UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

JODY MAICK ARAUJO DE MATOS

Graph-Based Algorithms for Transistor Count Minimization in VLSI Circuit EDA Tools

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

Advisor: Prof. Dr. André Reis
Coadvisor: Prof. Dr. Renato Ribas

Porto Alegre
March 2014

CIP — CATALOGING-IN-PUBLICATION

Matos, Jody Maick Araujo de

Graph-Based Algorithms for Transistor Count Minimization in VLSI Circuit EDA Tools / Jody Maick Araujo de Matos. – Porto Alegre: PGMICRO da UFRGS, 2014.

85 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2014. Advisor: André Reis; Coadvisor: Renato Ribas.

1. Benchmark circuits. 2. Transistor count. 3. Logic synthesis. 4. Technology mapping. I. Reis, André. II. Ribas, Renato. III. Título.

"I may speak in the tongues of men,
even angels; but if I lack love, I have become
merely blaring brass or a cymbal clanging."
— 1 Corinthians 13:1

To my beloved wife, Joseane. Thank you
for teaching me the true meaning of love.

ACKNOWLEDGMENT

Thanks, God, for giving me the strength and health to successfully complete this journey of my life. None of this would be possible if it was not Thy will. *"For from Him and through Him and for Him are all things"* (Romans 11:36).

I would like to express my gratitude to those that contributed to this achievement. I get this way by thanking my wife, Joseane. Thank you very much for the unconditional support, no matter the circumstances, for your patience, love and partnership. We are one. You own this achievement as well. I love you!

I am also very thankful to my parents. It is very good to count on you. Thank you for supporting me, no matter what.

I have special thanks to my advisor, Prof. André Reis. It is an honor for me to have you as my advisor. I am really thankful for such valuable lessons about both intellectual matters, life, the Universe and everything. Your brilliant technical observations also have helped me to shape this work. Thanks for guiding me in such a good way. Life is beautiful!

I also would like to thank my coadvisor, Prof. Renato Ribas, and to LogiCS Labs. The experience of working with you was very important to me. Thank you for your useful advises during this work.

My thanks to the defense jury, Prof. Jordi Carrabina (UAB, Barcelona), Prof. Marcelo Johann and Prof. Sergio Bampi. Thank you all for your time and for contributing with this work. Thanks to CAPES for financial support.

Lastly, I am grateful to all other teachers/professors who are part of my education, since the first classes until today. Education will always be the best game changing resource.

ABSTRACT

This master's thesis introduces a set of graph-based algorithms for obtaining reduced transistor count VLSI circuits using simple cells. These algorithms are mainly focused on minimizing node count in AIG representations and mapping this optimized AIG using simple cells (NAND2 and NOR2) with a minimal number of inverters. Due to the AIG node count minimization, the logic sharing is probably highly present in the optimized AIG, what may derive intermediate circuits containing cells with unfeasible fanout in current technology nodes. In order to fix these occurrences, this intermediate circuit is subjected to an algorithm for fanout limitation. The proposed algorithms were applied over a set of benchmark circuits and the obtained results have shown the usefulness of the method. The circuits generated by the methods proposed herein have, in average, 32% less transistor than the previous reference on transistor count using simple cells. Additionally, when comparing the presented results in terms of transistor count against works advocating for complex cells, our results have demonstrated that previous approaches are sometimes far from the minimum transistor count that can be obtained with the efficient use of a reduced cell library composed by only a few number of simple cells. The simple-cells-based circuits obtained after applying the algorithms proposed herein have presented a lower transistor count in many cases when compared to previously published results using complex (static CMOS and PTL) cells.

Keywords: Benchmark circuits. transistor count. logic synthesis. technology mapping.

Algoritmos Baseados em Grafos para Minimização de Transistores em Ferramentas EDA para Circuitos VLSI

RESUMO

Esta dissertação de mestrado introduz um conjunto de algoritmos baseados em grafos para a obtenção de circuitos VLSI com um número reduzido de transistores utilziando células simples. Esses algoritmos têm um foco principal na minimização do número de nodos em representações AIG e mapear essa estrutura otimizada utilizando células simples (NAND2 e NOR2) com um número mínimo de inversores. Devido à minimização de nodos, o AIG tem um alto compartilhamento lógico, o que pode derivar circuitos intermediários contendo células com *fanouts* infactíveis para os nodos tecnológicos atuais. De forma a resolver essas ocorrências, o circuito intermediário é submetido a um algoritmo para limitação de *fanout*. Os algoritmos propostos foram aplicados num conjunto de circuitos de *benchmark* e os resultados obtidos mostram a utilidade do método. Os circuitos resultantes tiveram, em média, 32% menos transistores do que as referências anteriores em números de transistores utilizando células simples. Adicionalmente, quando comparando esses resultados com trabalhos que utilizam células complexas, nossos números demonstraram que abordagens anteriores estão algumas vezes longe do número mínimo de transistores que pode ser obtido com o uso eficiente de uma biblioteca reduzida de células, composta por poucas células simples. Os circuitos baseados em células simples obtidos com a aplicação dos algoritmos proposto neste trabalho apresentam um menor número de transistores em muitos casos quando comparados aos resultados previamente publicados utilizando células complexas (CMOS estático e PTL).

Palavras-chave: Circuitos de benchmark, número de transistores, síntese lógica, mapeamento tecnológico.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CMOS	Complementary Metal-Oxide-Semiconductor
DAG	Directed Acyclic Graph
EDA	Electrical Design Automation
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
LSI	Large Scale Integration
NRE	Non-Recurring Engineering
OTR	Odd-level Transistor Replacement
PI	Primary Input
PLA	Programmable Logic Array
PO	Primary Output
PTL	Pass-Transistor Logic
RTL	Register-Transfer Level
SOP	Sum-Of-Products
TSBDD	Terminal-Suppressed Binary Decision Diagram
VLSI	Very-Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

According to the International Technology Roadmap for Semiconductors, the semiconductor industry has achieved a distinctive position among other industries by the speed of improvements in its products during the last five decades (ITRS, 2012). Many of these improvements are mainly due to the industry's ability to exponentially decrease the minimum feature sizes used to fabricate integrated circuits (IC). Among the main trends which confirm this fact, such as microprocessors throughput, mobile devices battery life, and small and light-weighted products, the most frequently cited trend is the integration level, which is usually expressed as Moore's Law (MOORE, 1965). However, all these trends, sometimes called "scaling", have impact on the complexity while designing Very-Large Scale Integration (VLSI) circuits. In this sense, designing integrated circuits is becoming an increasingly hard task (ITRS, 2012; KAHNG et al., 2011).

The VLSI circuit designs demand the optimization of different (and sometimes divergent) cost functions. The most frequently cited cost functions entities in the sense of scaling are chip area, circuit delay, power consumption, testability and time-to-market (KAHNG et al., 2011; HACHTEL; SOMENZI, 1996; GEREZ, 1999). With high-level complexity while designing ICs, the use of Electronic Design Automation (EDA) tools become imperative to semiconductor industry. In this sense, The progress achieved on EDA tools' algorithms can be stated as another crucial factor to obtain the results achieved by the semiconductor industry during the last five decades (ITRS, 2012; KAHNG et al., 2011). Intrinsic optimization tasks of the design flow are modeled as computational problems, solved by EDA tools, and, due to the use of efficient EDA tools, integrated circuits with an improved final quality are produced (HACHTEL; SOMENZI, 1996; GEREZ, 1999).

This chapter is organized as follows. We will initially present a brief review of VLSI circuit design methods, to better explain where our contribution lies. After, we detail our motivation. Next, we present the objectives of this master's thesis. This chapter ends by detailing the organization of this master's thesis.

1.1 VLSI Circuit Design Methods

Usually, VLSI circuit design methods are classified into three main categories: *custom*, *semicustom* and *programmable* designs (GEREZ, 1999; MICHELI, 2003; WESTE;

HARRIS, 2009). In *custom* designs, both functional and physical designs are "hand-crafted". On the other hand, the *semicustom* design method consists in establishing design restrictions, such as limiting the number of primitives, when compared against custom designs. The semicustom design method limits the ability of optimization of a circuit, but it makes easier the development of EDA tools by the reduction on the solution space. The more degrees of freedom there are, the bigger the search space for the optimal design (GEREZ, 1999; MICHELI, 2003; WESTE; HARRIS, 2009).

The most known custom design method is called *full-custom* design. Full-custom designers must use graphic editor tools to draw the circuit layout one rectangle/polygon at time. In this case, the entire system is designed at mask level. A variation of this full-custom method is called *symbolic layout.* Rather than dealing with rectangles and polygons on various mask levels, the primitives are transistors, contacts, wires, and ports (points of connection) (GEREZ, 1999; WESTE; HARRIS, 2009).

Among the semicustom design methods, the most known is the *cell-based* design, in which instances of a *standard cell library* are used as the basic building block of the design. The design's logic structure is mapped into standard cells, which are placed in appropriate positions. After that, their interconnections are routed. This approach is classified as semicustom because the library vendors use a custom method to develop the cell library (GEREZ, 1999; MICHELI, 2003; WESTE; HARRIS, 2009). However, there are EDA vendors, such as NanGate Inc, that provide solutions for automatic library generation. There also exist approaches claiming to improve the performance of cell-based designs by using a library-free methodology (REIS et al., 1995; MARQUES et al., 2007).

The custom methodology requires highly skilled designers and a great effort in order to fine-tune features of the circuit. However, this high cost may be compensated by a high quality design. Cell-based designs can deliver smaller, faster, and lower-power chips than the programmable logic counterpart, such as Field-Programmable Gate Arrays (FPGA), but has high non-recurring engineering (NRE) costs to produce the custom mask set. When compared to full-custom design, cell-based design offers much higher productivity because it uses predesigned cells with layouts (GEREZ, 1999; MICHELI, 2003; WESTE; HARRIS, 2009).

Nowadays, the number of cell-based designs outnumbers custom designs. Digital complementary metal-oxide-semiconductor (CMOS) ICs are made using custom mask design only for the highest of volume parts, such as microprocessor datapaths. However, analog and RF designs, cell libraries, memories, and I/O cells still frequently use custom design (WESTE; HARRIS, 2009).

Our work will be centered in cell-based semicustom designs, which are the more used application specific integrated circuit (ASIC) design methodology. One of the focus is the constitution of a standard cell library. Next section presents a detailed discussion about the types of cells and how they constitute different kinds of libraries.

1.2 Motivation

In a cell-based VLSI circuit design, it is common to use a cell library that contains simple cells (e.g. AND, OR, NAND and NOR), and also may comprise some more complex cells (e.g. AOI and OAI cells) (REIS et al., 1995; PIGUET et al., 2001; SEO et al., 2008; RICCI; MUNARI; CIAMPOLINI, 2007; GAVRILOV et al., 1997). In the particular case of library-free approaches, the use of complex cells is explored more extensively (REIS et al., 1995; MARQUES et al., 2007). In such design methods, complex cells are usually obtained as transistor-level gate networks using series-parallel (REIS et al., 1995; GAVRILOV et al., 1997) or pass-transistor logic (PTL) topologies (YANBIN; SAPATNEKAR; BAMJI, 2001; SHELAR; SAPATNEKAR, 2005). Historically, studies about library-free approach have tried to demonstrate the advantages of using complex cells when compared to simple cells (REIS et al., 1995; GAVRILOV et al., 1997; YANBIN; SAPATNEKAR; BAMJI, 2001; SHELAR; SAPATNEKAR, 2005). The main advantage reported by those works is the reduction in the number of devices needed to implement digital ICs. Published results using complex cells claim reductions of the order of 40% in terms of transistor count when compared to simple-cell implementations. The reduced number of transistors is expected to have a positive impact on design cost functions, such as area and power consumption. Thus, it is related to the quality of the final circuits, and justifies the importance of obtaining reduced transistor count circuits.

It is important to remark that logic synthesis has evolved considerably in recent years. The most recent logic synthesis works are based on a type of data structure called and-inverter graphs (AIGs) (Berkeley Logic Synthesis and Verification Group, 2013; MISHCHENKO; CHATTERJEE; BRAYTON, 2006). The AIG data structure can be viewed as a graph composed of 2-input AND (AND2) nodes, connected by direct or negated edges (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). State-of-the-art logic synthesis tools, like ABC (Berkeley Logic Synthesis and Verification Group, 2013), minimize the number of (AND2) nodes in AIGs. This number can be directly correlated to the number of cells in implementations using 2-input simple cells, such as ANDs, ORs, NANDs and NORs. These cells are variants of the primitive AND2, obtained by applying De Morgan's law and inversions. Notice that De Morgan's law modifies the polarity of the input and output signals. AIGs with a minimized number of nodes can be used almost straightforwardly to generate minimal transistor count circuits based on simple (2-input) cells. Note also that 2-input NAND (NAND2) and NOR (NOR2) cells require four transistors in standard static CMOS topology, whereas 2-input AND (AND2) and OR (OR2) cells require six transistors. Therefore, it is possible to choose the polarity of internal nodes, such that NAND2 and NOR2 gates are preferred. This way, the transistor count may be reduced by choosing cells with a low number of transistors, whereas a few inverters are added.

Our motivation is to verify if the state-of-the-art tools that provide AIG node count minimization can derive competitive results in terms of transistor count using simple cells compared to previously published results using complex cells. The obtained circuits could be used to directly generate final implementations using simple cells, or even they could be resynthesized using complex cells, since the solution given by logic synthesis tools depends on the input circuit. For this, we need an efficient polarity assignment approach, to minimize inverters, such that the implementation from AIGs uses more NAND2 and NOR2 cells (which have cheaper transistor count costs compared to AND2 and OR2 cells).

1.3 Objective

The objective of this master's thesis is to introduce a set of algorithms for obtaining reduced transistor count for semicustom digital VLSI circuits based on simple cells. These algorithms are mainly focused on starting from minimized node count AIG representations, obtained through state-of-the-art logic synthesis tools, and mapping this optimized AIG using simple cells (NAND2 and NOR2) with a minimal number of inverters. This way, the problem is formulated as a polarity assignment and inverter minimization problem, in order to map optimized AIGs into NAND2/NOR2/INV implementations with minimum transistor count. A second aspect that we are addressing is that, due to the AIG node count minimization, the logic sharing is highly present in the optimized AIG, sometimes resulting in circuits containing some cells with unfeasible fanout in current technology nodes. In order to fix these occurrences, we propose an algorithm for fanout limitation, which is combined to the inverter minimization procedure. The presented algorithms and approaches are graph-based and they are variations and improvements of previous techniques of graph coloring for inverter minimization (JAIN; BRYANT, 1993), combined with current state-of-the-art logic synthesis tools, such as ABC (Berkeley Logic Synthesis and Verification Group, 2013), for minimizing the node count in the AIG representation of the circuit being synthesized.

Applying the proposed algorithms, this work also tries to demonstrate that the start-points reported by previous works in terms of transistor count using simple cells are far from optimal, mainly when considering current optimization techniques. We present better reference start-points and we also demonstrate that the gains obtained by prior approaches comparing implementations using complex cells against circuits using simple cells are not so significant (some gains even become losses) when measured against the start-points presented herein.

1.4 Master's Thesis Organization

The next chapters are organized as follows:

Chapter 2: *Technical Background* — reviews all basic and established knowledge that is needed to understand the concepts presented in this work, such as the integrated circuit design flow, the logic synthesis flow, common data structures, and the polarity assignment problem.

Chapter 3: *Logic Synthesis Review* — traces an evolutionary line over the logic synthesis, describing some previous and recent works that are connected to the work presented in this thesis.

Chapter 4: *Proposed Approach* — presents an overview of the proposed approach, providing examples and a general point of view in such a way to provide a better understanding of the proposed synthesis flow.

Chapter 5: *Reducing Transistor Count in Benchmark Circuits* — provides a detailed explanation of each step and substep of the proposed approach, as well as presents pseudocodes of the proposed algorithms.

Chapter 6: *Results* — presents and discusses the obtained results when running the proposed technique in a set of benchmark circuits, analyzing the influence of each proposed parameter in the final result.

Chapter 7: *Conclusions and Future Works* — provides the main conclusions, summarizes the contributions of this work and presents the intended future works.

2 TECHNICAL BACKGROUND

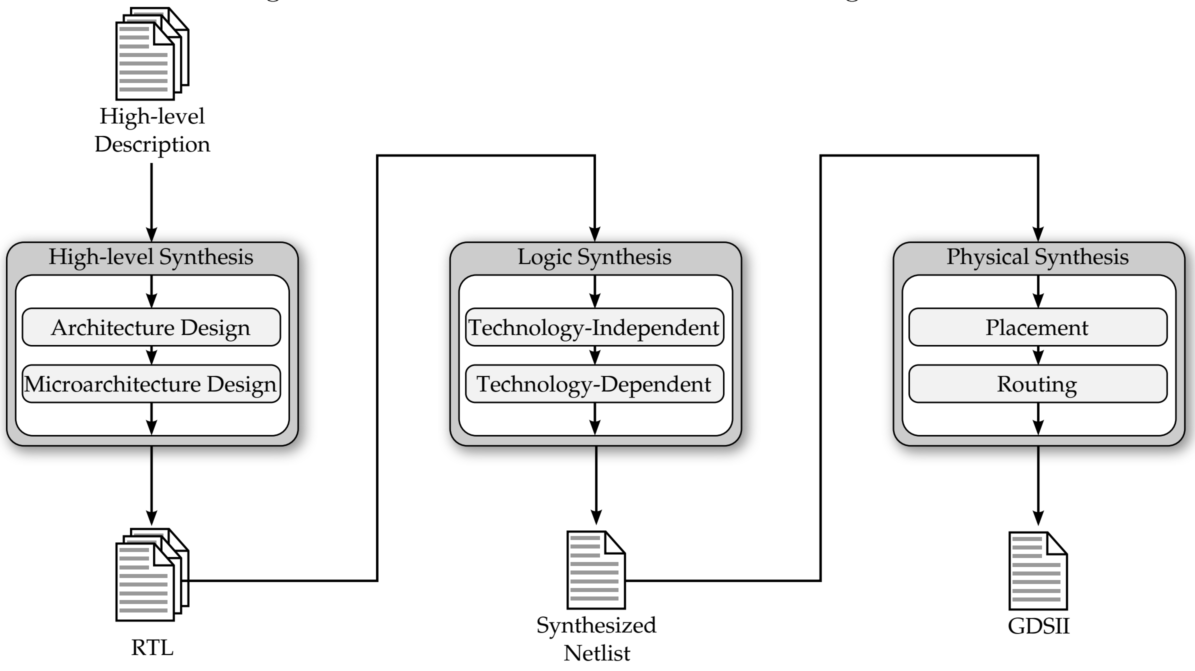
This chapter provides a basic technical background highlighting the concepts necessary to understand this work. Several topics, including the integrated circuit cell-based design flow, the logic synthesis design flow and its main data structures, and the polarity assignment problem are reviewed.

2.1 Integrated Circuits Synthesis

Historically, the VLSI circuits design flow is based on synthesis tasks. At the early steps of this flow, the design description is in a degree of abstraction that is always higher than the equivalent descriptions at the late stages. The sooner is the step, the greater the level of abstraction. The later is the step, the lower the level of abstraction (GEREZ, 1999; MICHELI, 2003).

The design of integrated circuits is often split into three major steps: high-level (or architectural) synthesis, logic synthesis and physical synthesis (GEREZ, 1999; MICHELI, 2003). Figure 2.1 depicts a simplified design flow, omitting some internal sub-tasks and verification tasks during the flow.

Figure 2.1: Conventional VLSI Circuits Design Flow.



Source: Author.

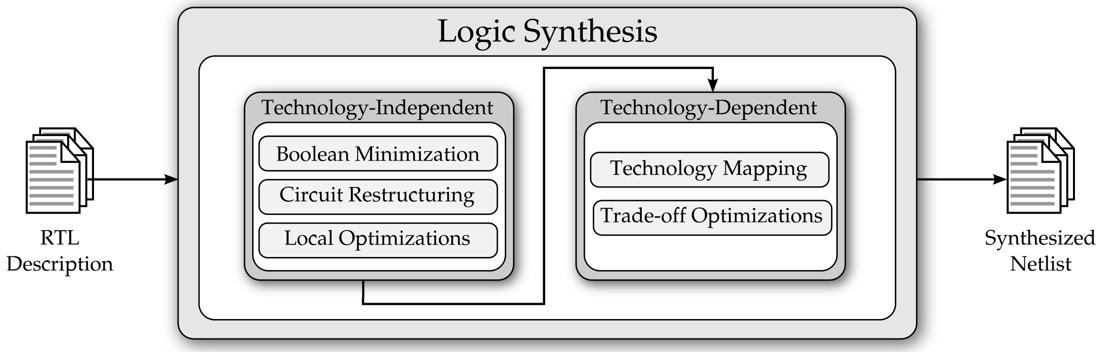
According to Gerez (1999), Micheli (2003) and Weste and Harris (2009), the high-level synthesis step transforms the architecture and microarchitecture system descriptions into an equivalent structural description in Register-Transfer Level (RTL). This RTL structural description must be synthesized into a netlist through technology-independent and technology-dependent optimizations, which are performed in the logic synthesis step. The resulting netlist, a description based on cell instances and their interconnections, shall be placed and routed in the physical synthesis step. The final product of the design flow is the circuit layout implementing the initial architectural and behavioral description of the system, typically represented in a GDSII file.

The work presented in this thesis is concerned to the logic synthesis step, specifically with the technology-independent optimizations. Our approach takes an and-inverter graph (AIG) as input, which could be obtained in the early stages of technology-independent optimizations, and produces an intermediate circuit comprised of simple cells. This intermediate circuit (with a minimized number of transistors) can be directly mapped to a netlist by instantiating cells from a simple standard cell library, or even it could be used as input of the technology-dependent optimizations. An initial circuit with a minimized number of transistors is expected to have a positive impact on design cost functions at later synthesis steps, such as area and power consumption.

2.2 Logic Synthesis Flow

Logic synthesis is the step of integrated circuit design flow that defines the internal structure of the logic used to implement a design. Current state-of-the-art logic synthesis tools are commonly described as tools to synthesize multi-level circuits, which have multiple levels of logic gates on any path between a primary input and a primary output (HACHTEL; SOMENZI, 1996; MICHELI, 2003). Logic synthesis has two main goals. The first objective is to perform an automatic translation of an RTL description, usually in a Hardware Description Language (HDL), into a netlist of interconnected logic gates instantiated from a library. Second, these tools try to optimize the resulting circuit in terms of typical cost functions, such as chip area, power consumption, critical path delay and the degree of testability; while satisfying the design constraints imposed by the designers. Design constraints can include the desired operation frequency, as well as different time requirements on different input/output paths (HACHTEL; SOMENZI, 1996; MICHELI, 2003). The logic synthesis step is commonly divided into two major stages: the technology-independent optimizations and the technology-dependent optimizations. Figure 2.2 depicts the logic synthesis flow for cell-based VLSI circuit designs. The following subsections detail both technology-independent and technology-dependent optimization stages

Figure 2.2: Logic Synthesis flow for cell-based VLSI circuit designs.



Source: Author.

2.2.1 Technology-Independent Optimizations

The main tasks performed in the technology-independent stage are Boolean minimizations, circuit restructuring and local optimizations. The cost functions that are optimized during this process are technology independent, e.g the number of nodes in an AIG, or the depth of the AIG in terms of number of nodes.

In this optimization stage, the logic structure is not based on standard cells, commonly refered to as a generic circuit. Due to this reason, it is said to be technology-independent.

2.2.2 Technology Mapping and Technology-Dependent Optimizations

In the technology-dependent stage, the generic circuit resulting is mapped using the standard cell library into a synthesized netlist. This mapping task is called *technology mapping*. During and after mapping, all the information from the characterized cells are used for obtaining trade-off optimizations, such as gate sizing, delay optimizations on critical paths, buffer insertion and fanout limiting. During this step, the cost functions are related to the final technology of fabrication, usually coming from a pre-characterized cell library containing detailed information on area, delay and power consumption for each cell, including the different sizes available in the library.

Technology mapping is the process by which the technology-independent logic circuit is implemented in terms of the logic elements available in a particular technology library of standard cells (LAVAGNO; SCHEFFER; MARTIN, 2010; MARQUES et al., 2009). Each logic element is associated with information about delay, area, and internal and external capacitances. The optimization problem is to find an implementation meeting some user defined constraints (as a target frequency) while minimize other cost functions, such as area and power consumption. This process is frequently described into three main phases: decomposition, matching and covering (LAVAGNO; SCHEFFER; MARTIN, 2010; MARQUES et al., 2009; CHATTERJEE, 2007). These phases can be presented as follows.

2.2.2.1 Decomposition

In this phase, the data structure to be used during the technology mapping is created. The representation of the initial circuit to be mapped as a data structure is called *subject description*. According to the used data structure, the subject description can be a subject tree or a subject graph. The specification of this new representation relies strongly on the mapping strategy adopted. Some approaches break the graph into trees. State-of-the-art technology mapping algorithms work on subject graphs.

An important issue during this phase is to ensure that each node of the subject description will have at least one match against the cells of the library. This is important to guarantee that the technology mapping process completes successfully.

2.2.2.2 Matching

In matching phase, the algorithms try to find a set of equivalences between the functionality in each node of the subject description and the cells in the technology library. Two main approaches are commonly used, described as follows (LAVAGNO; SCHEFFER; MARTIN, 2010; MARQUES et al., 2009; CHATTERJEE, 2007):

Pattern (or Structural) matching: identifies common structural patterns between portions of the subject graph and cells of the library. Most approaches reduce the structural matching problem to a graph isomorphism problem. Due to the reduced size of both graphs, the isomorphism problem (commonly intractable) become tractable (BRAYTON et al., 1987).

Boolean matching: performs the matching considering Boolean functions of the same equivalence class. It usually performs the matching using binary decision diagrams (BDD) by trying different variable orderings, until a matching is found. Boolean matching is computationally more expensive than structural matching, but it can lead to better results (MAILHOT; MICHELI, 1993).

2.2.2.3 Covering

Once the whole subject graph has been matched, the final phase of technology mapping needs to cover the entire logic network choosing among the matches which of them minimize the objective function. This function is often the total area, the worst case delay, the power consumption, or a composition of these (LAVAGNO; SCHEFFER; MARTIN, 2010; MARQUES et al., 2009; CHATTERJEE, 2007).

2.3 Logic Synthesis Data Structures

A logic design may be represented by an assortment of data structures in EDA tools. Each of them has its particular strengths and weakness, being more or less suitable to specific manipulations. The following subsections describe the most known data structures used in logic synthesis algorithms.

2.3.1 Binary Decision Diagram

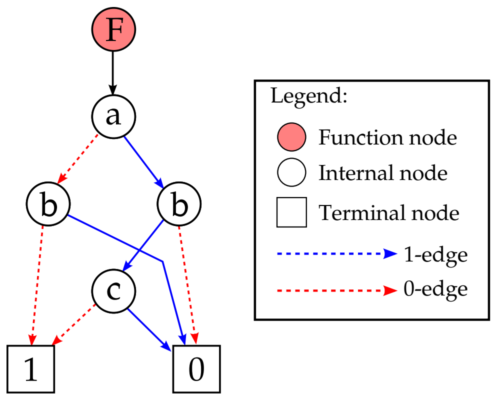
A *Binary Decision Diagram* (BDD) is a graph representation of Boolean functions. In this sense, a BDD represents a set of binary-valued decisions, culminating in an overall decision that could be *TRUE (1)* or *FALSE (0).* Though BDDs are relatively old (LEE, 1959; AKERS, 1978), they just began attracting the research community's attention when Bryant (1986) brought out their advantages as canonical representations (HACHTEL; SOMENZI, 1996; MICHELI, 2003).

Formally, a BDD is a directed acyclic graph (i.e., a directed graph with no directed cycles) with two terminal nodes, called *1-terminal* and *0-terminal*, which denote the *TRUE* and *FALSE* decision, respectively. The nodes of a BDD are partitioned into three subsets: function nodes Φ, internal nodes V, and the terminal nodes $\{0, 1\}$. A function node $\phi \in \Phi$ denotes the function being represented, has one outgoing edge and have no incoming edges. Each internal node $v \in V$ has a label $l(v) \in S_F$, where S_F denotes the *support* of a function F, i.e., each label represents a variable on which F actually depends. The internal nodes have two outgoing edges: the *0-edge*, which denotes the *FALSE* decision with respect to Source: Autor node of the edge; and the *1-edge*, which denotes the *TRUE* decision with respect to Source: Autor node of the edge (HACHTEL; SOMENZI, 1996; MICHELI, 2003; GEREZ, 1999). Figure 2.3 depicts the BDD (2.3(b)) and the truth table (2.3(a)) representing the function $F = ab\bar{c} + \bar{a}\bar{b}$.

Figure 2.3: Truth-table (a) and BDD (b) representing the function $F = ab\bar{c} + \bar{a}\bar{b}$.

a	**b**	**c**	**F**
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

(a)



(b)

Source: Author.

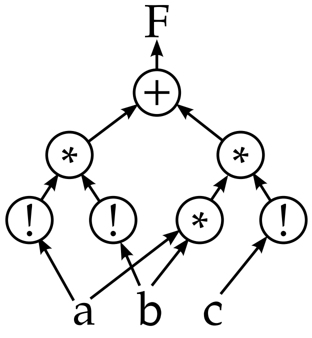
A serious issue with BDDs is the ordering. To obtains the canonical form of a BDD, the graph must be reduced and the variables in the support of the function being represented must be ordered (BRYANT, 1986; HACHTEL; SOMENZI, 1996; MICHELI, 2003; GEREZ, 1999). Unfortunately, the size of the BDD depends critically on the specific ordering chosen. In some cases, this dependence is so drastic that it is impractical to build the BDD at all. It is also known that BDD's size may grows exponentially with variable count, independent of the order. Considering $|S_F|$ as the number of variables in the support of the function F, the maximum number of nodes $|V|$ of a BDD is given by Equation 2.1. Thus, the use of BDDs become unpractical in some cases and limited by the number of variables.

$$|V| = \frac{2^{|S_F|}}{|S_F|} \tag{2.1}$$

2.3.2 Directed Acyclic Graph

In the context of logic synthesis, a data structure named directed acyclic graph (DAG) is a directed graph with no directed cycles that have three specific types of nodes: primary input (PI), primary output (PO) and logic operator nodes, such as AND and OR (HACHTEL; SOMENZI, 1996; MICHELI, 2003). It is important to observe that the name given to this particular data structure can be misleading, as directed acyclic graph is a name that can be used for a particular class of graphs with certain properties (WESTE; HARRIS, 2009; CORMEN et al., 2001). Given a node v, the set of edges arriving in v, $iedge(v)$, denotes the inputs of v. The set of edges leaving from v, $oedge(v)$, denotes the outputs of v. An example of DAG representing the function $F = ab\overline{c} + \overline{a}\overline{b}$ can be seen in Figure 2.4.

Figure 2.4: DAG representing the function $F = ab\overline{c} + \overline{a}\overline{b}$.



Source: Author.

Currently, the main application of DAGs in logic synthesis algorithms is on graph covering of technology mapping. However, due to the growth of the designed circuits nowadays, the use of DAGs tends to suffer of scalability problems and this data structure is being replaced AIG-based structures (LAVAGNO; SCHEFFER; MARTIN, 2010; CHATTERJEE, 2007).

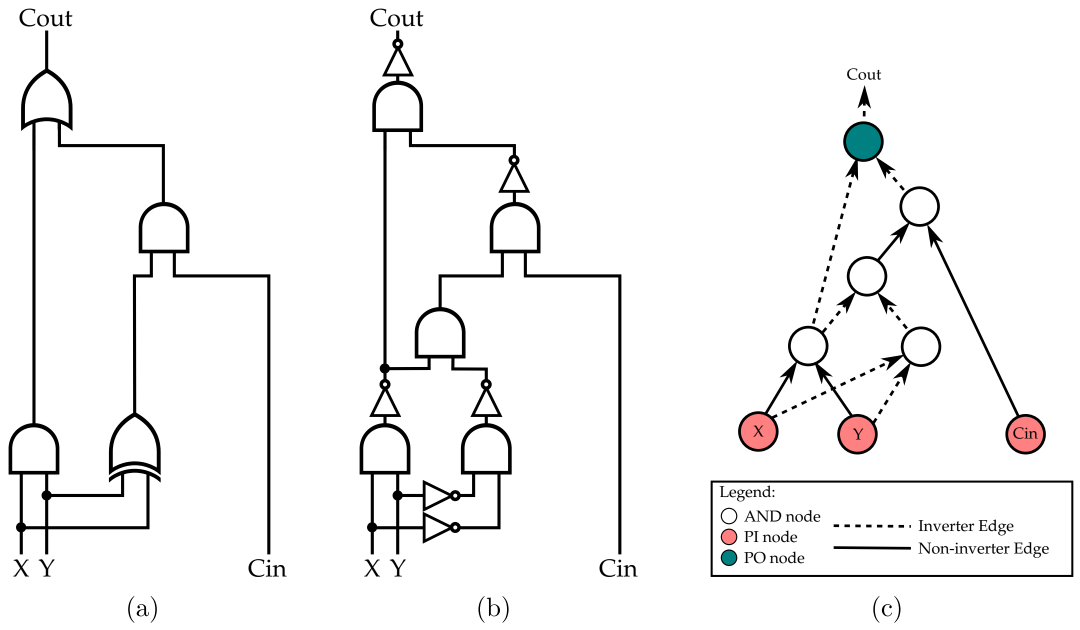
2.3.3 And-Inverter Graph

An and-inverter graph (AIG) is the data structure used in current state-of-the-art logic synthesis tools, like ABC (Berkeley Logic Synthesis and Verification Group, 2013). The name AIG is relatively recent. However, circuit transformations based on and-inverter representations are older than its name, including the works of Hellerman

(HELLERMAN, 1963) and Darringer (DARRINGER et al., 1981). A popular description format is the AIGER format (BIERE, 2007). AIGs are directed acyclic graphs (DAGs) with specific types of nodes: 2-input AND (AND2) nodes, primary input (PI) nodes, and primary output (PO) nodes. Primary input nodes have no incoming edges. AND2 nodes have two incoming edges. Any node of an AIG can be labeled as an output node (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

The edges of an AIG have a specific property: they are either in their positive or complemented form. A Boolean signal arriving at the target node via a positive edge has the same polarity as the Source: Autor node. The complemented form of an AIG edge indicates the Boolean inversion operation of its signal (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Figure 2.5 shows a possible AIG of the logic function $cout = (x \otimes y \cdot cin) + (x \cdot y)$.

Figure 2.5: A combinational circuit (a) and its representation with AND gates and inverters (b), which derives the AIG representation (c).



Source: Author.

Notice that state-of-the-art logic synthesis tools, like ABC (Berkeley Logic Synthesis and Verification Group, 2013), minimize the number of (AND2) nodes in AIGs. By adequately selecting the polarity of the signals in the inputs and outputs of an AIG node, it can be transformed into 2-input AND, OR, NAND and NOR cells. These cells are variants of the primitive AND2 AIG nodes, obtained by applying De Morgan's law. In order to obtain an implementation with minimum transistor count, NAND2 and NOR2 gates have to be preferred, because they have four transistors, compared to the six transistors needed for AND2 and OR2 cells.

2.4 Polarity Assignment Problem

2.4.1 Problem Definition

The polarity assignment problem is often referred to as the case in which the polarity (also called phase) of inputs and outputs (I/Os) of a combinational network can be changed in such a way that yields the signals and their complements. These changes are possible without modifying the functionality only when the I/Os starts and ends on registers or pads. Thus, it may be useful to search for matches with polarity assignments that reduce the cost of an objective function of interest (MICHELI, 2000).

Jain and Bryant (1993) have proposed a way to perform inverter minimization in multi-level logic networks composed of simple cells by applying a variation of the polarity assignment problem. Their approach is based on phase-constraint graphs for allowed cells in the library and a polarity graph to represent the complete circuit. The approach proposed in this work applies a variation of the inverter minimization procedure proposed by Jain and Bryant as a way for obtaining reduced transistor count circuits using simple cells. In the following, we review the phase-constraint and polarity graphs proposed by Jain and Bryant.

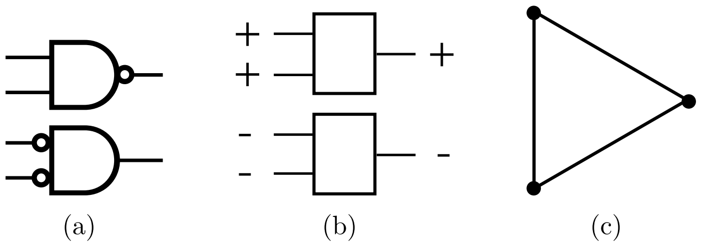
2.4.2 Phase-Constraint Graph

A phase-constraint graph defines constraints between the inputs and outputs of a given base function (e.g. NAND). For simplicity, we will refer to these inputs and outputs as pins. This base function defines how other allowed functions can be obtained by permuting the phase assignments on its pins. Thus, phase-constraint graphs derive from a set of allowed phase assignment permutations to input and output pins of their base functions (JAIN; BRYANT, 1993). Each node in a phase-constraint graph corresponds to a pin of the base function. Connected nodes imply that the corresponding pins must have the same phase assignment in all patterns in the set.

Figure 2.6 shows a NAND/NOR phase-constraint graph. In this example, the

NAND cell is the base function and so, by definition, its pattern has a positive phase assignment in all pins (JAIN; BRYANT, 1993). The NOR operation can be obtained by permuting all phase assignments from NAND cell ($\overline{A+B} \equiv \overline{A} \cdot \overline{B} \equiv \overline{\overline{\overline{A} \cdot \overline{B}}}$). This way, the NOR pattern has a negative phase assignment in all pins. In this case, NAND and NOR cells define the phase-constraint set. The resulting phase-constraint graph has three nodes, corresponding to the number of pins in its base function. The edges specify that all three pins should have the same phase assignment (either all positive or all negative).

Figure 2.6: NAND/NOR phase-constraint graph (c), derived from its allowed phase assignment permutations (a) and respective patterns (b).



Source: Author.

2.4.3 Polarity Graph

The polarity graph states how each gate in a given logic network differs from the phase-constraint graph of the corresponding base function (JAIN; BRYANT, 1993). For this, let n_i be the i_{th} net on the logic network, and let g_k be its k_{th} gate. For every input net n_i of a gate g_k, $\gamma(g_k, n_i)$ denotes whether the net is in its positive form ($\gamma_1(g_k, n_{i_1}) = +$) or in its complemented form ($\gamma_2(g_k, n_{i_2}) = -$). All output nets n_o have their positive form with respect to their gate g_k, i.e., $\gamma_3(g_k, n_o) = +$ (JAIN; BRYANT, 1993).

Each vertex v_i in the polarity graph corresponds to a net in the logic network (JAIN; BRYANT, 1993). Each edge e_{ij}, between vertex v_i and vertex v_j, implies that there is a constraint of phase assignment in the phase-constraint graph. All edges have a label $\lambda(e_{ij}) \in \{+, -\}$. A positive edge ($\lambda(e_{1_{ij}}) = +$) represents the fact that the nodes v_i and v_j are required to have the same phase assignment. A negative edge ($\lambda(e_{2_{il}}) = -$) specifies that the vertices v_i and v_l should have opposite phases.

The edge labels in the polarity graph are obtained by applying the *times operator*, represented by the symbol $\bullet$. The times operator must be applied between the polarity of the nets in the logic network, i.e., over the elements of the set $\{+, -\}$. Let $\gamma(g_k, n_i)$ and

$\gamma(g_k, n_j)$ be the polarity of nets n_i and n_j for the gate k. The label of edge e_{ij}, denoted by $\lambda(e_{ij})$, is defined by operating the times operator, such that:

$$\lambda(e_{ij}) = \gamma(g_k, n_i) \bullet \gamma(g_k, n_j) = \begin{cases} +, & \text{if } \gamma(g_k, n_i) \text{ and } \gamma(g_k, n_j) \text{ have the same label} \\ -, & \text{if } \gamma(g_k, n_i) \text{ and } \gamma(g_k, n_j) \text{ have different labels} \end{cases} \tag{2.2}$$

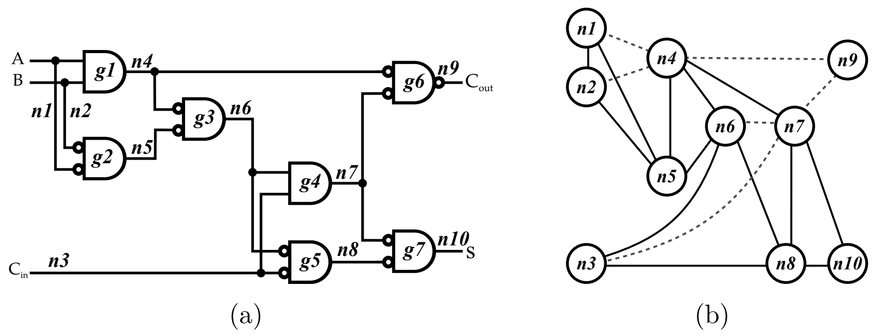
This way, according to Equation 2.2, $\lambda(e_{ij})$ could be:

$$(+ \bullet +) = (- \bullet -) = + \tag{2.3}$$

$$(+ \bullet -) = (- \bullet +) = - \tag{2.4}$$

Figure 2.7 shows a logic network and its corresponding polarity graph for the NAND/NOR phase-constraint. In Figure 2.7(b), the positive edge (solid line) between nodes n_1 and n_2, introduced due to gate g_1, indicates that the vertices should have the same phase in order to avoid adding explicit inverters. In the same way, the negative edge (dotted line) between nodes n_1 and n_4, also introduced due to gate g_1, indicates that the vertices should have opposite phases.

Figure 2.7: Logic network (a) and the corresponding polarity graph (b).



(a) (b)

Source: Author.

3 LOGIC SYNTHESIS REVIEW

This chapter traces an evolutionary line over logic synthesis, describing some previous and recent works that are connected to the work presented in this thesis. First, a historical perspective is presented. After, AIG rewriting approaches are reviewed. We also review standard-cell- and library-free-based approaches for technology mapping. Finally, a discussion around simple and complex cells is presented.

3.1 Logic Synthesis Historical Perspective

The logic synthesis step has had tremendous commercial success and tools for logic synthesis are effectively used by designers. Historically, logic synthesis started as two-level synthesis (e.g. ESPRESSO (BRAYTON et al., 1982)), where the goal was to minimize the number of literals in Sum-Of-Products (SOP) representations. The number of literals was correlated with the final area of Programmable Logic Array (PLA) implementations, used as the most common layout for early large scale integration (LSI) physical design implementations.

The two-level synthesis was forced to evolve into multi-level logic synthesis as the most common layout implementations moved from PLAs to cell-based layouts relying on the use of standard cell libraries. The move towards cell-based multi-level layout implementations led to heuristic logic synthesis tools for multi-level logic implementations. Examples of multi-level logic synthesis tools include MIS (BRAYTON et al., 1987) and SIS (SENTOVICH et al., 1992). The multi-level logic synthesis was divided into two main steps (technology independent and technology dependent) due to the use of a cell library. Early logic synthesis tools for multi-level networks (BRAYTON et al., 1987) were based in an internal representation composed of a set of logic nodes, where each node could contain a fanout free equation. Only the output of logic nodes could have fanout greater than one. The goal of the technology independent step of these tools was to minimize the total sum of the literals of the equations in the internal to the nodes. That means that, from a technology independent point-of-view, literal minimization was still important.

However, the double intent of having a technology independent step, that reduces literal count, and a technology dependent step, that minimizes the sum of the costs of the instantiated cells, led to an intent mismatch in the logic synthesis flow. Specifically, minimizing the number of literals not necessarily leads to a minimized area after technology mapping. For this reason, the logic synthesis community started to prefer minimizing the number of nodes of AIG representations instead of the minimization of literals (Berkeley Logic Synthesis and Verification Group, 2013).

Nowadays, it is well accepted that the total number of nodes in a AIG is a better area estimator than literal count in a set of interconnected fanout-free equations. Additionally, it is also well accepted that reducing the structural depth of an AIG tends to reduce the delay of a circuit after mapping. This way, current logic synthesis tools, such as ABC (Berkeley Logic Synthesis and Verification Group, 2013), are based in using an AIG as their data structure representation.

3.2 AIG Rewriting

Logic optimization approaches can be divided into *algorithmic-based* methods, which are based on global transformations, and *rule-based* methods, which are based on local transformations (BRAYTON et al., 1984). Rule-based methods, also called *rewriting*, use a set of rules which are applied when certain patterns are found. The AIG rewriting is a rule-based greedy algorithm for optimize an objective function, such as minimizing the number of nodes of the AIG. This approach iteratively selects subgraphs and replaces them with pre-computed logically equivalent subgraphs (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

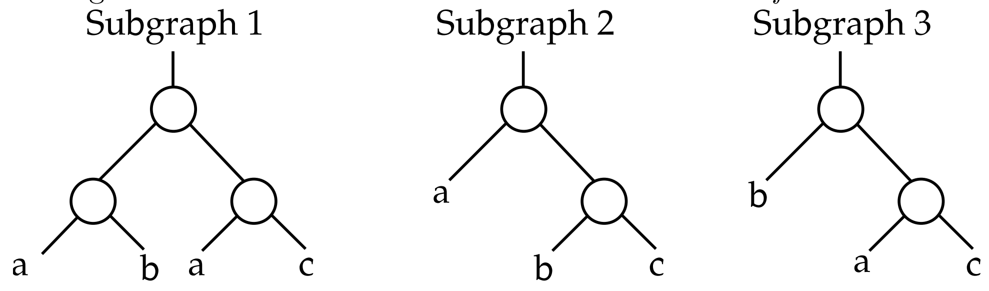
In 1982, Brayton and McMullen proposed an algorithm for decomposition and factorization in Boolean expressions (BRAYTON; MCMULLEN, 1982). This algorithm was implemented in SIS (SENTOVICH et al., 1992). Additionally, an AIG rewriting method called *refactor* is implemented in ABC tool. This method chooses large subgraphs for each AIG node, extracts the Boolean function of this subgraph and performs the Brayton and McMullen's factorization algorithm. The result of factorization is converted back to an AIG and replaces the original subgraph if the number of nodes is reduced (Berkeley Logic Synthesis and Verification Group, 2013).

Cortadella (2003) proposed an algebraic balancing approach in DAG structures claiming reductions on logic depth and timing optimizations. This algorithm was adapted to AIGs and it is also implemented in ABC tool, called *balance* (Berkeley Logic Synthesis and Verification Group, 2013). The approach is based on finding the minimum-depth tree for a Boolean function building the tree from root to leaves by using bi-decomposition techniques. The depth reduction is achieved by means of rewrite rules that apply the associative, commutative and distributive laws of the Boolean algebra.

The most known algorithm for AIG rewriting was proposed by Mishchenko, Chatterjee and Brayton (2006) and it is called DAG-Aware AIG Rewriting. It is also implemented in ABC, called *rewrite*. This method has three steps and it uses a hash table of pre-computed AIGs for all functions with up to four inputs. In this sense, there are 222 different NPN-equivalent classes of functions up to four variables, i.e., 222 sets of functions which can derive all functions up to four variables by combining three operations: negation of any variables, permutation of any variables, and/or negation of the function itself. The first step of this algorithm is pre-compute all AIG subgraphs for every 222 classes of functions. After that, the method traverses the AIG in topological order, from inputs to outputs. For each node, all possible subgraph up to four inputs are enumerated. Each of these subgraph is compared with the pre-computed graphs in the hash table. Those subgraphs that reduce the number of nodes without increasing the height of the region, or even subgraphs that add shared nodes are considered. After trying all available subgraphs for a node, the one that leads to the greatest improvement replaces the original subgraph.

Figure 3.1 depicts three possible AIG representations of function $f = a * b * c$. These AIGs must be pre-computed and stored in a hash-table. Figure 3.2 presents two examples of AIG rewriting. In Figure 3.2(a), *Subgraph 1* is replaced by *Subgraph 2*, which reduces one node in the AIG. Figure 3.2(b) shows *Subgraph 2* been replaced by *Subgraph 1*, which reduces one node in the graph due to the sharing of nodes.

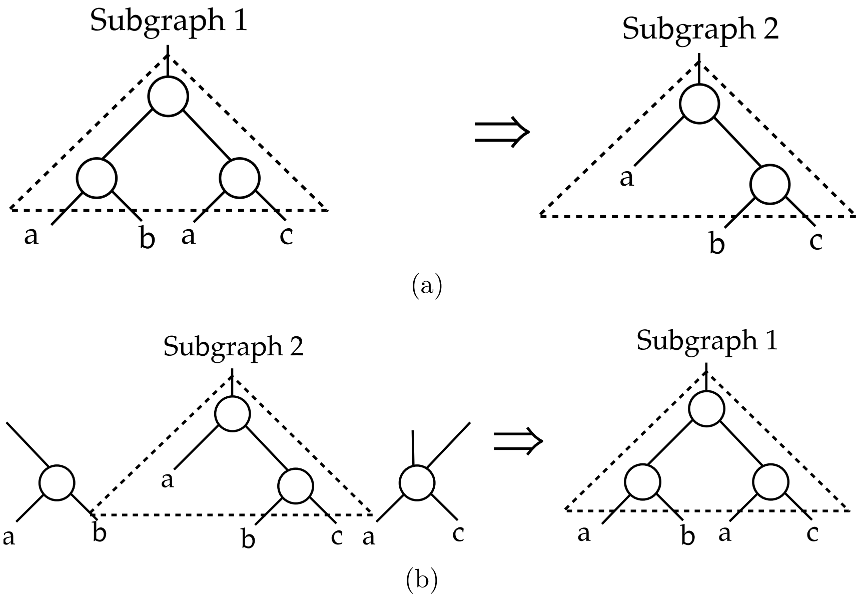
Figure 3.1: Different structures of AIG for function $f = a * b * c$.



Source: Autor: (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

The second and third steps of the *rewrite* approach consist in balancing and refactoring the AIG using the methods *balance* and *refactor*, respectively. The authors suggest a script that traverses the structure 10 times, as follows: *b, rw, rf, b, rw, rwz, b, rfz, rwz, b*. In the abbreviated form, *b* stands for balancing; *rw/rf* stand for AIG rewriting

Figure 3.2: Examples of AIG rewriting.



Source: Autor: (MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

and refactoring; and *rfz/rwz* are also rewriting and refactoring, but allowing replacements with zero improvement. The authors claim that this approach leads to a reduction of area in the order of 10% and improvements in delay of 5%, whereas the runtime is reduced by a factor ranging between 7 and ~1000, when comparing with previous approaches.

3.3 Approaches for Standard Cell Technology Mapping

The technology mapping step on first approaches for automatic synthesis of digital circuits was based on applying a set of rules over a structure that represents the circuit (DARRINGER et al., 1981; GREGORY et al., 1986). These methods perform local optimizations, trying to reduce the cost of a region of the circuit, which not necessarily lead towards global minimals.

The first algorithmic solutions for technology mapping were proposed only in 1987 (KEUTZER, 1987; DETJENS et al., 1987). Keutzer had presented DAGON as a compiler-based approach. According to Keutzer, search patterns between intermediate representations of a computer program and a given set of machine instructions is similar to the pattern matching between cells of a library and subgraphs of a circuit representation. However, the search space to be explored by mapping become limited by the structural matching and the initial representation of the circuit, which directly affects the quality of the mapped circuit. Additionally, DAGON approach requires all isomorphic matches to be stored in each node of the tree until the end of covering step. Thus, this work precludes the use of very large libraries, when the number of patterns found is usually higher.

Detjens et al. (1987) proposed to use trees as the subject graph and to insert pairs of inverters in this tree. This algorithm increases the solution space. However, the aimed advantages rely on performing several decompositions for each element of the library, what also turns unfeasible the use of large libraries. In 1989, Rudell extends the Detjens approach, in which he proposes two main different improvements: (1) to use pattern graphs allowing leaf-DAGs nodes, what makes possible non-tree gates (such as multiplexers and XORs) to be matched; and (2) to replace every wire in the subject graph by a pair of inverters in series, that makes larger the set of matches. Rudell's algorithm became known as "the inverter-pair heuristic".

The functional verification to identify patterns was introduced only in 1993 (MAILHOT; MICHELI, 1993). This work also had proposed a Boolean matching using BDDs. In this approach, finding matches did not depend anymore on the structure of these sub-trees. Nonetheless, this algorithm was computationally expensive. In 1995, Lehman proposed to integrate the decomposition and the pattern matching phases dynamically reorganizing the subject graph. Thus, the search space increased due to each node be associated with subgraphs functionally equivalent but structurally different. Even so, this approach becomes unpractical for large circuits because the graph grows very fast. Kukimoto, Brayton and Sawkar (1998) and Stok, Iyer and Sullivan (1999) have proposed two approaches to DAG covering ensuring optimality in terms of speed. The main weakness of these works is on their gain-based load-independent delay models, which means that they ignore the load of the cell, taking into account only its propagation delay.

Currently, the state-of-the-art in technology mapping for standard cell approaches is implemented into the ABC tool (Berkeley Logic Synthesis and Verification Group, 2013). It was proposed by Chatterjee et al. (2006) and is based on Kukimoto's algorithm, with two main differences: (1) Boolean matching instead of structural matching; and (2) AIG covering instead of DAG covering. The implemented Boolean matching is an improved version, which extends the Lehman's approach. Chatterjee proposed to encode multiple DAGs (without breaking them in trees) in an AIG called "AIG with choices", postponing detailed comparisons, what made his algorithm faster than previous approaches. In this sense, the AIG with choices also prevents memory overload. He also applied a technique similar to Rudells's inverter-pair approach to consider matches in both polarities.

3.4 Approaches for Library-Free Technology Mapping

Along with the approaches for standard cell technology mapping, algorithms based on library-free have also been proposed. The first presented method was proposed by Berkelaar and Jess (1988). Expressions of sums-of-products and product-of-sums with a prefixed notation are represented as graphs and are traversed from outputs to inputs partitioning into logic cells. As the cuts are made top-down, the logic depth of what is below is unknown. So, this greedy algorithm cannot guarantee a solution with minimum logic depth or with minimum number of cells.

In 1992, two important works were presented. Liem and Lefebvre (1992) have proposed a constructive matching, in which both the number of inputs and logic depth of a cell was considered, beyond maximum values for transistor chains. However, the method is hardly dependent on the initial structure and it is memory greedy, precluding the method for mapping large circuits. The work of Abouzeid et al. (1992) claimed the use of a large number of logic cells by partitioning the initial DAG into n-ary trees. This representation decreases the dependence on the initial graph, allowing change of structure in a given set of nodes. Although the cuts are generated from inputs to outputs, they are made in a greedy way, not contemplating logic depth minimization.

Reis et al. (1995) and Reis (1999) proposed to use a dynamic reordering on the initial representation of the circuit and to represent each logic cone in a different way: a special type of BDD, called Terminal-Suppressed Binary Decision Diagram (TSBDD). An interesting property of this structure is the direct association of the arcs of the BDD to transistors, although it faces the same problems of representation by trees.

Later on, Yanbin, Sapatnekar and Bamji (2001) have proposed the Odd-level Transistor Replacement (OTR) method, which works directly on an electrical diagram at transistor level, represented as a graph. The goal of the algorithm is to select which gates can be collapsed in order to achieve a better performance, but the method also suffers for depending strongly on the initial decomposition of the circuit.

In 2004, an algorithm proposed by Correia & Reis considers several decompositions of sub-trees dynamically (at a low computational cost), leading to a minimum coverage using dynamic programming. However, for using trees, the method cannot provide a broader view of the circuit. The VIRMA algorithm was presented by Marques et al. (2007), which performs the library-free mapping over a DAG aiming the reduction of the circuit delay (SCHNEIDER et al., 2005).

3.5 Simple Cells and Complex Cells into Standard Cell Libraries

Standard cell libraries are commercially available, optimized to achieve high quality results in terms of speed, area and power consumption. Such sets usually contain hundreds of elements, ranging from simple cells (i.e., straightforward implementations of elementary Boolean functions, as NAND2 and NOR2) to cells of higher complexity. Thus, the composition of libraries may vary according to the designer requirements and application constraints. The improvement of libraries may be thought by adding new drive strengths (DOOD; LEE; ALBERS, 2006), new available functions (GAVRILOV et al., 1997) and even particular transistor arrangements (MARQUES et al., 2007; SCHNEIDER et al., 2005; KAGARIS; HANIOTAKIS, 2007).

Some approaches advocate the use of a reduced set of cells. There are works in which the library was previously reduced by the designer and they claims that the logic synthesizers work better with a reduced number of gates (PIGUET et al., 2001); others where only *1*- or *2*-input cells were considered, which claims that the use of larger standard cells increases the number of long wires and may undermine circuit delay optimization at $65nm$ and below (SEO et al., 2008); or even works that reduce the library iteratively and statistically, claiming that the circuit performance, with respect to full-size library synthesis, do not appreciably degrades, and in several cases actually improves, whereas the synthesis time decreases and library maintenance and characterization tasks can thus be significantly reduced (RICCI; MUNARI; CIAMPOLINI, 2007).

On the other way around, there are approaches considering larger libraries (comprising complex gates) as a better choice. Some works claim that libraries with a large number of cells shall be preferred in order to bring more flexibility (GUAN; SECHEN, 1996); and works reporting that using different simple gate start-points, the gain of remapping the circuit using complex gates can still be obtained (REIS et al., 1995).

The work presented in this master's thesis partially agrees with works of Piguet et al. (2001), Ricci, Munari and Ciampolini (2007) and Seo et al. (2008), in the sense that good quality circuits can be obtained using simple cells. However, we also agree with the results reported by Reis et al. (1995) with respect to the gain of remapping the circuit using complex gates can still be obtained. This way, we believe it is possible that the transistor count of the circuits using simple cells can be further reduced by mapping with complex gates.

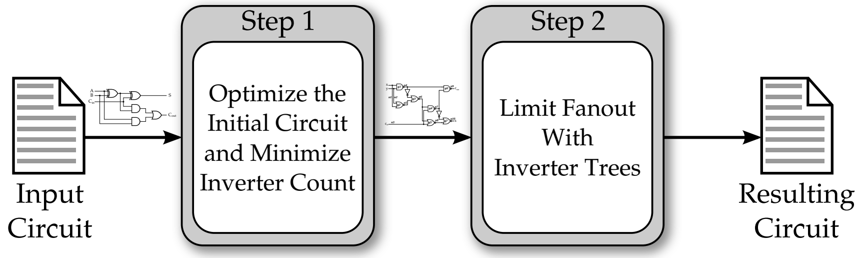
4 PROPOSED APPROACH

The approach proposed herein is mainly based on two steps, which can be split in substeps. This chapter presents a brief overview of the proposed approach, focusing on the adopted flow to achieve our claims. Details of each step and substep are presented on Chapter 5.

The set of algorithms proposed herein aims to provide reduced transistor count for digital semicustom VLSI circuits based on simple cells. The resulting circuits can be used for two main purposes: to implement efficient circuits using simple cells; or to use the optimized circuit as input of technology-dependent optimizations using complex cells. The proposed algorithms are mainly focused on starting from minimized node count AIG representations, obtained through state-of-the-art logic synthesis tools, and mapping this optimized AIG using simple cells (NAND2 and NOR2) with a minimal number of inverters.

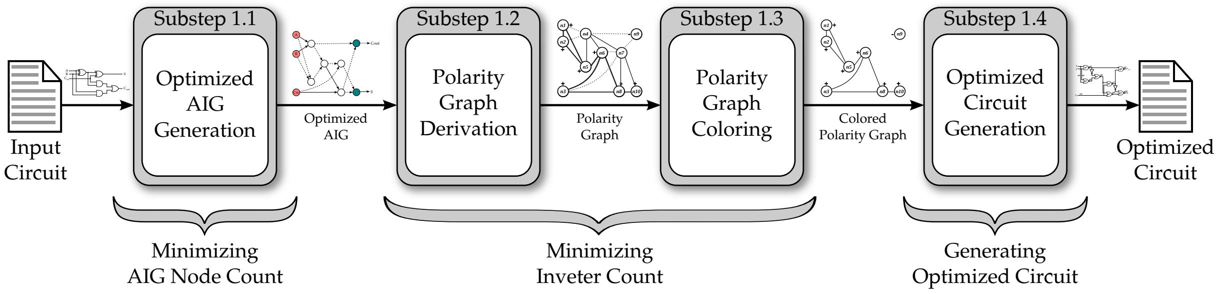
In order to obtain efficient circuits mapped using simple cells, this work follows an approach with two major steps, depicted in Figure 4.1. This chapter provides an overview of the proposed approach. The first step is to optimize the initial circuit in terms of number of gates and to minimize the number of inverters. After obtaining a minimal gate count, the intermediate circuit might have cells with fanout larger than the acceptable. The second step is to verify these occurrences and fix all of them.

Figure 4.1: Proposed flow for obtaining reduced transistor count circuits mapped using simple cells.


Step 1
Optimize the Initial Circuit and Minimize Inverter Count
Step 2
Limit Fanout With Inverter Trees
Input Circuit
Resulting Circuit

Source: Author.

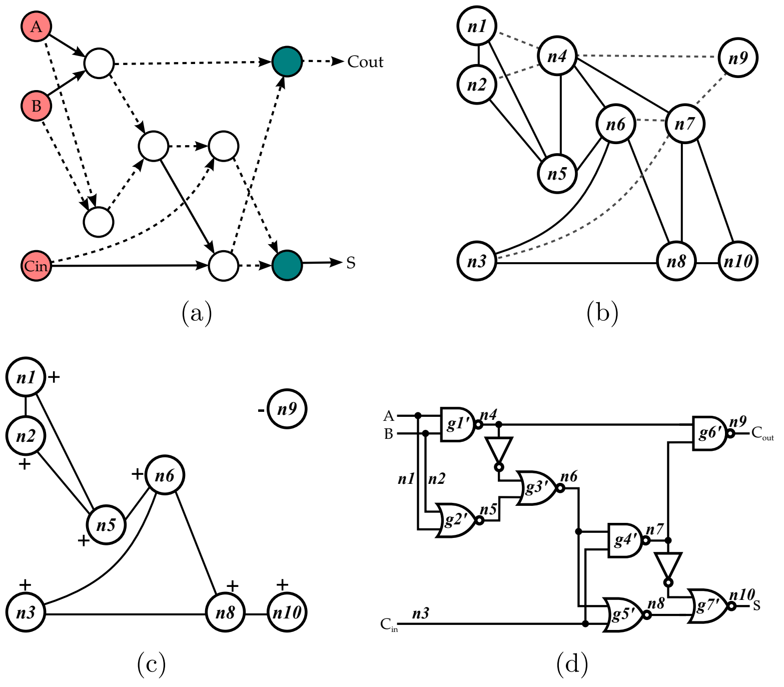
Figure 4.2: Proposed flow for obtaining reduced transistor count circuits mapped using simple cells.



Source: Author.

The optimization of number of cells is based on reducing the number of nodes in its initial AIG representation. To minimize the inverter count, our approach applies a procedure adapted from the work proposed by Jain and Bryant (1993), which is based on graph coloring. For obtaining the resulting fanout-limited circuit, we propose a novel method for fanout limiting using inverter trees. The following sections provide overviews of these steps.

Figure 4.3: Graph coloring process for inverter minimization on a full adder AIG (a). The resulting gate representation (d) is derived from the colored polarity graph (c) after removing nodes *n4* and *n7* from (b).



Source: Author.

4.1 Optimizing the Initial Circuit and Minimizing Inverter Count

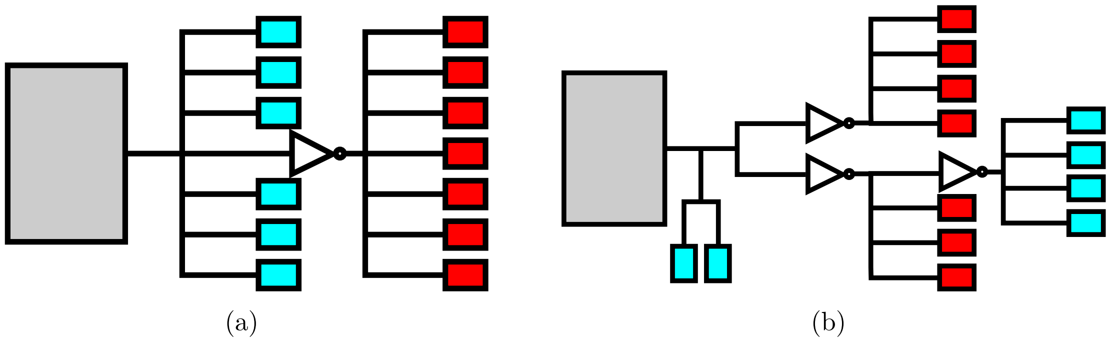
In order to achieve the aims claimed by the Step 1 of the proposed flow, an AIG representation of the initial circuit is generated. After that, the AIG node count must be minimized, the polarity graph is derived from the AIG, the obtained polarity graph needs be colored and the intermediate circuit can be generated. Figure 4.2 depicts the flow adopted in this step.

Figure 4.3 presents an example of applying the Step 1. The initial circuit is a full adder implementation, represented in an AIG in Figure 4.3(a). The polarity graph obtained from the AIG is depicted in Figure 4.3(b). Figure 4.3(d) shows the intermediate circuit obtained from the colored polarity graph in Figure 4.3(c).

4.2 Limiting Fanout with Inverter Trees

The proposed algorithm for fanout limitation with inverter trees takes into account the maximum fanout of the source cell, the maximum fanout of inverters, the number of positive consumer cells (i.e., the output cells driven directly from the source cell), and the number of negative consumer cells (i.e., the output cells driven from the source cell through an inverter). Using a mathematical formulation, to be described further, the proposed method computes the minimal number of inverters to drive both positive and negative cells, respecting the given maximum fanout limits. Figure 4.4 shows an example of applying the proposed algorithm in a cell whose maximum fanout limit was not respected. In the shown example, consider both the maximum fanout of the cell and the maximum fanout of inverters as 4.

Figure 4.4: Example of fanout violation (a) and fanout limiting using an inverter tree (b).



Source: Author.

In Figure 4.4, the gray boxes represents the source cell, the blue boxes represent the positive consumer cells, and the red boxes represent the negative consumer cells. Notice that, considering both the maximum fanout of the cell and the maximum fanout of inverters as 4, Figure 4.4(a) depicts fanout violations because both the source cell and the inverter have fanout 7. In Figure 4.4(b), these violations were fixed adding two more inverters and, this way, both the source cell and the three inverters have a maximum fanout of 4 cells.

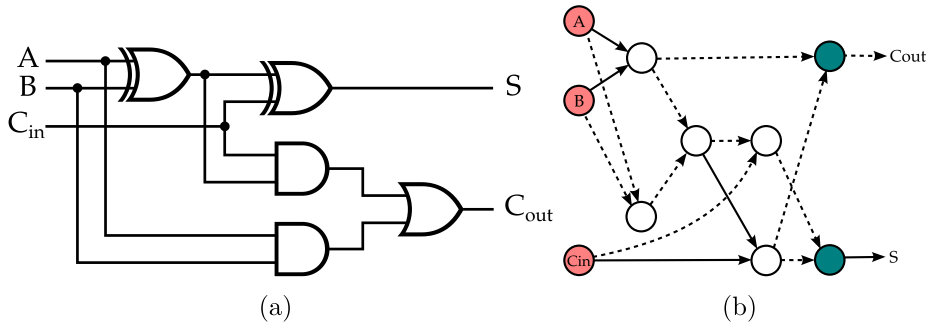
5 REDUCING TRANSISTOR COUNT IN BENCHMARK CIRCUITS

This chapter presents details of steps and substeps proposed herein. From Section 5.1 to Section 5.5, the basis of our approach is presented. Sections 5.6 to 5.8 present proposed techniques for improving the final results.

5.1 Reducing Node Count in AIGs

The first step in the proposed approach (Step 1.1) is to reduce the node count in an AIG representation of the initial circuit. To achieve this claim, we propose to use ABC tool, from the Berkeley Logic Synthesis and Verification Group (2013), to read the input circuit file and to minimize the node count in its AIG representation. This is straightforward and we use several different scripts to obtain reduced node count AIGs. The scripts have to be reiterated until no more gain can be obtained, as the solution given by logic synthesis tools depends on the input circuit. Figure 5.1 shows a full adder gate representation and an AIG, obtained after Step 1.1.

Figure 5.1: Deriving an AIG representation (b) from a full adder gate representation (a) using ABC tool (Berkeley Logic Synthesis and Verification Group, 2013).



Source: Author.

The main ABC scripts applied in this initial process are based on the following commands (Berkeley Logic Synthesis and Verification Group, 2013):

balance: performs algebraic balancing of the multi-input AND-gates contained in the original AIG;

rewrite: performs rewriting of the AIG taking into account its DAG features;

refactor: performs iterative collapsing and refactoring of logic cones in the AIG.

These commands and their variations were combined and interleaved with each other. The most reiterated ABC scripts were *resyn*, *resyn2*. By iterating these scripts, the AIG size reduces substantially and it tends to reduce its number of AIG levels and nodes (Berkeley Logic Synthesis and Verification Group, 2013).

Algorithm 5.1 presents a pseudocode that performs the Step 1.1. This method has only one input: the initial circuit. After deriving an AIG from the initial circuit (line 3 in Algorithm 5.1), the ABC scripts are invoked repeatedly until saturation, i.e., no more gain can be obtained in a given number of sequential iterations (called *satCount* in Algorithm 5.1). The number of sequential iterations used as test for saturation was 10. The final and optimized AIG is the method return.

Algorithm 5.1: Reducing AIG node count.

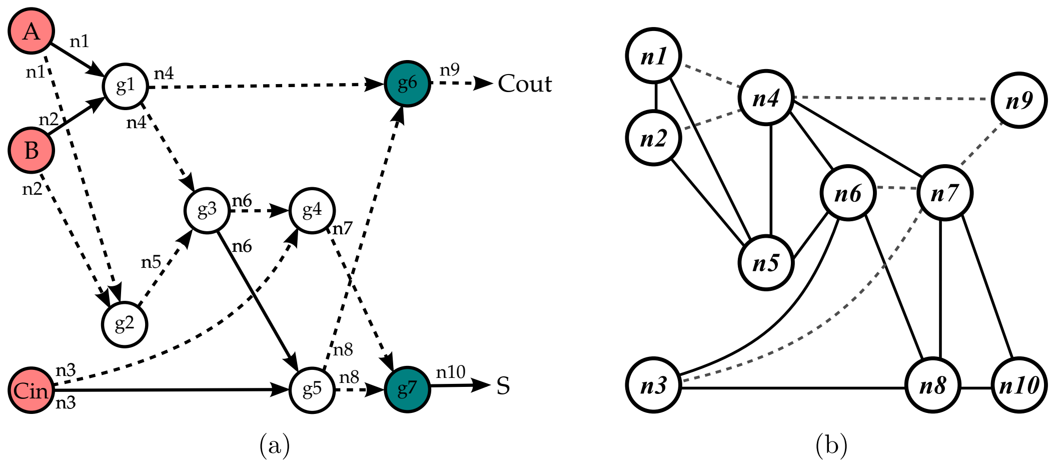
```
1  aig deriveMinimizedNodeCountAig(initialCircuit){
2      $aig$ = deriveAigFromCircuit($initialCircuit$);
3      $nodeCountBefore = 0$, $nodeCountAfter = 0$, $satCount = 0$;
4      do
5          $nodeCountBefore = aig.getNodeCount()$;
6          minimizeNodeCountWithABC($aig$);
7          $nodeCountAfter = aig.getNodeCount()$;
8          if ($nodeCountAfter < nodeCountBefore$)then
9              $satCount = 0$;
10         else
11             $satCount++$;
12     while ($satCount < 10$);
13     return $aig$;
```

5.2 Deriving Polarity Graph from AIG

Once the minimal AIG node count implementation has been obtained, the inverter minimization step begins and the polarity graph must be derived. The derivation procedure used herein is a variant of the procedure proposed by (JAIN; BRYANT, 1993), illustrated in Figure 5.2. In this work, we derive polarity graphs using the NAND2 as base function and applying NAND2/NOR2 phase constraints.

Figure 5.2(a) shows the AIG representation obtained after Step 1.1. The resulting polarity graph for NAND2/NOR2 phase constraints is presented in Figure 5.2(b). Each

Figure 5.2: The optimized AIG (a) which results in the polarity graph (b).



Source: Author.

set of outgoing edges from each node of the AIG corresponds to a net in a circuit and, so, derives a node in the polarity graph. The AND node *g1*, shown in Figure 5.2(a), determines the relationship between the polarities of nodes *n1*, *n2* and *n4* in the polarity graph. The AND node *g1* denotes an AND2 operation and so, by the NAND2/NOR2 phase constraints, the polarities between the inputs (*n1* and *n2*) and the output (*n4*) of *g1* have to be different. Therefore, node *n4* is connected to nodes *n1* and *n2* by two negative edges (dotted lines). Similarly, the AND node *g2*, shown in Figure 5.2(a), determines the relationship between the polarities of nodes *n1*, *n2*, and *n5* in the polarity graph. The AND node *g2* denotes a NOR2 operation (as De Morgan's law is applied). By the NAND2/NOR2 phase constraints, the polarities between the inputs (*n1* and *n2*) and the output (*n5*) of *g2* have to be the same. This way, node *n5* is connected to *n1* and *n2* by positive edges (solid lines). Applying this process to all gates in the circuit results in the complete polarity graph shown in Figure 5.2(b).

Algorithm 5.2 presents a pseudocode for Step 1.2. The proposed algorithm has one input: the optimized AIG. After creating the polarity nodes from the AIG nets (lines from 2 to 4), the edges between the polarity nodes are created (from line 5 to 31). The derived polarity graph is the method's return.

Algorithm 5.2: Deriving a polarity graph from an AIG.

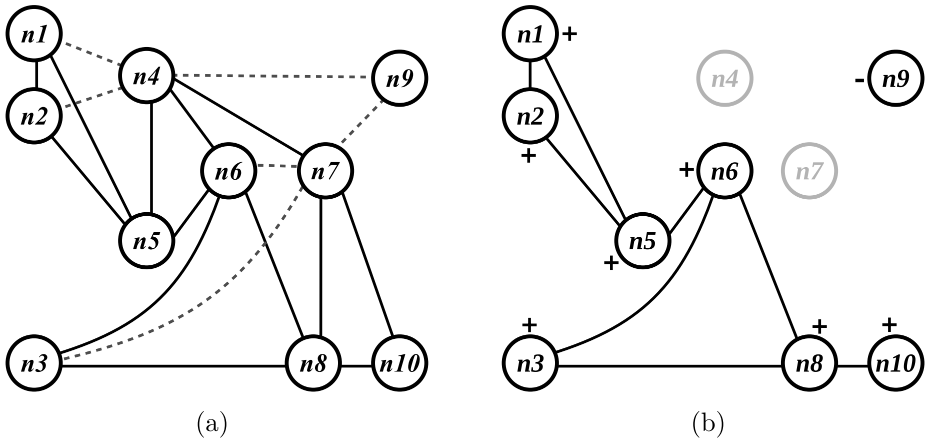
```
polGraph derivePolGraphFromAig(aig){
    "create a polarity node for each net on AIG";
    "map each polarity node with the related AIG nodes";
    "add each polarity node to the polarity graph";
    foreach (aigNode ∈ aig)do
        input1 = aig.getPolNode(aigNode.getInput1);
        input2 = aig.getPolNode(aigNode.getInput2);
        output = aig.getPolNode(aigNode);
        if
        (aigNode.isInput1Inverted() == aigNode.IsInput2Inverted())then
            polGraph.createPositiveEdgeBetween(input1,input2);
        else
            polGraph.createNegativeEdgeBetween(input1,input2);
        if (aigNode.getNOutputs() == 1 &&
        aigNode.IsOutputNodePrimaryOutput())then
            outputNode = aigNode.getOutputNode();
            if (aigNode.isInput1Inverted()! = outputNode.isInverted())then
                polGraph.createPositiveEdgeBetween(input1,outputNode);
            else
                polGraph.createNegativeEdgeBetween(input1,outputNode);
            if (aigNode.isInput2Inverted()! = outputNode.isInverted())then
                polGraph.createPositiveEdgeBetween(input2,outputNode);
            else
                polGraph.createNegativeEdgeBetween(input2,outputNode);
        else
            if (aigNode.isInput1Inverted())then
                polGraph.createPositiveEdgeBetween(input1,outputNode);
            else
                polGraph.createNegativeEdgeBetween(input1,outputNode);
            if (aigNode.isInput2Inverted())then
                polGraph.createPositiveEdgeBetween(input2,outputNode);
            else
                polGraph.createNegativeEdgeBetween(input2,outputNode);
```

5.3 Coloring the Obtained Polarity Graph

The inverter minimization problem is now reduced to coloring the vertices of the polarity graph with a unique polarity in $\{+, -\}$ such that the polarity constraints given by the edges are satisfied. This task is performed in Step 1.3.

The graph depicted in Figure 5.2(b) cannot be colored in this way due to the presence of cycles with an odd number of negative edges (called odd cycles hereafter), e.g. cycle {*n2*, *n4*, *n5*, *n2*} and cycle {*n6*, *n7*, *n8*, *n6*} (JAIN; BRYANT, 1993). These two cycles can be removed by deleting nodes *n4* and *n7*, leading to the graph seen in Figure 5.3(b), which is colorable. Figure 5.3 shows an example of applying the graph coloring approach.

Figure 5.3: The colored polarity graph (b) after removing nodes *n4* and *n7* from the uncolorable graph (a).



Source: Author.

Each deleted node during the coloring process (i.e., *n4* and *n7* in Figure 5.2(b)) will derive one inverter in the intermediate circuit. Therefore, minimizing the number of inverters is equivalent to find a minimum transversal of all cycles with an odd number of negative edges. This problem is NP-hard, since it is a special case of the minimum odd cycle transversal problem (LEWIS; YANNAKAKIS, 1980).

Jain and Bryant (1993) propose two heuristics to search for the minimum transversal of all cycles with an odd number of negative edges: the *QuickColor* and *GoodColor* heuristics. Both of them were applied in the work proposed herein. The *QuickColor* heuristic picks an arbitrary odd cycle from the graph and then selects the vertex in this cycle with the maximum double-edge degree, i.e., the vertex with the highest number of two edges (one positive and one negative) between the same neighbor. If two vertices have the same double-edge degree, then *QuickColor* shall select the one with the maximum edge degree. After removing the selected vertex and incident edges, attempt recoloring the remaining graph. Applying this process to all nodes in the polarity graph results in the colored polarity graph shown in Figure 5.3(b).

Algorithm 5.3 presents a pseudocode of the *QuickColor* approach. The method has one input: the polarity graph to be colored. While the polarity graph has cycles with an odd number of negative edges, the algorithm picks an arbitrary odd cycle and marks a node as removed. After the required number of iterations, the polarity graph is colorable. The method return is the graph after colored.

Algorithm 5.3: Polarity graph coloring with the *QuickColor* Heuristic.

```
1 polGraph quickColor(polGraph){
2     while (polGraph.hasOddCycle())do
3         oddCycle = polGraph.getOddCycle();
4         nodeToBeRemoved = oddCycle.getNodeToBeRemoved();
5         polGraph.markNodeAsRemoved(nodeToBeRemoved);
6     return polGraph.graphColoring();
```

Notice that, in fact, no node is removed. Instead, the nodes are marked as removed. This is because these “removed nodes” will generate inverters in the intermediate circuit.

Different from *QuickColor*, which takes an arbitrary next odd cycle, the *GoodColor* heuristic tries to pick a “good” next odd cycle. This approach picks the smallest cycle in the graph as the next candidate. Another remarkable feature is that the *GoodColor* tries to recover itself from an early “potentially bad” choice. According to Jain and Bryant (1993), each eliminated odd cycle in the graph is said to be covered by the vertex chosen to be removed. This way, it is possible to know when the next odd cycle shares vertices with a previously eliminated cycle. For explaining this approach, let $cycle_n$ be the next odd cycle; $cycle_p$, a previously eliminated cycle, which is covered by vertex v_p and shares vertices with $cycle_n$; and $\aleph(v_p)$, be the number of eliminated cycles containing the vertex v_p. If $\aleph(v_p) = 1$, the *GoodColor* heuristic “undo” the removing of vertex v_p and removes one of the shared vertices covering both cycles, $cycle_n$ and $cycle_p$.

Algorithm 5.4 presents a pseudocode of the *GoodColor* approach. The method also has only one input: the polarity graph to be colored. While the polarity graph has odd cycles, the algorithm picks the smallest one and chooses a node to be removed (lines 3 and 4). If any node of the cycle has been previously covered (lines 5 and 6), the method finds the cycle breaker of the previously removed cycle (line 7). In case of the ℵ(cycle breaker) equals to 1, undo this previous removal and selects the shared node to be removed (lines from 8 to 11). After the required number of iterations, the polarity graph is colorable. The method's return is the graph after colored.

Algorithm 5.4: Polarity graph coloring with the *GoodColor* Heuristic.

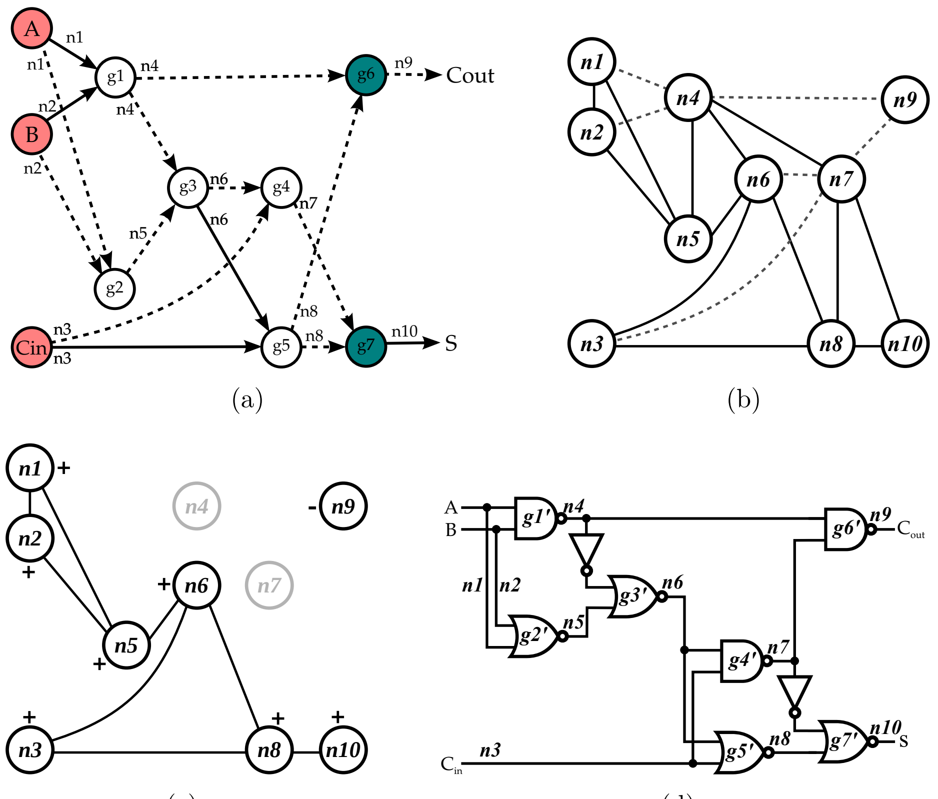
```
1  polGraph goodColor(polGraph){
2      while (polGraph.hasOddCycle())do
3          oddCycle = polGraph.getSmallestOddCycle();
4          nodeToBeRemoved = polGraph.getNodeToBeRemoved();
           /* search for previously covered vertices                    */
5          foreach (cycleNode ∈ oddCycle)do
6              if (ℵ(cycleNode) > 0)then
7                  coveringRoot = cycleNode.getCovergingRoot();
8                  if (ℵ(coveringRoot) == 1)then
9                      polGraph.undoRemoving(coveringRoot);
10                     nodeToBeRemoved = cycleNode;
11                     break;
           /* cover the cycle                                           */
12         foreach (cycleNode ∈ oddCycle)do
13             ℵ(cycleNode) + +;
14         polGraph.markNodeAsRemoved(nodeToBeRemoved);
15     return polGraph.graphColoring();
```

5.4 Deriving the Final Circuit

Step 1.4 takes the colored graph derived from Step 1.3 and generates the intermediate circuit. Figure 5.4 shows an example of applying the circuit deriving procedure from a colored polarity graph. The NAND2/NOR2/INV circuit shown in Figure 5.4(d) can be derived from the colored graph in illustrated Figure 5.4(c).

Using the times operator (•), the final color of each node n_i ($color(n_i)$) in the polarity graph (corresponding to an input net of AND node g_k in the AIG), and its initial γ must lead to a known pattern in the phase-constraint set. By deriving each node n_i, the colored graph produces a final circuit containing only gates in the phase-constraint set. For instance, the three nodes $n1$, $n2$ and $n5$, depicted in Figure 5.3(b), have + color, as well as the node $n4$ should have the $-$ color with respect to nodes $n1$ and $n2$. Applying

Figure 5.4: The colored polarity graph (a) and the derived intermediate circuit (b).



Source: Author.

the • operator between the colored nodes ($n1$, $n2$, $n4$ and $n5$) and the form γ of these nest in 5.4(a) produces:

$$color(n1) \bullet \gamma(n1, g1) = + \bullet + = + \tag{5.1}$$

$$color(n2) \bullet \gamma(n2, g1) = + \bullet + = + \tag{5.2}$$

$$color(n4) \bullet \gamma(n4, g1) = - \bullet - = + \tag{5.3}$$

$$color(n1) \bullet \gamma(n1, g2) = + \bullet - = - \tag{5.4}$$

$$color(n2) \bullet \gamma(n2, g2) = + \bullet - = - \tag{5.5}$$

$$color(n5) \bullet \gamma(n5, g2) = + \bullet - = - \tag{5.6}$$

The result of Equations (5.1), (5.2) and (5.3) implies that node $g1$ is a NAND gate ($g1'$ in Figure 5.4(d)). Similarly, the result of Equations (5.4), (5.5) (5.6) implies that node $g2$ is a NOR gate ($g2'$ in Figure 5.4(d)). Applying this process to all nodes in the colored graph results in the circuit shown in Figure 5.4(d). Notice that the nodes marked as removed in the polarity graph ($n4$ and $n7$ in Figure 5.4(c)) generate inverters in the nets of the intermediate circuit (nets $n4$ and $n7$ in Figure 5.4(d)).

Algorithm 5.5 presents a pseudocode of Step 1.4. The method has only one input: the colored polarity graph. The proposed algorithm starts by creating the input pins (lines 2 to 4) and the output pins (lines 5 to 6). After creating one inverter for each removed node, the core of the deriving procedure (from line 7 to line 31) derives the NAND and NOR gates. Notice that inverters are created on input and output nodes, depending on the final color and the derived gate. These inverters are created in line 3 and lines 30 to 31, respectively. The method's return is the circuit after mapped.

5.5 Limiting Fanout with Inverter Trees

The algorithms for reducing node count in AIGs tend to increase the logic sharing in AIG nodes. At first, this feature may seem a good approach because the greater is the logic sharing, the smaller tends to be the AIG, mainly when it is result of an algorithm for minimizing the node count. However, a cell with excessive fanout has negative impact in the circuit performance for most of current technology fabrication (RABAEY; CHANDRAKASAN; NIKOLIC, 2002; WESTE; HARRIS, 2009).

In this sense, as the polarity graph is obtained directly from the AIG and the intermediate circuit, in turn, comes from the polarity graph, the cells of the intermediate circuit, generated after Step 1.4, tend to have fanout greater than desired. The algorithm proposed in this section fix any fanout violation inserting inverter trees. This approach takes into account the maximum fanout for the source cells and maximum fanout for inverters, both of them given by the algorithm designer. The root of the tree is the source cell.

The algorithm has two substeps: (1) find the minimum tree height for attending both the positive consumer cells and the negative consumer cells; and (2) create the inverter tree. To find the minimum tree height, in Step 2.1, the proposed algorithm has an exhaustive search approach. It starts with the smallest heights (0 for positive consumer

Algorithm 5.5: Derive the intermediate circuit from the colored polarity graph.

```
circuit deriveCircuitFromColoredGraph(polGraph){
    “create an input pin for each input node”;
    “create an inverter for each non-removed, negative-colored input node”;
    “mark input nodes as mapped”;
    “create an output pin for each output node”;
    “insert these output pins into the circuit”;
    // polarity nodes w.r.t.  output pins will be mapped further.
    “create an inverter for each removed node”;
    mapped = false;
    while (!mapped)do
        mapped = true;
        foreach (polNode ∈ polGraph.getNoInputNodes())do
            if (polNode.isNotMapped())then
                if (polNode.noneInputsWereRemoved())then
                    “map this node according to its inputs’ coloring pattern”;
                    “NAND patterns derive NAND cells and NOR patterns
                    derive NOR cells”;
                else
                    if (“removed input is already mapped”)then
                        if (polNode.getNonRemovedInputPattern == “NAND
                        pattern”)then
                            if ((removedInput.isNandCell() &&
                            correspondingAigNodeIsNotInverted) ||
                            (removedInput.isNorCell() &&
                            correspondingAigNodeIsInverted))then
                                nandCell.connectToInvertedInput();
                            else
                                nandCell.connectToDirectedInput();
                            circuit.addNandCell(nandCell);
                            polNode.markNodeAsMapped();
                        else if (polNode.getNonRemovedInputPattern ==
                        “NOR pattern”)then
                            “do as in lines from 18 to 24, considering the
                            opposite polarities.
                    else
                        mapped = false;
                        break;
                        /* the removed inputs shall be mapped first  */
    foreach (nonInvertedOutput ∈ polGraph.getOutputNodes())do
        “consider inverting this output performing the same tests on lines from
        lines 18 to 24”;
    return circuit;
```

cells and 1 for negative consumer cells) and verifies if the fanout of the cell can be attended with these levels. Otherwise, the method tries incrementally until find the levels which do not lead to fanout violations.

Let MFC be the maximum fanout of source cells, MFI be the maximum fanout of inverters, ip be the index for positive consumers (i.e. the maximum tree level for positive consumer cells), in be the index for negative consumers (i.e. the maximum tree level for negative consumer cells), PCC be the number of positive consumers, and NCC be the number negative consumers. In the base case, when $ip = 0$, Equation 5.7 and Equation 5.8 define the proposed conditions for verifying the fanout attendance. Otherwise, Equation 5.9 and Equation 5.10 shall be used when in is higher than ip; and Equation 5.11 and Equation 5.12 should be used in the verification when ip is higher than in.

$$MFC^{(ip+1)} \geq PCC \tag{5.7}$$

$$(MFC^{(ip+1)} - PCC) * MFI \geq NCC \tag{5.8}$$

$$MFI^{(ip+1)} \geq PCC \tag{5.9}$$

$$(MFI^{(ip+1)} - PCC) * MFI \geq NCC \tag{5.10}$$

$$MFI^{(in+1)} \geq NCC \tag{5.11}$$

$$(MFI^{(in+1)} - NCC) * MFI \geq PCC \tag{5.12}$$

Algorithm 5.6 presents a pseudocode of the proposed method for finding the minimum tree height. This method has three inputs: the source cell with fanout violation (*cell* in Algorithm 5.6); the maximum fanout of inverters (*MFI* in Algorithm 5.6); and the maximum fanout of source cells (*MFC* in Algorithm 5.6). The return informations are the index of positive consumer cells, the index of negative consumer cells and the tree height.

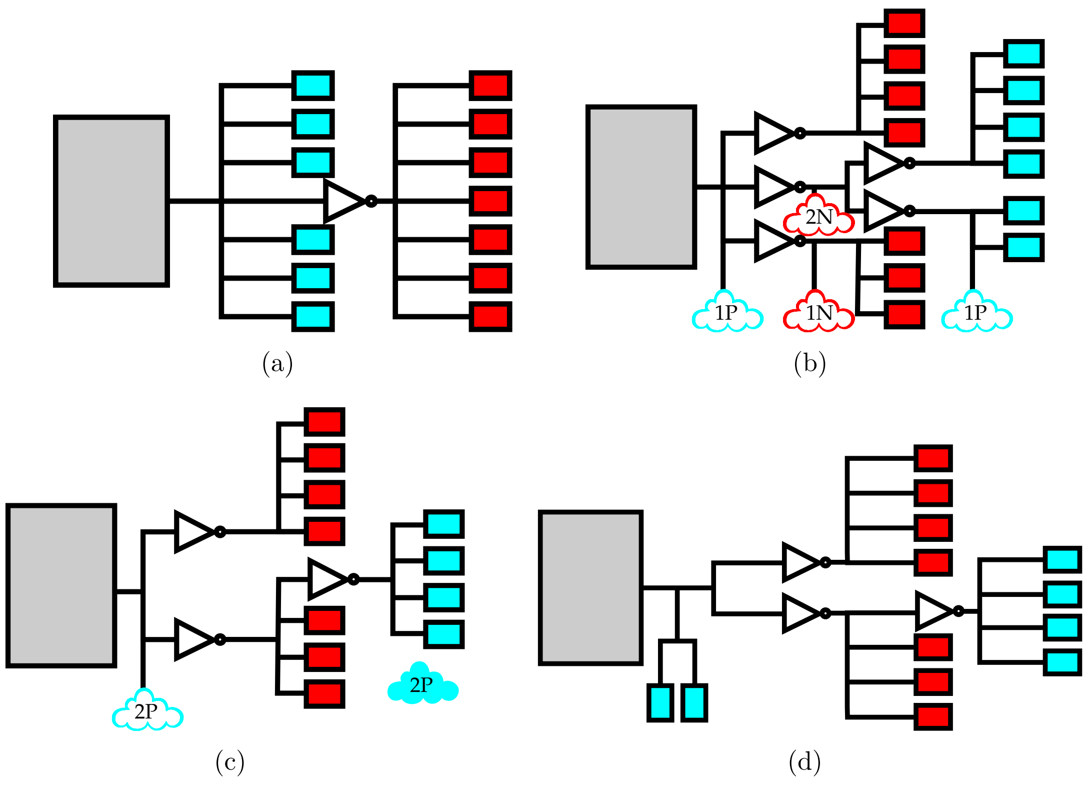
Algorithm 5.6: Finding the minimum tree height.

```
cell, MFI, MFC{
    PCC = cell.numPosConsumerCells();
    NCC = cell.numNegConsumerCells();
    $idxPosCells = 0$;
    $idxNegCells = 1$;
    if ($MFC^{idxPosCells+1} \geq PCC$ && $(MFC^{idxPosCells+1} - PCC) * MFI \geq NCC$)then
        $done$ = true;
    else
        $done$ = false;
    for $(i = 1; !done; i++)$do
        if ($i$ is even)then
            $idxPosCells = i$;
            $idxNegCells = i + 1$;
            if ($MFI^{idxPosCells+1} \geq PCC$ && $(MFI^{idxPosCells+1} - PCC) * MFI \geq NCC$)then
                $done$ = true;
        else $i$ is odd
            $idxPosCells = i + 1$;
            $idxNegCells = i$;
            if ($MFI^{idxNegCells+1} \geq NCC$ && $(MFI^{idxNegCells+1} - NCC) * MFI \geq PCC$)then
                $done$ = true;
    int [3] treeInfo;
    treeInfo[0] = $idxPosCells$;
    treeInfo[1] = $idxNegCells$;
    treeInfo[2] = max(idxPosCells,idxNegCells);
    return treeInfo;
```

Once the minimum tree height is known and the indexes of both positive and negative consumer cells were obtained, Step 2.2 uses this information to create the inverter tree. The proposed algorithm starts allocating all positive and negative consumer cells in their respective indexes and propagating the required number of inverters. After that, it tries to reallocate the cells from higher to lower levels in order to reduce the required number of inverters at the highest levels.

Figure 5.5 depicts the algorithm of Step 2.2 being applied. In this figure, the gray boxes denotes the source cell, the blue boxes are the positive consumer cells, and the red boxes are the negative consumer cells. For this example, consider both the maximum fanout of inverters and the maximum fanout of cells as 4. This way, Figure 5.5(a) illustrates two fanout violations, since both the source cell and the inverter have fanout 7. Applying Step 2.1, the minimum tree height is 2, the index of positive cells is 2 and the index of negative cells is 1. Figure 5.5(b) shows the subcircuit after allocating all positive and negative consumer cells in their respective indexes and propagate the required number of inverters. Figure 5.5(c) depicts the subcircuit after removing two positive consumer cells for further reallocation. Notice that, after propagating the required inverters again, the number of inverters reduces from 5 to 3. Figure 5.5(d) illustrates the subcircuit after reallocating the two cells previously removed from level 2, now allocated on level 0.

Figure 5.5: Example of fanout violation (a), substeps to limit fanout (b and c) and fanout limited using an inverter tree (d).



Source: Author.

Algorithm 5.7 presents a pseudocode of Step 2.2. This method has three inputs: the circuit to be analyzed, the maximum fanout of inverters and the maximum fanout of cells. If a fanout violation is detected (line 5 in Algorithm 5.7), Step 2.1 (line 6) and Step 2.2 (from line 7 to line 19) are applied. After running the required number of iterations, the fanout of the cell is limited using a inverter tree.

Algorithm 5.7: Fanout limiting algorithm.

```
void limitFanout(circuit, MFI, MFC){
    foreach (cell ∈ circuit) do
        PCC = cell.getNumPosConsumerCells();
        NCC = cell.getNumNegConsumerCells();
        if (PCC > MFC || NCC > MFI) then
            int [ ] treeInfo = findMinimumTreeHeight(cell,MFI,MFC);
            idxPosCells = treeInfo[0];
            idxNegCells = treeInfo[1];
            treeHeight = treeInfo[2];
            int [treeHeight][2] invAlloc;
            invAlloc[idxPosCells][1] = PCC;
            invAlloc[idxNegCells][1] = NCC;
            invAllocation = propagateInvAllocation(invAllocation,MFI);
            do
                oldInvAlloc = calcInvAlloc(invAlloc);
                invAlloc = minPosInvAlloc(invAlloc,idxPosCells,MFI);
                invAlloc = minNegInvAlloc(invAlloc,idxNegCells,MFI);
                invAlloc = minBoth(invAlloc,idxPosCells,idxNegCells,MFI);
                newInvAlloc = calcInvAlloc(invAlloc);
            while (oldInvAlloc! = newIncAllocs);
```

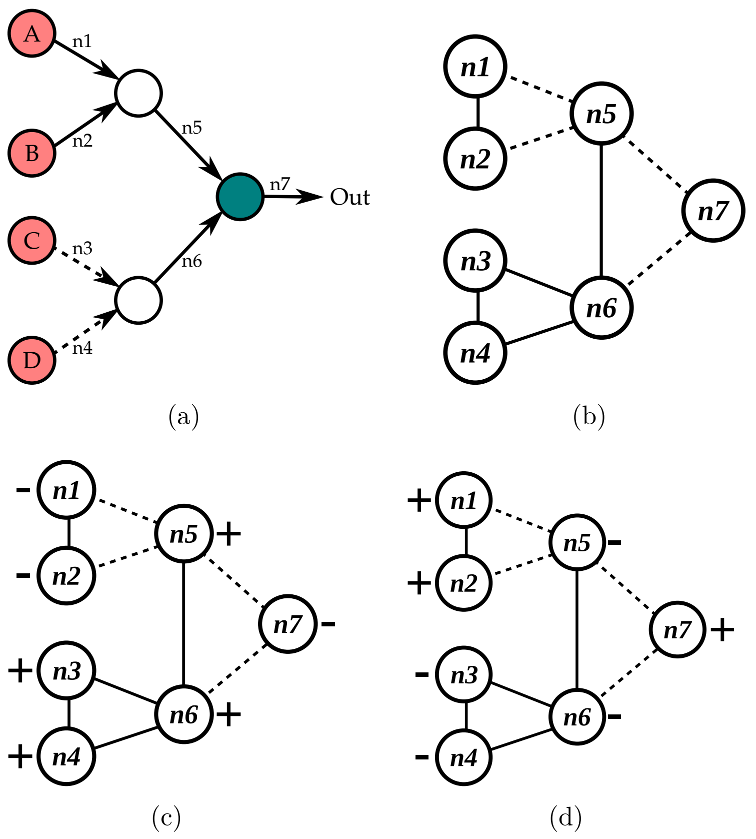
5.6 Forcing Colors in the Graph Using a Positive Polarity Inducing Node

The inverter minimization procedure proposed by Jain and Bryant (1993) has a remarkable feature: it minimizes the number of inverters in the circuit by pulling/pushing them to input or output pins. This feature become even more clear analyzing the results presented in the paper (JAIN; BRYANT, 1993), in which the authors do not consider the inverters on input and output pins. Figure 5.6 and Figure 5.7 depict an example of this feature.

Consider the AIG presented in Figure 5.6(a) as input of the Jain and Bryant's inverter minimization procedure. The derived polarity graph is presented in Figure 5.6(b). As with any graph coloring approach, the resulting colored polarity graph depends on the first color chosen and the node by which the algorithm starts. In the particular of polarity graphs, only two resulting graphs are possible. These two possibilities are presented in Figure 5.6(c) and in Figure 5.6(d).

As the derived circuit depends directly on the final colors of the colored polarity graph, the two possibilities of graph coloring derives two different, logically equivalent

Figure 5.6: A given AIG (a) and its polarity graph (b). The two possibilities of graph coloring (c and d) from the uncolored graph (b).

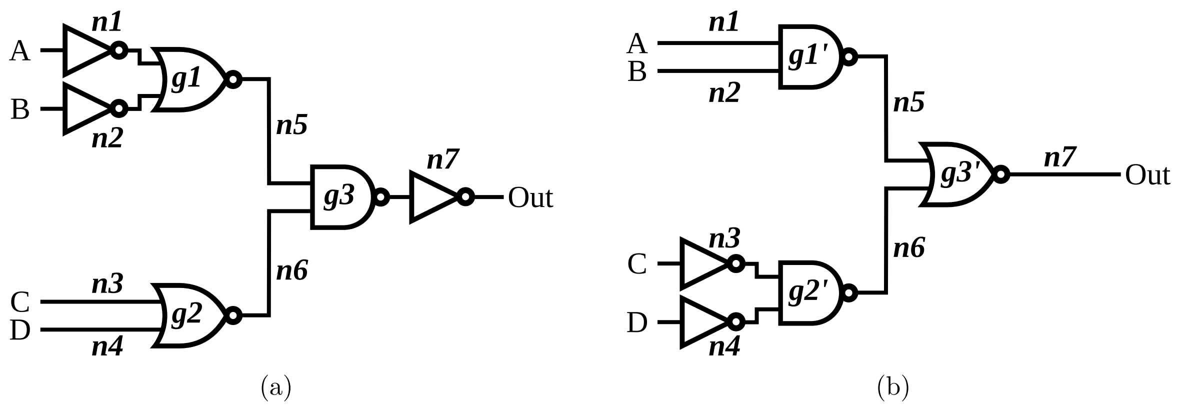


Source: Author.

possibilities of intermediate circuit. Figure 5.7 illustrates the two possible intermediate circuits, both derived from the two possibilities of graph coloring in Figure 5.6. Notice that, although there were no removed nodes, the intermediate circuit in Figure 5.7(a) has three inverters, whereas the intermediate circuit in Figure 5.7(b) has two inverters.

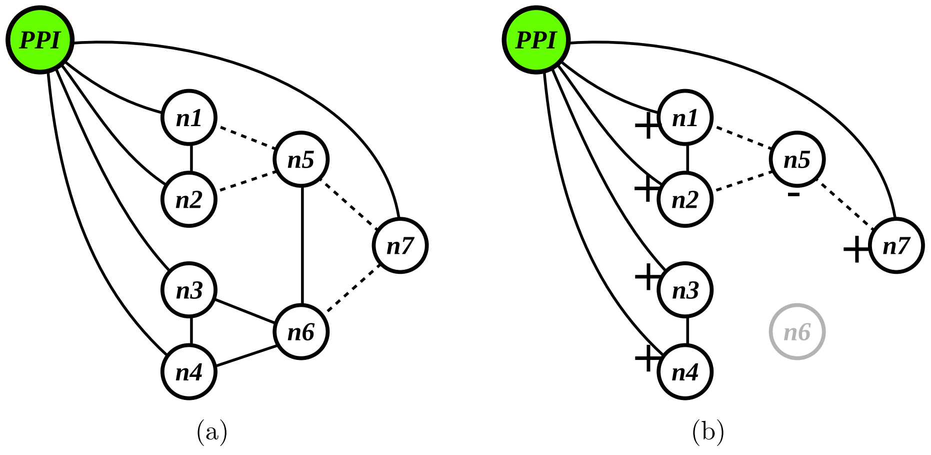
However, it is possible to obtain another circuit, also logically equivalent to those

Figure 5.7: The two possible intermediate circuits, both derived from the two possibilities of graph coloring in Figure 5.6.



Source: Author.

Figure 5.8: The polarity graph with a PPI node (a) and the obtained colored graph (b).



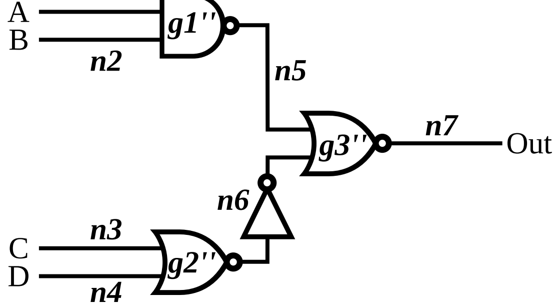
Source: Author.

depicted in Figures 5.7(a) and 5.7(b), and which has only one inverter. As once more contribution of this thesis, we propose an algorithm to force colors on strategic nodes in the way to get better results. The approach to force colors of nodes is based on a positive polarity inducing (PPI) node. All nodes that receive a positive edge from PPI node must have a positive color. In the same way, all nodes that receives a negative edge from PPI node must have a negative color.

Figure 5.8 presents an example of the proposed approach. Notice that, at this time, the colored polarity graph has one removed node. Nevertheless, the derived circuit has only one inverter, as Figure 5.9 shows.

Algorithm 5.8 presents a pseudocode for propagating the colors using the super node. The method has two inputs: the polarity graph to be colored and the super node. In this approach, the nodes that are neighbors of the super node have their color forced according to the edges. The method return is the polarity graph after force the colors.

Figure 5.9: The intermediate circuit derived from the colored graph in Figure 5.8(b).



Source: Author.

Algorithm 5.8: Polarity graph coloring using a PPI node.

```
1 polGraph derivePpiNodeColors(polGraph, PpiNode){
2     foreach ($neighborNode \in ppiNode.neighbors()$)do
3         if (“neighborNode is connect to ppiNode throug positive edge”)then
4             neighborNode.forcePositiveColor();
5         else
6             neighborNode.forceNegativeColor();
7     return polGraph;
```

It is important to remark that all the three circuits presented in Figures 5.7(a), 5.7(b) and 5.9 are logically equivalent. The AIG in Figure 5.6(a) depicts the Boolean function $Out = AB\bar{C}\bar{D}$. The logic circuit in Figure 5.7(a) illustrates the Boolean function $Out = \overline{(\bar{A}+\bar{B})}\,\overline{(C+D)}$. The logic circuit in Figure 5.7(b) represents the Boolean function $Out = \overline{(\overline{AB}+\overline{\bar{C}\bar{D}})}$. Figure 5.9 presents a logic circuit which depicts the Boolean function $Out = \overline{\overline{AB}+(C+D)}$. Table 5.1 presents a truth-table which demonstrates the equivalence among the AIG and the logic circuits in Figures 5.7(a), 5.7(b) and 5.9.

Table 5.1: Truth table of the four Boolean functions represented by AIG and the logic circuits in Figures 5.6(a), 5.7(a), 5.7(b) and 5.9.

A	**B**	**C**	**D**	$AB\bar{C}\bar{D}$	$\overline{(\bar{A}+\bar{B})}\,\overline{(C+D)}$	$\overline{(\overline{AB}+\overline{\bar{C}\bar{D}})}$	$\overline{\overline{AB}+(C+D)}$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	1	1	1	1
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

5.7 Removing Polarity Don't Care Nodes

Both *QuickColor* and *GoodColor* heuristics for coloring the polarity graph have a remarkable feature: the order in which the odd cycles are found and, consequently, the nodes are removed has an important role on the quality of the final result (JAIN; BRYANT, 1993). If the odd cycles are found in a bad order, then the removed nodes will also be bad choices. Even *GoodColor* approach has its resulting colored graph influenced by the order of coloring. This way, the number of nodes to be removed tends to get higher, indicating that the number of inverters in the intermediate circuit tends to get farther than optimal.

In this sense, some specific nodes may have greater significance than other nodes. Choosing these specific nodes to be removed before starting the search for odd cycles may increase the quality of the final result. In the work presented in this thesis, one of these possibilities was verified.

Some nodes may be available in both polarities. This can be due to different reasons. For instance, a given primary input signal can be available in both polarities because the input has an external inverter (MACHADO et al., 2012). Similarly, high fanout nodes will require inverter tree insertion to limit fanout, so the signal will be available in both polarities anyway, after inverters are inserted. These nodes can be considered as being available in both polarities and it makes no sense to color these nodes with a single color to minimize inverters. This way, polarity don't care nodes can be removed from the polarity graph. This approach further improves the results because removing polarity don't care nodes has the potential to remove some odd cycles before to start the graph coloring procedure, making the graph coloring instance easier while placing inverters in nodes where they are already available or where they will be required to reduce fanout.

Algorithm 5.9 presents a pseudocode for removing specific nodes. The method has two inputs: the polarity graph and the designer proposed threshold. For each polarity node in the polarity graph, the algorithm verifies the fanout of the related AIG node. In case of a fanout greater than threshold, the polarity node is marked as removed. After iterating over all polarity nodes, the specific nodes according to adopted criterion will be removed.

Algorithm 5.9: Removing Polarity Don't Care Nodes.

```
void removePolDontCareNodes(polGraph, threshold){
    foreach (polNode ∈ polGraph)do
        aigNode = polNode.getRelatedAigNode();
        if (aigNode.getFanout() > threshold)then
            polNode.markNodeAsRemoved();
```

5.8 Improving the Results with Trade-off Optimizations

There are no guarantees that neither forcing colors using PPI nodes (Section 5.6) nor removing polarity don't care nodes (Section 5.7) will lead to better results. Actually, forcing wrong colors using PPI nodes or removing wrong nodes could lead to worse results than if these approaches were not applied. Additionally, it is not possible to predict which is the best order for coloring the polarity graph.

This Section describes a conservative technique for using the approaches of forcing colors and removing polarity don't care nodes, but accept their results only if they are better then not using them, as well as to coloring the polarity graph using two different orders: (1) using a breadth-first search (BFS); and (2) using a depth-first search (DFS). In the way for scientific analysis, this trade-off approach also considers coloring the graph using two known heuristics: *QuickColor* and *GoodColor*.

The developed technique is based on brute-force. All possible parameters are computed, independently and combined. After that, a trade-off analysis verifies the best solution. The proposed parameters are five, as following: (1) choose between *QuickColor* and *GoodColor*; (2) use a BFS or a DFS; (3) use force colors on inputs; (4) force colors on outputs; and (5) remove specific nodes. Thus, to verify these five binary parameters, 32 iterations (2^5) are necessary.

Algorithm 5.10 presents a pseudocode for trade-off optimization. The method has only one input: the polarity graph. The five parameters are combined in all cases. For each combination, an intermediate circuit is derived. After deriving all possible circuits, the circuit with best transistor count is the method return.

Algorithm 5.10: Deriving a circuit with trade-off analysis.

```
circuit deriveCircuitWithTradeOff(polGraph){
    quickColor = true;
    bfs = true;
    ppiOnIns = true;
    ppiOnOuts = true;
    PolDCs = true;
    for ($heurist = 0; heurist < 2; heurist++, quickColor = !quickColor$) do
        for ($order = 0; order < 2; order++, bfs = !bfs$) do
            for ($ins = 0; ins < 2; ins++, ppiOnIns = !ppiOnIns$) do
                for ($outs = 0; outs < 2; outs++, ppiOnOuts = !ppiOnOuts$) do
                    for ($dc = 0; dc < 2; dc++, polDCs = !polDCs$) do
                        listiOfCircuits.add(deriveCircuit(polGraph, quickColor,
                        bfs, ppiOnIns, ppiOnOuts, polDCs));
    return bestCircuit(listOfCircuits);
```

5.9 High-level Algorithm

After describing in details all steps and substeps of the proposed flow (Sections 5.1 to 5.8), the overall algorithm for obtaining a circuit with reduced transistor count using simples gates must be detailed. Algorithm 5.11 presents a pseudocode for the proposed flow. The method has only one input: the initial circuit. Once the minimized AIG is obtained (line 2) and the polarity graph is derived (line 3), the steps of graph coloring and intermediate circuit generation are performed using the trade-off analysis (line 4). After that, the final circuit is obtained by applying the fanout limiting algorithm (line 5). The final circuit is the method's return.

Algorithm 5.11: Obtaining reduced transistor count circuits using simple cells.

```
1 circuit reduceTransistorCount(initialCircuit){
      /* Step 1.1 (minimize node count in the AIG)                    */
2     aig = deriveMinimizedNodeCountAig(initialCircuit);
      /* Step 1.2 (derive polarity graph)                             */
3     polGraph = derivePolGraphFromAig(aig);
      /* Step 1.3 (graph coloring) and 1.4 (derive intermediate
         circuit) using the trade-off analysis                        */
4     intermediateCircuit = deriveCircuitWithTradeOff(polGraph);
      /* Step 2 (limit fanout)                                        */
5     return fanoutLimiting(intermediateCircuit);
```

6 RESULTS

This chapter provides the main results obtained by applying the proposed approach. The method was applied using simple cells (i.e. NAND2, NOR2 and inverters) over a set of benchmark circuits and a transistor count was performed. After that, the fanout of both cells and inverters in these optimized circuits was limited up to 4 and another transistor count was performed. Section 6.1 reviews the published results using simple cells, and also considering approaches using both CMOS and PTL complex cells. Section 6.2 presents the obtained results compared against simple-cell implementations obtained from ABC tool (Berkeley Logic Synthesis and Verification Group, 2013). In Section 6.3, we present an analysis of how each parameter (presented in Section 5.8) influences the final results. Section 6.4 and Section 6.5 present a fanout and runtime analysis, respectively, and compare them to ABC.

6.1 Review of Published Results in terms of Transistor Count

Table 6.1 presents the simple cells reference implementations with minimum transistor count from prior works (REIS et al., 1995). These numbers are compared to the novel reference transistor counts introduced by applying the method proposed herein. The weighted average values were obtained using the number of transistors as weights.

Table 6.1: Transistor count benchmark circuits in old reference using simple cells compared to the reference transistor count obtained by applying the method proposed herein.

		OUR			
CIRCUIT	OLD REF	FANOUT UNLIMITED		FANOUT LIMITED	
	XTORS	XTORS	%	XTORS	%
C1355	2244	1802	-19.70%	1886	-15.95%
C1908	3146	1628	-48.25%	1690	-46.28%
C2670	4976	2414	-51.49%	2540	-48.95%
C3540	7154	4038	-43.56%	4330	-39.47%
C432	796	596	-25.13%	610	-23.37%
C499	1556	1788	14.91%	1872	20.31%
C5315	10656	5906	-44.58%	6354	-40.37%
C6288	10112	8454	-16.40%	8710	-13.86%
C7552	14376	6406	-55.44%	6588	-54.17%
AVG.		-	-32.18%	-	-29.13%
WEIGHTED AVG.		-	-39.96%	-	-32.21%

Notice that, for most benchmark circuits, the number of transistors is reduced. For the best case, circuit *C7552* was reduced in 55.44%, while the average decrease was 32.18% and the weighted average decrease was 39.96%. The only exception, circuit *C499*, has specific reasons for the obtained values: its initial circuit is mainly implemented using XOR cells, which uses 10 transistors with common Static CMOS implementations for the XOR gate, against 14 transistors for the best case using NAND, NOR and inverters. However, it is important to remark that circuit *C1355* is functionally equivalent to *C499*, but it is implemented without using XOR cells. In this case, the transistor count has been decreased down to 19.70%. After all, notice also that, despite the percent reduction of circuit *C499* be worse than the circuit *C1355*, the resulting transistor count of circuit *C499* is lower than the circuit *C1355*.

Table 6.2 and Table 6.3 present reviews of published results using static complex CMOS cells and PTL topologies, respectively. The aim of presenting the values in these tables is not to provide a transistor count reduction compared to prior complex-cells-based results. The main goal is to review the gains provided by complex cells when compared to a fair minimum transistor count reference implementation introduced herein. The main contribution of our results is to demonstrate that previous gains were overestimated.

In this sense, Table 6.2 reviews published results using static complex CMOS cells. For both compared results, SP(4,4) (REIS et al., 1995) and Resynthesis (GAVRILOV et al., 1997), columns "Old Ref" present the percentage gain when the prior approach is compared to the old reference transistor count using simple cells; and columns "New Ref" present the percentage gain when the prior approach is compared to the reference transistor count obtained by applying the method proposed herein. Notice that SP(4,4) achieves decreases that range between 7.97% and 45.33% when compared to the old reference. However, when SP(4,4) results are compared against our approach, the maximum decrease is 19.91%, but there are increases between 18.14% and 32.31% (gains become losses). In the same way, Resynthesis report decreases of 15.15% and 46.28%, which become increases of 5.66% and 3.81% when compared against the results presented herein (gains become losses).

Table 6.3 reviews results from the literature with PTL topologies. As before, for both compared results, PTLS (SHELAR; SAPATNEKAR, 2005) and OTR (YANBIN; SAPATNEKAR; BAMJI, 2001), columns "Old Ref" present the percentage gain when the prior approach is compared to the old reference transistor count using simple cells, and columns "New Ref" present the percentage gain when the prior approach is compared

Table 6.2: Review of published results using static complex CMOS cells, considering the reference start-point circuits proposed herein.

CIRCUIT	SP(4.4)		Resynthesis	
	Old Ref	New Ref	Old Ref	New Ref
C1355	-30.48%	-13.43%	-15.15%	5.66%
C1908	-31.53%	32.31%	-46.28%	3.81%
C2670	-42.68%	18.14%	N/A	N/A
C3540	-43.89%	-0.59%	N/A	N/A
C432	-14.82%	13.76%	N/A	N/A
C499	-7.97%	-19.91%	N/A	N/A
C5315	-45.33%	-1.35%	N/A	N/A
C6288	-19.94%	-4.23%	N/A	N/A
C7552	-42.01%	30.13%	N/A	N/A

to the reference transistor count obtained by applying the method proposed herein. Notice that, when compared to the old reference, PTLS achieves decreases that range between 22.85% and 63.60% and there is one increase of 39.45%. However, when PTLS results are compared against our approach, the maximum decrease is 44.18%, but there are increases of 86.24% (gains become losses). In the same way, OTR report decreases that range between 13.11% and 47.99%. Nonetheless, when OTR results are compared against our approach, the maximum decrease is 27.64%, but there are increases of 29.53% (gains become losses).

Table 6.3: Review of published results using PTL cells, considering the reference start-point circuits proposed herein

CIRCUIT	PTLS		OTR	
	Old Ref	New Ref	Old Ref	New Ref
C1355	-53.79%	-42.45%	-41.89%	-27.64%
C1908	-63.60%	-29.67%	-40.94%	14.13%
C2670	-42.20%	19.14%	-42.89%	17.73%
C3540	-33.51%	17.81%	-43.92%	-0.64%
C432	39.45%	86.24%	-16.58%	11.41%
C499	-35.86%	-44.18%	-13.11%	-24.38%
C5315	-22.85%	39.20%	-47.99%	-6.16%
C6288	-22.92%	-7.81%	-20.97%	-5.46%
C7552	-62.81%	-16.53%	-42.28%	29.53%
C880	N/A	5.24%	N/A	N/A

6.2 Comparing the Proposed Approach with ABC

In the way for validating the method proposed herein, the same set of benchmark circuits was applied using simple cells on both the proposed approach and ABC tool. The methodology for this comparison diverges after Step 1.1, i.e., after the AIG with minimal node count has been obtained. At this moment, the same AIG is used as input for both approaches. Algorithm 6.1 presents an example of running ABC for obtaining the necessary data and generate the transistor count using simple cells. After reading the input AIG (line 3), the library containing only simple cells (described in genlib format) is read (line 4). Then, the AIG is mapped using the library (line 5) and the final circuit is generated (line 6). The output messages needed for reports and analysis are printed from line 7 to line 9.

Algorithm 6.1: Example of obtaining transistor count with ABC using simple cells.

```
1  user@computer-description ~ $ ./abc > abc-report.log
2  UC Berkeley, ABC 1.01 (compiled Aug 30 2013 09:30:25)
3  abc 01> read_aig example.aig
4  abc 02> read_library minimal.genlib
5  abc 03> map -v
6  abc 04> write_verilog output-circuit.v
7  abc 05> print_stats
8  abc 06> print_gates
9  abc 07> print_fanio
10 abc 08> quit
11 user@computer-description ~ $
```

For each input circuit (and, consequently, AIG), two output circuits are generated using ABC: (1) mapping using *"map"* command, which tries to minimize the circuit delay by limiting the fanout of internal nodes; and (2) mapping using *"map -a"* command, which ignores the fanout information and performs the mapping optimizing only the area of the final circuit. In this sense, for each cell in the given genlib, the area information was changed to the number of transistors. Thus, both ABC and the approach proposed herein could have the same cost function.

Table 6.4 presents the transistor count of benchmark circuits mapped using simple cells by running both ABC and the approach proposed herein. In this table, the columns labeled "UNLIMITED FANOUT" and "LIMITED FANOUT" present, respectively, the transistor count first without considering the fanout of nodes and after limiting the fanout of nodes.

Table 6.4: Transistor count of benchmark circuits mapped using simple cells obtained both from ABC and the approach proposed herein.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	ABC	OUR	%	ABC	OUR	%
C1355	1876	1802	-3.94%	2724	1886	-30.76%
C1908	1704	1628	-4.46%	2098	1690	-19.45%
C2670	2478	2414	-2.58%	2670	2540	-4.87%
C3540	4260	4038	-5.21%	4724	4330	-8.34%
C432	600	596	-0.67%	896	610	-31.92%
C499	1862	1788	-3.97%	2472	1872	-24.27%
C5315	6098	5906	-3.15%	6388	6354	-0.53%
C6288	8484	8454	-0.35%	12346	8710	-29.45%
C7552	6556	6406	-2.29%	6704	6588	-1.73%
C880	1440	1394	-3.19%	1468	1466	-0.14%
fullAdder	36	32	-11.11%	42	32	-23.81%
i10	8374	8116	-3.08%	8968	8636	-3.70%
AVG.			-3.67%			-14.91%
WEIGHTED AVG.			-2.73%			-13.18%

Notice that the method proposed in this work obtained a lower transistor count for all tested circuits, both before and after limiting fanout. In the first case, before limiting fanout, the percent difference range between -0.35% and -11.11%, with an average decrease of 3.67% and a weighted average decrease of 2.73%. In the second case, after limiting fanout, the percent difference range between -0.14% and -31.92%, with an average decrease of 14.91% and a weighted average decrease of 13.18%.

Table 6.5 presents the percent increase of transistor count w.r.t. fanout limitation, both from ABC and the approach proposed herein. Notice that the increase of transistor count in ABC range between 1.94% and 49.33%, against increases between 0.00% and 7.59%.

This higher increase of transistor count from ABC approach is mainly because ABC limits fanout by duplicating nodes. This is worse than the approach proposed herein in two main aspects: (1) each node duplication derives another gate in the final circuit, which means an increase of four transistors for positive consumers and six transistors for negative consumers; and (2) primary input nodes cannot have their fanout limited in ABC approach because there is no way to duplicate primary input nodes.

Table 6.5: Transistor count increase due to fanout limitation.

CIRCUIT	#XTORS INCREASE (%)	
	OUR	ABC
C1355	4.66%	45.20%
C1908	3.81%	23.12%
C2670	5.22%	7.75%
C3540	7.23%	10.89%
C432	2.35%	49.33%
C499	4.70%	32.76%
C5315	7.59%	4.76%
C6288	3.03%	45.52%
C7552	2.84%	2.26%
C880	5.16%	1.94%
fullAdder	0.00%	16.67%
i10	6.41%	7.09%

6.3 Analyzing the Influence of Each Parameter on the Proposed Approach

In this section, all parameters proposed in the approach described in this work are analyzed in terms of their influence in the final circuit. Table 6.6 shows the parameters used on best obtained results. The following subsections provide an study of how these parameters have influence on final results.

Table 6.6: Parameters used on best results.

CIRCUIT	GOOD/ QUICK	BFS/DFS	FORCE INPUTS	FORCE OUTPUTS	REMOVE NODES
C1355	Good	BFS	Yes	Yes	Yes
C1908	Good	DFS	Yes	Yes	Yes
C2670	Good	DFS	Yes	No	Yes
C3540	Quick	DFS	Yes	No	No
C432	Good	DFS	Yes	Yes	No
C499	Good	BFS	Yes	Yes	Yes
C5315	Good	DFS	Yes	No	No
C6288	Quick	DFS	No	No	Yes
C7552	Good	DFS	Yes	No	Yes
C880	Quick	DFS	Yes	No	No
fullAdder	Good	BFS	No	No	No
i10	Good	DFS	Yes	No	Yes

6.3.1 Polarity Graph Coloring in Different Orders

For the sake of analyzing if the order in which the polarity graph is colored really has influence on the final circuit, Table 6.7 presents the transistor count for both BFS and DFS best cases, before and after limiting fanout. In this analysis, the variation of other parameters was allowed, e.g. the comparison between BFS forcing input colors and DFS not forcing input colors is a valid comparison.

Table 6.7: Transistor count of best BFS and DFS iterations allowing variation of all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BEST BFS	BEST DFS	%	BEST BFS	BEST DFS	%
C1355	1802	1804	0.11%	1886	1888	0.11%
C1908	1632	1628	-0.25%	1692	1690	-0.12%
C2670	2442	2414	-1.15%	2568	2540	-1.09%
C3540	4046	4038	-0.20%	4342	4330	-0.28%
C432	620	596	-3.87%	634	610	-3.79%
C499	1788	1792	0.22%	1876	1872	-0.21%
C5315	5936	5906	-0.51%	6370	6354	-0.25%
C6288	8472	8454	-0.21%	8728	8710	-0.21%
C7552	6446	6406	-0.62%	6616	6588	-0.42%
C880	1404	1394	-0.71%	1478	1466	-0.81%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8174	8116	-0.71%	8682	8636	-0.53%

Notice that, for almost all tested cases, the best result using DFS is better than the best result using BFS. This confirms what can be seen in Table 6.6, where the results of DFS are better than the results of BFS. However, this does not means that using DFS is always better than using BFS.

Table 6.8 presents a different analysis of transistor count for both BFS and DFS, before and after limiting fanout. In this table, the results with largest percent difference keeping static all other parameters were used, i.e., the comparison between BFS forcing input colors and DFS not forcing input colors is not a valid comparison in this table. Notice that, now, it is possible to see that using DFS not always leads to better results. Examples are circuit *C2670*, where the DFS result has 4.20% less transistors than the BFS result, and circuit *C432*, where the DFS leads to a result with 5.67% more transistors then the BFS.

Table 6.8: Transistor count of largest percent difference on BFS and DFS iterations keeping static all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BFS	DFS	%	BFS	DFS	%
C1355	1860	1842	-0.97%	1992	1964	-1.41%
C1908	1672	1650	-1.32%	1734	1702	-1.85%
C2670	2572	2464	-4.20%	2702	2602	-3.70%
C3540	4056	4098	1.04%	4348	4386	0.87%
C432	600	634	5.67%	616	652	5.84%
C499	1874	1850	-1.28%	1964	1938	-1.32%
C5315	5906	5978	1.22%	6364	6430	1.04%
C6288	8684	8784	1.15%	8940	9040	1.12%
C7552	6406	6460	0.84%	6588	6662	1.12%
C880	1400	1426	1.86%	1472	1496	1.63%
fullAdder	34	32	-5.88%	34	32	-5.88%
i10	8166	8284	1.45%	8636	8740	1.20%

6.3.2 QuickColor and GoodColor Heuristics

Table 6.9 presents the transistor count for both *QuickColor* and *GoodColor* best cases, before and after limiting fanout. These numbers allow us to analyze if changing the heuristics used for coloring the polarity graph has influence on the final circuit. In this analysis, the variation of other parameters was allowed, e.g. the comparison between *QuickColor* forcing input colors and *GoodColor* not forcing input colors is a valid comparison.

Table 6.9: Transistor count of best QuickColor and GoodColor iterations allowing variation of all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BEST QUICK	BEST GOOD	%	BEST QUICK	BEST GOOD	%
C1355	1848	1802	-2.49%	1934	1886	-2.48%
C1908	1656	1628	-1.69%	1716	1690	-1.52%
C2670	2424	2414	-0.41%	2556	2540	-0.63%
C3540	4038	4050	0.30%	4330	4342	0.28%
C432	608	596	-1.97%	622	610	-1.93%
C499	1840	1788	-2.83%	1930	1872	-3.01%
C5315	5944	5906	-0.64%	6400	6354	-0.72%
C6288	8454	8608	1.82%	8710	8864	1.77%
C7552	6462	6406	-0.87%	6620	6588	-0.48%
C880	1394	1400	0.43%	1466	1468	0.14%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8156	8116	-0.49%	8664	8636	-0.32%

Notice that, as in previous analysis, for almost all tested cases, the best results using *GoodColor* are better than the best results using *QuickColor*. This also confirms what can be seen in Table 6.6, where the results of *GoodColor* are better than the results of *QuickColor*. However, this does not means that using *GoodColor* is always better than using *QuickColor*.

Table 6.10 presents a different analysis of transistor count for both *QuickColor* and *GoodColor*, before and after limiting fanout. In this table, the results with largest percent difference keeping static all other parameters were used, i.e., the comparison between *QuickColor* forcing input colors and *GoodColor* not forcing input colors is not a valid comparison in this table. Notice that, now, it is possible to see that using *GoodColor* not always leads to better results. Examples are circuit *C1908*, where the *GoodColor* result has 3.06% less transistors then the *QuickColor* result, and circuit *C6288*, where the *GoodColor* leads to a result with 4.63% more transistors then the *QuickColor*.

Table 6.10: Transistor count of largest percent difference on QuickColor and GoodColor iterations keeping static all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	QUICK	GOOD	%	QUICK	GOOD	%
C1355	1850	1802	-2.59%	1934	1886	-2.48%
C1908	1702	1650	-3.06%	1754	1702	-2.96%
C2670	2464	2534	2.84%	2602	2670	2.61%
C3540	4050	4112	1.53%	4346	4392	1.06%
C432	610	596	-2.30%	624	610	-2.24%
C499	1842	1788	-2.93%	1930	1872	-3.01%
C5315	6062	5962	-1.65%	6498	6408	-1.39%
C6288	8472	8864	4.63%	8728	9120	4.49%
C7552	6576	6498	-1.19%	6666	6588	-1.17%
C880	1414	1446	2.26%	1486	1518	2.15%
fullAdder	32	34	6.25%	32	34	6.25%
i10	8182	8292	1.34%	8694	8822	1.47%

6.3.3 Forcing Colors on Input Nodes

In order to analyze the influence on the final circuit by forcing the colors on input nodes, Table 6.11 presents the best transistor count for both not forcing and forcing colors on input nodes, before and after limiting fanout. In this analysis, the variation of other parameters was allowed, e.g. the comparison between not forcing input colors using BFS and forcing input colors using DFS is a valid comparison.

Table 6.11: Best transistor count of forcing input colors and not forcing input colors iterations allowing variation of all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BEST NOT FORCING	BEST FORCING	%	BEST NOT FORCING	BEST FORCING	%
C1355	1806	1802	-0.22%	1890	1886	-0.21%
C1908	1634	1628	-0.37%	1690	1692	0.12%
C2670	2432	2414	-0.74%	2576	2540	-1.40%
C3540	4046	4038	-0.20%	4336	4330	-0.14%
C432	600	596	-0.67%	616	610	-0.97%
C499	1794	1788	-0.33%	1872	1872	0.00%
C5315	5944	5906	-0.64%	6370	6354	-0.25%
C6288	8454	8454	0.00%	8710	8710	0.00%
C7552	6498	6406	-1.42%	6694	6588	-1.58%
C880	1398	1394	-0.29%	1468	1466	-0.14%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8174	8116	-0.71%	8682	8636	-0.53%

Notice that, in 96% of tested cases, the best results forcing input colors are better than or equal to the best results not forcing input colors. Again, this confirms what can be seen in Table 6.6, where the best results forcing input colors are higher than the number of best results not forcing input colors. However, this does not means that forcing input colors always leads to better results than not forcing input colors.

Table 6.12 presents a different analysis of transistor count for both forcing and not forcing input colors, before and after limiting fanout. In this table, the results with largest percent difference keeping static all other parameters were used, i.e., the comparison between not forcing input colors using BFS and forcing input colors using DFS is not a valid comparison in this table. Notice that, now, it is possible to see that forcing colors on input nodes not always leads to better results. Examples are circuit *C432*, where the result forcing input colors has 3.23% less transistors then the result not forcing input colors, and circuit *C2670*, where forcing colors on input nodes leads to a result with 5.97% more transistors than not forcing input colors.

Table 6.12: Transistor count of largest percent difference on forcing input colors and not forcing input colors iterations keeping static all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	NOT FORCING	FORCING	%	NOT FORCING	FORCING	%
C1355	1842	1870	1.52%	1928	1958	1.56%
C1908	1650	1682	1.94%	1702	1738	2.12%
C2670	2414	2558	5.97%	2540	2686	5.75%
C3540	4084	4112	0.69%	4370	4342	-0.64%
C432	620	600	-3.23%	636	616	-3.14%
C499	1850	1880	1.62%	1938	1968	1.55%
C5315	5944	6078	2.25%	6402	6514	1.75%
C6288	8608	8824	2.51%	8864	9080	2.44%
C7552	6462	6620	2.45%	6620	6778	2.39%
C880	1400	1446	3.29%	1472	1516	2.99%
fullAdder	34	32	-5.88%	34	32	-5.88%
i10	8116	8220	1.28%	8636	8742	1.23%

6.3.4 Forcing Colors on Output Nodes

In the interest of analyzing if forcing colors on output nodes has influence on the final circuit, Table 6.13 presents the transistor count for both best cases, before and after limiting fanout. In this analysis, the variation of other parameters was allowed, e.g. the comparison between not forcing output colors using BFS and forcing output colors using DFS is a valid comparison.

Table 6.13: Best transistor count of forcing and not forcing output colors iterations allowing variation of all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BEST NOT FORCING	BEST FORCING	%	BEST NOT FORCING	BEST FORCING	%
C1355	1832	1802	-1.64%	1916	1886	-1.57%
C1908	1650	1628	-1.33%	1702	1690	-0.71%
C2670	2414	2424	0.41%	2540	2554	0.55%
C3540	4038	4038	0.00%	4340	4330	-0.23%
C432	608	596	-1.97%	620	610	-1.61%
C499	1850	1788	-3.35%	1936	1872	-3.31%
C5315	5906	5924	0.30%	6354	6358	0.06%
C6288	8454	8476	0.26%	8710	8732	0.25%
C7552	6406	6412	0.09%	6588	6616	0.43%
C880	1394	1396	0.14%	1466	1468	0.14%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8116	8156	0.49%	8636	8690	0.63%

Notice that, in 71% of tested cases, the percent variation of forcing and not forcing colors on output nodes are less 1% in absolute value. However, in the remaining 29% where the percent difference are more then 1%, forcing colors on output nodes has lead to better results.

Table 6.14 presents a different analysis of transistor count for both forcing and not forcing output colors, before and after limiting fanout. In this table, the results with largest percent difference keeping static all other parameters were used, i.e., the comparison between not forcing output colors using BFS and forcing output colors using DFS is not a valid comparison in this table. Notice that forcing colors on output nodes may lead to worse results. In the tested benchmark circuits, forcing colors on output nodes and keeping static the other parameters provides better results only in a third of cases.

Table 6.14: Transistor count of largest percent difference on forcing and not forcing output colors iterations keeping static all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	NOT FORCING	FORCING	%	NOT FORCING	FORCING	%
C1355	1848	1894	2.49%	1934	1992	3.00%
C1908	1634	1680	2.82%	1690	1730	2.37%
C2670	2438	2572	5.50%	2576	2702	4.89%
C3540	4078	4054	-0.59%	4364	4342	-0.50%
C432	596	608	2.01%	616	628	1.95%
C499	1840	1910	3.80%	1930	2012	4.25%
C5315	5994	6078	1.40%	6424	6514	1.40%
C6288	8608	8864	2.97%	8864	9120	2.89%
C7552	6582	6504	-1.19%	6764	6700	-0.95%
C880	1404	1442	2.71%	1468	1510	2.86%
fullAdder	34	32	-5.88%	34	32	-5.88%
i10	8210	8166	-0.54%	8720	8664	-0.64%

6.3.5 Removing Polarity Don't Care Nodes

In order to analyze the influence on the final circuit of removing polarity don't care nodes, Table 6.15 presents the best transistor count for both not removing and removing these nodes, before and after limiting fanout. The threshold used was 4. In this analysis,

Table 6.15: Best transistor count of not removing and removing specific nodes iterations allowing variation of all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	BEST NOT REMOVING	BEST REMOVING	%	BEST NOT REMOVING	BEST REMOVING	%
C1355	1816	1802	-0.77%	1918	1886	-1.67%
C1908	1648	1628	-1.21%	1702	1690	-0.71%
C2670	2424	2414	-0.41%	2558	2540	-0.70%
C3540	4038	4038	0.00%	4330	4330	0.00%
C432	596	600	0.67%	610	616	0.98%
C499	1806	1788	-1.00%	1892	1872	-1.06%
C5315	5906	5910	0.07%	6364	6354	-0.16%
C6288	8454	8454	0.00%	8710	8710	0.00%
C7552	6414	6406	-0.12%	6610	6588	-0.33%
C880	1394	1394	0.00%	1466	1466	0.00%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8166	8116	-0.61%	8682	8636	-0.53%

the variation of other parameters was allowed, e.g. the comparison between not removing specific nodes using BFS and removing these nodes using DFS is a valid comparison.

Notice that, in 88% of tested cases, the best results removing polarity don't care nodes are better then or equal to the best results not removing these nodes. Once again, this confirms what can be seen in Table 6.6, where the number of best results removing these specific nodes is higher than the the number of best results not removing them. However, this does not means that removing these nodes always leads to better results than not removing them.

Table 6.16 presents a different analysis of transistor count for both removing and not removing polarity don't care nodes, before and after limiting fanout. In this table, the results with largest percent difference keeping static all other parameters were used, i.e., the comparison between not removing polarity don't care nodes using BFS and removing these nodes using DFS is not a valid comparison in this table. Notice that, now, it is possible to see that removing polarity don't care nodes not always leads to better results. Examples are circuit *2670*, where the result removing these nodes has 4.40% less transistors than the result not removing them, and circuit *6288*, where removing these specific nodes leads to a result with 2.64% more transistors than removing these nodes.

Table 6.16: Transistor count of largest percent difference on removing and not removing polarity don't care nodes keeping static all other parameters.

CIRCUIT	UNLIMITED FANOUT			LIMITED FANOUT		
	NOT REMOVING	REMOVING	%	NOT REMOVING	REMOVING	%
C1355	1836	1870	1.85%	1920	1958	1.98%
C1908	1634	1676	2.57%	1690	1738	2.84%
C2670	2544	2432	-4.40%	2680	2576	-3.88%
C3540	4052	4098	1.14%	4342	4386	1.01%
C432	620	596	-3.87%	636	610	-4.09%
C499	1794	1824	1.67%	1936	1972	1.86%
C5315	5962	6040	1.31%	6424	6480	0.87%
C6288	8636	8864	2.64%	8892	9120	2.56%
C7552	6524	6504	-0.31%	6662	6634	-0.42%
C880	1404	1422	1.28%	1468	1492	1.63%
fullAdder	32	32	0.00%	32	32	0.00%
i10	8216	8284	0.83%	8738	8822	0.96%

6.4 Analysis of Fanout

This section presents an analysis of fanout in the final circuits, both running ABC and the approach proposed herein. The maximum fanout of both inverters and source cells that was considered in this analysis is 4. Table 6.17 shows the worst and average fanout, before and after limiting fanout, in the final circuits obtained running ABC. Table 6.18 shows the same information when running the method proposed in this work. It is important to remark that ABC values do not consider the fanout of primary input nodes.

Table 6.17: Fanout analysis running ABC.

	WORST FANOUT			**AVERAGE FANOUT**		
CIRCUIT	BEFORE	AFTER	%	BEFORE	AFTER	%
C1355	7	6	-14.29%	1.68	1.51	-10.12%
C1908	11	6	-45.45%	1.67	1.54	-7.78%
C2670	16	16	0.00%	1.44	1.43	-0.69%
C3540	20	22	10.00%	1.66	1.63	-1.81%
C432	9	11	22.22%	1.46	1.35	-7.53%
C499	7	6	-14.29%	1.69	1.5	-11.24%
C5315	21	19	-9.52%	1.52	1.5	-1.32%
C6288	17	17	0.00%	1.87	1.79	-4.28%
C7552	116	113	-2.59%	1.58	1.57	-0.63%
C880	8	8	0.00%	1.48	1.48	0.00%
fullAdder	2	2	0.00%	1.29	1.27	-1.55%
i10	28	24	-14.29%	1.65	1.63	-1.21%
AVG.			-5.68%			-4.01%

Table 6.18: Fanout analysis running the method proposed herein.

	WORST FANOUT			**AVERAGE FANOUT**		
CIRCUIT	BEFORE	AFTER	%	BEFORE	AFTER	%
C1355	8	4	-50.00%	1.66	1.47	-11.45%
C1908	15	4	-73.33%	1.68	1.54	-8.33%
C2670	22	4	-81.82%	1.36	1.28	-5.88%
C3540	41	4	-90.24%	1.8	1.45	-19.44%
C432	9	4	-55.56%	1.48	1.43	-3.38%
C499	8	4	-50.00%	1.66	1.48	-10.84%
C5315	41	4	-90.24%	1.68	1.42	-15.48%
C6288	17	4	-76.47%	1.79	1.62	-9.50%
C7552	170	4	-97.65%	1.63	1.53	-6.13%
C880	9	4	-55.56%	1.65	1.47	-10.91%
fullAdder	2	2	0.00%	1.5	1.5	0.00%
i10	26	4	-84.62%	1.72	1.48	-13.95%
AVG.			-67.12%			-9.61%

Notice that, in the resulting circuits obtained with ABC after limiting fanout, the worst fanout decreases in a range between 2.59% and 45.45%, but there are increases of 22.22%. Even so, the worst fanout decreases in an average of 5.68%. When running the approach proposed herein, after limiting fanout, the worst fanout decreases in a range between 50.00% and 97.65%, and the worst fanout decreases in an average of 67.12%. Another remarkable feature is that the average fanout decreases in a range between 0.69% and 11.24% after ABC limiting fanout. When running the approach proposed herein, the average fanout of the circuits decreases in a range between 3.38% and 19.44%.

6.5 Analysis of Runtime

In this section, the runtime for obtaining reduced transistor count in benchmark circuits using simple cells when running the proposed approach is analyzed. The runtime information is compared to the runtime for obtaining the circuits using ABC.

Due to the proposed parameters and their variations, one complete iteration of the proposed approach runs 32 sub-iterations, which derives 32 different circuits. Table 6.19 presents the slower, faster and average runtime of these sub-iterations for each tested circuit, as well as the runtime of the sub-iteration that derived the best transistor count.

Table 6.19: Runtime analysis of the proposed approach.

CIRCUIT	SLOWER (ms)	FASTER (ms)	AVG. (ms)	BEST #XTORS (ms)
C1355	282.05	20.26	69.17	40.59
C1908	118.45	8.20	23.21	18.90
C2670	89.91	11.99	31.02	17.77
C3540	58.57	19.70	30.77	24.03
C432	7.75	3.85	5.40	5.43
C499	93.99	6.86	14.43	13.94
C5315	88.26	32.57	51.07	60.38
C6288	282.98	62.68	121.23	78.51
C7552	90.85	51.40	64.22	65.49

Table 6.20 shows the runtime for each circuit when running ABC and the average runtime when running the proposed approach. As the sub-iterations derives different circuits, this table has a third column that shows how many circuits (in % values) derived from the 32 sub-iterations have lower transistor count comparing against ABC.

Table 6.20: Runtime analysis of average runtime comparing against ABC.

CIRCUIT	ABC RUNTIME	OUR RUNTIME	ABC % LOSSES
C1355	200 *ms*	69.17 *ms*	75.00%
C1908	200 *ms*	23.21 *ms*	100.00%
C2670	200 *ms*	31.02 *ms*	62.50%
C3540	200 *ms*	30.77 *ms*	100.00%
C432	<100 *ms*	5.40 *ms*	3.13%
C499	200 *ms*	14.43 *ms*	56.25%
C5315	200 *ms*	51.07 *ms*	100.00%
C6288	700 *ms*	121.23 *ms*	50.00%
C7552	400 *ms*	64.22 *ms*	65.63%
C880	100 *ms*	13.34 *ms*	87.50%
fullAdder	<100 *ms*	6.74 *ms*	100.00%
i10	500 *ms*	95.39 *ms*	100.00%

Table 6.21 shows again ABC runtime and also the runtime of the proposed approach after the 32 sub-iterations.

Table 6.21: Runtime analysis of the 32 sub-iterations comparing against ABC.

CIRCUIT	ABC RUNTIME	OUR RUNTIME
C1355	200 *ms*	974.86 *ms*
C1908	200 *ms*	471.67 *ms*
C2670	200 *ms*	723.24 *ms*
C3540	200 *ms*	897.88 *ms*
C432	<100 *ms*	129.45 *ms*
C499	200 *ms*	248.58 *ms*
C5315	200 *ms*	1429.55 *ms*
C6288	700 *ms*	3222.91 *ms*
C7552	400 *ms*	1960.83 *ms*
C880	100 *ms*	184.70 *ms*
fullAdder	<100 *ms*	9.84 *ms*
i10	500 *ms*	3047.30 *ms*

7 CONCLUSIONS AND FUTURE WORKS

This work is centered on methods to obtain minimum transistor count implementations of digital VLSI circuits based on simple cells. Thus, we provide three main contributions in the work presented herein: (1) the introduction of a set of efficient algorithms for obtaining reduced transistor count; (2) a new reference for transistor count in benchmark circuits using simple cells; and (3) an analysis of previous works which advocate for complex cells when comparing them against the new reference proposed herein.

The set of algorithms comprising the first contribution has four main aspects: (*i*) the extension of the inverter minimization procedure proposed by Jain and Bryant (1993) to work over AIGs instead of logic networks; (*ii*) the improvement of the cited approach introducing PPI nodes, which makes it possible to force chosen polarities on selected nodes; (*iii*) the introduction of an algorithm to remove polarity don't care nodes; and (*iv*) a novel algorithm for limiting fanout by inserting inverter trees. The presented algorithms, specifically the ones related with the aspects i, ii and iii, obtain reduced transistor count implementations from AIGs with minimized number of nodes. The proposed approach was validated by generating benchmark circuits using a simple library containing only NAND2, NOR2 and inverter as available cells. The obtained results have shown the usefulness of the method. The circuits generated by our algorithms have, in average, 32.07% less transistor than the previous reference on transistor count using simple cells (39.92% less transistor in a weighted average). Intrinsically, these numbers comprise the second contribution of this thesis: a new reference for transistor count in benchmark circuits using simple cells, which can be used as a fair start-point to compare circuit implementations using simple gates against circuit implementations using complex cells. One could argue that, due to the high logic sharing obtained after reducing node count in AIGs, the generated circuits have cells with unfeasible fanout. This argument leads to aspect *iv* of the first contribution: a novel algorithm for limiting fanout by inserting inverter trees. The presented algorithm is able to provide an output circuit in which no cell has a fanout larger than a given threshold limit. The proposed algorithm for limiting fanout was applied over the generated circuits. The obtained results also have shown the usefulness of the method. The obtained circuits after limiting fanout of their cells up to 4 still have lower transistor count when compared against the previous reference implementations using gates (-29.07% in average and -37.19% in weighted average). Comparing the presented results against ABC tool, our approach was able to fix all “fanout violations” with an average transistor count increase of 4.30%, whereas ABC did not fix all “fanout violations” and has achieved an average transistor count increase of 20.56%.

Additionally, when comparing the presented results in terms of transistor count against works advocating for complex cells, our results have demonstrated that previous approaches are sometimes far from the minimum transistor count that can be obtained with the efficient use of a reduced cell library composed by only a few number of simple cells. The simple-cells-based circuits obtained after applying the algorithms proposed herein have presented a lower transistor count in many cases when compared to previously published results using complex (static CMOS and PTL) cells, which is surprising and counter-intuitive. These results strongly suggest that the gains in using complex cells were overestimated by previous publications and this analysis comprises the third contribution of this work.

There is still much work to be carried on. The impact of all the proposed algorithms in terms of area, power consumption and delay must be analyzed. It can be argued that the additional inverters inserted in the circuit to perform fanout limitation may lead to area and power overheads with respect to minimum transistor count implementations. Even son, it is not realistic to consider circuits without any fanout restriction. With respect to delay, we believe that the current circuits could have equivalent results when compared against the circuits obtained by state-of-the-art tools. Nevertheless, these expectations need to be evaluated. It is important to highlight that the methods proposed herein are able to produce high quality subject descriptions for further technology mapping without restriction to simple cells. This claim is justified as technology mapping usually start from a description in terms of simple cells, and this step is known to be dependent of the initial description used. This way, providing a better start point for further technology mapping is an important contribution.

Another future work is to use the generated circuits as input for a technology mapping algorithm using complex cells. We agree with the results reported by Reis et al. (1995) on circuit 9sym, where complex cells can reduce transistor count from a given start point. Therefore, we believe that using these circuits as start-points for a new synthesis iteration, now allowing more complex cells, can potentially produce better results for future approaches, as logic synthesis algorithms depend on the initial circuit.

Nonetheless, even if the presented research opened the way for several future works, the current results have shown the computational and practical viability of the methods presented herein. The proposed algorithms have been proved useful for obtaining benchmark circuits with reduced transistor count using simple cells. This usefulness has been demonstrated both with respect to previously published references with simple cells, as well as with respect to current state of the art tools, like ABC.

REFERENCES

ABOUZEID, P. et al. Logic synthesis for automatic layout. In: PROC. OF EURO ASIC. **Proceedings...** [S.l.: s.n.], 1992. p. 146–151.

AKERS, S. B. Binary Decision Diagrams. **IEEE Trans. on Computer**, IEEE, v. 27, n. 6, p. 509–516, 1978.

BERKELAAR, M.; JESS, J. Technology mapping for standard-cell generators. In: PROC. OF INT'L CONF. ON COMPUTER-AIDED DESIGN (ICCAD). **Proceedings...** [S.l.: s.n.], 1988. p. 470–473.

Berkeley Logic Synthesis and Verification Group. **ABC: A System for Sequential Synthesis and Verification**. 2013. Release 20130425. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/>. Accessed in: 2013-01-25.

BIERE, A. **AIGER Format**. 2007. Available from Internet: <fmv.jku.at/aiger/>.

BRAYTON, R. K. et al. A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization. In: PROC. OF INT'L SYMP. ON CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** [S.l.: s.n.], 1982. p. 42–48.

BRAYTON, R. K.; MCMULLEN, C. The decomposition and factorization of boolean expressions. In: PROC. OF INT'L SYMP. ON CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** [S.l.: s.n.], 1982.

BRAYTON, R. K. et al. MIS: A multiple-level logic optimization system. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 6, n. 6, p. 1062–1081, 1987.

BRAYTON, R. K. et al. **Logic Minimization Algorithms for VLSI Synthesis**. [S.l.]: Kluwer Academic Publishers, 1984.

BRYANT, R. E. Graph-based algorithms for Boolean functions manipulation. **IEEE Trans. on Computer**, IEEE, v. 35, n. 8, p. 677–691, 1986.

CHATTERJEE, S. **On Algorithms for Technology Mapping**. [S.l.]: University of California, Berkeley, 2007.

CHATTERJEE, S. et al. Reducing structural bias in technology mapping. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,**, IEEE, v. 25, n. 12, p. 2894–2903, 2006.

CORMEN, T. H. et al. Chpater 5: Graphs. In: _____. INTRODUCTION TO ALGORITHMS. **Proceedings...** [S.l.]: MIT press Cambridge, 2001. v. 2.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: PROC. OF THE CONF. ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.: s.n.], 2004. p. 254–259.

CORTADELLA, J. Timing-driven logic bi-decomposition. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 22, n. 6, p. 675–685, 2003.

DARRINGER, J. a. et al. Logic Synthesis Through Local Transformations. **IBM Journal of Research and Development**, v. 25, n. 4, p. 272–280, 1981.

DETJENS, E. et al. Technology mapping in mis. In: PROC. OF IEEE INT'L CONF. ON COMPUTER AIDED DESIGN (ICCAD). **Proceedings...** [S.l.: s.n.], 1987. p. 116–119.

DOOD, P. de; LEE, B.; ALBERS, D. **Optimization of circuit designs using a continuous spectrum of library cells**. 2006. US Patent 7107551.

GAVRILOV, S. et al. Library-less synthesis for static CMOS combinational logic circuits. In: PROC. OF IEEE INT'L CONF. ON COMPUTER AIDED DESIGN (ICCAD). **Proceedings...** [S.l.: s.n.], 1997. p. 658–662.

GEREZ, S. H. **Algorithms for VLSI Design Automation**. 1. ed. West Sussex, England: John Wiley & Sons Ltd, 1999. 326 p.

GREGORY, D. et al. Socrates: A system for automatically synthesizing and optimizing combinational logic. In: DESIGN AUTOMATION CONF. (DAC). **Proceedings...** [S.l.: s.n.], 1986. p. 79–85.

GUAN, B.; SECHEN, C. Large standard cell libraries and their impact on layout area and circuit performance. In: INT'L CONF. ON COMPUTER DESIGN (ICCD). **Proceedings...** [S.l.: s.n.], 1996. p. 378–383.

HACHTEL, G. D.; SOMENZI, F. **Logic Synthesis and Verification Algorithms**. 1. ed. Norwell, Massachusetts: Kluwer Academic Publishers, 1996. 564 p.

HELLERMAN, L. A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits. **IEEE Trans. on Electronic Computers**, EC-12, n. 3, p. 198–223, 1963.

ITRS. The International Technology Roadmap for Semiconductors: 2012 UPDATE. In: THE NINTH INT'L NANOTECHNOLOGY CONF. ON COMMUNICATIONS AND COOPERATION (INC9). **Proceedings...** Berlin, Germany: [s.n.], 2012. Available from Internet: <http://www.itrs.net/reports.html>.

JAIN, A.; BRYANT, R. Inverter minimization in multi-level logic networks. In: PROC. OF IEEE INT'L CONF. ON COMPUTER AIDED DESIGN (ICCAD). **Proceedings...** [S.l.]: IEEE Comput. Soc. Press, 1993. p. 462–465.

KAGARIS, D.; HANIOTAKIS, T. A Methodology for Transistor-Efficient Supergate Design. **IEEE Trans. on Very Large Scale Integration (VLSI) Systems**, v. 15, n. 4, p. 488–492, 2007.

KAHNG, A. B. et al. **VLSI Physical Design: From Graph Partitioning to Timing Closure**. Dordrecht: Springer Netherlands, 2011. 1–310 p.

KEUTZER, K. Dagon: technology binding and local optimization by dag matching. In: DESIGN AUTOMATION CONF. (DAC). **Proceedings...** [S.l.: s.n.], 1987. p. 341–347.

KUKIMOTO, Y.; BRAYTON, R. K.; SAWKAR, P. Delay-optimal technology mapping by dag covering. In: DESIGN AUTOMATION CONF. (DAC). **Proceedings...** [S.l.: s.n.], 1998. p. 348–351.

LAVAGNO, L.; SCHEFFER, L.; MARTIN, G. **EDA for IC Implementation, Circuit Design, and Process Technology**. [S.l.]: Taylor & Francis, 2010. (Electronic Design Automation for Integrated Circuits Hdbk).

LEE, C. Y. Binary Decision Programs. **Bell System Technical Journal**, American Telephone and Telegraph Company, v. 38, n. 4, p. 985–999, 1959.

LEWIS, J. M.; YANNAKAKIS, M. The node-deletion problem for hereditary properties is NP-complete. **Journal of Computer and System Sciences**, v. 20, n. 2, p. 219–230, abr. 1980.

LIEM, C.; LEFEBVRE, M. A constructive matching algorithm for cell generator based technology mapping. In: PROC. OF INT'L SYMP. ON CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** [S.l.: s.n.], 1992. v. 6, p. 2965–2968.

MACHADO, L. et al. KL-cut based digital circuit remapping. In: NORCHIP. **Proceedings...** [S.l.: s.n.], 2012.

MAILHOT, F.; MICHELI, G. D. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. **IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,**, IEEE, v. 12, n. 5, p. 599–620, 1993.

MARQUES, F. S. et al. Mapeamento Tecnológico no Projeto de Circuitos Integrados Digitais. In: MATTOS, J. C. B.; JR, L. S. R.; PILLA, M. L. (Ed.). Desafios e Avanços em Computação: O Estado da Arte. **Proceedings...** 1. ed. [S.l.]: Editora da Universidade Federal de Pelotas, 2009.

MARQUES, F. S. et al. DAG based library-free technology mapping. In: PROC. OF GREAT LAKES SYMP. ON VLSI (GLSVLSI). **Proceedings...** [S.l.: s.n.], 2007.

MICHELI, G. de. Cell-based Logic Optimizations. In: BORGER, E. (Ed.). Architecture Design and Validation Methods. **Proceedings...** 1. ed. [S.l.]: Springer-Verlag, 2000.

MICHELI, G. de. **Synthesis and optimization of digital circuits**. [S.l.]: Tata McGraw-Hill Education, 2003.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: PROC. OF ACM/IEEE DESIGN AUTOMATION CONF. (DAC). **Proceedings...** [S.l.: s.n.], 2006. p. 532–535.

MOORE, G. Cramming More Components onto Integrated Circuits. **Electronics**, p. 82–85, 1965.

PIGUET, C. et al. Low-power low-voltage library cells and memories. In: INT'L CONF. ON ELECTRONICS, CIRCUITS AND SYSTEMS (ICECS). **Proceedings...** [S.l.: s.n.], 2001. v. 3, p. 1521–1524.

RABAEY, J. M.; CHANDRAKASAN, A. P.; NIKOLIC, B. **Digital integrated circuits**. [S.l.]: Prentice hall Englewood Cliffs, 2002.

REIS, A. et al. Associating CMOS transistors with BDD arcs for technology mapping. **Electronics Letters**, v. 31, n. 14, p. 1118–1120, 1995.

REIS, A. I. Covering strategies for library free technology mapping. In: PROC. OF THE CONF. ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.: s.n.], 1999. p. 180–183.

RICCI, A.; MUNARI, I. D.; CIAMPOLINI, P. An evolutionary approach for standard-cell library reduction. In: PROC. OF THE 17TH ACM GREAT LAKES SYMP. ON VLSI. **Proceedings...** [S.l.: s.n.], 2007. (GLSVLSI '07), p. 305–310.

RUDELL, R. **Logic synthesis for VLSI design**. Thesis (PhD) — University of California, Berkeley, 1989.

SCHNEIDER, F. et al. Exact lower bound for the number of switches in series to implement a combinational logic cell. In: INT'L CONF. ON COMPUTER DESIGN (ICCD). **Proceedings...** [S.l.: s.n.], 2005. p. 357–362.

SENTOVICH, E. M. et al. **SIS: A System for Sequential Circuit Synthesis**. [S.l.], 1992.

SEO, J.-s. et al. On the decreasing significance of large standard cells in technology mapping. In: PROC. OF INT'L CONF. ON COMPUTER-AIDED DESIGN (ICCAD). **Proceedings...** [S.l.: s.n.], 2008. p. 116–121.

SHELAR, R.; SAPATNEKAR, S. BDD decomposition for delay oriented pass transistor logic synthesis. **IEEE Trans. on Very Large Scale Integration (VLSI) Systems**, v. 13, n. 8, p. 957–970, aug. 2005.

STOK, L.; IYER, M. A.; SULLIVAN, A. J. Wavefront technology mapping. In: DESIGN AUTOMATION AND TEST IN EUROPE CONF. AND EXHIBITION (DATE). **Proceedings...** [S.l.: s.n.], 1999. p. 531–536.

WESTE, N. H.; HARRIS, D. M. **CMOS VLSI Design: a Circuits and Systems Perspective**. 4. ed. [S.l.]: Addison-Wesley, 2009.

YANBIN, J.; SAPATNEKAR, S. S.; BAMJI, C. Technology mapping for high-performance static CMOS and pass transistor logic designs. **IEEE Trans. on Very Large Scale Integration (VLSI) Systems**, v. 9, n. 5, p. 577–589, 2001.