	4551.37	213.44	3838.7	132.83	4395.76	218.71
pci_bridge32	22705	22	6620.36	117	7283.43	176.15	6267.65	112.96	7066.71	181.53
RISC	74651	36	21036.62	173.98	22300.24	292.43	19603.18	176.62	21418.24	274.66
sasc	770	8	244.23	44.93	295.97	57.96	222.46	44.93	291.44	54.92
simple_spi	1039	10	310.71	55.86	351.92	73.25	284.28	55.86	349.40	69.78
spi	3760	31	1168.6	151	1360.67	225.52	1104.65	150.92	1293.89	237.17
ss_pcm	405	7	139.63	37.86	176.02	60.01	140.94	37.86	193.91	50.19
systemcaes	12242	44	3684.48	227.84	4170.10	299.46	3221.1	213.41	3703.90	325.93
systemcdes	2877	23	1280.54	125.73	1513.26	203.44	1255.43	120.78	1543.88	209.73
tv80	9461	43	3012.04	201.21	3481.76	308.55	2729.71	202.68	3213.50	307.72
usb_funct	15715	23	4541.27	126.57	4745.54	204.59	4280.98	126.78	4567.75	205.20
usb_phy	452	9	151.48	48.12	185.98	51.50	144.72	48.12	189.38	51.06
wb_conmax	47520	18	11957.33	107.21	13002.19	184.52	11501.42	109.29	12682.78	195.85
Reduction							**5.75%**	**1.38%**	**2.77%**	-0.63%

better area, leading to circuits with smaller areas but more inverters, which complicates the sizing phase. This complication may result in a slight degradation in delay due to the failure to balance the paths for more nodes, and an increase in area due to the need for larger gate sizes to compensate for delay imbalances. Nevertheless, *emap* still achieves better area, and the difference in delay is small and can be recovered with post-mapping re-synthesis over the critical paths.

Additionally, *emap* finds delay-area Pareto point results in 20 out of 21 benchmarks, being dominated in both area and delay in only one case. Conversely, ABC finds delay-area Pareto points in 15 out of 21 benchmarks, being dominated in both area and delay by *emap* in six cases. This shows that the two mappers explore different solutions within the design space.

Mapping for best area using the ASAP 7*nm* standard cell library

Similarly to Table 4.8, in this experiment we compare *emap* against the mappers in ABC for the EPFL and IWLS 2005 benchmarks using the ASAP7 standard cell library. However, this experiment is carried out using a large timing constraint of 11 times the minimal one to test the area minimization algorithms of the different mappers.

Table 4.11 shows the experimental results. Our mapper achieves the best area in almost every benchmark with a 15.87% improvement compared to *map* and 16.51% compared to *&nf*, while obtaining better run time. Additionally, we also report the delay results for each mapper.

Mapping for best area using an industrial 28*nm* standard cell library

Similarly to Table 4.11, in this experiment we compare *emap* against the mappers in ABC for the EPFL and IWLS 2005 benchmarks for area-oriented mapping. However, this experiment

Table 4.11: Area-oriented technology mapping comparing mappers in ABC and a mapper implementing the algorithms proposed in this chapter using the ASAP7 library.

Benchmark	**Baseline**		**ABC *map -a***			**ABC *&nf -p -R 1000***			**Our work *emap -a***		
	Size	Depth	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
adder	1020	84.03	57.40	3548.84	0.13	100.62	2577.43	0.15	57.40	3548.84	0.02
bar	3336	216.32	191.81	238.53	0.18	159.64	211.47	0.12	128.74	199.39	0.05
div	51725	4673.59	3427.10	66322.78	2.55	5044.47	44201.14	1.49	3079.58	53745.74	1.13
hyp	214335	15237.73	14378.94	300243.28	11.83	15539.40	196733.14	7.29	13167.04	272523.78	6.21
log2	31950	1737.34	1643.99	6734.09	3.42	2023.00	3941.43	4.78	1501.99	5409.33	1.55
max	2865	172.77	166.18	2862.23	0.26	224.39	2213.23	0.15	139.06	2952.58	0.19
multiplier	26953	1654.58	1495.54	4961.61	1.84	1850.21	2738.95	0.95	1284.50	3448.00	1.09
sin	5399	407.4	309.11	3056.93	0.68	298.91	2820.36	0.88	270.05	2782.29	0.36
sqrt	24216	1683.2	1461.52	106239.08	1.45	1831.71	48222.85	2.24	1336.59	102003.83	0.83
square	18302	1084.48	1168.02	3711.58	1.35	1143.31	2510.24	0.66	1048.11	3508.29	0.57
arbiter	11839	766.32	569.40	1018.69	1.34	566.69	1018.69	0.97	557.72	999.87	1.04
cavlc	690	37.52	39.01	221.56	0.09	37.73	217.37	0.10	34.13	263.16	0.02
ctrl	170	7.85	8.07	131.57	0.08	8.35	133.32	0.09	7.20	127.51	0.00
dec	304	27.44	27.50	85.83	0.12	27.38	85.83	0.14	27.06	86.33	0.03
i2c	1275	69.38	77.90	214.29	0.14	74.87	233.04	0.15	67.16	267.86	0.04
int2float	235	11.31	13.00	205.02	0.09	12.92	209.05	0.09	11.04	197.60	0.01
mem_ctrl	46820	2427.28	2673.88	1799.50	2.33	2624.15	1778.17	1.90	2278.47	1706.20	1.66
priority	863	82.25	62.56	2795.01	0.12	79.77	2501.95	0.11	51.17	2918.99	0.03
router	257	18.41	15.01	448.06	0.10	13.16	397.66	0.12	12.96	400.89	0.04
voter	13029	1538.53	851.87	1297.25	1.02	834.67	1128.34	0.64	802.13	1185.61	0.93
ac97_ctrl	14241	11	843.50	200.11	0.42	814.65	203.37	0.30	651.48	189.23	0.18
aes_core	21376	21	1247.41	385.16	1.24	1239.28	401.15	0.96	1098.21	368.51	0.95
des_area	4808	27	263.19	537.25	0.48	247.69	512.26	0.48	230.67	491.67	0.28
des_perf	80101	17	5465.26	345.29	6.00	5328.04	331.43	5.67	4639.28	319.74	3.65
DMA	24278	21	1395.02	434.69	1.42	1321.40	408.02	1.32	1103.13	355.39	0.94
DSP	45004	52	2598.94	977.43	3.20	2468.64	919.30	3.54	2114.64	938.35	1.82
ethernet	86576	27	5124.71	565.67	6.21	4818.10	476.60	5.84	3720.73	492.35	3.56
iwls05_i2c	1132	12	68.07	251.97	0.12	65.30	232.61	0.11	57.82	247.68	0.04
leon2	788859	45	53556.03	820.28	106.44	51439.15	776.81	120.68	41732.05	754.24	90.91
leon3_opt	973338	51	57367.05	906.88	123.57	55515.76	867.16	157.00	48307.89	789.17	85.85
leon3	1087244	47	66424.74	869.10	125.82	63118.07	834.30	129.35	51309.65	745.15	87.16
leon3mp	651721	47	39221.58	927.04	67.03	36829.42	877.81	63.50	30191.65	806.49	44.63
iwls05_mem_ctrl	15048	29	772.76	571.96	1.49	732.12	534.52	1.74	644.41	550.57	1.02
netcard	803417	36	48046.33	608.63	75.74	45229.71	635.83	78.75	34460.36	588.33	58.48
pci_bridge32	22705	22	1406.23	474.02	1.28	1322.72	467.52	1.01	1121.24	385.79	0.63
RISC	74651	36	4247.88	765.35	3.92	4036.42	684.59	3.53	3533.69	668.20	2.01
sasc	770	8	48.03	148.58	0.10	45.75	151.43	0.09	35.48	140.61	0.01
simple_spi	1039	10	62.25	192.83	0.10	60.05	175.63	0.11	48.55	203.07	0.02
spi	3760	31	216.03	533.14	0.33	207.97	522.13	0.32	180.45	465.15	0.13
ss_pcm	405	7	28.23	143.90	0.08	27.49	148.33	0.08	22.34	128.21	0.00
systemcaes	12242	44	726.39	860.58	0.65	694.64	770.41	0.46	558.74	769.85	0.28
systemcdes	2877	23	185.60	468.71	0.26	179.82	428.12	0.26	150.86	435.87	0.12
tv80	9461	43	545.34	832.24	0.74	519.43	831.59	0.77	460.87	770.59	0.38
usb_funct	15715	23	924.28	452.48	0.78	891.83	445.49	0.64	781.38	398.10	0.41
usb_phy	452	9	30.49	171.61	0.09	29.57	144.43	0.08	27.00	157.39	0.01
vga_lcd	126636	18	8092.58	361.80	11.53	7516.49	377.67	10.43	6069.10	342.92	7.00
wb_conmax	47520	18	2361.62	369.91	4.15	2220.53	348.03	3.54	2096.30	329.90	2.23
Total					572.31			613.58			**408.50**
Reduction						-2.17%	9.61%		**15.87%**	**6.99%**	

uses a commercial $28nm$ standard cell library, which mitigates the advantages of hybrid matching since it contains cells up to 6 inputs.

Table 4.12 shows the experimental results. Also in this experiment our mapper largely achieve the best area in almost every benchmark with a 10.24% improvement compared to *map* and 8.16% compared to *&nf*, while obtaining better run time. Additionally, we also report the delay results for each mapper.

4.7 Summary

In this chapter, we proposed methods to overcome the limitations of current state-of-the-art technology mappers for standard-cell-based design. Fist, we introduced a novel matching

Table 4.12: Area-oriented technology mapping comparing mappers in ABC and a mapper implementing the algorithms proposed in this chapter using a $28nm$ commercial library.

Benchmark	**Baseline**		**ABC *map -a***			**ABC *&nf -p -R 1000***			**Our work *emap -a***		
	Size	Depth	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)	Area (μm^2)	Delay (ps)	Time (s)
adder	1020	84.03	375.87	2284.33	0.1	366.06	2029.36	0.1	337.55	2560.36	0.02
bar	3336	216.32	856.08	114.82	0.2	828.3	116.29	0.2	760.61	125.36	0.07
div	51725	4673.59	17624.05	35829.66	2.74	21763.22	19201.46	18.71	15834.6	23840.95	1.23
hyp	214335	15237.73	76654.84	151990.69	15.27	94276.13	89353.77	18.15	66647.15	167634.52	7.22
log2	31950	1737.34	9157.06	2989.39	3.03	8904.6	2797.39	137.42	8440.25	2805.58	1.51
max	2865	172.77	844	1325.56	0.27	796.51	1546.34	0.29	773.05	1551.2	0.25
multiplier	26953	1654.58	7756.58	1994.23	2.13	7589.57	1970.18	6	6942.06	1938.19	0.95
sin	5399	407.4	1722.94	1432.1	0.59	1642.97	1412.49	2.96	1494.39	1362.79	0.29
sqrt	24216	1683.2	7339.48	31852.83	1.6	10576.89	21948.81	1.01	6966.77	41640.97	0.81
square	18302	1084.48	6350.65	1907.74	1.46	6024.63	2268.19	3.94	5443.06	2187.8	0.65
arbiter	11839	766.32	3146.76	399.02	1.41	3212.76	398.66	1.45	3135.79	398.95	1.87
cavlc	690	37.52	204.02	111.46	0.1	191.31	119.51	0.11	183.57	122.78	0.02
ctrl	170	7.85	41.88	60.42	0.08	41.26	66.14	0.09	38.45	65.29	0
dec	304	27.44	117.46	29.21	0.1	116.9	30.98	0.13	116.9	30.98	0.03
i2c	1275	69.38	393.04	103.58	0.13	377.5	108.38	0.13	355.9	103.76	0.04
int2float	235	11.31	67.48	86.84	0.08	64.86	103.44	0.1	61.06	92.23	0.01
mem_ctrl	46820	2427.28	13515.31	854.09	2.19	13187.28	938.18	2.09	11901.52	935.74	1.67
priority	863	82.25	298.52	1132.28	0.1	275.81	1499.92	0.11	258.7	1614.15	0.03
router	257	18.41	84.03	175.34	0.09	80.49	238.54	0.09	76.85	194.27	0.02
voter	13029	1538.53	4110.95	578.31	1.02	4029.04	616.14	5.84	3868.29	645.91	0.6
ac97_ctrl	14241	11	4087.06	83.8	0.47	3930.97	104.02	0.33	3672.19	99.86	0.23
aes_core	21376	21	6434.22	178.01	1.09	6280.62	191.4	0.87	5924.81	172.71	0.95
des_area	4808	27	1362.16	244.22	0.63	1296.5	237.05	0.36	1251.99	223.18	0.3
des_perf	80101	17	27852.02	156.79	5.6	27091.65	157.84	4.67	24641.65	154.73	4.43
DMA	24278	21	6749.92	170.05	1.32	6576.28	192.92	1.13	6049.54	179.49	0.79
DSP	45004	52	12847.23	474.54	2.81	12270.53	459.88	2.9	11373.52	433.21	1.73
ethernet	86576	27	24216.97	193.45	4.08	23778.47	235.17	3.02	20299.31	221.09	2.99
iwls05_i2c	1132	12	349.02	88.2	0.11	331.7	100.69	0.12	313.56	92.54	0.04
leon2	788859	45	271761.88	353.69	52.42	262458.59	357.5	85.61	246587.02	354.86	56.55
leon3_opt	973338	51	275260.66	361.19	66.58	263908.06	401.34	64.06	248699.8	369.91	68.61
leon3	1087244	47	315741.41	350.78	68.63	299940.06	385.99	69.8	286991.09	357.78	69.26
leon3mp	651721	47	187435.73	366	39.06	178476.89	395.38	39.65	170576.23	389.75	34.3
iwls05_mem_ctrl	15048	29	4090.16	215.3	0.97	3926.52	236.88	1.16	3678.84	214.65	0.97
netcard	803417	36	221703.8	270.64	46.1	213137.97	288.62	37.66	193249.52	290.26	36.26
pci_bridge32	22705	22	6555	171.58	1.21	6300.16	182.46	0.91	5758.37	176.96	0.6
RISC	74651	36	21513.6	284.33	3.7	20377.79	337.15	2.79	19039.07	320.35	2.17
sasc	770	8	229.08	59.16	0.09	226.09	73.95	0.09	202.89	63.44	0.01
simple_spi	1039	10	305.55	75.53	0.1	293.83	85.42	0.11	272	82.72	0.02
spi	3760	31	1080.05	228.7	0.29	1055.59	260.35	0.3	988.09	237.36	0.14
ss_pcm	405	7	136.12	49.8	0.08	121.58	69.15	0.09	111.2	61.4	0
systemcaes	12242	44	3653.4	356.33	0.63	3360.44	358.42	0.75	3018.73	367.3	0.34
systemcdes	2877	23	918.75	189.05	0.28	864.82	216.34	0.27	805.23	203	0.12
tv80	9461	43	2750.88	369.34	0.65	2615.18	367.78	0.7	2430.16	394.56	0.41
usb_funct	15715	23	4681.08	189.08	0.71	4451.22	225.73	0.56	4190.76	191.83	0.41
usb_phy	452	9	154.86	61.38	0.09	146.94	71.99	0.1	141.52	79.65	0.01
vga_lcd	126636	18	36607.27	141.52	7.11	34983.64	149.69	5.29	31371.96	145.47	4.72
wb_conmax	47520	18	12107.7	158.31	3.11	11804.37	171.31	3.18	11345.1	167.73	2.16
Total					340.61			525.40			**305.81**
Reduction						1.74%	-7.08%		**10.24%**	-6.58%	

method that combines pattern and Boolean matching. Its main advantage is to enhance the performance of Boolean matching enabling the use of large library cells, such as multiple-input AND-ORs. Experimental results have shown a significant decrease of the average area by 6.5% for similar delay. Second, we focused on increasing the support of multiple-output cells in technology mapping. We proposed algorithms to tackle the structural multiple-output cell detection problem and multiple-output cell covering, building the first open-source mapper with this kind of support. We demonstrated that multiple-output cells can be efficiently detected and mapped, with a 7.48% area decrease when mapping for minimal delay. Additionally, our approach reduces the area of approaches that separate the multiple-output cell mapping from single-output cell mapping by a remarkable 5%. Third, we discussed methods to improve area-oriented covering algorithm of technology mapping, showing a 4.66% average improvement compared to ABC command *map*.

In the last part of the chapter, we presented the technology mapper *emap*, which enhances the state-of-the-art algorithms for technology mapping [38] using the methods proposed in this chapter. When using the ASAP 7nm library, compared to the mappers in ABC, we showed a reduction in area by 9.16% and in delay by 2.95% before buffering and gate sizing. After buffering and gate sizing, the results maintain a 9.22% better area and 2.59% better delay. When using the commercial 28nm library without large cells, *emap* achieves 5.75% better area and 1.38% better delay. However, the results slightly degrade after buffering and gate sizing with a 2.77% better area and 0.63% worse delay. Nevertheless, *emap* achieves delay-area Pareto point solutions on 20 out of 21 benchmarks, while ABC *&nf* in only 15 out of 21. Additionally, *emap* achieves large reductions (> 8%) in area-oriented mapping for both standard cell libraries compared to the mappers in ABC, showing the effectiveness of the algorithms proposed in this chapter.

5 Mapping for Logic Synthesis

Chapter 3 and 4 were dedicated to technology mapping algorithms for field-programmable gate arrays (FPGAs) and standard-cell-based design. In this chapter, we study how approaches similar to technology mapping and innovations in logic rewriting can enhance technology-independent logic synthesis. Specifically, this chapter presents: (i) a versatile mapping approach for graph mapping and logic rewriting of technology-independent graph representations; (ii) algorithms to leverage don't care conditions in graph mapping and logic rewriting; (iii) efficient methods to optimize the factored form literal count in multi-level Boolean networks, with applications in standard-cell-based design flows and transistor-level synthesis. The content of this chapter is largely based on the publications in [196, 200, 201, 202].

The remainder of this chapter is organized as follows. First, we present the motivations of this chapter in Section 5.1. Next, Section 5.2 proposes a versatile mapping algorithm for graph mapping and logic rewriting of technology-independent graph representations. This approach can efficiently map into different graph data structures while optimizing them and can perform logic rewriting with a global view. The experimental results show that versatile mapper can map *and-inverter graphs* (AIGs) to *majority-inverter graphs* (MIG), *xor-and graphs* (XAGs), and *xor-majority graphs* (XMGs) and reduce the number of gates by 32.11%, 27.58%, 43.17%, respectively. Additionally, we show that mapping based rewriting performs better than previous state-of-the-art methods. Then, Section 5.3 presents algorithms to efficiently leverage Boolean don't care conditions during graph mapping and logic rewriting. The experimental results show that an MIG flow implementing don't care-based logic rewriting reduces the number of gates by 4.31% compared to the state-of-the-art flow. Moreover, we show that this method contributes to obtain the best-known results in MIG size for the EPFL benchmarks. Next, Section 5.4 proposes modern algorithms to optimize the factored form literal count (FFLC) in multi-level Boolean networks. This is motivated by the correlation between factored form literals and the number of transistors in the CMOS implementation. We propose a portfolio of methods that includes mapping, rewriting, resubstitution, and refactoring for FFLC. We show that these methods help reduce the area of standard-cell-based designs by 2.8%, on average, compared to the state-of-the-art AIG synthesis flow. Moreover, we discuss applications in transistor-level synthesis and auto-creation of standard cells. Finally, Section 5.5 concludes and summarizes this chapter, highlighting the key findings and contributions.

5.1 Motivation

The performance of modern integrated circuits is largely affected by the capabilities of logic synthesis tools. Traditionally, Boolean networks have been represented using networks of *sum-of-products* (SOPs) or *and-inverter graphs* (AIGs). However, over the last decade, several other logic representations have been proposed to enhance the quality of synthesis tools. Hence, modern logic synthesis should become more versatile by supporting multiple logic representation, different cost function, and applications.

Originally, logic synthesis utilized 2-input NANDs and NORs, together with inverters, as primitives in graph representations thanks to their universality. Then, the number of literals in the factored forms became the standard metric for area in technology-independent synthesis, leading to the development of many optimization methods for this metric [28]. Consequently, circuits were modeled as logic networks where nodes are represented in SOP form. Although SOPs and factored forms are not unique, heuristics were developed to compute a good factorization and the literal count. As logic synthesis evolved and integrated circuits became larger and more complex, scalability emerged as a crucial issue. This led to the adoption of *and-inverter graphs* (AIGs) [75, 97], consisting of 2-input AND gates and inverters, as the most common technology-independent representation. The simplicity of AIGs allowed for efficient representation and the development of scalable and effective optimization algorithms [132]. With the transition to AIGs, the cost metric shifted from the number of factored form literals to the number of AIG nodes and the AIG logic depth. In addition to AIGs, *majority-inverter graphs* (MIGs) [5, 6], consisting of 3-input majority gates and inverters, have been proposed as an alternative representation. MIGs were motivated by a more expressive potential and by many majority-based emerging technologies, e.g., spin-wave devices [91], quantum-dot cellular automata [116], and adiabatic quantum-flux parametron [191]. Optimization algorithms based on majority Boolean algebra have been developed for MIGs, leading to significant delay reductions in arithmetic-intensive designs, even for conventional technologies.

This development has sparked further research into other logic representations for applications in logic synthesis. For instance, *xor-and graphs* (XAGs) [72] and *xor-majority graphs* (XMGs) [68] have been proposed for their compactness in arithmetic circuits and as a basis for logic rewriting. Additionally, XAGs have been used in design flows for FPGAs, security applications (see, e.g., [27]), cryptography applications (see, e.g., [205, 219]), and quantum computing (see, e.g., [130]). Recent work has also investigated 3-input gates as new graph representations to address logic synthesis [127]. The proliferation of various graph representations has motivated the development of versatile logic synthesis tools that support optimization across multiple representations. A notable example is the logic synthesis toolbox introduced in [169], which facilitates optimization over a range of graph-based representations.

Since different graph representations and cost functions unique to different applications are available to support logic synthesis, in Section 5.2 we present a versatile mapping technique to map from a graph representation to another and to perform global Boolean rewriting. This approach is referred to as *graph mapping*. Graph mapping performs mapping using a database of optimum graph structures obtained with exact synthesis. One of its main advantages is its versatility since its target graph representation depends on a database. Our research on graph mapping produced remarkable results in obtaining compact MIGs, XAGs, and XMGs, and has been adopted in multiple state-of-the-art flows and projects for these representations [20, 104, 107, 109, 110, 129, 164, 194].

In Section 5.3, we extend versatile graph mapping and Boolean rewriting to leverage additional degrees of freedom in optimization given by *don't care* conditions. Don't cares in logic synthesis are typically leveraged by Boolean resubstitution. However, resubstitution is based on sophisticated resynthesis heuristics, which can be hard to design and, consequently, sub-optimal in many non-conventional data structures, including MIGs. Conversely, Boolean rewriting can be a viable solution for exploiting don't care conditions with local optimality guarantees, but this is possible only when exact synthesis is used to compute optimum replacement on the fly [167]. However, exact synthesis is computationally expensive, making it impractical in industrial tools and common design workflows. For this reason, optimum structures are typically pre-computed and stored in a database, commonly limited to four inputs, but this method does not support the use of don't cares. We address this limitation by proposing a technique to enable the usage of don't cares in graph mapping and logic rewriting with pre-computed databases. We demonstrate how to process the database and perform Boolean matching with Boolean don't cares, with negligible run time overhead. Moreover, we show that this method contributes to obtaining the best-known results in MIG size for the EPFL benchmarks.

As modern logic synthesis has moved to using AIGs as a scalable and efficient logic representation, the cost metric transitioned from the number of factored form literals to the number of AIG nodes. However, the *factored form literal count* (FFLC) remains significant because it correlates strongly with the number of transistors needed to implement a Boolean function in CMOS technology. Consequently, FFLC optimization is a powerful tool for a fully custom design flow, optimizing logic for transistor-level mapping, which inherently produces complex custom standard cells. Surprisingly, the connection between factored form literal optimization and AIG optimization has not been extensively studied. Moreover, traditional FFLC optimization methods are restricted to small Boolean functions only, motivating research into scalable FFLC optimization methods that can be applied at the logic-network level. In Section 5.4, we study the connection between AIG optimization and FFLC optimization and propose a new scalable framework for FFLC optimization. We show how to perform FFLC optimization directly over an AIG without converting AIGs into logic networks, as required by traditional FFLC minimization techniques. Additionally, we present a portfolio of AIG-based optimization techniques enhanced for FFLC optimization. This is the first approach to address FFLC optimization at the global network level. We demonstrate that AIG-based FFLC optimization can improve the design flow for standard cells. Furthermore, we discuss applications in transistor-level synthesis and custom standard-cell creation. These latter applications are further addressed in the patents [203, 213].

5.2 Versatile Graph Mapping

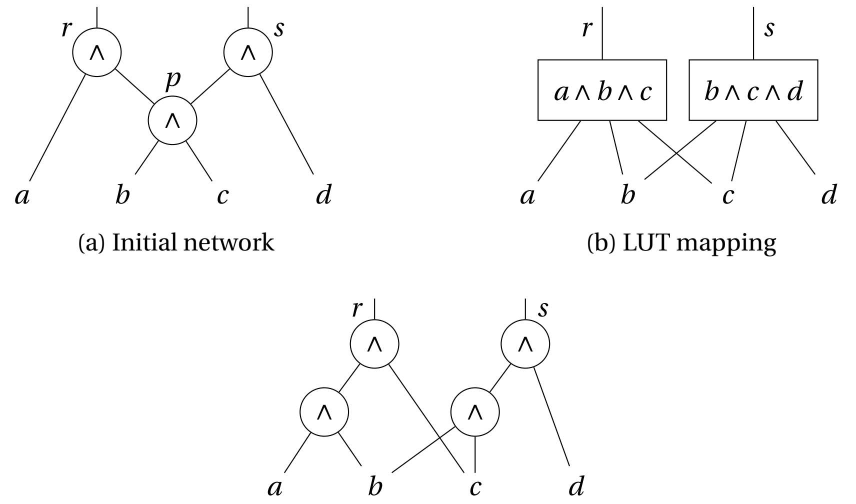
In this work, we present a mapping approach, called *graph mapping*, that supports mapping from and to different graph data representations, such as AIG, XAG, MIG, and XMG. Informally, graph mapping performs technology-independent mapping. Additionally, it can perform global logic restructuring. Its versatility finds extensive applications in different technologies. For instance, the *Adiabatic Quantum Flux Parametron* (AQFP) [33, 129] superconducting technology, and quantum-dot cellular automata [94] are inherently majority-based. Our tool provides an efficient rewriting to MIGs that is crucial for specialized tools such as [204] for AQFP design. Re-configurable nano-technologies (RFET) make use of XMGs as an efficient representation to preserve self-duality [164]. In cryptography and security applications, XAGs are used to represent circuits and analyze the multiplicative complexity of Boolean functions which correlates with vulnerability against algebraic attacks [27]. Furthermore, additional applications are possible for logic optimization (e.g., logic rewriting), especially in arithmetic-intensive circuits. Since publicly available logic synthesis tools mostly rely on AIGs for logic optimization [30], this mapper provides a way to easily obtain a representation that is more suitable for a particular application while optimizing it. Additionally, we present technical improvements over previous logic restructuring methods on *logic sharing* and *global view*.

In the experiment, we evaluate the versatility and quality of graph mapping and compare it to state-of-the-art methods:

- We evaluate graph mapping for logic restructuring on MIGs. We test our solutions to improve logic sharing and optimize with a global view by comparing to previous state-of-the-art LUT-based rewriting and cut rewriting. Our mapper improves the average size by 9.45% and 20.64% respectively obtaining considerably better results for all the benchmarks.

- We evaluate mapping from AIGs into XAGs and XMGs. We improve previous work on XMG size optimization using LUT-based rewriting in [43, 68] by 12.22% in geometric mean and 27.45% in size-depth product.

In summary, graph mapping is the first tool to enable graph mapping and restructuring among various graph-based representations.



(a) Initial network

(b) LUT mapping

(c) LUT decomposition with exact synthesis

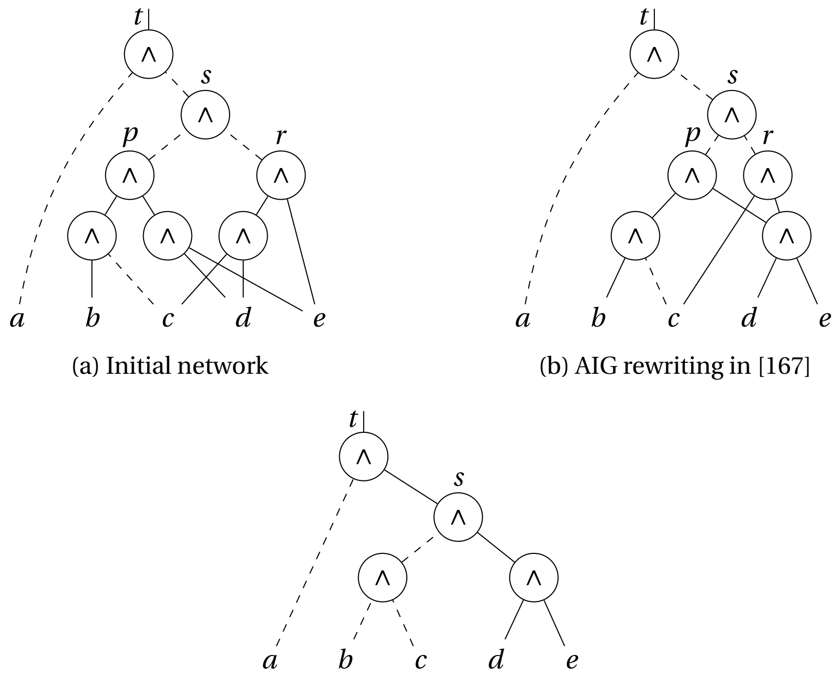
Figure 5.1: Logic sharing limitation in LUT-based rewriting

5.2.1 Related Work on Graph Mapping

LUT mapping [45] is a special case of technology mapping which covers a network using LUTs. State-of-the-art technology-independent mapping, here named *graph mapping*, relies on LUT mapping followed by a k-LUT decomposition using exact synthesis to obtain the target graph representation [70]. We refer to this method as LUT-based mapping. LUT-based mapping is often used for graph mapping and logic rewriting by iteratively remapping the circuit. Previous work implemented optimization flows that used LUT mapping and exact k-LUT decomposition on MIGs [70] and XMGs [68]. LUT-based mapping suffers from a limitation that decreases the quality of results. LUT mapping aims at mapping a network by minimizing the number of LUTs or LUT levels. By preferring larger LUTs to cover more logic, the *logic sharing* of the original network is often lost. Hence, when the LUTs are decomposed using exact synthesis, more nodes than necessary are added to the network.

Example 5.2.1. *In Figure 5.1a, an AIG network contains a shared node p. When the network is mapped to a 3-LUT network for size reduction, the network obtains the configuration in Figure 5.1b using the minimum number of two LUTs to cover the network. This operation loses the local information of the shared node p. When the LUTs are decomposed back to an AIG using exact synthesis, shown in Figure 5.1c, the two LUTs are matched to the same structure which creates an additional node with respect to the original network. To describe structurally the logic sharing, a better mapping would use one 2-LUT for each node of Figure 5.1a.* ▲

To restructure a circuit, another method is also available in the literature. Rewriting [137, 167] is a DAG-aware optimization method that aims at minimizing the size of a representation by replacing small parts of the network with smaller structures. The DAG-aware property makes it able to re-use existing logic and leverage structural hashing [139]. The logic structures



(a) Initial network

(b) AIG rewriting in [167]

(c) Best structure with a global view

Figure 5.2: Local view limitation in cut rewriting

are typically contained in a database or are computed on the fly. We analyze the DAG-aware cut rewriting approach described in [167] that can be used for graph mapping. The algorithm greedily collects the best local replacements over the whole graph, saving for each node the one with the best node reduction. Then, a cover using the best replacements is extracted in reverse topological order. However, local decisions create conflicts (e.g., two replacements cannot happen at the same time). The algorithm lacks of a *global view* to extract the best replacements globally.

Example 5.2.2. *Figure 5.2a shows the initial AIG network in which dashed lines represent negations. By rewriting the network using* 4*-feasible cuts, the best structure is obtained by replacing the cut with leaves* $\{b,c,d,e\}$ *rooted in* s*. The usage of this replacement depends on the replacement at the PO node* t*. In Figure 5.2b, cut rewriting heuristic selects the cut with leaves* $\{a,p,r\}$ *at root* t *since it has a greater or equal local gain compared to the other candidates at* t*. Consequently, the best replacement at* s *cannot be used since* s *is already included in the chosen cut at* t *(i.e.,* s *is not a leaf of a cut during covering). Cut rewriting replaces the sub-graphs rooted in* p *and* r*, thus leading to a size improvement of a single node. The optimal outcome, in Figure 5.2c, can be achieved by evaluating the conflicts globally. Alternatively, cut rewriting would need to be executed a second time to achieve this optimization. However, in large designs, cut rewriting tends to get stuck in local minima due to the lack of a global view.* ▲

5.2.2 Versatile Mapping

In this section, we describe our contribution. We present a versatile graph mapper that can map a generic technology-independent representation (e.g., AIG, XAG, MIG) into another while performing optimization. It uses a database of pre-computed optimum structures (e.g., obtained using exact synthesis) to map or rewrite the network. Our approach combines and extends state-of-the-art technology mapping [38] and logic rewriting [68, 167].

Our mapper implements the best characteristics of these two methods and addresses the LUT-based mapping and cut rewriting drawbacks. Boolean matching is used to bind the cuts to the available structures or primitives. Thus, accurate decomposition costs (size and depth) are available during mapping. The cover is minimized using size and depth instead of the number of LUTs and LUT levels. This helps to better exploit shared logic as compared to LUT-based mapping (e.g., our mapper maps each node in Figure 5.1a with a LUT since this cover has a lower decomposition cost than the one in Figure 5.1b, preserving the shared node p). The mapper executes multiple mapping refinements, from global to local optimization. In this way, the mapper generates the cover globally accounting for shared logic and then optimizes it locally, in the MFFCs. This approach helps choose better replacements with a global view (e.g., our method achieves the structure in Figure 5.2c when mapping from the structure in Figure 5.2a). Our mapper does not need to rewrite each cut. Nevertheless, an option exploits structural hashing during the last iteration to find shared nodes among the structures. This typical feature of rewriting is controlled by technology mapping algorithms to select the replacements.

The mapper is implemented in a flexible parameterized way so that it can switch to different cost functions for delay-oriented or area-oriented mapping. Algorithm 5.1 shows the mapping steps. The mapper maps for delay by executing a delay-oriented mapping followed by area-recovery iterations. Area-oriented mapping is achieved by bypassing the delay-oriented iteration or by relaxing the required time constraint. In this section, the terms area and delay are equivalently used as size and depth, respectively.

The algorithm can be summarized in the following four steps:

- Library generation
- Cut enumeration
- Boolean matching
- Mapping

Library generation

We define a library as a hash table that is used to classify structures for simple and fast Boolean matching. Given a Boolean function represented as a truth table, the library returns, if possible,

Algorithm 5.1: Versatile Mapper

```
Input: Boolean network N, cut size k, library, cut_sorting_func, constraints, skip_delay,
       AreaGlobalIter, AreaLocalIter, AreaStrashIter, rw_limit
Output: mapped network M
cuts ← compute_cuts(N, k, cut_sorting_func)
match_cuts(cuts, library)
if not skip_delay then
    delay_oriented_map(N, cuts)
repeat AreaGlobalIter times
    compute_required_times(N, cuts, constraints)
    global_area_oriented_map(N, cuts)
repeat AreaLocalIter times
    compute_required_times(N, cuts, constraints)
    local_area_oriented_map(N, cuts)
M ← new_network()
foreach primary input i ∈ N do
    create_input(M, i)
if graph mapping and AreaStrashIter then
    compute_required_times(N, cuts, constraints)
    local_area_strash_oriented_map(N, cuts, M, rw_limit)     ▷ Enhanced exact area
    remove_dangling(M)                                  ▷ Remove non-reachable nodes
else
    finalize_network(N, cuts, M)
return M
```

a set of structures that can implement that function.

Based on the target graph representation, a database of structures is generated using SAT-based exact synthesis. We utilize the algorithms described in [69], incorporating *single selection variable* (SSV) or *multiple selection variable* (MSS) encodings, to generate multiple optimum structures for each k-input $\mathcal{NPN}$ class ($k = 4$ in our experiments). Alternatively, also enumeration can be used. The pre-computed structures, classified into $\mathcal{NPN}$-equivalence classes, are saved in a Boolean matching library. Each class functionality is expressed by a representative truth table, which is computed by finding the lexicographically smallest truth table in the class. Contrarily to the one in Chapter 4 for standard cells, $\mathcal{NPN}$-configurations by permuting and negating variables are not enumerated for scalability reason. Note that in the worst case the number of configurations generated would be $k! \times 2^{k+1}$ for a k-input structure. Consequently, functions are matched to structures by canonicalization (more details in the Boolean matching subsection). For each structure, the pin-to-pin delay and the area are computed given a cost function. The pin-to-pin delay describes the depth of the longest path from an input pin to an output pin. The area is defined as the size of the structure. Additionally, also inverter costs are supported.

Cut enumeration

Cut enumeration computes a set of k-feasible cuts for each node in the subject graph (line 1 of Algorithm 5.1). The computation proceeds in topological order from the primary inputs (PIs) to the primary outputs (POs) as in [140]. The cut computation is independent of the graph representation and works for nodes with a variable number of inputs, as shown in Equation 2.5.

For each non-trivial cut, the corresponding truth table is computed. Truth tables are minimized by reordering variables and removing the ones without a functional support. This process eliminates "holes" in the truth table that prevent cuts from matching with the gates (whose truth table is minimized). In this case, the support of the cuts is reduced accordingly. An average of 0.06% additional cuts can be matched in the EPFL benchmark suite [3]. This number is not so small when considering the large amount of cuts that are typically generated. Moreover, these cuts are often good since they include don't care conditions.

Example 5.2.3. *A cut C with leaves $\{l_1, l_2, l_3, l_4\}$ and truth table* 0x0f05*, in hexadecimal format, can be minimized to cut C' with leaves $\{l_1, l_3, l_4\}$ and function* 0x31 *since l_2 in C does not have functional support.* ▲

During the enumeration phase, cuts are sorted on the fly based on their depth, area flow [40, 124], and size, under the unitary model. The cut prioritization is selected depending on the desired goal of the mapping. For a delay-oriented mapping, the sorting function primarily sorts for the depth while for area-oriented mapping, it orders primarily for area flow. To decrease the number of candidate cuts at each node, only a small number l is selected. On top of that, the trivial cut is added. This guarantees that at most $l+1$ cuts are saved at each node, so, for a node with fan-in size equal to m, a maximum of $(l+1)^m$ cuts are enumerated. This technique is referred to as priority cuts [47].

Boolean matching

Given a cut and the corresponding truth table, Boolean matching finds a set of gates that can implement that function. The pre-computed library of structures discussed in the library generation sub-section is used to achieve that. The library stores the database of structures in $\mathcal{NPN}$ classes. Boolean matching is achieved using function canonicalization to get the $\mathcal{NPN}$ class representative. The canonicalization procedure finds the lexicographically smallest truth table (the $\mathcal{NPN}$-class representative), the permutations, and the input negations to apply.

Mapping

Delay-oriented mapping aims at covering the subject graph by selecting the gates that minimize the arrival time at each node. The computation (line 4 of Algorithm 5.1) proceeds in topological order (from PIs to POs), over the internal nodes of the subject graph. For each node, the cut and the structure with the best arrival time is selected. The area overhead is then addressed during area recovery once the required times at the nodes are known. Area-oriented mapping or area recovery are performed in multiple passes over the nodes in the subject graph. We use a first heuristic called *area flow* [40, 124] for a global area optimization (line 7 in Algorithm 5.1) and a second method called *exact area* [47] for a local area optimization (line 10 in Algorithm 5.1). Our algorithm maps and adjusts the cover using these two methods iterated multiple times if necessary. The area passes are constrained by the required time so that the worst-case delay is not increased. Depending on the mapping phase, the cost criteria to select the best gate are shown in Table 5.1.

We extend exact area, detailed in Algorithm 4.1, by incorporating an option for high-effort area optimization in graph mapping to exploit structural hashing for identifying shared nodes (lines 14-17 in Algorithm 5.1). Exact area is a local refinement of the cut selection which is driven by the area in the MFFC. The area is locally reduced by selecting a cut so that the sum of the area of the best cuts in the MFFC is minimized. Given a current cover of the subject graph, the exact area for a node n can be computed using recursive cut referencing and dereferencing procedures. A recursive cut referencing (dereferencing) algorithm recursively explores the MFFC of a node in the cover and counts its area. The last local area iteration for graph mapping may include a rewriting of the l best cuts per node with structural hashing, called *rw_limit* in Algorithm 5.1, to find shareable nodes among the possible structures. In topological order, for each node, a candidate match is inserted in the network using one-level structural hashing by permuting and negating the inputs according to the $\mathscr{NPN}$ transformation. Then, the number of added nodes is measured using node referencing and dereferencing similarly to rewriting [137]. Exact area is computed normally using the measured area value instead of the pre-computed area of the match. The match that minimizes the exact area at the node is selected. Structural hashing in exact area helps to select matches that share nodes with other structures in the cover. This method is particularly effective also when multiple alternative structures are available per $\mathscr{NPN}$ class and exact area with structural hashing is executed only on the previously selected best cut (*rw_limit* is 1).

Example 5.2.4. *Figure 5.1 showed an example where LUT-based rewriting is unsuccessful. Let us assume that we have a database with multiple alternative structures and a circuit with the cover shown in Figure 5.1b. Exact area with structural hashing would identify the solution in Figure 5.1a, instead of the one in Figure 5.1c, thanks to the precise counting of shared nodes. Hence, our mapping algorithm achieves better logic sharing both at a coarse-grained level using*

Table 5.1: Gates selection criteria

Mapping Phase	**Cost criterion**	**Tie-breaker 1**	**Tie-breaker 2**
Delay	arrival time	area flow	cut size
Global area	area flow	arrival time	cut size
Local area	exact area	arrival time	cut size

accurate costing of cuts, and at a fine-grained level by looking at sharing opportunities within possible implementations. ▲

In the finalization process, the resulting network is created using the computed cover and the associated gates (line 19 of Algorithm 5.1).

5.2.3 Experimental Results

In this section, we evaluate the versatility of the mapper and compare it to state-of-the-art methods. Although the original implementation described in [201, 202] can be used for conventional technology mapping for standard cells, we focus on graph mapping only since the standard-cell mapping problem has been extensively discussed in Chapter 4. We first use the graph mapper to convert AIGs to MIGs and perform logic rewriting. We compare our results to state-of-the-art LUT-based rewriting and cut rewriting methods. Next, we map from AIGs into XAGs and XMGs. As baseline for all the experiments, we use the EPFL combinational benchmark suite [3] containing combinational circuits provided as AIGs.

The graph mapper has been implemented in C++17 in the open-source logic synthesis framework *Mockturtle*[1] [183] as a command *map*. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified using the combinational equivalent checker in *ABC*[2].

Mapping into MIGs and logic restructuring

In this experiment, we compare our mapper to LUT-based rewriting and cut rewriting to optimize MIGs. The LUT mapping is realized with the synthesis package ABC using the command `&if -a -K 4` followed by a node resynthesis in Mockturtle that decomposes each LUT with a matching structure contained in a database of optimum-size structures. Rewriting is achieved using the standard cut rewriting algorithm in [167] implemented in Mockturtle. For the experiment, we use a database obtained with exact synthesis with size-optimum structures for the 4-input $\mathscr{NPN}$ classes. Up to 10 alternative structures are available for each $\mathscr{NPN}$ class. The mapper computes cuts of size 4 and stores up to 25 cuts per node. The versatile mapper is set for area-oriented mapping with one round of global area, two rounds of local area, and a high-effort round rewriting the two best-matched cuts, for a low impact on run time. The restructuring methods are iterated until no more improvement.

The results are shown in Table 5.2. We evaluate the results in terms of size reduction with respect to the baseline. Graph mapping achieves better results in all the benchmarks compared to previous methods, reducing the average number of majority gates from 10% more, compared to LUT-based rewriting, to 20% more, compared to cut rewriting. Additionally,

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

[2] Available at: https://github.com/berkeley-abc/abc

Table 5.2: Experimental results for mapping and rewriting MIGs

Benchmark	Baseline		LUT-based rewriting [70]			Cut rewriting [167]			Versatile mapper		
	Size	Depth	Size	Depth	Time (s)	Size	Depth	Time (s)	Size	Depth	Time (s)
adder	1020	255	385	130	0.05	893	129	0.08	384	129	0.06
bar	3336	12	2940	14	0.15	2952	15	0.71	2588	13	1.79
div	57247	4372	48827	4288	22.77	41553	2276	157.27	36858	2235	14.95
hyp	214335	24801	163398	9168	15.80	178736	9330	93.93	137048	8885	28.83
log2	32060	444	25651	247	3.91	30056	420	8.88	24295	206	3.20
max	2865	287	2446	248	0.35	2346	240	0.85	2171	162	0.96
multiplier	27062	274	20309	138	3.07	24829	271	12.37	19299	142	2.97
sin	5416	225	4560	159	0.44	5049	201	3.66	4196	122	1.14
sqrt	24618	5058	21002	6132	2.29	23889	4941	11.69	17355	3846	45.75
square	18484	250	14050	155	1.24	17669	163	8.85	11924	126	2.39
Total					50.09			298.27			102.05
Reduction			22.66%	25.10%		11.47%	20.54%		32.11%	41.88%	

exact area with structural hashing contributes to further improve the standard implementation by up to 10% more and 1.23%, on average, in number of gates. These results validate our motivations and solutions for exploiting shared logic and incorporating global optimization. Furthermore, our mapper achieves a reduction in depth that other methods cannot match.

Mapping into XAGs and XMGs and logic restructuring

For this experiment, we used the XAG and XMG databases of structures obtained using exact synthesis and containing a single structure per NPN class to fairly compare against the state of the art. We run the graph mapping for area mapping until convergence using cuts of size 4. The results are shown in Table 5.3. The baseline is the same as the one reported in Table 5.2 containing only AND gates. The geometric mean is computed over size and depth. In the table, we compare our results with previous work on LUT-based rewriting XMG optimization. We use the results in [68], using cuts of size 4, and in [43], using cuts of size 6 and on-the-fly exact synthesis combined with decomposition. Our mapper obtains considerably better results in all the benchmarks for XMG optimization compared to the work in [68]. Moreover, our mapper obtains better results compared to the best results in previous work [43, 68] in 7 out of 10 benchmarks with an improvement of 12.22% in geomean and of 27.45% in size/depth product compared to [43]. Note that these results used LUT-mapping and exact synthesis on-the-fly on cuts of size 6. Consequently, their method needs significant run time to compute the optimum structures while rewriting since complete 6-input databases are too big to be pre-computed and stored. This result shows again the advantage of our approach over LUT-based mapping.

5.3 Scalable Logic Rewriting Using Don't Care Conditions

Logic rewriting is a powerful optimization technique that iteratively rewrites small sections of a Boolean network with better implementations, typically evaluated in terms of size or depth. Many varieties of logic rewriting methods have been proposed in the literature, such as

Table 5.3: Experimental results for rewriting XAGs and XMGs

Benchmark	**XAG**		**XMG**		**XMG (k = 4) in [68]**		**XMG (k = 6) in [43]**	
	Size	Depth	Size	Depth	Size	Depth	Size	Depth
adder	639	256	383	128	639	130	383	128
bar	3013	13	2944	14	3281	16	2149	14
div	29124	4316	17613	2300	29607	4371	37003	4243
hyp	158682	24912	114746	8984	155349	12507	99428	8755
log2	24330	327	21361	204	27936	275	22957	213
max	2766	234	1845	157	2296	296	1938	200
multiplier	18651	268	15642	134	17508	154	16357	133
sin	4259	175	3728	138	5100	176	3896	140
sqrt	12617	6122	9750	2431	20130	6031	17187	5169
square	13876	247	11250	126	15070	130	8325	156
Average	26,795.7	3.687.0	19,926.2	1,461.6	27961.6	2408.6	20,962.3	1,915.1
GeoMean	2,293.1		1,511.2		2,117.8		1,721.6	
Size · Depth	98,795,745.9		29,124,133.9		66,697,987.8		40,144,900.7	

DAG-aware rewriting [137], cut rewriting [167], LUT-based rewriting [70], and mapping-based rewriting [201, 202] (Section 5.2). Typically, SAT-based *exact synthesis* [69] is used to compute optimum replacements for sub-networks. However, exact synthesis is computationally expensive and generally limited to synthesizing networks up to four inputs. In industrial applications, on-the-fly computation of replacements using exact synthesis is generally run time prohibitive, even for small logic blocks. Hence, structures are typically pre-computed and saved in a database. Logic rewriting with databases employs Boolean matching [123] to retrieve implementations from the database given a Boolean function. However, while Boolean don't cares are supported by on-the-fly exact synthesis, they are not supported by logic rewriting using a pre-computed database. In the academic tools ABC [30] and Mockturtle [183], size-optimum databases for logic rewriting contain up to 4-input networks. In [8], a delay-optimum database has been constructed using exact synthesis to generate up to 4-input networks.

Logic rewriting is very effective at optimizing many graph representations, such as the *majority-inverter graph* (MIG) [5], composed of three-input majorities and inverters, with many applications in standard-cells flows, and majority-based emerging technologies [129, 168].

This work presents improvements to logic rewriting by enabling the use of Boolean don't cares (DCs) with pre-computed databases. First, we present the notion of *don't care class* that is used to classify a database based on don't cares and permissible functions. This computation typically takes less than half a second for databases containing up to 4 input functions. Then, we present a Boolean matching approach to access the database while leveraging Boolean don't care conditions. Finally, we propose an efficient integration of window-based don't care computation and matching in cut-based logic rewriting.

In the experiment section, we show that mapping-based rewriting with DCs reduces up to

13.21% and 0.62%, on average, the size compared to the state-of-the-art MIG flow over the EPFL benchmarks. Notably, in this experiment, we compare one algorithm against a flow of three algorithms composed of the standard mapping-based rewriting and variants of Boolean resubstitution. We achieve even more reductions in size up to 14.32% and 4.31% average after integrating logic rewriting with DCs in the state-of-the-art flow. Additionally, we show that logic rewritying with DCs contributes to obtain the best-known results in MIG size for the EPFL benchmarks.

5.3.1 Don't Care Classes

Due to controllability or observability don't cares (DCs) in a logic network, often a Boolean function f can be changed into another one, f', without affecting the intended behavior of the circuit at the primary outputs. Such a function f' is called a *permissible function,* and the functional flexibility is described by its don't care set dc. A set containing all the permissible functions of f is referred to as the *maximum set of permissible functions* (MSPF) [151].

In this section, we present a method to represent a database of structures and compute permissible functions under Boolean don't cares. At this phase, permissible functions are computed under all possible Boolean don't care conditions to enable efficient Boolean matching. First, the database is classified into $\mathcal{NPN}$-equivalence classes. Then, it is processed to compute all the don't care sets that lead to a permissible function with better cost. Finally, we present our algorithm for Boolean matching, assuming don't cares are provided.

Database

The database is internally represented as a compact data structure that facilitates fast Boolean matching. The database is classified into $\mathcal{NPN}$-equivalence classes to limit the number of entries (e.g., 222 for 4-input functions). Each class functionality is expressed by a representative truth table, which is computed by finding the lexicographically smallest truth table in the class. A class may list several implementations (Boolean networks), each realizing the class representative function and described by its area cost and pin-to-pin delay.

Don't care classes

Given a database classified into $\mathcal{NPN}$-equivalence classes, we compute minimal don't care sets that support moving from an $\mathcal{NPN}$ class into a function in a different $\mathcal{NPN}$ class. The definition of minimal is given later in the text as Definition 5.3.4. Informally, this problem can be seen as the construction of a directed graph where nodes are $\mathcal{NPN}$ classes and edges are don't care sets. This idea is similar to the work of Mailhot [123], where vertices are $\mathcal{NPN}$ classes, but edges link functions that differ by one minterm. Thus, our approach differs in the size of the graph, the number of edges, and the Boolean matching technique. The graph creation is achieved by enumerating and storing don't care sets for each class.

Don't care sets are represented as truth tables. For a function f, an entry b_i in its don't care set dc is '1' if the bit in position i of f can be flipped. This information introduces flexibility in the functionality potentially leading to a better implementation (Boolean simplification).

Example 5.3.1. *Let c be a* 2*-input cut in an* and-inverter graph *(AIG), composed two-input ANDs and inverters, with function* $f = 1001$ *and don't care set* $dc = 0001$*, which is extracted from conditions external to the cut. The cut represents an XNOR function that needs* 3 *AND nodes to be implemented. The don't care set introduces flexibility to flip bit* b_0 *of* f *to obtain* $f' = 1000$*, which is an AND function that needs only one AIG node, improving the AIG size.* ▲

We define a *don't care class* that belongs to an $\mathcal{NPN}$ class as a set of don't cares that supports Boolean transformations into better permissible implementations. Generally, for a function f on k variables, there exist 2^{2^k} possible don't care sets. Moreover, for each don't care set dc, there are 2^p possible permissible functions, where p is the number of *minterms* in dc, i.e., the number of bits at 1 in dc. Therefore, filtering mechanisms are necessary to enable the computation and limit the search space during Boolean matching. To enable Boolean don't cares, we use two assumptions that limit the number of matching possibilities, stored don't care sets, and permissible functions.

Assumption 5.3.2. *To evaluate the benefit offered by a don't care set, we use the best implementation area in the maximum set of permissible functions (MSPF).*

If a database contains the size optimum implementations, the best area coincides with the area optimum. This assumption prioritizes the area over other metrics for multiple reasons. First, logic rewriting is typically area-oriented. Second, the area is usually independent of the context of the rewriting, whereas propagation delay depends on the arrival time, and it cannot be evaluated offline. Third, often better area implementation offer also better delay (especially in the context of technology-independent optimization). This assumption is used as a filter. In other words, we only store don't care sets for which there exist a function in the MSPF that offers a better implementation in terms of area cost.

Assumption 5.3.3. *For each don't care set, we select one permissible function that minimizes the area.*

Given a don't care set for a function f, the size of the MSPF can be pretty large, offering many implementation options. Evaluating all of them during logic rewriting may significantly increase the run time without offering a considerable advantage. Hence, for each don't care set, our method stores a single permissible function that minimizes the area cost.

Before proceeding with the technical explanation, we remind the reader of a definition presented in Section 2.2.1. A truth table t_1 is said to *imply*, or *cover*, another truth table t_2 if each bit of t_1 is true also in t_2. This relationship is denoted as $t_1 \leq t_2$. Similarly, t_2 is said to be *implied* by t_1, denoted as $t_2 \geq t_1$. For instance, $1000 \leq 1001$.

To further filter the number of saved don't care sets, we employ the definition of *dominance*.

Definition 5.3.4. *For a function f, a don't care set t_1 is said to dominate a don't care set t_2 if $t_1 \leq t_2$ and the best area cost in the MSPF of f for t_1 is not worse than the one for t_2. The set t_2 is said to be dominated by t_1.*

Informally, we refer to a non-dominated don't care set as *minimal.* Non-minimal sets are redundant to store since they are implied and don't offer better implementations.

Algorithm 5.2 shows the procedure to compute don't care (DC) classes and permissible functions. The algorithm takes a database classified into $\mathcal{NPN}$-equivalence classes and the maximum number of input variables in the database, which is typically 4, as inputs. The procedure starts by iterating through each $\mathcal{NPN}$ class, assigning to f_i the class representative function. At line 2, the DC class for f_i is set to empty. Then, from line 3 to 12, the procedure iterates to all the other classes f_j with a better area cost than f_i. At this step, all the possible don't care sets that link f_i and f_j are computed. To achieve that, all the negations and permutations configurations of f_j are enumerated to capture all the functions g in the $\mathcal{NPN}$ class of f_j. Along with g, the enumeration generates the input permutation vector *perm* and input/output negation vector *neg* that store the information to transform g into f_j. The don't care set dc, which links f_i and g, is computed using the exclusive disjunction operator at line 8. Then, dc is checked for dominance following Definition 5.3.4. If the don't care set is currently minimal, previously computed dominated sets are removed, and the new one is inserted in dc_class. The set is inserted together with the input permutations and input/output negations to apply to f_i under don't care set dc to obtain f_j. Finally, dc_class is sorted by implementation area in ascending order. If a database is *partial,* i.e., it does not contain implementations for each $\mathcal{NPN}$ class (not *complete*), the best area of missing classes is assumed to be infinite.

Example 5.3.5. *Let us consider the $\mathcal{NPN}$-4 class $f_i = \text{0x033c}$, with the bit string represented in hexadecimal format, having best area of 4. First, let us consider the class $f_j = \text{0x0000} = \bot$[3] that represents constants, of cost 0. The two possible don't care sets that link the two classes are $dc_1 = \text{0x033c}$ and $dc_2 = \text{0xfcc3}$, since $f_i \wedge \neg dc_1 = \bot$ and $f_i \vee dc_2 = \top$. The two DC sets are minimal and are found by taking the Boolean difference between f_i and f_j for dc_1, and f_i and $\bar{f}_j$ for dc_2.* ▲

Example 5.3.6. *Let us consider the previous class $f_i = \text{0x033c}$ and a class $f_j = \text{0x003c} = \bar{x}_3 \wedge ((x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2))$ of with cost of 3. Along with the trivial DC set $dc_3 = \text{0x0300}$, there exist another one, $dc_4 = \text{0x000c}$, with permutations $\mathcal{P}_I : (x_0x_1x_2x_3 \rightarrow x_0x_3x_1x_2)$ and no negations. If we flip bits using the don't care conditions we obtain $g = f_i \wedge \neg dc_4 = \text{0x0330} = \bar{x}_1 \wedge ((x_2 \wedge \bar{x}_3) \vee (\bar{x}_2 \wedge x_3))$, which is a permutation $\mathcal{P}_I^{-1}$ of class f_j*[4]. ▲

[3] Symbol $\bot$ represents constant zero while symbol $\top$ represents constant one.

[4] Permutation $\mathcal{P}_I^{-1}$ represents the inverse (or transpose) of $\mathcal{P}_I$, if represented as a permutation matrix. In the example $\mathcal{P}_I^{-1} : (x_0x_3x_1x_2 \rightarrow x_0x_1x_2x_3)$.

Algorithm 5.2: Extracting don't care classes

```
Input: Database $data$, Number of variables $k$
Output: Don't care classes $dc\_class$
foreach function $f_i$ in $\mathcal{NPN}(k)$ do
    $dc\_class(f_i) \leftarrow \emptyset$
    foreach function $f_j$ in $\mathcal{NPN}(k)$ do
        $s_j \leftarrow best\_area(f_j, data)$
        if $s_j \geq best\_area(f_i, data)$ then
            break
        foreach $\{g, perm, neg\}$ in $npn\_enumeration(f_j)$ do
            $dc \leftarrow f_i \oplus g$
            if $is\_dominated(dc, dc\_class(f_i), s_j)$ then
                continue
            remove_dominated($dc$, $dc\_class(f_i)$, $s_j$)
            $dc\_class(f_i)$.add($dc$, $f_j$, $perm$, $neg$)
    sort_dc_class($dc\_class(f_i)$)
return $dc\_class$
```

Regarding the scalability of Algorithm 5.2, the computation of don't care classes takes less than half a second for databases up to 4-inputs and very low memory. For larger databases, this method would experience limitations due to the double exponential increase in the number of Boolean functions. Hence, it may necessitate restricting the computation to only *practical classes* for functions of more than 4 variables. Practical classes are a subset of $\mathcal{NPN}$ classes that are often observed in common designs and tend to be much less in number compared to the number of $\mathcal{NPN}$ classes. For instance, common practical functions are the fully- and partially-decomposable functions. In [68], the authors found only 286 unique $\mathcal{NPN}$ classes for 6-input functions when mining the EPFL benchmarks [3].

5.3.2 Matching with Don't Cares

Given a Boolean function and its don't care set, as truth tables, Boolean matching returns a list of implementations in the MSPF class with minimal cost computed by Algorithm 5.2.

The Boolean matching procedure is shown in Algorithm 5.3. Compared to standard Boolean matching over $\mathcal{NPN}$ classes, our algorithm adds the steps from line 2 to 8. The algorithm takes a function f, its don't care set dc, the database, and the don't care classes as inputs. First, function f is canonicalized by computing the lexicographically smallest truth table in its $\mathcal{NPN}$ class using fast enumeration [82]. The class representative f_c is returned along with its permutation and negation vectors. Then, the permutations are applied to the don't care set such that its bits respect the new permutation in f_c (line 2). Input and output negations are not applied since they don't affect the don't cares. Then, the don't care class of f_c is accessed to retrieve a better implementation. Each entry is accessed in order, from the smallest area implementations to the largest. Each entry is composed of its don't care set t, its

Algorithm 5.3: Boolean matching with don't cares

```
   Input: Function $f$, Don't care set $dc$, Database $data$, Don't care classes $dc\_class$
   Output: Matches $M$, Permutations $perm$, Negations $neg$
1  $\{f_c, perm, neg\} \leftarrow$ npn_canonicalize($f$)
2  $dc \leftarrow$ apply_permutations($dc$, perm)
3  foreach $\{t, f_i, p, n\}$ in $dc\_class(f_c)$ do
4      if $t \leq dc$ then
5          $perm \leftarrow$ apply_permutations($perm$, $p$)
6          $neg \leftarrow$ apply_permutations($neg$, $p$)
7          $neg \leftarrow neg \oplus n$
8          return $\{data(f_i), perm, neg\}$
9  return $\{data(f_c), perm, neg\}$
```

$\mathcal{NPN}$ class representative f_i, the permutation vector to apply p, and the negation vector to apply n. The entry is a permissible function if $t \leq dc$, i.e., the don't care set t implies dc. As soon as this is true, the algorithm returns the implementations for the best permissible function. Before returning, the previously computed permutation and negation vectors are adjusted to match the new $\mathcal{NPN}$ class and its representative (from line 5 to 7). This is required to match the functionality of the new class, as shown in Example 5.3.6. If no entry matching the given don't care set is found, the algorithm returns the implementations from f_c.

5.3.3 Logic Rewriting with Don't Cares

This section describes the integration of Boolean matching with don't cares into classical logic rewriting algorithms. The classification of the database and the computation of the don't care classes presented in Section 5.3.1 are independent of logic rewriting, are computed offline, and are not addressed in this section. Algorithm 5.4 reflects the implementation of DAG-aware rewriting [137] with an extension to support Boolean don't cares. Similarly, this method can be integrated into alternative rewriting or mapping techniques.

Algorithm 5.4 tries to replace small sections of the network defined by cuts with a better implementation. The algorithm processes the nodes in topological order and searches for the best replacements that locally improve the area. Compared to the standard rewriting, Algorithm 5.4 adds the steps between line 3 and 12. For each gate g, the k-feasible cuts rooted in n are computed using a cut enumeration procedure [47]. Then, a reconvergence-driven window of l inputs, having $l > k$, is extracted around gate g. The window can be single-output (a cut) in the case when only controllability don't cares are used, or multiple-output, expanded over the transitive fan-out of g when also observability don't cares are used. Then, complete simulation is performed over the window to extract complete truth tables for each covered node. The truth tables are on l variables and computed with respect to the inputs of the window. Next, for each cut, the best matches are evaluated. First, for a cut c, its function f is extracted. Then, if the cut fits in the window, i.e., all its leaves are contained in the window, its

Algorithm 5.4: Logic rewriting with Boolean don't cares

```
Input: Network $N$, Database $data$, Don't care classes $dc\_class$, Cut size $k$, Cut size $l$
foreach gate $g \in N$ in topological order do
    $C \leftarrow$ compute_cuts($N$, $g$, $k$)
    $W \leftarrow$ reconvergence_driven_window($N$, $g$, $l$)
    $S \leftarrow$ simulate_window($W$)
    $R \leftarrow \Lambda$
    $best\_gain \leftarrow 0$
    foreach cut $c \in C$ do
        $f \leftarrow$ truth_table($c$)
        $dc \leftarrow \perp$
        if $c \subset W$ then
            $dc \leftarrow$ compute_dont_cares($c$, $W$, $S$)
        $\{M, p, n\} \leftarrow$ bool_matching($f$, $dc$, $data$, $dc\_class$)
        $gain \leftarrow$ evaluate_gain($N$, $g$, $c$, $M$, $p$, $n$)
        if $gain > best\_gain$ then
            $R \leftarrow$ candidate_replacement($N$, $g$, $c$, $M$, $p$, $n$)
            $best\_gain \leftarrow gain$
    if $best\_gain > 0$ then
        replace($N$, $g$, $R$)
```

don't care set is computed from the window. If it doesn't, don't cares are ignored for the cut. Alternatively, a window may be computed to guarantee containment at the cut at the cost of additional run time. However, experimental results have shown that many cuts tend to be included in the window. Next, Boolean matching is performed according to Algorithm 5.3, and candidate replacements are evaluated. Finally, the candidate with the best area gain is used as a replacement.

The most runtime-intensive process of logic rewriting with Boolean don't cares is the computation of DCs. Boolean DCs for a cut are extracted starting from a window of logic that includes it. First, the window is simulated over its input to collect complete simulation patterns. Given simulation patterns, controllability don't cares (CDCs) are computed by checking which combinations of patterns appear at the leaves of the cut. Non-appearing patterns are CDCs for the cut. This process is called *projection* of the don't cares and its complexity is exponential in the number of leaves of the window. Observability don't cares (ODCs) at a gate g are instead computed by checking for which patterns the Boolean difference between the function of the gate g and its inverse is observable. Let $S(g)$ be the simulation pattern in the window for gate g and let O be the set of output of the windows. First, the window is re-simulated fixing the simulation of gate g to $\neg S(g)$ and obtaining the simulation patterns S' at the outputs. The

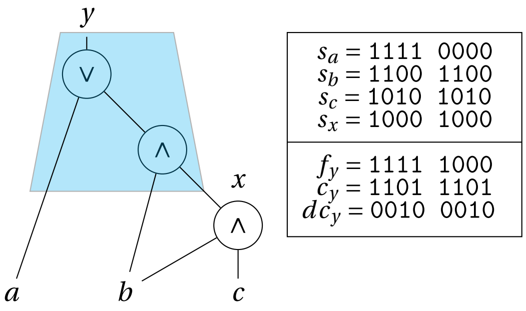


Figure 5.3: Example of projection of CDCs on a cut.

ODCs are then computed as follows:

$$ODC_g = \neg \bigvee_{o \in O} S(o) \oplus S'(o).$$

The ODCs need to be projected over the cut leaves as for CDCs. Despite being the run time bottleneck of logic rewriting, projections for multiple cuts can be computed in parallel, notably reducing the impact over run time.

Example 5.3.7. *Figure 5.3 shows an example of CDC computation and projection on an AIG. In this example, the window covers the entire circuit. First, the window is simulated, obtaining the patterns s_a, s_b, s_c, and s_x. Patterns s_a, s_b, and s_c are the input patterns of the window and are used to simulate all the input combinations (each bit b_i in every input pattern represents a combination). The section in blue represents a cut to optimize with function $f_y = a' \vee (b' \wedge x')$, computed considering a', b', and x' as inputs[5]. The care set c_y, of node y, i.e., the complement of the don't care set, is computed starting from the simulation patterns. Each combination of input patterns at the cut s_a, s_b, and s_x is used to set bits in c_y. For instance, taking b_0, the input pattern of the cut is* 000 *($s_a(0) = 0, s_b(0) = 0, s_x(0) = 0$). Hence, b_0 of c_y is set to one, as it is an appearing pattern. Taking b_6, the pattern is* 110 *($s_a(6) = 1, s_b(6) = 1, s_x(6) = 0$) setting b_6 of c_y to* 1*. At the end of this process, the care set has a "1" for occurring patterns. Finally, $dc_y = \neg c_y$ is computed to express the CDCs of the cut. Consequently, f_y can be simplified into* 11111010*, which corresponds to $a' \vee x'$. This transformation removes a node while preserving the correct functionality.* ▲

5.3.4 Experimental Results

In this section, we present experimental results on logic rewriting with Boolean don't cares. For our experiments, we use the EPFL combinational benchmark suite [3] containing several circuits provided as *and-inverter graphs* (AIGs).

The construction of the database, the generation of the don't care classes, and Boolean

[5] Variables a', b', and x' are "virtual" input variables of the cut. The link between a and a', as well as the one for other variables, is not visible by the cut.

matching with don't cares have been implemented in C++17 and used to extend the algorithms in the open-source logic synthesis framework *Mockturtle*[6]. The database of structures used in the experiments is available in the library and contains 4-input size-optimum implementations obtained using exact synthesis. Up to 10 structures are available for each $\mathcal{NPN}$ class. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified for functional equivalence.

Logic rewriting with Boolean don't cares

In this experiment, we test logic rewriting with don't cares to optimize *majority-inverter graphs* (MIGs) [5], which have many applications in standard-cells design flows [5], and majority-based emerging technologies [129, 168]. We compare our approach against the state-of-the-art flow published in [104], which is based on the most effective MIG methods known. The baseline flow carries the optimization by running the command *compress2rs* in ABC, the mapping-based logic rewriting algorithm presented in Section 5.2 (and [202]) three times, the MIG Boolean resubstitution in [168] until no more improvement, and the improved MIG resubstitution presented in the paper itself [104]. These results have been reproduced on our machine.

In our implementation, logic rewriting operates on a database of 4-input size-optimum MIG implementations, the same one used in [202]. The classification of the database and the computation of don't care classes took less than half a second on our machine. We extended the implementation of two logic rewriting algorithms to support don't cares. In particular, we improved the mapping-based logic rewriting algorithm in Section 5.2, referred to as *map*, and the DAG-aware rewriting algorithm in [137], referred to as *rw*. DAG-aware rewriting has been re-implemented following the versatile paradigm of Section 5.2. Both algorithms include the don't care computation as shown in Algorithm 5.4 for controllability don't cares. In the experiments, we don't use observability don't cares for two reasons: 1) ODCs are generally not compatible, i.e., not safe to use in parallel optimization (like *map* does) [52]. Hence, additional run time is required to compute compatible ODCs (CODCs); 2) experimental results using ODCs (and CODCs) in logic rewriting have not shown significant benefits in quality. In our implementation, don't care projections have not been parallelized.

Table 5.4 shows the experimental results. To test our approach, we implemented three flows with increasing optimization effort. All three flows are applied to initial results obtained by executing the optimization script *compress2rs* in ABC, like for the state-of-the-art flow. Our first flow, named "map with DCs", consists of 3 iterations of *map* with CDCs computed from 12-input cuts. Our second flow, named "map + rw with DCs", adds to flow one 3 iterations of *rw* with CDCs computed from 8-input cuts. Finally, flow three, named "MIG flow with DCs", adds Boolean resubstitution [104] to flow two.

Our first flow reduces the size by 0.62%, on average, and up to 13.21% compared to the

[6] Available at: https://github.com/lsils/mockturtle

Table 5.4: Comparison between state-of-the-art MIG results and multiple MIG flows using logic rewriting with don't cares.

Benchmark	**Flow in [104]**		***Map* with DCs**				***Map* + *rw* with DCs**			**Flow with DCs**		
	Size	Time (s)	Size	Red. (%)	T_{BM} (s)	Time (s)	Size	Red. (%)	Time (s)	Size	Red. (%)	Time (s)
adder	384	0.18	384	0.00%	0.02	0.16	384	0.00%	0.20	384	0.00%	0.22
bar	2588	0.82	2597	-0.35%	0.06	0.73	2445	5.53%	1.68	2433	5.99%	1.72
div	12532	4.54	12551	-0.15%	0.40	6.58	12498	0.27%	12.27	12462	0.56%	16.30
hyp	124177	58.73	115856	6.70%	3.41	54.10	115628	6.88%	91.12	115541	6.95%	118.51
log2	23109	36.22	22714	1.71%	0.49	12.59	22430	2.94%	24.69	22010	4.76%	45.64
max	2210	0.99	2202	0.36%	0.06	1.08	2191	0.86%	2.28	2190	0.90%	2.63
multiplier	18440	6.80	17474	5.24%	0.47	7.10	17155	6.97%	10.08	17112	7.20%	12.65
sin	3967	4.20	4005	-0.96%	0.13	2.50	3929	0.96%	5.69	3870	2.45%	8.55
sqrt	12423	10.38	12450	-0.22%	0.34	7.06	12388	0.28%	10.60	12357	0.53%	16.08
square	9498	2.72	8243	13.21%	0.23	3.30	8163	14.06%	4.59	8138	14.32%	5.33
arbiter	6719	4.34	6996	-4.12%	0.17	5.27	6869	-2.23%	7.70	6711	0.12%	9.81
cavlc	533	2.60	525	1.50%	0.01	0.09	517	3.00%	0.17	492	7.69%	1.72
ctrl	79	0.62	84	-6.33%	0.00	0.01	81	-2.53%	0.02	74	6.33%	0.30
dec	304	0.23	304	0.00%	0.00	0.03	304	0.00%	0.03	304	0.00%	0.06
i2c	932	0.37	898	3.65%	0.02	0.18	893	4.18%	0.37	871	6.55%	0.49
int2float	181	0.24	180	0.55%	0.00	0.03	178	1.66%	0.04	172	4.97%	0.11
mem_ctrl	34777	17.18	35218	-1.27%	0.82	14.59	34727	0.14%	33.91	32097	7.71%	43.75
priority	431	0.30	426	1.16%	0.01	0.15	420	2.55%	0.29	406	5.80%	0.35
router	151	0.17	155	-2.65%	0.00	0.04	154	-1.99%	0.08	147	2.65%	0.11
voter	4561	1.74	4819	-5.66%	0.12	1.95	4564	-0.07%	3.56	4555	0.13%	4.45
Average				0.62%				2.17%			4.31%	

state of the art. This is a major result considering that the comparison is between a single command and a flow. Our flow includes the column T_{BM}, which reports the total time taken by Boolean matching with don't cares. The matching time is a small fraction of the total time, which is mainly dominated by the computation and projection of CDCs. Our second flow further reduces the number of MIG nodes improving up to 14.06% and 2.17% on average the state of the art. Almost every result is already significantly better before employing Boolean resubstitution, which is the standard algorithm to leverage Boolean don't cares. Notably, logic rewriting is very effective at optimizing arithmetic benchmarks (the first 10 benchmarks). Finally, the third flow uses also Boolean resubstitution to obtain superior results for every benchmark reducing the size up to 14.32% and 4.31% on average.

Furthermore, we tested this approach on AIG optimization. While rewriting with DCs helps reduce the number of AIG nodes, the improvement is less significant compared to MIGs since AIG-resynthesis methods are much more mature. In particular, the area reduction compared to standard rewriting [137] is up to 6.8% and 0.42% on average over the EPFL benchmarks, previously optimized using the script *compress2rs* in ABC.

Best-known MIG results

We integrated the graph mapping and logic rewriting algorithms with Boolean don't cares in the design space exploration (DSE) engine in [106] to improve the best-known MIG results. We show that by iterating these algorithms and logic collapsing using a LUT mapper in a flow we can achieve a great improvement and the best-known results for MIG size. These results were published in [109]. We refer the reader to the papers [106, 109] for further details on the DSE engine.

Table 5.5: Latest best results for MIG size optimization.

Benchmark	**Results in Table 5.4**	**New Best Results**		
	Size	Size	Impr.	Depth
adder	384	384	**0.00%**	129
bar	2433	1906	21.7%	15
div	12462	12368	**0.75%**	2251
hyp	115541	115539	**0.00%**	9129
log2	22010	22008	**0.01%**	184
max	2190	1939	11.5%	172
multiplier	17112	17112	**0.00%**	137
sin	3870	3869	**0.03%**	124
sqrt	12357	12247	**0.89%**	2156
square	8138	8089	**0.60%**	126
arbiter	6711	792	88.20%	108
cavlc	492	374	23.98%	16
ctrl	74	60	18.91%	8
dec	304	304	**0.00%**	3
i2c	871	636	26.98%	16
int2float	172	115	33.14%	9
mem_ctrl	32097	6886	78.54%	26
priority	406	337	17.00%	23
router	147	97	34.01%	13
voter	4555	3894	14.51%	32
Total	242326	208956	13.8%	14677

Table 5.5 shows the experimental results of the the design space exploration engine compared to the flow in Table 5.4. The DSE achieves an average size reduction of 13.8%. On the one hand, for most arithmetic benchmarks the results in Table 5.4 are very close to the ones found by DSE or are exactly the best one known. This result highlights the power of DC-based rewriting in improving the quality of results in an orthogonal way compared to previous methods. On the other hand, control and random logic benchmarks need multiple optimization iterations and logic collapsing to converge to a good quality of results. For instance, we observe a drastic improvement on the benchmarks *arbiter* and *mem_ctrl*.

5.4 Factored Form Literals Optimization

It is well-known that the *factored form literal count* (FFLC) correlates strongly with the number of transistors required to implement a Boolean function [12]. Consequently, past research efforts in technology-independent synthesis focused on synthesizing small Boolean networks with a minimal FFLC [28, 100, 178]. Historically, FFLC minimization was performed on logic networks with nodes represented in SOP form [28]. To our knowledge, this approach is still in use in many EDA tools for standard-cell designs.

In the last few decades, substantial progress has been made by leveraging the simplicity of an *and-inverter-graph* (AIG) representation [97] for the technology-independent synthesis of Boolean networks [26]. AIG optimization methods can efficiently synthesize large AIGs while minimizing the number of AIG nodes and logic levels. However, this optimization does not inherently minimize the FFLC. Specifically, the relation between FFLC optimization and AIG-based optimization has not been yet studied. As a result, current AIG-based technology-independent synthesis may not work at its best for optimizing standard-cell-based designs. Moreover, a fully custom design methodology could significantly benefit from modern and scalable FFLC optimization approaches. This research is motivated by applications in transistor-level synthesis and the automated creation of custom optimized standard cells, which could drive significant improvements in efficiency and performance.

In this work, we investigate the relation between FFLC optimization and AIG-based optimization. Then, we propose several efficient AIG-based FFLC minimization methods that work without converting AIGs into logic networks, as required by traditional FFLC minimization techniques [28]. The portfolio FFLC optimization includes (i) an enhanced Boolean resubstitution [132], (ii) a modified version of AIG rewriting [137] and refactoring, and (iii) a dedicated FFLC minimization that performs AIG re-mapping using a versatile technology mapper [202]. In the experimental results, we show up to a 5.3% reduction in the literals count, up to a 7% reduction in the area after technology mapping, and no run time increase compared to a high-effort area-oriented AIG optimization. This demonstrates the ability of FFLC optimization to refine the structure of AIGs to be more suitable for technology mapping. Additionally, we discuss applications beyond traditional logic optimization, for transistor-level synthesis and auto-creation of standard cells.

5.4.1 Preliminaries

Logic representations

Logic representations are key for developing robust EDA tools. They enable compact data storage in memory and efficient implementation of optimization algorithms. One of the first standard representations of Boolean logic was the *Sum-Of-Products* (SOP) [28]. An SOP is a two-level representation consisting of the logic OR of *product terms*, which are logic ANDs of *literals* (variables or their complements). This representation was motivated by *programmable logic arrays* (PLAs) whose primitives are modeled directly using SOPs. Because of the simple structure of a two-level circuit, the optimization problems for SOPs are well understood, which led to the development of efficient heuristic and exact minimization methods. A powerful extension of SOPs into a multi-level representation are *factored forms* [28]. A factored form is defined recursively as follows. A literal is a factored form, and the logic OR or logic AND of two factored forms is a factored form. Informally, a factored form is an SOP whose inputs are other SOPs, etc.

Example 5.4.1. *Given function $f = ab + ac + ad + bcd$ in SOP form, we can derive a factored form by factoring with respect to variable a. This gives us $f_1 = a(b + c + d) + cbd$. Note that a* *factored form is not unique. For instance, by factoring with respect to variable b,* $f_2 = b(a + cd) + ac + ad$*. However, this latter form is less optimized as it contains more literals. Nevertheless, we can still factor it with respect to variable a, resulting in* $f_3 = b(a + cd) + a(c + d)$*, which is another factored form having the same number of literals of* f_1*.* ▲

Logic optimization

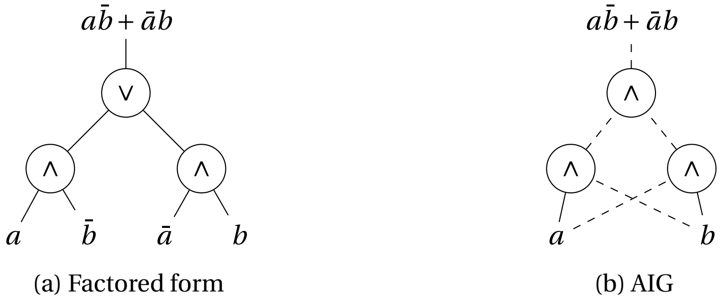
Logic optimization is a key step that enables the design of efficient circuits. Over the years, many techniques working on DAGs have been proposed. Choosing a few primitives to represent circuits as DAGs helps navigate through the logic and extract properties. State-of-the-art methods are primarily working on *And-Inverter Graphs* (AIGs). The tool ABC [30] is considered the state-of-the-art academic tool for logic optimization. ABC uses AIGs as the main logic representation. The most common and powerful optimization algorithms are *resubstitution, rewriting, refactoring,* and *balancing* [132, 137]. Most of the optimization scripts are composed of a combination of these algorithms:

- *Resubstitution*: Resubstitution [132], shortened to *resub*, (re)expresses the function of a node using other nodes, called *divisors*, that are already present in the network. The transformation is accepted if the new implementation of a node is better, according to a target metric (e.g., size), compared to the current implementation of the node in terms of its immediate fan-ins. This approach generalizes to *k-resubstitution*, which adds k new nodes and removes at least $k + 1$ nodes. The removed nodes are the ones present in the *maximum fan-out free cone* (MFFC) [132] of the node. The functionality of the new nodes is derived from a library of primitives used for resubstitution. In the AIG implementation, added gates are 2-input ANDs with optional inverters at the inputs/outputs.

- *Rewriting*: Rewriting [137] is a fast greedy algorithm that aims at minimizing the size of a logic network by iteratively replacing sub-graphs rooted in a node with smaller pre-computed structures while preserving the functionality at the root node. Typically, pre-computed structures cover all the 4-variable functions classified into the NPN equivalence classes for compactness [23].

- *Refactoring*: Refactoring is similar to rewriting. It iterates over large logic cones rooted in a node and tries to replace the logic structure of the cone with a factored form of the root function. The replacement is accepted if there is an improvement in the selected cost metric (usually the number of gates) [132, 137]. Unlike rewriting, it does not rely on a database implementation but instead uses methods that compute factored forms directly from SOP representations.

- *Balancing*: Balancing is a fast algorithm that reconstructs logic by balancing the structure using the associative property such that the logic depth is minimized.



(a) Factored form

(b) AIG

Figure 5.4: Translation of a factored form of a XOR2 (a) into an AIG (b). Dashed edges represent negations.

5.4.2 Factored Forms in AIGs

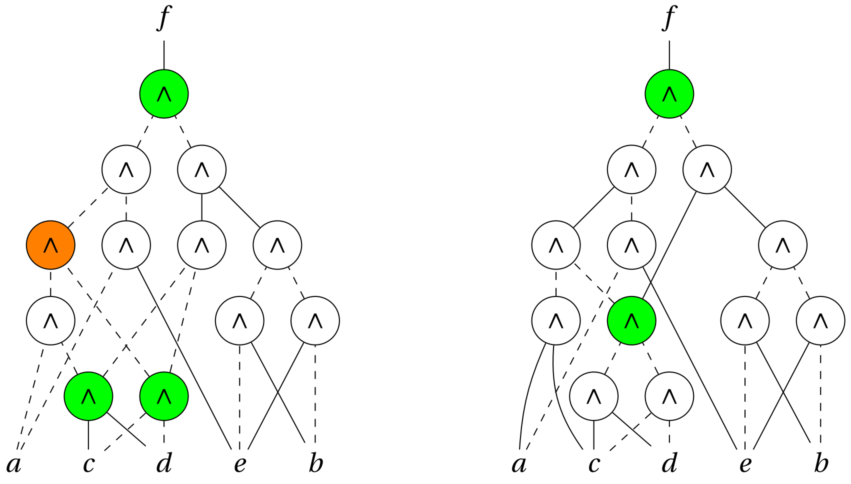
In this section, we describe the relationship between factored forms (FFs) and and-inverter graphs (AIGs). This allows us to introduce the notion of factored form literals of an AIG and propose algorithms to reduce the factored form literal count (FFLC). Unlike traditional logic synthesis [28, 178], our approach does not need to convert an AIG into a logic network. Hence, it offers better scalability for large designs.

One application of AIGs in synthesis is the representation of DAGs derived by Boolean decomposition or factoring. In particular, FFs can be represented as syntax trees where nodes are AND or OR operations, and leaves are literals (variables or their complements). Thus, FFs can be directly represented by an AIG by translating ANDs to ANDs, and ORs to ANDs using De Morgan's law $x \vee y = \overline{\bar{x} \wedge \bar{y}}$. An AIG representation of a FF is composed of primary inputs with multiple fan-outs, 2-input AND gates with a single fan-out (and possibly complemented inputs), and an output associated with a primary input or a 2-input AND. In a FF, the number of literals is given by its number of leaves. For instance, in Figure 5.4a the number of literals is 4. In the AIG representation of FFs, the number of literals is equal to the fan-out count of the inputs of the graph.

Example 5.4.2. *Figure 5.4 shows a representation of an XOR2 in FF and its translation into an AIG. In Figure 5.4a the number of literals is* 4 *since the FF has* 4 *leaves. From Figure 5.4b representing an AIG, the same result can be computed by summing the fan-out count of the primary inputs.* ▲

AIGs representing combinational logic are not FFs because AND nodes may have multiple fan-outs. Nevertheless, FFs can be used to cover an AIG. Deriving the FF cover can be done by a technology mapper.

In this work, we follow [39] and refer to nodes with a single fan-out as *tree nodes*, and to nodes with two or more fan-outs as *dag nodes*. Let us consider an arbitrary AIG. For each primary output, let us define cuts such that each node in the volume of cut is a tree node (except for the root) and the leaves are either dag nodes or primary inputs. Note that this



(a) Initial network: 15 lits (b) After *resub*: 14 lits

Figure 5.5: Optimization of an AIG for factored form literals. Figure (a) shows the initial network. Figure (b) shows the result with reduced literal count after *resubstitution* is applied to the orange node. Nodes in green are roots of factored forms.

definition is different from the notion of the MFFC of a node because the MFFC may contain dag nodes. By definition, each such cut covers a FF. Using this definition of a cut, we can cover the whole AIG by recursively creating new cuts from the leaves of the existing ones. Thus, FFs can cover an AIG and their number depends on the number of cuts in the cover. Hence, we can define the FFLC of an AIG as the sum of the literals of each FF contained in an AIG.

Without employing the notion of the cover, the FFLC of an AIG can be computed using a simple formula:

$$FFLC = O + 2 \times G - M, \tag{5.1}$$

where O is the number of primary outputs, G is the number of 2-input nodes, and M is the number of 2-input tree nodes. Alternatively, the FFLC of an AIG is the sum of the fan-out counts of the primary inputs and dag nodes.

Example 5.4.3. *Figure 5.5a depicts an AIG representation of the* 5*-input Boolean function of a partial product of a radix-*4 *Booth multiplier [156]. The AIG can be covered using* 3 *FFs rooted in the green nodes. The one rooted in f is connected to a, e, b, and the other two green nodes. The remaining two FFs rooted in the green nodes at the bottom are connected to c and d. The total number of FF literals is* 15 *and it is given by Formula 5.1 (*$1 + 2 \times 12 - 10 = 15$*) or by adding the fan-out counts of the primary inputs and the two dag nodes in green.* ▲

From the definition, it follows that FF literals in an AIG can be used as an alternative cost function to carry out the optimization of combinational logic. The simple definition of literal count makes it very efficient to compute.

5.4.3 Logic Optimization for Literal Count

In this section, we propose optimization algorithms aimed at reducing the factored form literal count in AIGs. We re-formulate Boolean resubstitution, Boolean rewriting, refactoring, and re-mapping to perform the literal count minimization. We suppose that each node n in an AIG has a reference counter showing the number of its fan-outs [132]. Reference counting is used for counting nodes in an MFFC and for efficient addition/removal operations for individual nodes and their MFFCs. We can also use the reference counters of the nodes to classify them into tree nodes and dag nodes and compute the FFLC.

Using Formula 5.1, we establish how local transformations affect the FFLC. A local change in the FFLC can be computed using the following formula:

$$\Delta FFLC = 2 \times \Delta G - \Delta M, \tag{5.2}$$

such that the dependency on O disappears. Moreover, peephole optimization algorithms have only to track the local change in the number of 2-input nodes G and tree nodes M to evaluate the reduction in FFLC, making the use of FFLC in optimization very efficient. Compared to traditional AIG-based optimization, the FFLC cost functions additionally presents the ΔM term.

Generally, if a 2-input dag node is removed from the graph, two literals are saved. If a tree node is removed from the graph, one literal is saved. However, every time a restructuring step increases the fan-out count of a tree node, the number of tree nodes decreases leading to an FFLC increase by one. Consequently, some transformations may decrease AIG size but increase the FFLC.

Example 5.4.4. *Figure 5.6 shows a case in which the FFLC increases due to a node substitution. In the example, the tree node p is substituted with a new node q. Removing p also removes any node in the MFFC such as t. Consequently, two tree nodes are removed decreasing G and M by two, and no dag node is transformed into a tree node. Node q is added starting from 2 tree nodes and substituted into p. Therefore, one new tree node (q) is added, increasing G and M by one. After adding a new node, two nodes increase their fan-out counts and become dag nodes. Hence, even though the AIG size is reduced, the total FFLC increases by one. Other transformations may increase or keep the AIG size constant but decrease the FFLC, as shown in Figure 5.5. This is not exploited by the state-of-the-art methods.* ▲

Resubstitution for factored form literals

In standard resubstitution, the improvement is measured by the difference between the count of the nodes removed and added. The nodes that can be removed are the ones included in the MFFC. Hence, the improvement is measured as |MFFC| − k where k is the number of added nodes. Similarly, in resubstitution for literals minimization, the literal saving is given by the difference between the literal count removed and added. The change in the number of literals

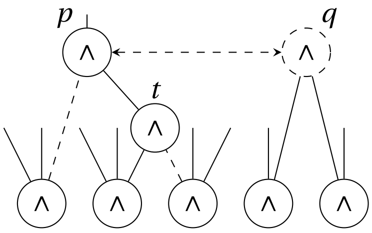


Figure 5.6: Substitution of p using a new node q that increases the number of factored form literals by one.

can be computed locally using node reference counters. The reference counters track the number of nodes removed/added (MFFC) and the tree nodes created/removed during the manipulation. Finally, the change is evaluated using Formula 5.1. The algorithm employed is a recursive dereferencing (referencing) that decreases (increases) the reference counter of a node and recurs over the fan-ins if the reference count is 0 (1). In particular, recursive dereferencing (referencing) is used to measure the MFFC. Literal savings are measured by recursively dereferencing the node to substitute (root node) and counting the nodes removed and the change in the number of tree nodes. Similarly, the creation of new nodes is measured using recursive referencing.

This approach also supports filtering rules for candidates to speed up resubstitution. To simplify the filtering rule, we do not account for the change in M for the root node when we compute the savings, because any new node created will inherit the fan-out of the root node leading to a zero change in M for the root. We can then use the support of the resubstitution as a filtering rule. For instance, 1-resub adds one node to the network increasing G by one adding 2 literals. For 2-resub, a minimum of 3 literals are added (one tree node), and so on.

Example 5.4.5. *Figure 5.5 depicts the FF literal optimization based on resubstitution on a Booth partial product. The orange node in Figure 5.5a is the target node for resubstitution. Figure 5.5b shows the result where the total literal count and the number of FFs needed to cover the AIG are reduced by one. Note that the AND number of nodes remains the same. The obtained structure of the graph is more suitable for technology mapping leading to a* 17% *area reduction after mapping the two implementations to a* 7nm *technology*[7] *[44].* ▲

Rewriting and refactoring for factored form literals

Rewriting and refactoring are enhanced similarly to resubstitution. Standard rewriting enumerates the 4-input cuts at a root node to match and evaluate the replacements. Refactoring uses MFFCs or reconvergence-driven cuts [132]. The improvement of a replacement is measured by counting the number of nodes in the cut that can be removed, i.e. the MFFC contained in the cut, minus the number of nodes added when the structure is inserted. Rewriting and

[7] The first design in (a) has been obtained after synthesis in ABC using the script *compress2rs*. We used *amap* in ABC for technology mapping.

refactoring for literals minimization evaluates the reduction equivalently for literals. Savings in the number of nodes and literals are calculated using the same method since they are both based on recursive dereferencing and referencing. This enables the use of both cost functions with one as a primary cost criterion and the other one as a tie-breaker. Our implementation of rewriting for FF literals optimizes for literals primarily and uses node savings as a tie-breaker. This is motivated by compactness since an AIG with fewer nodes is easier to manipulate.

Mapping for factored form literals

We implemented a global re-mapping method for AIGs targeting the minimization of FF literals. The method is similar to cost-based mappers applied to graphs [201, 202] and presented in Section 5.2. It consists of cost-driven mapping, followed by Boolean decomposition of each cut in the cover into an AIG, which can be seen as re-mapping. The algorithm works by computing cuts for each node using the fast cut enumeration procedure [155] and assigning to each cut a cost based on the FF representation. The FF is computed using the irredundant SOP (ISOP) extracted from the Boolean function of the cut. The SOP is then factored using algebraic or Boolean factoring [29]. Next, the technology mapper selects a cover to minimize the number of FF literals in the Boolean functions of the cuts used to cover the AIG.

5.4.4 Experimental Results

In this section, we evaluate the factored form optimization methods for technology-independent logic synthesis by showing the literal reductions and the results after technology mapping. We propose a resynthesis script called *compress2ff* for factored form optimization. This script has the same commands as *compress2rs* in ABC [132], but each command is modified to minimize FFLC rather than the node count. The two scripts have roughly the same runtime because the FF literal counting has negligible runtime overhead.

We set up our experiments for the manipulation of optimized designs for size and literal reduction before technology mapping. When performing technology-independent synthesis, we compare the new script to *compress2rs*, which is the default script in ABC for high-effort AIG size minimization [132]. Our baseline consists of two runs of *compress2rs* to obtain the initial compact representation of the AIG. Then, we create two flows: one running *compress2rs* two times, and the other running *compress2ff* two times. Finally, for technology mapping, we use `&nf -R 1000` in ABC for area-oriented mapping. We use ASAP7 [44] as the target standard cell library.

Table 5.6 shows the experimental results for the designs from the IWLS'05 benchmark suite [85]. Our methods reduce the AIG size, factored form literal count (FFLC), and area by 1.6%, 2.3%, and 2.8%, respectively, compared to the baseline. Instead, the flow that runs *compress2rs* has a limited improvement of only 0.6% in the AIG size, FFLC, and area compared to the baseline. In this experiment, we used a strong baseline to evaluate our methods. Even so,

Table 5.6: Experimental results for factored form literals optimization and technology mapping

Benchmark	**Baseline: 2×*compress2rs***					**ABC: 2×*compress2rs***					**Factored form opt: 2×*compress2ff***				
	Size	FFLC	Depth	Area	Delay	Size	FFLC	Depth	Area	Delay	Size	FFLC	Depth	Area	Delay
ac97_ctrl	10203	13039	10	7012.21	100.42	10145	12971	10	6976.53	99.33	10175	12979	10	6831.41	109.19
aes_core	19493	24738	23	14012.98	245.4	19167	24511	24	13864.05	244.74	19210	23972	23	13702.23	233.87
des_area	4274	5177	32	2835.45	295.91	4263	5162	32	2825.57	295.63	4255	5093	32	2856.12	289.84
des_perf	67904	99730	28	60836.15	292.37	67141	99000	28	60431.14	287.44	67145	95820	30	59012.16	294.43
DMA	21974	26614	26	14254.06	240.5	21954	26589	26	14242.05	253.43	21358	25706	26	13533.02	247.11
DSP	37559	47734	100	26216.06	958.63	37068	47187	103	25927.76	998.47	36849	46561	98	25573.38	866.02
ethernet	55708	69226	34	35478.61	427.08	55659	69160	34	35482.25	441.46	55600	69011	34	35382.93	435.39
i2c	858	1136	19	648.46	188.32	849	1124	19	642.85	188.32	832	1087	19	604.65	188.32
mem_ctrl	7983	10410	44	5717.15	417.43	7881	10303	46	5675.89	458.48	7815	10150	45	5624.12	385.83
pci_bridge32	16092	20628	46	11262.44	505.24	16077	20616	46	11252.62	505.24	16014	20477	47	11210.79	556.1
RISC	60048	75005	100	40591.26	1055.71	59723	74631	105	40332.13	1034.93	59012	73260	101	39634.05	1126.49
sasc	546	729	9	430.9	101.77	546	729	9	430.9	101.77	542	722	9	426.24	108.77
simple_spi	732	961	19	548.09	154.85	731	960	20	548.05	155.43	728	950	19	544.11	187.98
spi	3112	3797	32	2096.5	303.82	3070	3736	32	2062.54	307.17	3055	3692	31	2035.31	294.66
ss_pcm	389	497	9	301.26	70.78	389	497	9	301.26	70.78	389	497	9	298.22	70.78
systemcaes	9582	11542	38	7002.48	426.35	9548	11444	40	7007.27	453.54	9307	11157	38	6512.72	423.49
systemcdes	2276	3252	28	1872.14	333.19	2256	3231	28	1851.12	290.58	2164	3061	27	1727.16	309.89
tv80	6856	8603	48	4752.52	417.16	6768	8507	53	4664.8	456.63	6506	8158	49	4449.26	411.61
usb_funct	12582	16145	36	8920.83	329.01	12509	16059	41	8854.99	374.44	12482	15986	41	8801.6	407.19
usb_phy	347	524	10	307.72	104.98	347	524	10	307.72	104.98	346	512	10	297.72	104.98
vga_lcd	88633	112806	35	59943.02	353.96	88628	112802	35	59933.86	346.89	88616	112747	35	59755.3	339.89
wb_conmax	37146	43041	18	20746.44	195.03	36640	42338	18	20442.97	213.27	36258	41476	20	20144.39	193.95
Geomean	7508.1	9662.5	27.3	5382.8	273.8	7460.1	9605.6	27.9	5351.4	278.8	7387.8	9435.6	27.7	5232.7	278.3
Ratio	1.000	1.000	1.000	1.000	1.000	0.994	0.994	1.023	0.994	1.018	**0.984**	**0.977**	1.013	**0.972**	1.016

our approach enables further improvement showing the importance of FFLC optimization. For a fair comparison, our script mirrors *compress2rs* without exploring other FFLC optimization opportunities.

Generally, some designs respond to the FFLC optimization better than others. The optimization can lead to a significant improvement in the literal count, AIG size, and area after mapping for some benchmark, such as *DMA*, *i2c*, *systemcaes*, *systemcdes*, and *tv80*. For instance, our approach reduces the area of *systemcaes* by 7%. For other benchmark, our flow does not lead to significantly better results, compared to the standard flow, such as in *des_area*, *ethernet*, and *vga_lcd*. We noticed that our approach is more effective for control and random logic. On the other hand, arithmetic circuits are less impacted by the proposed optimization due to structural regularity. We expect better results with richer standard cell libraries, which can map large factored forms better.

5.4.5 Applications

Logic Optimization

In the previous section, we presented novel optimization algorithms to reduce the FF literal count in combinational logic, aiming at improving the area after technology mapping into standard cells. The positive impact of the proposed FF-based network optimization on the CMOS implementation offers new opportunities to restructure combinational logic represented as an AIG while preparing it for technology mapping. While size is currently the main measure of the graph complexity for the area, we found that the literal count is a powerful metric to guide fine-grain optimization leading to better quality after mapping. It is expected that deploying the aforementioned methods as part of an industrial synthesis flow would

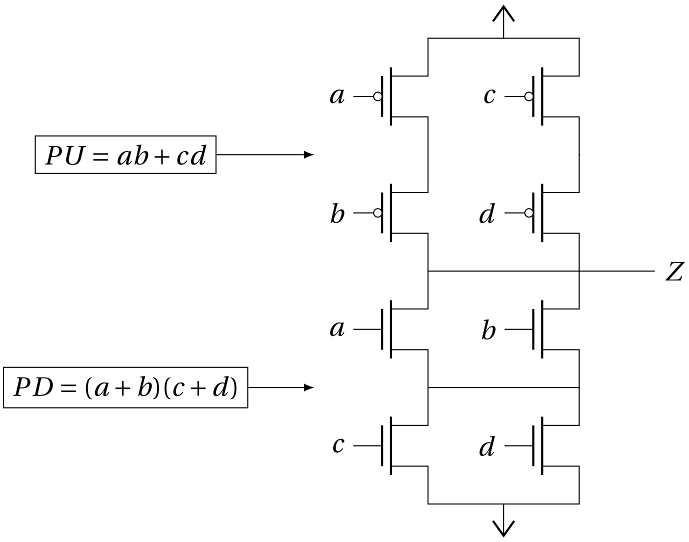


Figure 5.7: CMOS network for function $Z = \overline{(a+b)(c+d)}$ and the respective pullup (PU) and pulldown (PD) networks.

improve power, performance, and area.

Transistor-level synthesis

The literal count in FFs is a well-known proxy for transistor count in CMOS transistor networks. Transistor count is a fundamental measure that strongly correlates with area. Even if transistor count alone does not capture other important factors affecting area and power, such as transistor ordering, placement, and routing, it is one of the best estimator for area. In particular, FFs describe the serial-parallel connection of transistors. A serial connection is described using an AND gate. A parallel connection is described using an OR gate. This relation allows us to generate CMOS transistor networks from factored forms. Since the pulldown and pullup networks in CMOS are complementary, two FFs are needed, one being the dual of the other.

Example 5.4.6. *Figure 5.7 illustrates the mapping of function $Z = \overline{(a+b)(c+d)}$, in factored form, into a transistor network consisting of the pulldown network $PD = (a+b)(c+d)$ and its dual pullup network $PU = ab + cd$.* ▲

Since an AIG can contain many FFs, it naturally describes the connection of transistors in a multi-stage network. Using this relation, we can extract a transistor-level network after minimizing the inverters and mapping each FF into CMOS using the natural translation of factored forms, or using other methods [122, 162]. This property opens up to transistor-level synthesis offering flexibility in functionality, not restricted by standard cell libraries, and compact layout from automated transistor-level placement and routing approaches[103, 111]. In particular, the methods discussed in this work enable efficient transistor-level optimization and synthesis for large designs while working directly on an AIG representation thanks to the correlation between the number of literals and transistors. For instance, the proposed methods find the best transistor networks implementing multiple-input XORs, by constructing a multi-stage network of 2-input XORs, and many other cells.

Example 5.4.7. *For the Booth partial product function our approach optimizes and translates the function into a transistor-level network with* 34 *transistor after employing our patented factored form-based mapper. The transistor-level network generated is not included in the ASAP7 standard cell library [44]. It is a custom implementation at the transistor level assuming a pullup/pulldown network structure, as shown in Figure 5.7, and transistor stacking limits. Instead, if we map the function directly to the ASAP7 standard-cell library [44] while minimizing area, the resulting netlist consists of 40 transistors. This means that our approach generates a* 15% *reduction in transistor count for the Booth partial product implementation, which generally translates into a similar amount of area reduction after transistor placement and routing.* ▲

Our preliminary tests on transistor-level synthesis show that some useful transistor-level networks are not included in the ASAP7 standard cell library. The layout creation of those transistor-level networks and integration in an industrial flow is beyond the scope of this work, but these problems have been addressed in two patents, see [203, 213].

5.5 Summary

This chapter focused on modern technology-independent logic synthesis. In the first part, motivated by the need for advanced logic synthesis tools that leverage multiple graph representations, we proposed a technology mapping approach to logic optimization. We presented a versatile method for performing graph mapping from one representation to another while optimizing for circuit size or depth. When the target representation matches the starting one, it performs global logic restructuring. Graph mapping uses a database of structures of the target representation, extracted using SAT-based exact synthesis, as a library for mapping. We tested this approach for logic optimization on multiple logic representations. In MIG optimization, we demonstrated an average reduction in the number of majority gates by 32.11%, an additional 10% improvement compared to the best state-of-the-art rewriting methods. Our method also achieved significant results in XAG optimization, with an average gate reduction of 27.58%. Furthermore, we compared our method against the state-of-the-art rewriting method for XMGs, achieving a 27.45% reduction in the size/depth product. In the second part, we addressed don't care-based optimization on non-conventional graph representations for which high-quality resynthesis heuristics are not available, such as MIGs. We presented how to perform scalable graph mapping and logic rewriting while leveraging don't care conditions. We proposed methods for fast Boolean matching with don't cares and don't care extraction in cut-based algorithms. In MIG synthesis, we demonstrated that this method reduces the number of majority gates by up to 14.32%, with an average reduction of 4.31% compared to the state-of-the-art MIG flow. Additionally, we presented the best-known results for MIG synthesis on the EPFL benchmark suite, showing that logic rewriting with don't cares is responsible for most of the best results on arithmetic circuits. In the third part of this chapter, we revisited optimization based on factored form literal count (FFLC) with applications is design flows for standard cells and transistor-level synthesis. We studied the connection between AIG optimization and FFLC optimization, highlighting differences and analogies. Then, we formalized the FFLC optimization problem over the AIG and proposed several AIG optimization algorithms that minimize the FFLC count instead of the number of AIG nodes. We introduced the first approach to address FFLC optimization at the global logic-network level. We demonstrated that a flow combining conventional AIG optimization with AIG-based FFLC optimization improves the area of a design flow for standard cells by 2.8% on average after technology mapping. Additionally, we discussed applications in transistor-level synthesis and automatic standard-cell creation.

6 Specializing Synthesis for Superconducting Technologies

The previous chapters of this thesis were dedicated to technology mapping algorithms for established technologies and to technology-independent logic synthesis. This chapter focuses on synthesis for advanced emerging technologies based on superconductivity. *Superconducting electronics* (SCE) stands out as one of the most promising post-CMOS technologies, offering high-speed computation and power-efficient solutions. Despite being based on switching logic, numerous differences between SCE and CMOS necessitate specialized tools for synthesis and technology mapping. Specifically, two SCE constraints complicate the design flow: *path balancing* and *fan-out branching*. To satisfy the path-balancing and fan-out-branching requirements, technology mapping for SCE needs to insert delay registers and *splitters*. However, the potentially large number of these additional elements introduces further optimization challenges in SCE. This chapter proposes logic synthesis and technology mapping algorithms tailored for SCE, focusing on the two most mature logic families, namely the *adiabatic quantum-flux parametron* (AQFP) and the *single-flux quantum* (SFQ). Specifically, this chapter presents: (i) depth-optimal technology mapping algorithms for AQFP circuits; (ii) a post-mapping optimization algorithm for AQFP circuits based on minimum-register retiming; (iii) a logic synthesis and technology mapping framework for SFQ circuits based on the *xor-and graph* (XAG) representation. The content of this chapter is largely based on the publications in [110, 194, 195].

The remainder of this chapter is organized as follows. First, Section 6.1 presents the motivations of this chapter and Section 6.2 introduces the relevant background on SCE. Then, Section 6.3 presents two depth-optimal technology mapping algorithms for AQFP circuits that satisfy the path-balancing and fan-out-branching constraints. Additionally, it proposes a post-mapping optimization algorithm to recover area after technology mapping. The experimental results show that our approach reduces the number of delaying registers (buffers) and splitters up to 14% compared to the state of the art while guaranteeing optimal depth. Additionally, results demonstrate the scalability of these methods on circuits that are 10 to 100 times larger than the designs that any other related work could handle. Next, Section 6.4 presents a logic synthesis and technology mapping framework for SFQ circuits. It proposes a synthesis flow consisting of multiple delay-driven algorithms working with the xor-and graph (XAG), which efficiently abstract the SFQ logic primitives. The experimental results show an average reduction in the area and delay of 43% and 34%, respectively, compared to state of the art. Finally, Section 6.5 concludes and summarizes this chapter, highlighting the key findings and contributions.

6.1 Motivation

Recent advances in semiconductor electronics are pushing CMOS technology close to its physical limits. CMOS technology is experiencing higher fabrication costs and challenges in further downscaling transistor dimensions, with limited improvements in energy consumption and speedup. Additionally, with the increasing need for data management, storage, high-performance computing, and cloud computing, data centers surpassed 1% of the world's energy consumption in 2018, and their demand is predicted to grow significantly in the future [86]. Consequently, data centers and computing clusters contribute noticeably to global energy consumption, necessitating more energy-efficient computation paradigms. Environmental protection requirements motivates the research into power-efficient electronics, the so-called *green electronics*. Physical limitations of CMOS, such as heat generation, support the recent interest in power-efficient technologies based on *superconducting electronics* (SCE).

Superconducting electronics offers effective solutions to the challenges faced by modern CMOS systems, such as stagnating clock frequencies and prohibitive power density. SCE systems can achieve up to 100× lower operating power and 10-100× higher clock frequencies than CMOS [78, 89]. Additionally, SCE systems operate at cryogenic temperatures with millivolt-level signals, producing minimal noise. Whereas this chapter focuses exclusively on digital superconducting electronics, superconducting sensors and communication primitives can also be realized. These advantages have led to SCE applications in areas such as high-resolution sensors for medical and scientific measurements, fast signal processing for wireless communications, and interfaces for quantum computing with superconductive qubits. However, the real potential for SCE electronics lies in applications for data centers and computing clusters.

Superconducting electronics operates at few degree Kelvin (typically 4K), where resistive effects can be neglected and are based on the *Josephson junction* (JJ), consisting of superconductors separated by a barrier. Modern SCE technologies are based on two major superconductive effects, namely, *Josephson effect* and *magnetic flux quantization*. The most mature superconducting logic family leveraging the Josephson effect is the *rapid single-flux quantum* (RSFQ) [119], developed in the late 80s. Meanwhile, the most advanced logic family leveraging the magnetic flux quantization is the *adiabatic quantum-flux parametron* (AQFP) [218]. Both logic families are addressed in this chapter. For further details on SCE, EDA for SCE, and its history, we refer the reader to [18, 20, 119].

Despite successful applications, the scope of SCE applications remains narrow compared to its potential. Most circuits have been designed with significant human intervention. To fully realize the benefits of this technology by scaling up circuit complexity, new *electronic design automation* (EDA) tools are required. Conventional EDA tools for CMOS are not suited for SCE due to fundamental differences between the two technologies. One of the most important differences in these technologies is that logic evaluation at each gate is triggered by a clock signal, due to the unique nature of SCE of representing zeros and ones. Consequently, the inputs at a gate must be available in specific timeframes for the computation to be correct. This often requires the use of delaying registers on certain circuit paths. In literature, this problem is referred to as *path balancing*. Another key difference is the poor driving capacity of SCE gates, due to the small currents involved. This limitation requires the addition of special gates called *splitters* to the logic, which distribute signals to multiple destinations without degrading the signal integrity. In literature, this problem is referred to as *fan-out branching*.

Despite the advances in logic synthesis for SCE, the number of delaying registers required for path balancing and splitters can be prohibitively large, often contributing to 50% of the total area and energy consumption [13, 35, 88, 148]. This significantly degrades the efficiency and yield of superconducting integrated systems. This challenge motivates our research into EDA solutions for superconducting electronics. Specifically, we focus on the problem of technology mapping, which is more complex than the one for CMOS due to the additional requirements of path balancing and fan-out branching.

In Section 6.2, we introduce the two most mature SCE logic families: the *adiabatic quantum-flux parametron* (AQFP) and the *single-flux quantum* (SFQ), along with their features and constraints. Next, Section 6.3 focuses on technology mapping for AQFP circuits. Specifically, we address the problem of satisfying the path balancing and fan-out branching constraints during technology mapping. In this regard, we propose a fully automatic technology mapping flow that solves this problem guaranteeing optimal circuit depth. Then, in Section 6.4, we present a comprehensive framework for synthesizing and optimizing SFQ circuits. This includes algebraic and Boolean optimization techniques on the bases {AND, XOR, NOT} (*xor-and graph* (XAG)), which closely abstract the logic primitives of the SFQ technology, and a technology mapping method to satisfy the path-balancing and fan-out constraints.

6.2 Preliminaries

In this section, we present the two most advanced superconducting logic families: the *adiabatic quantum-flux parametron* (AQFP) and the *single-flux quantum* (SFQ). Additionally, we focus on formalizing the path balancing and fan-out branching, which differ in the two technologies.

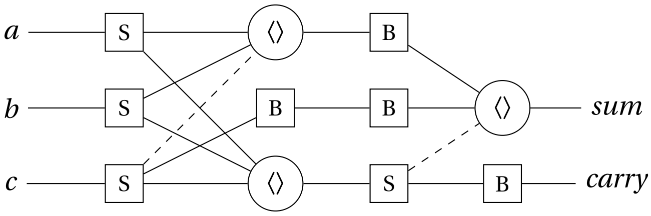


Figure 6.1: An AQFP full adder circuit. Splitter cells (squares labeled S) are used to drive multiple fan-out; 3-input majority cells (circles labeled ⟨⟩) realize the desired logic function; dashed edges indicate an inverted connection; and buffers (squares labeled B) path-balance the circuit. Compared to a MIG realization, the circuit depth increases by two due to splitter cells.

6.2.1 Adiabatic Quantum-Flux Parametron

The *adiabatic quantum-flux parametron* (AQFP) is a superconducting logic family that targets low-energy consumption. In an AQFP circuit, instead of transistors, *Josephson junctions* (JJs) are the active components. In this technology, adiabatic switching operations drastically reduce the dynamic power consumption compared to other superconducting logic families, and achieve zero static power consumption [191]. AQFP circuits operate at frequencies up to 10 gigahertz with a power dissipation of two orders of magnitude lower compared to CMOS, when accounting also for the cryo-cooling energy [14, 42].

The basic circuit components in AQFP are the buffer cell and the branch cell. A majority-3 logic gate can be constructed by combining three buffer cells with a 3-to-1 branch cell, from which other logic gates, such as the AND gate and the OR gate, can be built with constant cells (biased buffer cells). Input negation of logic gates is realized using a negative mutual inductance and is of no extra cost [192]. The commonly used cost metric of AQFP circuits is the JJ count. A buffer requires two JJs, a branch cell is of no JJ cost, and a logic gate based on majority-3 costs six JJs [192].

In AQFP circuits, due to a different encoding of the information compared to CMOS, each logic gate needs an alternating excitation current that periodically releases and resets the computation [189]. The excitation current is delivered as a clock [190]. Thus, data at each gate must be present at specific time frames for correct functionality. This may require the insertion of clocked *buffers* such that all data paths at each gate's fan-in have the same length. This design constraint is called *path-balancing*. Logic gates also have limited driving capabilities. Branching elements called *splitters* are necessary for multiple fan-outs to amplify the output current. A splitter cell is composed of a buffer cell and a 1-to-n branch cell (usually, $2 \leq n \leq 4$) and is also clocked. As the cost of splitters comes mostly from the buffer cells, in this chapter, we do not distinguish buffers from splitters and we will model them with the same abstraction. This second design constraint is called *fan-out-branching*.

To illustrate the AQFP technology constraints, Figure 6.1 shows a full adder implemented as an AQFP circuit that satisfies the path-balancing and fan-out-branching requirements.

Splitters (*S* squares) are inserted to drive multiple gates and buffers (*B* squares) are used to balance paths at the inputs of all gates and over all outputs.

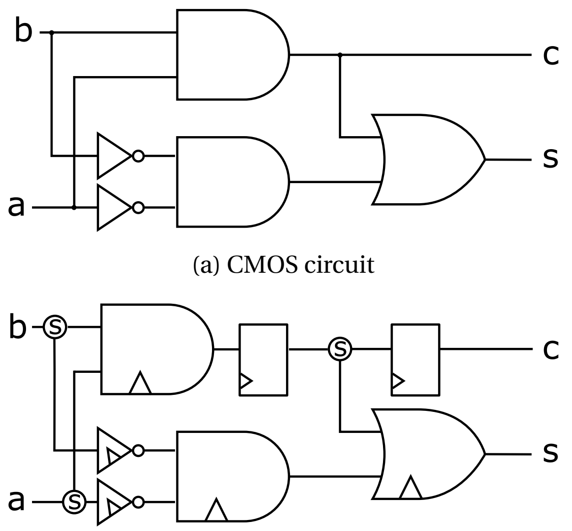
Path-balancing and fan-out-branching constraints complicate the design process and significantly affect area and delay. In some applications, buffers and splitters (B/S) may occupy half of the total area even after optimization [13, 33, 35, 107, 126, 195]. Hence, developing EDA tools able to minimize the number of buffers and splitters is of primary importance. Existing work considered AQFP constraints during logic optimization to reduce imbalances and high-fan-outs by modifying the logic [33, 126, 204]. Other previous work developed techniques to insert and minimize the number of buffers and splitters needed in an AQFP circuit after logic synthesis [35, 80, 107].

In Section 6.3, we propose technology mapping algorithms for inserting buffers and splitters with depth-optimality guarantees, thereby satisfying the path-balancing and fan-out-branching requirements. Additionally, we describe a post-mapping algorithm based on minimum-register retiming to optimize the number of buffer and splitters after the initial insertion.

6.2.2 Single-Flux Quantum

Rapid Single-Flux Quantum (RSFQ) is a fast and energy-efficient superconducting logic family [119] that operate at a few degrees Kelvin (typically 4K) where resistive effects are negligible. The particularity of the RSFQ technology is that it is based on pulsing logic utilizing *Josephson junctions* (JJs) as the primary switching elements. The switching speed of Josephson junctions supports the realization of RSFQ circuits clocked up to several tens of Gigahertz [89] with a considerably lower power consumption compared to CMOS, even considering the refrigeration power [78]. Different variants of RSFQ logic have been proposed in the literature to improve the energy efficiency, such as the *energy-efficient single-flux quantum* (eSFQ, [149]), reciprocal quantum logic (RQL, [76]), and low-voltage RSFQ (LVRSFQ, [193]). In this chapter, we refer to all the variants of this technology as *SFQ*.

Unlike CMOS, SFQ circuits encode the logic “true” in a small voltage pulse and the logic “false” in a pulse absence. Consequently, most SFQ logic gates are clocked to discern between these two states. Furthermore, SFQ gates necessitate a bias current, provided by a bias network, to be able to switch correctly. SFQ logic gates function as latches, with a clock input and one or more data inputs. When a pulse arrives at a data input, it alters the internal state of the gate. Subsequently, a clock pulse resets the gate to its initial state and may generate an output pulse based on the internal state. As SFQ circuits rely on the clock signal, they implement gate-level pipelining. To ensure correct data propagation (to have data at each gate present at specific time-frames for correct computations) SFQ circuits require delay registers (DFFs) in the combinational paths so that every path from primary inputs to logic gates traverses the same number of clocked gates. This constraint is referred to as *path balancing*. Additionally, to enable gate-level pipelining, so that new data can be provided every clock cycle, primary



(a) CMOS circuit

(b) Equivalent RSFQ circuit

Figure 6.2: Mapping of a CMOS circuit (a) into a circuit in SFQ technology (b). First, each CMOS gate is replaced by the corresponding clocked SFQ gate. Then, DFF cells are inserted to satisfy the path-balancing constraint and to balanced POs. Finally, splitter cells (circles labeled S) are used to drive multiple fan-outs. The circuit depth of the SFQ circuit in (b) is of 3 clock cycles.

output (PO) must also be balanced. Balanced POs are an important constraint in sequential systems where the register-level clock is not a (slower) multiple of the gate-level clock.

Due to the quantized nature of SFQ pulses, most RSFQ primitives have a maximum driving capacity of one gate. Consequently, special cells called *splitters* are necessary to drive multiple fan-outs. This requirement is referred to as *fan-out branching*. In contrast with splitter cells in AQFP technology, splitter cells for SFQ are asynchrounous, i.e., they do not need a clock signal. This simplifies the design process of SFQ circuits. Additionally, SFQ splitters can drive only two gates.

Cell libraries in SFQ technology typically comprise of a set of basic combinational blocks that can be realized using JJs, splitters, *mergers* (by reversing the splitter cell), and loops [119]. These basic gates are the DFF, INV, AND2, OR2, and XOR2 [119, 159, 217] all of which require a clock signal to function correctly. Besides the splitter, SFQ cell libraries often contain another asynchronous cell called the *confluence buffer* (CB), or *merger*, which directs signals from two input branches to one output. As for AQFP circuits, the commonly used cost metric of SFQ area, prior to place and route, is based on the JJ count. Differently from AQFP circuits, the delay is expressed as the maximum number of cycles required by the circuit to complete its computation. This value corresponds to the length of the path that traverses the highest number of clocked cells in the combinational logic. The clock frequency is typically neglected before place and route as being hard to characterize [159].

Table 6.1: Comparison between AQFP and SFQ logic families.

	AQFP	SFQ
Clock frequency	Up to 5-10 GHz	Up to several tens of GHz
Path balancing	Clocked buffers	Clocked DFFs
Fan-out branching	Clocked splitters	Asynchronous splitters
Gate driving capacity	1	1
Splitting capacity	1-to-n (usually, $2 \leq n \leq 4$)	1-to-2
Gate library	MAJ, constant	AND2, OR2, XOR2, INV

Figure 6.2 shows the difference between a CMOS circuit and an equivalent SFQ circuit. Notably, in Figure 6.2b gates are clocked and DFFs are inserted to satisfy the path-balancing constraint. Specifically, a DFF is placed before the OR gate, and another one before the output C. Last, splitter cells are inserted to drive multiple fan-outs.

In Section 6.4, we present a comprehensive exploration of methods for synthesizing and optimizing SFQ circuits. Our approach includes algebraic and Boolean optimization techniques based on the *xor-and graph* (XAG) representation of logic, a technology mapping method to satisfy path balancing and fan-out constraints, and a synthesis flow for SFQ circuits.

6.2.3 Key Points

In this section, we briefly presented the two most mature logic families in superconducting electronics. Their differences and similarities are summarized in Table 6.1. The *adiabatic quantum-flux parametron* (AQFP) is a majority-based logic that supports clock frequencies up to 5-10 gigahertz. Gates require an alternating excitation current that act as a clock. Hence, circuits need to be path-balanced using clocked buffer gates. Gates cannot drive multiple outputs without using a splitter, which is also clocked. Conversely, the *rapid single-flux quantum* (RSFQ) is an AND-OR-XOR-based logic that supports clock frequencies up to several tens of gigahertz. It represents zeros and ones using the absence or presence of a voltage pulse. Consequently, gates are clocked to discern between the two states. Path balancing is implemented using clocked DFF cells. Gates cannot drive multiple outputs without using a splitter, which is an asynchronous gate.

6.3 Technology Mapping for AQFP Circuits

High-performance computing of data centers and computing clusters contributes to a noticeable percentage of the world's energy consumption, demanding more energy-efficient computation paradigms. The *adiabatic quantum-flux parametron* (AQFP) is an emerging superconducting technology shown to achieve promising energy efficiency [191] and has attracted increasing attention in the past decade. While the technology is rapidly evolving [74, 174, 189, 190, 192] and larger-scale systems are being developed [14, 207], design automation for AQFP is also an extensively researched topic [13, 129, 212].

One major challenge in AQFP design automation is the legalization of the logic circuit to fulfill two unconventional technology constraints, *path balancing* and *fan-out branching*, before physical design. Due to its gate-level clocking property, AQFP gates require all input signals to arrive at the same time, thus buffers have to be inserted on shorter data paths to balance the delay along the longer paths. Moreover, splitters are needed at the output of AQFP gates driving multiple signals, and these splitters are also clocked. Thus, logic circuits generated by technology-independent logic synthesis must be *legalized* for the AQFP technology by inserting buffers and splitters. Legalization of AQFP circuits is essential to unlock its potential of pipelined computation while maintaining correct functionality.

In a legalized AQFP circuit, buffers and splitters (B/S) often contribute to over 50% of the *Josephson junction* (JJ) count, which is the commonly used cost metric related to area as well as energy consumption. Thus, optimization algorithms for AQFP legalization are important to reduce the overhead and increase scalability of AQFP circuits. In this work, we present a scalable and flexible framework for AQFP technology legalization and optimization. First, we prove that the depth-optimal B/S insertion problem is tractable with polynomial complexity. Next, based on the formulation of the AQFP B/S insertion problem as a scheduling problem [105], we propose depth-optimal scheduling algorithms, forming the basis for obtaining an initial legalized circuit. Then, we present a heuristic optimization algorithm to further optimize the B/S count based on minimum register retiming. Finally, we present an AQFP legalization and optimization flow consisting of two depth-optimal schedules, iteratively optimizing them separately, and then choosing the better one.

Our experiments demonstrate remarkable results:

1. We show a reduction in the number of buffer and splitters up to 14% compared to the previous state-of-the-art scheduling-based B/S insertion algorithm in [105], with no depth optimality guarantees. Additionally, we show a logic depth improvement up to 15%.

2. Compared to the current state of the art, we show competitive results that are depth optimal. More importantly, we demonstrate that our approach can perform the B/S insertion in a very short run time, making possible to use these methods in a design-space exploration (DSE) engine. Additionally, we show that our method achieves near-optimal quality as the state-of-the-art ILP-based algorithm within very little runtime in small benchmarks.

3. We show that our methods are the first to scale to benchmarks that are 10 to 100 times larger compared to typical benchmarks used in AQFP logic synthesis, more than any other related work could handle.

6.3.1 Preliminaries

In this section, we provide terminology and describe the technology mapping problem for AQFP circuits. Additionally, we summarize previous work that formalizes the AQFP buffer and splitter insertion problem as a scheduling problem.

Terminology

A *(logic) network* is a directed acyclic graph defined by a pair (V, E) of a set V of nodes and a set E of directed edges. The node set $V = I \cup O \cup G$ is disjointly composed of a set I of *primary inputs* (PIs), a set O of *primary outputs* (POs), and a set G of *(logic) gates* chosen from a library. In this paper, we assume that an AQFP-compatible gate library (e.g., composed of AND2, OR2, MAJ3, with optional input negation) is used. Each PI has in-degree 0 and unbounded out-degree, whereas each PO has in-degree 1 and out-degree 0. The out-degree of each gate is unbounded and the in-degree is a fixed number depending on the type of the gate. For any gate $g \in G$, the *fan-ins* of g, denoted as $FI(g)$, is the set of gates and PIs connected to g on an incoming edge. Similarly, the *fan-outs* of a gate (or a PI) g, denoted as $FO(g)$, is the set of gates and POs connected to g on an outgoing edge.

A *mapped network* N' is a network whose node set V' is extended with a set B of *buffers*. A buffer is a node with in-degree 1. In a mapped network, the definition of the fan-out of a gate is modified by ignoring any intermediate buffers, i.e., a path from a gate g to one of its fan-outs $g_o \in FO(g) \subset (G \cup O)$ may include any number of buffers in B, but never another gate. The definition of fan-ins is modified similarly. The *fan-out tree* of a gate (or a PI) n, denoted by $FOT(n)$, is the set of buffers between n and any gate or PO in $FO(n)$.

A *schedule* of a network is a function $\mathcal{S} : V \to \mathbb{Z}_{\geq 0}$ that assigns a non-negative integer $\mathcal{S}(n)$ to each node $n \in V$, called the *level* of n. The depth of a network $N = (V = I \cup O \cup G, E)$ with a schedule $\mathcal{S}$ is defined as $d(N) = \max_{o \in O} \mathcal{S}(o)$. If the schedule is omitted, then the depth of a network is the length of the longest path from any PI to any PO.

Problem formulation

To fulfill the needs in the AQFP technology for fan-out-branching and path-balancing, we define the following properties subject to the *splitting capacities* $s_i = 1, s_g = 1$, and $s_b > 1$ of PIs, gates, and buffers, respectively.

Definition 6.3.1. *Given a mapped network* $N' = (V' = I \cup O \cup G \cup B, E')$,

1. *N' is* path-balanced *if there exists a schedule $\mathscr{S}$ of N' such that*

$$\forall n_1, n_2 \in V' : (n_1, n_2) \in E' \Rightarrow \mathscr{S}(n_1) = \mathscr{S}(n_2) - 1, \tag{6.1}$$

$$\forall i \in I : \mathscr{S}(i) = 0, \textit{ and} \tag{6.2}$$

$$\forall o \in O : \mathscr{S}(o) = d(N'). \tag{6.3}$$

2. *N' is* properly branched *if every PI has an out-degree no greater than $s_i = 1$, every gate has an out-degree no greater than $s_g = 1$, and every buffer has an out-degree no greater than s_b.*

3. *N' is* legal *if it is both path-balanced and properly branched.*

In an AQFP design automation flow, the logic synthesis stage after RTL synthesis and before physical design converts an input specification netlist (represented as, e.g., an *AND-Inverter Graph* (AIG) or a *Majority-Inverter Graph* (MIG)) into a legal mapped network whose gates are all AQFP-compatible. The problem to be solved is formulated as follows:

Problem 6.3.2 (AQFP technology mapping)**.** *Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in G, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$ such that:*

1. *N and N' are logically equivalent.*

2. *All gates in G' are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation).*

3. *N' is legal (i.e., path-balanced and properly branched).*

Problem 6.3.2 may be solved as one problem, or it may be divided into two problems to be solved independently:

Problem 6.3.3 (Majority-based logic restructuring)**.** *Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in G, find a network $N^* = (V^* = I \cup O \cup G^*, E^*)$, such that:*

1. *N and N^* are logically equivalent.*

2. *All gates in G^* are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation).*

Problem 6.3.4 (AQFP technology legalization)**.** *Given a network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and the value of s_b, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$, such that:*

1. *N' is legal (i.e., path-balanced and properly branched).*

2. *$G' = G^*$, and for all gates $g \in G^*$, $FO(g)$ and $FI(g)$ remain the same in N' as in N^*.*

Moreover, for all three problems, in addition to finding a network fulfilling the requirements, we also optimize common metrics. For the main problem to solve, Problem 6.3.2, common optimization objectives are minimizing the JJ count (#JJs $= 6 \cdot |G'| + 2 \cdot |B|$) and minimizing JJ depth $d(N')$.

Problem 6.3.3 is equivalent to mapping into and optimizing a *majority-inverter graph* (MIG) [5], which is a logic network where all gates are MAJ3 and edges may contain inverters, because AND2 and OR2 gates are equivalent to MAJ3 with a constant (0 and 1, respectively) input. Graph mapping [202] and MIG optimization [5, 104, 168, 196] are well-researched problems with existing algorithms to use. These algorithms usually optimize the MIG size ($|G^*|$) or depth ($d(N^*)$).

In Sections 6.3.3 and 6.3.4, we focus on solving Problem 6.3.4. Since $G' = G^*$, this problem is often referred to as the AQFP buffer (and splitter) insertion problem. Minimizing JJ count in Problem 6.3.2 is equivalent to minimizing $|B|$ in Problem 6.3.4.

AQFP legalization as a scheduling problem

A close collaborator, Siang-Yun Lee, demonstrated that the AQFP legalization problem, or buffer and splitter insertion problem, (Problem 6.3.4) can be seen as a scheduling problem on the umapped network. In this subsection, we review Lee's work on scheduling for AQFP networks, as it forms the foundation for our contribution.

Once a schedule is given, the minimal-size mapped network can be derived in linear time using an *irredundant* buffer insertion algorithm [105, 107, 110]. An irredundant mapped network is defined as follows.

Definition 6.3.5. *A mapped network is said to be* irredundant *if the following two conditions hold.*

1. *There is no dangling buffer, i.e., every buffer has at least one outgoing edge.*
2. *There does not exist any pair of buffers whose incoming edges are connected from the same splitter and both of them have out-degrees smaller than s_b.*

Otherwise, the network is redundant.

A schedule of the network is *legal* if and only if a mapping function $f : (N, \mathscr{S}) \to N'$ exists such that buffers and splitters can be inserted respecting the path-balancing and fan-out-branching constraints while maintaining each node $n \in V$ at the assigned level $\mathscr{S}(n)$ in the schedule.

Algorithm 6.1 from [105] shows such function. For each PI or gate n, Algorithm 6.1 iterates over all levels l between n and its fan-out. Initially, the set A contains the fan-outs (gates

Algorithm 6.1: Irredundant buffer insertion

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and a schedule $\mathscr{S}$ for $N^*$
Output: Legalized mapped network $N'$
$N' \leftarrow N^*$
foreach $n \in I \cup G^*$ do
    $l_{\max} \leftarrow \max_{n_o \in FO(n)} \mathscr{S}(n_o)$
    $A \leftarrow \{n_o \in FO(n) : \mathscr{S}(n_o) = l_{\max}\}$
    for $l = l_{\max} - 1$ downto $\mathscr{S}(n) + 1$ do
        Create $\left\lceil \frac{|A|}{s_b} \right\rceil$ buffers at level $l$ in $N'$
        $B \leftarrow$ the set of newly created buffers
        for $i = 1$ to $|A|$ do
            Remove $n$ from $A[i]$'s fan-ins in $N'$
            Add $B[\lceil \frac{i}{s_b} \rceil]$ as $A[i]$'s fan-in in $N'$
        $A \leftarrow B \cup \{n_o \in FO(n) : \mathscr{S}(n_o) = l\}$
    assert $|A| = 1$
    Add $n$ as $A[1]$'s fan-in in $N'$
return $N'$
```

and POs, if any) of n at the highest level $l_{\max}$. At each level l, enough buffers ($|B| = \lceil \frac{|A|}{s_b} \rceil$) are inserted, where $|A|$ is the number of nodes at level $l+1$. Then, gate n is removed from the fan-ins of the i-th element in A, and the $\lceil \frac{i}{s_b} \rceil$-th buffer in B is added instead. Finally, A is updated as the newly created buffers and the fan-outs at the current level.

Algorithm 6.1 runs in linear time with respect to $\sum_{n \in I \cup G^*} |FO(n)| \leq |E^*|$. Moreover, the constructed mapped network is irredundant because in each fan-out tree, only the minimum number of buffers is inserted at each level l and only at most one of them has fan-out count smaller than s_b. An irredundant network is size-optimal with respect to the given schedule because no buffer can be removed while keeping the network legal.

Given Algorithm 6.1, the AQFP legalization problem reduces to finding a legal schedule that minimizes the number of buffers and splitters. In Section 6.3.3, we propose scheduling algorithms for AQFP with depth-optimality guarantees.

6.3.2 Related Works

In this section, we introduce existing works solving the three problems formulated in Section 6.3.1. We first present related work in MIG optimization, corresponding to Problem 6.3.3. Then, we transition to AQFP technology mapping problems that tackle Problem 6.3.4. Finally, we present the state-of-the-art approaches for Problem 6.3.2.

Majority-inverter graph optimization

MIG was proposed as an alternative technology-independent logic representation with an advantage in depth optimization, especially in arithmetic circuits [5]. Due to special properties of some emerging technologies, including AQFP, MIG also become a good logic synthesis data structure for these technologies [204]. Various logic synthesis and optimization algorithms have been proposed and tailored for MIGs. To convert an AIG into an MIG, the simplest way is to translate each AND2 gate into an MAJ3 gate with a constant 0 input. Alternatively, a versatile graph mapping algorithm can also map from AIGs (or other types of networks) to MIGs while optimizing for depth and/or size in the process [196, 202] (e.g., see Chapter 5). Prominent examples of tailored MIG optimization algorithms include algebraic rewriting, which applies special Boolean algebraic rules to reduce MIG depth [5], Boolean rewriting [196], and Boolean resubstitution, which resynthesizes a small part of the network using majority gates to reduce MIG size [104].

Buffer and splitter insertion and optimization

(Rapid) Single-Flux Quantum (RSFQ or SFQ) [119] is a sibling superconducting technology of AQFP and has similar path-balancing and fan-out-branching constraints, thus also requiring buffer and splitter insertion [88, 159, 194] (Section 6.4). However, a key difference between the two technologies makes the problem computationally distinct for them: in SFQ, splitters are not clocked and not considered in path balancing, so fan-out branching and path balancing can be considered separately; whereas AQFP splitters are clocked, thus fan-out branching and path balancing must be considered together to discover potential optimizations. The interplay between buffers and splitters makes the B/S optimization problem for AQFP a challenging one.

In the earliest AQFP design automation tools, legalization was done by first inserting splitters (as balanced trees) at the output of all multi-fan-out gates, and then inserting buffers on all imbalanced paths [212]. This was a rather naive approach that guaranteed the correct operation of the AQFP circuit but often resulted in a large portion of JJ count taken by buffers and splitters. Thus, a local optimization technique called retiming [13] or buffer merging [34] was proposed. While this approach is called retiming, it does not perform global retiming, or global optimization, but only moves buffers across a multi-fan-in gate or a multi-fan-out splitter when locally convenient. For example, moving buffers from the fan-ins of a MAJ3 gate to its fan-out reduces by three times the number of buffers (Figure 8 in [34]); alternatively, moving buffers from the fan-outs of a splitter to its fan-in can be seen as sharing buffers or delayed splitting and also reduces the buffer count (Figure 5 in [13]). This idea was elaborated in [35] as a B/S insertion algorithm using the notion of virtual splitters.

Further improvements to the B/S optimization problem involving more sophisticated algorithms were made in the following years. In [80], the authors attempted to localize the optimization problem to a single wire and proposed a locally optimal algorithm subject to a complex cost function involving maximum and total additional delay and the number of B/S. The local insertion algorithm has a quadratic complexity. In [105], the authors formulate the B/S insertion problem as a scheduling problem and propose algorithms based on the *as-soon-as-possible* (ASAP) and *as-late-as-possible* (ALAP) strategies. This approach has a linear complexity relative to the number of nodes but does not guarantee optimality in terms of circuit depth or size. To optimize the number of B/S after the initial insertion, the authors propose an algorithm to move chunks on logic up or down by reconstructing splitters and moving buffers. In [62], the authors proposed to first solve for a schedule of the mapped network, formulated as an *integer linear programming* (ILP) problem with an objective function estimating the B/S count, followed by another locally optimal splitter-tree insertion algorithm subject to the same cost function defined in [80]. This local insertion algorithm has a cubic time complexity.

Exact methods solving for the global size-optimal B/S insertion were also researched. In [105], the B/S optimization problem was first formulated as a scheduling problem, encoded as an optimization modulo linear integer arithmetic problem, and solved by a *satisfiability modulo theory* (SMT) solver. The global minimum B/S insertion results were obtained for some small benchmarks. Then, an ILP encoding was proposed in [125] which led to some improvement in efficiency, and optimal results for more benchmarks were reported. However, the exact methods remain applicable only to small or medium-size regular designs.

While previous heuristic work did not offer size or depth optimality guarantees, this work introduces two B/S insertion algorithms that ensure depth optimality. Additionally, we propose an optimization method for B/S based on global register retiming.

AQFP logic synthesis

Existing AQFP logic synthesis flows can be categorized into two approaches: solving Problem 6.3.3 and Problem 6.3.4 separately, or considering Problems 6.3.3 and 6.3.4 together. The earliest works took the first approach to adapt available CMOS-based design automation tools for AQFP [13, 212]. Problem 6.3.3 was addressed by AND-based technology-independent logic synthesis followed by technology mapping into an AQFP-compatible library, and Problem 6.3.4 was solved separately in an additional buffer insertion stage before physical design. Later, to better leverage the intrinsic MAJ function in AQFP circuits, MAJ-based logic synthesis was adopted [34, 204]. At this time, Problem 6.3.4 was still solved separately using the naive insertion approach.

Although solving the two problems separately is easier, it is hard to predict the impact of legalization in the logic restructuring stage. The smallest MIG in size may not be still the smallest after legalization. Thus, in [126], the authors proposed to consider the two problems together and optimize directly for the final cost function. A database of optimal AQFP subcircuits is used in restructuring, and legalization is done during the process. This algorithm was used in a flow consisting of graph mapping, AQFP resynthesis, and post-synthesis buffer optimization [129].

The latest work on AQFP synthesis [110], presenting currently the best results, took the first approach (separating the two problems) and used a design-space exploration engine to find the best AQFP by combining MIG optimization (including the methods proposed in Chapter 5) to the B/S algorithms proposed in [105] and this chapter.

6.3.3 Depth-Optimal Buffer and Splitter Insertion

This section presents our first contribution, which proposes how to efficiently approach Problem 6.3.4. As discussed in Section 6.3.1, common cost metrics to be considered for AQFP circuits are network size and depth. Unlike in many other technologies where circuit area and delay are often inversely related in a Pareto curve and engineers must trade one for the other, we observe that in the AQFP buffer insertion problem, the size of an irredundant mapped network correlates with the depth of the provided schedule. Intuitively, in Problem 6.3.4, the unmapped network and any mapped network have roughly the same number of paths and similar logic sharing (slight differences may only exist in how fan-outs are split), and the size of a mapped network is the sum of all path lengths, which depends on the network depth and the sizes of the shared cones. In other words, a larger network depth results in longer (balanced) paths and thus larger network size due to the presence of more buffers. This motivates us to present scheduling algorithms that also optimize for depth besides being fast (having a linear time complexity) and giving legal results.

In this section, we present depth-optimal buffer and splitter insertion algorithms, based on the well-known *As-Soon-As-Possible* (ASAP) and *As-Late-As-Possible* (ALAP) scheduling strategies. Following [105], we formulate the AQFP technology legalization problem as a scheduling problem. These methods provide a legal schedule for an unmapped network, allowing the derivation of an irredundant legal mapped network using Algorithm 6.1. These algorithms are designed to serve as quick initial scheduling methods, which will undergo further optimization to reduce the number of B/S elements (Section 6.3.4).

This section is organized as follows. We begin by presenting a depth-optimal algorithm that assigns a node to a level in the schedule, ensuring that its fan-out tree is of minimum height. Next, we introduce two algorithms to compute a schedule based on the ALAP and ASAP scheduling strategies.

Depth-optimal scheduling

Given a partial schedule $\mathscr{S}$ where some nodes, including node n but excluding all fan-outs of n, have not been assigned a level, Algorithm 6.2 computes the value to be assigned to $\mathscr{S}(n)$, such that the fan-out tree of n has the minimum-possible height. Variable *edges* counts the number of nodes (thus edges) needed to be connected at each level; variable *splitters* computes the number of splitters (buffers) needed at each level. The foreach-loop (lines 3 to

Algorithm 6.2: Depth-optimal single node scheduling

```
   Input: A node $n$ and a partial schedule $\mathscr{S}$
   Output: Level $\mathscr{S}(n)$ assigned to node $n$
1  $l_{prev} \leftarrow \max_{n_o \in FO(n)} \mathscr{S}(n_o)$
2  $edges \leftarrow 0$
3  foreach $n_o \in FO(n)$ in a descending order of $l \leftarrow \mathscr{S}(n_o)$ do
4      $splitters \leftarrow \left\lceil \frac{edges}{s_b^{(l_{prev}-l)}} \right\rceil$
5      $edges \leftarrow splitters + 1$
6      $l_{prev} \leftarrow l$
7  while $edges \neq 1$ do
8      $edges \leftarrow \left\lceil \frac{edges}{s_b} \right\rceil$
9      $l_{prev} \leftarrow l_{prev} - 1$
10 $\mathscr{S}(n) \leftarrow l_{prev} - 1$
11 return $\mathscr{S}(n)$
```

6) iterates over the fan-outs of n in descending order of their levels, and variable l_{prev} keeps the level of the previous iteration. If the level does not change from the previous to the current iteration, variable *splitters* is equal to *edges* because $l_{prev} = l$ and ${s_b}^0 = 1$ (line 4). As a result, *edges* is simply increased by 1 in this iteration, counting the fan-out itself (line 5). Otherwise, when a fan-out at a lower level is encountered, we compute the minimum number of buffers needed at level l to drive *edges* nodes at level l_{prev} as follows. A complete binary tree of height h has at most 2^h leaves. Similarly, a splitter tree rooted at level l can split into at most ${s_b}^h$ fan-outs at level $l + h$. To drive *edges* fan-outs at level l_{prev}, at least $\left\lceil \frac{edges}{s_b^{(l_{prev}-l)}} \right\rceil$ splitter trees rooted at level l are needed (line 4). Moreover, at most one of them is not full, i.e., they are irredundant. In line 5, this value, plus one for the fan-out node itself, is used to update variable *edges*. Finally, after all fan-outs of node n have been processed, the algorithm finds the highest level where *edges* is one to schedule n (lines 7 to 10).

Example 6.3.6. *Figure 6.3 shows an example to illustrate Algorithm 6.2. The node n to be scheduled has four fan-outs, assigned respectively to levels* 8 *(*n_1*,* n_2*,* n_3*) and* 7 *(*n_4*) in the partial schedule. The splitting capacity is* $s_b = 2$*. In Figure 6.3,* $edges_{(v,l)}$ *indicates the value of variable edges in Algorithm 6.2 when node* n_v *at level* $\mathscr{S}(n_v) = l$ *is considered in the foreach-loop (lines 3 to 6). First,* $edges_{(1,8)} = 1$*,* $edges_{(2,8)} = 2$ *and* $edges_{(3,8)} = 3$ *are computed, essentially counting the number of fan-outs at level* $l = 8$*. When node* n_4 *at a lower level,* $l = 7$*, is encountered, the number of buffers needed at level* 7 *to drive all nodes at the previously considered level* $l_{prev} = 8$ *is computed by* $\lceil 3/2^{8-7} \rceil = 2$*. The loop ends with* $l_{prev} = 7$ *and* $edges = 3$*. Finally, in the while-loop (lines 7 to 9),* edges *is updated two times before it reaches value* 1*, resulting in* $l_{prev} = 5$*. Thus, node n is scheduled at* $\mathscr{S}(n) = 4$*.* ▲

With the following Lemma, we show that the computation in line 4 of Algorithm 6.2 has the equivalent effect of iterating the splitter counting $splitters \leftarrow \left\lceil \frac{\text{edges}}{s_b} \right\rceil$ for $l_{prev} - l$ levels, as in line 6 of Algorithm 6.1.

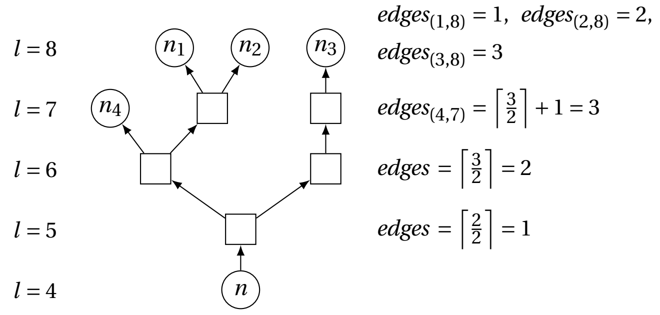

$l = 8$
$l = 7$
$l = 6$
$l = 5$
$l = 4$
n_1
n_2
n_3
n_4
n
$edges_{(1,8)} = 1,\ edges_{(2,8)} = 2,$
$edges_{(3,8)} = 3$
$edges_{(4,7)} = \left\lceil \frac{3}{2} \right\rceil + 1 = 3$
$edges = \left\lceil \frac{3}{2} \right\rceil = 2$
$edges = \left\lceil \frac{2}{2} \right\rceil = 1$

Figure 6.3: Example sub-network to illustrate Algorithm 6.2 with ($s_b = 2$).

Lemma 6.3.7. *Let b be a positive integer and $A = a_0, a_1, \dots, a_n$ be a sequence of $n+1$ positive integers related by $a_{i+1} = \lceil \frac{a_i}{b} \rceil, 0 \le i < n$. Then, $a_n = \lceil \frac{a_0}{b^n} \rceil$.*

Proof. We first prove that for any positive integers a and b, $\left\lceil \frac{\lceil \frac{a}{b} \rceil}{b} \right\rceil = \lceil \frac{a}{b^2} \rceil$. Let $x = \lceil \frac{a}{b} \rceil$, by definition, we have

$$\frac{a}{b} \le x \Rightarrow \frac{a}{b^2} \le \frac{x}{b} \Rightarrow \left\lceil \frac{a}{b^2} \right\rceil \le \left\lceil \frac{x}{b} \right\rceil.$$

Suppose, for the sake of contradiction, that $\lceil \frac{a}{b^2} \rceil < \lceil \frac{x}{b} \rceil$ (the equality is removed), then there must exist an integer y such that $\frac{a}{b^2} \le y < \frac{x}{b}$. Multiplying by b and using y, we have

$$\frac{a}{b} \le b \cdot y < x = \left\lceil \frac{a}{b} \right\rceil \le \lceil b \cdot y \rceil = b \cdot y < x,$$

which leads to the absurd statement $x < x$. Thus, by contradiction, $\lceil \frac{a}{b^2} \rceil = \lceil \frac{x}{b} \rceil = \left\lceil \frac{\lceil \frac{a}{b} \rceil}{b} \right\rceil$ and the statement is proved by induction on i. ■

Next, we prove the legality and optimality of Algorithm 6.2 with the following theorem.

Theorem 6.3.8. *Given a legal partial schedule $\mathscr{S}$, Algorithm 6.2 assigns the largest level to $\mathscr{S}(n)$ such that $\mathscr{S}$ is still legal.*

Proof. Let the value returned by Algorithm 6.2 be l_n and assume, for the sake of contradiction, a schedule $\mathscr{S}'$ where $\mathscr{S}'(n_o) = \mathscr{S}(n_o)\ \forall n_o \in FO(n)$ and $\mathscr{S}'(n) = l'_n > l_n$. Let $l_m = \min_{n_o \in FO(n)} \mathscr{S}(n_o)$. If $l'_n \ge l_m$, $\mathscr{S}'$ is obviously illegal. Assume $l'_n < l_m$. Let e be the value of variable *edges* when the foreach-loop in Algorithm 6.2 (lines 3 to 6) ends. The while-loop in Algorithm 6.2 has $l_m - l_n - 1$ iterations, so the value of variable *edges* before the last iteration is, by Lemma 6.3.7, $\left\lceil e / s_b^{(l_m - l_n - 2)} \right\rceil > 1$.

Now, consider an execution of Algorithm 6.1 using $\mathscr{S}'$, in particular the iteration of the outer loop processing the considered node n, we have $|A| = e$ after line 11 in Algorithm 6.1 in the iteration $l = l_m - l'_n$. The loop (lines 5-11 in Algorithm 6.1 has $l_m - l'_n - 1$ more iterations before it ends, in which line 11 can be replaced by "$A \leftarrow B$" because there are no more fan-outs.

Algorithm 6.3: Depth-optimal ALAP scheduling

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$
Output: A schedule $\mathscr{S}$ for $N^*$
$\lambda \leftarrow d(N^*) \cdot (1 + \max_{n \in V^*} \lceil \frac{\log(|FO(n)|)}{\log(s_b)} \rceil)$
foreach $o \in O$ do
    $\mathscr{S}_\lambda(o) \leftarrow \lambda$
foreach $n \in I \cup G^*$ in a reversed topological order do
    $\mathscr{S}_\lambda(n) \leftarrow$ schedule_node($n, \mathscr{S}_\lambda$)                ▷ Run Algorithm 6.2
$l_{\min} \leftarrow \min_{i \in I} \mathscr{S}_\lambda(i)$
foreach $i \in I$ do
    $\mathscr{S}(i) \leftarrow 0$
foreach $n \in O \cup G^*$ do
    $\mathscr{S}(n) \leftarrow \mathscr{S}_\lambda(n) - l_{\min}$
return $\mathscr{S}$
```

By the end of the loop, $|A| = \left\lceil e/s_b^{(l_m - l'_n - 1)} \right\rceil \geq \left\lceil e/s_b^{(l_m - l_n - 2)} \right\rceil > 1$. Thus, we conclude that $\mathscr{S}'$ is illegal, and l_n is indeed the largest possible value for $\mathscr{S}(n)$. ■

If all fan-outs of a node n are scheduled at the largest level, then the level of n obtained by Algorithm 6.2 is also the largest. Formally, this is written as follows.

Corollary 6.3.9. *Given a legal schedule $\mathscr{S}$ and a node n, let $\mathscr{S}(n)$ be the level of n computed by Algorithm 6.2. If there does not exist a legal schedule $\mathscr{S}'$ such that* $\max_{o \in O} \mathscr{S}'(o) = \max_{o \in O} \mathscr{S}(o)$ *and $\exists n_o \in FO(n), \mathscr{S}'(n_o) > \mathscr{S}(n_o)$, then there does not exist a legal schedule $\mathscr{S}'$ such that* $\max_{o \in O} \mathscr{S}'(o) = \max_{o \in O} \mathscr{S}(o)$ *and $\mathscr{S}'(n) > \mathscr{S}(n)$.*

Algorithm 6.2 requires that a node is only scheduled after all of its fan-outs have been scheduled. In other words, a reversed topological order is required. Thus, it is suitable to use an ALAP scheduling scheme, which first schedules all POs of a network to an upper bound λ, and then schedules the remaining nodes to the largest-possible level ("as late as possible") in a reversed topological order. We present Algorithm 6.3 for this purpose. It first computes a sufficiently large upper bound λ on the depth of the mapped network for ALAP scheduling, assuming each node would need a balanced splitter tree to drive the maximum fan-out in the network. POs are first scheduled at λ. Then, each node is scheduled using Algorithm 6.3 in a reversed topological order. Finally, to obtain a schedule independent of the value of λ, post-scheduling correction is applied: PIs are moved to level 0 to fulfill Equation 6.2, and the levels of all other nodes are reduced by the smallest PI level before correction. This algorithm has a linear time complexity with respect to the network size.

Depth-optimal ALAP scheduling

We have shown with Corollary 6.3.9 that the depth-optimal scheduling problem has optimal substructure when nodes are scheduled in a reversed topological order. Now, we can prove that Algorithm 6.3 achieves optimal depth.

Theorem 6.3.10. *Given an unmapped network N^*, let the schedule for N^* returned by Algorithm 6.3 be $\mathscr{S}$. The irredundant mapped network N', obtained by running Algorithm 6.1 with N^* and $\mathscr{S}$ as inputs, is legal and its depth $d(N')$ is minimal.*

Proof. At line 6 of Algorithm 6.3, the depth of schedule $\mathscr{S}_\lambda$ is $\max_{o \in O} \mathscr{S}_\lambda(o) = \lambda$ by definition. After the correction in lines 6-10, the maximum level becomes $\lambda - l_{\text{min}}$, which is also the resulting depth $d(N')$. Thus, minimizing depth $d(N')$ is equivalent to maximizing the lowest PI level l_{min} during scheduling because λ is a constant.

In Algorithm 6.3, levels of POs are maximized to λ. By Corollary 6.3.9, each node is scheduled at the largest level because all of its fan-outs are scheduled before it and they are also scheduled at their largest possible levels. By induction, levels of all nodes are maximized and thus depth is minimized. The legality of $\mathscr{S}$ is similarly proved by Theorem 6.3.8. ■

In conclusion, Algorithm 6.3 guarantees to find a legal schedule for an unmapped network. Followed by Algorithm 6.1, a legal mapped network is obtained in linear time. By Theorem 6.3.10, such mapped network is depth-optimal.

Depth-optimal ASAP scheduling

The methods previously presented give a depth-optimal mapped network, but size optimality is not guaranteed. Indeed, the AQFP size optimization problem is likely a difficult one without an algorithm that is both optimal and has polynomial time complexity. Thus, we propose to use depth-optimal networks as starting points and further optimize for size with heuristic algorithms presented in Section 6.3.4. As heuristics are often biased by the starting point, we present in this section an alternative depth-optimal scheduling method based on ASAP instead of ALAP scheduling.

An ASAP scheduling scheme schedules each node, in a topological order, to the lowest-possible level according to the schedule of its fan-ins. To do so, we define a *mobility function* $\mu : V^* \to \mathbb{Z}_{\geq 0}$ representing the maximum negative displacement that can be made to a node (from $\mathscr{S}_{\text{ALAP}}(n)$ to $\mathscr{S}_{\text{ASAP}} = \mathscr{S}_{\text{ALAP}}(n) - \mu(n)$) while keeping the schedule legal and depth-optimal. Algorithm 6.4 computes (a lower bound on) the mobility of each node and uses these values to obtain an ASAP schedule using a given ALAP schedule.

Mobility is initialized to infinite for gates and to 0 for PIs (lines 1-4). For each node n in topological order, first, n is scheduled to a lower level based on its ALAP schedule and mobility

Algorithm 6.4: Depth-optimal ASAP scheduling

```
   Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and its ALAP schedule $\mathscr{S}_{\mathrm{ALAP}}$
   Output: ASAP schedule $\mathscr{S}_{\mathrm{ASAP}}$ for $N^*$
1  foreach $i \in I$ do
2      $\mu(i) \leftarrow 0$
3  foreach $n \in G^*$ do
4      $\mu(n) \leftarrow \infty$
5  $\mathscr{S}_{ASAP} \leftarrow \mathscr{S}_{ALAP}$
6  foreach $n \in G^*$ in a topological order do
7      $\mathscr{S}_{ASAP}(n) \leftarrow \mathscr{S}_{ASAP}(n) - \mu(n)$
8      foreach $n_o \in FO(n)$ do
9          $T(n_o) \leftarrow 0$
10     $l_{prev} \leftarrow \max_{n_o \in FO(n)} \mathscr{S}_{ASAP}(n_o)$
11     $edges \leftarrow 0$
12     foreach $l = \mathscr{S}(n_o) : n_o \in FO(n)$ in descending order do
13         $mobility \leftarrow 0$
14         for $l_{prev} - l$ iterations do
15             if $edges = 1$ then
16                 $mobility \leftarrow mobility + 1$
17             $edges \leftarrow \lceil \frac{edges}{s_b} \rceil$
18         foreach $n'_o \in FO(n) : \mathscr{S}(n'_o) > l$ do
19             $T(n'_o) \leftarrow T(n'_o) + mobility$
20         $edges \leftarrow edges + 1$
21         $l_{prev} \leftarrow l$
22     $mobility \leftarrow 0$
23     for $l = \mathscr{S}_{ASAP}(n)$ upto $l_{prev} - 2$ do
24         if $edges = 1$ then
25             $mobility \leftarrow mobility + 1$
26         $edges \leftarrow \lceil \frac{edges}{s_b} \rceil$
27     foreach $n_o \in FO(n)$ do
28         $\mu(n_o) \leftarrow \min(\mu(n_o), T(n_o) + mobility)$
29 return $\mathscr{S}_{ASAP}$
```

(line 7). Then, the mobilities of its fan-outs are updated using a similar computation as in Algorithm 6.2. A map T stores the temporary mobilities of the fan-outs of n, initialized to zero (lines 8-9). The foreach-loop in lines 12-21 is similar to lines 3-6 in Algorithm 6.2, except that the computation of variable *splitters* in Algorithm 6.2 is rewritten as a loop (lines 14-17) to compute the local mobility (variable *mobility*), which is the number of buffers needed to balance the splitter tree from level l_{prev} to l, and is added to the temporary mobilities T of all the processed fan-outs (lines 18-19). Again, the for-loop in lines 23-26 is similar to lines 7-9 in Algorithm 6.2, where the local mobility is also similarly computed. Finally, μ is updated for each fan-out, but to guarantee a legal schedule, it is only updated if the computed temporary mobility is smaller (lines 27-28). In other words, from the perspective of n_o, the minimal mobility among the values computed via its different fan-ins as n will be taken.

6.3.4 Buffer and Splitter Optimization

The scheduling-based legalization approach presented in the previous section allows us to find one or two legal mapped networks that are depth-optimal. In some scenarios, this may already be good enough, but it is still possible to further optimize the obtained mapped network to reduce its size. In this section, given a mapped network, we attempt to find a better schedule to minimize $|B|$ using a heuristic approach based on minimum-register retiming. Then, we present an optimization flow for technology legalization of AQFP circuits.

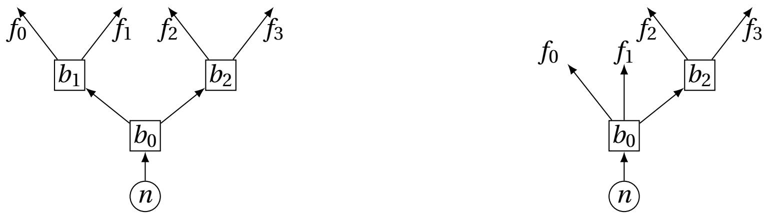
Retiming-based buffer and splitter optimization

The optimization of buffers and splitters in an AQFP circuit is reminiscent of retiming for the register minimization problem. *Minimum register retiming* is the problem of relocating the registers of a circuit in order to minimize their number while preserving the functionality. Retiming is formulated as a linear problem dual to the minimum-cost flow problem for which many polynomial algorithms exist [115].

In this section, we propose the AQFP B/S retiming algorithm, which minimizes buffers and splitters in an AQFP network, similar to how registers are minimized in minimum register retiming. Previous work applied a retiming-like optimization to AQFP logic [13, 35]. However, their approach does not perform global retiming but moves buffers locally from the output of splitters to the input. This optimization is subsumed by Algorithm 6.1 in the definition of irredundant mapped networks.

Minimizing the number of buffers can be seen as maximizing the sharing of buffers on multiple paths. Without accounting for fan-out-branching, e.g., assuming that buffers have an infinite splitting capability, the minimum number of buffers is achievable in polynomial time using a minimum register retiming algorithm considering each buffer as a register. Retiming preserves the path-balancing constraint since each path traverses the same number of registers before and after retiming. Previous works successfully applied this idea to the RSFQ technology family [88], but when the fan-out-branching constraint in AQFP comes into consideration, splitter relocation is conditional on respecting the splitting capacity. Hence, retiming is only a heuristic for AQFP B/S optimization instead of an optimal algorithm.

Example 6.3.11. *Figure 6.4a shows an example mapped sub-network under retiming, where $s_b = 3$ is assumed. This sub-network is redundant because b_1 and b_2 have out-degree $2 < s_b$ (Definition 6.3.5). Indeed, a mapped network can become redundant temporarily during retiming. Not all splitters can be retimed at the same time, and this example shows two such cases. First, b_0 cannot be retimed because its movement would increase the fan-out count of n to 2, violating the fan-out constraint of gates ($s_g = 1$). Second, only one of the splitters b_1 and b_2 can be selected for retiming since the movement of both of them would increase the fan-out count of*



(a) Before retiming

(b) After moving b_1

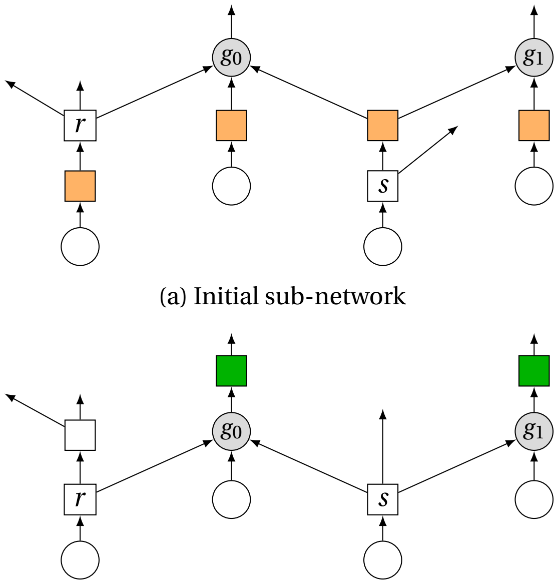
Figure 6.4: Example sub-network for retiming. ($s_b = 3$)

Algorithm 6.5: B/S retiming

```
   Input: Mapped network $N'$, Retiming direction $dir$
   Output: Optimized mapped network $N'$
1  while improvement do
2      select_retimeable_buffers($N'$)
3      set up retiming direction to $dir$
4      maximum_flow($N'$)
5      get_minimum_cut($N'$)
6      $N' \leftarrow$ move_retimed_buffers($N'$)
7  while improvement do
8      select_retimeable_buffers($N'$)
9      set up retiming direction to $\neg dir$
10     maximum_flow($N'$)
11     get_minimum_cut($N'$)
12     $N' \leftarrow$ move_retimed_buffers($N'$)
13 $N' \leftarrow$ reconstruct_fan-out_trees($N'$)
14 return $N'$
```

b_0 to 4, violating the fan-out constraint of buffers ($s_b = 3$). Also, fan-outs of splitters in the same fan-out tree originating from the same gate are exchangeable, and such exchanges may affect possible retiming optimizations. For example, instead of $FO(b_1) = \{f_0, f_1\}$, $FO(b_2) = \{f_2, f_3\}$ in Figure 6.4a, $FO(b_1) = \{f_0, f_2\}$, $FO(b_2) = \{f_1, f_3\}$ is also possible and may unlock more retiming on b_1 and b_2. Figure 6.4b shows the fan-out tree after the relocation of splitter b_1 to its transitive fan-out cone (not shown). ▲

The B/S retiming algorithm is shown in Algorithm 6.5, which takes a legal mapped network as input and outputs of an optimized mapped network. The retiming problem is formulated as a binary maximum-flow problem similar to [83], which separates flow computation into forward and backward directions. The input variable *dir* defines which direction to execute first. This parameter will be explained in the next section. Generally, *forward* is the preferred first direction if the circuit has an ALAP configuration since most of the registers would be placed closer to the PIs. *Backward* is instead a better first direction on an ASAP configuration. The algorithm performs two optimization loops in both directions until no more improvements can be made. A loop starts by selecting buffers to be retimed (lines 2 and 8), which are buffers that can be relocated without exceeding the splitting capacity of its fan-in node.



(a) Initial sub-network

(b) Optimized sub-network

Figure 6.5: Example of forward retiming. ($s_b = 3$)

In the case of mutually exclusive selections (i.e., two splitters cannot be retimed at the same time), one is picked randomly. Each selected buffer is a source and a sink of a unitary flow. In this step, filtering rules are applied to avoid selecting redundant elements such as more than one buffer in buffer chains. The filter rules help improve the run time since they reduce the number of elements for which an augmenting path algorithm starts. Next, the algorithm proceeds by selecting the retiming direction (lines 3 and 9), computing the binary maximum flow using the augmenting path algorithm (lines 4 and 10), getting the minimum cut (lines 5 and 11) and moving the selected buffers to the new position if there is a reduction (lines 6 and 12). Since retiming movements may create redundant fan-out trees, the algorithm terminates by reconstructing each fan-out tree irredundantly using Algorithm 6.1 (line 13).

Example 6.3.12. *An example of a forward retiming iteration is depicted in Figure 6.5. Figure 6.5a shows the initial sub-network, where* $s_b = 3$*. The algorithm selects the buffers and splitters in orange to perform retiming. The B/S elements* r *and* s *are not selected. Figure 6.5b shows the optimized sub-network after retiming. Two new buffers are inserted (in green). The number of buffers is reduced from* 6 *to* 5 *while maintaining the same path lengths.* ▲

Buffer and splitter optimization flow

We present a flow to minimize the number of buffers and splitters after an initial insertion. In this flow we employ: (i) the retiming-based buffer and splitter optimization algorithm in this section; (ii) the chunk movement algorithm in [105] (briefly described in Section 6.3.2); (iii) a deterministic circuit randomization function. The optimization flow is shown in Algorithm 6.6.

Algorithm 6.6: Buffer and splitter optimization

```
Input: Mapped network $N'_{\text{init}}$, Retiming direction $dir$
Output: Optimized mapped network $N'_{\text{opt}}$
$N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{init}}$, $dir$)            ▷ Run Algorithm 6.5
repeat
    $N'_{\text{opt}} \leftarrow N'_{\text{tmp}}$
    $N'_{\text{tmp}} \leftarrow$ chunked_movement($N'_{\text{opt}}$)            ▷ Run Algorithm in [105]
    $N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{tmp}}$, $dir$)            ▷ Run Algorithm 6.5
    $N'_{\text{tmp}} \leftarrow$ randomize($N'_{\text{tmp}}$)
until $|N'_{tmp}| \geq |N'_{opt}|$
return $N'_{opt}$
```

Algorithm 6.7: AQFP technology legalization flow (solves Problem 6.3.4)

```
Input: MIG network $N^*$
Output: Mapped network $N'$
$\mathscr{S}_{\text{ALAP}} \leftarrow$ ALAP($N^*$)            ▷ Run Algorithm 6.3
$\mathscr{S}_{\text{ASAP}} \leftarrow$ ASAP($N^*$, $\mathscr{S}_{\text{ALAP}}$)            ▷ Run Algorithm 6.4
$N'_{\text{ALAP}} \leftarrow$ insert_buffers($N^*$, $\mathscr{S}_{\text{ALAP}}$)            ▷ Run Algorithm 6.1
$N'_{\text{ASAP}} \leftarrow$ insert_buffers($N^*$, $\mathscr{S}_{\text{ASAP}}$)            ▷ Run Algorithm 6.1
$dir \leftarrow$ forward
$N'_{\text{ALAP}} \leftarrow$ optimize($N'_{\text{ALAP}}$, $dir$)            ▷ Run Algorithm 6.6
$dir \leftarrow$ backward
$N'_{\text{ASAP}} \leftarrow$ optimize($N'_{\text{ASAP}}$, $dir$)            ▷ Run Algorithm 6.6
if $|N'_{ALAP}| < |N'_{ASAP}|$ then
    return $N'_{ALAP}$
else
    return $N'_{ASAP}$
```

Algorithm 6.6 combines retiming and *chunk movement* [105] to achieve better results than the individual algorithms. Additionally, we use a deterministic randomization function to select a different topological order and to rearrange the fan-out of nodes in the network. As the creation of a splitter tree is also influenced by the fan-out processing order (Algorithm 6.1), this method may lead to different splitter trees, thereby enabling further optimizations. Generally, the deterministic randomization function unlocks an additional 3.6% reduction in the number of buffers and splitters.

6.3.5 AQFP Technology Mapping

We present a flow for AQFP technology mapping consisting of buffer and splitter insertion followed by B/S optimization. In this flow, we employ the depth-optimal buffer and splitter insertion algorithms described in Section 6.3.3 and the buffer and splitter optimization approach in Algorithm 6.6. The optimization flow is shown in Algorithm 6.7.

In Algorithm 6.7, two initial scheduling, ALAP and ASAP are obtained with the depth-optimal scheduling algorithms and result in two mapped networks by inserting buffers irredundantly. Then, the two mapped networks are optimized independently using the portfolio optimization flow. Finally, the better one with a smaller size is adopted. This method performs better than choosing the mapped network with the best size after depth-optimal scheduling and then performing optimization. Specifically, in numerous cases, optimizing the mapped network with the worse result after scheduling also leads to a better size. Since the B/S optimization is run twice, the adopted method takes almost double the time compared to the other mentioned.

6.3.6 Experimental Results

All of the algorithms and flows presented in this paper are implemented in the open-source C++ logic synthesis library *mockturtle*[1] [183]. In this section, we present experimental results of our methods solving Problem 6.3.4 alone. First, we present the experimental results against the state of the art when this work was published in [195]. Then, we presents results against the current state of the art. Finally, we demonstrate the scalability of the proposed B/S insertion algorithm on large benchmarks. To be consistent with previous works that we compare to, we use $s_b = 4$ for the splitting capacity of buffers. All results and baselines are verified and published[2] for third-party verification.

Results for the AQFP technology mapping problem, (Problem 6.3.2), obtained by combining MIG-based optimization with the technology legalization flow in Algorithm 6.7, are available in [110] and show a reduction in area, delay, and energy-delay product (EDP) by 36%, 12%, and 44%, respectively, compared to the state-of-the-art AQFP synthesis flow [61].

Depth-optimal buffer and splitter insertion and optimization

First, we compare the performance of our B/S insertion and optimization flow in Algorithm 6.7 against the state-of-the-art (SoTA) algorithm on solving the same problem using scheduling algorithms that are not depth optimal [105]. The method in [105] first generates non-depth-optimal ASAP and ALAP schedules, selects the best one, and then further optimizes the number of B/S elements using the chunk movement algorithm. To fairly compare against [105], we modify Algorithm 6.7 to select the best mapped network after depth-optimal ALAP (Algorithm 6.3) and ASAP (Algorithm 6.4) scheduling for carrying the optimization. For our baseline, we use all the benchmarks used in the first work on AQFP B/S insertion [35].

The results are shown in Table 6.2. The number of gates ($|G^*|$) and the depth ($d(N^*)$) of the initial MIGs, as well as the number of buffers ($|B|$), the JJ count (#JJs) and the depth ($d(N')$) of the mapped networks are listed. For our approach, we also include the results right after B/S insertion, indicated with the term "Ins.". Our approach (the simplified Algorithm 6.7)

[1] https://github.com/lsils/mockturtle
[2] https://github.com/lsils/SCE-benchmarks

Table 6.2: Our technology legalization results comparing to the non-depth-optimal buffer and splitter insertion.

Benchmark	**Initial MIG N^***		**Non-depth-optimal SoTA [105]**				**Ours (simplified Algorithm 6.7)**					
	$\lvert G^*\rvert$	$d(N^*)$	$\lvert B\rvert$	#JJs	$d(N')$	Time (s)	$\lvert B\rvert$ Ins.	$d(N')$	Time Ins. (s)	$\lvert B\rvert$	#JJs	Total Time (s)
adder1	7	4	16	74	8	0.00	16	8	0.00	16	74	0.00
adder8	77	17	371	1204	33	0.00	374	33	0.00	372	1206	0.01
mult8	439	35	1869	6372	71	0.00	1824	70	0.00	1688	6010	0.06
counter16	29	9	65	304	17	0.00	70	17	0.00	65	304	0.00
counter32	82	13	155	802	23	0.00	170	23	0.00	154	800	0.00
counter64	195	17	352	1874	30	0.00	377	30	0.00	347	1864	0.01
counter128	428	22	760	4088	38	0.00	807	38	0.00	747	4062	0.02
c17	6	3	12	60	5	0.00	12	5	0.00	12	60	0.00
c432	121	26	874	2474	38	0.00	862	37	0.00	839	2404	0.02
c499	387	18	1275	4872	31	0.00	1198	29	0.00	1173	4668	0.02
c880	306	27	1703	5242	41	0.01	1691	40	0.00	1511	4858	0.10
c1355	389	18	1290	4914	31	0.00	1206	29	0.00	1184	4702	0.03
c1908	289	21	1298	4330	35	0.01	1318	34	0.00	1236	4206	0.04
c2670	368	21	2132	6472	30	0.02	2080	28	0.00	1932	6072	0.09
c3540	794	32	2266	9296	55	0.10	2678	52	0.00	1972	8708	0.16
c5315	1302	26	6026	19864	42	0.12	7430	40	0.01	5646	19104	0.45
c6288	1870	89	9893	31006	180	0.12	14119	179	0.01	9009	29238	0.23
c7552	1394	33	8759	25882	66	0.12	10149	56	0.01	7505	23374	1.01
sorter32	480	15	480	3840	30	0.00	480	30	0.00	480	3840	0.01
sorter48	880	20	880	7040	35	0.00	960	35	0.00	880	7040	0.02
alu32	1513	100	14655	38388	171	0.84	15207	169	0.01	13837	36752	0.82
Total			55131	178398	1010	1.34	61796	**982**	0.04	**50605**	**169346**	3.08

performs significantly better in almost every benchmark reducing the number of B/S elements up to 14%. Additionally, the depth is also significantly reduced, up to 15% in benchmark *c7552*. After depth-optimal B/S insertion, the B/S retiming algorithm is responsible of the 84% of the total area reduction on average. Our results improve even more using the standard flow in Algorithm 6.7. Those results are reported in the next experiment in Table 6.3.

Technology legalization and buffer optimization

First, we compare the performance of our B/S insertion and optimization flow in Algorithm 6.7 against the current state-of-the-art (SoTA) on solving the same problem [62]. For the sake of completeness, we list all of the benchmarks used in the first work on AQFP B/S insertion [35], but the total results are computed only with the benchmarks presented in [62].

The results are shown in Table 6.3. The number of gates ($|G^*|$) and the depth ($d(N^*)$) of the initial MIGs, as well as the number of buffers ($|B|$), the JJ count (#JJs) and the depth ($d(N')$) of the mapped networks are listed. Moreover, the runtime (Time) used by our flow is presented. The run time data is not available in [62] and results are not reproducible since the implementation is not openly available. In the last column, we list the known global optimum results obtained by ILP solving [125] to have an idea of how far the heuristics are from optimal. Some of the numbers are only an upper bound because the ILP formulation used by [125] could not solve these problems within reasonable time, and some of the benchmarks are too big for the ILP solver to return any partial result.

From Table 6.3, we can notice that the heuristic methods achieve optimum for the smaller

Table 6.3: Technology legalization results comparing to the state-of-the-art and global optimum.

Benchmark	**Initial MIG N^***		**SoTA [62]**			**Ours (Algorithm 6.7)**				**ILP [125]**		
	$\lvert G^*\rvert$	$d(N^*)$	$\lvert B\rvert$	#JJs	$d(N')$	$\lvert B\rvert$	#JJs	$d(N')$	Time (s)	$\lvert B\rvert$	#JJs	$d(N')$
adder1	7	4	-	-	-	16	74	8	0.00	16	74	8
adder8	77	17	-	-	-	371	1204	33	0.01	371	1204	33
mult8	439	35	1681	5996	70	1690	6014	70	0.18	≤1724	≤6082	≤70
counter16	29	9	66	306	17	65	304	17	0.00	65	304	17
counter32	82	13	156	804	23	154	800	23	0.01	154	800	23
counter64	195	17	351	1872	30	347	1864	30	0.02	347	1864	30
counter128	428	22	755	4078	38	747	4062	38	0.07	747	4062	38
c17	6	3	-	-	-	12	60	5	0.00	12	60	5
c432	121	26	829	2384	37	839	2404	37	0.02	829	2384	37
c499	387	18	1173	4668	29	1173	4668	29	0.09	1173	4668	29
c880	306	27	1536	4908	40	1511	4858	40	0.15	-	-	-
c1355	389	18	1186	4706	29	1184	4702	29	0.06	1178	4690	29
c1908	289	21	1253	4240	34	1234	4202	34	0.09	1232	4198	34
c2670	368	21	1869	5954	28	1912	6032	28	0.32	≤1804	≤5816	≤28
c3540	794	32	1963	8690	52	1943	8650	52	0.81	≤1926	≤8516	≤52
c5315	1302	26	5505	18942	40	5640	19092	40	2.06	≤6260	≤20332	≤42
c6288	1870	89	8832	28884	179	8647	28514	179	2.56	-	-	-
c7552	1394	33	6768	21908	58	7437	23238	56	4.20	-	-	-
sorter32	480	15	-	-	-	480	3840	30	0.06	480	3840	30
sorter48	880	20	-	-	-	880	7040	35	0.20	880	7040	35
alu32	1513	100	13976	37030	169	13836	36750	169	2.74	-	-	-
Total*			47899	155370	873	48359	156154	871	13.38			

benchmarks and are fairly close to optimum for most of the benchmarks. While our flow obtains slightly worse results in average size than SoTA, the difference is very small (0.96% in number of buffers and 0.5% in JJ count). Thanks to the depth-optimal scheduling, we obtain a better depth in one benchmark (c7552). Most importantly, these results are obtained in a very short run time. Thus, our flow can be used in design-space exploration (DSE), where legalization is called extensively, such that large improvements can be achieved. While the design of a DSE engine for AQFP circuits is not addressed in this thesis, results using the techniques of this chapter have been published in [110]. These results demonstrate a reduction in area, delay, and energy-delay product (EDP) by 36%, 12%, and 44%, respectively, compared to the state-of-the-art AQFP synthesis flow [61].

It is also worth mentioning that, out of the 21 benchmark circuits, the legalization and optimization result starting from an ASAP scheduling is selected (i.e., better than the one from an ALAP scheduling) in 16 benchmarks. We can see that ASAP may provide better quality in more cases, but this is not definitive. Thus, trying both starting points, as in Algorithm 6.7, helps achieve better results when the runtime budget is sufficient.

Scalable technology legalization

To demonstrate the scalability of our AQFP legalization approach, we use the largest 10 benchmarks in the EPFL benchmark suite [3] for experiment, which are 10 to 100 times larger in size compared to the benchmarks generally used in previous works on AQFP logic synthesis.

Table 6.4: Technology legalization results on the 10 largest EPFL benchmarks.

Benchmark	**Initial MIG N^***		**Non-d.-opt. legal.+opt. [105]**			**D.-opt. legal. (Alg. 6.3, 6.4 + Alg. 6.1)**			**D.-opt. legal.+opt. (Alg. 6.7)**		
	$\|G^*\|$	$d(N^*)$	$\|B\|$	$d(N')$	Time (s)	$\|B\|$	$d(N')$	Time (s)	$\|B\|$	$d(N')$	Time (s)
div	57300	2217	2084772	4918	271.71	1881255	4371	0.87	-	4371	>300
hyp	136109	8762	-	17910	>300	9035578	17246	2.78	-	17246	>300
log2	24456	200	98047	414	194.92	129547	379	0.10	86705	379	64.18
multiplier	19710	133	79651	286	13.21	102005	264	0.08	63414	264	43.50
sin	4303	110	17470	225	5.67	18905	188	0.01	14886	188	4.12
sqrt	23238	3366	1751742	8191	5.64	1791005	6628	0.49	1343705	6628	284.10
square	12180	126	60552	256	42.71	89516	251	0.03	63630	251	18.30
arbiter	7000	59	31011	65	5.80	27566	63	0.01	25721	63	1.28
mem_ctrl	42758	73	305689	182	87.86	216927	114	0.27	215202	114	10.55
voter	7860	47	18044	99	5.43	19263	86	0.01	15736	86	0.92

The MIGs are obtained using delay-oriented graph mapping [202] (in Section 5.2). In Table 6.4, we compare our results obtained using a simple depth-optimal legalization flow (the best between Algorithm 6.3 and Algorithm 6.4 followed by Algorithm 6.1, column "D.-opt. legal.") as well as depth-optimal legalization with further optimization (Algorithm 6.7, column "D.-opt. legal.+opt.") against results of non-depth-optimal legalization with optimization presented in [105] (column "Non.-d.-opt. legal.+opt."). A timeout limit of 300 seconds is enforced. This experiment demonstrates that simple legalization without optimization is very fast. Thus, such a flow can still be used in design-space exploration even when benchmarks are large. Comparing the mapped network depths, the proposed depth-optimal scheduling reduces the depth by about 9% on average.

6.4 Logic Synthesis for SFQ Circuits

Rapid Single-Flux Quantum (RSFQ) is the most mature superconducting logic family [119]. Multiple variants of RSFQ, such as the eSFQ [149], are commonly grouped under the term SFQ. Unlike CMOS, SFQ circuits encode the logic "true" in a small voltage pulse and the logic "false" in a pulse absence. Consequently, most SFQ logic gates are clocked to discern between these two states. These gates function as latches, with a clock input and one or more data inputs. When a pulse arrives at a data input, it alters the internal state of the gate. Subsequently, a clock pulse resets the gate to its initial state and may generate an output pulse based on the internal state. As SFQ circuits rely on the clock signal, they necessitate pipelining at the gate level. To ensure correct data propagation, i.e., data at each gate must be present at specific time-frames for correct computations, SFQ circuits require delay registers (DFFs) in the combinational paths so that every path from primary inputs to logic gates traverses the same number of clocked gates. This constraint is referred to as *path balancing*. Furthermore, due to the quantized nature of SFQ pulses, most RSFQ primitives have a maximum driving capacity of one gate. Consequently, a special cell called *splitter* is necessary to drive multiple fan-outs.

In this work, we present an innovative synthesis flow to carry out the optimization and mapping for SFQ technology. In particular, we focus on delay optimization which is key to synthesizing efficient SFQ circuits. Technology-independent optimization is carried over the *xor-and graph* (XAG) (or *xor-and-inverter graph* (XAIG)) since it closely abstracts the logic primitives of SFQ. In fact, both 2-input XOR and 2-input AND (OR) gates have similar delay and area costs. Moreover, XAGs are more compact than the commonly used *and-inverter graph* (AIGs) offering better opportunities to restructure logic through additional rewriting rules and Boolean methods. We present several techniques namely, mapping, re-mapping, algebraic rewriting, *exclusive sums-of-products* (ESOP) balancing, Boolean rewriting, and resubstitution. Technology mapping is performed directly on the XAG without previous decomposition into an AIG. We describe post-mapping methods to satisfy the path-balancing and fan-out constraints. Finally, we use minimum-register retiming to optimally minimize the number of inserted balancing DFFs.

In the experiments, we show that our synthesis flow efficiently reduces the delay without causing an area explosion. We compare against the state-of-the-art showing 43% and 34% reduction on average in area and delay, respectively.

6.4.1 Related Works

In this section, we introduce existing works on SFQ circuits focusing on reducing the path-balancing cost. First, we present related work in buffer and splitter insertion. Then, we describe existing technology-independent synthesis and technology mapping methods for SFQ logic. Finally, we describe other orthogonal techniques based on clocking schemes and SFQ cell design.

Buffer and splitter insertion and optimization

Adiabatic Quantum-Flux Parametron (AQFP) [218] is a sibling superconducting technology of SFQ and has similar path-balancing and fan-out-branching constraints, thus also requiring buffer and splitter insertion [35, 80, 105, 110, 195] (Section 6.3). However, a key difference between the two technologies makes the problem computationally distinct for them: in AQFP, splitters are clocked and considered in path balancing, so fan-out branching and path balancing have to be addressed together; whereas SFQ splitters are not clocked, thus the two constraints can be considered separately, simplifying the mapping problem.

The first complete buffer and splitter insertion method for SFQ was proposed in [88]. The method performs classical technology mapping into a SFQ cell library. Then, balancing DFF are inserted heuristically for each path. Note that this process is trivial since splitters are not inserted at this stage. Next, the number of DFFs is minimized using minimum-register retiming [115]. Contrarily to AQFP circuits, since splitters are asynchronous, the fan-out change of a gate does not limit relocation of registers. Consequently, given a network, the number of balancing DFFs can be optimally minimized. Finally, also splitter trees are inserted. In this work, we follow a similar strategy.

The path balancing problem has also been addressed on sequential SFQ circuits [160]. The authors formulate the problem as a level assignment problem to a cyclic directed graph. Then, the circuit is path balanced by equalizing the paths as for combinational logic.

SFQ logic synthesis

Due to path-balancing DFFs and splitters, the area of SFQ circuits can grow prohibitively large. Consequently, multiple research works address minimizing the path-balancing cost before and during technology mapping.

In [158], algebraic factoring and Boolean rewriting have been extended to account for the path-balancing cost during logic optimization on an AIG representation of the logic. A similar extension was proposed in [159] for technology mapping. This approach replaces the area heuristics of optimization algorithms with a heuristic for imbalances, based on dynamic programming, ensuring local-optimal balancing for logic trees. However, typical designs are DAGs, and in this scenario, this path-balancing cost tends to overestimate the number of DFFs, leading to suboptimal results. Additionally, when integrating path-balancing costs in technology-independent algorithms [158] also real critical paths are not known. For instance, since inverters are not represented, their contribution to the delay is not considered during the optimization leading to incorrect balancing.

Recently, Synopsys developed a RTL-to-GDSII flow for SFQ technology [2, 148]. Logic synthesis is performed on the *and-or-inverter graphs* (AOIG) using algebraic rules and Boolean rewriting. Additionally, the work underlines the importance of depth optimization as a proxy for imbalance reduction.

All previous work on synthesis for SFQ addressed the logic synthesis problem using *and-inverter graphs* (AIG) or *and-or-inverter graphs* (AOIG). Instead, in this work we formulate the logic optimization using *xor-and graph* (XAG), to better abstract the logic primitives of the SFQ technology and explore more optimization opportunities through additional rewriting rules and Boolean methods.

Clocking schemes and SFQ cell design

To reduce the area and power overhead due to path-balancing DFFs, several works re-design the clock network to SFQ gates. In [157], the authors design a dual-clock network composed of a fast clock for logic and a slow clock for sequential registers. Sequential registers are implemented using non-destructive readout (NDRO) registers, such that the same data inputs are propagated through the logic for multiple iterations. This approach trades the circuit's throughput for a reduction in the number of path-balancing DFFs. If the ratio between the frequency of the slow and fast clock matches the logic depth of the circuit, path-balancing DFFs are no longer required. However, this technique necessitates the use of relatively expensive NDRO DFFs, the duplication of the clock distribution network, and the generation of coordinated clocks. A dual clocking scheme has also been utilized in an RTL-to-GDSII flow, demonstrating a significant area improvement [148]. An extension of this method considers multi-phase clocking, which eliminates the need for NDRO DFFs at the cost of additional clock networks and a few balancing DFFs [19, 118]. Another interesting scheme is the clock-follow-data clocking [64], which asynchronously delays the clock lines so that a single clock pulse carries the data through the whole circuit. However, designing and verifying such a clock network is challenging.

Other approaches focus on reducing path-balancing DFFs by minimizing the number of clocked elements in a circuit. In dynamic SFQ (DSFQ) logic [173], gates reset to the initial state after the specified period of time, removing the need for a clock. The design of DSFQ circuits is therefore similar to CMOS circuits where large combinational blocks can be synchronized using relatively few synchronous elements [96]. A similar approach based on clockless logic gates is proposed in [89]. These approaches offer advantages over conventional SFQ, including smaller area, lower clock network complexity, and simpler path balancing. However, evaluating the interaction among input skew tolerance, clock frequency, bias margins [96], and timing poses a significant challenge. Other techniques explore using confluence buffers and splitters (asynchronous gates) to perform logic computation without a clock signal. In [17, 92], the authors propose special compound gates that internally use a mix of asynchronous and synchronous computations. Specifically, in [16], the authors present a cell library implementing any 4-input function in one clock cycle. Using this library significantly reduces the logic depth of SFQ circuits and, consequently, the number of imbalances.

6.4.2 XAG-based Logic Synthesis

This section presents our first contribution on logic synthesis for SFQ circuits. Technology libraries for SFQ are typically simple and implement basic functions. Notably, logic cells within these libraries are all clocked and have similar areas. Interestingly, XOR cells demonstrate a similar level of efficiency as AND cells. Based on this observation, we center our technology-independent synthesis flow for SFQ on *xor-and graphs* (XAGs). XAGs have multiple advantages over the commonly used *and-inverter graphs* (AIGs). XAGs are more compact since they contain one additional primitive, which is implemented using three 2-input ANDs in AIGs. Consequently, circuits represented by XAGs tend to be smaller and shallower. Moreover, XAGs offer more opportunities to restructure logic through additional rewriting rules and Boolean methods.

Minimizing the logic depth is essential for a fast and compact SFQ circuit. First, the logic depth directly relates to the circuit latency. Latency is generally dominated by the number of clock cycles needed to realize a function. Since each SFQ gate is clocked, the latency can be approximated as the logic depth of the circuit. Second, delay optimization helps minimize the number of necessary balancing DFFs. Intuitively, longer critical paths require more DFF elements due to longer paths to balance.

In this section, we present several techniques aimed at optimizing delay and area over the XAGs, thereby enhancing the overall efficiency and performance of SFQ circuits. First, we present a method to obtain an optimized XAG representation. Then, we present three efficient algorithms, one algebraic and two Boolean, to optimize depth over the primitives AND and XOR. Finally, we describe two methods to reduce the area.

Derive the initial XAG

Given a generic Boolean network, the first problem to address is how to obtain a good initial *xor-and graph* (XAG) representation. To achieve it, we employ the state-of-the-art graph mapping strategy based on the versatile mapper in Section 5.2 (or [202]). It consists of a delay-oriented mapping algorithm that leverages a database of size-optimum XAG structures as a library. The XAG structures are derived using SAT-based exact synthesis over the 4-input NPN classes [69]. Multiple minimum structures with different pin-to-pin delays are stored for each class. The utilization of this method yields significantly improved results compared to merely identifying XOR gates, as it incorporates Boolean optimization and rewriting by utilizing locally optimum structures.

Depth optimization with XAG algebraic rewriting

Boolean algebra defines primitive transformations and properties, referred to as a set Ω. In the context of delay optimization, a subset of these fundamental properties plays an important role. In particular, for XAGs, we can identify the following algebraic rules contained in Ω:

$$\Omega.A_{AND}: \ a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{6.4}$$

$$\Omega.A_{XOR}: \ a \oplus (b \oplus c) = (a \oplus b) \oplus c \tag{6.5}$$

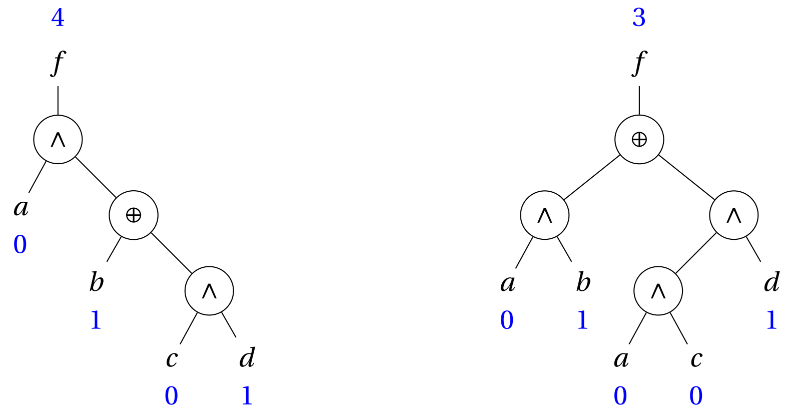
$$\Omega.D_{AND-OR}: \ a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \tag{6.6}$$

$$\Omega.D_{AND-XOR}: \ a \wedge (b \oplus c) = (a \wedge b) \oplus (a \wedge c) \qquad , \tag{6.7}$$

where 6.4 is the associative property of AND, 6.5 is the associative property of XOR, 6.6 is the distributive property of AND over OR, and 6.7 is the distributive property of AND over XOR. Note that the associative property of OR is contained in 6.4. In XAGs, transformations 6.4 and 6.5 can push critical signals forward towards the POs, when the associative property holds. Transformations 6.6 and 6.7, on the other hand, are less directly applicable. Experimental results have shown that these latter transformations are rarely useful, especially in optimized circuits. Nevertheless, two powerful transformations can be derived from 6.4, 6.5, 6.6, and 6.7 when considering logic cones of three operators and the interplay between AND-OR and AND-XOR. We refer to the extended set as Ψ:

$$\Psi.D_{AND-OR}: \qquad a \wedge (b \vee (c \wedge d)) = (a \wedge b) \vee ((a \wedge c) \wedge d) \tag{6.8}$$

$$\Psi.D_{AND-XOR}: \qquad a \wedge (b \oplus (c \wedge d)) = (a \wedge b) \oplus ((a \wedge c) \wedge d). \tag{6.9}$$



(a) Before rewrite

(b) After rewrite

Figure 6.6: Rewriting with AND-XOR distributivity in Ψ (6.9).

These two rules define the distribution of AND over OR (6.8) and AND over XOR (6.9). Note that in an XAG, the OR is represented as an AND with inverted inputs and output according to De Morgan's laws.

Example 6.4.1. *Figure 6.6 shows an example of how the derived AND-XOR distributive rule in Equation 6.9 can reduce the delay of a circuit. The transformation is applied to the XAG in Figure 6.6a. The critical signal d binds the arrival time at the output. The transformation pushes signal a forward by distributing signal a through the XOR operation and applying AND-associativity to signal d. The result in Figure 6.6b shows a delay reduction by one, from* 4 *to* 3*, at the cost of one additional AND operation.* ▲

The algebraic rules are graph-based; hence, they are extracted structurally from an XAG. We employ rules to detect properties and apply transformations based on input arrival times, inverted edges, and gate type. Node duplication is also enabled in the case of nodes with multiple fan-outs.

Algorithm 6.8 shows how the algebraic rewriting rules are carried out on an XAG. The algorithm applies rewriting rules on critical paths reachable from one PO at a time. The critical paths are updated after every successful move. The algorithm first tries to apply moves based on associativity, which incur less area increase than distributive rules. When associativity fails, distributive rules are attempted.

Depth optimization with ESOP balancing

Exclusive-Sums-of-Products (ESOPs) are a two-level representation of Boolean functions composed of an exclusive OR of product terms. Differently from *Sums-of-Products* (SOPs), which are based on the primitives AND and OR, ESOPs utilize AND and XORs. In the SFQ context, ESOPs are interesting since XORs are efficient gates. Moreover, for many Boolean functions, the

Algorithm 6.8: XAG depth algebraic rewriting

```
Input: XAG $N$
Output: Optimized XAG $N$
foreach edge $p \in PO(N)$ do
    if in_critical_path(N, p) then
        foreach node $n \in$ critical_path(N, p) do
            $\Omega.A_{AND}(N, n)$
            $\Omega.A_{XOR}(N, n)$
            $\Psi.D_{AND-OR}(N, n)$
            $\Psi.D_{AND-XOR}(N, n)$
            update_timing($N$, $n$)
return $N$
```

number of cubes in minimal ESOPs is lower than the number of cubes in minimal SOPs [175].

SOP-balancing [143] is a scalable technique to optimize for delay. It consists of extracting small cones of logic, generally up to ten variables, converting them into SOPs, and applying AND balancing to each term and the sum. In practice, each term is considered as a multiple-input AND and it is decomposed into 2-input ANDs while minimizing the arrival time of the term root. The sum is decomposed similarly into 2-input ORs.

In this work, we employ ESOP-balancing to achieve delay optimization over the primitives AND and XOR. Algorithm 6.9 presents a high-level view of ESOP balancing. Given a large XAG, *cut enumeration* [47] is used to break the circuit into multiple logic cones. For each cone, an initial ESOP cover is extracted using the algorithm described in [54] that computes an exact minimum Pseudo-Kronecker expression (PKRMs). PKRMs are a specific subset of ESOP expressions that can be generated using only positive or negative Davio expansion and Shannon expansion. In our implementation, we extract PKRMs directly from truth tables without involving BDDs as in the original formulation. Then, the initial cover is minimized to obtain a compact ESOP using the *EXORCISM* family of heuristics [147]. To reduce the run time, ESOPs are cached for later reuse using a hash table. Next, AND- and XOR-balancing is performed to generate a decomposition tree that minimizes the arrival time at root n. The arrival time of a leaf is defined as the arrival time of the best cut computed at the leaf. Algorithm 6.9 works similarly to a technology mapper [143, 202]. In the first section, from line 2 to line 10, the algorithm computes the cuts and selects one having the best arrival time for each node. At line 11, the area is recovered by selecting lower-cost decompositions in paths where the slack is positive. Area recovery works similarly to [202]. This step is crucial to minimize the area increase derived from balancing due to logic duplication. Finally, at line 12, a cover is extracted and converted into a balanced and delay-optimized XAG.

Algorithm 6.9: ESOP balancing

```
Input: XAG $N$
Output: Optimized XAG $M$
$C \leftarrow \emptyset$
foreach node $n \in N$ in topological order do
    $C(n) \leftarrow$ compute_cuts($N$, $C$, $n$)
    foreach cut $c \in C(n)$ do
        $tt \leftarrow$ compute_truth_table($N$, $c$)
        $esop \leftarrow$ compute_exact_pkrm($tt$)
        exorcism($esop$)
        compute_balancing_cost($N$, $C$, $n$, $esop$)
        if better delay than best cut then
            set_best_cut($C(n)$, $c$)
area_recovery($N$, $C$)
$M \leftarrow$ extract_cover($N$, $C$)
return $M$
```

Depth optimization with remapping

XAG-remapping [202] is a rewriting technique that maps an XAG network to obtain a new XAG implementation. A library of pre-computed XAG structures is used to optimize the logic. This method is equivalent to XAG mapping but with the difference that the input and output data structures are the same. In the synthesis flow, remapping minimizes the delay first and then recovers the area constrained by the found delay.

Area recovery

In area recovery, Boolean *rewriting* and *resubstitution* have been extended to work for XAG optimization.

DAG-aware rewriting [137] is a fast greedy algorithm that aims at minimizing the size of a logic network by iteratively replacing sub-graphs rooted in a node with smaller pre-computed structures while preserving the functionality, and (possibly) the delay, at the root node. A database of pre-computed structures covers all the 4-variable functions classified into the NPN equivalence classes for compactness [23]. In our implementation, pre-computed structures are the same as those used for previous mapping tasks. Differently from remapping, this algorithm is DAG-aware, i.e., it can reuse nodes that are already present in the network. Hence, it is more suited for area optimization. Our implementation constrains transformations to not increase the circuit depth. Thus, the required times are used to filter transformations. We use the implementation presented in Algorithm 5.4 (in Section 5.3 or in [196]) to perform XAG rewriting.

Resubstitution (re)expresses the function of a node using other nodes, called *divisors*, that are already present in the network. The transformation is accepted if the new implementation of a node is better in size compared to the current implementation of the node in terms of its immediate fan-ins. This approach generalizes to *k-resubstitution*, which adds k new nodes and removes at least $k+1$ nodes. In XAG-resub [9], added gates are 2-input ANDs and XORs with optional inverters at the inputs/outputs.

6.4.3 Technology Mapping

After technology-independent optimization, technology mapping translates the optimized XAG in terms of the connection of cells from an SFQ cell library. This process involves 3 steps: mapping, balancing DFF insertion, and splitter insertion.

Technology mapping to the SFQ cell library

In our approach, we adopt a direct mapping strategy that starts from *xor-and graphs* (XAGs) as the subject graph, enabling us to efficiently map into the SFQ cell library. To achieve reduced latency and area, the mapper is configured for minimal delay, focusing on optimizing performance. To further enhance the quality of the mapping and minimize delays, we introduce a pre-computed library of *supergates* [37]. These supergates are single-output networks constructed from a few library cells, treated as a single complex cell. The use of supergates provides the distinct advantage of mitigating the structural bias of the mapping algorithm, which often heavily relies on the initial subject graph structure. The supergates are generated recursively in multiple rounds using an enumeration process, ensuring a thorough exploration of possibilities.

Previous work in SFQ mapping introduced a technology mapper called PBMap [159], which incorporates an approximate path-balancing cost into the area cost function. The approach defines this cost based on dynamic programming, ensuring local-optimal balancing for logic trees. However, when dealing with optimized logic in a DAG format, where nodes may have multiple fan-outs, using path-balancing cost in technology mapping presents three key disadvantages.

First, PBMap overlooks area costs and relies solely on path-balancing costs that only work for trees of logic. In contrast, DAG-mapping approaches and heuristics have been demonstrated to be superior in comparison to minimizing delay and area [98]. Second, the heuristic used in PBMap treats the path-balancing cost for each cell as independent, disregarding the potential sharing of DFFs among cells connected to the same node. Consequently, PBMap overestimates the penalty of imbalanced solutions. Additionally, the exact DFF sharing information remains unknown until the entire network is mapped, making it challenging for their dynamic programming approach to efficiently capture this complexity. Moreover, the number of padding DFFs can be further optimized in a post-processing phase by moving DFFs upwards or backward through logic. Last, the simplicity of SFQ libraries enables technology mappers to already map logic trees with optimal solutions by prioritizing local delay first and area second, without the need to incorporate additional balancing costs. It is worth noting that this proposition may not hold for libraries containing multiple-input cells (with more than 2 inputs). However, these complex cells lack efficient implementations in SFQ. Experimental results confirmed these claims, indicating that delay-oriented mapping yields better area results on average compared to the cost function in PBMap. Furthermore, attempts to use the balancing cost as a tie-breaker during cell selection heuristics have not demonstrated any significant advantage.

Similar considerations apply to the integration of the path-balancing cost in technology-independent algorithms [158], where also true critical paths are not known. For instance, since inverters are not represented, their contribution to the delay is not considered during the optimization leading to potentially incorrect balancing.

Technology legalization: path balancing

After technology mapping, our approach inserts padding DFFs to fulfill the balancing constraint of the circuit for internal nodes and POs. The DFFs are inserted utilizing ASAP scheduling, ensuring that the arrival times at each cell's input are synchronized (balancing constraint), while maintaining a constant delay at their outputs (ASAP balancing policy). To optimize the area, our method shares DFFs among nodes connected to the same input node at the same clock level. This algorithm guarantees an optimal DFF insertion for the ASAP schedule, and it operates linearly with respect to the number of gates in the circuit.

After initial insertion, balancing DFFs are minimized using the minimum-register retiming algorithm in [83, 195]. This algorithm iteratively pushes DFFs backward toward the PIs in order to globally minimize their number.

Technology legalization: splitter insertion

As the final stage of technology mapping, we introduce splitter cells to address the fan-out constraint. Splitters are inserted as balanced trees, considering that logic is inherently balanced. Given that splitters in SFQ have a driving capacity of two gates, their number for an arbitrary gate n is equivalent to $|FO(n)| - 1$, where $FO(n)$ represents the fan-out of gate n.

6.4.4 Experimental Results

The methods and the synthesis flow have been implemented in C++17 in the open-source logic synthesis framework *Mockturtle* [183]. For our experiments, we use the same benchmarks of the state-of-the-art work in [159] provided as *and-inverter graphs* (AIGs). The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified to be functionally equivalent to the original circuit and to fulfill the path-balancing and fan-out constraints.

Algorithm 6.10: Synthesis flow

```
Input: Boolean network $N$, Iterations $i$, Library $L$
Output: SFQ circuit $M$
xag ← map_to_xag($N$)
fast_area_opt(xag)
for $i$ iterations do
    xag_depth_algebraic_rw(xag)
    xag_resub(xag)
    xag ← map_to_xag(xag)
    xag ← esop_balancing(xag)
    xag_boolean_rw(xag)
$M$ ← map(xag, $L$)
path_balance($M$)
min_area_retime($M$)
insert_splitters($M$)
return $M$
```

We first propose a simple but effective synthesis flow for SFQ that includes the XAG methods in Section 6.4.2 and the technology mapping approach in Section 6.4.3. Then, we present the experimental results comparing our flow to the state of the art.

SFQ synthesis flow

Algorithm 6.10 shows the synthesis flow used in our experiments. The initial point is a Boolean network which is simply an *and-inverter graph* (AIG) in our experiments. First, the AIG is mapped to an XAG and some fast and light area optimizations are performed using Boolean rewriting and resubstitution. The objective is to get a compact starting point by removing logic redundancy. It is crucial to not over-optimize the area at this step, and transformations are accepted only if there is a significant improvement and zero delay increase. Then, the core optimization starts using the algorithms in Section 6.4.2. An iteration alternates delay optimization with area recovery. On the one hand, the more iterations the better the delay that can be obtained in the final SFQ circuits. On the other hand, over-optimization leads to area increase. In our experiments, we perform one iteration. After, technology-independent optimization, the SFQ circuit is mapped and optimized using the methods in Section 6.4.3. To maintain tractability and efficiency, we impose constraints on the supergates. Specifically, we generate 6294 supergates limiting the number of inputs to 5 and the number of cell levels to 3.

The most computationally intensive algorithm of the synthesis flow is the minimization of path-balancing DFFs using retiming. The complexity of retiming depends on the delay of the circuit and the number of path-balancing DFFs. Nevertheless, problems of a million of DFFs can be solved in a few seconds. All the other algorithms are linear (or quadratic) w.r.t the number of nodes in the circuit.

Table 6.5: Evaluation of our method against the state of the art.

Benchmark	**Baseline**		**PBMap [159]**			**Our method**					
	Size	Depth	Area (JJs)	DFFs	Delay	Size	Depth	Area (JJs)	DFFs	Delay	Time (s)
c499	398	19	7758	476	13	217	9	4157	276	9	0.74
c880	325	25	12909	774	22	311	13	7187	482	14	0.85
c1908	341	27	12013	696	20	163	11	3634	287	11	0.74
c3540	1024	41	28300	1159	31	782	21	16278	798	22	2.03
c5315	1776	37	52033	2908	23	1096	17	23849	1735	16	2.08
c7552	1469	26	48482	2429	19	1010	17	22582	1824	16	2.97
s1196	477	19	15332	746	18	454	12	9701	506	12	1.35
s1238	532	23	17617	864	19	499	12	10060	572	13	1.41
s38417	9219	30	208289	8405	21	7492	16	209775	24125	17	5.64
sin	5416	225	215318	13666	182	5254	85	110254	5550	82	13.10
cavlc	693	16	16339	522	17	644	12	11888	381	12	1.70
dec	304	3	5469	8	4	304	3	5096	8	4	0.48
int2float	260	16	6432	270	16	210	10	3891	149	10	0.75
priority	978	250	102085	9064	127	490	13	10099	572	14	4.53
Improvement						**21.72%**	**46.30%**	**43.00%**	**24.20%**	**34.44%**	

Comparing against the state of the art

In our experiments, we conducted a comparison against the current state-of-the-art RSFQ results in PBMap [159]. To perform the evaluation, we utilized the Suny RSFQ cell library [187]. This library shares the same cells as the one employed in [159], which is not openly available. Although the JJ count per cell might vary slightly, delay and DFF count remain unaffected. We report the results over the ISCAS [73] and EPFL [3] benchmarks. The baseline consists of un-optimized designs in the AIG format. We evaluate the quality of the design in terms of JJ count for area and logic depth for latency, as clock frequency cannot be truly characterized before place and route. This is in line with prior work. Differently from PBmap, we map the benchmarks to enable gate-level pipelining. Hence, in our method, also POs are balanced.

Table 6.5 shows the results of the comparison. For our approach, we present the size and depth of the optimized XAG, and the area (number of JJs), number of path-balancing DFFs, and delay (as number of cycles) of the obtained RSFQ circuit. The time shows the total synthesis and mapping run time. For PBMap, we show the area, number of DFFs, and delay. Our synthesis algorithms considerably reduce the average size and depth of the baseline by 21.72% and 46.30%, respectively. After technology mapping, area, DFFs, and delay are reduced by 43.00%, 24.20%, and 34.44% compared to PBMap. Our approach obtains higher area and DFF count only on benchmark *s38417* due to additional DFFs for PO balancing.

6.5 Summary

In this chapter, we focused on logic synthesis for the two most advanced superconducting logic families: the single-flux quantum (SFQ) and the adiabatic quantum-flux parametron (AQFP). We identified challenging problems related to path balancing and fan-out branching, and we proposed technology mapping algorithms to address them. First, we studied the technology legalization problem of AQFP circuits, which involves inserting buffers and splitters elements

(B/S) to meet the path-balancing and fan-out-branching requirements. This problem is particularly challenging since splitter cells are clocked. We demonstrated that there is a polynomial time algorithm to insert buffers and splitters that leads to an optimal depth circuit. We proposed two depth-optimal buffer and splitter insertion techniques based on ALAP scheduling and ASAP scheduling. Additionally, we presented a post-mapping optimization heuristic based on minimum-register retiming to reduce the number of B/S elements. We demonstrated a reduction in the number of B/S elements up to 14% and logic depth up to 15% compared to the previous state-of-the-art scheduling-based B/S insertion method. Our method also achieved competitive results compared to the current state of the art with very short run times, making possible to use these methods in a design-space exploration (DSE) engine. Additionally, we showed that our method achieves near-optimal results on small benchmarks within minimal runtime and scales to benchmarks 10 to 100 times larger than those typically used in AQFP synthesis. Second, we presented a complete logic synthesis flow for the single-flux quantum (SFQ) logic family. We described the technology-independent logic synthesis problem as an XAG minimization problem. The XAG closely abstract the SFQ logic primitives and offers better opportunities to restructure logic compared to AIGs. We proposed several techniques focusing on minimizing the circuit depth as a proxy to reduce circuit imbalances and, consequently, path balancing costs. These methods include advanced algebraic rules, ESOP balancing, and graph mapping for XAGs. Our technology mapping methodology consists of library binding using supergates, buffer insertion and optimization, and splitter insertion. We realized an automatic tool for SFQ synthesis by combining these methods. Our technology-independent synthesis flow reduced the average depth and size of designs by 46% and 21.72%, respectively. Additionally, we demonstrated an average reduction in area, number of DFFs, and delay by 43.00%, 24.20%, and 34.44%, respectively, compared to the state of the art.

7 Conclusions

In this thesis, we investigated technology mapping and optimization algorithms for logic synthesis of advanced technologies. Motivated by (i) the need for novel techniques to improve EDA tools for established technologies; and (ii) promising power-efficient superconducting technologies, we researched and improved technology mapping and synthesis solutions to enhance the quality of modern integrated circuits for CMOS and superconducting electronics.

7.1 Summary of Thesis Contributions

We list the contributions of this thesis following the same order they appear in the thesis.

- **Chapter 3 - Technology Mapping for FPGAs: we significantly improved performance-driven technology mapping for FPGAs using novel Boolean decomposition algorithms.**

  First, we introduced efficient algorithms to compute the Ashenhurst-Curtis decomposition (ACD) of functions into lookup tables (LUTs). We proposed several improvements that make ACD applicable to LUT mappers and resynthesis engines. We provided algorithms to minimize the decomposition cost in terms of the number of LUTs, edges, and delay, considering input arrival times. We demonstrated that our approach runs up to 80 times faster compared to state-of-the-art Boolean decomposition methods while achieving the decomposition success of an optimum SAT-based implementation. Second, we presented a performance-driven LUT mapper that integrates our formulation of ACD for delay minimization. We showed remarkable delay improvements by 12.39%, on average, compared to the state-of-the-art LUT mapping with structural choices. Furthermore, our mapper discovered some of the best (public) results for combinational benchmark circuits [3]. Third, we proposed a LUT mapper that leverages non-routable FPGA connections between adjacent LUTs to optimize for performance. We demonstrated that our implementation outperforms the state-of-the-art solution for this problem in delay, area, edge count, and run time by 6.22%, 3.82%, 3.09%, and

20%, respectively. Additionally, while we mainly focused on performance, our findings demonstrated the great potential of these methods also for area optimization.

- **Chapter 4 - Technology Mapping for Standard Cells: we improved technology mapping for standard cells by (i) developing novel matching techniques, (ii) extending mapping to support multiple-output cells, and (iii) enhancing area optimization heuristics under delay constraints.**

  First, we introduced a matching technique called *hybrid matching* to improve on top of Boolean matching by supporting large cells and leveraging structural redundancies. We demonstrated a 6.5% average reduction in area compared to Boolean matching, for similar delay, on modern libraries. Second, we developed methods to support multiple-output cells during technology mapping. We addressed the problem of detecting and matching multiple-output cells and introduced the first selection and covering algorithm for multiple-output cells. Our findings showed strong results in reducing the area compared to alternative methods that consider multiple-output cells as white boxes during technology mapping. Third, we studied technology mapping covering algorithms and proposed solutions for improving area optimization under delay constraints. Finally, we developed a modern technology mapper implementing the discussed features. We demonstrated remarkable results in area and run time compared to open-source state-of-the-art technology mappers [30]. On a 7nm standard cell library [44], our mapper achieved an average area reduction of 9.22% and an average delay reduction of 2.59% after buffering and gate sizing.

- **Chapter 5 - Mapping for Logic Synthesis: we improved technology-independent logic synthesis by (i) facilitating the optimization of multiple data structures and the conversion between them, and (ii) developing a framework for the optimization of factored form literals.**

  First, motivated by multiple existing graph representations to support optimization in logic synthesis, we presented a versatile technique called *graph mapping* to convert a homogeneous logic network into another while performing global Boolean optimization. When the destination representation matches the starting one, graph mapping performs logic rewriting. We demonstrated that graph mapping is one of the most effective logic optimization approaches over MIGs, XAGs, and XMGs. Second, we presented algorithms to efficiently leverage Boolean don't care conditions during graph mapping and logic rewriting. We demonstrated that an MIG flow implementing logic rewriting with don't care conditions reduces the number of gates by 4.31%, on average, compared to the state-of-the-art flow. Additionally, we showed that the combination of graph mapping and don't care-based logic rewriting significantly contributes to obtaining the best-known results in MIG size for the EPFL benchmarks [3]. Last, we focused on the optimization of factored form literals, a critical metric that correlates strongly with the number of transistors required in CMOS implementation. We developed a comprehensive portfolio of algorithms to optimize the factored form literal count in Boolean networks modeled as AIGs. We demonstrated that these methods help to reduce the area of standard cell designs by 2.8%, on average, compared to the state-of-the-art AIG synthesis flow. Additionally, we discussed applications in transistor-level synthesis and in the automatic creation of custom standard cells.

- **Chapter 6 - Specializing Synthesis for Superconducting Technologies: we improved logic synthesis for superconducting electronics by (i) studying the theoretical properties of technology mapping with unconventional technological constraints and (ii) developing novel technology mapping and optimization algorithms.**

  First, we investigated the buffer and splitter (B/S) insertion problem in AQFP circuits, which is a step of technology mapping. We proved that the depth-optimal B/S insertion problem is tractable with polynomial complexity, and we provided two depth-optimal algorithms based on the *as-late-as-possible* (ALAP) and *as-soon-as-possible* (ASAP) strategies. Additionally, we developed a post-mapping area-oriented B/S optimization algorithm based on minimum-register retiming. We showed remarkable results in quality, with up to 14% reduction in the number of B/S elements, and demonstrated the scalability of these methods over benchmarks that are from 10 to 100 times larger than the ones that any other related work could handle. Second, we presented a framework for SFQ logic consisting of algebraic and Boolean synthesis transformations for XAGs focusing on depth minimization, and technology mapping satisfying the SFQ technological requirements. We showed strong results compared to state-of-the-art SFQ flows with an average reduction in the area and delay of 43% and 34%, respectively.

7.2 Open Problems

In this section, we provide future research directions in logic synthesis inspired by the problems addressed in this thesis.

- **Improving Boolean decomposition for LUT mapping.** In Chapter 3, we introduced several flexible and powerful algorithms to solve the Ashenhurst-Curtis decomposition (ACD) problem. However, three opportunities remain (partially) unexplored: (i) ACD for incompletely-specified Boolean functions; (ii) ACD targeting more than two levels of LUTs; and (iii) run time acceleration using GPUs. First, the proposed ACD method can be extended to handle incompletely-specified Boolean functions by incorporating an additional step that minimizes the number of unique cofactors during variable partitioning under the available don't care conditions. Second, while we developed an algorithm capable of performing decomposition across multiple levels of logic, its algorithmic complexity and dependency on truth tables currently limits its scalability during mapping or resynthesis. Significant progress can be achieved by leveraging an initial variable order, which can be determined by arrival time, and by using BDDs. These strategies can reduce complexity and improve the practicality of our multi-level decomposition approach. Third, our proposed algorithms can be re-engineered to run on GPUs. This approach could significantly reduce run times by enabling multiple decompositions to be computed in parallel during technology mapping (e.g., lowering the run times in Table 3.8).

- **Enhancing area-driven LUT mapping with Boolean decomposition.** Towards the end of Chapter 3, we briefly explored the significant potential of ACD for area-oriented LUT mapping, demonstrating remarkable results for certain benchmarks. However, as our primary focus in this chapter was on performance optimization, we did not develop highly efficient engines for area optimization. This omission presents an opportunity for future research to advance area-oriented LUT mapping techniques.
- **Non-routable connection structure in FPGAs.** In Chapter 3, we proposed algorithms to leverage non-routable connections between adjacent LUTs in recent AMD FPGAs [10], which form a single-rail cascade structure, to reduce the delay. However, an FPGA using a multiple-rail cascade structure could further improve the performance of typical designs. This presents an opportunity for future research to investigate which FPGA architectures should be implemented to maximize performance, specifically by analyzing the mappability of functions and their delay (and area) across various LUT structures.
- **Revisiting delay models for technology mapping.** During our experiments in Chapter 4, we observed that the gain-based delay model for certain standard cell libraries lacked accuracy. This presents a significant opportunity to research more sophisticated models. Additionally, instead of proposing new strategies to estimate delay, gain-based delay models could be tuned in multiple delay iterations of technology mapping using approximated results from static timing analysis. This approach would help estimate the impact of load and slew on critical and near-critical paths, potentially leading to better optimization of circuit performance.
- **Inverter minimization in technology mapping.** Technology mapping for standard cells is a complex problem involving multiple heuristic choices and trade-offs. For instance, positive slack on paths can be leveraged to reduce area, but selecting the optimal path location where to reduce area among multiple eligible options is challenging. One particularly difficult problem is inverter placement. The initial placement of inverters often influences subsequent mapping decisions. During our experiments, we observed that improper inverter placement could result in up to a 15% increase in area after mapping. For example, in the IWLS benchmark *leon3* [85], sub-optimal gate selection could lead to a network with thousands of unnecessary inverters. Recovering from incorrect inverter placement is a difficult task, as it requires significant adjustments to reference estimations. By tweaking referencing policies in Section 4.5.2, alternative solutions can be found, but no single method consistently works in practice. This highlights a significant opportunity for future research to develop more robust strategies for inverter placement and overall technology mapping optimization.
- **Post-mapping optimization for standard cell design.** While this thesis focused on technology-independent synthesis and technology mapping for standard cell designs,

significant opportunities lie in post-mapping optimization. Technology-dependent synthesis can leverage more accurate models of power, delay, and area to achieve superior results. For instance, the critical path identified in a technology-independent model often does not align with the critical path after mapping, as load effects are not considered in the technology-independent models. Currently, there are no open-source tools that offer post-mapping optimization methods for standard cell designs, presenting a substantial opportunity for further research and development in this domain.

Bibliography

[1] Akers. “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516.

[2] Luca Amaru, A. Ajami, S. Chen, Y. Zhang, T.-L. Tung, T. Arifin, T. Liu, M. Pan, G. Naveen, J. C. Vujkovic, P. Moceyunas, L. Clark, S. Whiteley, E. Mlinar, S. Lu, R. Singh, J. Chase, A. Belov, D. Rawlings, S. Anderson, A. Salz, R. Freeman, J. Kawa, and S. Chase. “First demonstration of a superconducting electronics microcontroller RTL-to-GDSII flow”. In: *GOMACTech* (2021), pp. 1–4.

[3] L. Amarù, P.-E. Gaillardon, and G. De Micheli. “The EPFL Combinational Benchmark Suite”. In: *Proc. IWLS.* 2015.

[4] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Biconditional Binary Decision Diagrams: A Novel Canonical Logic Representation Form”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 4.4 (2014), pp. 487–500.

[5] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Majority-Inverter Graph: A New Paradigm for Logic Optimization”. In: *IEEE Trans. CAD* 35.5 (2016).

[6] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization”. In: *Proc. DAC.* San Francisco, CA, USA, 2014.

[7] Luca Amarú, Pierre-Emmanuel Gaillardon, Subhasish Mitra, and Giovanni De Micheli. “New Logic Synthesis as Nanotechnology Enabler”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2168–2195.

[8] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Pierre-Emmanuel Gaillardon, Janet Olson, Robert Brayton, and Giovanni De Micheli. “Enabling exact delay synthesis”. In: *Proc. ICCAD.* 2017.

[9] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Janet Olson, Robert Brayton, and Giovanni De Micheli. “Improvements to boolean resynthesis”. In: *Proc. DATE.* 2018, pp. 755–760.

[10] *AMD Versal CLB documentation.* Accessed: 2024-09-06. URL: https://docs.amd.com/r/en-US/am005-versal-clb/Look-Up-Table.

[11] Yuki Ando, Ryo Sato, Masamitsu Tanaka, Kazuyoshi Takagi, Naofumi Takagi, and Akira Fujimaki. “Design and Demonstration of an 8-bit Bit-Serial RSFQ Microprocessor: CORE e4”. In: *IEEE Transactions on Applied Superconductivity* 26.5 (2016), pp. 1–5.

[12] R. L. Ashenhurst. “The decomposition of switching functions”. In: *Proc. Int. Symp. Theory Switch.* 1957, pp. 74–116.

[13] Christopher L Ayala, Ro Saito, Tomoyuki Tanaka, Olivia Chen, Naoki Takeuchi, Yuxing He, and Nobuyuki Yoshikawa. “A semi-custom design methodology and environment for implementing superconductor adiabatic quantum-flux-parametron microprocessors”. In: *Superconductor Science and Technology* 33.5 (2020).

[14] Christopher L. Ayala, Tomoyuki Tanaka, Ro Saito, Mai Nozoe, Naoki Takeuchi, and Nobuyuki Yoshikawa. “MANA: A Monolithic Adiabatic iNtegration Architecture Microprocessor Using 1.4-zJ/op Unshunted Superconductor Josephson Junction Devices”. In: *IEEE Journal of Solid-State Circuits* 56.4 (2021), pp. 1152–1165.

[15] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. “Algebraic decision diagrams and their applications”. In: *Proc. ICCAD.* 1993, pp. 188–191.

[16] Rassul Bairamkulov, Alessandro Tempia Calvino, and Giovanni De Micheli. “Synthesis of SFQ Circuits with Compound Gates”. In: *Proc. VLSI-SoC.* 2023.

[17] Rassul Bairamkulov and Giovanni De Micheli. “Compound Logic Gates for Pipeline Depth Minimization in Single Flux Quantum Integrated Systems”. In: *Proc. GLVLSI.* 2023.

[18] Rassul Bairamkulov and Giovanni De Micheli. “Superconductive Electronics: A 25-Year Review [Feature]”. In: *IEEE Circuits and Systems Magazine* 24.2 (2024), pp. 16–33.

[19] Rassul Bairamkulov and Giovanni De Micheli. “Towards Multiphase Clocking in Single-Flux Quantum Systems”. In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC).* 2024, pp. 582–587.

[20] Rassul Bairamkulov, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, Mingfei Yu, and Giovanni De Micheli. “Technology-Aware Logic Synthesis for Superconducting Electronics”. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE).* 2024, pp. 1–6.

[21] F. Beeftink, P. Kudva, D. Kung, and L. Stok. “Gate-size selection for standard cell libraries”. In: *IEEE/ACM International Conference on Computer-Aided Design.* 1998, pp. 545–550.

[22] L. Benini, P. Vuillod, and G. De Micheli. “Iterative remapping for logic circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.10 (1998), pp. 948–964.

[23] Luca Benini and Giovanni De Micheli. “A Survey of Boolean Matching Techniques for Library Binding”. In: *ACM Trans. Design Autom. Electr. Syst.* (July 1997).

[24] V. Bertacco and M. Damiani. "Boolean function representation based on disjoint-support decompositions". In: *Proc. Int. Conf. on Comp. Design.* 1996, pp. 27–32.

[25] V. Bertacco and M. Damiani. "The disjunctive decomposition of logic functions". In: *Proc. ICCAD.* 1997, pp. 78–82.

[26] P. Bjesse and A. Boralv. "DAG-aware circuit compression for formal verification". In: *IEEE/ACM ICCAD.* 2004.

[27] Joan Boyar, René Peralta, and Denis Pochuev. "On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$". In: *Theoretical Computer Science* 235.1 (2000), pp. 43–57.

[28] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. "MIS: A Multiple-Level Logic Optimization System". In: *IEEE Trans. CAD* (1987).

[29] Robert Brayton and C. McMullen. "The decomposition and factorization of Boolean expression". In: *Proc. ISCAS.* 1982.

[30] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification.* Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. 2010. URL: https://github.com/berkeley-abc/abc.

[31] Frank Markham Brown. *Boolean reasoning: the logic of Boolean equations.* Dover Publications, 2nd edition, 2012.

[32] R. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. on Computers* C-35.8 (1986), pp. 677–691.

[33] Ruizhe Cai, Olivia Chen, Ao Ren, Ning Liu, Caiwen Ding, Nobuyuki Yoshikawa, and Yanzhi Wang. "A Majority Logic Synthesis Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits". In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI.* ACM, 2019.

[34] Ruizhe Cai, Olivia Chen, Ao Ren, Ning Liu, Caiwen Ding, Nobuyuki Yoshikawa, and Yanzhi Wang. "A Majority Logic Synthesis Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits". In: *Proceedings of GLSVLSI.* 2019, pp. 189–194.

[35] Ruizhe Cai, Olivia Chen, Ao Ren, Ning Liu, Nobuyuki Yoshikawa, and Yanzhi Wang. "A Buffer and Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits". In: *Proceedings of ICCD.* 2019, pp. 429–436.

[36] Vinicius Callegaro, Felipe S. Marranghello, Mayler G. A. Martins, Renato P. Ribas, and Andre I. Reis. "Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis". In: *ICCD.* 2015, pp. 680–687.

[37] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. "Reducing structural bias in technology mapping". In: *Proc. ICCAD.* 2005, pp. 519–526.

[38] Satrajit Chatterjee. "On Algorithms for Technology Mapping". PhD thesis. EECS Department, University of California, Berkeley, 2007.

[39] Satrajit Chatterjee, Alan Mishchenko, and Robert Brayton. “Factor Cuts”. In: *2006 IEEE/ACM International Conference on Computer Aided Design*. 2006, pp. 143–150.

[40] D. Chen and J. Cong. “DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs”. In: *Proc. ICCAD*. 2004.

[41] G. Chen and J. Cong. “Simultaneous Logic Decomposition with Technology Mapping in FPGA Designs”. In: *Proc. FPGA*. 2001, pp. 48–55.

[42] Olivia Chen, Ruizhe Cai, Yetang Wang, Fei Ke, Taiki Yamae, Ro Saito, Naoki Takeuchi, and Nobuyuki Yoshikawa. “Adiabatic Quantum-Flux-Parametron: Towards Building Extremely Energy-Efficient Circuits and Systems”. In: *Scientific Reports* 9 (2019).

[43] Zhufei Chu, Mathias Soeken, Yinshui Xia, and Giovanni De Micheli. “Functional decomposition using majority”. In: *ASP-DAC*. 2018, pp. 676–681.

[44] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. “ASAP7: A 7-nm finFET predictive process design kit”. In: *Microelectronics Journal* (2016).

[45] J. Cong and Y. Ding. “FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”. In: *Trans. CAD* 13.1 (1994), pp. 1–12.

[46] J. Cong and Yean-Yow Hwang. “Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design”. In: *DAC*. 1996, pp. 726–729.

[47] J. Cong, C. Wu, and Y. Ding. “Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution”. In: *Proc. FPGA*. 1999.

[48] Jason Cong, Yuzheng Ding, Tong Gao, and Kuang-Chien Chen. “LUT-based FPGA technology mapping under arbitrary net-delay models”. In: *Computers & Graphics* 18.4 (1994), pp. 507–516.

[49] J. P. Curtis. *A New Approach to the Design of Switching Circuits*. D. Van Nostrand, 1962.

[50] M. De Marchi, D. Sacchetto, S. Frache, J. Zhang, P.-E. Gaillardon, Y. Leblebici, and G. De Micheli. “Polarity control in double-gate, gate-all-around vertically stacked silicon nanowire FETs”. In: *2012 International Electron Devices Meeting*. 2012, pp. 8.4.1–8.4.4.

[51] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. “Optimal State Assignment for Finite State Machines”. In: *Trans. CAD* 4.3 (1985), pp. 269–285.

[52] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[53] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. “Technology mapping in MIS”. In: *Proc. ICCAD*. 1987, pp. 116–119.

[54] R. Drechsler. “Pseudo-Kronecker expressions for symmetric functions”. In: *IEEE Transactions on Computers* 48.9 (1999), pp. 987–990.

[55] R. Drechsler, N. Drechsler, and W. Gunther. "Fast exact minimization of BDD's". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.3 (2000), pp. 384–389.

[56] W. C. Elmore. "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers". In: *Journal of Applied Physics* 19.1 (Jan. 1948), pp. 55–63. eprint: https://pubs.aip.org/aip/jap/article-pdf/19/1/55/18307672/55_1_online.pdf.

[57] *EPFL Synthesis Competition Best Results [2023]*. URL: https://github.com/lsils/benchmarks/tree/v2023.1/best_results.

[58] Longfei Fan and Chang Wu. "FPGA Technology Mapping with Adaptive Gate Decomposition". In: *Proc. FPGA*. 2023, pp. 135–140.

[59] A. H. Farrahi and M. Sarrafzadeh. "Complexity of the lookup-table minimization problem for FPGA technology mapping". In: *IEEE Trans. CAD* 13.11 (1994), pp. 1319–1332.

[60] R.J. Francis, J. Rose, and K. Chung. "Chortle: a technology mapping program for lookup table-based field programmable gate arrays". In: *DAC*. 1990, pp. 613–619.

[61] Rongliang Fu, Junying Huang, Mengmeng Wang, Nobuyuki Yoshikawa, Bei Yu, Tsung-Yi Ho, and Olivia Chen. "BOMIG: A Majority Logic Synthesis Framework for AQFP Logic". In: *Proceedings of DATE*. 2023.

[62] Rongliang Fu, Mengmeng Wang, Yirong Kan, Nobuyuki Yoshikawa, Tsung-Yi Ho, and Olivia Chen. "A Global Optimization Algorithm for Buffer and Splitter Insertion in Adiabatic Quantum-Flux-Parametron Circuits". In: *Proceedings of ASP-DAC*. 2023, pp. 769–774.

[63] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis". In: *Proceedings of the European Conference on Design Automation*. 1992, pp. 50–54.

[64] Kris Gaj, Eby G. Friedman, and Marc J. Feldman. "Timing of Multi-Gigahertz Rapid Single Flux Quantum Digital Circuits". In: *High Performance Clock Distribution Networks*. Ed. by Eby G. Friedman. Springer US, 1997, pp. 135–164.

[65] D. Gregory, K. Bartlett, A. deGeus, and G. Hachtel. "SOCRATES: A system for automatically synthesizing and optimizing combinational logic". In: *Proceedings of the Design Automation Conference*. 1986, pp. 580–586.

[66] Antoine Grosnit, Cedric Malherbe, Rasul Tutunov, Xingchen Wan, Jun Wang, and Haitham Bou Ammar. "BOiLS: Bayesian Optimisation for Logic Synthesis". In: *DATE*. 2022, pp. 1193–1196.

[67] R. Gupta, B. Tutuianu, and L.T. Pileggi. "The Elmore delay as a bound for RC trees with generalized input signals". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.1 (1997), pp. 95–104.

[68] W. Haaswijk, M. Soeken, L. Amarù, P. Gaillardon, and G. De Micheli. “A novel basis for logic rewriting”. In: *Proceedings Asia and South Pacific Design Automation Conference*. 2017.

[69] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli. “SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism”. In: *IEEE Trans. CAD* (2020).

[70] Winston Jason Haaswijk, Mathias Soeken, Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “LUT Mapping and Optimization for Majority-Inverter Graphs”. In: *Proc. IWLS*. 2016.

[71] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Springer New York, NY, 1996.

[72] Ivo Háleček, Petr Fišer, and Jan Schmidt. “Are XORs in logic synthesis really necessary?” In: *IEEE Proc. DDECS*. 2017.

[73] M. C. Hansen, H. Yalcin, and J. P. Hayes. “Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering”. In: *IEEE Des. Test. Comput.* (1999).

[74] Yuxing He, Christopher L Ayala, Yu Zeng, Xihua Zou, Lianshan Yan, Wei Pan, and Nobuyuki Yoshikawa. “Low clock skew superconductor adiabatic quantum-flux-parametron logic circuits based on grid-distributed blocks”. In: *Superconductor Science and Technology* 36.1 (2022), p. 015006.

[75] Leo Hellerman. “A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits”. In: *IEEE Transactions on Electronic Computers* EC-12.3 (1963), pp. 198–223.

[76] Quentin P. Herr, Anna Y. Herr, Oliver T. Oberg, and Alexander G. Ioannidis. “Ultra-low-power superconductor logic”. In: *Journal of Applied Physics* 109.10 (2011), p. 103903.

[77] Gage Hills, Christian Lau, Andrew Wright, Samuel Fuller, Mindy Bishop, Tathagata Srimani, Pritpal Kanhaiya, Rebecca Ho, Aya Amer, Yosi Stein, Denis Murphy, Arvind Arvind, Anantha Chandrakasan, and Max Shulaker. “Modern microprocessor built from complementary carbon nanotube transistors”. In: *Nature* 572 (Aug. 2019), pp. 595–602.

[78] D. Scott Holmes, Andrew L. Ripple, and Marc A. Manheimer. “Energy-Efficient Superconducting Computing—Power Budgets and Requirements”. In: *IEEE Trans. on Applied Superconductivity* (2013).

[79] Bo Hu, Y. Watanabe, and M. Marek-Sadowska. “Gain-based technology mapping for discrete-size cell libraries”. In: *Proceedings Design Automation Conference*. 2003, pp. 574–579.

[80] Chao-Yuan Huang, Yi-Chen Chang, Ming-Jer Tsai, and Tsung-Yi Ho. “An Optimal Algorithm for Splitter and Buffer Insertion in Adiabatic Quantum-Flux-Parametron Circuits”. In: *ICCAD*. 2021.

[81] Juinn-Dar Huang, Jing-Yang Jou, and Wen-Zen Shen. “Compatible class encoding in Roth-Karp decomposition for two-output LUT architecture”. In: *Proc. ICCAD*. 1995, pp. 359–363.

[82] Zheng Huang, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko. “Fast Boolean matching based on NPN classification”. In: *Intern. Conf. on Field-Programmable Technology*. 2013, pp. 310–313.

[83] Aaron P. Hurst, Alan Mishchenko, and Robert K. Brayton. “Fast Minimum-Register Retiming via Binary Maximum-Flow”. In: *Formal Methods in Computer Aided Design (FMCAD’07)*. 2007.

[84] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai. “A rule based logic reorganization system LORES/EX”. In: *Proceedings 1988 IEEE International Conference on Computer Design: VLSI*. 1988, pp. 262–266.

[85] *IWLS 2005 Benchmarks*. http://iwls.org/iwls2005/benchmarks.html. Accessed: 2024-07-06.

[86] Nicola Jones. “How to stop data centres from gobbling up the world’s electricity”. In: *Nature* 561 (Sept. 2018), pp. 163–166.

[87] S.K. Karandikar and S.S. Sapatnekar. “Logical effort based technology mapping”. In: *IEEE/ACM International Conference on Computer Aided Design*. 2004, pp. 419–422.

[88] Naveen Kumar Katam and Massoud Pedram. “Logic Optimization, Complex Cell Design, and Retiming of Single Flux Quantum Circuits”. In: *IEEE Trans. on Applied Superconductivity* (2018).

[89] T. Kawaguchi, M. Tanaka, K. Takagi, and N. Takagi. “Demonstration of an 8-Bit SFQ Carry Look-Ahead Adder Using Clockless Logic Cells”. In: *International Superconductive Electronics Conference*. 2015.

[90] K. Keutzer. “DAGON: Technology Binding and Local Optimization by DAG Matching”. In: *Proceedings of Design Automation Conference*. 1987.

[91] Alexander Khitun and Kang L. Wang. “Non-volatile magnonic logic circuits engineering”. In: *Journal of Applied Physics* 110.3 (2011), p. 034306.

[92] Nobutaka Kito, Kazuyoshi Takagi, and Naofumi Takagi. “Logic-Depth-Aware Technology Mapping Method for RSFQ Logic Circuits With Special RSFQ Gates”. In: *IEEE Transactions on Applied Superconductivity* 32.4 (2022), pp. 1–5.

[93] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. 1st. Addison-Wesley Professional, 2015.

[94] Kun Kong, Yun Shang, and Ruqian Lu. “An Optimized Majority Logic Synthesis Methodology for Quantum-Dot Cellular Automata”. In: *IEEE Transactions on Nanotechnology* 9.2 (2010), pp. 170–183.

[95] V. N. Kravets and K. A. Sakallah. “Constructive Multi-Level Synthesis by Way of Functional Properties”. PhD thesis. University of Michigan, 2001.

[96] G. Krylov and E. G. Friedman. *Single Flux Quantum Integrated Circuit Design*. Springer, 2022.

[97] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. “Robust Boolean reasoning for equivalence checking and functional property verification”. In: *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems* 21 (2002), pp. 1377–1394.

[98] Y. Kukimoto, R.K. Brayton, and P. Sawkar. “Delay-optimal technology mapping by DAG covering”. In: *Proc. DAC*. 1998, pp. 348–351.

[99] Yung-Te Lai, M. Pedram, and S.B.K. Vrudhula. “BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis”. In: *DAC*. 1993.

[100] Eugene L. Lawler. “An Approach to Multilevel Boolean Minimization”. In: *J. ACM* 11.3 (1964), pp. 283–295.

[101] Eugene L. Lawler. *Combinatorial Optimization*. Holt Rinehart Winston, 1976.

[102] Choong Y. Lee. “Representation of switching circuits by binary-decision programs”. In: *Bell System Technical Journal* 38 (1959), pp. 985–999.

[103] Daeyeal Lee, Dongwon Park, Chia-Tung Ho, Ilgweon Kang, Hayoung Kim, Sicun Gao, Bill Lin, and Chung-Kuan Cheng. “SP&R: SMT-Based Simultaneous Place-and-Route for Standard Cell Synthesis of Advanced Nodes”. In: *IEEE Trans. CAD* (2021).

[104] Siang-Yun Lee and Giovanni De Micheli. “Heuristic Logic Resynthesis Algorithms at the Core of Peephole Optimization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

[105] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Beyond local optimality of buffer and splitter insertion for AQFP circuits”. In: *Proceedings of DAC*. 2022, pp. 445–450.

[106] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Customizable On-the-fly Design Space Exploration for Logic Optimization of Emerging Technologies”. In: *International Logic Synthesis Workshop (IWLS)*. 2023.

[107] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Irredundant Buffer and Splitter Insertion and Scheduling-Based Optimization for AQFP Circuits”. In: *in Proc. IWLS*. 2021.

[108] Siang-Yun Lee, Heinz Riener, Alan Mishchenko, Robert K. Brayton, and Giovanni De Micheli. “A Simulation-Guided Paradigm for Logic Synthesis and Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.8 (2022), pp. 2573–2586.

[109] Siang-Yun Lee, Alessandro Tempia Calvino, Heinz Riener, and Giovanni De Micheli. “Late Breaking Results: Majority-Inverter Graph Minimization by Design Space Exploration”. In: *Proc. Design Automation Conference*. 2024.

[110] Siang-Yun Lee, Alessandro Tempia Calvino, Heinz Riener, and Giovanni De Micheli. “Technology Legalization and Optimization for Adiabatic Quantum-Flux Parametron”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[111] Tai-Cheng Lee, Cheng-Yen Yang, and Yih-Lang Li. "ITPlace: Machine Learning-Based Delay-Aware Transistor Placement for Standard Cell Synthesis". In: *Proc. ICCAD*. 2020.

[112] C. Legl, B. Wurth, and K. Eckl. "A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs". In: *DAC*. 1996, pp. 730–733.

[113] C. Legl, B. Wurth, and K. Eckl. "Computing support-minimal subfunctions during functional decomposition". In: *Trans. VLSI* 6.3 (1998), pp. 354–363.

[114] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. "Logic decomposition during technology mapping". In: *Trans. CAD* 16.8 (1997), pp. 813–834.

[115] Charles E. Leiserson and James B. Saxe. "Retiming Synchronous Circuitry". In: *Algorithmica* 6.1–6 (1991), pp. 5–35.

[116] C S Lent, P D Tougaw, W Porod, and G H Bernstein. "Quantum cellular automata". In: *Nanotechnology* 4.1 (1993), p. 49.

[117] I. Levin and R.Y. Pinter. "Realizing expression graphs using table-lookup FPGAs". In: *Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference*. 1993, pp. 306–311.

[118] Xi Li, Min Pan, Tong Liu, and Peter A. Beerel. "Multi-Phase Clocking for Multi-Threaded Gate-Level-Pipelined Superconductive Logic". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022, pp. 62–67.

[119] K.K. Likharev and V.K. Semenov. "RSFQ logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems". In: *IEEE Trans. on Applied Superconductivity* (1991).

[120] Tianji Liu, Lei Chen, Xing Li, Mingxuan Yuan, and Evangeline F. Y. Young. "FineMap: A Fine-Grained GPU-Parallel LUT Mapping Engine". In: *Proc. ASP-DAC*. 2024, pp. 392–397.

[121] Lucas Machado and Jordi Cortadella. "Support-Reducing Decomposition for FPGA Mapping". In: *TCAD* 39.1 (2020), pp. 213–224.

[122] F. Mailhot and G. DeMicheli. "Automatic layout and optimization of static CMOS cells". In: *Proceedings 1988 IEEE International Conference on Computer Design: VLSI*. 1988.

[123] Frédéric Mailhot and Giovanni De Micheli. "Technology mapping using boolean matching and don't care sets". In: *Proceedings of the European Conference on Design Automation*. 1990.

[124] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. "Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping". In: *IEEE Trans. CAD* 25.11 (2006), pp. 2331–2340.

[125] Dewmini Sudara Marakkalage and Giovanni De Micheli. "Fanout-Bounded Logic Synthesis for Emerging Technologies". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.5 (2024), pp. 1415–1428.

[126] Dewmini Sudara Marakkalage, Heinz Riener, and Giovanni De Micheli. “Optimizing Adiabatic Quantum-Flux-Parametron (AQFP) Circuits using an Exact Database”. In: *NANOARCH*. 2021, pp. 1–6.

[127] Dewmini Sudara Marakkalage, Eleonora Testa, Heinz Riener, Alan Mishchenko, Mathias Soeken, and Giovanni De Micheli. “Three-Input Gates for Logic Synthesis”. In: *IEEE Trans. CAD* (2020).

[128] Osvaldo Martinello, Felipe S. Marques, Renato P. Ribas, and André I. Reis. “KL-Cuts: A new approach for logic synthesis targeting multiple output blocks”. In: *Proc. DATE*. 2010, pp. 777–782.

[129] Giulia Meuli, Vinicius Possani, Rajinder Singh, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, Patrick Vuillod, Luca Amaru, Scott Chase, Jamil Kawa, and Giovanni De Micheli. “Majority-based Design Flow for AQFP Superconducting Family”. en. In: *DATE* (2022), p. 6.

[130] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. “Xor-And-Inverter Graphs for Quantum Compilation”. In: *npj Quantum Information* 8.7 (2022).

[131] Shin-ichi Minato. “Zero-suppressed BDDs for set manipulation in combinatorial problems”. In: *Proceedings of the 30th International Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1993, pp. 272–277.

[132] A. Mishchenko and R. Brayton. “Scalable Logic Synthesis using a Simple Circuit Structure”. In: *Proc. IWLS*. 2006.

[133] A. Mishchenko, R. Brayton, and S. Chatterjee. “Boolean factoring and decomposition of logic networks”. In: *Proc. ICCAD*. 2008, pp. 38–44.

[134] A. Mishchenko, R. Brayton, and S. Jang. “Global Delay Optimization Using Structural Choices”. In: *Proc. FPGA*. 2010, pp. 181–184.

[135] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. “Scalable Don’t-Care-Based Logic Optimization and Resynthesis”. In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (2011).

[136] A. Mishchenko and R.K. Brayton. “SAT-based complete don’t-care computation for network optimization”. In: *Design, Automation and Test in Europe*. 2005, 412–417 Vol. 1.

[137] A. Mishchenko, S. Chatterjee, and R. Brayton. “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis”. In: *Proc. DAC*. 2006.

[138] A. Mishchenko, S. Chatterjee, and R. Brayton. *Fast Boolean matching for LUT structures*. Tech. rep. EECS Dep., UC Berkeley, 2007.

[139] A. Mishchenko, S. Chatterjee, and R. Brayton. *FRAIGs: A unifying representation for logic synthesis and verification*. Tech. rep. EECS Dep., UC Berkeley, 2005.

[140] A. Mishchenko, S. Chatterjee, and R. K. Brayton. “Improvements to Technology Mapping for LUT-Based FPGAs”. In: *IEEE Trans. CAD* (2007).

[141] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. “Combinational and sequential mapping with priority cuts”. In: *Proc. ICCAD*. 2007.

[142] A. Mishchenko and T. Sasao. “Encoding of Boolean Functions and Its Application to LUT Cascade Synthesis”. In: *Proc. IWLS*. 2002.

[143] Alan Mishchenko, Robert Brayton, Stephen Jang, and Victor Kravets. “Delay optimization using SOP balancing”. In: *Proc. ICCAD*. 2011, pp. 375–382.

[144] Alan Mishchenko, Robert Brayton, Alessandro Tempia Calvino, and Giovanni De Micheli. “Boolean Decomposition Revisited”. In: *Proc. IWLS*. 2023.

[145] Alan Mishchenko, Robert K. Brayton, Walter Lau Neto, Pierre-Emamnuel Gaillardon, and Luca Amarù. “Control logic restructuring for area optimization”. In: *International Workshop on Logic & Synthesis*. 2022.

[146] Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton, and Maciej J. Ciesielski. “An Integrated Technology Mapping Environment”. In: *International Workshop on Logic & Synthesis*. 2005.

[147] Alan Mishchenko and Marek Perkowski. “Fast Heuristic Minimization of Exclusive-Sums-of-Products”. In: *Intern. Reed-Muller Workshop* (2001).

[148] Eric Mlinar, Stephen Whiteley, Anton Belov, Song Chen, Luca Amaru, Tong Liu, Yalan Zhang, Taufik Arifin, Min Pan, Troy Barbee, Rajinder Singh, Amir Ajami, Danny Rawlings, Giulia Meuli, Rajesh Kumar, Arturo Salz, Scott Chase, and Jamil Kawa. “An RTL-to-GDSII Flow for Single Flux Quantum Circuits Based on an Industrial EDA Toolchain”. In: *IEEE Transactions on Applied Superconductivity* 33.5 (2023), pp. 1–7.

[149] Oleg A. Mukhanov. “Energy-Efficient Single Flux Quantum Technology”. In: *IEEE Transactions on Applied Superconductivity* (2011), pp. 760–769.

[150] R. Murgai. “Performance optimization under rise and fall parameters”. In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*. 1999, pp. 185–190.

[151] S. Muroga, Y. Kambayashi, H.C. Lai, and J.N. Culliney. “The transduction method-design of logic networks based on permissible functions”. In: *Trans. on Computers* 38.10 (1989).

[152] Cody D. Murray and R. Ryan Williams. “On the (non) NP-hardness of computing circuit complexity”. In: *Proceedings of the 30th Conference on Computational Complexity*. CCC ’15. Portland, Oregon: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 365–380.

[153] Walter Lau Neto, Luca Amarú, Vinicius Possani, Patrick Vuillod, Jiong Luo, Alan Mishchenko, and Pierre-Emmanuel Gaillardon. “Improving LUT-based optimization for ASICs”. In: *Proc. DAC*. 2022.

[154] Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. “FlowTune: End-to-End Automatic Logic Optimization Exploration via Domain-Specific Multi-armed Bandit”. In: *Trans. CAD* 42.6 (2023), pp. 1912–1925.

[155] Peichen Pan and Chih-Chang Lin. “A New Retiming-Based Technology Mapping Algorithm for LUT-Based FPGAs”. In: *Proc. ACM/SIGDA Sixth International Symposium on FPGA*. 1998.

[156] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition*. USA: Oxford University Press, 2010.

[157] Ghasem Pasandi and Massoud Pedram. “An Efficient Pipelined Architecture for Superconducting Single Flux Quantum Logic Circuits Utilizing Dual Clocks”. In: *IEEE Trans. on Applied Superconductivity* (2020).

[158] Ghasem Pasandi and Massoud Pedram. “Balanced Factorization and Rewriting Algorithms for Synthesizing Single Flux Quantum Logic Circuits”. In: *Proc. GLSVLSI*. 2019.

[159] Ghasem Pasandi and Massoud Pedram. “PBMap: A Path Balancing Technology Mapping Algorithm for Single Flux Quantum Logic Circuits”. In: *Trans. on Applied Superconductivity* (2019).

[160] Ghasem Pasandi and Massoud Pedram. “qSeq: Full Algorithmic and Tool Support for Synthesizing Sequential Circuits in Superconducting SFQ Technology”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 133–138.

[161] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J.S. Zhang. “Decomposition of multiple-valued relations”. In: *Proc. Inter. Symp. on Mult.- Valued Logic*. 1997, pp. 13–18.

[162] Vinicius Neves Possani, Vinicius Callegaro, André I. Reis, Renato P. Ribas, Felipe de Souza Marques, and Leomar Soares da Rosa. “Graph-Based Transistor Network Generation Method for Supergate Design”. In: *IEEE Trans. VLSI Systems* (2016).

[163] Gianluca Radi, Alessandro Tempia Calvino, and Giovanni De Micheli. “In Medio Stat Virtus*: Combining Boolean and Pattern Matching”. In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2024, pp. 404–410.

[164] Shubham Rai, Alessandro Tempia Calvino, Heinz Riener, Giovanni De Micheli, and Akash Kumar. “Utilizing XMG-Based Synthesis to Preserve Self-Duality for RFET-Based Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.3 (2023), pp. 914–927.

[165] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen. “Mapping into LUT structures”. In: *Proc. DATE*. 2012, pp. 1579–1584.

[166] Giovanni V. Resta, Alessandra Leonhardt, Yashwanth Balaji, Stefan De Gendt, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Devices and Circuits Using Novel 2-D Materials: A Perspective for Future VLSI Systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.7 (2019), pp. 1486–1503.

[167] Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. “On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis”. In: *DATE*. 2019.

[168] Heinz Riener, Eleonora Testa, Luca Amaru, Mathias Soeken, and Giovanni De Micheli. “Size Optimization of MIGs with an Application to QCA and STMG Technologies”. In: *Proc. NANOARCH.* 2018.

[169] Heinz Riener, Eleonora Testa, Winston Haaswijk, Alan Mishchenko, Luca Amarú, Giovanni De Micheli, and Mathias Soeken. “Scalable Generic Logic Synthesis: One Approach to Rule Them All”. In: *Proc. DAC.* 2019, pp. 1–6.

[170] J. P. Roth and R. M. Karp. “Minimization Over Boolean Graphs”. In: *IBM Journal of Research and Development* 6.2 (1962), pp. 227–238.

[171] R.L. Rudell and A. Sangiovanni-Vincentelli. “Multiple-Valued Minimization for PLA Optimization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.5 (1987), pp. 727–750.

[172] Richard Rudell. “Logic Synthesis for VLSI Design”. PhD thesis. EECS Department, University of California, Berkeley, 1989.

[173] Sergey V. Rylov. “Clockless Dynamic SFQ and Gate With High Input Skew Tolerance”. In: *IEEE Transactions on Applied Superconductivity* 29.5 (2019), pp. 1–5.

[174] Ro Saito, Christopher L Ayala, and Nobuyuki Yoshikawa. “Buffer reduction via N-phase clocking in adiabatic quantum-flux-parametron benchmark circuits”. In: *IEEE Trans. Appl. Supercond.* 31.6 (2021), pp. 1–8.

[175] Tsutomu Sasao and Masahira Fujita. “Representations of Logic Functions Using EXOR Operators”. In: *Representation of Discrete Functions.* Springer, 1996.

[176] H. Savoj, R.K. Brayton, and H.J. Touati. “Extracting local don’t cares for network optimization”. In: *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers.* 1991, pp. 514–517.

[177] B. Schmitt, A. Mishchenko, and R. Brayton. “SAT-based area recovery in structural technology mapping”. In: *Proc. ASP-DAC.* 2018, pp. 586–591.

[178] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho W. Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. “SIS : A System for Sequential Circuit Synthesis”. In: *Technical Report UCB/ERI, M92/41, ERL* (1992).

[179] K. L. Shepard, S. M. Carey, E. K. Cho, B. W. Curran, R. F. Hatch, D. E. Hoffman, S. A. McCabe, G. A. Northrop, and R. Seigler. “Design methodology for the S/390 Parallel Enterprise Server G4 microprocessors”. In: *IBM Journal of Research and Development* 41.4.5 (1997), pp. 515–547.

[180] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. “Busy man’s synthesis: Combinational delay optimization with SAT”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* 2017, pp. 830–835.

[181] Mathias Soeken, Winston Haaswijk, Eleonora Testa, Alan Mishchenko, Luca G. Amarù, Robert K. Brayton, and Giovanni De Micheli. "Practical exact synthesis". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 309–314.

[182] Mathias Soeken, Alan Mishchenko, Ana Petkovska, Baruch Sterin, Paolo Ienne, Robert K. Brayton, and Giovanni De Micheli. "Heuristic NPN Classification for Large Functions Using AIGs and LEXSAT". In: *Theory and Applications of Satisfiability Testing*. Ed. by Nadia Creignou and Daniel Le Berre. 2016.

[183] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, and Giovanni De Micheli. "The EPFL Logic Synthesis Libraries". In: *CoRR* arXiv:1805.05121v3 (2022). eprint: arXiv:1805.05121v3.

[184] F. Somenzi and R. K. Brayton. "Minimization of Boolean relations". In: *IEEE International Symposium on Circuits and Systems* (1989), pp. 738–743.

[185] T. Stanion and C. Sechen. "Quasi-algebraic decompositions of switching functions". In: *in Advanced Res. VLSI*. 1995, pp. 358–367.

[186] L. Stok, M.A. Iyer, and A.J. Sullivan. "Wavefront technology mapping". In: *Proc. DATE*. 1999, pp. 531–536.

[187] *Suny RSFQ Cell Library*. http://www.physics.sunysb.edu/Physics/RSFQ/Lib/contents.html.

[188] Ivan E. Sutherland and Robert F. Sproull. "The theory of logical effort: designing for speed on the back of an envelope". In: *Advanced Research in VLSI*. 1991.

[189] Naoki Takeuchi, Shuichi Nagasawa, Fumihiro China, Takumi Ando, Mutsuo Hidaka, Yuki Yamanashi, and Nobuyuki Yoshikawa. "Adiabatic quantum-flux-parametron cell library designed using a 10 kA cm^{-2} niobium fabrication process". In: *Superconductor Science and Technology* 30.3 (2017), p. 035002.

[190] Naoki Takeuchi, Mai Nozoe, Yuxing He, and Nobuyuki Yoshikawa. "Low-latency adiabatic superconductor logic using delay-line clocking". In: *Applied Physics Letters* 115.7 (2019). eprint: https://doi.org/10.1063/1.5111599.

[191] Naoki Takeuchi, Dan Ozawa, Yuki Yamanashi, and Nobuyuki Yoshikawa. "An adiabatic quantum flux parametron as an ultra-low-power logic device". In: *Superconductor Science and Technology* 26.3 (2013).

[192] Naoki Takeuchi, Yuki Yamanashi, and Nobuyuki Yoshikawa. "Adiabatic quantum-flux-parametron cell library adopting minimalist design". In: *Journal of Applied Physics* 117.17 (2015), p. 173912.

[193] M. Tanaka, A. Kitayama, T. Koketsu, M. Ito, and A. Fujimaki. "Low-Energy Consumption RSFQ Circuits Driven by Low Voltages". In: *IEEE Transactions on Applied Superconductivity* 23.3 (2013), pp. 1701104–1701104.

[194] Alessandro Tempia Calvino and Giovanni De Micheli. "Algebraic and Boolean Methods for SFQ Superconducting Circuits". In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2024, pp. 588–593.

[195] Alessandro Tempia Calvino and Giovanni De Micheli. "Depth-Optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits". In: *Proceedings of ASP-DAC*. 2023, pp. 152–158.

[196] Alessandro Tempia Calvino and Giovanni De Micheli. "Scalable Logic Rewriting Using Don't Cares". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6.

[197] Alessandro Tempia Calvino and Giovanni De Micheli. "Technology Mapping Using Multi-Output Library Cells". In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 2023, pp. 1–9.

[198] Alessandro Tempia Calvino, Giovanni De Micheli, Alan Mishchenko, and Robert Brayton. "Enhancing Delay-driven LUT Mapping with Boolean Decomposition". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[199] Alessandro Tempia Calvino, Alan Mishchenko, Giovanni De Micheli, and Robert Brayton. *Practical Boolean Decomposition for Delay-driven LUT Mapping*. 2024. arXiv: 2406.06241 `[cs.LO]`.

[200] Alessandro Tempia Calvino, Alan Mishchenko, Herman Schmit, Ethan Mahintorabi, Giovanni De Micheli, and Xiaoqing Xu. "Improving Standard-Cell Design Flow using Factored Form Optimization". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6.

[201] Alessandro Tempia Calvino, Heinz Riener, Shubham Rai, and Giovanni De Micheli. "From Logic to Gates: A Versatile Mapping Approach to Restructure Logic". In: *Proc. IWLS*. 2021.

[202] Alessandro Tempia Calvino, Heinz Riener, Shubham Rai, Akash Kumar, and Giovanni De Micheli. "A Versatile Mapping Approach for Technology Mapping and Graph Optimization". In: *ASP-DAC*. 2022.

[203] Alessandro Tempia Calvino, Xiaoqing Xu, and Schmit Herman. "Transistor-level synthesis". US20240169134A1. 2024.

[204] Eleonora Testa, Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. "Algebraic and Boolean Optimization Methods for AQFP Superconducting Circuits". In: *Proc. ASP-DAC*. 2021.

[205] Eleonora Testa, Mathias Soeken, Luca Amarù, and Giovanni De Micheli. "Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications". In: *2019 56th ACM/IEEE DAC*. 2019.

[206] Hervé Touati. "Logic Synthesis for VLSI Design". PhD thesis. EECS Department, University of California, Berkeley, 1989.

[207] Naoki Tsuji, Yuki Yamanashi, Naoki Takeuchi, Christopher Ayala, and Nobuyuki Yoshikawa. “Design and implementation of scalable register files using adiabatic quantum flux parametron logic”. In: *Proceedings of ISEC*. 2017.

[208] Navin Vemuri, Priyank Kalla, and Russell Tessier. “BDD-based logic synthesis for LUT-based FPGAs”. In: *ACM Trans. Des. Autom. Electron. Syst.* 7.4 (2002), pp. 501–525.

[209] T. Villa and A. Sangiovanni-Vincentelli. “NOVA: state assignment of finite state machines for optimal two-level logic implementation”. In: *Trans. CAD* 9.9 (1990), pp. 905–924.

[210] Feng Wang, Liren Zhu, Jiaxi Zhang, Lei Li, Yang Zhang, and Guojie Luo. “Dual-Output LUT Merging during FPGA Technology Mapping”. In: *Proc. ICCAD*. Virtual Event, USA, 2020.

[211] Claire Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/.

[212] Qiuyun Xu, Christopher L Ayala, Naoki Takeuchi, Yuki Murai, Yuki Yamanashi, and Nobuyuki Yoshikawa. “Synthesis flow for cell-based adiabatic quantum-flux-parametron structural circuit generation with HDL back-end verification”. In: *IEEE Trans. Appl. Supercond.* 27.4 (2017), pp. 1–5.

[213] Xiaoqing Xu, Herman Schmit, and Alessandro Tempia Calvino. “Auto-creation of custom standard cells”. US20240176943A1. 2024.

[214] Congguang Yang and M. Ciesielski. “BDS: a BDD-based logic optimization system”. In: *IEEE Trans. CAD* 21.7 (2002), pp. 866–876.

[215] S. Yang and M.J. Ciesielski. “Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization”. In: *Trans. CAD* 10.1 (1991), pp. 4–12.

[216] W. Yang, L. Wang, and A. Mishchenko. “Lazy Man’s Logic Synthesis”. In: *Proc. ICCAD*. 2012, pp. 597–604.

[217] S Yorozu, Y Kameda, H Terai, A Fujimaki, T Yamada, and S Tahara. “A single flux quantum standard logic cell library”. In: *Physica C: Superconductivity* 378-381 (2002), pp. 1471–1474.

[218] N. Yoshikawa, D. Ozawa, and Y. Yamanashi. “Ultra-Low-Power Superconducting Logic Devices Using Adiabatic Quantum Flux Parametron”. In: *Extended Abstracts of the International Conference on Solid State Devices and Materials*. 2011.

[219] Mingfei Yu, Dewmini Sudara Marakkalage, and Giovanni De Micheli. “Garbled Circuits Reimagined: Logic Synthesis Unleashes Efficient Secure Computation”. In: *Cryptography* 7.4 (2023).

[220] Jianyong Yuan, Peiyu Wang, Junjie Ye, Mingxuan Yuan, Jianye Hao, and Junchi Yan. “EasySO: exploration-enhanced Reinforcement Learning for Logic Synthesis Sequence Optimization and a Comprehensive RL Environment”. In: *ICCAD*. 2023, pp. 1–9.

Alessandro TEMPIA CALVINO

Ph.D. Candidate

1020 Renens - Switzerland
✉ *alessandro.tempiacalvino@epfl.ch*
🌐 *https://aletempiac.github.io*
in *alessandro-tempia-calvino*

My research interests include electronic design automation, logic synthesis, algorithms, new data structures, verification, digital design, and emerging technologies.

Education

Sep 2020 pres	**Doctor of Philosophy - Ph.D. in Computer and Communication Sciences**, *EPFL (École Polytechnique Fédérale de Lausanne)*, Lausanne, Switzerland *Thesis advisor: Prof. Giovanni De Micheli*
Sep 2018 Sep 2020	**Master of Science in Computer Engineering**, *Télécom Paris*, EURECOM - Biot, France *Specialization in Smart Objects - joint master program with Politecnico di Torino*, **GPA 4.0/4**
Sep 2017 Mar 2020	**Master of Science in Computer Engineering**, *Politecnico di Torino*, Torino, Italy *Specialization in Embedded Systems - joint master program with Télécom Paris*, **Full marks with honor (110 cum laude/110)**
Sep 2014 Sep 2017	**Bachelor's Degree in Computer Engineering**, *Politecnico di Torino*, Torino, Italy

Experience

Sep 2020 pres	**Doctoral Researcher**, *EPFL (École Polytechnique Fédérale de Lausanne)*, Lausanne, Switzerland Integrated Systems Laboratory - Electronics design automation (EDA), logic synthesis, and emerging technologies.
Jun 2022 Oct 2022	**Research Intern**, *X, the Moonshot Factory (Google X)*, Mountain View, USA Logic synthesis and EDA.
Mar 2020 Aug 2020	**R&D Engineer**, *Télécom Paris*, EURECOM - Biot, France Implementation of a model-checker for embedded system models.
Jul 2019 Dec 2019	**Intern**, *Synopsys*, Montbonnot-Saint-Martin, France Implementation of timing-driven algorithms for the synthesis of digital circuits.

Publications

Conference and Workshop Papers:

- A. Costamagna, **A. Tempia Calvino**, A. Mishchenko, G. De Micheli, "Area-Oriented Optimization After Standard-Cell Mapping", in Asian and South Pacific Design Automation Conference (ASP-DAC), *accepted*, 2025.
- M. Yu, **A. Tempia Calvino**, M. Soeken, G. De Micheli, "Back-end-aware Fault-tolerant Quantum Oracle Synthesis", in Asian and South Pacific Design Automation Conference (ASP-DAC), *accepted*, 2025.
- M. Yu, S. Carpov, **A. Tempia Calvino**, G. De Micheli, "On the Synthesis of High-performance Homomorphic Boolean Circuits", in Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC), *accepted*, 2024.
- S.-Y. Lee, **A. Tempia Calvino**, H. Riener, G. De Micheli, "Late Breaking Results: Majority-Inverter Graph Minimization by Design Space Exploration", in Design Automation Conference (DAC), 2024.
- **A. Tempia Calvino**, G. De Micheli, A. Mishchenko, R. Brayton, "Practical Boolean Decomposition for Delay-driven LUT Mapping", in International Workshop on Logic & Synthesis (IWLS), 2024.
- A. Costamagna, **A. Tempia Calvino**, A. Mishchenko, G. De Micheli, "Post-Mapping Resubstitution For Area-Oriented Optimization", in International Workshop on Logic & Synthesis (IWLS), 2024.
- A. Costamagna, **A. Tempia Calvino**, A. Mishchenko, G. De Micheli, "Area-Oriented Resubstitution For Networks of Look-Up Tables", in International Workshop on Logic & Synthesis (IWLS), 2024.

- **A. Tempia Calvino**, G. De Micheli, "Scalable Logic Rewriting Using Don't Cares", in Design Automation and Test in Europe Conference (DATE), 2024.
- R. Bairamkulov, S.-Y. Lee, **A. Tempia Calvino**, D. Marakkalage, M. Yu, G. De Micheli, "Technology-Aware Logic Synthesis for Superconductive Electronics", in Design Automation and Test in Europe Conference (DATE), 2024.
- **A. Tempia Calvino**, G. De Micheli, "Algebraic and Boolean Methods for SFQ Superconducting Circuits", in Asian and South Pacific Design Automation Conference (ASP-DAC), 2024.
- G. Radi, **A. Tempia Calvino**, G. De Micheli, "In Medio Stat Virtus: Combining Boolean and Pattern Matching", in Asian and South Pacific Design Automation Conference (ASP-DAC), 2024.
- **A. Tempia Calvino**, G. De Micheli, "Technology Mapping Using Multi-output Library Cells", in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2023.
- R. Bairamkulov, **A. Tempia Calvino**, G. De Micheli, "Synthesis of SFQ Circuits with Compound Gates", in 2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC), 2023.
- **A. Tempia Calvino**, A. Mishchenko, H. Schmit, E. Mahintorabi, G. De Micheli, X. Xu, "Improving Standard-Cell Design Flow using Factored Form Optimization", in ACM/IEEE Design Automation Conference (DAC), 2023.
- **A. Tempia Calvino**, G. De Micheli, "Technology Mapping Using Multi-output Library Cells", in International Workshop on Logic & Synthesis (IWLS), 2023.
- A. Mishchenko, R. Brayton, **A. Tempia Calvino**, G. De Micheli, "Boolean Decomposition Revisited", in International Workshop on Logic & Synthesis (IWLS), 2023.
- **A. Tempia Calvino**, G. De Micheli, "Depth-Optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits", in Asian and South Pacific Design Automation Conference (ASP-DAC), 2023.
- **A. Tempia Calvino**, G. De Micheli, "Depth-optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits", in International Workshop on Logic & Synthesis (IWLS), 2022.
- G. Meuli, V. Possani, R. Singh, S.-Y. Lee, **A. Tempia Calvino**, D. Marakkalage, P. Vuillod, L. Amarù, S. Chase, J. Kawa, and G. De Micheli, "Majority-based design flow for AQFP superconducting family", in Design Automation and Test in Europe Conference (DATE), 2022.
- L. Apvrille, P. de Saqui-Sannes, O. Hotescu, **A. Tempia Calvino**, "SysML models verification relying on dependency graphs", in International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2022.
- **A. Tempia Calvino**, H. Riener, S. Rai, G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization", in Asian and South Pacific Design Automation Conference (ASP-DAC), 2022.
- **A. Tempia Calvino**, H. Riener, S. Rai, G. De Micheli, "From logic to gates: A versatile mapping approach to restructure logic", in International Workshop on Logic & Synthesis (IWLS), 2021.
- **A. Tempia Calvino**, L. Apvrille, "Direct model-checking of SysML models", in International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2021.

Journal Papers and Book Chapters:

- **A. Tempia Calvino**, G. De Micheli, A Mishchenko, R. Brayton, "Enhancing Delay-driven LUT Mapping with Boolean Decomposition", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2024.
- Andrea Costamagna, **A. Tempia Calvino**, A Mishchenko, G. De Micheli, "Area-Oriented Resubstitution For Networks of Look-Up Tables", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), *under review*, 2024.
- S.-Y. Lee, **A. Tempia Calvino**, G. De Micheli, "Technology Legalization and Optimization for Adiabatic Quantum-Flux Parametron", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2024.
- R. Bairamkulov, **A. Tempia Calvino**, G. De Micheli, "Synthesis of SFQ Circuits with Compound Gates", in VLSI-SoC 2023: Silicon Innovations for Trustworthy Artificial Intelligence, Springer, 2024.
- L. Apvrille, P. Saqui-Sannes, O.A. Hotescu, **A. Tempia Calvino**, "Dependency Graphs to Boost the Verification of SysML Models", in Model-Driven Engineering and Software Development, MODELSWARD 2021-2022, Springer, 2023.
- S. Rai, **A. Tempia Calvino**, H. Riener, G. De Micheli, A. Kumar, "Utilizing XMG-based Synthesis to Preserve Self-duality for RFET-based Circuits", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.

Preprints:

- M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, **A. Tempia Calvino,** D. S. Marakkalage, G. De Micheli, "The EPFL logic synthesis libraries", arXiv preprint arXiv:1805.05121v3, 2022.

Patents

- **A. Tempia Calvino**, Xiaoqing Xu, Herman Schmit, "Transistor-level synthesis", US 18462628, 2024.
- Xiaoqing Xu, Herman Schmit, **A. Tempia Calvino**, "Auto-creation of custom standard cells", US 18235437, 2024.

Invited and Conference Talks

Invited Talks:

- **A. Tempia Calvino**, "Technology-aware logic synthesis for superconducting electronics", in International Workshop on Quantum, Cryogenic and Superconductive Computing, Fukuoka, Japan, 2024.
- **A. Tempia Calvino**, "The EPFL Logic Synthesis Libraries: open-source tools for classical and emerging technologies", in Free Silicon Conference (FSiC), Paris, France, 2024.
- **A. Tempia Calvino**, "EPFL Benchmark Results Update", in International Workshop on Logic & Synthesis (IWLS), Zurich, Switzerland, 2024.
- **A. Tempia Calvino**, "Improving Technology Mapping for Standard Cells", at IBM Thomas J. Watson Research Center (online), 2024.
- **A. Tempia Calvino**, "Improving Delay-driven LUT Mapping with Boolean Decomposition", at Efinix Inc. (online), 2024.
- **A. Tempia Calvino**, "Improving Delay-driven LUT Mapping with Boolean Decomposition", at AMD Inc. (Vivado team) (online), 2023.
- **A. Tempia Calvino**, "Technology Mapping Using Multi-output Library Cells", at Google X (online), 2023.
- **A. Tempia Calvino**, "Improving Standard-Cell Design Flow using Factored Form Optimization", at Cadence Design Systems Inc., San Jose (CA), USA, 2023.
- **A. Tempia Calvino**, "EPFL Benchmark Results Update", in International Workshop on Logic & Synthesis (IWLS), Lausanne, Switzerland, 2023.
- **A. Tempia Calvino**, "EPFL Benchmark Results Update", in International Workshop on Logic & Synthesis (IWLS) (online), 2022.
- **A. Tempia Calvino**, "EPFL Benchmark Results Update", in International Workshop on Logic & Synthesis (IWLS) (online), 2021.

Conference Talks:

- **A. Tempia Calvino**, "Practical Boolean Decomposition for Delay-driven LUT Mapping", in International Workshop on Logic & Synthesis (IWLS), Zurich, Switzerland, 2024.
- **A. Tempia Calvino**, "Area-Oriented Resubstitution For Networks of Look-Up Tables", in International Workshop on Logic & Synthesis (IWLS), Zurich, Switzerland, 2024.
- **A. Tempia Calvino**, "Scalable Logic Rewriting Using Don't Cares", in Design Automation and Test in Europe Conference (DATE), Valencia, Spain, 2024.
- **A. Tempia Calvino**, "Algebraic and Boolean Methods for SFQ Superconducting Circuits", in Asian and South Pacific Design Automation Conference (ASP-DAC), Incheon, South Korea, 2024.
- **A. Tempia Calvino**, "In Medio Stat Virtus: Combining Boolean and Pattern Matching", in Asian and South Pacific Design Automation Conference (ASP-DAC), Incheon, South Korea, 2024.
- **A. Tempia Calvino**, "Technology Mapping Using Multi-output Library Cells", in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Francisco (CA), USA, 2023.
- **A. Tempia Calvino**, "Improving Standard-Cell Design Flow using Factored Form Optimization", in ACM/IEEE Design Automation Conference (DAC), San Francisco (CA), USA, 2023.
- **A. Tempia Calvino**, "Technology Mapping Using Multi-output Library Cells", in International Workshop on Logic & Synthesis (IWLS), Lausanne, Switzerland, 2023.
- **A. Tempia Calvino**, "Depth-Optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits", in Asian and South Pacific Design Automation Conference (ASP-DAC), Tokyo, Japan, 2023.
- **A. Tempia Calvino**, "Depth-optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits", in International Workshop on Logic & Synthesis (IWLS), online, 2023.

- **A. Tempia Calvino**, "A versatile mapping approach for technology mapping and graph optimization", in Asian and South Pacific Design Automation Conference (ASP-DAC), online, 2022.
- **A. Tempia Calvino**, "From Logic to Gates: A Versatile Mapping Approach to Restructure Logic", in International Workshop on Logic & Synthesis (IWLS), online, 2021.

Honors and Awards

- **O-1** U.S. work visa - *individuals with an extraordinary ability in sciences.*
- Best Student Paper Award at International Workshop on Logic & Synthesis 2024 (IWLS) for the paper "Area-Oriented Resubstitution For Networks of Look-Up Tables".
- Best Student Paper Nomination at International Workshop on Logic & Synthesis 2024 (IWLS) for the paper "Practical Boolean Decomposition for Delay-driven LUT Mapping".
- Best paper award at 2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration System-on-Chip (VLSI-SoC) for the paper "Synthesis of SFQ Circuits with Compound Gates".
- First place in the International Workshop on Logic & Synthesis (IWLS) Contest 2022: "Synthesis of small circuits for completely-specified multi-output Boolean functions represented using truth table".
- Best Student Paper Candidate at International Workshop on Logic & Synthesis 2021 (IWLS) for the paper "From Logic to Gates: A Versatile Mapping Approach to Restructure Logic".
- Best Poster Award at International Conference on Model-Driven Engineering and Software Development (MODELSWARD) 2021 for the paper "Direct Model-checking of SysML Models".
- EDIC I&C Ph.D. Fellowship, EPFL, 2020.

Teaching Assistantships

- Design Technologies for Integrated Systems, M.Sc. course, Fall 2023, EPFL.
- Digital System Design, B.Sc. course, Spring 2023, EPFL.
- Real time embedded systems, M.Sc. course, Spring 2022, EPFL.
- Design Technologies for Integrated Systems, M.Sc. course, Fall 2021, EPFL.
- Real time embedded systems, M.Sc. course, Spring 2021, EPFL.
- Object-Oriented Programming, B.Sc. course, Spring 2018, Politecnico di Torino.
- Algorithms and Programming in C, B.Sc. course, Fall 2017, Politecnico di Torino.

Professional service

Software Development of *Mockturtle*: an open-source logic synthesis library, *available at*: https://github.com/lsils/mockturtle

Service Contributing and maintaining open-source software. Maintaining the EPFL logic synthesis libraries, *available at*: https://github.com/lsils/lstools-showcase and contributing to the logic synthesis tool ABC, *available at*: https://github.com/berkeley-abc/abc.

Service Maintainer of the EPFL Combinational Benchmark Suite and its competition. New updates and best results are presented annually at the International Workshop on Logic & Synthesis (IWLS). *Available at*: https://github.com/lsils/benchmarks.

Reviewer Reviewed for several top-tier conferences and journals such as TCAD, TODAES, ICCAD, DATE, DAC, IWLS, DDECS, and ISVLSI.

Member Member of IEEE, ACM, and IEEE Young Professionals

Technical and Personal Skills

Domains of expertise

Logic synthesis, electronic design automation, digital design, microelectronics, algorithms, data structures, computer architectures.

Programming Languages

C, C++, VHDL, Verilog, Java, Python, TCL, Bash, LaTeX, and more.

Languages

- English, Italian, French